Karlsruhe Institute of Technology

Real-Time Shading with Polyhedral Lights using Silhouette Detection

Bachelor Thesis of

Bastian Urbach

At the Department of Informatics
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

March 23, 2021

Reviewer: Prof. Dr.-Ing. Carsten Dachsbacher
Second reviewer: Prof. Dr. Hartmut Prautzsch
Advisor: Christoph Peters

Contents

1 Introduction **3**

2 Related Work **5**

3 Basics **7**
3.1 BRDF 7
3.2 Rendering Equation and Area Lights 9
3.3 Spherical Triangles and Polygons 10
3.4 Polygonal Light Shading with Linearly Transformed Cosines 11
3.5 Monte Carlo Integration 13
3.6 Solid Angle Sampling 14

4 Finding Silhouettes for Polyhedral Light Shading with LTCs **16**
4.1 Contour and Silhouette of a Polyhedron 17
4.2 Convex Polyhedra 18
4.3 Non-Convex Polyhedra 20

5 Precomputed Silhouettes with Binary Space Partitioning **24**
5.1 Convex Polyhedra 24
5.2 Non-Convex Polyhedra 25
5.3 Building the Tree 27
5.4 Shader Implementation Details 28

6 Using Silhouettes for Solid Angle Sampling **30**
6.1 Convex and Star-Shaped Polyhedra 30
6.2 General Polyhedra 31
6.2.1 Triangulation Algorithm 31
6.2.2 Handling Cases Where No Triangulation Exists 32
6.2.3 Precomputation with Binary Space Partitioning 33

7 Results **37**
7.1 Rendered Results 37
7.2 Run Times 41
7.2.1 Shading 41
7.2.2 Precomputation 43
7.3 Size of Precomputed Data 44

8 Conclusion and Future Work **45**

Bibliography **48**

Abstract

In this thesis, I explore using silhouettes of polyhedral light sources for real-time shading. The central problem that I solved for this was to efficiently determine the silhouette for a shading point in a fragment shader. I present a simple solution for convex polyhedra that can be executed entirely in the shader. I also present a more sophisticated method that requires a preprocessing phase but provides better performance and can be applied to non-convex polyhedra as well. This method consists of an algorithm for finding the silhouette of any polyhedron, an algorithm and data structure based on binary space partitioning for storing precomputed silhouettes for every point in the scene, and the algorithm for using that data structure in the shader.

Furthermore, I explain how to use silhouettes for area light shading with linearly transformed cosines and for ray tracing with solid angle sampling. For solid angle sampling, I use a technique for sampling spherical triangles, which means that I have to triangulate the silhouette. I present a simple and efficient way of doing so for convex and star-shaped light sources. For other light sources, I explain how to modify the data structure used to find the silhouette to store triangulated silhouettes instead.

I tested the performance of my techniques for a variety of light sources and found that they give relevant performance improvements over simpler techniques that do not use silhouettes. For the tested light sources, the techniques are well suited for real-time applications on current graphics hardware.

Zusammenfassung

In dieser Arbeit befasse ich mich mit der Verwendung von Silhouetten polyedrischer Lichtquellen für Echtzeit-Shading. Das zentrale Problem, das ich hierfür gelöst habe ist die effiziente Bestimmung der Silhouette für einen Oberflächenpunkt in einem Fragment-Shader. Für konvexe Polyeder beschreibe ich eine einfache Lösung, die vollständig im Shader ausgeführt werden kann. Außerdem präsentiere ich eine weitere Methode, die zwar eine Vorberechnungsphase erfordert, dafür aber schnellere Berechnung im Shader ermöglicht und auch für nicht-konvexe Polyeder verwendet werden kann. Diese Methode besteht aus einem Algorithmus zur Bestimmung der Silhouetten, einem Algorithmus und zugehöriger Datenstruktur basierend auf Binary Space Partitioning zur Bereitstellung vorberechneter Silhouetten für jeden Punkt in einer Szene und dem Algorithmus zur Verwendung dieser vorberechneten Daten im Shader.

Ich erkläre außerdem, wie ich Silhouetten für Beleuchtungsberechnung mit linear transformierten Kosinusverteilungen (linearly transformed cosines, LTCs) und für Raytracing mit Raumwinkelabtastung (solid angle sampling) verwende. Für die Raumwinkelabtastung nutze ich eine Methode für die Abtastung sphärischer Dreiecke und muss daher die Silhouette zunächst triangulieren. Hierfür beschreibe eine einfache und effiziente Methode für konvexe und sternförmige Lichtquellen. Für andere Lichtquellen beschreibe ich eine Abwandlung meiner Methode zur Vorberechnung von Silhouetten, mit der auch die Triangulierungen vorberechnet werden.

Ich habe die Effizienz meiner Verfahren für eine Auswahl von Lichtquellen geprüft und bin zu dem Ergebnis gekommen, dass sie im Vergleich zu einfacheren Verfahren, die keine Silhouetten verwenden relevante Verbesserungen bieten. Für die geprüften Lichtquellen sind die Verfahren gut für Echtzeitanwendungen auf aktueller Grafikhardware geeignet.

1. Introduction

Calculating shading of a surface due to direct illumination is an important problem in computer graphics. In real-time applications, light sources are commonly approximated using point and directional lights for the simplicity of the resulting lighting calculations. Area lights reproduce the appearance of many real-world light sources more accurately but shading them is significantly more expensive and the formulas are difficult to derive for many commonly used shading models. To alleviate this, several cheaper approximations have been proposed and used, including one that replaces the cosine-weighted BRDF that models the reflective properties of a surface by a simpler function from the family of linearly transformed cosines (LTCs) [HDHN16]. Various light shapes, including arbitrary polygons, can be used for real-time shading using this technique.

I explore an efficient approach to extending shading with LTCs to polyhedral light sources by replacing them with their silhouette polygon as seen from the shading point. I describe algorithms for calculating the silhouette of polyhedra as well as methods for moving the bulk of necessary computations to a preprocessing phase, which makes the presented technique suitable for real-time applications. While the silhouette generally depends on the shading point, I identify cells in the scene where the overall structure of the silhouette remains the same. I precompute this for each cell once per unique light source and then determine the exact silhouette at the time of shading. I organize the cells using Binary Space Partitioning (BSP) to allow for fast indexing of the precomputed data. I describe each of the main components of my technique for convex light sources first and then explain how to overcome the additional challenges arising from non-convex ones.

The ability to use polyhedral light sources efficiently is of great practical significance as any solid can be approximated by a polyhedron. Light sources in reality and fiction have a wide variety of complex shapes that cannot always be recreated accurately with other geometric primitives. Examples include various kinds of light fixtures, neon signs, hot metal objects, and some depictions of fictional objects and materials like magical items. Furthermore, sophisticated tools for modeling polyhedra are readily available and widely used. Polyhedral light shading therefore enables artists to apply their workflows and experience from 3D-modeling of other objects to light sources. Similarly, many techniques for scanning real objects produce polyhedral descriptions of the scanned object.

While LTCs make it easy to calculate shading for an unoccluded light source, they don't reproduce shadows cast by other objects in the scene. Recent developments in graphics hardware make ray tracing a viable approach for soft real-time shadows. Particularly good

results are achieved using solid angle sampling. While solid angle sampling is possible using just the polyhedron's faces, I describe how using polyhedron silhouettes as produced by the techniques presented here speeds up the process. I use a solid angle sampling technique for triangles [Pet20] that I extend to polyhedra by triangulating their silhouettes.

After explaining my techniques, I evaluate how practical they are by showing their quality and limitations in example renderings and by discussing performance metrics that I measured for various example light sources. I conclude the thesis by reflecting on the presented techniques, their advantages, and their limitations and by outlining my ideas for improving them with further research.

2. Related Work

Area light shading for various light shapes has already been explored in great detail. One approach [WLWF08] approximates an area light by replacing it with a single point light chosen dynamically for each surface point. The position of this point light is chosen as the point that maximizes the BRDF. The intensity of the point light is determined by integrating the intensity distribution of the light source over a disk around the point light. Empirical estimates are used to determine the radius of the disk. A more recent technique [HDHN16] for polygonal lights uses the actual shape of the light source but approximates the cosine-weighted BRDF with an LTC, which is then integrated analytically for a spherical polygon. This technique forms the basis for large parts of this thesis and is described in more detail in 3.4. Shading with LTCs does not reproduce shadows but a separate publication [HHM18] describes how to combine it with stochastic raytraced soft shadows. To do so, shading is calculated three times: once analytically with the approximated BRDF, once with a Monte Carlo ray tracing approach with shadowing, and once with the same approach and the same random numbers but ignoring shadowing. The ratio of the last two is then multiplied with the first one to give the final result. This results in noise-free shading except for the penumbra, which exhibits the typical noise of stochastic shadows. A denoising technique is applied to the shadow term to reduce the noise in the penumbra while details from textures and normal maps are preserved. A different combination of ray tracing and LTCs [DGJ+20] uses LTCs as an approximation of the BRDF for pathguiding. Using the approximated BRDF, more rays are cast in the directions that have the greatest effect on the result.

Ray tracing for area light shading yields better results when a sampling technique is used that distributes samples uniformly over the solid angle of the light source. A technique for solid angle sampling of triangles exists [Arv95] and can be extended to polygons and polyhedra by triangulating them. A method based on rejection sampling has been presented for disks and cylinders [Gam16]. Instead of sampling the solid angle of the light source directly, a spherical rectangle that contains the solid angle of the light source is sampled until a sample that lies within the solid angle of the light source is found.

An analytic approach to calculating shadows in scenes of opaque convex polyhedra and convex polyhedral light sources [DG09] determines boundaries of umbra and penumbra as polygons. This is made possible by first building a data structure that partitions the scene using a set of planes that they call visual event surfaces where relevant changes to the view of the scene occur. I use a similar approach in section 5.

Different types of contours and silhouettes of polyhedra have been studied in various contexts. A survey [IFH+03] compares many different techniques for detecting contour edges, mostly for highlighting edges in depictions of polyhedra for artistic purposes or to make the shape more easily recognizable. Many of the presented algorithms rely on screen-space pixel buffers but analytically determined silhouettes are also explored to some extent. Arthur Appel introduces the notion of quantitative invisibility [App67] as the number of surface points between the observer and the background. This plays an important role in algorithms for determining silhouettes as the silhouette is what separates an object from the background, i.e. the set of points where the quantitative invisibility changes between one and zero. A complete and efficient algorithm based on a sweep line approach [KW97] uses this idea to analytically determine the silhouette of a polyhedron. However, this algorithm uses a planar projection of the polyhedron, such as a perspective or orthographic projection, which is not always possible in polyhedral light shading. Furthermore, the algorithm does not appear to be well suited for being used in a fragment shader. For example, it requires an efficient sorting algorithm, which is challenging in a fragment shader.

It is worth noting that the inverse problem, namely finding a solid that produces a set of given silhouettes has also been explored in great detail. For example, [FB09] describes an efficient algorithm for use in computer vision that produces a polyhedral model from polygon silhouettes.

3. Basics

My work is based on research on a wide range of topics. In this section, I give an overview of some of the concepts and techniques of these areas of research that are particularly relevant for my thesis. I explain the fundamentals of area light shading and BRDFs, give an introduction to the relevant concepts of spherical geometry and briefly explain the techniques of polygonal light shading with LTCs and with Monte Carlo integration and solid angle sampling.

3.1 BRDF

To shade a surface, it is necessary to describe how light reflects off that surface. A common way of doing so is with a BRDF, which is defined by the following equation [PJH16]:

$$f(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{\mathrm{d}L_o(x, \vec{\omega}_o)}{L_i(x, \vec{\omega}_i)(\vec{n} \cdot \vec{\omega}_i)\mathrm{d}\vec{\omega}_i} \tag{3.1}$$

- L_i and L_o are the incoming and outgoing radiance
- $\vec{\omega}_i$ and $\vec{\omega}_o$ are the directions of incoming and outgoing light
- $f(x, \vec{\omega}_i, \vec{\omega}_o)$ is the BRDF
- $\vec{n}$ is the normal vector

The BRDF takes a direction of incoming radiance and one of outgoing radiance. It returns a value that is a measure of how much of the incoming radiance is reflected in the given direction.

While the BRDF could be measured and passed to a shader directly, this results in impractically large amounts of data and is logistically difficult in projects that require many different materials or many slight variations of a material. To solve this, analytical descriptions of BRDFs for various materials have been developed that are then parameterized with more compact and intuitive properties of the surface, such as color or roughness. The two that I mainly used are given below.

Lambert

Lambert diffuse lighting models ideally diffuse surfaces as reflecting light equally in all directions. The BRDF is therefore simply a constant that is a measure of the brightness or color of the surface [SM09].

$$f(\vec{\omega}_i, \vec{\omega}_o) = const. \tag{3.2}$$

While there are more accurate diffuse BRDFs for realistic materials, Lambert lighting is noteworthy in the context of shading with LTCs because it already is a cosine distribution, which means that using LTCs does not introduce any approximation error.

GGX

GGX [WMLT07] is a physically based specular BRDF for surfaces of varying roughness. It is based on the Torrance-Sparrow lighting model as described in [TS67] which models a surface as consisting of many small, randomly arranged mirrors called microfacets (figure 3.1). By assuming certain statistical distributions for the arrangement of these microfacets, a BRDF can be derived. The general formula for a Torrance-Sparrow BRDF is

$$f(\vec{\omega}_i, \vec{\omega}_o) = \frac{F(\vec{\omega}_i, \vec{h}) D(\vec{n}, \vec{h}) G(\vec{\omega}_i, \vec{\omega}_o, \vec{n})}{4|\vec{\omega}_i \cdot \vec{n}||\vec{\omega}_o \cdot \vec{n}|} \tag{3.3}$$

- $\vec{h}$ is the unit vector halfway between $\vec{\omega}_i$ and $\vec{\omega}_o$
- F is the Fresnel term. A common approximation for this is

$$F(\vec{\omega}_i, \vec{h}) = (1 - \vec{\omega}_i \cdot \vec{h})^5 \tag{3.4}$$

- D describes the distribution of the microfacets. For GGX, D is given by

$$D(\vec{n}, \vec{h}) = \frac{\alpha^2}{\pi((1 + (\alpha^2 - 1)(\vec{n} \cdot \vec{h})^2)^2} \tag{3.5}$$

  where α is the roughness of the surface between one for ideally rough surfaces and zero for perfect mirrors.

- G describes how much microfacets shadow and mask each other. For GGX, G is given by

$$G(\vec{\omega}_i, \vec{\omega}_o, \vec{n}) = G_1(\vec{\omega}_i, \vec{n}) G_1(\vec{\omega}_o, \vec{n}) \tag{3.6}$$

$$G_1(\vec{\omega}, \vec{n}) = \frac{2}{1 + \sqrt{1 + a^2((\frac{1}{\vec{n} \cdot \vec{\omega}})^2 - 1)}} \tag{3.7}$$

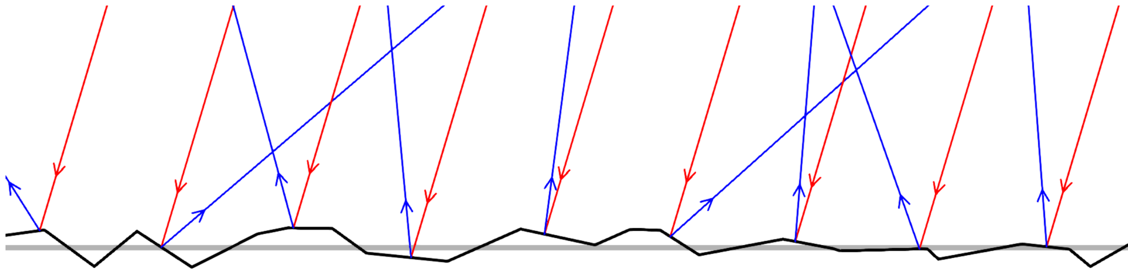

Figure 3.1: Light rays (incoming red, reflected blue) are reflected by the microfacets (black) of a surface (gray).

3.2 Rendering Equation and Area Lights

Illumination of a surface point in a three-dimensional scene is described by the rendering equation [ICG86] as follows:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int_\Omega L_i(x, \vec{\omega}_i) f(x, \vec{\omega}_i, \vec{\omega}_o) \max\{0, \vec{\omega}_i \cdot \vec{n}\} d\vec{\omega}_i \tag{3.8}$$

- $L_o(x, \vec{\omega})$ is the outgoing radiance at a surface point x in a direction $\vec{\omega}$
- $L_e(x, \vec{\omega})$ is the radiance emitted by a surface point x in a direction $\vec{\omega}$
- $L_i(x, \vec{\omega})$ is the incoming radiance at a surface point x from a direction $\vec{\omega}$
- $f(x, \vec{\omega}_i, \vec{\omega}_o)$ is a BRDF of the surface
- $\vec{n}$ is the normal vector of the surface at x
- Ω is the unit sphere

The outgoing radiance is the sum of the radiance emitted by the surface itself and the reflected radiance. The reflected radiance is an integral over the unit sphere because light can hit the surface from any direction and be reflected in a different direction. The integrand is the product of incoming radiance, the BRDF, and the clamped dot product of incoming light direction and surface normal. The clamped dot product accounts for the effect that light hitting the surface at a smaller angle is distributed over a larger area.

In this general version of the rendering equation, light can hit a surface point from any direction as described by $L_i(x, \vec{\omega})$. An important subproblem of calculating the total outgoing radiance is to only consider the light that has not been reflected yet (direct light). This subproblem is easier to solve, especially if the light sources have relatively simple shapes. Point and directional lights are commonly used to keep lighting calculations simple but other shapes are also used, such as spheres, disks, rectangles, and polygons. To distinguish them from point and directional lights, light sources that are not infinitesimally small are called *area lights*. The contributions of each light source to the integral can be calculated separately and then summed up to give the value of the integral.

For area lights, it is still necessary to calculate an integral but the integrand is now zero for any direction that is not in the solid angle of the light source. The solid angle of a light source for a shading point is the set of directions that point from the shading point towards a point on the light source:

$$\Omega_S(x) = \left\{ \frac{p - x}{||p - x||} \mid p \in S \right\} \tag{3.9}$$

- S is a light source,
- x is a shading point,
- $\Omega_S(x)$ is the solid angle of S as seen from x.

To further simplify the calculation, I restrict light sources to Lambertian emitters, i.e. light sources that emit the same radiance from each surface point and in each direction. With this simplification, the radiance reflected by a surface point from a single, unoccluded light source is described by the following equation:

$$L_o(x, \vec{\omega}_o) = L_i \int_{\Omega_S(x)} f(x, \vec{\omega}_i, \vec{\omega}_o) \max\{0, \vec{\omega}_i \cdot \vec{n}\} d\vec{\omega}_i \tag{3.10}$$

- $\Omega_S(x)$ is the solid angle of the light source as seen from x,

- L_i is the incoming radiance, which is now the same for each direction in the solid angle of the light source and can therefore be moved outside the integral.

This means that the challenging part of shading consists mainly of determining the solid angle of the light source and integrating the cosine-weighted BRDF over that solid angle. Nonetheless, this is often difficult depending on the shape of the light source and the BRDF.

3.3 Spherical Triangles and Polygons

As explained in section 3.2, the solid angle of a light source is an important concept for area light shading. So far, the solid angle was defined in a very general way but because I deal with polyhedral light sources in this thesis, it is useful to also have more specialized ways of describing the solid angles of specifically these light sources. The solid angle of a polyhedron is a spherical polygon. For solid angle sampling, I will triangulate these spherical polygons to obtain an equivalent set of spherical triangles. In this section, I explain spherical triangles and polygons and how I represent them for the algorithms and data structures I describe.

The solid angle of a polyhedron is the union of the solid angles of its polygonal faces. The solid angle of a polygon is a spherical polygon (figure 3.2). The boundary of that spherical

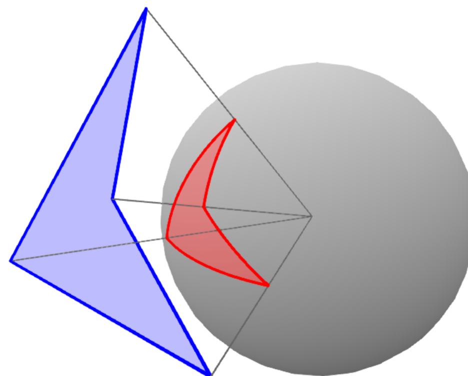

Figure 3.2: A spherical polygon (red) that is created by projecting a polygon (blue) onto a sphere.

polygon is the projection of the edges of the polygon. When a line segment is projected onto a sphere, it becomes a great circle arc. A great circle is a a circle with the same radius and center as the sphere. The great circle arc corresponding to a line segment is simply the shortest connection of the projections of its endpoints on the sphere. In this thesis, I will only use great circle arcs that are projections of line segments so I simply use their endpoints to describe them.

The solid angle of a polyhedron is a spherical polygon as well. However, it is not necessarily the projection of any single planar polygon as it can be longer than π. This raises the question of what the inside of a spherical polygon is. If a spherical polygon was always a projection of a polygon then one could simply say that its inside is the side that corresponds to the inside of the polygon. For the solid angle of a polyhedron, it makes sense that the inside must be the side that corresponds to the interior of the polyhedron. In other words,

the projection of a point inside the polyhedron must also be inside its solid angle. If however a spherical polygon is just given as a set of edges then there is no way of telling which side is the inside. One way of adding that information would be to also store a point inside the spherical polygon along with the set of edges. In this thesis, however, I chose a different way. I describe each edge as an ordered pair of endpoints and define that the edges of a spherical polygon must be counter-clockwise around its inside, meaning that the inside must be on the left of each edge. This aligns with the commonly used convention in computer graphics that counter-clockwise edge loops describe filled shapes while clockwise edge loops describe holes to be cut out from a shape. It also has the advantage that for any single edge it is always clear, which side of the edge is the inside of the spherical polygon.

For solid angle sampling, I triangulate the solid angle of a light source, i.e. I split the spherical polygon into multiple spherical triangles. A spherical triangle in this thesis is always the projection of a triangle so I simply use its three vertices to represent it. At first I also require a triangulation to use only the vertices of the spherical polygon but I drop this requirement in section 6.2.2 for practical reasons. My goal is still to use mostly the vertices of the spherical polygon but sometimes it is necessary to add additional ones.

3.4 Polygonal Light Shading with Linearly Transformed Cosines

Shading with LTCs [HDHN16] is an efficient and accurate approximation for polygonal light sources. The cosine weighted BRDF is approximated by a spherical distribution from the class of LTCs, which can efficiently be integrated analytically over a polygon. I will later extend this technique to polyhedral light sources but first, I explain how to use it for polygonal ones as described in the original paper.

A spherical distribution assigns a density to every point on the unit sphere. A linearly transformed spherical distribution is obtained by applying a linear transformation to the direction vectors of a spherical distribution. The linear transformation is represented by a 3×3 matrix. The density of a linearly transformed spherical distribution for a direction $\vec{\omega}$ is given by

$$\hat{D}(\vec{\omega}) = D\left(\frac{M^{-1}\vec{\omega}}{||M^{-1}\vec{\omega}||}\right)\frac{|M^{-1}|}{||M^{-1}\vec{\omega}||^3} \tag{3.11}$$

- D is the original spherical distribution
- $\hat{D}$ is the linearly transformed spherical distribution
- M is the matrix encoding the linear transformation

An important property of linearly transformed spherical distributions in the context of area light shading is that the integral of the transformed spherical distribution over a polygon is equivalent to integrating the original spherical distribution over a different polygon that is obtained by applying the inverse transformation to the original one:

$$\int_P \hat{D}(\vec{\omega})d\vec{\omega} = \int_{M^{-1}P} D(\vec{\omega})d\vec{\omega} \tag{3.12}$$

This means that if D is a distribution that can be integrated easily over a polygon then the same is true for $\hat{D}$. A distribution where that is the case is the clamped cosine distribution:

$$D_c(\vec{\omega}) = \max\left\{0,\ \vec{\omega} \cdot \begin{bmatrix}0\\0\\1\end{bmatrix}\right\} \tag{3.13}$$

To integrate the clamped cosine distribution over a polygon, the polygon is first clipped to the upper hemisphere. This has the effect that the parts of the polygon where the dot

product is negative and would therefore be clamped to zero are excluded from the integral. The integral for the clipped polygon is then given by a sum over a closed-form expression for each edge of the polygon [BRW89]:

$$\int_P D_c(\vec{\omega}) = \frac{1}{2\pi} \sum_{(a,b)\in E} \arccos(a \cdot b) \left(\frac{a \times b}{||a \times b||} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) \tag{3.14}$$

E is the set of edges of the polygon, given as tuples of start and endpoint. The points are given as unit vectors since only their direction from the shading point matters, not their distance to it. Note that the order of start and endpoint matters. Swapping them flips the sign of the contribution of the edge. Edges should be given such that they form a closed edge loop around the interior of the polygon. The sign of the result reflects whether the edge loop is clockwise or counterclockwise.

The clamped cosine distribution corresponds to the Lambert lighting model if the surface normal is the positive z-axis of the coordinate system. Integrating it over a polygon is equivalent to Lambert shading for a polygonal light source. That the normal corresponds to the positive z-axis can be ensured by choosing the coordinate system accordingly. Various other lighting models are approximated by applying a linear transformation to the clamped cosine distribution. For example, specular reflection resembles a clamped cosine distribution that has been transformed to have its maximum in the reflected view direction and that has been scaled to give a certain size of the specular highlight. The transformation is generally not constant as it may depend on several parameters such as the view direction or the roughness of the surface.

Because calculating the transformation matrix that best approximates the desired BRDF is too expensive to happen during shading, it is precalculated for a large number of configurations of all its parameters and stored in a lookup table. Note that for isotropic BRDFs, the view direction is only a one-dimensional parameter since the rotation of the surface around the normal has no effect on the result. The coordinate system used for lighting calculations can therefore be rotated freely around the surface normal. In particular, it can be chosen such that one of its two remaining axes (that are not the surface normal) is as close as possible to the view direction. This choice of the coordinate system is illustrated in figure 3.3. The view direction in this coordinate system is then uniquely determined by

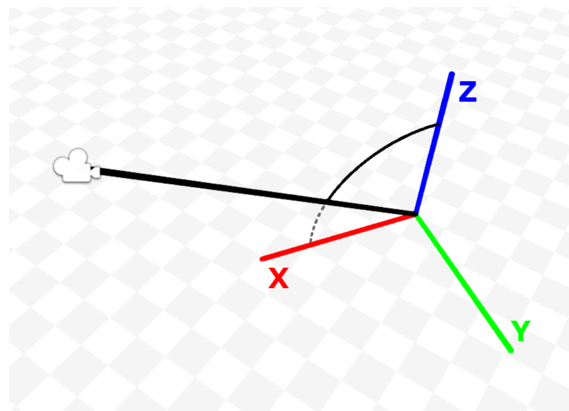

Figure 3.3: In the coordinate system used for shading with LTCs, the Z-axis (blue) is the surface normal and the X-axis (red) is as close as possible to the view direction (black), so that the view direction always lies in the XZ-plane and is therefore uniquely determined by its angle to the Z-axis.

its angle to the surface normal. If roughness is chosen as an additional parameter for the

transformation matrix as is useful for specular reflection then a two-dimensional lookup table is necessary, which in practice is easily possible with sufficient accuracy.

Finding the matrix that best approximates the BRDF is a non-linear optimization problem that is solved with the Nelder-Mead method [NM65] in the example code provided with the paper. The error function that is minimized is the numerically calculated L^3 error of the approximation. To calculate the error, Monte Carlo integration (section 3.5) with importance sampling is used.

To sum up the technique, the following steps are necessary to calculate shading with LTCs:

1. Precalculate a lookup table that stores the inverse of the transformations that best approximate the cosine-weighted BRDF for all configurations of the parameters it depends on (excluding the direction of incoming light),
2. In the fragment shader, determine the parameters for the lookup table and read the inverse transformation matrix,
3. Transform the light source into the coordinate system of the shading point and apply the inverse transformation matrix,
4. Clip the light source to the upper hemisphere,
5. Calculate the integral.

3.5 Monte Carlo Integration

Polygonal light shading with LTCs calculates the shading integral analytically. Another option is to calculate it numerically. One way of doing so is Monte Carlo integration, which I explain in this section based on a paper by Shirley et al. [SWZ96].

Monte Carlo integration is a randomized algorithm that approximates an integral over a domain numerically by taking several samples from the domain and calculating a sum over these samples. The expected value of a function of a random variable is an integral over the domain of the random variable:

$$E[f(X)] = \int_S f(x)p(x)d\mu(x) \tag{3.15}$$

- S is a domain,
- f is a function defined on S,
- μ is a measure on S,
- p is a probability density on S,
- X is a random variable in S with probability density p.

In the most simple case where S is an interval $[a, b]$ in $\mathbb{R}$, μ is the Lesbesque measure and X is uniform in S, this simply means that

$$E[f(X)] = \frac{1}{|a-b|}\int_a^b f(x)dx \tag{3.16}$$

However, I will use integrals over solid angles and with non-uniform distributions so the more general form is necessary. The expected value and thereby the integral can be approximated numerically using a sum over random samples instead:

$$E[f(X)] = \int_S f(x)p(x)d\mu(x) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{3.17}$$

- $x_1, x_2, ..., x_N$ are instantiations of X.

For large N, this approximation converges almost certainly towards the correct value according to the law of large numbers. The integral of $f(x)p(x)$ can therefore be approximated with a simple randomized algorithm. To use this in practice, equation 3.17 is rearranged so that an integral of just $f(x)$ is approximated instead:

$$\int_S f(x)d\mu(x) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{3.18}$$

The choice of p affects the variance of the approximation. If p is proportional to f, then $\frac{f(x)}{p(x)}$ is constant and the variance therefore zero. That distribution is usually not known but choosing one that is close to being proportional to f already reduces the variance compared to other distributions. This approach is known as importance sampling. Another approach is to pick a distribution that makes sampling more efficient so that more samples are calculated in the same time.

3.6 Solid Angle Sampling

For area light shading, a function must be integrated over the solid angle of the light source. If this is done with Monte Carlo integration then a sampling technique for the solid angle of the light source is required. Solid angle sampling typically gives better results than area sampling. I explain the difference between the two in this section.

Area sampling refers to sampling the surface of the light source uniformly and using the direction from the shading point to that point as the sample. This however does not give a uniform distribution over the solid angle of the light source so a geometry term is introduced into equation 3.18. The density of the distribution depends on the distance of the sample to the shading point and on the angle between the surface normal and the vector from the shading point to the sample. Parts of the light source that are further away or are viewed at a smaller angle have a smaller solid angle and should therefore contribute less to the result. The following geometry term must be used:

$$p_g(x) = \frac{(x-s)\cdot\vec{n}}{||x-s||^2} \tag{3.19}$$

- s is the shading point,
- $\vec{n}$ is the surface normal at the sample.

This almost always increases the variance of the Monte Carlo integration, compared to uniform sampling of the solid angle. Greater variance gives more noisy results because on average, the result deviates more from the correct value.

Solid angle sampling on the other hand refers to picking samples uniformly from the solid angle of the light source. This eliminates the geometry term, which leads to lower variance. Sampling the solid angle of most light sources is more difficult than sampling their area. Nonetheless, solutions for many relevant types of light sources have been found. For example, a technique for sampling the solid angle of a triangle exists [Arv95], which can be extended to polygons by first triangulating the polygon. The technique first splits a spherical triangle into two smaller ones by adding a new arc from one of its vertices to the opposite side. The solid angle of the first sub-triangle is picked uniformly between zero and the total solid angle. The sampling direction is then picked with an appropriate distribution from the arc splitting the triangle. Overall this results in a uniform distribution over the spherical triangle.

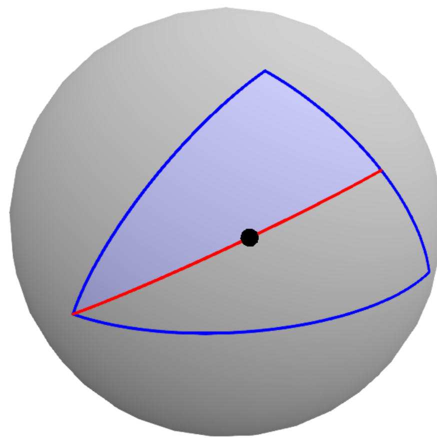

Figure 3.4: To sample a spherical triangle, the triangle (blue) is first split into two by adding a new arc (red). The solid angle of the first smaller triangle (light blue) is chosen uniformly between zero and the total solid angle. The sample (black) is then taken from the new arc.

4. Finding Silhouettes for Polyhedral Light Shading with LTCs

As described in 3.4, LTCs can be used to efficiently shade polygonal light sources but many real-world light sources are not flat and are therefore not approximated well when using this technique directly. In this section, I extend the technique first to convex polyhedral light sources and then to non-convex ones. My approach replaces the light source with its silhouette polygon so an important problem I discuss in this section is how I determine the silhouette of a polyhedron. In this context, I also explain the concept of the contour of a polyhedron, which is a superset of the silhouette and the first step for finding the silhouette.

As established in equation 3.10, the relevant characteristics of the light source for area light shading are its emitted radiance L_i and its solid angle $\Omega_L(x)$. Therefore, the same result is achieved when the light source is replaced with one that preserves these two characteristics. Since LTCs work well with polygons, it makes sense to replace a polyhedral light source with an equivalent polygonal light source. One way of doing this would be to make each face of the polyhedron a separate polygonal light source but this has two disadvantages:

First, polygonal light shading with LTCs mainly consists of evaluating an expression for every edge of the light source. If this needs to be done for every edge of every front-facing face of a polyhedral light source then all edges that belong to two front-facing faces are calculated twice and their contributions cancel out. This means that a considerable amount of processing time is wasted on inconsequential calculations.

Second, non-convex polyhedra can occlude themselves. If faces are treated as separate light sources then the directions where two faces overlap are counted twice, leading to a wrong result. For example, overlapping parts of the light source cause the corresponding parts of a specular highlight to appear much brighter than they should. While this is just one effect of the limitation that shading with LTCs does not reproduce shadows, it is visually distinct and in some scenes very noticeable. Ignoring shadowing for other objects merely results in a lack of shadows while ignoring self-occlusion of light sources adds bright artifacts to the scene (figure 4.1).

Instead, I therefore replace the light source with the interior of its *silhouette* as seen from the shading point. This solves both of the problems described above.

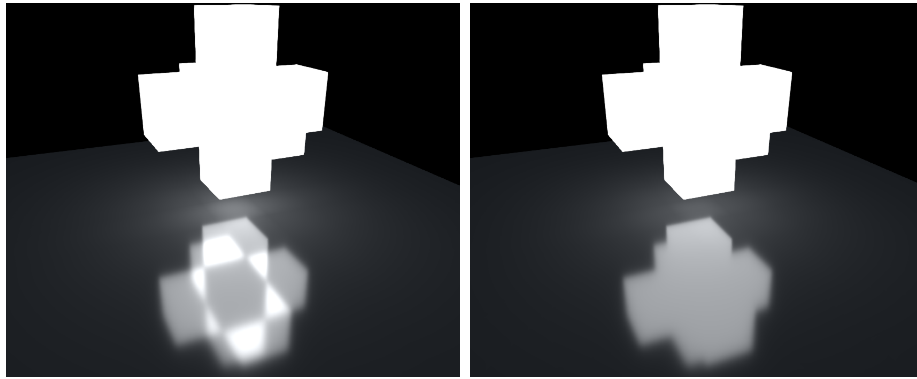

Figure 4.1: A specular highlight with bright artifacts due to treating each face as a separate light source (left) and the correct specular highlight without artifacts (right)

4.1 Contour and Silhouette of a Polyhedron

In other literature, the term *silhouette* is used for mainly two different concepts that I call *silhouette* and *contour* in my thesis. I follow the definitions used by Kettner and Welzl [KW97], which I explain in this section. While my technique mostly revolves around silhouettes, the contour is also relevant as it is a superset of the silhouette and finding the contour is the first step of my techniques for finding the silhouette. Figure 4.2 compares the contour and silhouette of a polyhedron.

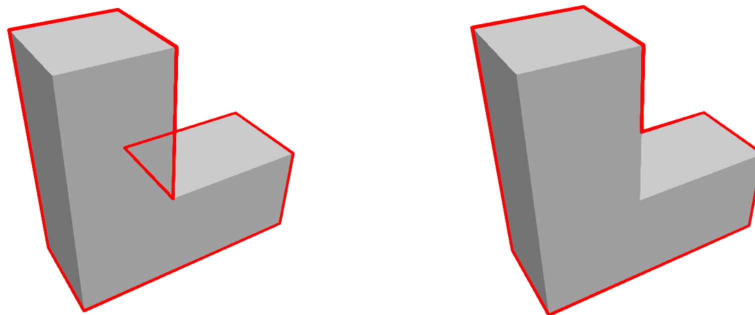

Figure 4.2: The contour (left) and silhouette (right) of a polyhedron

The silhouette is what separates the polyhedron from the background when it is viewed from a given point (in my thesis always the shading point). More precisely, the silhouette is the subset of a polyhedron where the projection of each point in that subset lies on the boundary of the projected polyhedron. The projection of the silhouette is a spherical polygon. If the shading point is coplanar with a face of the polyhedron then that entire face may be part of the silhouette by the definition above. However, I will mostly ignore these cases as the probability of this happening for a random point in a scene is zero and any case where it does happen can be dealt with by applying an arbitrarily small translation to the shading point. I therefore assume that the silhouette consists of segments of edges of the polyhedron, as is the case for all other points. This simplifies my algorithms for finding silhouettes.

The contour is the subset of the edges of the polyhedron where exactly one of the two faces adjacent to each edge is facing an observer located at a given point. Equivalently to this, the two faces adjacent to a contour edge as seen by the observer lie on the same side of the edge. This is also a necessary condition for a silhouette edge since any edge where the two faces lie on opposite sides of the edge clearly has parts of the polyhedron touching it on either side (namely the two adjacent faces). Therefore, it cannot separate the polyhedron from the background. My approach to finding the silhouette is to first find the contour and then reject the segments of it that do not belong to the silhouette.

Whether a face is front-facing is determined by the sign of the dot product of its normal and the vector from any point on the face to the observer:

$$f \text{ is front-facing} \iff \vec{n} \cdot (F - O) > 0 \tag{4.1}$$

- f is a face of the polyhedron,
- $\vec{n}$ is the normal of f,
- F is a point on f,
- O is the observer.

Whether an edge is a contour edge is therefore given by the sign of this expression (a negative sign indicates a contour edge):

$$(\vec{n}_1 \cdot (E - O))(\vec{n}_2 \cdot (E - O)) \tag{4.2}$$

- $\vec{n}_1$ and $\vec{n}_2$ are the normals of the two adjacent faces
- E is a point on the edge (and therefore on both faces)

4.2 Convex Polyhedra

Convex polyhedra are an important subclass of polyhedra. Many problems are easier to solve for convex polyhedra and polyhedral light shading is no exception to this. In this section, I describe my technique for convex polyhedral light shading with LTCs. I explain how I find the silhouette and how I clip the silhouette to the upper hemisphere so that I can integrate a clamped cosine distribution as described in section 3.4.

A convex polyhedron is one where the line segment connecting any two points inside the polyhedron is contained entirely in the polyhedron. A convex polyhedron is the intersection of a finite set of half-spaces and each of its faces lies in the plane of one of those half-spaces. Many problems are easier to solve for convex shapes than for non-convex ones and finding the silhouette is no exception to this. It has already been established that the silhouette consists of segments of contour edges. For convex polyhedra, determining these segments is trivial once the contour is known: every contour edge is a silhouette edge.

To understand why this is the case, let c be a convex contour edge. The two adjacent faces of c lie on the same side of it as seen by the observer. The intersection I of the corresponding half-spaces also lies entirely on that side of c. The polyhedron is a subset of I so it must lie entirely on that side of c as well. Because c is an edge of the polyhedron but the polyhedron is only on one side of it as seen from the shading point, it must be a silhouette edge.

With these considerations, my algorithm for finding the silhouette of a convex polyhedron is very straightforward: iterate over all edges of the polyhedron and emit those that are contour edges (equation 4.2) as silhouette edges.

To use this algorithm for polyhedral light shading, I pass the light source to the shader as a buffer containing one element for each edge of the polyhedron. Each element stores the start and endpoint of the edge as well as the two normals of the adjacent faces. The order of the edges is irrelevant as their contributions to the result are simply summed up. However, as mentioned in section 3.4, the order of start and endpoint of each edge matters. Each edge should be oriented such that the interior of the silhouette is always on the left-hand side of the edge so that the edges could be arranged to form a counter-clockwise edge loop. This is a problem because the silhouette polygon is not known when the edge-buffer is created. An interesting observation in this context is that the correct direction of the edge is always counter-clockwise not just around the silhouette but also around the adjacent face that is facing the observer. Recall that always exactly one of the two adjacent faces is front-facing since the edge would not be a contour edge otherwise. Knowing this, I choose the direction of an edge such that it is correct under the assumption that the first one of the adjacent faces is front-facing. If at the time of shading I determine that this is not the case, I simply swap start and endpoint since a polyhedron edge is always clockwise around one of its adjacent faces and counter-clockwise around the other one.

Another problem I had to solve is clipping the silhouette to the upper hemisphere to account for the clamping in the cosine distribution as described in section 3.4. To do so, I categorize each edge as one of three cases:

1. The edge is entirely in the upper hemisphere and is simply taken as it is,
2. The edge is entirely in the lower hemisphere and is ignored entirely,
3. One endpoint of the edge is in the upper hemisphere, the other one in the lower hemisphere.

In the last case, the edge must be shortened so that only the segment in the upper hemisphere remains. To do so, I replace the point in the lower hemisphere with the point where the edge intersects the XY-plane:

$$B_{new} = A + \frac{A_z}{A_z - B_z}(B - A) \tag{4.3}$$

(A, B) is the edge and B is in the lower hemisphere.

This correctly clips all existing edges but clipping can also create new edges where the polygon intersects the XY-plane. For convex polyhedra, at most one such edge is created because no more than two silhouette edges intersect the XY-plane. The new edge connects these two intersections if they exist.

In summary, I implement convex polyhedral light shading with LTCs and silhouettes as following:

1. Store the edges of the polyhedron along with the normals of the adjacent faces in a buffer. Each edge must be oriented such that it is counter-clockwise around the first adjacent face,
2. For each edge:
   - Ignore the edge unless it is a contour edge,
   - Transform the edge into the coordinate system of the shading point and apply the LTC-transformation,
   - Ignore the edge if it is entirely in the lower hemisphere,

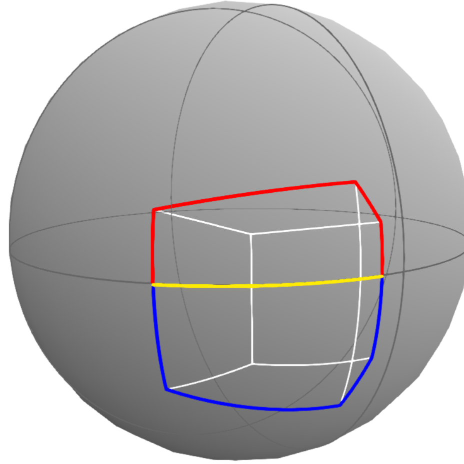

Figure 4.3: To clip this projected silhouette of a cube, it is split into a part in the upper hemisphere (red) and one in the lower one (blue), which is then discarded. One new edge (yellow) is created in the clipping plane to close the shape.

- Clip the edge if it is partially in the lower hemisphere and store the intersection with the XY-plane,
- Flip the edge if the first face is back-facing,
- Calculate the contribution of the edge and add it to the result,

3. Calculate the contribution of the new edge connecting the two intersections and add it to the result.

In section 4.3, I describe a different technique for clipping the silhouette of any polyhedron (not just convex ones), which is more complicated but similarly fast so one may choose to implement clipping that way for convex polyhedra as well.

4.3 Non-Convex Polyhedra

Two new problems arise when the technique described above is extended to non-convex polyhedra: First, determining the contour edge segments that are silhouette edges is more difficult because a segment of a contour edge can now be directly in front of or directly behind a different part of the polyhedron and may therefore not separate the polyhedron from the background. If that happens, it is necessary to split the contour edge into multiple segments and reject some of them. Second, more than one new edge may be created when clipping the silhouette, which makes finding the new edges much more difficult. In this section, I describe how I solved these two problems in principle. My technique for finding the silhouette is not fast enough for real-time shading as it is but, in section 5, I solve this problem by moving significant parts of it to a preprocessing phase.

To find the silhouette edges, I first generate a larger set of contour edge segments that contains all the desired segments but also makes it easy to reject the undesired ones. When projected onto the sphere around the shading point, the silhouette consists of closed edge loops. Therefore, a silhouette edge starts and ends where other silhouette edges start or end. This also means that every silhouette edge starts and ends on another contour edge because each silhouette edge is a segment of a contour edge. I will use the term

apparent intersection of a pair of edges to refer to the direction where the projections of the edges intersect, i.e. the direction where the edges appear to intersect as seen from the shading point. If every silhouette edge starts and ends at an apparent intersection with another contour edge then cutting contour edges into segments at these points yields a set of contour edge segments from which the silhouette can be assembled.

To determine the apparent intersection of two edges, I first determine the planes that their projections (which are great circle arcs) lie in. These planes contain the shading point and the two endpoints of the edge. The apparent intersection of the edges must lie in the intersection of the two planes. If the planes are not parallel, their intersection is the line through the shading point that is orthogonal to both normals. This leaves two points for the intersection. If either of them lies on both arcs then it is the apparent intersection of the two edges. I determine the two candidates using three cross products:

$$p_1 = ((a_1 - s) \times (a_2 - s)) \times ((b_1 - s) \times (b_2 - s)) \tag{4.4}$$

- (a_1, a_2) and (b_1, b_2) are the edges,
- s is the shading point.

To find all apparent intersections, I iterate over all pairs of contour edges, which requires quadratic time and is therefore not ideal for a fragment shader. If the polyhedron is projected into a plane, a sweep line algorithm [KW97] would solve the problem faster. However, this does not appear to be well suited for a fragment shader either as it requires an efficient sorting algorithm. Also, adapting it to spherical projections rather than planar ones is not trivial. This is one of the reasons why I extensively use preprocessing as described in section 5.

The next step after splitting the contour edges into segments is to iterate over those segments and decide whether they are silhouette edges or must be discarded. A segment must be discarded if and only if it is directly in front of or directly behind a different part of the polyhedron. To test whether this is true for a single point on the edge, I use a ray-polyhedron intersection test. If the ray through the point on the segment does not intersect the polyhedron (other than at the edge that the point lies on) then this point must lie on the silhouette. Testing a single point for each segment is sufficient because the edges have already been segmented as much as necessary. It is no longer possible for a part of a segment to be a silhouette edge while another part is not. These considerations lead to the following algorithm for finding silhouette edges:

1. Find the contour edges,
2. Find the apparent intersections of contour edges,
3. Split the contour edges at the apparent intersections,
4. Return all segments where the ray through the midpoint does not intersect the polyhedron at any other point.

Unfortunately, this part of the algorithm is not well suited for being executed in a fragment shader either. While GPU ray tracing does exist, tracing one or more rays for each contour edge is still a relatively slow process. Again, I refer to section 5, where I explain how to precompute this part of the algorithm.

Clipping the silhouette to the upper hemisphere has to happen in the shader and seems to be a challenging task at first. Clipping each silhouette edge works exactly as for convex polyhedra but finding the new edges in the clipping plane is more difficult because there can be more than one, which means that it is not obvious which pairs of intersections between edges and the XY-plane should be connected by a new edge. One way of doing so would

be to sort the intersections by their polar angle in the plane and then connect neighboring points by an edge if that edge would be inside the polygon. However, finding the edges is not actually necessary. I found a more efficient way of calculating the contribution of the new edges to the shading integral by reevaluating how the edges would be used in the formula. Equation 3.14 calculates the shading integral as the sum of an expression for each edge:

$$\int_P D(\vec{\omega}) = \frac{1}{2\pi} \sum_{(a,b)\in E} \arccos(a \cdot b) \left(\frac{a \times b}{||a \times b||} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right)$$

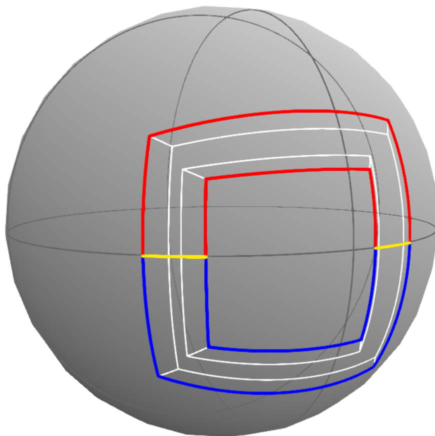

Figure 4.4: To clip this projected silhouette of a non-convex polyhedron, it is split into a part in the upper hemisphere (red) and one in the lower one (blue), which is then discarded. Contrary to convex polyhedra, multiple new edges (yellow) are necessary to close the shape.

The contribution of each edge is a product of two factors. The first one is simply the angle between the endpoints of the edge. The second one is equal to $+1$ or -1 for all of the new edges. All newly created edges lie in the XY-plane. Therefore, the cross product of the two endpoints is parallel to the Z-axis and because the cross product is then normalized, its Z-coordinate has an absolute value of one. This means that the second factor is merely a sign that determines whether the angle covered by the edge should be added to or subtracted from the sum. Using the arccosine of the dot product is just one way of calculating the angle between two unit vectors in the plane. Another one is to take the difference of the polar angles of the endpoints:

$$\text{angle}(A, B) = \text{atan2}(B_y, B_x) - \text{atan2}(A_y, A_x) \tag{4.5}$$

Note that this gives a signed angle with the same sign as one would get from the second factor in the previous equation. With this method of calculating the angle, I rewrite the contribution of the new edges in the clipping plane as a sum of an expression for each endpoint of a new edge. Therefore, I no longer need to find pairs of intersections with the XY-plane that form a new edge. All that remains to be done is finding the correct sign for each intersection. The sign indicates whether the new edge starts at the intersection or ends there. Because the silhouette edges are always counter-clockwise around the polyhedron, this can be determined simply by checking if the edge intersecting the XY-plane is directed

upwards or downwards. The complete formula for calculating the integral is now:

$$\int_P D(\vec{\omega}) = \frac{1}{2\pi} \sum_{(a,b) \in E} \arccos(a \cdot b) \left(\frac{a \times b}{||a \times b||} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) + \frac{1}{2\pi} \sum_{i \in I} \operatorname{atan2}(i_y, i_x) s(i) \tag{4.6}$$

- E now contains only the clipped silhouette edges, *not* the new edges from clipping
- I contains all points where a silhouette edge intersects the XY-plane
- $s(i)$ is $+1$ if the edge of i is directed upwards and -1 otherwise

To speed up the calculation of the second sum, I use the atan2 sum identity:

$$\begin{aligned} \operatorname{atan2}(y_1, x_1) + \operatorname{atan2}(y_2, x_2) &= \operatorname{atan2}(y_1 x_2 + y_2 x_1, x_1 x_2 - y_1 y_2) \\ \operatorname{atan2}(y_1, x_1) - \operatorname{atan2}(y_2, x_2) &= \operatorname{atan2}(y_1 x_2 - y_2 x_1, x_1 x_2 + y_1 y_2) \end{aligned} \tag{4.7}$$

That way, I only calculate one atan2 in total, instead of two for each new edge.

5. Precomputed Silhouettes with Binary Space Partitioning

Finding the silhouette of a polyhedron using the techniques described so far is relatively expensive for a fragment shader. The method for convex light sources works well in practice but it requires iterating over all edges of the light source, even though usually only very few of them are silhouette edges. The method for non-convex light sources on the other hand requires complicated operations like finding all intersections in a set of great circle arcs and intersecting rays with the light source. In both cases, I identified intermediate results that require a significant portion of the processing time and can be precomputed. These intermediate results are chosen such that a single one is always valid for all points inside of a convex polyhedral cell. I store these cells in a BSP-tree so that the correct one for a given point can be found quickly. BSP refers to a common technique for partitioning three-dimensional scenes. A BSP-tree is a binary tree where each inner node stores a plane and each leaf node represents a cell. To determine which cell contains a given point, the tree is traversed starting at the root and traversing to the right child whenever the point is on the positive side of the plane and to the left one otherwise. This results in convex polyhedral cells. The tree structure makes finding the correct cell relatively fast compared to iterating over all of them.

5.1 Convex Polyhedra

Finding the silhouette of a convex polyhedron mainly consists of finding the contour edges. Whether an edge is a contour edge only depends on whether the adjacent faces are front-facing or back-facing. As an observer moves through the scene, faces only transition between being front-facing and back-facing if the observer moves from one side of the plane that the face lies in to the other. This means that as long as the observer stays on the same side of each face of a convex polyhedron, the set of contour edges stays the same. I therefore use these planes as the partitioning planes for a BSP-tree. Figure 5.1 visualizes the planes and the cells they create. The set of contour edges is then the same for all points in the same cell (figure 5.2). In the fragment shader, I determine the cell that the shading point lies in by traversing the tree and then fetch the precomputed set of contour edges from the corresponding leaf. Using this technique improves performance in several ways:

- Instead of iterating over all edges of the light source, the fragment shader now iterates

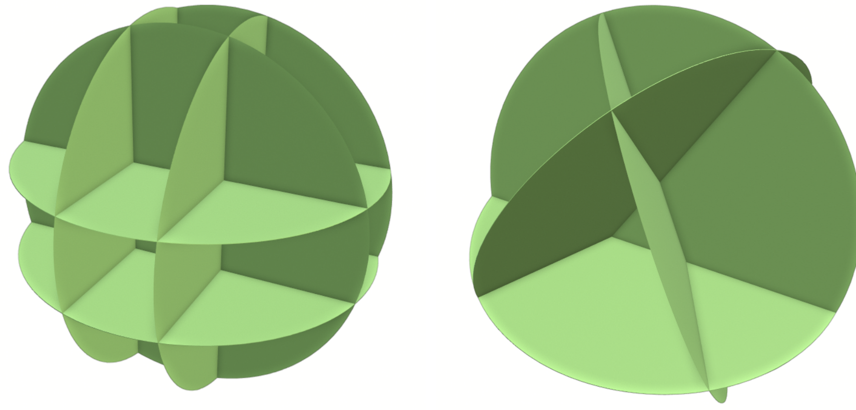

Figure 5.1: The planes of the faces of a cube (left) and a tetrahedron (right), partitioning space into 27 and 15 cells respectively (one of these cells is the polyhedron itself).

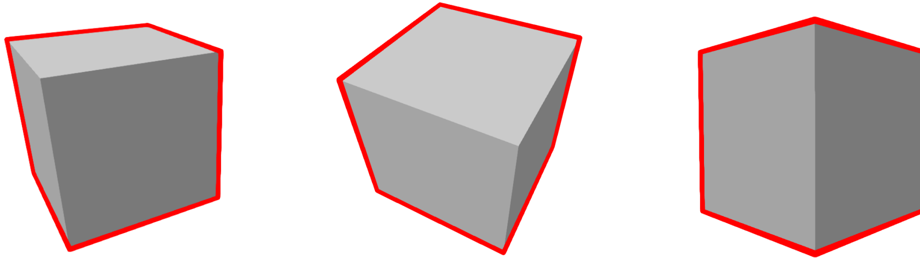

Figure 5.2: The silhouette of a cube from two different points in the same cell (left and middle) and from a point in a different one (right). The silhouette consists of the same edges for the first two but of different ones for the third one.

over faces of the light source. A polyhedron always has fewer faces than edges (because each face has at least three edges and each edge belongs to two faces).

- In most cases, not all faces need to be tested. Often a leaf node is reached after only a few inner nodes.
- I store the precomputed silhouettes as an ordered list of points where each point is connected to its two neighbors. This has the advantage that each vertex of the silhouette only needs to be transformed and normalized once. If the silhouette is given as an unordered list of edges then each vertex must be transformed and normalized once for each of the two adjacent edges.
- I precompute the silhouette as a counter-clockwise edge loop, which makes flipping edges in the fragment shader unnecessary.

5.2 Non-Convex Polyhedra

The technique for convex light sources is not well suited for non-convex ones. I could of course determine the contour edges in the same way but for non-convex light sources, finding the contour edges would only solve a relatively minor problem while the main

problem of finding the right segments of these contour edges remains just as difficult as before. Nonetheless, I decided to stick with the same core idea and adapt it to non-convex shapes. My goal was to precompute and store the entire overall structure of the silhouette, including the start and endpoints of each segment. I had to solve two problems for that: First, a silhouette edge can end at the apparent intersection of two edges, which is not a static point in three-dimensional space. It can move to any point along the two edges depending on the position of the observer. This means that the point itself cannot be precomputed. Second, the planes of the faces of the light source are not sufficient for partitioning the scene. Because the precomputed data now also contains information about intersections of contour edges, I had to further subdivide the scene such that the same apparent intersections between edges exist for all points in a cell.

To solve the first problem, I decided to store each silhouette edge not as a pair of points directly but as a pair of descriptions of how the actual points should be calculated in the shader. I distinguish two possible cases: either the point is a vertex of the light source or it is the apparent intersection of two edges of the light source. In theory, the first case is just a special case of the second one since any vertex of the light source is also the intersection of three or more edges but it is more efficient to treat them differently. To find the apparent intersection of two edges, four vertices must be fetched (the four endpoints of the two edges). This is slower than fetching a single vertex and the first case is far more common for most practically relevant light shapes.

For the second problem, I made the observation that the sets of points from where two edges appear to intersect are bounded by the planes that three of the four endpoints of the edges lie in. This became clear to me by examining the border cases where one endpoint of one edge appears to lie on the other edge. For that to happen, that endpoint, the other edge, and the observer must be coplanar (figure 5.3). This observation is relevant for

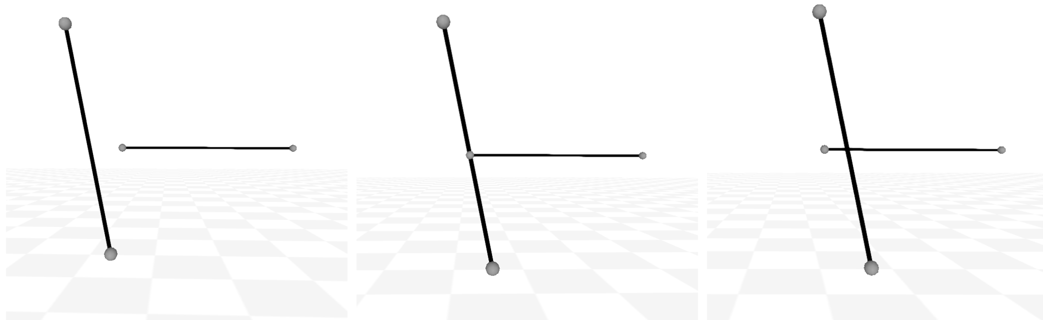

Figure 5.3: Two edges viewed from a point where they do not appear to intersect (left), one where they just begin to (middle) and one where they clearly do (right). Note that three endpoints and the observer are coplanar in the second image.

finding a valid set of partitioning planes because the precomputed data as described above captures which edges are contour edges and how these edges intersect. My first approach was therefore to use these four planes for each pair of edges as the partitioning planes. Interestingly, this would already include the planes that the faces of the polyhedron lie in and therefore be a superset of the planes I chose for convex polyhedra. This works but results in impractically large sets of planes ($4n^2$ if n is the number of edges). The first obvious optimization is to remove duplicates. Every endpoint of an edge is also the endpoint of at least two more edges, which results in each plane being added multiple times. The second optimization I made is to ignore pairs of edges that only appear to intersect if at least one of them is not a contour edge. Because I'm only interested in

intersections between contour edges, these pairs are irrelevant. However, this may also remove some of the planes that the faces of the polyhedron lie in so I add these again because they are still relevant for determining which edges are contour edges. With these two optimizations, the set becomes sufficiently small for many practically relevant light sources.

5.3 Building the Tree

Building the BSP-tree is not difficult in principle and not fundamentally different from building a BSP-tree for any other purpose. Nonetheless, a few details are specific to this problem and some choices need to be made about the overall structure of the algorithm. In this section, I explain my implementation, which consists of three steps: finding the partitioning planes, generating the overall structure of the tree, and calculating the data for each leaf node. The focus of this section is on the second step as the other two have already been discussed in the previous sections. However, I briefly go over the other two again to clarify the input and output of the second step. For the second step, I first explain how to find a valid tree and then how I refined the algorithm to find more balanced ones.

For convex light sources, the partitioning planes are simply the planes that the faces of the light source lie in. For non-convex ones, that set of planes is extended by four planes for certain pairs of edges to account for apparent intersections between contour edges. In both cases, the output of the first step is simply a list of planes.

Similarly, the third step requires the same input for both convex and non-convex light sources. All that is needed to precompute the data for each leaf is the polyhedron itself and one representative point inside each cell. I then execute the silhouette algorithm for each of those points and store the intermediate results in the corresponding leaves. No additional knowledge about the cells is required.

The second step is therefore well isolated from the other two and works the same way in both cases. I use a recursive algorithm that starts at the root and invokes itself for each child node that it generates unless that node is a leaf. In each invocation, I have to pick one of the partitioning planes to be used for the current node. The plane mainly has to fulfill the requirement that it actually splits the cell represented by the current node, meaning that the cell is not entirely on one side of the plane. If none of the planes fulfill this requirement, then the node must be a leaf node as it cannot be split further. Often several of the given planes fulfill the requirement. In principle, any of these is a valid choice but some lead to a more balanced tree than others. This is discussed in more detail towards the end of this section.

Using BSP for precomputed silhouettes differs slightly from using it as an acceleration structure for ray tracing or collision detection. My goal is not to partition a given set of primitives using any planes useful for that purpose. Rather, I have a given set of planes and need to subdivide the entire three-dimensional space as far as possible with these planes. This means that I have to explicitly pass the cell represented by the current node to the algorithm in some way, instead of passing a subset of the given primitives. This is necessary for two important tasks. First, it allows me to determine whether a plane splits the cell into two non-empty parts and therefore to pick the right planes and to figure out which nodes are leaves. Second, it allows me to calculate a point that is representative of the cell of each leaf and that I can therefore use to generate the silhouette data for that leaf.

I considered multiple ways of representing the current cell. One complication is that a cell may be unbounded. In particular, the root node represents the entire three-dimensional space. To simplify the algorithm, I restricted it to a large bounding box. This box can be

chosen arbitrarily large so that it contains the entire scene. With this restriction, every cell is a (bounded) convex polyhedron, which enables a variety of different representations. The one that I decided to use is to store the convex polyhedron as a list of edges. This keeps all necessary operations relatively simple. To test whether a plane splits the cell into two non-empty parts, it is sufficient to test if there are endpoints of edges on both sides of the plane. To get a representative point inside a cell, I simply calculate the arithmetic mean of the endpoints. To split a cell with a plane, I first split each edge and then create new edges where the faces of the polyhedron intersect the plane. I do this by sorting all intersections of edges with the plane in counter-clockwise order and then connecting neighbors in that sorted list with edges. This is possible because the cells are convex polyhedra and their intersection with the plane is therefore a convex polygon.

As mentioned before, the choice of the plane for each node affects how balanced the tree is. Balanced trees have shorter paths from the root to the leaves, which improves performance because fewer planes need to be tested. A major problem I encountered in trying to generate a balanced tree was that when I create a node, I do not have a useful metric for how balanced a cut is. One may be tempted to prefer planes that give an even cut with respect to the volume of the cell but since the leaf cells typically vary wildly in volume and larger ones are not necessarily more important, this is not a big improvement over picking planes randomly. I noticed however that because the scene is partitioned as far as possible with the given planes, the final cells are the same, no matter how the tree is built. This means that a good tree for the given planes is simply a tree that organizes these cells well, or, even simpler, one that organizes the representative points that I generate for each cell well. These points are not known when the tree is being built but only when it is completed. However, since they are the same for any tree built from these planes, I decided to simply build two trees. I use the first one only to get the representative points and then build the second one using these points to select the plane for each node. To get a well-balanced tree, I always select the plane that gives the evenest split of the remaining points. Generating the second tree is relatively simple. There is no longer any need for dealing with polyhedral cells, I simply partition a list of points using a set of planes.

5.4 Shader Implementation Details

Once the tree is built, it must be passed to the shader for the light source. The shader traverses the tree to find the cell of the shading point and then iterates over the precomputed data, using it directly for shading. My implementation avoids register spilling, which would occur if the silhouette was first stored entirely in GPU registers and then indexed dynamically. In this section, I explain in detail how I implemented my technique in a fragment shader. I also explain my storage formats, which I designed to keep memory usage and the amount of data read from buffers low.

I store the precomputed data in two buffers. The first buffer contains the structure of the tree as a list of nodes, the second one the precomputed silhouette data for each cell. I use the same format to store inner nodes and leaf nodes, although their data serves a slightly different purpose. Each node stores a plane and two indices. The indices are signed integers. Their sign indicates whether they refer to a position in the first or the second buffer. An inner node has one positive index for each child, a leaf node one negative index for the beginning of the precomputed data, and one for the end of it. In leaf nodes, the fields for storing the plane are unused.

For convex light sources, the precomputed data in the second buffer is an ordered list of points that describes the silhouette polygon. Each point is stored in 32 bits, either as an index into a list of the vertices of the light source or directly as three integers (two with eleven bits, one with ten) that are then scaled to the bounds of the light source.

The second option has lower but usually sufficient precision and requires only one lookup, rather than one for the index and one for the point.

For non-convex light sources, the data is an unordered list of silhouette edges stored as pairs of points. The reason why I do this differently than for convex ones is that the silhouette of a non-convex polyhedron can have holes and can therefore not always be described as a single edge loop. Each endpoint of an edge is either a vertex of the light source or the apparent intersection of two edges of the light source. I decided to limit my implementation to shapes with no more than 256 vertices. This enables me to store any such point in 32 bits just as for convex light sources. I store an apparent intersection of two edges as the four indices of the endpoints of the two edges. A vertex of the light source uses the same format but stores the same index four times. This makes the two cases easy to distinguish without any additional field.

I store all vertices of the light source as well as the partitioning planes in the local coordinate system of the light source. I traverse the tree in local coordinates and I apply an affine transformation to transform the vertices to world space. This allows me to move, rotate and scale light sources without recreating the buffers. This also means that multiple light sources can share the same buffers, which reduces overall memory usage significantly in scenes with many instances of the same light source.

The shader consists of two loops: one that traverses the tree and one that iterates over the precomputed data. The first loop fetches a node in each iteration and performs the plane test to find the next node. It stops when a leaf is reached. The second loop then iterates over the range given in the leaf node and immediately adds each edge to the integral. The full silhouette is never stored in registers and no register spilling occurs due to indexing parts of the silhouette. The only new data that persists over multiple iterations of the loop are the partial sums from equation 4.6.

6. Using Silhouettes for Solid Angle Sampling

As mentioned before, a major limitation of shading with LTCs is that it does not account for shadowing. As an alternative to LTCs that does support shadows, I use ray tracing with solid angle sampling based on a technique for solid angle sampling of triangles [Pet20] that is based on Arvo's technique [Arv95] but improves the numerical stability. To apply this to polyhedral lights, it is necessary to triangulate the solid angle of the light source. This could be done by triangulating the faces of the polyhedron but that has the disadvantage that it often produces more triangles than necessary as it also represents geometry inside the silhouette which is not relevant for shading and that some samples must be discarded as they are shadowed by the light source itself. Instead, I triangulate the silhouette of the light source, which results in a smaller number of triangles and takes care of self-occlusion without discarding samples. I found simple and efficient solutions for convex and star-shaped polyhedra. For other polyhedra, I use a similar preprocessing technique as for finding the silhouette. In doing so I encountered the problem that a triangulation is not necessarily valid for an entire BSP cell, and to find one that is, I had to resort to a randomized algorithm that is likely but not guaranteed to give a correct result.

6.1 Convex and Star-Shaped Polyhedra

Triangulating the silhouette of a convex polyhedron is trivial: I simply construct a triangle fan around the first vertex. Because the polyhedron is convex, any line segment connecting two of its vertices is entirely within the polyhedron, and its projection therefore entirely within the silhouette. Any arc connecting a vertex with any other vertex is therefore a valid edge for triangulation.

I use a very similar technique for non-convex polyhedra as well if they are *star-shaped*. A star-shaped set is one that contains a point that I will call its center, such that any point in the set can be connected to the center with a line segment that is entirely within the set. I triangulate the silhouette of a star-shaped polyhedron by constructing a triangle fan around the projection of the center. Similarly to convex polyhedra, this is possible because any line segment connecting a point of the polyhedron with the center is entirely within the polyhedron, and its projection therefore entirely within the silhouette. Any arc connecting a silhouette vertex with the projected center is therefore a valid edge for triangulation. If the center lies on the silhouette, degenerate triangles with solid angle zero are created. These triangles receive no samples.

Figure 6.1 shows examples of triangle fans for a convex and a star-shaped polyhedron. Many practically relevant light sources are covered by these two cases and constructing a triangle fan in the shader takes almost no time compared to sampling the triangles. I therefore recommend using this approach whenever possible, instead of the one described in section 6.2. Despite using precomputed triangulations, it does not perform better than using a triangle fan.

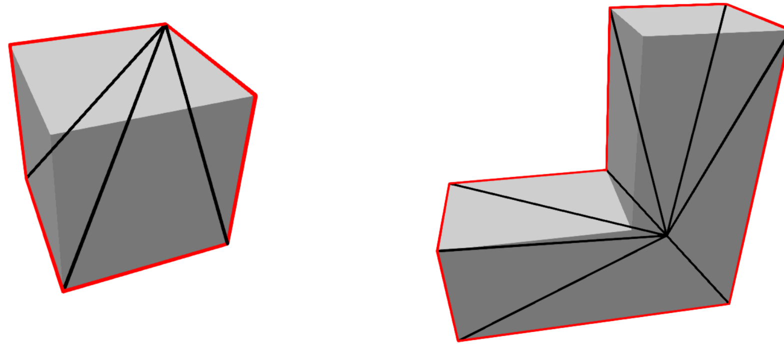

Figure 6.1: Triangulations of the silhouettes of a convex polyhedron (left) and a star-shaped polyhedron (right).

6.2 General Polyhedra

While triangulating silhouettes of convex and star-shaped polyhedra is relatively easy, solving the problem for any polyhedron proved to be quite difficult. Finding a triangulation directly in the shader did not seem promising to me so I decided to extend my BSP-based preprocessing technique so that I could also use it to precompute triangulations of silhouettes, instead of just the silhouettes themselves. In the first part of this section, I explain my algorithm for triangulating silhouettes in the preprocessing phase. However, not all silhouettes have a triangulation using just the vertices of the silhouette. In the second part of this section, I explain these cases and how I handle them by using additional vertices for the triangulation. In theory, this enables me to triangulate any silhouette but using precomputed triangulations further complicates the algorithm as a triangulation is not just required to be valid for a single shading point but an entire cell. I address this problem in the last part of this section with a randomized algorithm that tries multiple triangulations and tests them for a set of random samples.

6.2.1 Triangulation Algorithm

Since the silhouettes consist of relatively few edges and performance is not as relevant in the preprocessing phase as in the shader, I use a simple but slow triangulation algorithm. It should be noted that polyhedron silhouettes commonly have holes and can therefore not be assumed to be simple polygons. This prevents me from using some common triangulation techniques such as ear clipping. My algorithm represents the silhouette as an undirected graph and then simply tries to connect each pair of vertices by a diagonal. If the diagonal intersects an existing edge (other than at the endpoints), it is invalid and therefore skipped. By changing the order in which diagonals are tested and added, many different triangulations of the same polygon can be found. This will later be relevant for finding a triangulation for each cell of the BSP-tree. When all diagonals have been tried,

I assume that the polygon has been triangulated. Some silhouettes however cannot be triangulated this way because edges longer than π would be needed and I require edges to be the shortest connection of their endpoints, which is always shorter than π. I address this in section 6.2.2.

My next step is to find the triangles for the triangulation in the graph. I do this by first finding all cycles of length three and then eliminating unwanted ones. Clearly, each cycle of length three forms a spherical triangle but only some of them are part of the triangulation (figure 6.2). The algorithm may create edges outside the silhouette, for example, to connect the two vertices around a concave corner. This can result in triangles that are outside the silhouette and it can create triangles made of multiple smaller ones. These triangles should therefore not be added to the triangulation. I first reject all triangles that contain another vertex, so that only "atomic" triangles remain. I then perform a ray-polyhedron intersection test for each triangle to reject the triangles outside of the silhouette. If the ray from the shading point through the center of a triangle does not hit the polyhedron then the triangle is outside of the silhouette.

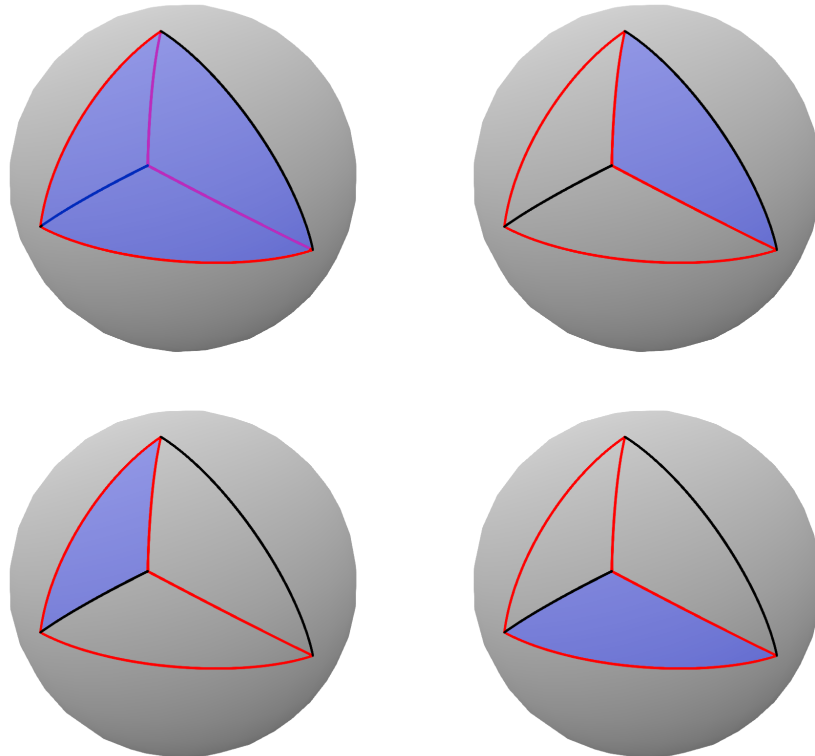

Figure 6.2: A spherical polygon (red), the diagonals added by the triangulation algorithm (black) and the four different spherical triangles (blue) that can be found in the graph. Only the last two are used for the triangulation. The first one is rejected because it contains another vertex, the second one is rejected because it is outside the silhouette, which is checked with a ray polyhedron intersection test.

6.2.2 Handling Cases Where No Triangulation Exists

Some spherical polygons do not have triangulations if only great circle arcs shorter than π are allowed. An example of this is shown in figure 6.3. Contrary to line segments in

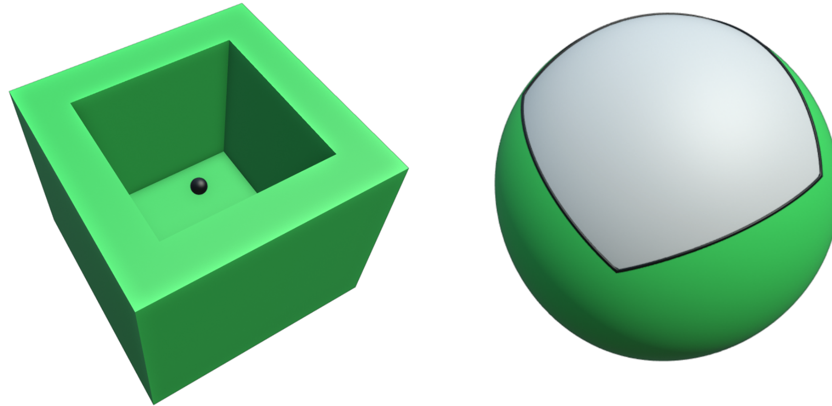

Figure 6.3: A polyhedron (green) and shading point (black) where the silhouette has no triangulation that uses only arcs shorter than π (left) and the silhouette projected onto the sphere (right). Note that the interior that needs to be triangulated (green) is the larger part of the sphere and that no diagonal shorter than π can be added.

the plane, there are always two great circle arcs connecting two points on the sphere and only one of them is the shortest connection. Spherical triangles are usually understood as projections of triangles onto the sphere, which means that each edge must be the shorter arc connecting the two vertices. Some spherical polygons however are such that the longer arc between two vertices is required to triangulate them. I solve this problem by adding additional vertices when necessary. These additional vertices allow bridging distances greater than π with multiple edges. A special case of this is the approach I previously described for star-shaped polyhedra where the center is also used in the triangulation. Only one additional point is necessary to triangulate any spherical polygon where each edge is shorter than π [O'R08]. However, since I need a triangulation to be valid for an entire BSP cell, I instead use the set of all vertices of the light source that are endpoints of concave edges. The projection of any edge of the polyhedron is always shorter than π so adding these vertices enables the algorithm to use the projections of the concave edges for the triangulation. If two vertices of the silhouette are further than π apart then they must be separated by concave edges since the projection of any convex set is shorter than π in all directions (if the shading point is outside of it). Therefore, no edge longer than π is necessary for the triangulation after these vertices have been added.

Allowing my algorithm to use additional points is easy: I simply add them to the graph and the algorithm will try to connect them to the other vertices. However, this means that the algorithm will always use them, whether they are needed or not, which can lead to a significantly higher triangle count than necessary. Instead, I pass two separate sets of points to the algorithm: required points that are used in the silhouette and optional points that only aid in finding a triangulation. I first try to triangulate the silhouette using just the required points. I then remove all optional points that are already inside of a triangle since they are not necessary to triangulate that part of the silhouette. I then add the new edges that are possible with the remaining optional points and add the new triangles created that way to the result.

6.2.3 Precomputation with Binary Space Partitioning

Because my triangulation algorithm for non-star-shaped polyhedra is not fast and simple enough to be used in the shader directly, I decided to move it to the preprocessing phase

where I already determine the silhouettes. The idea is very simple: instead of storing a set of edges as pairs of points for each cell, I store a set of triangles as sets of three points. Each point is stored exactly as before in one of the two possible formats described in section 5.4. This also works for the new optional points as described in section 6.2.2 since those are also just vertices of the light source. A triangulation that is valid for one point in a BSP cell is not necessarily valid for all other points in the cell. A triangle that is correct for one point may flip when moving to another point, resulting in a back-facing triangle that can be partially outside the silhouette and that can intersect other triangles (figure 6.4).

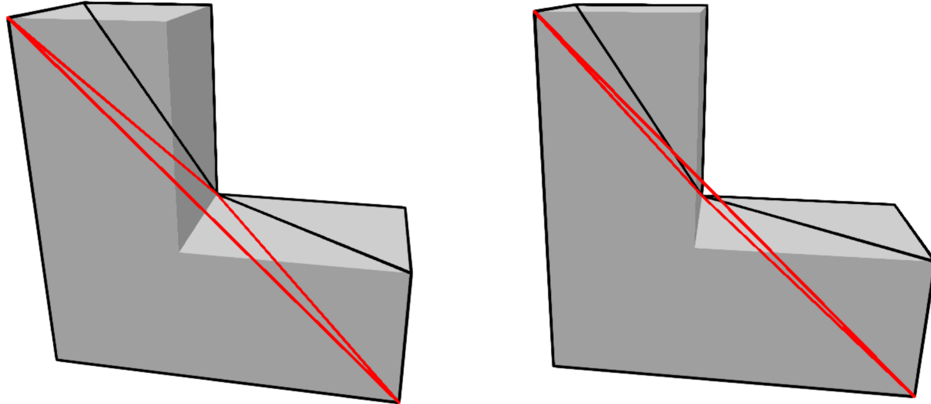

Figure 6.4: The same triangulation viewed from a point where it is correct (left) and from one where it is not (right). One triangle (red) flips, is partially outside the silhouette and intersects other triangles. The two points are in the same BSP cell.

While not every possible triangulation for a point is valid for the entire cell, some usually are (figure 6.5). My algorithm cannot just be used to find a single triangulation but also

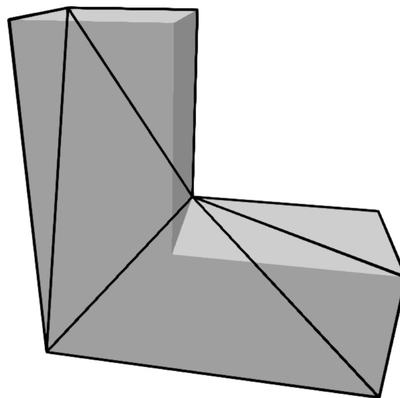

Figure 6.5: A different triangulation for the situation in figure 6.4 that does not have the same problem.

to find a large variety of triangulations for the same shading point and polyhedron by

changing the order in which new edges are tested and added. It is likely that one of them works for the entire cell as a triangle flipping like in figure 6.4 is a relatively rare occurrence. Testing whether a triangulation is valid for a given point is easy: I simply test if any of the triangles is back-facing when viewed from that point. However, testing whether it is correct for the entire cell is difficult. It may seem as if testing the triangulation for each vertex of the convex polyhedral cell is sufficient and that would certainly be true if the triangulation only used fixed points in space. However, a vertex of the silhouette can also be the apparent intersection of two edges, which changes position depending on the shading point. This means that even if a triangulation is valid for both endpoints of a line segment, it may still be invalid somewhere in between (figure 6.6). A triangulation

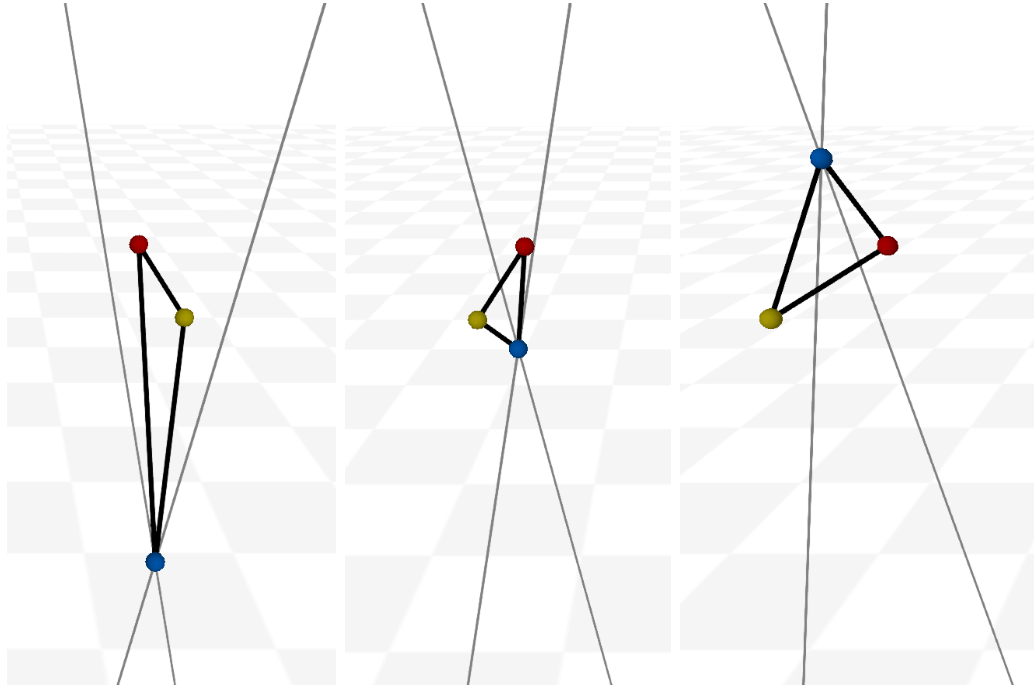

Figure 6.6: A triangle that flips twice as the observer moves in a straight line from left to right. The first (red) and second (yellow) vertex are fixed points in space while the third one (blue) is the apparent intersection of two lines (gray). From the first to the second picture, the triangle flipped as the first vertex has passed the second one in the projection because they are at different depth. From the second to the third picture, the triangle flipped again as the third vertex has moved to a much higher position because the two lines are at different depth.

is therefore not necessarily valid for an entire convex polyhedron just because it is valid for its vertices. It may still be the case for the cells that I actually use but even if it is, it seems very difficult to prove.

I am therefore stuck with only being able to verify a triangulation for individual points so I decided to use a randomized algorithm to test a triangulation for an entire cell: I take a large number of samples from the cell and test whether the triangulation is valid for each of them. This is not a guarantee that it is valid for the whole cell but with larger numbers of samples, it is increasingly unlikely that it is not. There are mainly two ways of generating these samples. Either they are generated for each cell as linear combinations of the vertices of the cell or they are generated globally and then assigned to the cell that they are in. I decided to use the first option because it avoids traversing the tree for each sample. Choosing the weights for the linear combination uniformly results in a large

number of samples near the center and very few samples near the boundary. This is a bad distribution for this purpose because triangulations commonly fail only near the boundary of the cell. As a very simple but effective solution to this, I use the third power of the uniformly generated weights to push samples closer to the boundary.

My complete algorithm consists of generating random triangulations for a point in the cell and testing the triangulation for random samples in the cell. I use the first triangulation that is valid for all samples. Note that testing a triangulation for a point is very cheap compared to finding a triangulation so using hundreds of samples is viable. Still, finding a method that guarantees that the triangulation is valid for the entire cell would be preferable and is an interesting problem for further research.

7. Results

In this section, I present the results that I achieved with the techniques described in this thesis. I show example renderings created with them and I give and explain my measurements for how well the techniques perform for various polyhedra. This includes measurements of run times of the shaders and of the preprocessing phase as well as the size of the precomputed data.

7.1 Rendered Results

The following renderings were created with the techniques I described to demonstrate what results can be achieved with them.

Figure 7.1 compares ray tracing with shading with LTCs in a simple scene, demonstrating the strengths and weaknesses of the two approaches. LTCs produce noise-free results quickly but they do not take shadowing into account. Ray tracing does produce correct shadows but it is computationally expensive and produces noisy images. My techniques improve the performance of both approaches so the best fit for each use case can be used or the two can even be combined [HHM18].

Figure 7.2 demonstrates correct shading of an object surrounded by a light source. While this is not unique to polyhedral light sources, most other commonly used types of light sources are convex (e.g. disks, rectangles, spheres, cylinders) so that situations like this cannot occur with a single light source. Handling them correctly with my techniques requires representing silhouettes as spherical polygons instead of simply projecting light sources into a plane with a standard perspective projection.

Figures 7.3 and 7.4 show more complete and realistic use cases with multiple light sources and other objects. Many objects are approximated well by polyhedra so having an efficient technique for using them as light sources gives artists a great deal of freedom in how they illuminate a scene. LTCs and ray tracing are already used for other types of light sources in popular game engines and my techniques are easy to integrate into existing workflows and renderers.

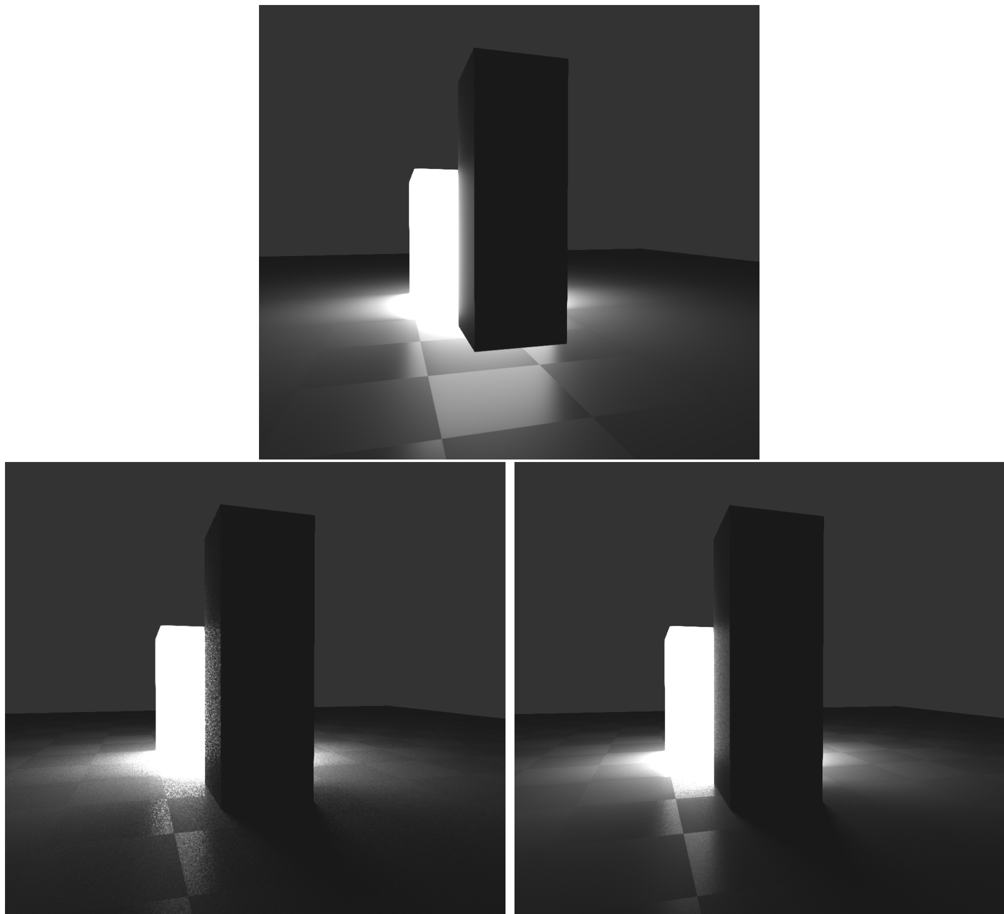

Figure 7.1: A cube light source partially occluded by another object rendered once with LTCs (top) and twice with ray tracing with solid angle sampling (bottom). The first image with ray tracing was made with 16 samples, the second one with 256. The version with ray tracing has shadows but is more expensive and produces a noisy result. Using more samples is slower but reduces the noise.

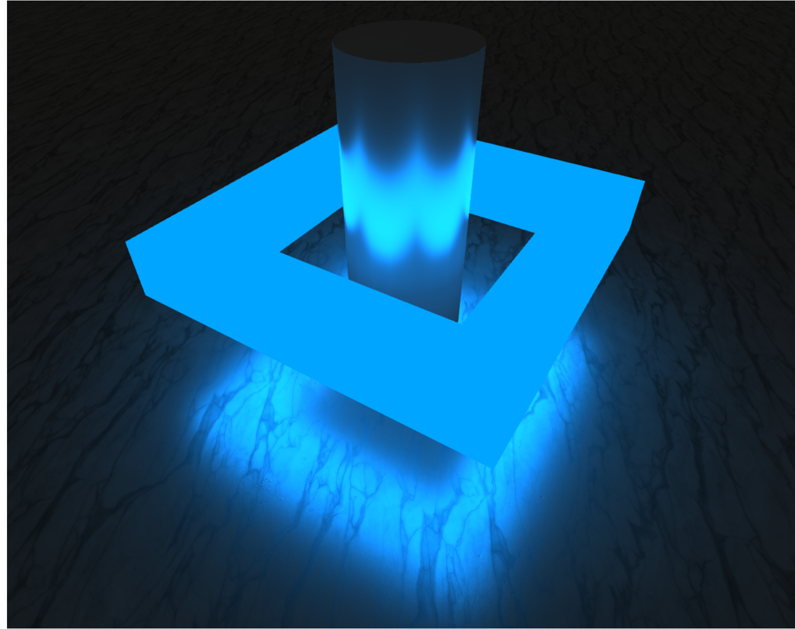

Figure 7.2: The reflection of a light source surrounding a cylinder. This demonstrates correct treatment of cases where the light source cannot be projected into a plane using a standard perspective projection.

Figure 7.3: A bedroom illuminated by three polyhedral light sources.

Figure 7.4: Neon letters reflected in the facade of a building. Each letter is a separate polyhedral light source.

7.2 Run Times

The techniques I present are meant and suitable for real-time applications. The most important property in this context is the time used for shading in each frame. The measurements I present in this section show that interactive framerates are achieved for a large range of light sources on modern graphics hardware. Another concern is the preprocessing time necessary for each unique polyhedron. This would typically happen before an application is shipped or at least only once when the scene is loaded so it is not as relevant as the processing time used for each frame. Nonetheless, short preprocessing times facilitate development and testing and expand the possible use cases of the techniques. My measurements show acceptable times for the tested polyhedra but faster preprocessing may be desirable for specific use cases where it affects the user experience.

I measured run times for a variety of different polyhedra, including convex, star-shaped, and non-star-shaped ones as listed in table 7.1 and shown in figure 7.5. I refer to them by the names in this table throughout this section. Of course, they are all polyhedra so "sphere" for example is just a sphere approximation, not an actual sphere.

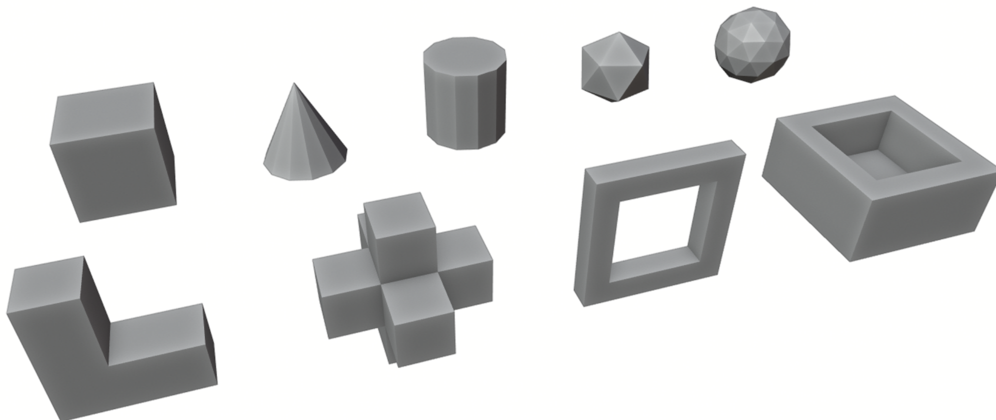

Figure 7.5: The polyhedra used for performance measurements. Top row: "cube", "cone", "cylinder", "icosahedron", "sphere". Bottom row: "corner", "extruded cube", "hole", "bowl".

7.2.1 Shading

I measured shading times in a deferred renderer so shading is largely independent of the complexity of the other objects in the scene. The scene I used for performance measurements consists of one light source at a time and some simple geometry covering the entire screen in the background. All measurements were performed on an Nvidia GTX 1070 TI at a resolution of 1024×768 pixels.

For the variant using LTCs, I compared three different approaches. The first method treats each face as a separate light source (although I still only used one draw call for the entire polyhedron), the second one determines the silhouette in the shader and the last one uses precomputed silhouettes. I did not implement the second one for non-convex light sources so I do not have results for those. Note also that using one light source per face produces less accurate results for non-convex light sources than using silhouettes as the latter does not handle self-occlusion of the light source properly (figure 4.1). I measured

Table 7.1: Names and relevant metrics of the example polyhedra shown in figure 7.5. The table lists vertex counts, edge counts, face counts and whether the polyhedra are convex or star-shaped.

Name	**Vertices**	**Edges**	**Faces**	**Convex**	**Star-shaped**
Cube	8	12	6	Yes	Yes
Cone	13	24	13	Yes	Yes
Cylinder	24	36	14	Yes	Yes
Icosahedron	12	30	20	Yes	Yes
Sphere	42	120	80	Yes	Yes
Corner	12	18	8	No	Yes
Extruded cube	32	60	30	No	Yes
Hole	16	24	10	No	No
Bowl	16	24	11	No	No

Table 7.2: Shading times for the example polyhedra using three different variants of shading with LTCs. The first method treats the faces of the polyhedron as individual light sources, the second one calculates the silhouette in the shader and the third one uses precomputed silhouettes.

Polyhedron	**Individual Faces**	**Silhouettes**	**Precomputed Silhouettes**
Cube	0.79 ms	0.38 ms	0.36 ms
Cone	1.49 ms	0.64 ms	0.51 ms
Cylinder	2.23 ms	0.89 ms	0.70 ms
Icosahedron	1.85 ms	0.67 ms	0.46 ms
Sphere	7.16 ms	2.45 ms	0.80 ms
Corner	1.18 ms	-	0.89 ms
Extruded cube	3.47 ms	-	1.93 ms
Hole	1.42 ms	-	1.00 ms
Bowl	1.53 ms	-	0.71 ms

only the time used for the deferred shading pass of the light source, not the time used for the entire frame. I calculated both specular and diffuse shading with one LTC for each of those (Lambert and GGX). The timings are listed in table 7.2.

The results show a significant performance gain from using silhouettes instead of one light source per face. Using precomputed silhouettes improves performance significantly for more complex objects but only slightly for simple ones. Notably, using silhouettes performs better even for non-convex polyhedra, despite being more accurate.

For the variant using ray tracing and solid angle sampling, I compared my techniques for triangulating the silhouette to using the triangulated polyhedron faces. I used one sample per pixel and I did not do any actual ray tracing because that would dominate the timings, especially on my hardware since it does not have dedicated ray-tracing cores. Instead, I simply used every sample without checking if it is occluded. Just like for the technique with LTCs, I only measured the deferred shading pass. The timings are listed in table 7.3.

The results show a noticeable performance improvement from using silhouettes but especially for simple light sources, the difference is smaller than with LTCs. This matches my expectations since the expensive calculations for solid angle sampling are performed per triangle, not per edge. The number of triangles is not much lower for simple shapes when using silhouettes. For light sources where many faces are entirely inside the silhouette,

Table 7.3: Shading times for the example polyhedra using Monte Carlo integration with solid angle sampling of the triangulated faces of the polyhedra and of their triangulated silhouettes. One sample per pixel was used.

Polyhedron	**Triangulated Faces**	**Triangulated Silhouettes**
Cube	0.89 ms	0.72 ms
Cone	1.34 ms	1.03 ms
Cylinder	2.66 ms	1.28 ms
Icosahedron	1.29 ms	0.86 ms
Sphere	5.03 ms	1.46 ms
Corner	1.44 ms	0.99 ms
Extruded Cube	3.71 ms	2.04 ms
Hole	2.04 ms	1.58 ms
Bowl	1.57 ms	0.84 ms

such as the sphere approximation, the difference is much more significant.

Overall the results demonstrate that the techniques I presented are indeed suitable for real-time applications and are in most cases a significant improvement over using the faces of the light source directly.

7.2.2 Precomputation

Using precomputed silhouettes has the potential to greatly reduce shading times but a relatively expensive preprocessing phase is necessary instead. Moving most calculations to the preprocessing phase has the advantage that they are only performed once per light source, when possible even at design time so that it does not affect the user experience. Preprocessing is commonly used in real-time computer graphics, for example for occlusion culling and indirect lighting so integrating even a long preprocessing phase into existing workflows should rarely be a problem. However, preprocessing time is still relevant as shorter ones enable faster development and enable a wider range of use cases. Some applications may benefit from creating new types of light sources dynamically, which imposes a more strict limit on acceptable preprocessing times.

I measured preprocessing times for the same polyhedra as the shading times. The calculations were performed single-threaded on an AMD Ryzen 5 3600 at a clock frequency of 3.59 GHz. I implemented the algorithms in C#. Table 7.4 lists the timings I measured for precalculating the BSP-tree and the silhouette for each leaf. For the triangulated silhouettes, I used 256 samples per cell to verify the triangulation.

The results show acceptable timings for precomputing the silhouettes of the tested polyhedra but they also show a rapid increase in preprocessing time as the complexity of the polyhedron increases. To some extent, this is an inherent problem of the technique. The number of leaves in the tree already increases rapidly and on top of that, each silhouette takes longer to determine for more complex polyhedra. However, my choice of algorithms also contributed to it. A faster method for finding silhouettes of non-convex polyhedra is likely possible, for example by modifying the algorithm described by Kettner and Welzl [KW97] to also apply it to spherical projections of polyhedra.

Triangulated silhouettes are significantly slower to precompute with my technique than just the silhouettes. This is mostly due to the relatively slow triangulation algorithm I use but also due to having to try multiple triangulations for some cells. Note that preprocessing is not necessary for star-shaped polyhedra, even though some are listed in the table.

Table 7.4: Number of elements and approximate total size of the precomputed buffers for the example polyhedra for the various types of preprocessing that I use. The total size was calculated assuming 20 bytes per node and four bytes per vertex.

Silhouettes of convex polyhedra				
Polyhedron	**Time**	**Nodes**	**Silhouette vertices**	**Total size**
Cube	0.0034 s	53	138	1.6 kB
Cone	0.025 s	418	1,838	15 kB
Cylinder	0.0016 s	436	2,674	19 kB
Icosahedron	0.027 s	1,828	6,245	62 kB
Sphere	3.7 s	147,660	947,142	6,700 kB
Silhouettes of non-convex polyhedra				
Polyhedron	**Time**	**Nodes**	**Silhouette edges**	**Total size**
Corner	0.023 s	332	1,165	16 kB
Extruded Cube	0.92 s	5,232	44,569	460 kB
Hole	0.022 s	414	1,215	18 kB
Bowl	0.010 s	198	407	7.2 kB
Triangulated Silhouettes				
Polyhedron	**Time**	**Nodes**	**Silhouette triangles**	**Total size**
Corner	0.15 s	332	853	17 kB
Extruded Cube	14 s	5,232	40,027	580 kB
Hole	0.23 s	414	1,022	21 kB
Bowl	0.073 s	198	271	7.2 kB

7.3 Size of Precomputed Data

To efficiently execute the shader, the precomputed data must be loaded in graphics memory. This means that keeping the data small is desirable to keep more memory free for other purposes like storing textures and models or for loading more light sources at the same time. I store the precomputed data in two buffers. The first one contains one entry for each node of the tree (including leaves in my implementation), the second one contains one entry for each vertex of a silhouette, for each edge of a silhouette, or for each triangle of a triangulated silhouette depending on which variation of the technique is used. The number of elements in these buffers and their total size for the example light sources is given in table 7.4.

This shows that for light sources of similar complexity to the tested ones, memory usage is usually not a problem on modern desktop graphics hardware, which typically has several gigabytes of memory available. However, just like the preprocessing time, the size of the data also grows rapidly with the complexity of the light source. I outline two possible approaches for reducing the size of precomputed data to allow for more complex light sources in section 8.

8. Conclusion and Future Work

To conclude this thesis, I reflect on the presented techniques and my findings in evaluating their usefulness for real-time rendering. I also explain the limitations of my techniques and outline my ideas for extending or improving them.

Polyhedral approximations of three-dimensional objects are commonly used in computer graphics due to their versatility, simplicity, and the many good tools for creating them. However, mainly for performance reasons, they are rarely used as light sources for real-time shading. Simpler primitives like spheres and rectangles are far more widespread but do not represent the large variety of light sources that exist in the real world well. The techniques I presented in this thesis are an important step for making polyhedral light sources more useful in real-time applications. Using the silhouette of a polyhedral light source for shading improves the performance of shading with LTCs as well as of solid angle sampling for ray tracing. Furthermore, it handles self-occlusion of the light source, which poses a problem for both techniques.

I presented multiple variations of my techniques. Based on my performance measurements, I recommend using precomputed silhouettes whenever possible as they provided shorter shading times in all cases. The variant without preprocessing for convex polyhedra has the advantage that the light source can be animated but this is very limited due to the requirement that the light source has to be convex at all times. Nonetheless, it may be useful in some specific use cases. For solid angle sampling, I recommend using triangle fans for convex and star-shaped polyhedra. Precomputing triangulations the way I do it only makes sense for non-star-shaped polyhedra where I did not find a simpler solution.

Aside from the algorithms for finding silhouettes, my implementation of area light shading with Monte Carlo integration is very simple and could be improved in various ways. I only described uniform solid angle sampling but even better results are achievable by combining my techniques for finding the silhouette of a polyhedron with BRDF importance sampling techniques for polygonal or triangular light sources. I also recommend using denoising techniques to achieve less noisy results with fewer samples.

A limitation of my techniques is that the size of the precomputed data grows rapidly with the complexity of the light source. While it is acceptable for the tested light sources, it is still concerning with regards to more complex light sources that may become interesting as graphics hardware continues to improve. It may however be possible to significantly reduce the size by storing each edge of the silhouette as far up in the tree structure as possible. The tree subdivides the scene as far as necessary for knowing the entire silhouette

of each cell. However, most silhouette edges are the same for entire subtrees, rather than being decided in the last inner node. These edges could already be stored at the root of the subtree that they are valid for. Modifying my technique to store silhouettes that way is an interesting problem for further research. The algorithm for building the tree needs to be modified to detect these cases and store the edges accordingly. To achieve the best compression, the tree may need to be restructured to give the largest possible subtrees for each edge.

A different approach to solve the same problem is splitting complex polyhedra into multiple smaller ones and generating one tree for each of them. To properly handle self-occlusion of the light source, this requires an efficient algorithm for determining the union of two spherical polygons in the fragment shader. This is almost certainly slower than using a single BSP-tree but it may still be considerably faster than finding the silhouette entirely in the shader and it would significantly reduce the total size of the precomputed data.

To use precomputed triangulations of silhouettes, I use a randomized algorithm that is not guaranteed to give a correct result. The main problem is testing whether a triangulation is valid for all shading points in a convex polyhedron. I do this by testing a large number of random samples but it would be preferable to find a deterministic algorithm for this. Alternatively, one could also try to solve the problem of how the BSP cells need to be subdivided further to guarantee that any triangulation that is valid for one point is valid for the entire cell.

Overall I believe my techniques to be already quite useful in practice in their current state but improving them as described above would allow them to be used with less care in an even larger range of applications.

Acknowledgements

I thank my advisor Christoph Peters for his valuable feedback and advice in writing this thesis. I thank Unity Technologies for providing the Unity engine that I based my implementation on. I also thank TurboSquid and its users RenderLuc7, CG Stuffs, and GothamNeedsMe for providing 3D models that I used for example renderings. Finally, I thank my family for their continuous support throughout my life and especially in my studies.

Bibliography

[App67] A. Appel, "The notion of quantitative invisibility and the machine rendering of solids," in *Proceedings of the 1967 22nd National Conference*, ser. ACM '67. Association for Computing Machinery, 1967, p. 387–393. [Online]. Available: https://doi.org/10.1145/800196.806007

[Arv95] J. Arvo, "Stratified sampling of spherical triangles," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. Association for Computing Machinery, 1995, p. 437–438. [Online]. Available: https://doi.org/10.1145/218380.218500

[BRW89] D. R. Baum, H. E. Rushmeier, and J. M. Winget, "Improving radiosity solutions through the use of analytically determined form-factors," *SIGGRAPH Comput. Graph.*, vol. 23, no. 3, p. 325–334, Jul. 1989. [Online]. Available: https://doi.org/10.1145/74334.74367

[DG09] J. Demouth and X. Goaoc, "Computing Direct Shadows Cast by Convex Polyhedra," in *25th European Workshop on Computational Geometry - EuroCG 2009*, Brussels, Belgium, Mar. 2009. [Online]. Available: https://hal.inria.fr/inria-00431544

[DGJ+20] S. Diolatzis, A. Gruson, W. Jakob, D. Nowrouzezahrai, and G. Drettakis, "Practical product path guiding using linearly transformed cosines," *Computer Graphics Forum*, vol. 39, no. 4, pp. 23–33, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14051

[FB09] J.-S. Franco and E. Boyer, "Efficient Polyhedral Modeling from Silhouettes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 3, pp. 414–427, Mar. 2009. [Online]. Available: https://hal.inria.fr/inria-00349103

[Gam16] M. N. Gamito, "Solid angle sampling of disk and cylinder lights," *Comput. Graph. Forum*, vol. 35, no. 4, p. 25–36, Jul. 2016.

[HDHN16] E. Heitz, J. Dupuy, S. Hill, and D. Neubelt, "Real-time polygonal-light shading with linearly transformed cosines," *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2897824.2925895

[HHM18] E. Heitz, S. Hill, and M. McGuire, "Combining analytic direct illumination and stochastic shadows," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '18. Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190834.3190852

[ICG86] D. S. Immel, M. F. Cohen, and D. P. Greenberg, "A radiosity method for non-diffuse environments," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, p. 133–142, Aug. 1986. [Online]. Available: https://doi.org/10.1145/15886.15901

[IFH+03] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte, "A developer's guide to silhouette algorithms for polygonal models," *IEEE Comput. Graph. Appl.*, vol. 23, no. 4, p. 28–37, Jul. 2003. [Online]. Available: https://doi.org/10.1109/MCG.2003.1210862

[KW97] L. Kettner and E. Welzl, *Contour Edge Analysis for Polyhedron Projections*. Springer Berlin Heidelberg, 1997, pp. 379–394. [Online]. Available: https://doi.org/10.1007/978-3-642-60607-6_25

[NM65] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 01 1965. [Online]. Available: https://doi.org/10.1093/comjnl/7.4.308

[O'R08] J. O'Rourke, "Computational geometry column 51," *SIGACT News*, vol. 39, no. 3, p. 58–62, Sep. 2008. [Online]. Available: https://doi.org/10.1145/1412700.1412714

[Pet20] C. Peters, "Brdf importance sampling for polygonal lights," *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 40, no. 4, 2020, to appear.

[PJH16] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation (3rd ed.)*, 3rd ed. Morgan Kaufmann Publishers Inc., Nov. 2016.

[SM09] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*, 3rd ed. A. K. Peters, Ltd., 2009.

[SWZ96] P. Shirley, C. Wang, and K. Zimmerman, "Monte carlo techniques for direct lighting calculations," *ACM Trans. Graph.*, vol. 15, no. 1, p. 1–36, Jan. 1996. [Online]. Available: https://doi.org/10.1145/226150.226151

[TS67] K. E. Torrance and E. M. Sparrow, "Theory for off-specular reflection from roughened surfaces∗," *J. Opt. Soc. Am.*, vol. 57, no. 9, pp. 1105–1114, Sep 1967. [Online]. Available: http://www.osapublishing.org/abstract.cfm?URI=josa-57-9-1105

[WLWF08] L. Wang, Z. Lin, W. Wang, and K. Fu, "One-shot approximate local shading," Tech. Rep., 2008.

[WMLT07] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, "Microfacet models for refraction through rough surfaces," in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, ser. EGSR'07. Eurographics Association, 2007, p. 195–206.

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den 23. März, 2021

(Bastian Urbach)