Integrating Dynamic Area Lights into the Real-Time Rendering Pipeline of the Godot Engine

Emil Dobetsberger

Supervisor: Dr. Sandro Spina

Co-Supervisor: Dr. Keith Bugeja

June 2025

Submitted in partial fulfilment of the requirements for the degree of Master of Science in Digital Games.

L-Università ta' Malta
Institute of Digital Games

L-Università ta' Malta

FACULTY/INSTITUTE/CENTRE/SCHOOL Institute of Digital Games

DECLARATIONS BY POSTGRADUATE STUDENTS

(a) Authenticity of Dissertation

I hereby declare that I am the legitimate author of this Dissertation and that it is my original work.

No portion of this work has been submitted in support of an application for another degree or qualification of this or any other university or institution of higher education.

I hold the University of Malta harmless against any third party claims with regard to copyright violation, breach of confidentiality, defamation and any other third party right infringement.

(b) Research Code of Practice and Ethics Review Procedures

I declare that I have abided by the University's Research Ethics Review Procedures. Research Ethics & Data Protection form code ICT-2025-00118.

As a Master's student, as per Regulation 77 of the General Regulations for University Postgraduate Awards 2021, I accept that should my dissertation be awarded a Grade A, it will be made publicly available on the University of Malta Institutional Repository.

Abstract

To render physically accurate light sources in 3D graphics applications, the surface area and shape of the lights must be taken into account; however, real-time rendering engines often implement only punctual lights. This Master's thesis evaluates the applicability of several algorithms for calculating rectangular area light shading and soft shadows in real-time rendering pipelines by integrating them into the free and open-source Godot game engine, with the aim of enhancing its realism and versatility. The results of this thesis not only provide an evaluation of various techniques, offering insights into integrating complex lighting models within open-source game engines, but also serve as a basis for a pull request to add an area light feature to the engine by providing an overview of the systems that the light has to interface with in the engine's rendering architecture.

We implement stochastic shading methods based on Monte Carlo sampling, most representative point sampling, and an analytic technique based on linearly transformed cosine distributions (LTC) and compare their accuracy and performance. We further describe a method to prevent artifacts in LTC shading and demonstrate that the required lookup tables can be approximated with polynomials at a negligible cost of quality and similar performance up to a certain number of lights. We then implement soft shadows through sampling points on the light to render shadow maps from and discuss an algorithm for adaptive light source subdivision using shadow reprojection, which we modify to be suitable for real-time rendering engines. We find that even though the technique can partially reduce the number of shadow maps when the penumbra occupies only a small area of the screen, its applicability is limited by its computational cost and its complex integration.

Acknowledgements

Writing this thesis would not have been possible without the abundant advice and feedback from my supervisors Sandro Spina and Keith Bugeja who took time to frequently meet me online or in person. I want to thank them for sharing their knowledge, always showing patience with my progress, and encouraging and inspiring me to write this thesis in the first place.

I would like to express my gratitude to all contributors of the Godot engine for putting this complex software together and allowing for accessible game development on hardware of all ends. Special thanks go to John Clay from the team of core contributors to the Godot engine, for giving me green light for this project and promptly answering any questions about the engine that occured to me.

An incredible help for writing this thesis was the abundance of free and open-source software I was able to use to conduct experiments, analyze the gathered data, format images, and compile this very thesis, and I want to thank all contributors, who worked together to conceive those programs: LaTeX, Blender, Python, SciPy, Numpy, matplotlib, OpenEXR, PIL, OpenCV, OpenImageIO, Git, Gimp, and TikZ.

I would also like to thank the creators of the scenes and models used for testing the algorithms in this thesis. A comprehensive list of credits can be found in Appendix B.

Finally, I want to express my respect to all computer graphics researchers laying the groundwork for modern real-time game engines, as writing this thesis has made me aware of the sheer amount of innovation that was necessary to conceive the incredible wonders of software that game engines and games are.

Contents

List of Figures

List of Tables

List of Abbreviations

BRDF Bidirectional reflectance distribution function.

GI Global illumination.

LTC Linearly transformed cosines.

LUT Lookup table.

MRP Most representative point.

PBR Physically based rendering.

PCF Percentage-closer filtering.

PCSS Percentage-closer soft shadows.

PDF Probability density function.

RID Resource identifier.

RMSE Root mean square error.

1 Introduction

Not only video games but also simulators, visualization programs (architecture, vehicles, e.g.), and films benefit from the constant advancement of real-time graphics as we continue to enhance the performance and realism of real-time rendering algorithms. A large part of any rendering program is concerned with how surfaces interact with light, and hence, the light and shadow computations largely define the quality and performance of a real-time graphics program. In the real world, lights come in many shapes and sizes, and so the greater the variety of lights that can be modeled in computer graphics, the more accurate the virtual representation of the world.

While in reality, any surface, such as a hot wire or a fluorescent substance, can emit light, in real-time graphics, we usually define light sources separately from the visual geometry or effect, and we use approximations to make light computations more efficient. Hereinafter, we will refer to a visual element that an observer expects to radiate light as the *diegetic* light source, borrowing the term from the field of cinematography and game studies, regardless of whether this element is generated through geometry, particle effects, textures, or other means of display. Whenever we do not explicitly refer to a light source as diegetic, we are referring to the *algorithmic* component used in shading and shadowing computations.

Computer graphics programs define different components to compute the effect of various types of diegetic light sources. Sunlight is typically approximated as light coming from a single uniform direction; hence, its virtual counterpart is usually called a *directional light* [1, 2]. Light sources for lanterns, torches, or lightbulbs are computed as if the source was an infinitely small point, hence called *point lights*. These point lights can be restricted in one direction, which is when we call them *spotlights*, or omnidirectional, which is when we call them *omni lights*. While illumination from these point light sources can be computed very efficiently with simple function evaluations, a much more complex task is to calculate the illumination for light sources that have some dimensionality, such as lines, disks, or rectangles, collectively referred to as *area lights*. This type of light enables us to more accurately complement diegetic light sources such as fluorescent tubes, softboxes, screens, billboards, or light panels.

While an ideal point light does not exist in the real world, point lights in computer graphics can accurately simulate illumination, given that its diegetic counterpart emits the light from a small enough volume. However, even as the diegetic light source slightly increases in size, rendering the scene with a point light will start to feel unrealistic. A more physically accurate area light rendering implementation should result in shading being softer, specular reflections appearing larger, less intense, and matching the shape of the source, and shadows having no hard edges. If all these effects are accounted for, the

realism of rendering certain scenes can improve greatly. This is shown by the example of a living room scene illuminated by a television screen in Figure 1.1.

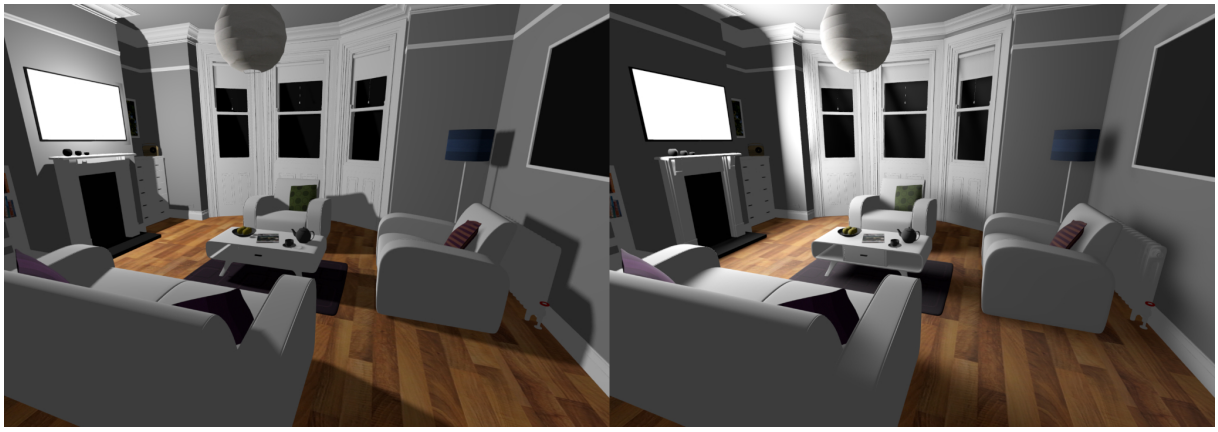


Figure 1.1 Comparison of illumination by an omnidirectional point light (left) and an area light (right)

Realistic area light simulation is solved by many offline render programs but continues to present a challenge in real-time applications. Game engines, especially so-called *general purpose* game engines [3], have particular constraints when it comes to the features they include or the computations they perform. In the case of lighting, this means that some algorithms are better suited for implementation and integration than others, as they might provide better runtime performance, quality, or other benefits. Several algorithms, which will be discussed in Chapter 3, exist that allow area lighting to be simulated, although research to make them more efficient or more accurate is ongoing. However, not every algorithm that is presented as novel and promising turns out to be feasible in practice, as performance or quality might decrease under certain circumstances, or as they might make requirements in hardware or software architecture that cannot be met at the current time. Another aspect that is often disregarded in research is how easy or hard a feature is to maintain in a rendering program that is actively being developed. Changes or feature additions to the shading model or shadow calculations might require adjusting the implementation of every light source, with area lights being an additional burden due to their complexity. Finally, the code for every feature that is used in a game needs to be bundled into or somehow distributed with the game's executable, so an added feature will not only increase the file size for all developers downloading the engine but also any players downloading a game made with the engine. If an algorithm heavily relies on lookup tables or adds a substantial amount of code, the impact on disk size should be taken into account.

This research thus puts into question which area light shading and shadowing algorithms are the most suitable for implementation into a game engine under active development in terms of performance, quality, integration effort, and maintainability. Apart from comparing a selection of them in terms of their computation time and realism,

this thesis will also examine what their integration entails and describe problems and improvements through the specific case study of integrating them into the Godot engine [4].

The engine was chosen in particular because it is free and open-source [4] and actively developed by its community; hence, its source code is easily accessible, and communication with the core maintainers is possible via online forums. Furthermore, despite multiple feature proposals having been made, at the time of writing, it yet includes neither an area light nor a ray-tracing implementation [5], which makes it an ideal candidate for the goal of this thesis. While we aim to offer broad insights into integrating complex lighting models within open-source game engines, a secondary objective of this thesis is to also provide a basis for a future pull request to the Godot engine to fulfill one of the aforementioned proposals of adding area lights, that is well-grounded in research and further enhances the visual capabilities and versatility of the engine.

In terms of shading, we will focus exclusively on *local illumination models*, where only light that bounces off a surface once after being emitted, and that can be computed in real-time, is taken into account in the rendering to narrow down the subject of research. *Global illumination (GI)*, where multiple light bounces are considered before light hits the camera, is a vast topic on its own and is left for future research. Although some definitions consider shadow rendering a part of GI [1], we refer to GI as systems calculating the illumination of light that has bounced off of other surfaces rather than the lack thereof.

Our research offers several contributions around the topic of area lights, which we identified as the following:

- We obtain comparative performance data for area light shading algorithms based on Monte Carlo sampling, most representative point sampling [6], and linearly transformed cosines [7] and compare them to a path-traced ground truth.
- An argument is made about choosing a bounding volume for a unidirectional rectangular area light that is limited by a range parameter, proving that a bounding box has a smaller volume than a bounding sphere, no matter the light's extents.
- We describe and apply the most representative point shading technique [6], offering some details about implementation and quality.
- A combination of the technique with Monte Carlo sampling is suggested, such that specular reflections have reduced noise.
- A fix for a precision issue is shown that appears in area light shading with linearly transformed cosines [7], which, to our knowledge, has not been discussed in any research as of yet.

- It is demonstrated that the lookup table used by area light shading with linearly transformed cosines [7] can be approximated by polynomials.
- Soft shadow rendering with many shadow maps and percentage-closer filtering [8] is demonstrated, and an argument is made about the optimal resolution and number of shadow maps.
- An argument is made about the effectiveness of adaptive light source sampling with shadow reprojection [9], supported by performance data.
- The thesis lays the groundwork for a pull request for integrating dynamic area lights into the Godot engine.

More theory and previous work will be covered in the following Chapters 2 and 3 before the methods for our research, as well as the selection criteria for implemented algorithms will be discussed in Chapter 4. In Chapter 5, we explain how we add area lights to the Godot engine and implement and compare a number of shading algorithms in terms of performance and quality, as well as discuss the encountered challenges and applied modifications in detail. In Chapter 6, we present an implementation for rendering soft shadows and discuss and compare various optimizations to it. A brief final overview of our findings regarding all algorithms is given in Chapter 7.

2 Background

The complexity of rendering engines warrants a chapter about some of the underlying concepts. We will briefly review the basic structure of a rendering pipeline and discuss shading models and the rendering equation, as well as their specific implementation in the Godot engine, to provide a foundation for the later sections on integrating particular area light implementations into the engine.

2.1 Rendering Pipelines

To render three-dimensional graphics, we define a virtual scene in terms of all its surfaces, a virtual camera from which the scene is viewed, a number of light sources that illuminate the scene, and visual properties of surfaces describing how they react to the light [1]. Surfaces are commonly defined as triangles composed of vertices, which can hold data such as position, normal, tangents, texture coordinates (UV), and vertex colors. In many rendering engines, the properties of the surfaces on a model are summarized in a data structure known as a *material*.

When positions or vectors in 3D space are described, they are always defined relative to some coordinate space. In this way, the vertices on a model are described relative to the model's origin in the so-called *model space*. The object's position in the scene is defined in *world space*. To generate an image, however, we need to also define the coordinates relative to where the virtual camera is with the system known as *view space*. To convert the three-dimensional scene into a two-dimensional image, we need to perform a projection of the points in the scene. The perspective or orthographic projections map points in view space to points in *homogeneous clip space*, where the space is normalized to the canonical view volume, which is a frustum for perspective projection, hereinafter referred to as *view frustum*, and a rectangular box for orthographic projection. With a graphics API such as OpenGL or Vulkan, the triangles in clip space are then rasterized into fragments, which are shaded according to the implementation of the fragment shader, resulting in the color values of the output pixels of the image. When discussing the shading calculation of a specific fragment, the point on the surface that the calculation is performed on is referred to as the *shading point*.

A rendering pipeline typically consists of several render passes. Although the passes can be defined as needed, one of the first passes usually includes drawing the depth buffer. Afterward, the opaque pass draws the majority of most scene geometry before transparent surfaces are drawn in their own pass, usually in back-to-front order. Additional render passes, such as drawing the background or sky, or post-processing effects like motion blur or fog, are also typically encountered in a complex rendering pipeline.

2.2 Rendering Equation

Lighting greatly contributes to the realism of a scene [1]. The interaction between lights, surfaces, and volumes is established in the *light transport model*. If only direct lighting is considered (emitted light that bounces from a surface directly onto the virtual camera), it is called a *local illumination model*. If, however, indirect lighting is taken into account, i.e., we try to model light that bounces more than once, a *global illumination model* is required. Many approaches to calculating global illumination models exist, some of which are designed to only model a specific visual phenomenon, like reflections or caustics, while techniques like ray tracing attempt to model a wide range of such phenomena.

The most notable local lighting models are the *Phong lighting model*, which divides the lighting into ambient, diffuse, and specular terms, and the *Blinn-Phong* model, which slightly alters the specular component to improve runtime. They can be generalized into the *bidirectional reflectance distribution function (BRDF)*, which is a function of the viewing direction, the shading point (normal, material properties), and incident light that returns the ratio of reflected radiance along that viewing direction on the unit sphere. Figure 2.1 shows a plot of an example BRDF, where the view angle and the properties of the shading point are fixed. The output color on a point on this unit sphere is the amount of light that is reflected when the light direction goes through that point.

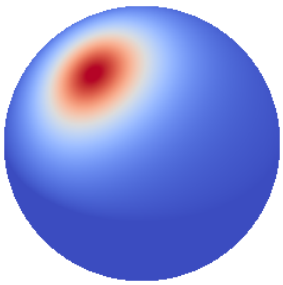


Figure 2.1 Example BRDF. The plot shows the ratio of reflected light at all incident light directions on the unit sphere under a fixed view direction that spans 30 degrees to the normal of a shading point. Generated with the BRDF fitting code by Heitz et al. [7].

For opaque and non-emissive surfaces, we can describe the total amount of outgoing light (light that is reflected) on a shading point in terms of the total incoming radiance (irradiance) from all directions of the hemisphere. The following Equation (2.1) is called the *rendering equation*, where we integrate over all directions h_i of incoming light, and where v is the view direction (from the shading point towards the eye), ρ is the BRDF of the shading point, $L_f(h_i)$ is the field radiance from some direction h_i, θ_i the angle of incidence of direction h_i, and $d\sigma_i$ the differential solid angle [10].

$$L(v) = \int_{\text{all } h_i} \rho(h_i, v) L_f(h_i) \cos\theta_i d\sigma_i. \tag{2.1}$$

Since integrating over the entire hemisphere is impractical, in a simple rendering engine, the rendering equation is implemented in a discretized form as a sum of the reflections of a finite number of incident light vectors, which typically correspond to the point lights in the scene. A rendering equation in this discretized form is shown in Equation (2.2), where k_i is the direction from the point light of index i to the shading point, and L_i the radiance of the light i.

$$L(v) = \sum_{\text{all } k_i} \rho(k_i, v) L_i \cos\theta_i. \tag{2.2}$$

If shadows are to be taken into consideration, the summand is multiplied by a visibility term that evaluates to 1 if the point from which the light is emitted is visible at the shading point, to 0 if it is blocked, and to a value between 0 and 1 for a half-shadow, also called *penumbra*. Although point lights should technically not create penumbras, they are often programmed to do so in computer graphics, as ideal point lights are an unrealistic approximation. To find out the visibility value at a specific point for a specific point on the light, shadow maps are commonly used, as described below in Section 2.4.

2.3 Shading Models

As mentioned above, there are several local lighting models that can compose the BRDF of a surface. In models like Phong or Blinn-Phong, the light computation is split up into three terms [1]. *Ambient light* is the first term and models an approximated amount of indirect light present everywhere in the scene. *Diffuse light*, the second term, accounts for light that is scattered at many angles, thus appears mostly equally bright regardless of the viewing angle, and makes up most of the light that bounces off rough, non-metallic surfaces. The third term describes the *specular light*, which models the highlights (appearing in the surface color for metallic materials and in the light color for non-metallic materials) that can be observed when viewing surfaces such that incident light reflects directly into the eye. The specular light is the only term dependent on the view direction; hence, when moving the camera around in a scene, the specular illumination changes, while the diffuse and ambient illumination does not. Contrary to the ambient component, which is calculated independently of any light sources in the scene, the specular and diffuse lighting must be computed for all area lights in a manner consistent with the light model used for point lights to yield plausible visuals. For this reason, we will cover the shading model used in this thesis in greater detail.

Modern computer graphics programs such as game engines often claim to support physically based rendering (PBR), which implies that their rendering algorithms aim for physical correctness, i.e., the rendering algorithms compute light in a way that accurately reflects how it would be in the real world [2]. Hence, the BRDFs we use in games should

be based on physical theory as well. As Phong-shading is just an empirically developed approximation from the computer graphics field, different lighting models have been suggested to account for various physical phenomena, typically based on research in the field of optics [11].

2.3.1 Specular lighting

Several models have been suggested to account for *microfacets*, which are microscopic sections of a surface. With microfacets, one can explain that even though there is an average normal direction of a shading point that we want to render, which can be described by a normal map, the normal direction might vary on a subpixel level, leading to less intense specular reflections [10, 12]. The variance of the microfacet normals is determined by a *roughness* parameter in most material systems. A BRDF that accounts for microfacets was suggested by Cook and Torrance [13]. This model is shown in Equation (2.3). There are four constants, α being the surface roughness, n the surface normal vector, f_0 the reflectance at normal incidence, and f_{90} the reflectance at grazing incidence. The vector h is the unit half vector between the eye vector v and light vector l, which are the function parameters. The Cook-Torrance model accounts not only for a *Fresnel* term $F(v, h, f_0, f_{90})$, describing how light is reflected specularly on smooth microfacets depending on the incident light and view angle, but also a geometrical attenuation factor $G(v, l, \alpha)$ to account for masking and shadowing which occurs as some microfacets occlude others. $D(h, \alpha)$ is the normal distribution function, describing which fraction of the facets are oriented in the direction of h and thus reflect light into the eye. Although the original equation puts a factor of π in the denominator, this has been corrected to a factor of 4 in later research [11, 12, 14].

$$f_r(l, v) = \frac{D(h, \alpha)G(v, l, \alpha)F(v, h, f_0, f_{90})}{4(n \cdot l)(n \cdot v)} \tag{2.3}$$

The Fresnel term F is often calculated with Schlick's approximation [15] to improve computation time [10, 12], which is shown in Equation (2.4):

$$F(v, h, f_0) = f_0 + (f_{90} - f_0)(1 - v \cdot h)^5 \tag{2.4}$$

While Cook and Torrance [13] also suggest formulas for the D and G terms, alternative terms have been recommended. They refer to Smith [16], who defined the masking function G as a product of two monodirectional shadowing terms $G(l, v, n) = G_1(l, n)G_1(v, n)$, although proposed models use a variety of implementations for G_1. A popular lighting model was proposed by Walter et al. [11], known as *GGX* (the cryptic term stands for "ground glass" as they used physical measurements of light reflecting from a glass sphere to develop their model), proposing terms for G_1 and D. Colloquially, the BRDFs in PBR rendering programs are often referred to as *GGX* if they are based on this specular model.

Although the initial proposal was more complex, requiring some expensive computations, Hammon [14] and Guy and Agopian [12] show how the entire Cook-Torrance specular model can be approximated by replacing G with a visibility function V_{GGX} that cancels the denominator of Equation (2.3). The function V is shown in Equation (2.5), for the derivation of which we point to Guy and Agopian [12].

$$V_{GGX}(l, v, \alpha) = \frac{0.5}{(1-\alpha)2(n \cdot l)(n \cdot v) + \alpha((n \cdot l) + (n \cdot v))} \tag{2.5}$$

Equation (2.6) presents the normal distribution function D in the GGX model, as described by Guy and Agopian [12].

$$D_{GGX}(h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)(\alpha^2 - 1) + 1)^2} \tag{2.6}$$

The entire approximated and optimized specular light model can then be calculated as in Equation (2.7).

$$f_r(l, v) = D_{GGX}(h, \alpha)V_{GGX}(v, l, \alpha)F(v, h, f_0, f_{90}) \tag{2.7}$$

2.3.2 Diffuse lighting

When it comes to diffuse shading models, the choice is usually between the Lambertian and Burley [12]. The Lambertian diffuse term is uniform regardless of the view direction and based on Lambert's cosine law [17].

The other diffuse shading model, described by Burley [18], does not assume that light is scattered uniformly, such as an ideal diffuse or Lambertian surface would. Despite yielding more accurate results than the Lambertian model, it is fortunately much less complex than the calculation of the specular lighting. We use the same function F as described above, pass the parameters $f_0 = 1$ and $f_{90} = 0.5 + 2\alpha(L \cdot H)^2$ and calculate the diffuse term f_d as shown in Equation (2.8).

$$f_d(v, l) = \frac{1}{\pi}F(n, l, 1, f_{90})F(n, v, 1, f_{90}) \tag{2.8}$$

Although the Godot engine supports several diffuse shading models at the time of writing, the area light implementations in this thesis will focus on the Lambertian and Burley diffuse BRDF and the specular GGX BRDF described above.

2.4 Shadow Calculation via Shadow Maps

Shadows occur when light is blocked on its way from the light source to a surface [1]. Mathematically, it can be modeled by adding a visibility function $V(k_i)$ to the summand of Equation (2.2), which calculates the light reflected on the shading point into viewing

direction v. While there are two prevalent ways to calculate this visibility function in real-time computer graphics, we will not cover *shadow volumes* [19] and instead dedicate this section to *shadow maps*, a technique first proposed by Williams [20], as it is the approach used in the Godot engine [21] and more predictable in terms of its run time [22]. Although real-time graphics programs usually allow turning off shadows where not necessary, in order to optimize performance, rendering shadows greatly improves perception of the shapes and spatial relationships of objects in a rendered scene [23]. We thus strive to enable them for our area light implementation as well.

A shadow map is a depth texture that is rendered from the point of view of the light source [1]. For omnidirectional lights, this involves rendering a spherical view, either as a cube map or as two paraboloids. The shadow map is rendered for each light source that is supposed to cast shadows before the opaque objects are shaded. In the opaque pass, we calculate the position of the current fragment on the shadow map using the light's transformation matrix and sample the depth on the shadow map at that position. We check if a fragment is in shadow by calculating its distance to the light source and comparing it to the depth value we just sampled from the shadow map. If the pre-opaque pass drew geometry at a lower depth than the current shading point on the shadow map at the sampled position, this geometry must occlude the shading point from the light's point of view and hence create a shadow.

Limitations of the shadow map technique in this form are that shadows can only be rendered for fully opaque objects (techniques to overcome this problem come with their own set of disadvantages [24, 25]), they are prone to aliasing artifacts, and are only physically accurate for point lights. We will discuss how to adjust this algorithm to render accurate shadows for area lights in Chapter 6. In the next sections, we will discuss the Godot engine and its rendering architecture.

2.5 Godot Engine

The advent of general purpose game engines gave rise to not only commercial products for general use like Unity [26], or Unreal Engine [27], but also to a number of open-source graphics and game engine frameworks, such as MonoGame, Ogre, or LibGDX. Born out of a commercial game engine, the Godot engine was released under the open-source MIT license by Argentinian developers Juan Linietsky and Ariel Manzur in 2014 [28]. Funded through donations and sponsors [4], the Godot engine has since then become one of the most popular open-source game engines according to the number of stars on the code-hosting platform GitHub [29] and one of the most popular game engines for independent developers according to the number of games published on the online marketplace itch.io at the time of writing [30].

The source code for the Godot engine is publicly available on GitHub [5] and has

been written by more than 2500 contributors at the time of writing. The engine is primarily built in C++, utilizing the build system SCons, and relies on a number of third-party frameworks. It provides different rendering pipelines, namely the *Compatibility* Renderer built on OpenGL, and the *Forward+* and *Mobile* renderers built on Vulkan. Shaders are mainly written in GLSL and compiled to SPIR-V for the Vulkan-based renderers. Internally, the engine constructs a directed acyclic render graph to carry out all render commands. In several systems, compiled code written in other languages is not stored as individual files but rather baked directly into the C++ source code files as a byte array or string. Similarly, not a single texture file can be found in the engine repository. It is also notable that all its UI is written in C++, and the engine is compiled into a single executable file, including the editor, the size of which is smaller than other commercial engines, likely due to all these design decisions. To keep it that way, discussions are frequently had on GitHub about whether the addition of a feature warrants the (often arguably small) added memory footprint.

Apart from a comprehensive 2D game development solution, the Godot engine, as of version 4.3, features 3D graphics systems, such as a physically based material system [21]. Its material system supports a microfacet BRDFs and implements several basic features like albedo, roughness, normal, emission, and ambient occlusion textures, and advanced effects such as transmission, clearcoat, rim lighting, anisotropy, subsurface scattering, and refraction, among others. The engine's lighting system allows for omni lights, spotlights, and directional lights and provides three global illumination systems based on baked lightmaps, voxels, and signed distance fields. Shadows are implemented exclusively by shadow maps, and for soft shadows, PCSS is currently the only available option. Screen-space effects, such as ambient occlusion or reflections, can be defined through the environment node, where glow, fog, and background sky can be added, too.

2.6 Godot Engine Rendering Architecture

A significant portion of this thesis is spent describing the integration of algorithms into the Godot engine, so it is sensible to briefly cover the architecture of the engine's rendering loop.

The main loop of the engine first calls an update function in each iteration to advance the game state, physics, audio, etc., before making its rendering server call any viewports to draw their images for display by rendering the active camera. After setting up the camera view frustum, an optimization known as *frustum culling* is performed. Culling excludes any visual instances entirely from the rendering process, given that their bounding volume is outside the view frustum. Contrary to the real world, lights in game engines usually have a range parameter, defining until what distance a light should be effective, such that lighting computations can ignore lights that are far away from the

shading point. For omni lights, this means that if a sphere around the light's position with the same radius as the light's range does not intersect the view frustum, it is culled. For spotlights, this is checked with an axis-aligned bounding box. To determine whether a volume should be culled, the most efficient method is to check if that volume is on the outside of each of the six planes that comprise the view frustum [1].

After the culling and subsequent setup processes required for shadows or global illumination, the actual rendering is initiated. There are dedicated render passes for shadows, opaque objects, transparent objects, post-processing, etc. While culling is always carried out regardless of the rendering method (also referred to as the *rendering pipeline* or *renderer*), the actual render passes are performed by the selected renderer. The Godot engine includes three rendering methods [21]:

- Forward+, the default, Vulkan-based renderer using a clustering optimization algorithm
- Forward Mobile, a Vulkan based-renderer that limits the number of lights that can illuminate the same mesh
- Compatibility, an OpenGL-based renderer for WebGL builds and low-end devices that do not support Vulkan

Although a complete integration of an area light feature would entail taking into account all three rendering pipelines, for the sake of simplicity, we will focus exclusively on the Forward+ renderer in this thesis, which is the default when opening the Desktop version of the engine. Integrating the feature into the other two renderers is left for future work. At the time of writing, the Godot engine does not include a deferred renderer. In the codebase, the default renderer is usually referred to as `ForwardClustered` due to clustering algorithm employed to optimize the number of light sources, decals, and reflection probes.

The clustering algorithm was suggested by Olsson et al. [31] and, as is the case for tiled shading, requires splitting up the screen space into tiles that occupy an $N \times N$ pixel area on the rendering output, as shown in Figure 2.2. However, in clustered shading, the view frustum is also split up in the z-direction, i.e., the depth domain, into a fixed number of volumes, 32 in the case of the Godot engine. Thus, one ends up with $N \times M \times K$ frustum-shaped volumes in view space, as shown in Figure 2.3, to which lights (and other clustered objects) are assigned. In the cluster structure, a light occupies only those clusters within its bounding volume, which depends on its range, and for a spotlight on its direction and angle. The cluster structure allows ignoring any lights that do not affect the cluster that the current shading point is part of, significantly increasing performance when a large number of lights is used, albeit at the cost of minor performance overhead.

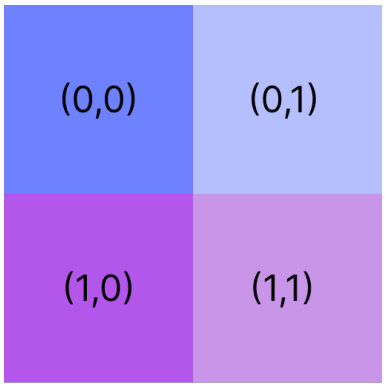


Figure 2.2 Subdivision of the screen into four tiles in tiled shading

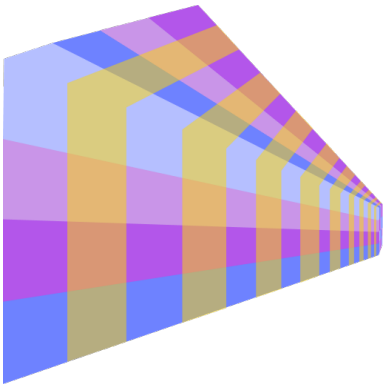


Figure 2.3 Subdivision of the view frustum into clusters in clustered shading

In the Godot engine, a cluster buffer is created for this purpose and passed as a uniform in any shaded draw calls. For each clustered element type (such as a spotlight), a bit mask is stored per tile (the set of clusters that have the same x and y position, i.e. lie on a straight line going into the view frustum), defining which elements affect the tile. Then, an integer is stored for each cluster that encodes the range of clustered elements that may affect the cluster. The layout of this buffer is shown in Figure 2.4 for subdivision of $2 \times 2 \times 32$. Note that in reality, the screen is subdivided into many more tiles, e.g., 60×34 for a Full HD resolution. The information in the buffer is then used in the shaders to decide whether a clustered element needs to be considered or not. The clusters are built with a compute shader that is run each frame before any shading passes.

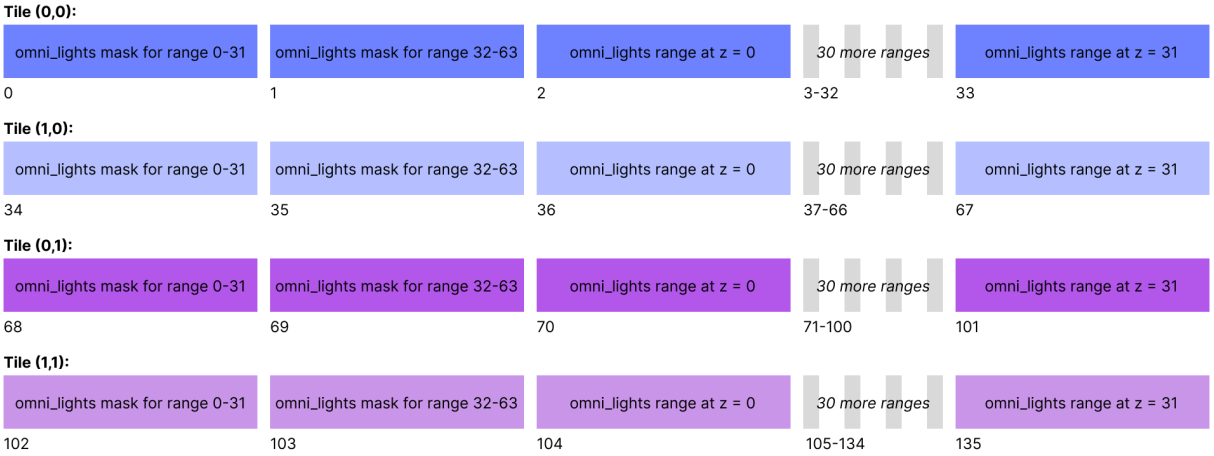


Figure 2.4 Layout of the cluster buffer for omni lights if the maximum number of clustered objects is 64 and the screen is divided into four tiles

The clusters are built in the *pre-opaque pass*. This pass also renders shadow maps. Shadows will be covered in greater detail in Chapter 6. Subsequently, the majority of objects in most scenes are drawn in the opaque pass. This process is done according to the specified materials assigned to the opaque objects by a suite of GLSL core shaders.

In the Godot engine, core shaders are split into several files using standard GLSL preprocessing and can define multiple stages within one file. At the same time, multiple *shader versions* are specified within one shader through the use of preprocessor branching. Depending on which stage in the rendering pipeline we are running the shader in (such as color, depth pass, global illumination, etc.) and which features are enabled on the material that is currently being drawn, a specific shader version is being used. The forward clustered scene shader acts as an uber shader for all standard materials in the Forward+ renderer. When the renderer uses it, the cull variant (backface, front face, disabled), primitive draw mode (point, line, triangle, line strip, triangle strip), and pipeline version are defined. Each material that the user defines injects some code in the Godot shader language into this core shader, specifying the features and parameters, or contains user-written shader code, and thus needs to compile all the required shader versions before it can be used. This implies that if we need to add another preprocessing branch that should only be active in certain render passes for a new feature, it would cause the number of shader versions to increase by a substantial amount, affecting load times and memory requirements, and should thus be thoroughly considered.

After all opaque objects were drawn in the opaque pass, the same is done for the transparent objects in their own pass before any post-processing is applied in subsequent passes. The result, which was rendered to a viewport texture, is then presented in the program window. To enable as much parallelization as possible on the GPU during rendering, a directed acyclic graph for all render commands is constructed, automatically reordering commands by topological sorting. This ensures that dependencies are satisfied (e.g. shadow maps are completed before shading the opaque objects), while maximizing parallelization [32]. Due to this optimization, the order of render commands on the GPU cannot be controlled directly.

2.7 Open Source Development

The Godot engine claims to be community-driven and is distributed as free and open-source software under the MIT license [5]. As is common for open-source software [33], it is maintained by a core team, but bug reports, ideas, and code are frequently contributed by people outside via issues and pull requests on the engine's GitHub repository [21]. Moreover, discussions are being held on community forums and channels, and even translations are provided by the large number of contributors to the software.

Developing software under open source comes with advantages like the increased number of people with a variety of skills working on the project, as well as disadvantages, such as organizational conflicts or differences of opinion on aspects like feature prioritization or development direction, which frequently arise [33]. Godot contributors manage most of their discussions on GitHub, but consensus is not always reached, even years after

an issue has been raised.

As described in the documentation of the Godot engine [21], pull requests are only accepted if certain conditions are met. Not only should features be finished and bug-free, but they also have to solve a frequent or complex enough problem and be dedicated to that problem, rather than trying to solve several problems at once. This is in order to not bloat the codebase with features that are used only by a very limited number of users or that are not needed at all but also to reduce the effort required to maintain the engine as a whole. For problems that can be solved in too many different ways and only affect a select number of users, the Godot engine provides a plugin API, allowing users to create and distribute modifications to the engine without needing to integrate their features into the engine's source code. In order to make a contribution to the engine, our feature should thus be useful to a large number of users, while being as minimal as possible yet as complete as necessary, as it also needs to integrate well into other systems of the engine. Maintenance efforts will inevitably slightly increase should the feature be merged.

When we evaluate the area light algorithms in subsequent chapters, we shall thus keep these guidelines in mind. Before making any recommendations, it is essential to be informed about any existing solutions described in literature, which will be summarized in the following chapter.

3 Literature Review

Area lights have been the subject of extensive research by computer graphics researchers in the past, as they pose numerous computational challenges, some of which we have yet to overcome. Previous investigations into shading objects illuminated by light sources with nonzero areas, as well as calculating the soft shadows they produce, are summarized in the following sections.

3.1 Area Light Shading

Area lights are far more challenging to implement than point lights, as for every shading point, we need to calculate the integral of the BRDF over the area of the light. The problem can be further complicated by the requirement that the color and intensity of the light be determined by a texture instead of a constant value across the area. In offline rendering, area lights are easily handled by a ray tracer, and accurate results are obtained via Monte Carlo integration, as demonstrated by Shirley et al. [34]. Ways to take the most efficient samples were extensively researched, such as by Ureña et al. [35]. Especially in the field of ray tracing, improvements to this technique have been discovered, such as by reusing information from adjacent pixels and previous frames in the *ReSTIR* algorithm [36].

However, achieving a performant approximation for real-time rendering remains a subject of ongoing research. One approach to approximate such a solution is to divide the area into virtual point lights. As regular sampling is either inefficient or inaccurate, Nichols and Wyman [37] suggested an optimization to select the virtual point light sampling depending on the continuity of the luminance of the texture.

Schwenk [38] puts forth a similar approach but employs hierarchical irradiance volumes to shade three-dimensional cells instead of fragments. They argue that this is faster and simpler than virtual point lights but struggle with specular reflections.

Another way to approximate a plausible area light is via the *most representative point* method, described by Picott [39], where among the infinite number of points on the area irradiance is calculated with a point that is believed to make the largest or most representative contribution to the integral on the shading point. This approach is used to represent linear lights and tube lights in several notable games and game engines [40–42].

Later, two different analytic approximations were found for the integral over a spherical polygon of an area light. Lecocq et al. [43] found a way using *irradiance tensors* together with an edge splitting strategy to evaluate the integral, but they do not support textures, and struggle with representing the reflections of discs at grazing angles. Around the same time, Heitz et al. [7] discovered a way to approximate area lights by transforming

the light's projected spherical rectangle over the BRDF to a different rectangle over the clamped cosine distribution, such that the value of the integrals are approximately equal. They calculate this through a sum of edge integrals, and just as Lecocq et al. [43] do, they also support arbitrary polygons. Their method, however, also includes a way to texture area lights, albeit one that does not accurately match their ray-traced ground truth for highly heterogeneous textures while still visually plausible for more homogeneous textures. They subsequently presented a technique to also include lines, disks, and spheres efficiently [44].

Luksch et al. [45] elaborate on a similar approach for rendering polygonal lights with photometric intensity profiles, however, do not include any performance tests in their research. Tao et al. [46] provide a way to render anti-aliased specular reflections by separating the surface roughness into different scales and representing them all by linearly transformed cosines, claiming to achieve similar or equal performance to Heitz et al. [7]. Kuge et al. [47] found a way to also calculate non-convex polygonal shapes and shapes with internal holes efficiently using the same linearly transformed cosines technique.

Other research has focused on the area of spherical harmonics, which we will not focus on in this thesis but will nevertheless include in the literature review for the sake of completeness.

Wang and Ramamoorthi [48] presented a closed-form solution to calculate the spherical harmonic coefficients for uniform polygonal area lights in pre-computed radiance transfer systems. Their solution enables area light sources with soft shadows and complex light transport effects, although they do not provide a way to use textured area lights, and in their tests, performance is impaired when using a high polygon count or several area lights. Wu et al. [49] significantly improved on this, developing a novel analytic formula for the spherical harmonic coefficients as they reduce the boundary integrals by differentiating the surface integral. Their implementation drastically improved performance over the algorithm of Wang and Ramamoorthi [48] by reducing the time complexity of rendering from linear to constant with respect to the number of area lights in the scene. While their solution is still not able to handle textured area lights, they suggest splitting a textured area light into a high number of uniform area lights to overcome this problem.

Mézières and Paulin [50] demonstrated in a highly technical paper that there is a simple one-way relationship between the computations of specular and diffuse contributions and describe a different way to calculate a spherical harmonic product between an arbitrary function and a clamped cosine, to replace the triple product allowing them to speed up spherical harmonics calculations by two orders of magnitude. Mézières et al. [51] found an analytic formula that efficiently computes the spherical harmonics (SH) gradients with uniform spherical lights. Liu et al. [52] present an algorithm for rendering the iridescence effect that certain materials exhibit under the special case of area lights.

3.2 Soft Shadows

One of the earlier techniques for rendering shadows in real-time computer graphics was suggested by Crow [19] and involves extruding geometry into shadow volumes. This technique was subsequently explored further, for example by Chin and Feiner [53], and also applied in games such as *Doom 3* [54]. In the early 2000s, efforts were made to enable the rendering of penumbras (half-shadows) with shadow volumes via the extrusion of *penumbra wedges*, primarily by Akenine-Möller and Assarsson. A comprehensive write-up of these efforts can be found in Assarsson [55]. These techniques, however, have not been widely applied.

One of the reasons that shadow volumes did not become the dominant technique was the more efficient approach proposed by Williams [20] of using shadow maps. To render shadows, the distance of the first point on any geometry seen by the light source in the direction from the light source to the shading point is looked up on the map, which is obtained by rendering a depth texture of the scene from the perspective of the light source. The precision of this method is mainly limited by the resolution of the rendered depth texture, which can produce strong aliasing if it is too small. This technique was later expanded to render soft shadows in different ways. Most notably, Fernando [56] proposed an efficient, though not physically accurate technique called *percentage-closer soft shadows (PCSS)*, which is widely used in games today. Similar algorithms were proposed by Atty et al. [57], although their algorithms do not account for self-occlusion, and Ali et al. [58].

Another approach to rendering soft anti-aliased shadows via a form of shadow maps was developed by Peters et al. [24]. *Moment shadow maps* store additional information that allows them to not only produce smooth soft shadows but also efficiently render shadows for translucent occluders, without the need for *deep shadow maps* as suggested by Lindbeck, Isak [25]. However, this type of soft shadow does not work for large area lights in a physically accurate way.

By subdividing an area light into *Virtual Point Lights*, Nichols et al. [59] generate soft shadows by marching rays toward each of those points.

A stochastic approach to generate physically accurate soft shadows that is simple to implement, yet computationally expensive, is to render a large number of shadow maps for the scene and use them to approximate the visibility function, as suggested by Heckbert and Herf [60]. An optimization to this technique is given by Schwärzler et al. [9], which will be a special focus of this thesis in the subsequent chapters. Further optimizations for large scenes were suggested by Zerari et al. [61], who added cascaded shadow maps.

An efficient method of rendering physically accurate area light shadows with a low number of samples was conceived by Heitz et al. [62]. Though requiring a ray-tracing implementation, it remains one of the most feasible algorithms for rendering both smooth

and accurate soft shadows. However, even in high-fidelity games of recent years, target hardware specifications and a requirement for a high amount of artistic freedom in terms of the number of lights in a scene can render this method unsuitable, favoring a single shadow map generated to best match the area of the light [42].

Finally, Okuno and Iwasaki [63] render soft shadows by using a *binary space partitioning visibility tree*, and although they claim to be both faster and more accurate than their reference implementations, their algorithm runs purely on the CPU and fails to achieve real-time framerates in their tests.

A study performed by Hecher et al. [23], examining various implementations of soft shadows, revealed that physically correct soft shadows tend to be preferred by both inexperienced and experienced users. Rendering efficient and physically correct soft shadows remains an active field of research, as a golden standard implementation has not yet been found.

4 Methodology

As we aim to give a recommendation of which algorithms are suitable for implementation into the Godot engine, we have to work within the constraints of its rendering pipeline. We acknowledge that analyzing certain algorithms will not be possible if they rely on frameworks that would require major alterations to the engine and are not feasible to implement within the scope of thesis. In this way, we will not evaluate any techniques based on ray tracing, which could be used for both shading and shadow computations, and we will also not investigate any algorithms that render shadow volumes since they are not supported in any major game engine that we are aware of.

To narrow down the scope of the research, we also limit ourselves to local illumination models and shadow rendering, discarding the analysis of any techniques for global illumination, like pre-computed radiance transfer. While an important field of research, we leave this part of the implementation open for future contributions to the Godot engine.

It is important to note that we are not seeking to provide a complete and exhaustive implementation that integrates with all subsystems of the engine. This would require extensive integration work for little academic output, potentially spiraling into an endless task as the engine is continuously being developed, making it difficult to stay on par with all features.

We separate the description of our implementation into two chapters, one for shading techniques and one for shadow algorithms. For area light shading, we choose to focus on the following techniques:

- Stochastic light sampling based on Monte Carlo integration. We calculate the illumination contributed by a number of points on the area light and compute an average for those points to obtain an estimate for the total radiance. This will be the first technique we analyze, as its implementation contains the least intricacies.
- Most representative point sampling, suggested by Picott [39] and further described by Drobot [6]. Similar to the first algorithm, but we attempt to sample exactly one point on the light source that best represents the average illumination of all points.
- Area light shading with linearly transformed cosines, presented by Heitz et al. [7]. As this is a more complex technique, we will focus on it last.

The shading algorithms and our modifications to them will be compared in terms of performance and quality. Our shadow implementations will be limited to shadow mapping due to the lack of a ray-tracing framework and the obsoleteness of shadow volumes. Nevertheless, we believe that performant solutions can be achieved within those constraints and that further research on shadow mapping can still produce valuable discoveries. We will focus our work on the following techniques:

- Stochastic soft shadows through sample points, proposed by Heckbert and Herf [60]. We sample a number of points on the light to render shadow maps from. Given enough sample points, the resulting soft shadows will be smooth.
- Percentage-closer filtering, proposed by Reeves et al. [8]. Combining the approach of rendering multiple shadow maps, we can blur them by sampling a greater number of points, leading to smoother shadows with fewer sample points.
- Fast accurate soft shadows with adaptive light source sampling, proposed by Schwärzler et al. [9], and expanded on by Zerari et al. [61]. By rendering just the shadows of a scene, we determine whether more or fewer sample points are required, ensuring soft shadows at any view distance.

We opt for a direct implementation of all algorithms into the target engine. Even though this involves extended compile times and encountering more bugs and dependency issues, it guarantees that our performance measures and any statements about compatibility will be indicative for a production version of the engine. We sometimes use the open-source editor SHADERed [64] to iterate more quickly on GPU code, yet all experiments will be conducted on the version we integrated into the Godot engine.

For measuring performance, we make use of the profiler built into the Godot engine. It provides the advantage of being able to inspect the performance of specific sections of GPU and CPU processes more precisely. We make a slight modification to it to allow for exporting the data and plotting it in the Python plotting library `matplotlib` afterward. Our performance tests are all carried out while rendering the scene at 1920×1080 using an NVIDIA RTX 3070 GPU and an AMD Ryzen 9 3900XT CPU.

Obtaining accurate quality measures for our algorithm will be more challenging. Since the engine provides no ray-tracing framework, we cannot simply evaluate a ground truth within the same program without going through the arduous task of integrating a ray-tracing framework. We thus make use of an external path-tracing engine at the expense of minor mismatches in the shading models of the two programs. This inevitably decreases the accuracy of our comparisons; however, any statements about the general tendencies of the quality metrics of our algorithms should not be affected. For the purpose of scene compatibility, we choose the path tracer of the 3D creation software Blender [65]. In order to export and compare images between the two, we use built-in functionality from the Godot engine to obtain the raw viewport texture as an `.exr` file, as well as export to the same file format from Blender. In this way, the color data is stored in scene linear color spaces without any loss due to color space transformation or tone mapping [66].

We utilize a variety of scenes to conduct our tests. All of the scenes are available online at the time of writing and are commonly used in computer graphics research. A comprehensive list, along with respective credits, is found in Appendix B.

5 Area Light Shading

To add an area light implementation to a rendering engine, we will first evaluate different shading algorithms and analyze shadow mapping techniques in a later section, as the manifestation of shadows presumes the existence of some illumination, in this case, emanating from a surface area. The question arises, which shape a surface that light is coming off from might have. Theoretically, any planar or non-planar surface could emit light, though game engines mostly deal with planar surfaces, and research about non-planar area lights is limited. Many shapes can be approximated with ellipses or a combination of triangles, so these shapes might offer the highest flexibility. However, we strive to fill a gap in the lighting system of the Godot engine with our implementation; hence, analyzing the existing lighting system helps us make an informed choice. While a triangle shape would be most helpful in modeling arbitrary emissive geometry, approximations of complex geometric shapes would require a high number of such triangular lights, impacting performance in a way that would only allow for static lighting in most cases, for which other lighting systems (GI systems) exist. Elliptical lights could be useful for dynamically modeling diegetic light sources that are large and round, such as a searchlight. However, the illumination, including specular reflections, caused by such a light source can be approximated well enough with a point light using the radius parameter that the Godot engine provides for such lights.

Figure 5.1 While the diffuse illumination is inconspicuous, the round specular reflection in the middle does not look like it is produced by the white quad in the back.

Specular reflections will noticeably appear incorrect when they manifest in a round shape, but are expected to be linear or rectangular, which is typically the case for diegetic light sources such as fluorescent tubes or electronic screens. An example of such a situation is shown in Figure 5.1. To provide a plug-and-play solution for these cases, we decide to implement an area light source that is rectangular, with the advantage that literature dealing with this case is more abundantly available than for disks or triangles.

Although we have established this focus, by no means does this mean that other shapes can not be added later on. Techniques to implement those shapes are mostly similar, and users could later switch to any of them by selecting a parameter on the area light node, should future work complete the feature.

5.1 Adding the Area Light Node

Since the Godot engine employs a node-based scene tree for its users to interact with, we first add a new node for our area lights, which we can do by extending a base class common for all light sources that provides basic parameters, such as the light energy or color. We also add new parameters specific to the area light, such as its extents (width and height), and make sure to bind them and register the light to the scripting API to make all parameters, including those that were inherited, accessible via scripts.

5.1.1 Integration in Rendering Server and Clustering

In order for our light's data to be forwarded to the rendering server, we need to account for it in the culling process, by checking whether a bounding sphere around the light lies outside of the view frustum. Since any point on the light can illuminate surfaces that are within the light's range, we set the center of the sphere to be at the light's center point and calculate the radius R_{cull} of the bounding sphere, taking the range r_{light} and adding half the diagonal, computed from the light's extents a and b, as shown in Equation (5.1).

$$R_{\text{cull}} = r_{\text{light}} + \frac{\sqrt{a^2 + b^2}}{2} \tag{5.1}$$

We adapt the light storage system to handle, for example, the area light's bounding box, which is used for occlusion culling and displayed in the editor viewport and this bounding box and other parameters to the light uniform, and bind it for rendering. Turning to shaders, we add a uniform buffer for our area lights and a loop that iterates over them, calculating their illumination. The code for setting up the uniform buffers is included in Appendix A.

The shaders handle direct lighting by looping over light sources, calculating their specular and diffuse illuminations, and summing them up. The loop is not done on all light sources in the uniform buffers but only on lights that are active in the cluster that the current shading point is part of. For an explanation of clustering, see Section 2.6. We expand the cluster buffer to account for area lights, considering their extents and range.

When setting the bounding volume of an area light, the question arises whether its volume of influence is better approximated by a sphere or by a box, the shapes commonly used due to their fast intersection computations. Although one might expect this to depend on the light's range and extents, we can prove that it is always better approximated

by a bounding box than by a bounding sphere. We first assume the light has an infinitely small area, reducing its volume of influence to a hemisphere. In this case, its volume is exactly $\frac{2}{3}\pi r^3$, with r being the range of the light. The bounding box of this hemisphere has a volume of

$$V_B = (2r)^2 r = 4r^3.$$

The volume of the bounding sphere would be

$$V_S = \frac{4\pi}{3} r^3.$$

Thus,

$$V_S = \frac{\pi}{3} V_B,$$

and as $\frac{\pi}{3} > 1$, we have $V_S > V_B$, for any $r > 0$. If the light does have some width and height $w, h > 0$, spanning up a diagonal $d = \sqrt{w^2 + h^2}$ the volume of the bounding sphere becomes

$$V_S = \frac{4\pi}{3}\left(r + \frac{d}{2}\right)^3 = \frac{4\pi}{3}\left(r^3 + \frac{3}{2}r^2 d + \frac{3}{4}r d^2 + \frac{d^3}{8}\right).$$

The volume of the bounding box would be

$$V_B = (2r + w)(2r + h)r.$$

Since $d > w$ and $d > h$, we know

$$\begin{aligned} V_B &< (2r + d)^2 r = 4r^3 + 4r^2 d + r d^2 \\ &< \frac{4\pi}{3}\left(r^3 + r^2 d + \frac{r d^2}{4}\right) \\ &< \frac{4\pi}{3}\left(r^3 + \frac{3}{2}r^2 d + \frac{3}{4}r d^2 + \frac{d^3}{8}\right) = V_S. \quad \square \end{aligned}$$

Hence, the bounding box occupies less volume than the bounding sphere, fitting around the true volume of influence more tightly. Two intriguing details about these insights are, for one, that the inequality would also hold for uni-directional disk-shaped light sources, which we can prove by setting d to be the diameter of the disk. For another, we postulate that the situation would be more ambiguous if the box had to be axis-aligned, as such a box might then encompass volume behind the light source as well, were the light to be rotated around its x-axis.

Due to the inequality discussed above, we decide to implement the bounding volume for our area light node in the clustering algorithm as a box, setting the correct transformation values and binding respective vertex and index arrays to the cluster-building

rendering commands. In the scene shader, where we find a loop for each clustered element, we add a new iteration for all the area lights that are active in the cluster that the current shading point is part of and preliminarily call a method that calculates the shading of an omni light.

5.1.2 Icons and Gizmos

The newly added area light source now produces the same illumination as a point light. While we will focus on an actual area light shading algorithm later on, we want to first add a way to depict the light in the user interface. To do so, we need to add representative icons, taking inspiration from other game engines and using the color scheme used for all icons in the Godot engine. These icons are shown in Figure 5.2.

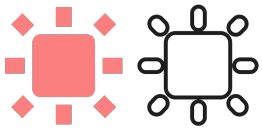


Figure 5.2 Icons for the newly added area light. Left: icon for the area light node in the database. Right: icon for display in the 3D viewport.

Apart from setting the extents of an area light node through the editor's inspector, users shall also be able to directly use a visual representation of the light in the 3D viewport, showing the light's outlines and providing two clickable manipulation points (*gizmos*) with which the extents can be changed. Similar gizmos are present for changing an omni light's range and a spotlight's range and aperture with similar outlines. The gizmos are illustrated in Figure 5.3, and the code to calculate them is provided in Appendix A.

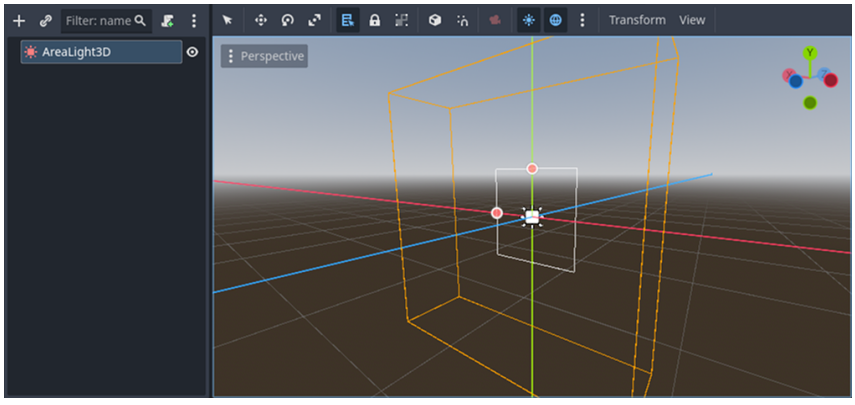


Figure 5.3 Outline and handles of the new `AreaLight3D` node

Once we have added the area light node, we need to implement an appropriate shading algorithm. To compute the irradiance at any point in the scene, we need to compute an integral over the area of the light. Although a precise closed-form calculation

is not yet known, researchers have come up with a number of algorithms that differ in quality and computational complexity for computing approximate solutions to this integral, some of which we will implement in the following sections.

5.2 Stochastic Monte Carlo Sampling

One way to calculate an approximation for the integral over an area light's surface is by Monte Carlo integration. In particular, we base our implementation of the Monte Carlo sampling algorithm on the implementation in the physically based rendering engine described by Pharr et al. [2]. Although their rendering engine relies exclusively on ray tracing, and we require a rasterization approach in the Godot engine, we can apply the same technique for sampling points on the area light to compute the shading.

The irradiance for the sampled points can then be calculated as the sum of the calculated contributions of the sampled points divided by their respective probability density (samples with a lower probability to be chosen contribute more to the result than samples with a higher probability), divided by the total number of points, and multiplied by the solid angle of the light [2]. Formally, let Ω_L be the solid angle of our area light, $f(x)$ the BRDF of the shading point with respect to some incident light vector x, and $p(x)$ the probability density function (PDF) of sampling a light vector x. Let Ω be the domain of the unit hemisphere. Through the Monte Carlo estimator, we know that the expected value E of the calculation described above is equal to the integral:

$$E[F_n] = E\left[\frac{\Omega_L}{n}\sum_{i=1}^{n}\frac{f(X_i)}{p(X_i)}\right] = \int_{\Omega} f(\omega)d\omega$$

For the BRDF $f(x)$, we use the same function that is defined in the renderer of the Godot engine for point lights, with the difference that for its input, instead of using the vector from the point light to the vertex position, we use the vector from the current sample on the area light to the vertex.

The algorithm we use for sampling points on a spherical rectangle is described by Ureña et al. [35]. As expected, the more samples we take, the better the result we obtain. For rough surfaces with high-frequency textures, a low number of samples can already lead to satisfying results, as they primarily reflect light diffusely. While a higher number of samples can improve the results for rough surfaces with flat textures, we observe that the same does not apply to glossy surfaces with this implementation. If we decrease the roughness of the surface from a value of 0.5, first, the noise in the specular reflection seems to increase; then, as the value approaches 0, the specular reflection gets darker until it disappears entirely. This is because the glossier a surface is, the smaller the cone of directions from within which a light contributes to the specular reflections [6]. Thus, a

small section of the area light is responsible for a large portion of the specular reflection at a particular shading point. For any samples we take from outside that section, the BRDF will produce low values in the specular term.

It is possible to mitigate this problem by splitting up the BRDF calculation by using a different set of samples for the diffuse and the specular term. We base our approach for taking the specular samples on Drobot [6]. Instead of taking the midpoint of the region that the author describes to sample the most representative point, we take a number of samples out of said region, again using the technique for sampling points uniformly on a spherical rectangle mentioned earlier. The results are shown in Figures 5.4 and 5.5.

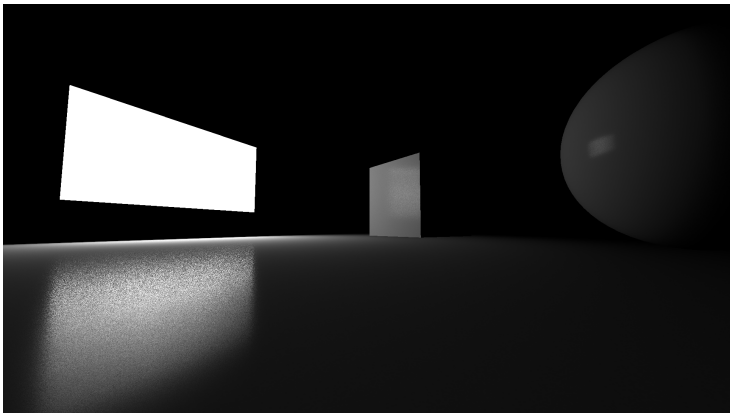


Figure 5.4 Shading with stochastic Monte Carlo sampling with 128 samples per pixel

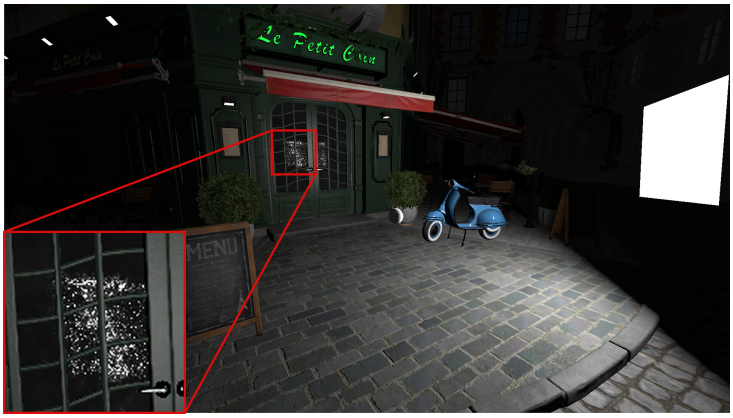


Figure 5.5 Shading with stochastic Monte Carlo sampling in Bistro scene with 32 samples per pixel. At diffuse surfaces, the lighting appears smooth, while considerable noise is discernible on the glass door.

We implement our random number generation, focusing on temporal stability. By taking the screen coordinate for specular samples as the seed, the specular reflections are stable until the camera moves. By taking the world coordinates quantized to a small length as a seed for the diffuse samples, the diffuse lighting of a point in the scene is stable even when the camera moves, given that the quantization step is large enough to account for the space that a pixel occupies in the scene.

5.3 Most Representative Point Shading

The idea of the most representative point (MRP) approximation of the area light integral is that specific regions within the area contribute to a significant part of the illumination, and hence, finding a point in the area that best represents the integral might allow us to calculate a good enough approximation [6]. With this technique, a different point will be chosen for each shaded fragment.

The advantage of this method is that the BRDF for only one light direction needs to be calculated per fragment, while the results are expected to be superior to those of single-sample Monte Carlo integration. Due to these advantages, it appears to be one of the more popular area light shading algorithms in production environments. It has been implemented in Epic Games' Unreal Engine [40], EA's Frostbite engine [41], Wicked Engine [67], and CD Projekt's RedEngine in *Cyberpunk 2077* [42].

We base our implementation on the technique described by Drobot [6]. The author divides the sampling into two steps. In the first step, the diffuse sample point is calculated and used for a diffuse shading calculation. In the second step, the specular sample point is found and used for a specular GGX calculation. To find the diffuse sample point p_d, the author clips the area light above the horizon of the shading point p, then finds the point p_r on this area that is the closest to the shading point. Then they find another point p_c on the area light, such that the vector $p\vec{p}_c$ has the smallest possible angle to the shading normal. This is seen in Figure 5.6. Although this technique appears simple to comprehend, a large series of ray-plane intersections are required, and especially the clipping operations are not trivial to perform. Notably, Drobot [6] also does not provide a clear explanation for how to obtain the point p_c if the light normal n_l is pointing away from the shading point ($n \cdot c_l > 0$), and we have not found an analytic way to do so either. Hence, we opted for implementing the approximation described by the author, which involves calculating the halfway vector between the closest point p_p on the light plane to the shading point p and the point p' on the light plane with the least angle to the surface normal of the shading point. In the aforementioned case, where the light normal points away from the shading point, the author recommends tilting the intersection line toward the light plane to obtain an intersection, yet does not describe a concrete method to do so. As a workaround, we suppose that by incrementally tilting the normal vector toward the light plane, the first intersection we obtain would be at an infinite distance, as the intersection line would be infinitely close to being parallel to the light plane. Instead of calculating the actual infinitely distant point p', we thus obtain a vector $\vec{p}''$ parallel to the light plane, with Equation (5.2), where $\vec{n}$ is the shading point normal and $\vec{n_l}$ is the light normal.

$$\vec{p}'' = (\vec{n_l} \times \vec{n}) \times \vec{n_l} \tag{5.2}$$

Furthermore, instead of the approximate halfway vector described in the source material, we calculate the angle bisector with Equation (5.3) from the normalized vectors $\hat{p''}$ and $\hat{pp}_p$.

$$\hat{p_d}' = \frac{\hat{p''} + \hat{pp}_p}{||\hat{p''} + \hat{pp}_p||} \tag{5.3}$$

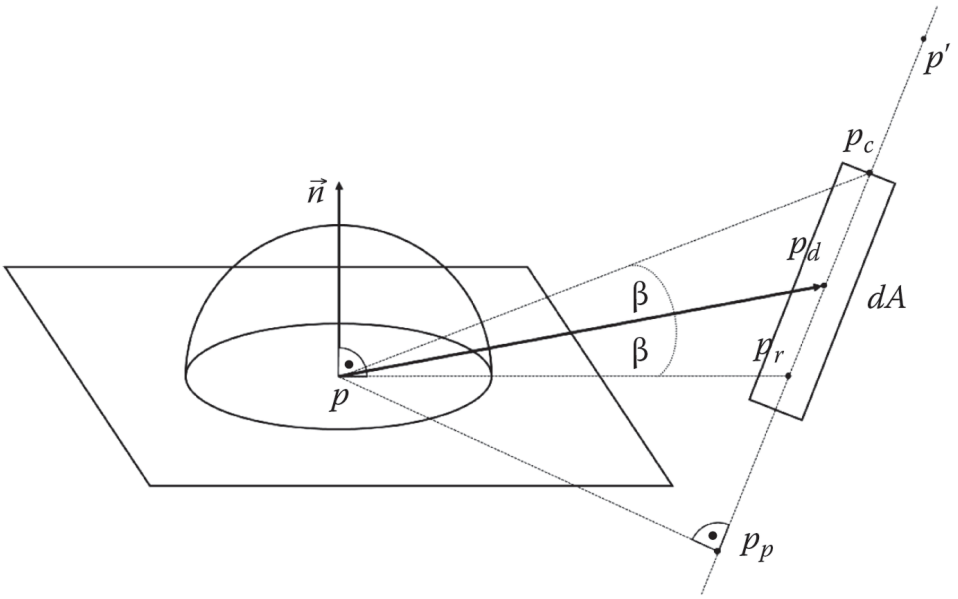


Figure 5.6 Calculation of the diffuse sample point p_d as the point halfway between the closest point p_r and the steepest point p_c. Image source: [6]

Drobot describes the implementation for Lambertian diffuse shading only. We instead use the light vector $\hat{p_d}'$ in the Burley shading computation to match the diffuse lighting used for other algorithms and other light sources.

Obtaining the specular sampling point p_{sc}, as described by Drobot [6], involves spanning a cone around the reflected eye vector $\vec{r}$ with a cone angle α, which is calculated based on the surface roughness. The author provides the formula in Equation (5.4) for this purpose, where g is the surface glossiness.

$$\alpha(g) = 2 * \sqrt{\frac{2}{g+2}} \tag{5.4}$$

In our tests, this function, adjusted for using a roughness parameter r instead of glossiness, resulted in specular reflections being too small for low roughness values due to the size of the cone. Thus, we empirically fitted the function in Equation (5.5).

$$\alpha(r) = \frac{\pi\sqrt{r}}{4} + 0.1 \tag{5.5}$$

Then, the cone C is intersected with the light plane, and a point p_{sc} is returned that is located at the center of the intersection between the light area and the circle at

the base of the cone. This is shown in Figure 5.7. With the obtained sample point, a specular reflection value is then calculated with the usual GGX function. Drobot [6] then also suggests normalization by the area A_S of the intersected ellipsoid. In our tests, this sampling strategy resulted in irregular specular reflection shapes; hence, we decided to change the sampling point to be the intersection point of the reflection vector with the light plane clamped to the light area. Empirically, we found that instead of the normalization, multiplying by the ratio of the solid angle of the ellipsoid divided by the solid angle of the total rectangle improved results. While extensive time was spent optimizing the cone angle function and trying different combinations of multiplications and normalizations by area and solid angle, a configuration that matches the ground truth in all scenarios was not found, although an error in our implementation cannot be entirely ruled out.

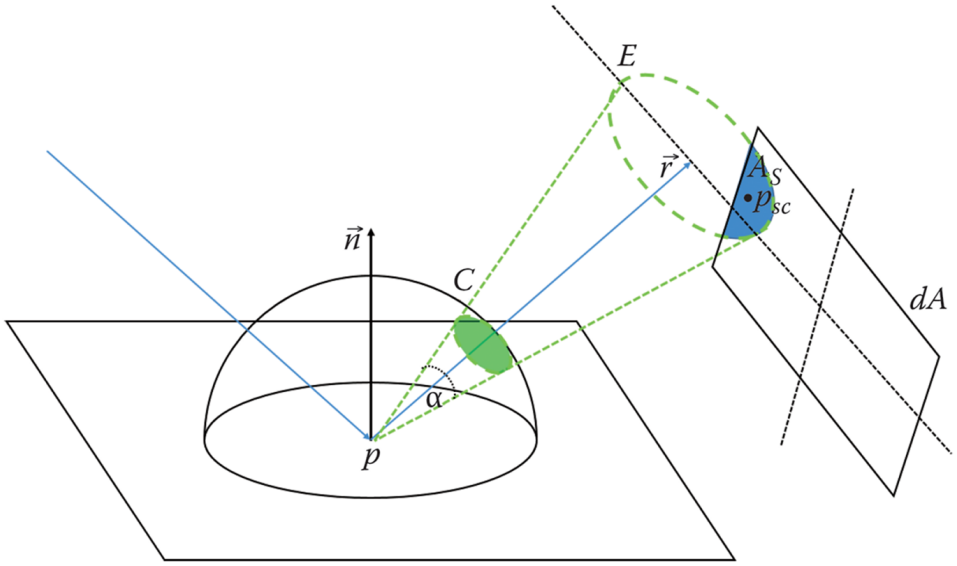


Figure 5.7 Calculation of the specular sample point p_{sc} at the center of the intersection area A_s of the light and the ellipse E at the base of cone C. Image source: [6]

The most representative points implementation yields decent results in many situations, yet the specular reflections do not accurately scale with increasing surface roughness. A further problem is that since the sampled point is not optimized for the Fresnel effect, a surface point that is not in the core of the specular reflection is often darker than it should be, leading to the highlights of rough surfaces being less visible than with other techniques. Some examples of the technique are shown in Figures 5.8, 5.9 and 5.10.

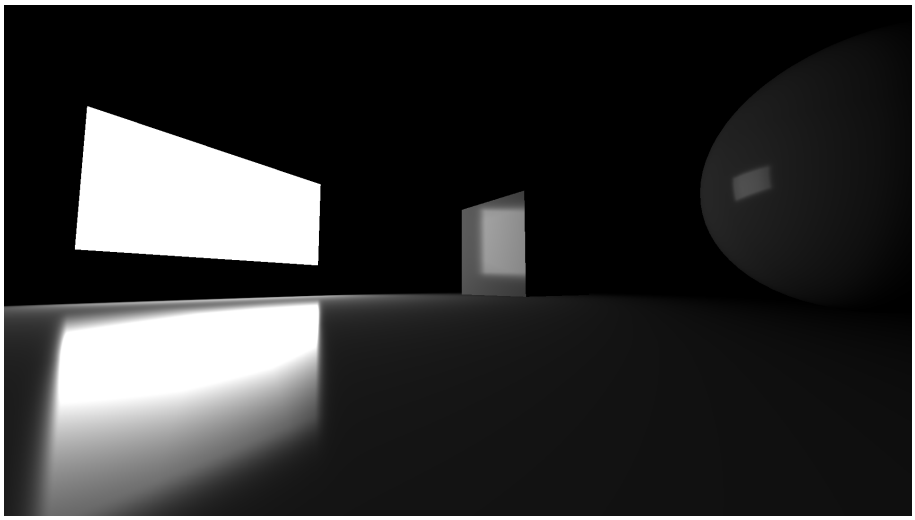


Figure 5.8 Most representative point reflection at a grazing angle

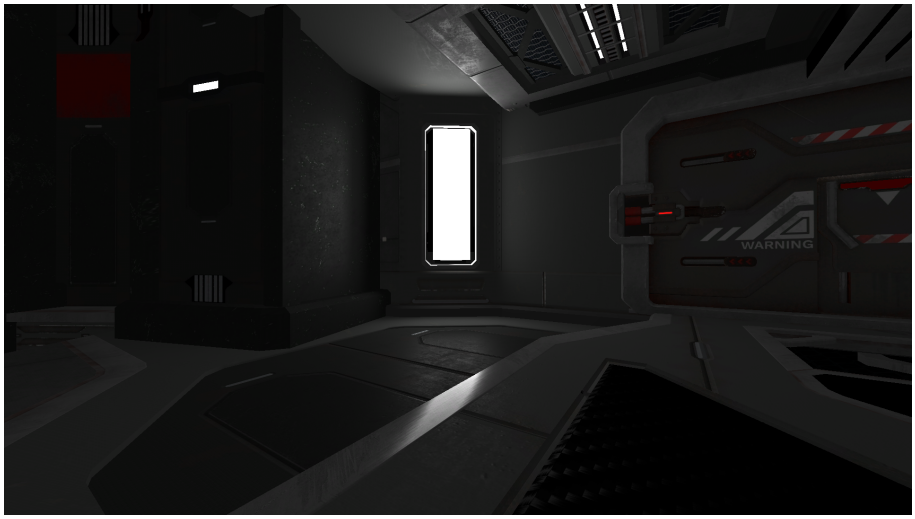


Figure 5.9 Most representative point shading in a hard surface scene

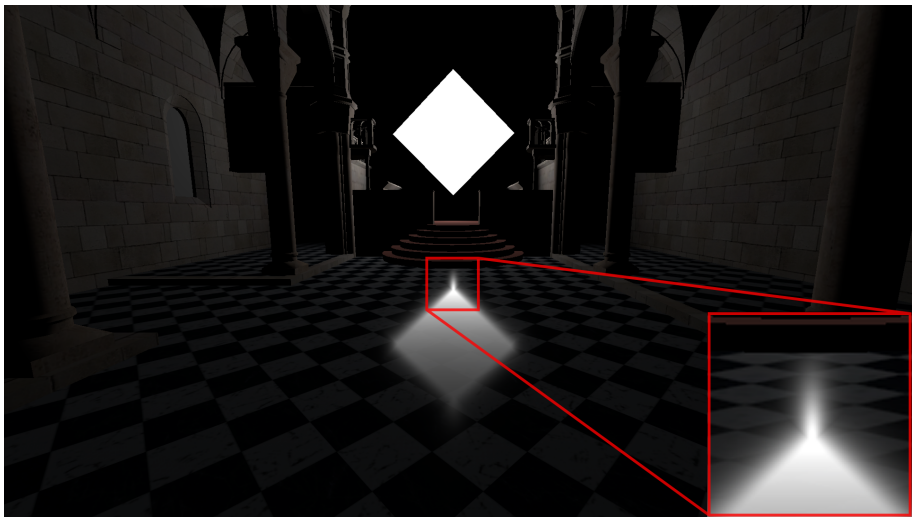


Figure 5.10 Artifacts at the corners of the specular reflection created through most representative point shading

5.4 Shading with Linearly Transformed Cosines

While the previous techniques focused on calculating the area integral via estimation by taking particular samples, Heitz et al. [7] published an approach to obtain an approximate value by actually computing an integral in closed form that yields a result similar to the actual value of the area light integral using linearly transformed cosines (LTC). Their idea is based on the fact that a cosine distribution can be integrated over a line on a sphere with a formula conceived in 1760 by Lambert [17], and that summing up the integrals of the edges of a spherical polygon equals the integral over the area of the spherical polygon. Strictly speaking, the cosine distribution can be linearly transformed through a matrix multiplication, such that it closely resembles the BRDF of a surface under a specific view direction. Figure 5.11 shows how a BRDF, as a function of the incident light direction under a fixed view direction, would be approximated as a linearly transformed cosine distribution. The color of each point on the spheres represents the proportion of light that would be reflected into the viewer's eye when the vector from the light to the shading point (center of the sphere) passes through that point.

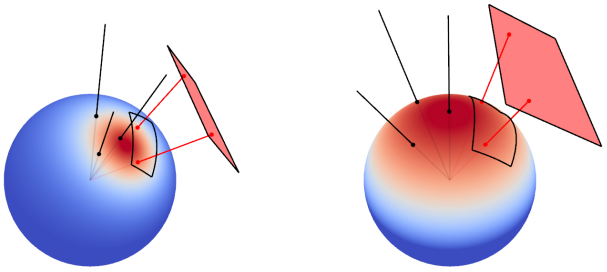


Figure 5.11 Approximation of a BRDF as a linearly transformed cosine distribution. Left: A distribution that closely resembles the BRDF of a shading point. Right: A cosine distribution. Integrating over both of the spherical rectangles gives the same result. Image Source: [7]

In a rendering context, we have a BRDF, but we need to obtain a matrix that approximately transforms it to match the cosine distribution over which we can integrate. Then, we apply this matrix to the corners of the area light and integrate the cosine distribution on the edges of the transformed light. Since the required matrix is unknown, Heitz et al. [7] precompute the matrix for a number of viewing angles and surface roughness values and store the matrix values in textures to be sampled at runtime. They do so by fitting a matrix M that transforms the cosine distribution such that it closely matches the BRDF, then compute the inverse M^{-1}, which would transform the BRDF back to closely match the cosine distribution, and store the components for this matrix in a lookup table (LUT). While the texture sampling and storage cause some performance and memory

overhead, the results appear convincingly realistic in most configurations.

Unlike Heitz et al. [7], we do not store the inverse matrix M^{-1}, but as the authors suggest in subsequent work, we store the components of the matrix M and compute the inverse dynamically, as this can be done very quickly and improves the accuracy of the results as it avoids storing abnormal values that cannot accurately be interpolated. The authors describe normalizing the inverse matrix M^{-1}, such that one of its components is 1, which we will skip and store one additional component in our lookup table instead.

5.4.1 Fitting the specular GGX BRDF

In order to calculate the matrix M^{-1} that transforms a point on the hemisphere over the BRDF to its equivalent on the hemisphere of the cosine distribution function, Heitz et al. [7] provide a program that minimizes the L^3 error between the selected BRDF and the cosine distribution transformed by the matrix M using the Nelder-Mead optimization algorithm. The shape of the matrix M is shown in Equation (5.6).

$$M = \begin{bmatrix} a & 0 & b \\ 0 & c & 0 \\ d & 0 & e \end{bmatrix} \tag{5.6}$$

The program calculates M for a range of roughness values and view angles and creates a lookup table in the form of a texture. We merely adjust it to store the matrix M instead of the inverse M^{-1} for the abovementioned reasons. To obtain M^{-1}, we calculate the scaling term s in Equation (5.7), and then use Equation (5.8)

$$s = \frac{1}{ae - bd} \tag{5.7}$$

$$M^{-1} = \begin{bmatrix} se & 0 & -sb \\ 0 & \frac{1}{c} & 0 \\ -sd & 0 & sa \end{bmatrix} \tag{5.8}$$

Transforming the result to account for the right-handed y-up coordinate system of the Godot engine, we obtain the conjugate matrix M_G^{-1} with Equation (5.9).

$$M_G^{-1} = \begin{bmatrix} 0 & -sd & -se \\ 0 & sa & sb \\ \frac{1}{c} & 0 & 0 \end{bmatrix} \tag{5.9}$$

Additionally, the lookup table stores two values, f_n and f_g, to calculate the Fresnel terms, as shown in Equation (5.10).

$$F = f_0 f_n + (f_{90} - f_0) f_g \tag{5.10}$$

Even though the supplied BRDF fitting program and the Godot engine both employ the GGX model, there are some subtle differences between the implementations. As described in Section 2.3, there are different ways and approximations to calculate the terms of GGX. The Godot engine and the LTC reference material differ in their shadow masking and normal distribution function. We incorporate the implementation used by the Godot engine into the fitting program and compute respective matrix lookup tables. The difference between using either of the two tables is then analyzed in Section 5.4.2.

5.4.2 Comparison of GGX implementations

The rendered scenes, along with the ground truth obtained through path tracing, are shown in Figure 5.13. Any differences between the images rendered with different lookup tables are hardly perceptible with the bare eye. We thus compare the FLIP and LPIPS values of the ground truth comparison in Table 5.1

GGX basis for LUT	Mean FLIP	Mean LPIPS
Godot GGX	0.396 06	0.203 39
Heitz et. al. GGX	0.394 35	0.203 16

Table 5.1 Mean errors of Godot GGX LUT and reference GGX LUT from ground truth, rounded to 5 decimals

Indeed, even FLIP and LPIPS values to a ground truth are highly similar, with a slight preference for the lookup table based on the GGX of the reference implementation, which might be explained by the reference using a more exact formula to calculate GGX. In subsequent experiments, we will thus use the lookup table from the reference implementation unless stated otherwise. The dimensions of the lookup tables we are using are 64×64, amounting to a total file size of approximately 112 kilobytes. While not a significant burden to the engine's size, it might still be interesting to explore whether the matrix values could be obtained in an alternative way.

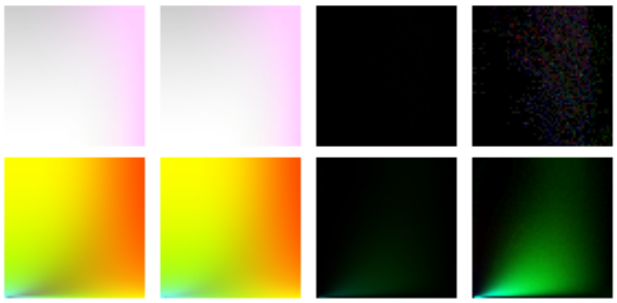


Figure 5.12 Lookup tables used for specular reflections visualized as textures. First column: tables for Godot GGX. Second column: tables for GGX by Heitz et al. [7]. Third column: Differences. Fourth column: Differences, scaled (×20 top, ×5 bottom).

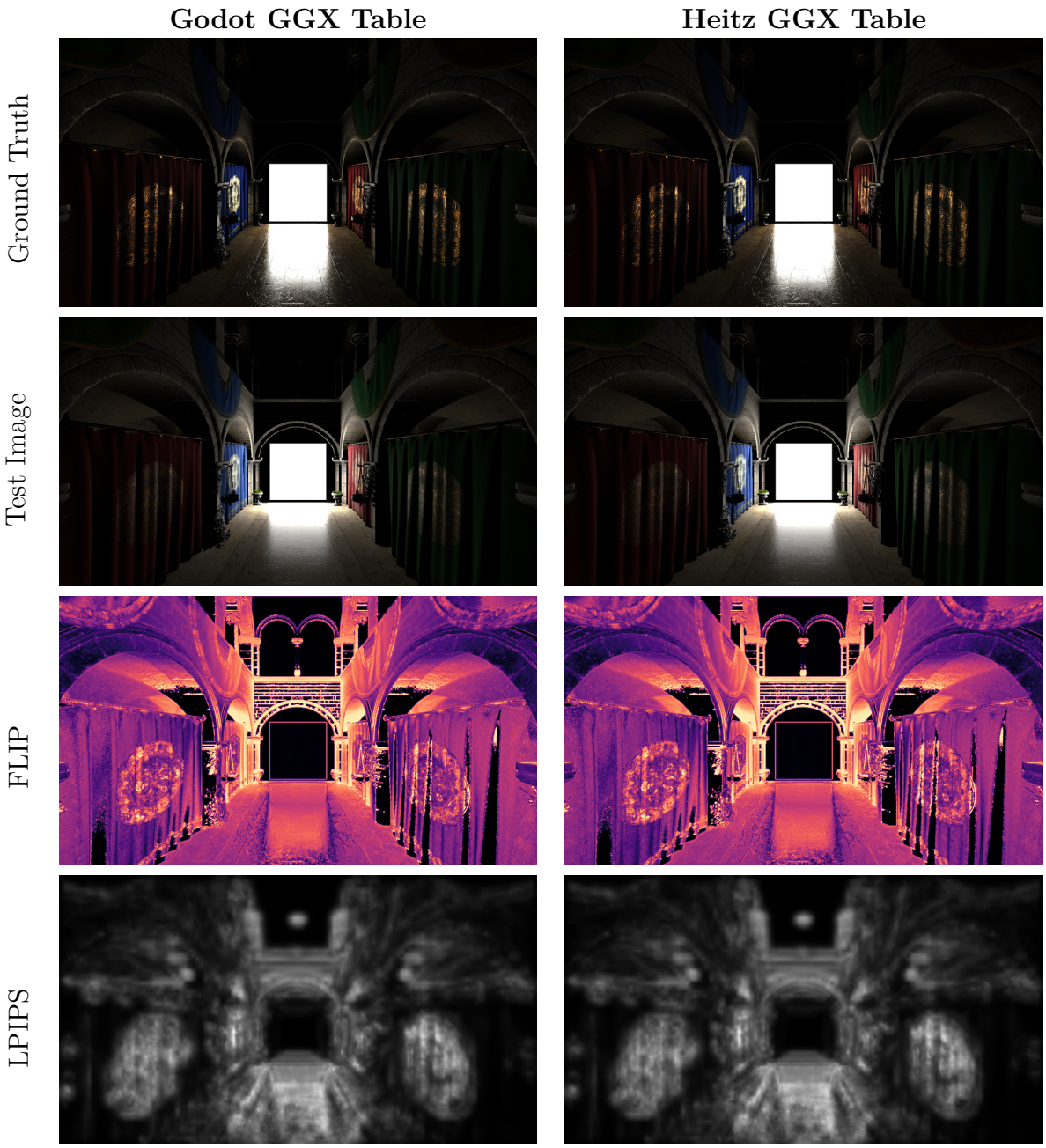
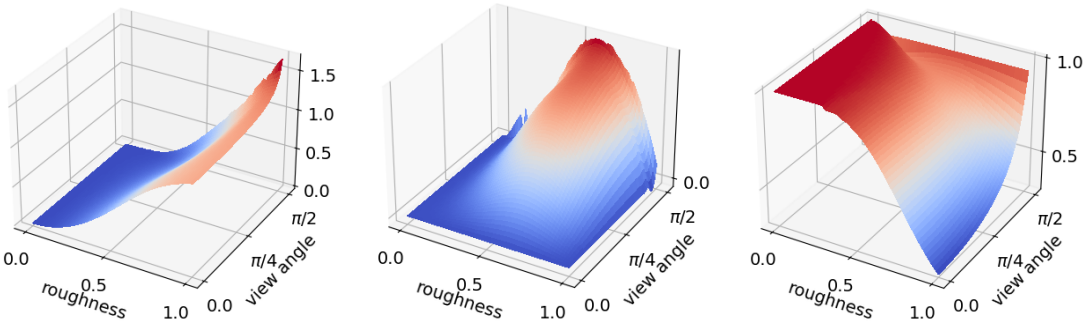


Figure 5.13 Area Light Shading with LTC using lookup tables calculated from the GGX implementation in the Godot engine and the GGX implementation of Heitz et al. [7] compared to a path-traced ground truth with FLIP [68] and LPIPS [69]

5.4.3 Polynomial fitting of Lookup Tables

Observing the lookup tables as textures in Figure 5.12 and in three dimensions in Figure 5.14, we see that they are mostly smooth. One might be curious whether they could thus be approximated by some function that can be evaluated at runtime. For the set of components $T = \{a, b, c, d, e, f_n, f_g\}$ of our lookup table, we will attempt to fit a polynomial $P_k(x, y)$, for $x = 1 + \alpha$ and $y = 1 + \theta$, with $\alpha \in [0; 1]$ being the roughness input, $\theta \in [0, \pi/2]$ being the view angle, and $k \in T$. We do so by least-squares fitting, using the python package `scipy.linalg`.

We first check how the root mean square error (RMSE) $\text{RMSE}(P_k, k) = \sqrt{(P_k - k)^2/N}$ of a polynomial P_k fitted to a component $k \in T$ behaves depending on the degree of the polynomial. As it turns out, $\text{RMSE}(P_k, k) < 0.11 \; \forall \; k \in T$ for P_k being of degree 5, so we decide to choose degree 5 for our tests. A graph of the RMSE and error variance of polynomials of varying degrees compared to the true lookup tables are shown in Figure 5.15 for the components a. The actual errors are shown in Figure 5.16. The error graphs of the other components behave similarly but have been omitted for brevity. The coefficients for all polynomials are included in Appendix C.



(a) Matrix component a (b) Matrix component b (c) Fresnel factor f_n

Figure 5.14 The lookup tables plotted over roughness and view angle in three dimensions for some components $k \in T$

In our tested scenarios, using the polynomial approximation yields an RMSE of less than 0.01 compared to the lookup table. As evident in Figure 5.17, the difference decreases with increasing roughness values, with no discernible differences to the bare eye, even at a very glossy surface. It remains to be answered whether calculating a degree 5 polynomial fit is severely more expensive than a texture lookup. Since a complex scene uses a high number of textures, the polynomial calculation time might amortize during texture loading.

In our tests, the texture lookup turned out to be faster as the number of lights in the scene increased, with no significant difference observed when the scene was lit by just one light. This can be explained by the texture being stored in a cache, whereas

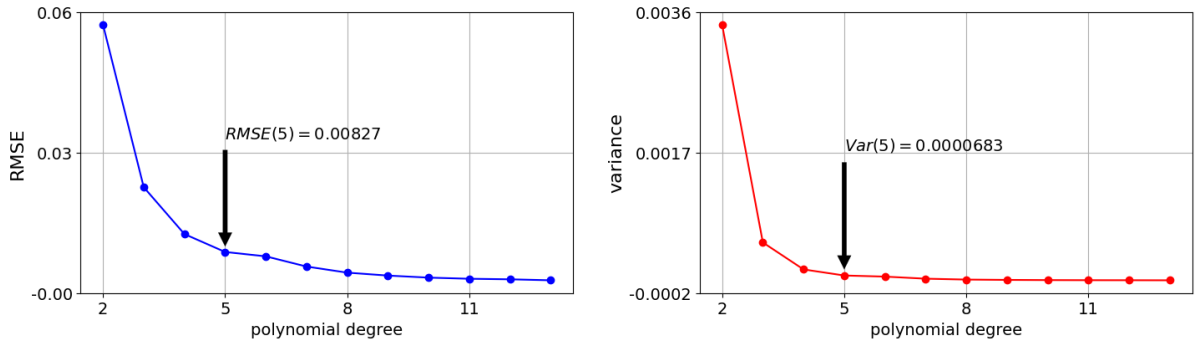
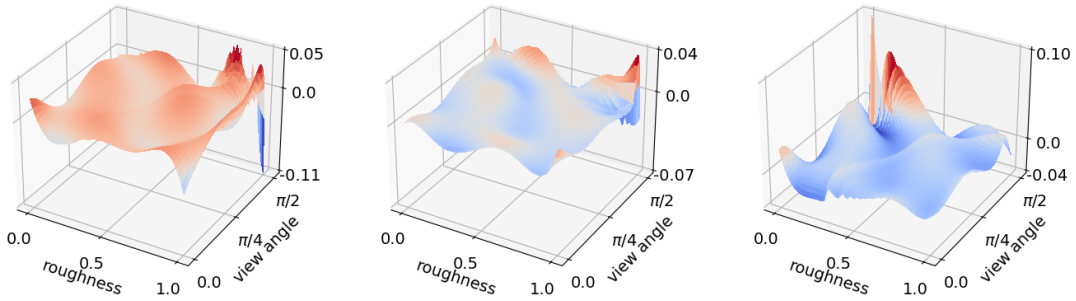


Figure 5.15 Exemplary graph for the mean error RMSE(P_a, a) and variance $Var(P_a - a)$ of a polynomial P_a fitted to the component a of the lookup table over the degrees of the polynomials



(a) Error $P_a - a$ (b) Error $P_b - b$ (c) Error $P_{fn} - f_n$

Figure 5.16 The errors of an approximation of several components of the lookup table with a polynomial of degree 5

polynomials need to be recalculated every frame. Performance graphs for rendering the TPS Demo scene with a varying number of lights are shown in Figure 5.18. We can see that while the average GPU rendering times are still very similar when rendering just 16 lights, the gap increases with the number of light sources.

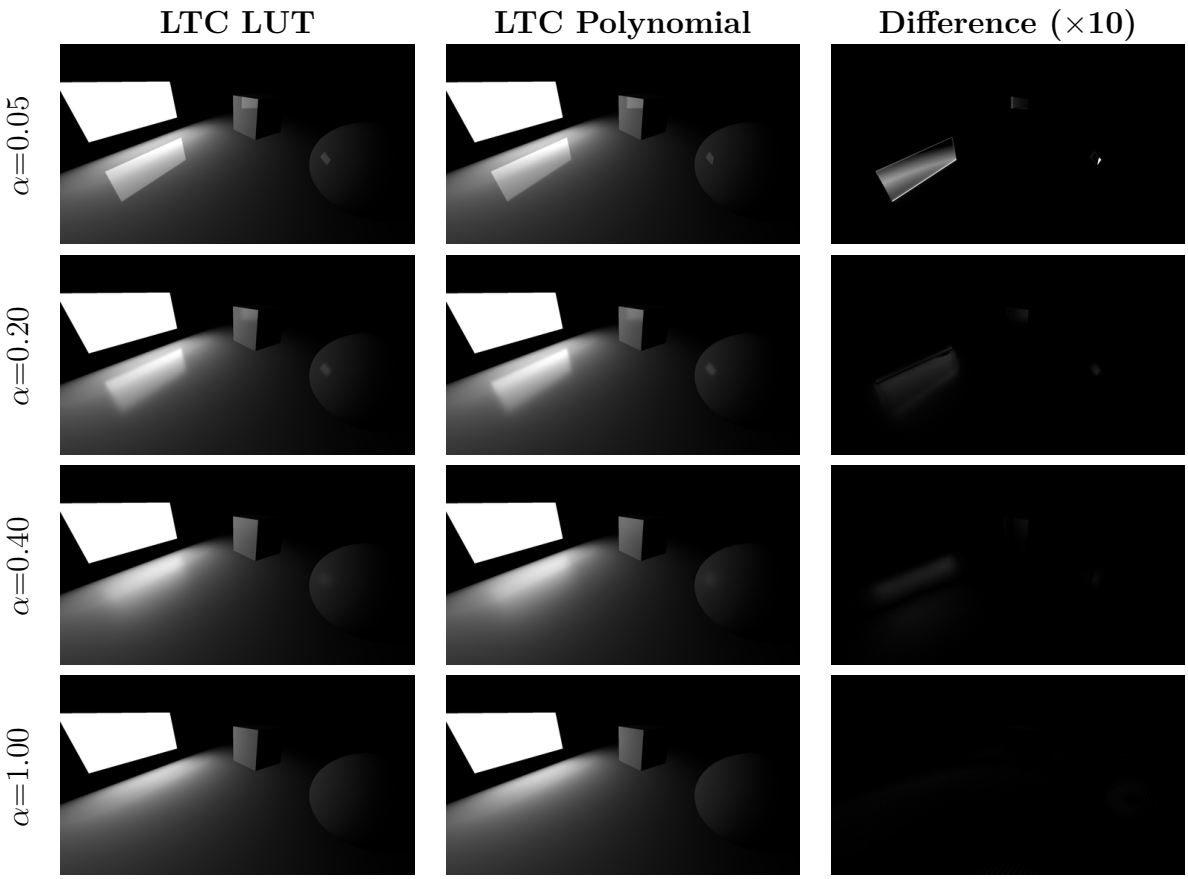


Figure 5.17 Area Light Shading with LTC using a lookup table and using a polynomial approximation for different surface roughness values α. The specular lighting has been doubled compared to the diffuse, and the difference image is scaled by a factor of 10.

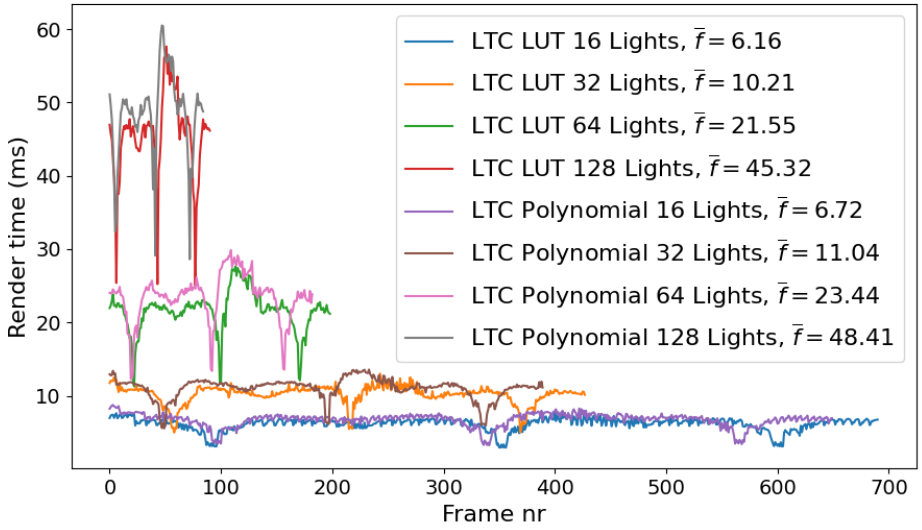
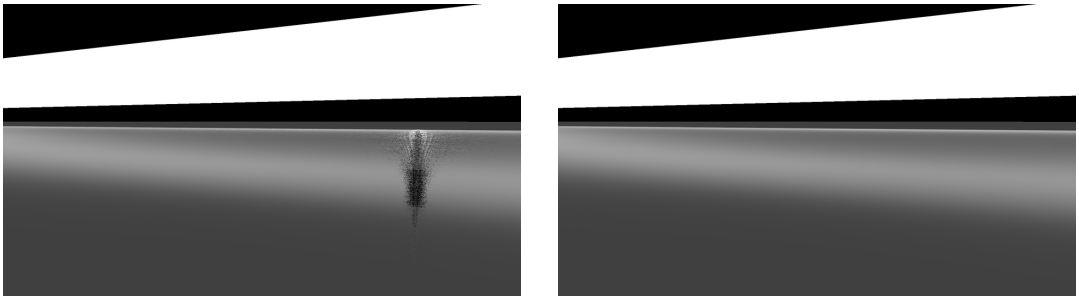


Figure 5.18 GPU rendering time in the TPS Demo scene with different numbers of lights using a lookup table versus calculating a polynomial for area light shading with LTC. Camera animation was used to make for a dynamic scenario.

5.4.4 Precision issues

When implementing the algorithm in the Godot engine, we encountered a precision problem with the integration. Although not apparent in the reference material provided by Heitz et al. [7], this issue appears to have been unresolved, as, to our knowledge, no discussion of it has been published in research. The problem arises from Lambert's integration formula, which calculates the edge integrals using cross products.

If we take an area light with transformed corner points v_i, with $i \in \{1, 2, 3, 4\}$ relative to a shading point o, with $v_{1x} > 0$ and $v_{2x} < 0$ and we project the points v_i to points p_i on the unit hemisphere around o, we end up with points p_1 and p_2, which may lie on almost exactly opposite sides of the sphere, with their dot product approaching -1 and cross product approaching zero, as $||v_{1x}||$ and $||v_{2x}||$ increase. This leads to noisy artifacts in the specular reflections that are especially noticeable as $||v_1 - v_2|| \gg ||v_2 - v_3||$.



(a) Precision issue (b) Our fix

Figure 5.19 The precision issue is resolved by splitting a problematic edge integration into two separate integrals.

Our idea to prevent the imprecision is that since the integral over any spherical polygon is the sum of the integrals over its edges, we can calculate one of those *problematic* edge integrals as the sum of two integrals by introducing a midpoint p_m on the edge of the transformed area light. We cannot just project the point halfway between v_1 and v_2, as that point could land anywhere on the projected spherical edge running from p_1 to p_2. Ideally, we want the point in the middle of the projected edge, but we need to take into account that there can be more than one way to connect two points on a sphere with a line. For example, two points on opposite sides of the equator could be connected via the equator or via the poles. To find the midpoint of the projection of the edge, we calculate the point v_m on the line from v_1 to v_2 that is closest to o and project it on the sphere to obtain p_m. This is shown in Figure 5.20.

If we implement this edge splitting method, we can then calculate and add the integrals over the spherical lines from p_1 to p_m and from p_m to p_2, with the same formula we would have used on p_1, and p_2. Once we do so, the artifacts are gone, as shown in Figure 5.19.

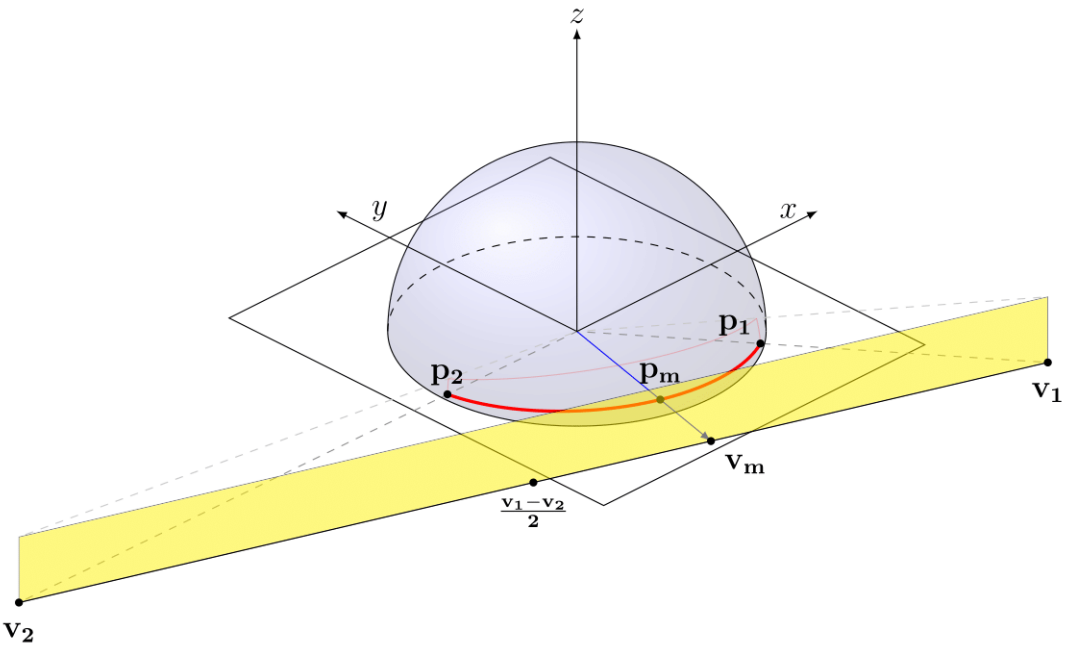


Figure 5.20 Obtaining the midpoint of the spherical edge requires calculating the closest point on the line from v_1 to v_2 and projecting it onto the unit sphere.

5.5 Quality and Performance Comparison of Shading Algorithms

To compare the area light algorithms we implemented in the previous sections, we perform various tests, comparing them to each other in terms of rendering time and to a path-traced ground truth to obtain measurements of quality. In our first test, we use the Sponza atrium (in its version with banners, curtains, and foliage to observe a wider variety of materials) as an example scene. The floor has been slightly tweaked to have a glossier surface and demonstrate specular reflections more clearly. An area light is added in front of the camera, and the scene is then rendered for five seconds, from which the average rendering times are calculated. We render the path-traced version of the same scene in the free and open-source graphics program Blender [65], employing its *Cycles* renderer as the ground truth for our tests.

5.5.1 Test setup

To match the scene of the Godot engine in Blender, we import the same 3D model file (`.obj` or `.gltf` files) and position the camera and the rectangular area light source at matching positions. The camera needs to be set to use a certain field-of-view angle, which differs in its specification from the Godot engine, as it is defined by vertical field-of-view scaling. The default field-of-view angle of 75° in the Godot engine matches a field of view of angle about 107.512 37° in Blender. We then set the rendering output to be at

1920×1080 pixels, with up to 1024 samples per pixel and denoising enabled. Shadows are completely disabled, and the light paths are set to bounce 0 times, such that only direct lighting contributes to the final image, as we are not trying to simulate global illumination effects. The image is output in OpenEXR format, ensuring that no color transform is applied and values are all in scene linear color spaces.

In the Godot engine, the camera resolution is also set to 1920×1080 pixels, and a script exports an OpenEXR image from the viewport texture at runtime. The images are then compared with another script, which calls library methods for the difference evaluators *FLIP*, developed by Andersson et al. [68], and *LPIPS*, developed by Zhang et al. [69], and writes the difference images to our disk. Mean error values are output to a file to numerically assess the quality of our algorithms. Matching the light intensities is a more challenging task, as there is no direct conversion formula from Blender's light intensity in units of Watt to the unitless light intensity in the Godot engine. Instead, we render each scene multiple times in the Godot engine with varying intensity values for each algorithm and compute the FLIP difference values. We then select the intensity value where the mean error decided by FLIP was the lowest.

For performance tests, there is no need for ground truth, and we simply render the scene in the Godot engine over a period of several seconds. The built-in profiler is used to evaluate GPU render times of each frame rendered, which are then exported to a `.csv` file. This means that the total number of frames rendered varies, depending on how many frames the algorithm was able to render in the given number of seconds. The first 20 frames are always discarded to eliminate any instabilities during engine startup, as frame rates tend to be significantly lower during this time. The output is then shown in a graph, and averages are computed.

5.5.2 Quality Test

In our first test, we render the Sponza atrium scene with one area light with each of our algorithms. For the stochastic shading algorithms, we render images using $2, 8, 32$, and 128 samples. The returned errors are shown in Table 5.2. The rendered images, along with the ground truth (GT), are shown in Figure 5.21.

We can immediately observe a number of aspects for each algorithm, starting with the stochastic Monte Carlo-based algorithm. While visually increasing the number of stochastic samples reduces visual noise, especially in specular reflections, the mean FLIP value does not respond to this change, whereas the mean LPIPS error does lower with more samples. In Table 5.2, we observe that FLIP assigns high rate error values to the stochastic algorithm in the curtains and banners, which can be attributed to the reduced specular lighting resulting from the Fresnel effect as a consequence of our specular sampling strategy. As we focus on taking samples that produce specular reflections, we

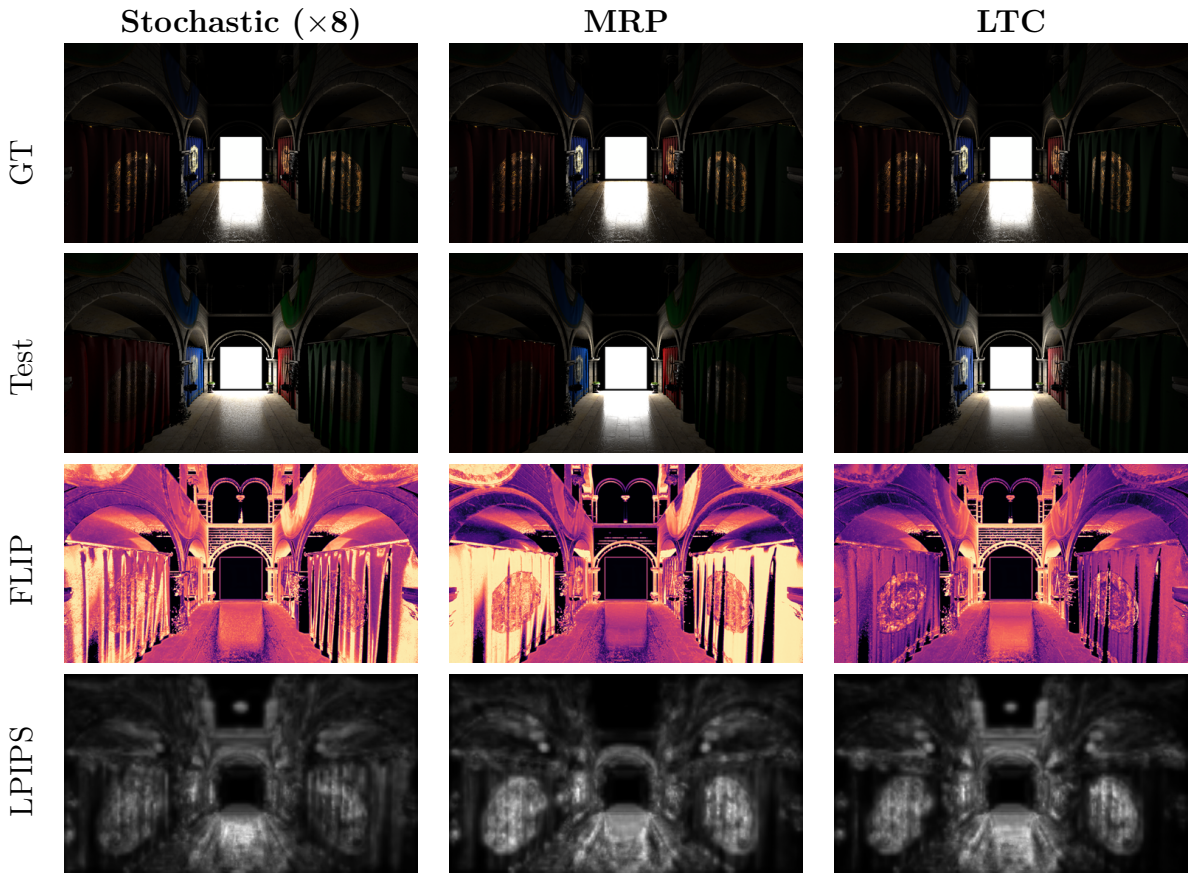


Figure 5.21 Area Light Shading with different algorithms compared to a path-traced ground truth with FLIP [68] and LPIPS [69]

underestimate the contribution of samples that are further from the reflection cone but still produce significant radiance on rough surfaces. The FLIP difference image also reveals high dissimilarities on surfaces directly around the light itself, where the ground truth is completely dark, yet we can see the silhouettes of some bricks, pillars, and arches in our image. It is unclear why this illumination does not appear in the ground truth, but a likely reason is the different implementation of the BRDF in the path tracer, as the illumination seems to be slightly more focused forward, even though its angle is set to 180 degrees. In the LPIPS difference image, we can see the strongest error in the specular reflection, which is due to the noise and its strength fading more strongly with distance. Notable are also the metallic embroideries on the curtains, which show a stronger and more saturated reflection in the ground truth. There is also a conspicuous white circle in the top center of the image, where a pillar is dimly lit in our implementation yet completely dark in the ground truth.

Moving on, the most representative point algorithm shows even stronger errors on curtains and banners due to its nearly total absence of Fresnel reflections. As we only take one sample per fragment for the specular reflections in MRP, we are not able to approximate the entire light's Fresnel effect, which is especially noticeable at the ridges of

Algorithm	Mean FLIP	Mean LPIPS
Stochastic (2 samples)	0.519 94	0.289 56
Stochastic (8 samples)	0.522 00	0.258 51
Stochastic (32 samples)	0.522 26	0.246 88
Stochastic (128 samples)	0.522 11	0.242 30
MRP	0.518 80	0.202 61
LTC	0.392 45	0.195 87

Table 5.2 Mean differences of Stochastic, MRP, and LTC area light shading algorithms compared to ground truth

curved surfaces, such as the wavy curtains. While the specular reflection arguably matches the ground truth the best, the entire image appears rather dark otherwise, as the single sample used for diffuse lighting underestimates illumination in some places compared to the other algorithms, while it overestimates it in others, such as immediately below the arches.

Evidently, the area light shaded with linearly transformed cosines performs the best in our test. Not only is the specular reflection smooth and noise-free, but the algorithm also achieves a decent simulation of the Fresnel effect, as curtains and banners are lit more softly than in the other algorithms without desaturating the image too much. Similar to the stochastic algorithm, the most noticeable differences are perceptible on surfaces directly around the light.

In all three columns, we notice that the area light's outline is not exactly matched, and specular reflections and surfaces at wide angles around the light are not lit in the same way. While a best effort was made to align the two renderers, some imperfections are evident; therefore the difference values, shown in Table 5.2, have to be interpreted in light of the described limitations.

5.5.3 Performance Tests

The performance of the algorithms tested in the previous section is shown in 5.22. Here, the LTC algorithm places second fastest, surpassed by the MRP algorithm by about 0.21 milliseconds on average. The stochastic algorithm naturally requires more time to calculate its results depending on the number of samples. Above 32 samples, performance drops considerably compared to the gained reduction in noise.

Since we also want to know how the algorithms scale, we include another test with varying numbers of area lights. We render the TPS Demo scene with a selected number of lights and see that, on average, the MRP algorithm is about 30% faster on the GPU when rendering 16 area light sources, as compared to the LTC algorithm with our precision fix (without the fix the difference is only slightly lower). This number even increases to about 35% when we increase the number of light sources to 64. We thus assume that the cost

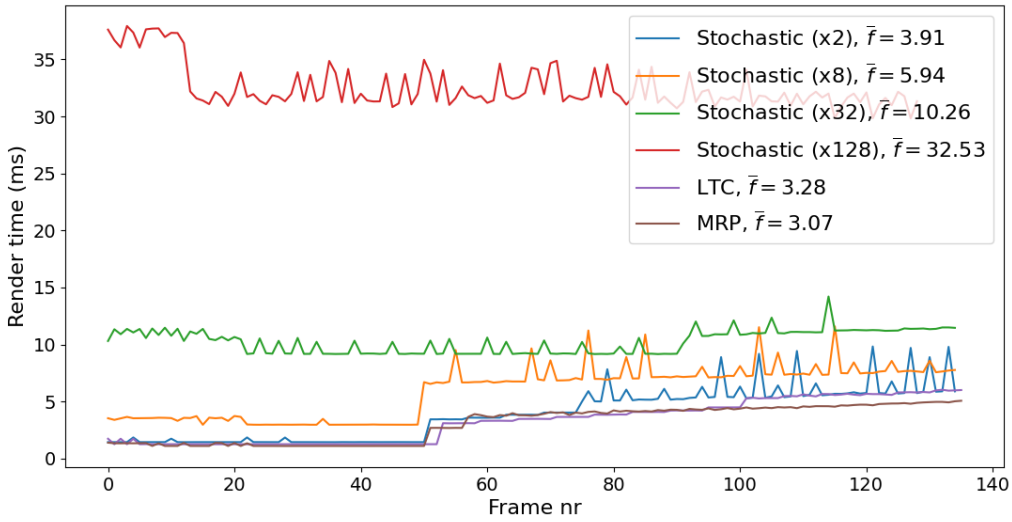


Figure 5.22 Performance measurement of the algorithms. The y-axis represents the time the GPU needed to render the viewport with one area light using a specific algorithm.

of sampling lookup table textures in the LTC algorithm on our system does not amortize as the textures get loaded into the cache with a higher number of lights when comparing it to MRP. The data shows that the stochastic algorithm in our test required more time when taking four times the number of lights than when taking four times the number of samples. Moreover, the algorithm performs worse than MRP even when taking just one sample, which can be explained by the calculations required to set up the sampling strategy, where a spherical rectangle needs to be projected, and weights for each corner need to be calculated.

On the CPU, there is no significant difference in the performances of the algorithms; hence, respective graphs have been omitted for brevity.

5.6 Performance of Area Lights versus Spotlights

To make a case for employing area lights over spotlights, we utilize an algorithm from the previous section and compare its computation time to that of using spotlights. While both LTC and MRP are viable options, we decide to use the LTC algorithm specifically for this test, as it not only demonstrates decent performance but also excels in visual quality, having the lowest mean error from the ground truth among all compared algorithms. For this test, we take the Bistro scene and add light sources at 22 positions. We then render the scene, once using spotlights and once using area lights. The performance is then shown in graphs.

Again, differences are the most noticeable on the GPU, where area light shading in the Bistro scene takes about twice as long as rendering spotlights, as shown in Figure 5.25. In our test with a moving camera and varying numbers of area lights in the TPS Demo

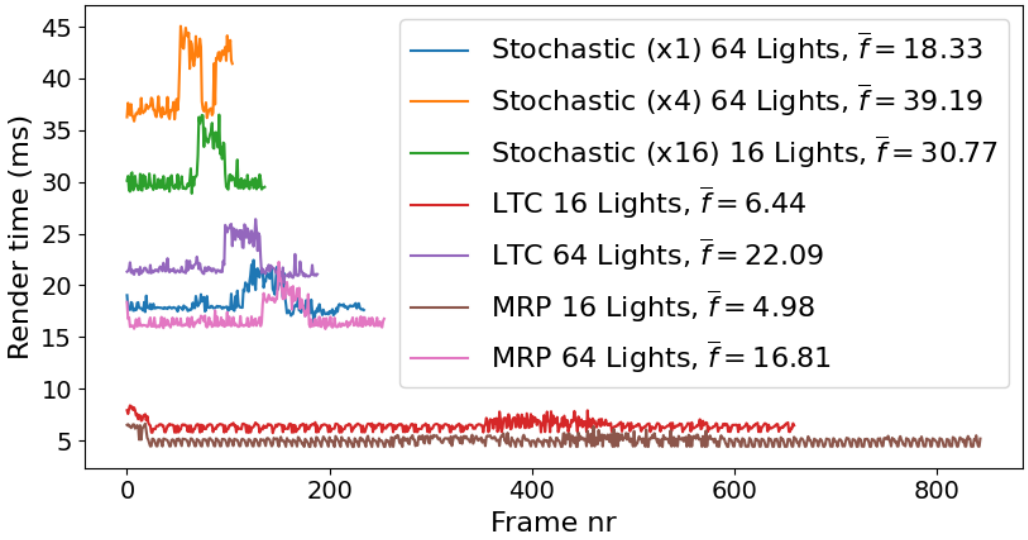


Figure 5.23 Performance measurement of the algorithms rendering a number of area lights. The y-axis shows the time the GPU needed to render the viewport with a certain number of area lights using a specific algorithm.

(a) Wide angle spotlights (b) Area lights

Figure 5.24 Comparison of shading with spotlights with a very wide angle (left) and area lights (right). The visual differences are subtle in this configuration.

scene, shown in Figure 5.26, we can also observe a similar trend, where approximately two spotlights can be shaded in the time it takes to shade one area light.

5.7 Results and Discussion

We observe that area light shading incurs a significant cost compared to more performant means of shading, so we recommend employing them where their use makes a substantial difference to point lights, i.e., for diegetic lights shining on glossy surfaces. In many situations, such as in Figure 5.24, even though the observer would expect an area light, a point light will still be a more efficient approximation, disregarding the impact of soft shadows for now. For situations where using an area light significantly enhances image realism, game engines must decide which shading algorithm to implement, as they come

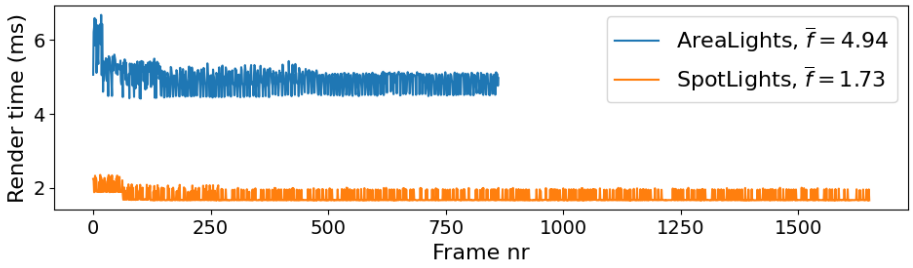


Figure 5.25 Comparison of rendering times when shading the Bistro scene with spotlights and area lights in a static camera configuration.

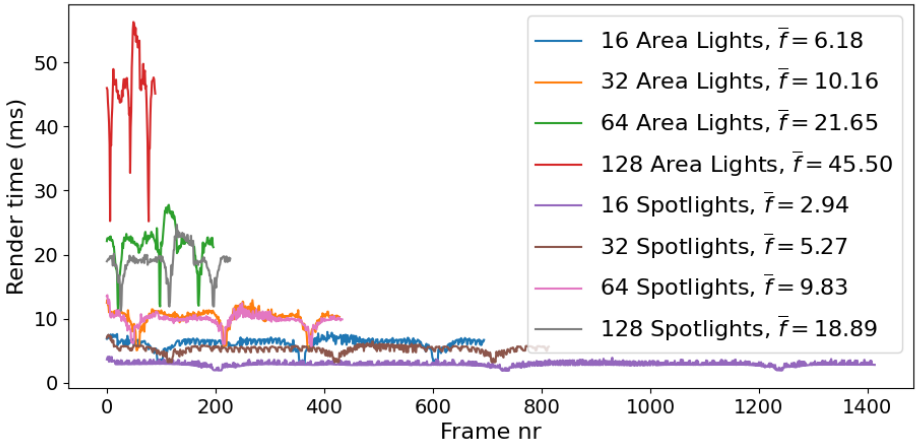


Figure 5.26 Comparison of GPU rendering times when shading the TPS Demo scene with varying numbers of spotlights and area lights.

with different advantages and drawbacks. While the MRP algorithm is computationally more efficient, it produces artifacts when viewed at certain angles, and it is also not the most physically accurate in situations where no artifacts occur. One of its advantages is that it integrates rather easily into existing shading systems of an engine as, for example, the code for the shading model or BRDF can be reused. Extending the algorithm to different shapes is mostly trivial, as shown by Drobot [6]. Code reuse and easy extension are also true for our stochastic shading algorithm, yet its performance is significantly worse. Though anti-aliasing and denoising could improve its visual quality to a point where it might result in better visuals than MRP, as it does not cause any artifacts, the performance implications mostly inhibit its use. The LTC algorithm most closely matches the ground truth, and its calculation time is competitive with MRP or point lights, albeit slightly higher. For a high-fidelity rendering engine, it thus is the most suitable candidate out of the three, in our opinion. Its disadvantages include the requirement of a lookup texture, though it can be replaced with a polynomial computation if one is to accept minor performance losses, and the computation thereof, as an additional software package for generating the lookup tables must be maintained to allow for integrating future developments. It allows for little code reuse, as its computation is almost entirely separate

from that of point lights. In its current implementation, the diffuse shading is limited to Lambertian shading models, as it is computed by integrating a cosine distribution. When it comes to extensibility to different shapes and material effects, Heitz and Hill [44], and KT et al. [70] have shown that line lights, disk-shaped lights, and anisotropy effects are possible, yet all require the precomputation of additional lookup tables, which indicates that there might be further possibility for future research around the topic of LTC, as there have been several publications expanding or using the technique in recent years [44, 46, 47, 70].

Our findings are, in part, limited by the quality of our ground truth calculation. Since we use an external rendering engine for this, we have less control over the way it calculates its shading. Thus, the data obtained by calculating difference images serves to provide an illustration of the quality differences between algorithms but does not contain accurate error values for one algorithm. Furthermore, our research is limited to local illumination of uniform light sources. We have not considered a range of material effects that an area light implementation ought to eventually support, such as translucency, subsurface scattering, refraction, and anisotropy. We also did not research any advanced effects that the Godot engine does not support, such as caustics or iridescence. Were this a requirement, it likely would have to be considered from the start when implementing a lighting algorithm. Another limitation of our data is that our profiling does not analyze in depth what exactly the GPU compute time was spent on, so we cannot accurately state the cost of, e.g., the texture lookup of the LTC algorithm, as this is hard to measure, and not supported by the profiler of the Godot engine.

6 Area Light Shadows

In the previous chapters, we have implemented several algorithms to compute the direct specular and diffuse illumination from an area light. However, as it is now, the light just shines through any geometry in the scene, lacking any form of occlusion. Therefore, we implement and compare several algorithms to compute shadows for our newly added area lights. In Chapter 4, we have discussed why we are focusing on stochastic algorithms based on shadow mapping. In this technique, described in more detail in Section 2.4, we compute a depth map from the point of view of a light source that we can sample to determine whether a shading point has a larger distance to the light source than the sampled value and thus lies in shadow, or not. An example of a shadow map can be seen in Figure 6.1.

Figure 6.1 A shadow map of a spotlight in the Bistro scene. The lighter a pixel, the closer the fragment is to the light source.

For a depth value d sampled from the shadow map of a light source at position l, we can define the visibility function $\tau(v)$ of a shading point v as shown in Equation (6.1).

$$\tau(v) = \begin{cases} 0, & \text{if } ||v - l|| > d \\ 1, & \text{otherwise} \end{cases} \tag{6.1}$$

To start with, we take l as the center of our area light and implement drawing a simple shadow map for it in the Godot engine. This will not give us a soft shadow yet, but we can expand the implementation from there. First, we need to choose the mapping. A spotlight simply draws its shadow map with perspective projection, using a field of view that matches the aperture angle of the illumination cone. This becomes evident, as it only works reliably until an angle of about 120 degrees. Area lights, however, should illuminate the entire 180 degree hemisphere. Hence, we need to resort to one of the mappings employed by omni lights, which are either cube maps or dual paraboloids. We chose the latter, as it circumvents having to render the depth for all cube faces, and the inaccuracy of dual paraboloid mapping should be less evident for soft shadows.

Since, in the Godot engine, shadow maps are not drawn needlessly for lights that

are culled anyway, the light source needs to be added to an array of shadows to be rendered during the culling procedure for a shadow map to be rendered. Furthermore, this is only done for shadows that are marked as dirty, which they are if any of the visual objects within the light's range have moved or their geometry has changed, e.g., through animation. The number of shadows to be updated within one frame is capped at 512. The shadow maps in the array are then drawn by the renderer, which fills instance lists with visible geometry instances for each shadow map to be drawn and adds all information to another array of shadow passes, just as for other light sources. The commands to render all shadows are sent to the GPU all at once, limiting the control we have over the order of execution. After modifying the implementation to also consider our area light source, we can enable rendering hard shadows in rendering the scene.

6.1 Stochastic Sampling of Soft Shadows

For a simple approach to achieve soft shadows, we follow a technique first proposed by Heckbert and Herf [60]. We sample a number of points on the area light, render shadow maps from them, and compute a visibility term between 0 and 1 as the proportion of samples for which the point lies in shadow. If the number of samples is sufficiently dense, we should obtain a soft penumbra. We thus calculate the proportion ψ of a light that is directly visible from the shading point by evaluating the visibility function τ_i for a number n of samples i, as shown in Equation (6.2).

$$\psi_n(v) = \frac{1}{n} \sum_{i=1}^{n} \tau_i(v). \tag{6.2}$$

Integrating this arguably simple technique into a complex rendering engine already poses several challenges that are not discussed in the reviewed literature. First of all, many shadow maps of the same light are required and have to be stored on the shadow atlas. A shadow atlas is typically divided into $s \times s$ quadratic sections for shadow maps, hereinafter referred to as *tiles*, which a light source can *occupy* by writing its shadow map in them. Depending on the size of the light or how close it is to the viewer, we might optimize the number of tiles that a single light uses in a way that it becomes larger the more the shadow is visible. This implies that it is not immediately known how many tiles a light will occupy and at which position on the atlas respective shadow maps will be drawn. In a scene with many light sources, the assignment of the tiles is not constant, as tiles are marked as inactive when a light source is culled or made invisible, and a new light that comes into view might then occupy any inactive slots and draw over the shadow map that had been placed there earlier. This means we need to organize the order in which an area light ocuppies tiles on the shadow atlas. We could either regard the tiles on the atlas as a one-dimensional memory space and write the maps for one light source on a continuous

series of tiles in this space or write the maps at any vacant tiles and maintain an array of indices for each light. The latter might scatter the maps all across the atlas, which would be less convenient for a user trying to debug a shadow mapping issue. The former approach will lead to fragmentation issues, just like with physical memory. Thus, we opt for the scattered approach, maintaining an index array of tiles occupied by a light.

Another feature that is not discussed in the reviewed literature is the tradeoff between the number of tiles on the atlas and the resolution of each shadow map, which affects performance and quality of the rendered shadows. We will analyze this in more detail in Section 6.2. The final point of consideration is how to divide the number of available tiles on the shadow atlas between multiple area lights. Ideally, each area light would obtain a number of tiles proportional to the size of their penumbras on the screen. However, the size of penumbras is not trivial to measure; hence, a heuristic metric taking into account the size of the light and its coverage of the screen could be considered. The metric will vary as the camera moves in the screen, begging the question of whether lights should be able to force other active area lights to vacate shadow map tiles (effectively, they *steal* them from other lights) if the metric concedes significantly greater importance to the light that is currently occupying fewer tiles.

Due to these special requirements of area lights, we decide not to conflict with the shadow atlas that is used by point lights but instead create a new shadow atlas specifically for area lights. This also saves the trouble of integrating with the point lights' subdivision of the atlas into quadrants, as we do not yet know how the number and size of shadow maps affect the quality of our penumbrae. The Godot engine maintains all data related to shadow atlases in a storage class, where we will add additional methods and fields for our new area light shadow atlas. We also add an index array to the light data uniform buffer object, allowing us to compute the position of each occupied tile on the atlas in the shaders. Notably, each viewport and each reflection probe (i.e., anything that renders a separate image of the scene) draws their own shadow atlas, which we have to account for. Additionally, we include settings to configure the resolution of the new atlas and the number of tiles it is divided into for the main viewport and any subviewports.

Finally, we need to decide how we sample the points on the area light. Since the number of samples we can take without significantly sacrificing performance will be much smaller than for stochastic area light shading, and we will frequently vary the number of samples the same light takes, we opt for a pre-determined regular sampling, in particular, the one described by Schwärzler et al. [9]. Samples are initially taken at the four corners of the light, spanning up a single quad. If more samples are needed, the quad is subdivided into four additional quads by adding a sample in the middle and one sample at the midpoint of each edge of the quad. Each of the four new quads can again be subdivided in the same way. This subdivision is illustrated in Figure 6.2, and the shadow maps generated from the four corners of an area light are shown in Figure 6.3.

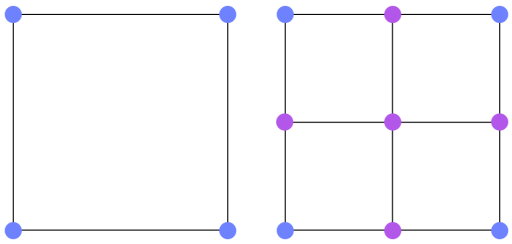


Figure 6.2 Subdivision of an area light for sampling. The four samples of the initial quad are added to by subdividing the quad into four new quads.

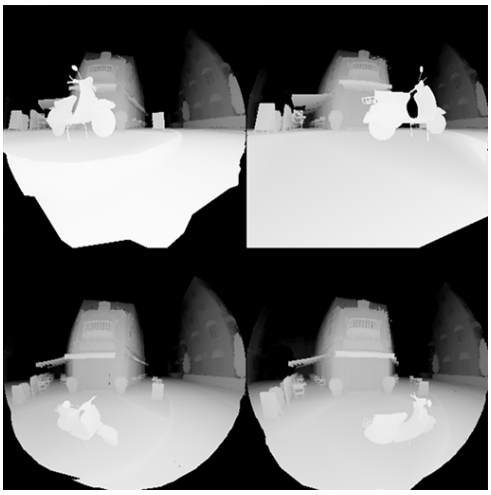


Figure 6.3 The parabolic shadow maps of an area light rendered at its four corners in the Bistro scene

When leaving the number of shadow maps at just one for each corner, we do not yet obtain a smooth penumbra but rather four individual shadows. Generating an enormous number of shadow maps would eventually yield a smooth penumbra. However, we can greatly reduce the required number by sampling several pixels on each of our shadow maps using a strategy like percentage-closer filtering (PCF). Conveniently, this functionality is already implemented for point lights in the Godot engine, allowing us to reuse it for area lights. Increasing the sampling radius in this way allows us to blur the shadows of our area light so we obtain a smooth penumbra. The difference is evident in Figure 6.4.

While shadows look plausible in Figure 6.4, four shadow maps will not be sufficient in some situations, such as when a small object is close to the light source, which will result in four distinct shadows that no amount of filtering will blend (at best the filtering will make the shadows disappear entirely). For these situations, we are looking for the solution proposed by Schwärzler et al. [9].

6.2 Number of Shadow Maps versus Resolution

We render shadows of a simple scene and draw shadows with increasing numbers of shadow maps, requiring more tiles on the shadow atlas while maintaining a resolution of the atlas constant. This means that with every increase in the number of tiles, the size of a single shadow map is halved. We then compare the rendered images to a path-traced ground truth rendered in Blender and check how the LPIPS error changes as we use more maps. We do so once for a shadow atlas size of 8192×8192 and once for a smaller shadow atlas size of 2048×2048.

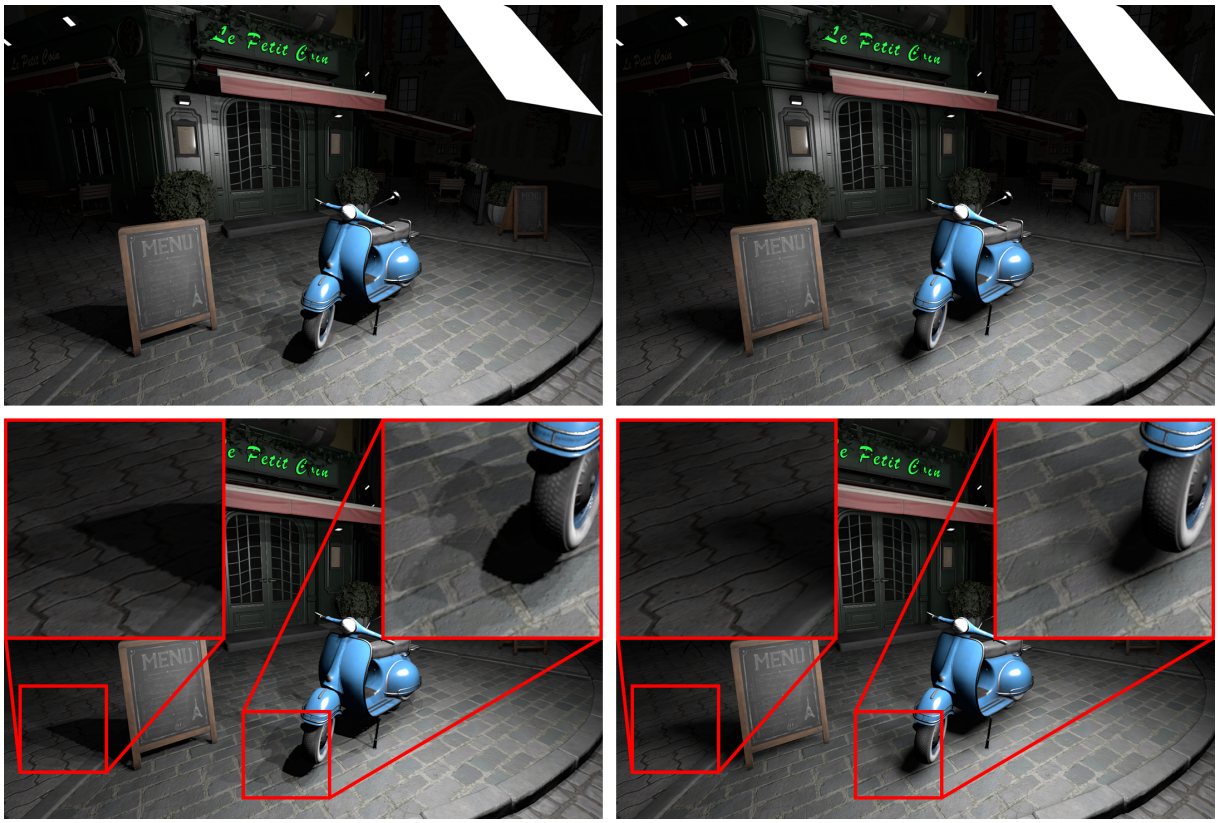


Figure 6.4 Impact of PCF on the use of four area light shadow maps. Left: without PCF. Right: With PCF, using a kernel of size 32.

We can observe that for an $8k$ shadow atlas, the mean LPIPS error decreases consistently as the number of shadow maps increases down to a value of about 0.071 for 81 shadows before it increases again for more shadow maps. For a $2k$ shadow atlas, this turning point is already reached at 25 shadows. Thus, in our example, results are not improved when the resolution of shadow maps is lowered below 256×256 pixels. While this number might change depending on the scene configuration, it can serve as a rough orientation for how large the shadow maps should be, at least. Comparing the images rendered with a $2k$ shadow atlas, we can also see that the size of the core shadow starts decreasing when increasing the number of shadow maps beyond 25 due to the low resolution of the shadow maps. We thus assume that depending on various factors like the size of the area light and the distance and size of the shadow caster, there is an optimal number of shadow maps to render both a soft penumbra and an accurate umbra.

In terms of performance, we found little difference between rendering a $2k$ shadow atlas and an $8k$ shadow atlas with the same number of shadow maps, even for an animated light that redraws its shadow map every frame. The framerate is almost solely defined by the number of shadow maps sampled for both the primitive scene in this example and the more complex Sponza scene.

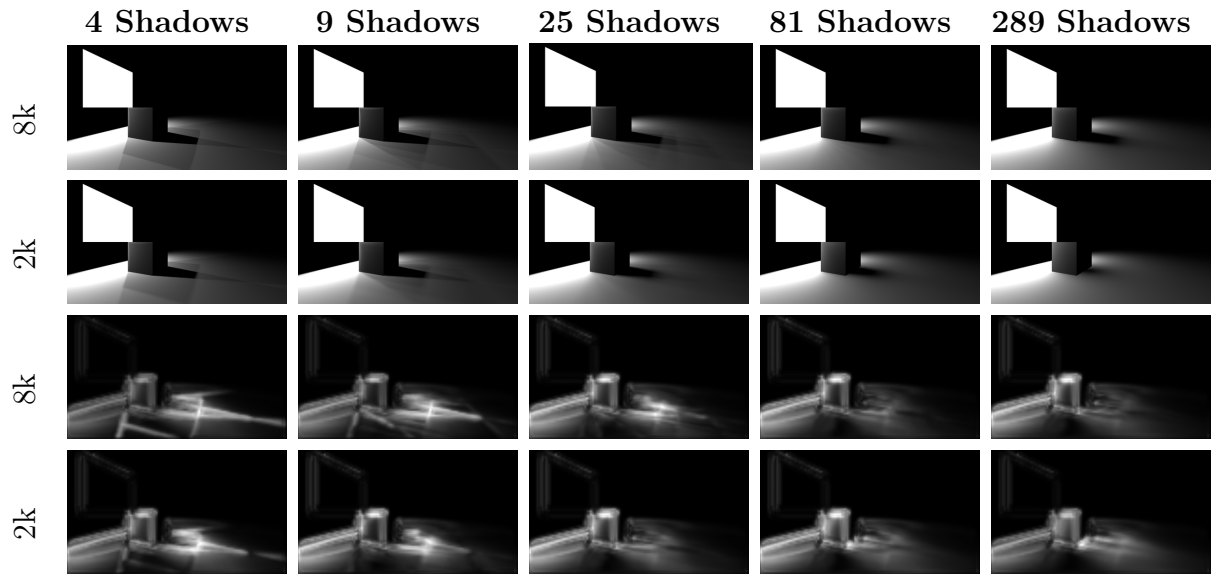


Figure 6.5 Area light shadows with a varying number of shadow maps. First row: rendered image using an 8k shadow atlas. Second row: rendered image using a 2k shadow atlas. Third and Fourth row: Error images between the first two rows and a ground truth generated with LPIPS [69].

6.3 Adaptive Light Source Sampling

Intuition might suggest that not all samples on the light source contribute equally to the appearance of soft shadows and that, in some cases, the penumbra occupies too small an area of the screen to be perceptible by the viewer. This observation motivates performance optimization through an adaptive subdivision of the light source based on perceptual relevance.

We decide to base our implementation on the algorithm invented by Schwärzler et al. [9], which first draws shadow maps as we would normally but performs a second *reprojection* pass immediately afterward, before the opaque objects are shaded. This pass renders the scene from the camera's point of view but only draws the shadow values of one area light. Then, this image, shown in Figure 6.6, is analyzed in terms of the smoothness of the penumbrae by scanning for any pixel that is part of a penumbra, i.e., neither completely black nor completely white, and is surrounded by pixels of the same value. If such a pixel is found, it is regarded as a banding artifact, warranting further subdivision of the area light. This reprojection is carried out for all sets of four sample points, i.e., the corners of the area light before any subdivisions, and then the corners of any quad that the larger quad was subdivided in, in a manner as shown previously in Figure 6.2. Any quad that indicates subdivision adds up to five new points (fewer if an adjacent quad has already subdivided and created a point that can then be shared), for which additional shadow maps are then rendered. The process is repeated until all quads are small enough that no reprojection pass produces banding artifacts, after which the

Figure 6.6 An image created by a reprojection pass for an area light in the Bistro scene. The scene is depicted from the camera's perspective, but only the shadow values of one sample quad of an area light are rendered.

scene is finally rendered.

While the algorithm guarantees physically accurate shadows, it arguably suggests a high performance cost. As an optimization that trades in smoothness, the authors suggest rendering the reprojection pass at a lower resolution, which not only yields fewer subdivisions but also saves on rendering time during the reprojection pass itself. With this optimization, we expect to achieve acceptable level of performance.

To add the additional reprojection pass in the Godot engine, we need to add a new shader pass mode to the material uber shader. We mentioned this to be problematic in Section 2.6, not only due to the complexity it adds to the rendering engine but also because the number of shader versions rises drastically for every mode added. We will later investigate whether the obtained optimization will be worthwhile. A shadow reprojection shader variant is thus added, and shaders are appended appropriate preprocessor branches to avoid any excess calculations and only write the shadow values to the fragment output in this mode.

Subsequently, we add a compute shader that takes the texture drawn onto by the reprojection pass and examines whether banding artifacts occur, in which case it sets a flag in an output buffer. An early exit branch is added in case the flag was already set when running the compute shader on any other fragment. The original paper suggests using hardware occlusion queries to compute the same artifacts, yet we found no benefit in doing so over using compute shaders and attribute this detail to the hardware constraints at the time of the algorithm's publication.

Although exceptional care was taken to ensure that adding the reprojection and compute passes to the engine would not break existing render passes, a major problem was encountered with regard to the real-time rendering engine. To utilize the GPU and CPU fully and not have one wait for the other, game engines can have preparations for the

next frame already starting on the CPU while the GPU is still rendering the current frame (the frame is said to be *in flight*). Although engines can insert rendering commands that utilize resources that other commands need to finish using first, such as the texture that the reprojection pass is drawing on, they are not always able to insert new commands, such as rendering another list of shadow maps, depending on the output of some pass, and much less so when this is to be repeated in a loop until some condition is met. Without removing the directed acyclic rendering graph and inserting a barrier to have the CPU manage this loop until the frame is ready to be presented, thus introducing a *stall*, we are unable to implement the adaptive sampling algorithm as it was conceived in the Godot engine. This is why we suggest a modification to the algorithm, slightly improving efficiency, although undeniably affecting its complexity.

Instead of rendering the full subdivision within one frame, our new algorithm attempts to reuse information gained in recent passes by performing the subdivision over several frames, which should stabilize within the fraction of a second that an observer would hardly perceive. As we would have for the original algorithm, we first define a quadtree of nodes, where each node represents a rectangular area of the area light in canonical space, i.e., the coordinates lie between zero and one, where $(0, 0)$ represents the bottom left corner of the light, and $(1, 1)$ the top right. *Expanding* a leaf node equals the subdivision of a rectangular area into four new rectangles. For, the outermost quad would be subdivided into four quads sharing the new point $(0.5, 0.5)$. *Pruning* a node (meaning to remove its children) equals an "unsubdivision", i.e., merging four rectangular areas created of the same parent rectangle back together. A node is considered *idle* if it is a leaf and already has been a leaf in the last frame, i.e., it has not recently been created or pruned. We then dynamically build a *subdivision tree* over several frames by performing reprojection passes at the end of each frame and using its evaluation in the next frame to determine whether further subdivision is needed. The procedure is formalized in Algorithm 1.

The algorithm requires maintaining a quadtree structure and several arrays for the banding tests ordered in the previous frame. However, it comes with the advantage of only having to run reprojection passes for leaf nodes and their parents in each frame rather than performing reprojections for all nodes in the tree. These gains, however, are less impactful than they might seem, as the majority of nodes of a quadtree are contained in its deepest layers. Disadvantages of our new algorithm include that the shadows for an area light change over several frames, so the observer might briefly perceive flickering shadows when a new light enters the view frustum. Moreover, the decision of subdividing or unsubdividing the tree of maps is always based on the computation of the previous frames, thus lagging behind the current state of the scene. Further optimization of the algorithm is possible by only rendering reprojection passes to test for tree expansion if there are any free tiles available on the shadow atlas.

Algorithm 1 Dynamic area light rectangle subdivision

```
root = Node(Rectangle((0,0), (1,1)))
uniquePoints = root.corners
RenderAllShadowMaps(uniquePoints)

while Light.isVisible do
    if ¬ isFirstFrame(Light) then
        for leaf ∈ root.leafs do
            if leaf.bandingFlag = true then
                leaf.isIdle = false
                newPoints = ExpandNode(leaf) // Add 4 subquads as new nodes
                uniquePoints = uniquePoints ⋃ newPoints
            else
                leaf.isIdle = true
            end if
        end for
        for parent ∈ checkedParents do
            if parent.bandingFlag = false then
                removedPoints = PrundeNode(parent) // Remove Children
                uniquePoints = uniquePoints \ removedPoints
            end if
        end for
    end if
    checkedParents = ∅
    if scene.isChanged then
        MarkAsDirty(uniquePoints)
    end if

    RenderAllDirtyShadowMaps(uniquePoints)
    RenderScene()

    for leaf ∈ root.leafs do
        RenderReprojection(leaf)
        ComputeBandingAndSetFlag(leaf)
    end for

    for node ∈ root.nodes do
        if HasChildren(node) ∧ HasNoGrandchildren(node) ∧ node.ChildrenIdle() then
            RenderReprojection(node)
            ComputeBandingAndSetFlag(node)
            checkedParents = checkedParents ⋃ node
        end if
    end for
end while
```

Although we are now able to build the shadow maps iteratively depending on their needs, we still need to introduce a stalling barrier before starting to render the next frame, as the CPU must wait for any banding tests to complete before issuing the next frame's shadow passes. While physics calculations and gameplay loop advancements can be performed prior to that, having to include this barrier is still a considerable downgrade of the rendering engine.

6.4 Performance of Adaptive Light Source Sampling

Assessing the effectiveness of the shadow reprojection algorithm is not a trivial endeavor, since there are many variables at play, such as the scene configuration, the resolution of the shadow atlas, the subdivision of the shadow atlas, the size of the light, and the size of the reprojection texture being rendered. To keep things as simple as possible, we set up a test using values that have been effective in previous tests. We set the shadow atlas resolution at 2048×2048 and subdivide it into tiles of size 256×256. This way, at most 64 shadows can be rendered at once. We then configure a scene where the camera is far away from the light source and moves closer while we analyze the rendering time of the shadow map rendering, the time required to draw the reprojection passes, and the overall viewport render time. We want to answer the question of whether it is better to render the shadows using a medium shadow count at all times (again, we animate the light source such that shadows are forced to redraw at all times) or use our reprojection algorithm to render few shadow maps when we are at a distance, and a high number of shadow maps when we are close to the light source.

To make this comparison, we need to find the most effective resolution for the reprojection texture. We first render it at the full resolution of the viewport, then halve the size several times until we have one-sixteenth of the viewport's resolution, and compare the performance using the setup described above. Naturally, performance improves as the size is decreased, yet once the camera is close enough to require the maximum number of shadow map samples, the performance difference becomes less noticeable. For the number of shadow maps to increase the most gradually, we decide on rendering the reprojection at one-eighth of the viewport resolution. The obtained performance metrics are shown in Figure 6.7

We then render the scene with a reprojection pass at one-eighth of the viewport size and observe that just rendering the reprojection pass takes between 3 and 4 milliseconds per frame on average on the GPU. The rendering of the shadow maps themselves also takes around 3 milliseconds. The visual transition from rendering a low number of shadow maps to a higher number is smooth and not noticeable, given enough blur using PCF. We also notice that during all runs, the performance improves slightly toward the end, as the shadow map is filled up and expansion reprojection passes are skipped, resulting in more

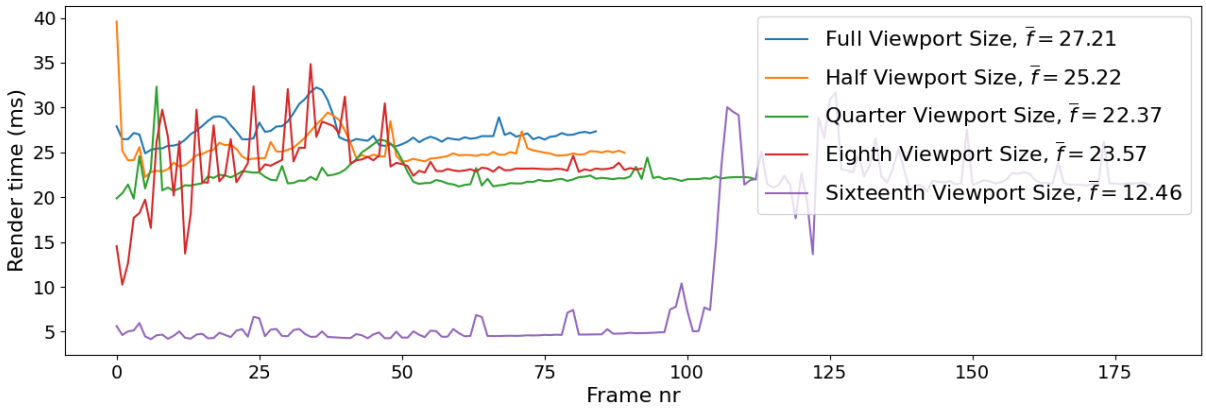


Figure 6.7 Viewport rendering performance depending on the resolution of the texture that the reprojection pass is drawn on. Rendering at one-sixteenth runs the fastest but makes the most drastic switch from minimum to maximum number of shadow maps. At one-eighth, the switch is more gradual.

stable frame rates.

Now disabling the shadow reprojection algorithm, we set the number of rendered shadow maps to 25, which corresponds to a five-by-five grid of samples on the light source, and render the same scene with the camera approaching the light source in the same way. The GPU performance of this run is shown in Figure 6.8. Since the reprojection algorithm causes a varying number of shadow maps ranging from 4 to 63 to be drawn each frame, this comparison is not entirely fair. Thus, we also include a test with a fixed number of 81 shadows but at a slightly smaller size per shadow map.

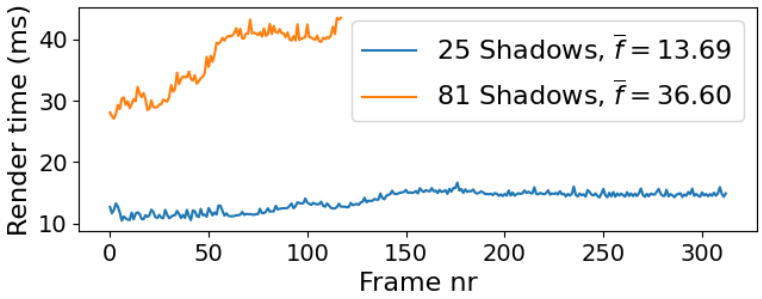


Figure 6.8 Viewport rendering performance without the reprojection algorithm for 25 and 81 shadow map samples

We observe that rendering a constant number of shadow maps leads to a more stable frame rate, which is faster on average when rendering just 25 shadows but slower when rendering 81 shadows than using the reprojection algorithm. This comes as no surprise, as the number of shadow maps rendered by the reprojection pass is somewhere in between during most of the rendering. While the reprojection algorithm might seem viable under this perspective, as it allows for faster rendering at a distance, there are some important considerations we need to make.

6.5 Discussion

Even during the first frames of our test, where only a few shadow maps are being rendered, the reprojection algorithm at its most effective setting performs worse than constantly rendering a medium number of shadows. This is in part because computation time is spent on testing for augmenting the number of shadow map samples with the reprojection passes, most of which do not contribute to an immediate improvement of the image. The algorithm performs substantially better only when the reprojection texture size is reduced to a point where only a minimum number of shadows is rendered. In that case, however, shadows are supposedly a minor detail to the image that little of the computational budget would be spent on in any case. Once they pass to be an important feature of the image, the rendering time jumps at once, defeating the purpose of the algorithm that is being adaptive to the requirements of the scene in a subtle way.

Another drawback of the algorithm is that it causes non-negligible slowdown on the CPU as well, as the maintenance of the quadtree and the conditional issuing of reprojection and compute passes do not come without a cost, and before a new frame can be started, the CPU also needs to wait on the GPU to finish its tasks.

Finally, the algorithm proposed by Schwärzler et al. [9] and our variation of it are complex to implement in practice. For our demonstration, we had to heavily alter the rendering pipeline of the graphics engine by adding a new shader variant and introducing a stalling barrier on the CPU. In the context of an open-source game engine, this results in a burden on the developers, as the required maintenance effort is inevitably increased. Even shader compile times are affected due to our newly introduced shader variants. While this might be an acceptable cost for a feature that is central to the engine's purpose, the addition of a minor optimization would likely encounter fierce opposition under such circumstances.

For this reason, we consider the introduction of cascades to the shadow maps, proposed by Zerari et al. [61], to not be a viable path to improving the shadow rendering and would instead point to other kinds of algorithms, such as the proposal of Heitz et al. [62], and suggest further research in the ways we generate, interpret, and use shadow maps, to improve soft shadow rendering in real-time, as there has been research, such as by Peters et al. [24], that still shows potential in this direction.

We would also like to point out that the shadow subdivision algorithm can be approximated more efficiently by adding a heuristic metric that is analytically computable on the CPU and estimates the number of shadow samples to be rendered. Depending on the distance of the camera to a light source and the field of view of the camera, shadow map samples could be increased or decreased dynamically. Though employing the technique in games would require level designers or lighting artists to actively adjust the heuristic to fit their needs, it could yield a far more effective soft shadow solution.

Another problem to solve is how tiles on the shadow atlas for area lights should be handled in the presence of multiple contending area lights. Since the lights are competing for resources, which, in this case, are tiles, we should ensure *fairness* in the concessions. This would involve ensuring that the light causing the most noticeable banding in the current scene configuration can claim more tiles, thereby avoiding cases of *starvation*, where the light is unable to claim any tiles over long periods of time. In the case where another area light has already claimed the remaining available tiles but is significantly better subdivided, the former area light should be able to steal or *evict* tiles from the other. However, we also need to prevent *pathologic eviction*, where two or more area lights competing for tiles could end up constantly evicting each other, potentially leading to unnecessary redraws of the shadow maps. While suggesting a resource-sharing algorithm dividing the shadow atlas among a number of area lights would be a reasonable endeavor, we will, for now, leave this topic for future work should interest in the reprojection technique ever reemerge.

7 Conclusion

Area lights and soft shadows can enhance the visual fidelity of a 3D render, yet real-time algorithms for rendering them can be far more complex than those used for rendering point lights and hard shadows. Choosing the right algorithm for implementation in a real-time rendering engine thus involves careful consideration of many factors, such as the quality of specular reflections, compatibility with diffuse and specular lighting models, their runtime performance, and implementation complexity.

While we did not implement a complete area light solution, we provided a basis for a subsequent pull request to the Godot engine as we explored how an area light node would have to interface with different engine components and which algorithms are feasible to implement, that provide both visual quality, not impair performance, and keep alterations to the engines source code and increase of maintenance effort at a minimum.

In terms of shading, we found the most representative point technique (MRP) to be the cheapest and least complex to integrate; however, it has some quality issues, such as the lack of Fresnel reflections and artifacts in specular reflections at certain rotations. These artifacts are avoided by an algorithm based on linearly transformed cosine distributions (LTC), which comes at a slightly higher performance cost and higher but still feasible integration efforts. Stochastic shading based on Monte Carlo integration by taking many light samples exhibits no artifacts but considerable noise at low sampling rates, making it unsuitable for real-time implementation. While the MRP and stochastic techniques allow for reusing shading code used for point lights, this is not the case for LTC, and material effects such as anisotropy have to be implemented using separate techniques. Nevertheless, we recommend the implementation of an LTC shading algorithm for area lights due to its accuracy and ongoing research focusing on the technique. The features of all discussed shading algorithms are summarized in Table 7.1.

	Stochastic	**MRP**	**LTC**
Accurate Specular Reflection	✓	✓	✓
Accurate Fresnel Effect	✗	✗	✓
Free of Artifacts	✓	✗	✓
Performance Rating	slow	very fast	fast
Code reusability	high	high	low

Table 7.1 Summary of features of the discussed shading algorithms

We implemented a stochastic soft shadow solution based on shadow maps, which can produce results close to the path-traced ground truth, as demonstrated by the LPIPS difference images we generated. It requires adding a new shadow atlas and expanding the uniform buffer object to include an array of indices for the light. We evaluated the

optimization suggested by Schwärzler et al. [9] and proposed a variant that works in a rendering engine employing a rendering graph, where we are given less control over the interplay between operations carried out on the CPU and the GPU.

While being functional, we found that the use of this technique is less suitable for general purpose game engines, as it requires altering the engine architecture in ways that are not insignificant even to users who do not work with area lights. Thus, we propose a manual setting for the number of shadow maps rendered for an area light, allowing level designers and lighting artists to control it by a simple heuristic.

To avoid creating a new shadow atlas, we also want to suggest inscribing a lower number of shadow maps, potentially just one per corner of the light shape, within a single tile of the shadow atlas used for punctual lights. Although this has not been tested in the context of this thesis, it may be worth investigating further. Using percentage-closer soft shadows, smooth penumbrae is believed to be achievable within reasonable computational bounds.

Despite the dominance of resource-intensive techniques like ray tracing and machine learning algorithms in the current research landscape, we believe that further research on soft shadows should still be carried out using shadow maps to enable improved shadow quality on lower-end devices that cannot make use of expensive GPUs. We would like to emphasize that predictable runtime performance and simple integration are paramount to the successful adoption of algorithms, meaning we shall strive to achieve the best results using a low and preferably constant number of shadow maps per light.

References

[1] J. Gregory, *Game Engine Architecture*, Third Edition. Taylor & Francis, 2018, ISBN: 978-1-138-03545-4.

[2] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering. From Theory to Implementation*, Fourth Edition. MIT Press, Mar. 2023, ISBN: 978-0-262-04802-6.

[3] M. Toftedahl and H. Engström, "A Taxonomy of Game Engines and the Tools that Drive the Industry," in *Proceedings of DiGRA 2019 Conference: Game, Play and the Emerging Ludo-Mix*, Kyoto, Japan: DiGRA, 2019.

[4] J. Linietsky, A. Manzur, and contributors, *Godot Engine - Free and open source 2D and 3D game engine*. Accessed: Mar. 29, 2025. [Online]. Available: `https://godotengine.org/`.

[5] Godot Engine, *godot*, GitHub repository. Accessed: Mar. 29, 2025. [Online]. Available: `https://github.com/godotengine/godot`.

[6] M. Drobot, "Physically Based Area Lights," in *GPU Pro 5: Advanced Rendering Techniques*, Taylor & Francis, 2014, pp. 67–100, ISBN: 978-1-4822-0863-4.

[7] E. Heitz, J. Dupuy, S. Hill, and D. Neubelt, "Real-Time Polygonal-Light Shading with Linearly Transformed Cosines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, Jul. 2016. DOI: `10.1145/2897824.2925895`.

[8] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering Antialiased Shadows with Depth Maps," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87, New York, NY, USA: Association for Computing Machinery, 1987, pp. 283–291. DOI: `10.1145/37401.37435`.

[9] M. Schwärzler, O. Mattausch, D. Scherzer, and M. Wimmer, "Fast Accurate Soft Shadows with Adaptive Light Source Sampling," in *Proceedings of the 17th International Workshop on Vision, Modeling, and Visualization (VMV 2012)*, Magdeburg, Germany: Eurographics Association, Nov. 2012, pp. 39–46.

[10] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of Computer Graphics*, Third Edition. CRC Press, Jul. 2009, ISBN: 978-1-4398-6552-1.

[11] 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, Grenoble, France: Eurographics Association, 2007, pp. 195–206.

[12] R. Guy and M. Agopian, *Physically Based Rendering in Filament*, Google, 2019. Accessed: Jan. 12, 2025. [Online]. Available: `https://google.github.io/filament/Filament.html`.

[13] R. L. Cook and K. E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Trans. Graph.*, vol. 1, no. 1, pp. 7–24, 1982, ISSN: 0730-0301. DOI: 10.1145/357290.357293.

[14] E. J. Hammon, *PBR Diffuse Lighting for GGX+Smith Microsurfaces*, San Francisco, CA, USA: Game Developers Conference (GDC), 2017. Accessed: Jun. 10, 2025. [Online]. Available: https://gdcvault.com/play/1024478/PBR-Diffuse-Lighting-for-GGX.

[15] C. Schlick, "An Inexpensive BRDF Model for Physically-based Rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994. DOI: https://doi.org/10.1111/1467-8659.1330233.

[16] B. Smith, "Geometrical Shadowing of a Random Rough Surface," *IEEE Transactions on Antennas and Propagation*, vol. 15, no. 5, pp. 668–671, 1967. DOI: 10.1109/TAP.1967.1138991.

[17] J. H. Lambert, *Photometria, sive de mensura et gradibus luminus, colorum et umbrae*. Augsburg: Eberhard Klett, 1760.

[18] B. Burley and Walt Disney Animation Studios, "Physically Based Shading at Disney," in *ACM SIGGRAPH 2012 Courses*, vol. 2012, vol. 2012, 2012, pp. 1–7.

[19] F. C. Crow, "Shadow Algorithms for Computer Graphics," *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 1977.

[20] L. Williams, "Casting Curved Shadows on Curved Surfaces," in *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '78, Atlanta, GA, USA: Association for Computing Machinery, 1978, pp. 270–274. DOI: 10.1145/800248.807402.

[21] J. Linietsky, A. Manzur, and the Godot community, *Godot Docs - 4.3 branch – Godot Engine 4.3 documentation in English*. Accessed: Mar. 29, 2025. [Online]. Available: https://docs.godotengine.org/en/4.3/.

[22] C. Peters, "Moment Shadow Mapping," M.S. thesis, University of Bonn, 2013.

[23] M. Hecher, M. Bernhard, O. Mattausch, D. Scherzer, and M. Wimmer, "A Comparative Perceptual Study of Soft-Shadow Algorithms," *ACM Trans. Appl. Percept.*, vol. 11, no. 2, Jun. 2014. DOI: 10.1145/2620029.

[24] C. Peters, C. Münstermann, N. Wetzstein, and R. Klein, "Improved Moment Shadow Maps for Translucent Occluders, Soft Shadows and Single Scattering," *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 1, pp. 17–67, 2017.

[25] Lindbeck, Isak, "Rendering Light and Shadows for Transparent Objects," M.S. thesis, Lund University, 2015.

[26] U. Technologies, *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine*. Accessed: Apr. 7, 2025. [Online]. Available: https://unity.com.

[27] Epic Games, Inc, *Unreal engine*. Accessed: Apr. 7, 2025. [Online]. Available: https://www.unrealengine.com/en-US/home.

[28] N. Gomez, *Godot Engine, Motor de Videojuegos Open Source*, Sep. 2017. Accessed: Apr. 7, 2025. [Online]. Available: https://www.headsem.com/godot-engine-el-motor-de-videojuegos-open-source-mas-completo/.

[29] GitHub, *Collection: Game Engines*. Accessed: Apr. 7, 2025. [Online]. Available: https://github.com/collections/game-engines.

[30] Itch.io, *Most used Game Engines*. Accessed: Apr. 7, 2025. [Online]. Available: https://itch.io/game-development/engines/most-projects.

[31] O. Olsson, M. Billeter, and U. Assarsson, "Clustered Deferred and Forward Shading," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, ser. EGGH-HPG'12, Paris, France: Eurographics Association, 2012, pp. 87–96.

[32] D. Banini, *GPU synchronization in Godot 4.3 is getting a major upgrade*, Godot Engine, 2024. [Online]. Available: https://godotengine.org/article/rendering-acyclic-graph/.

[33] W. Scacchi, "Free/Open Source Software Development: Recent Research Results and Emerging Opportunities," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 459–468. DOI: 10.1145/1287624.1287689.

[34] P. Shirley, C. Wang, and K. Zimmerman, "Monte Carlo Techniques for Direct Lighting Calculations," *ACM Trans. Graph.*, vol. 15, no. 1, pp. 1–36, Jan. 1996. DOI: 10.1145/226150.226151.

[35] C. Ureña, M. Fajardo, and A. King, "An Area-Preserving Parametrization for Spherical Rectangles," *Computer Graphics Forum*, vol. 32, no. 4, pp. 59–66, 2013. DOI: 10.1111/cgf.12151.

[36] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 39, no. 4, Jul. 2020. DOI: 10/gg8xc7.

[37] G. Nichols and C. Wyman, "Direct Illumination from Dynamic Area Lights," in *SIGGRAPH '09: Posters*, ser. SIGGRAPH '09, New Orleans, Louisiana: Association for Computing Machinery, 2009. DOI: 10.1145/1599301.1599383.

[38] K. Schwenk, "Real-Time Rendering of Dynamic Area and Volume Lights Using Hierarchical Irradiance Volumes," in *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*, Athens, Greece, 2011, pp. 136–139. DOI: 10.1109/VS-GAMES.2011.26.

[39] K. Picott, "Extensions of the Linear and Area Lighting Models," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, pp. 31–38, 1992. DOI: 10.1109/38.124286.

[40] B. Karis, "Real Shading in Unreal Engine 4," in *ACM SIGGRAPH 2013 Courses*, ser. SIGGRAPH '13, Anaheim, California: Association for Computing Machinery, 2013. DOI: 10.1145/2504435.2504457.

[41] S. Lagarde and C. de Rousiers, "Moving Frostbite to Physically Based Rendering," in *ACM SIGGRAPH 2014 Courses*, ser. SIGGRAPH '14, Vancouver, Canada: Association for Computing Machinery, 2014. Accessed: May 17, 2025. [Online]. Available: https://www.slideshare.net/slideshow/moving-frostbite-to-physically-based-rendering/41070721.

[42] P. Sikachev, G. D. De Francesco, K. Nowakowski, and K. Kowalczyk, "Area Light Sources in Cyberpunk 2077," in *ACM SIGGRAPH 2021 Talks*, ser. SIGGRAPH '21, Virtual Event, USA: Association for Computing Machinery, 2021. DOI: 10.1145/3450623.3464630.

[43] P. Lecocq, G. Sourimant, and J.-E. Marvie, "Accurate Analytic Approximations For Real-Time Specular Area Lighting," in *ACM SIGGRAPH 2015 Talks*, ser. SIGGRAPH '15, Los Angeles, California: Association for Computing Machinery, 2015. DOI: 10.1145/2775280.2792522.

[44] E. Heitz and S. Hill, "Real-Time Line and Disk Light Shading," in *ACM SIGGRAPH 2017 Courses*, ser. SIGGRAPH '17, Los Angeles, California: Association for Computing Machinery, 2017. DOI: 10.1145/3084873.3084893.

[45] C. Luksch, L. Prost, and M. Wimmer, "Real-time Approximation of Photometric Polygonal Lights," *Proceddings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 1, May 2020. DOI: 10.1145/3384537.

[46] C. Tao, J. Guo, C. Gong, B. Wang, and Y. Guo, "Real-Time Antialiased Area Lighting Using Multi-Scale Linearly Transformed Cosines," *Pacific Graphics*, 2021. DOI: 10.2312/pg.20211380.

[47] T. Kuge, T. Yatagawa, and S. Morishima, "Real-Time Shading with Free-Form Planar Area Lights Using Linearly Transformed Cosines," *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 1, pp. 1–16, 2022.

[48] J. Wang and R. Ramamoorthi, "Analytic Spherical Harmonic Coefficients for Polygonal Area Lights," *ACM Trans. Graph.*, vol. 37, no. 4, Jul. 2018. DOI: 10.1145/3197517.3201291.

[49] L. Wu, G. Cai, S. Zhao, and R. Ramamoorthi, "Analytic Spherical Harmonic Gradients for Real-Time Rendering with Many Polygonal Area Lights," *ACM Trans. Graph.*, vol. 39, no. 4, Aug. 2020. DOI: 10.1145/3386569.3392373.

[50] P. Mézières and M. Paulin, "Efficient spherical harmonic shading for separable BRDF," in *SIGGRAPH Asia 2021 Technical Communications*, ser. SA '21, Tokyo, Japan: Association for Computing Machinery, 2021. DOI: 10.1145/3478512.3488597.

[51] P. Mézières, N. Mellado, L. Barthe, and M. Paulin, "Recursive analytic spherical harmonics gradient for spherical lights," *Computer Graphics Forum*, vol. 41, no. 2, pp. 393–406, 2022. DOI: 10.1111/cgf.14482.

[52] Z. Liu, Y. Huo, Y. Yang, J. Chen, and R. Wang, "Real-Time Polygonal Lighting of Iridescence Effect using Precomputed Monomial-Gaussians," *Computer Graphics Forum*, vol. 43, no. 6, e14991, 2024. DOI: 10.1111/cgf.14991.

[53] N. Chin and S. Feiner, "Fast Object-Precision Shadow Generation for Area Light Sources Using BSP Trees," in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, ser. I3D '92, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1992, pp. 21–30. DOI: 10.1145/147156.147159.

[54] Ars Technica, *John Carmack: "This sucks."* Jul. 2004. Accessed: Jun. 11, 2025. [Online]. Available: https://arstechnica.com/uncategorized/2004/07/4048-2/.

[55] U. Assarsson, "A Real-Time Soft Shadow Volume Algorithm," Ph.D. dissertation, Chalmers University of Technology, Göteborg, Sweden, 2003.

[56] R. Fernando, "Percentage-Closer Soft Shadows," in *ACM SIGGRAPH 2005 Sketches*, ser. SIGGRAPH '05, Los Angeles, California: Association for Computing Machinery, 2005, p. 35. DOI: 10.1145/1187112.1187153.

[57] L. Atty, N. Holzschuch, M. Lapierre, J.-M. Hasenfratz, C. Hansen, and F. X. Sillion, "Soft Shadow Maps: Efficient Sampling of Light Source Visibility," *Computer Graphics Forum*, vol. 25, no. 4, 2006. DOI: 10.1111/j.1467-8659.2006.00995.x.

[58] H. H. Ali, H. Kolivand, and M. S. Sunar, "Soft bilateral filtering shadows using multiple image-based algorithms," *Multimedia Tools and Applications*, vol. 76, no. 2, pp. 2591–2608, Jan. 2017. DOI: 10.1007/s11042-016-3254-0.

[59] G. Nichols, R. Penmatsa, and C. Wyman, "Direct Illumination from Dynamic Area Lights With Visibility," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10, Washington, D.C.: Association for Computing Machinery, 2010. DOI: 10.1145/1730804.1730993.

[60] P. S. Heckbert and M. Herf, *Simulating Soft Shadows with Graphics Hardware*. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, 1997.

[61] A. E. M. Zerari, N. Azri, N. M. Mohamed, and C. B. Mohamed, "Accurate Approximation of Soft Shadows for Real-Time Rendering," *Revue d'Intelligence Artificielle*, vol. 37, no. 6, pp. 1493–1502, Dec. 2023.

[62] 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, Montreal, Quebec, Canada: Association for Computing Machinery, 2018. DOI: 10.1145/3190834.3190852.

[63] H. Okuno and K. Iwasaki, "Binary space partitioning visibility tree for polygonal and environment light rendering," *The Visual Computer*, vol. 37, no. 9, pp. 2499–2511, Sep. 2021. DOI: 10.1007/s00371-021-02181-8.

[64] dfranx, *SHADERed - Free and open source shader editor*. Accessed: Jun. 7, 2025. [Online]. Available: https://shadered.org/.

[65] Blender Foundation, *Blender.org - Home of the Blender project - Free and Open 3D Creation Software*. Accessed: Mar. 6, 2025. [Online]. Available: https://blender.org.

[66] Blender Documentation Team, *Color Management - Blender 4.4 Manual*. Accessed: Jun. 7, 2025. [Online]. Available: https://docs.blender.org/manual/en/latest/render/color_management.html.

[67] J. Turánszki, *Area Lights – Wicked Engine*, 2017. Accessed: Apr. 9, 2025. [Online]. Available: https://wickedengine.net/2017/09/area-lights/.

[68] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, "FLIP: A Difference Evaluator for Alternating Images," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 3, no. 2, Aug. 2020. DOI: 10.1145/3406183.

[69] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Salt Lake City, UT, USA, 2018. DOI: 10.48550/arXiv.1801.03924.

[70] A. KT, E. Heitz, J. Dupuy, and P. J. Narayanan, "Bringing Linearly Transformed Cosines to Anisotropic GGX," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 5, no. 1, pp. 1–18, May 2022. DOI: 10.1145/3522612.

Appendix A Implementation Details and Digital Artifacts

Repository link

The accompanying source code and executable examples for this project can be found at the following link:
`https://github.com/CookieBadger/master-thesis-artifacts`

Summary of Implementation

To enable easier navigation of the codebase and our changes to it, we provide some simplified code samples and a basic description of our additions and modifications to the rendering engine for enabling an area light node.

To add a new light source to the Godot engine for use as a node in a scene tree, we create a new class `AreaLight3D` inside `light_3d.h`, the common header file for all light nodes. Similar to `SpotLight3D` and `OmniLight3D`, we have the class extend `Light3D`, a node that provides various basic parameters to its extending classes, e.g., the light energy or color. We add the parameters specific to the area light, including width and height, range, and an attenuation value specifying how the light falls off over distance. Additional parameters specific to techniques, such as the number of shadow samples are also added here. We bind those parameters to the scripting API using the macro `ADD_PROPERTYI` and register the light in the scripting API using the macro `GDREGISTER_CLASS` in `register_scene_types.cpp`. All parameters, including those inherited from `Light3D`, are then accessible via scripts. In the `RenderingServer` base class, we also bind a method to create a new light source. This is summarized in Listing A.1.

Respective entries in the `LightType` and `LightParam` enums are required for the light to be forwarded to the `RenderingServer`. We add a new case in the method `_prepare_light()` of the class `RenderingLightCuller`, computing the radius of the bounding sphere for the culling to then decide whether it intersects the view frustum, or whether the light can be discarded entirely.

The class `LightStorage` is responsible for holding the data of lights and shadows and binding it to the uniforms before a render pass. When adding the newly added area light parameters to the `LightData` uniform buffer object, we need to make sure to maintain a strict alignment of 16 bytes, such that, e.g., a four-dimensional vector is stored at a memory address divisible by 16. In the class `RenderForwardClustered`, we retrieve the resource identifier (RID) of the light's uniform buffer from the `LightStorage` class

and bind it to a new uniform. Respective cases for updating the buffers are added in its `update_light_buffers()` method.

```
// light_3d.cpp
AABB Light3D::get_aabb() const {
  if (type == RenderingServer::LIGHT_AREA) {
    float len = param[PARAM_RANGE];
    float size_a = param[PARAM_AREA_SIDE_A] / 2.0 + len;
    float size_b = param[PARAM_AREA_SIDE_B] / 2.0 + len;
    return AABB(-Vector3(size_a, size_b, len), Vector3(size_a * 2, size_b *
    2, len));
  }
  else { /* handle other light types */ }
}
void AreaLight3D::_bind_methods() {
  ADD_GROUP("Area", "area_");
  ADD_PROPERTYI(PropertyInfo(Variant::FLOAT, "area_range",
   PROPERTY_HINT_RANGE, "0,4096,0.001,or_greater,exp,suffix:m"), "set_param
   ", "get_param", PARAM_RANGE);
  ADD_PROPERTYI(PropertyInfo(Variant::FLOAT, "area_attenuation",
   PROPERTY_HINT_RANGE, "-10,10,0.001,or_greater,or_less"), "set_param", "
   get_param", PARAM_ATTENUATION);
  ADD_PROPERTYI(PropertyInfo(Variant::FLOAT, "area_side_a",
   PROPERTY_HINT_RANGE, "0,4096,0.001,or_greater,exp,suffix:m"), "set_param
   ", "get_param", PARAM_AREA_SIDE_A);
  ADD_PROPERTYI(PropertyInfo(Variant::FLOAT, "area_side_b",
   PROPERTY_HINT_RANGE, "0,4096,0.001,or_greater,exp,suffix:m"), "set_param
   ", "get_param", PARAM_AREA_SIDE_B);
  ADD_PROPERTYI(PropertyInfo(Variant::INT, "area_shadow_sample_resolution",
    PROPERTY_HINT_RANGE, "1,32"), "set_param", "get_param",
   PARAM_AREA_SHADOW_SAMPLE_RESOLUTION);
}

// register_scene_types.cpp
void register_scene_types() {
  /* Countless other classes are registered here */
  GDREGISTER_CLASS(AreaLight3D);
}

// rendering_server.cpp
void RenderingServer::_bind_methods() {
    /* bind other methods and constants */
    ClassDB::bind_method(D_METHOD("area_light_create"), &RenderingServer::
   area_light_create);
}
```

Listing A.1 Code for setting up and registering an area light

The uber shader in the file `scene_forward_clustered.glsl` contains the illumination calculations, where we add a uniform buffer for our area lights and a loop that iterates over them. The code for setting up the uniform buffers is shown in Listing A.2.

```
// light_data_inc.glsl -> included in scene_forward_clustered.glsl
layout(set = 0, binding = 5, std430) restrict readonly buffer AreaLights {
  LightData data[];
} area_lights;

// render_forward_clustered.h
void RenderForwardClustered::_update_render_base_uniform_set() {
    /* add other uniforms */
    RD::Uniform u;
    u.binding = 5;
    u.uniform_type = RD::UNIFORM_TYPE_STORAGE_BUFFER;
    u.append_id(RendererRD::LightStorage::get_singleton()->
    get_area_light_buffer());
    uniforms.push_back(u);
}
```

Listing A.2 Code for setting up the uniform buffer for our new light source

Strongly simplified code for looping over all area lights and calculating their illumination and shadows in the shader is included in Listing A.3. The contents of the methods `light_process_area()` and `light_process_area_shadow()` depend on the employed technique.

```
// scene_forward_clustered.glsl
for (uint i = 0; i < lights_affecting_cluster_count; i++) {
  float shadow = light_process_area_shadow(lights_affecting_cluster[i],
   vertex, normal);
  light_process_area(lights_affecting_cluster[i], vertex, view, normal,
   material, out_diffuse_light, out_specular_light);
  total_diffuse_light += out_diffuse_light * shadow;
  total_specular_light += out_specular_light * shadow;
}
```

Listing A.3 Code for looping over all active area lights (simplified)

To account for area lights in the cluster buffer, we add a new entry to the enums `LightType` and `ElementType` in the `ClusterBuilderRD` class and add respective cases in the method `add_light()` and `bake_cluster()`, binding vertices for the light's bounding volume to the draw list. Adding a debug view for area light clusters in the class `RenderForwardClustered` additionally helps us make sure the implementation is correct.

To add gizmos to the editor, we change the file `light_3d_gizmo_plugin.cpp`, calculating the position of the mouse cursor (in world space, obtained by intersecting

a camera ray with the view plane) in light local coordinates through the light source's inverse transform. The code is provided below in Listing A.4.

For our shadow implementations to work, we have to add a new shadow atlas and perform culling operations in the `renderer_scene_cull.h` file. A case for area lights in the method `_light_instance_update_shadow()` is added, computing the bounding volume for our area light, and used for culling any visual instances that could potentially cast shadows but lie outside this volume, and therefore do not have to be drawn on the shadow atlas. We append all relevant instances to an array and update the transformation matrix of the light source such that it can be used in shadow computations in the shader.

Methods to manipulate shadow atlas parameters such as its size for each viewport need to be bound in the `RenderingServer` base class and extending classes, and an `AreaShadowAtlas` struct has to be created in the `LightStorage` class, together with an array to hold all atlases and the references to the viewports that own them. Methods to create the texture and framebuffer that the depth values are rendered to, to update and free the atlas, to invalidate shadow maps on it, and to find empty tiles for a light to write its maps on are implemented, and methods for the area shadow atlas API added. References to the shadow atlas need to be passed through numerous methods in the entire rendering pipeline. In the shader, a texture uniform for the atlas is added, which is set up in the render pass uniform set in the `RenderForwardClustered` class. In its `_pre_opaque_render()` method we iterate over each shadow sample of an area light that is marked dirty for each area light that was not culled, and call the `_render_shadow_pass()` method. There, a case is added for area lights to set up all parameters for a subsequent call to the `_render_shadow_append()` method.

For details on the implementations, the reader is advised to check the source repository linked at the beginning of this appendix.

```cpp
// light_3d_gizmo_plugin.cpp
void Light3DGizmoPlugin::set_handle(EditorNode3DGizmo *p_gizmo, int
    p_gizmo_id, Plane p_camera_plane, Vector3 p_mouse_pos_world) {
    Transform3D gi = light->get_global_transform().inverse();
    if(p_gizmo_id == 0) {
        Vector3 inv = gi.transform(p_mouse_pos_world);
        if (inv.x >= 0) {
            light->set_param(PARAM_AREA_SIDE_A, inv.x);
        }
    }
    else if(p_gizmo_id == 1) {
        Vector3 inv = gi.transform(p_mouse_pos_world);
    if (inv.y >= 0) {
        light->set_param(PARAM_AREA_SIDE_B, inv.y);
        }
    }
}

void Light3DGizmoPlugin::redraw(EditorNode3DGizmo *p_gizmo) {
    if (p_gizmo->is_selected()) {
        float a = light->get_param(Light3D::PARAM_AREA_SIDE_A);
        float b = light->get_param(Light3D::PARAM_AREA_SIDE_B);
        // Draw rectangle
        Vector<Vector3> points;
        points.push_back(Vector3(-a/2, b/2, 0));
        points.push_back(Vector3(a/2, b/2, 0));
        points.push_back(Vector3(a/2, b/2, 0));
        points.push_back(Vector3(a/2, -b/2, 0));
        points.push_back(Vector3(a/2, -b/2, 0));
        points.push_back(Vector3(-a/2, -b/2, 0));
        points.push_back(Vector3(-a/2, -b/2, 0));
        points.push_back(Vector3(-a/2, b/2, 0));
        p_gizmo->add_lines(points);
        // Create handles
        Vector<Vector3> handles = {
            Vector3(a/2, 0, 0),
            Vector3(0, b/2, 0)
        };
        p_gizmo->add_handles(handles, get_material("handles"));
    }
    const Ref<Material> icon = get_material("light_area_icon", p_gizmo);
    p_gizmo->add_unscaled_billboard(icon);
}
```

Listing A.4 Implementation for area light gizmo handles (simplified)

Appendix B Scenes used in Tests

Sponza

Source:	Hugo Locurcio, Sponza demo for Godot 4 (GitHub)
	`https://github.com/Calinou/godot-sponza`
Original Creator(s):	Crytek, Frank Meinl
	`https://digitalwerk.artstation.com/projects/K5bEr`

Bistro

Source:	John James Gutib, Bistro-Demo-Tweaked (GitHub)
	`https://github.com/Jamsers/Bistro-Demo-Tweaked`
Original Creator(s):	Amazon Lumberyard
	Open Research Content Archive (ORCA), 2017.
	`http://developer.nvidia.com/orca/amazon-lumberyard-bistro`

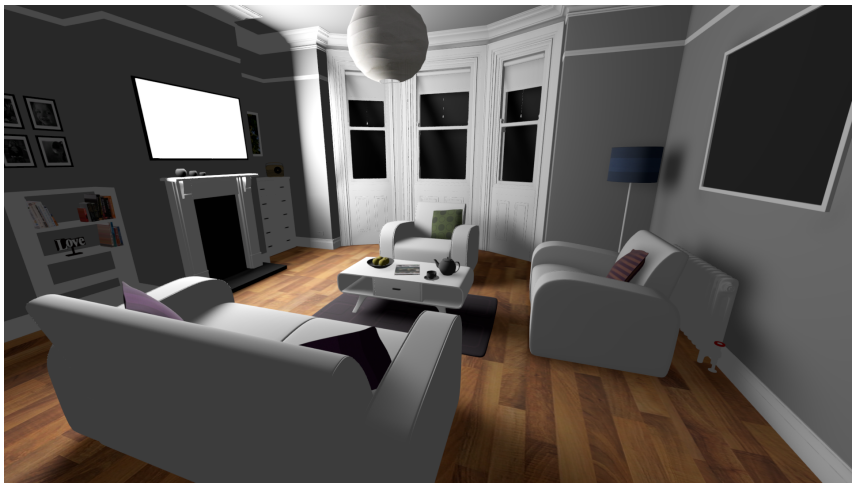
TPS Demo

Source:	Godot Engine, Third Person Shooter Demo (GitHub)
	`https://github.com/godotengine/tps-demo`
Image Source:	Ibid.
Original Creator(s):	Juan Linietsky, Fernando Miguel Calabró

Sibenik Cathedral

Source:	McGuire Computer Graphics Archive
	`https://casual-effects.com/data/`
Original Creator(s):	Marko Dabrovic, Kenzie Lamar, Morgan McGuire

Living Room



Source:	McGuire Computer Graphics Archive
	`https://casual-effects.com/data/`
Original Creator(s):	Jay, Benedikt Bitterli, Nicholas Hull, Morgan McGuire
	`http://www.blendswap.com/user/Jay-Artist`

Appendix C LTC LUT Polynomials

With the following set of equations we approxiate the lookup table for area light shading with linearly transformed cosines (LTC), where $\alpha \in [0;1]$ is the surface roughness parameter, and $\theta \in [0;\pi/2]$ is the angle between the eye vector and the surface normal.

$$x = 1 + \alpha$$
$$y = 1 + \theta$$

$$\begin{aligned} P_a(x,y) = {} & (40.022068) + (-21.786636)y + (-6.711040)y^2 + (7.836000)y^3 \\ & + (-2.238067)y^4 + (0.207919)y^5 + (-113.968298)x + (72.582053)xy \\ & + (-10.971895)xy^2 + (-0.902927)xy^3 + (0.364781)xy^4 + (115.774423)x^2 \\ & + (-59.489534)x^2y + (8.060379)x^2y^2 + (-0.421001)x^2y^3 + (-57.958025)x^3 \\ & + (19.685421)x^3y + (-1.225889)x^3y^2 + (15.368255)x^4 + (-2.398787)x^4y \\ & + (-1.802015)x^5 \end{aligned}$$

$$\begin{aligned} P_b(x,y) = {} & (-32.371729) + (20.622668)y + (-10.246609)y^2 + (5.042130)y^3 \\ & + (-1.433458)y^4 + (0.153338)y^5 + (98.859420)x + (-37.623269)xy \\ & + (5.569236)xy^2 + (-0.449615)xy^3 + (0.097781)xy^4 + (-124.243103)x^2 \\ & + (32.867644)x^2y + (-3.804820)x^2y^2 + (-0.068663)x^2y^3 + (76.969383)x^3 \\ & + (-10.908016)x^3y + (1.064901)x^3y^2 + (-24.185140)x^4 + (0.910078)x^4y + \\ & (3.179056)x^5 \end{aligned}$$

$$\begin{aligned} P_c(x,y) = {} & (11.583158) + (16.195132)y + (-21.187278)y^2 + (7.114936)y^3 \\ & + (-1.288357)y^4 + (0.090490)y^5 + (-52.766376)x + (-2.864469)xy \\ & + (21.805870)xy^2 + (-4.130327)xy^3 + (0.381285)xy^4 + (69.234074)x^2 \\ & + (-16.785586)x^2y + (-8.971769)x^2y^2 + (0.573957)x^2y^3 + (-40.192157)x^3 \\ & + (12.390364)x^3y + (1.546020)x^3y^2 + (11.068497)x^4 + (-2.629684)x^4y \\ & + (-1.169427)x^5 \end{aligned}$$

$$
\begin{aligned}
P_d(x,y) = \; & (56.560977) + (-26.447422)y + (8.007063)y^2 + (-2.938371)y^3 \\
& + (0.778390)y^4 + (-0.079369)y^5 + (-174.029262)x + (55.590827)xy \\
& + (-7.595242)xy^2 + (0.586064)xy^3 + (-0.052755)xy^4 + (218.422343)x^2 \\
& + (-49.772234)x^2y + (4.705369)x^2y^2 + (-0.110559)x^2y^3 + (-135.668807)x^3 \\
& + (18.670509)x^3y + (-1.020345)x^3y^2 + (42.042080)x^4 + (-2.349413)x^4y \\
& + (-5.275442)x^5
\end{aligned}
$$

$$
\begin{aligned}
P_e(x,y) = \; & (38.635232) + (-53.895233)y + (20.158410)y^2 + (-2.038722)y^3 \\
& + (-0.486485)y^4 + (0.086644)y^5 + (-83.242356)x + (110.795093)xy \\
& + (-35.490292)xy^2 + (4.713696)xy^3 + (-0.117706)xy^4 + (66.288968)x^2 \\
& + (-78.463860)x^2y + (16.126788)x^2y^2 + (-1.280718)x^2y^3 + (-23.625243)x^3 \\
& + (24.954272)x^3y + (-2.127836)x^3y^2 + (3.208161)x^4 + (-3.198377)x^4y \\
& + (0.024170)x^5
\end{aligned}
$$

$$
\begin{aligned}
P_{fn} = \; & (31.627915) + (-9.242613)y + (-18.316942)y^2 + (12.243966)y^3 \\
& + (-2.814830)y^4 + (0.224888)y^5 + (-93.286029)x + (63.261605)xy \\
& + (-1.313849)xy^2 + (-4.396929)xy^3 + (0.656442)xy^4 + (88.299669)x^2 \\
& + (-61.632692)x^2y + (7.258499)x^2y^2 + (0.096766)x^2y^3 + (-32.964044)x^3 \\
& + (22.033172)x^3y + (-1.578708)x^3y^2 + (3.228726)x^4 + (-2.787282)x^4y \\
& + (0.427175)x^5
\end{aligned}
$$

$$
\begin{aligned}
P_{fg} = \; & (-0.874912) + (2.405966)y + (0.444356)y^2 + (-3.465719)y^3 \\
& + (0.463044)y^4 + (0.187222)y^5 + (0.660714)x + (-7.715301)xy \\
& + (11.305963)xy^2 + (1.880087)xy^3 + (-1.186281)xy^4 + (3.182991)x^2 \\
& + (-5.075303)x^2y + (-9.942256)x^2y^2 + (1.837908)x^2y^3 + (-0.220191)x^3 \\
& + (9.148312)x^3y + (0.226042)x^3y^2 + (-2.274063)x^4 + (-1.613646)x^4y \\
& + (0.621338)x^5
\end{aligned}
$$