School of Computer Science
FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Endless Cinematic Area Lighting: Scalable Real-Time LTC Polygonal Lights with Clustered Forward Shading

Liam Swarbrick

Submitted in accordance with the requirements for the degree of MEng, BSc Computer Science with High-Performance Graphics and Games Engineering

2024/25

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (01/05/25)
Link to online code repository	URL	Sent to supervisor and assessor (01/05/25)
Link to Demo	URL	Sent to supervisor and assessor (01/05/25), and in Appendix C

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) *Liam Swarbrick*

Summary

Real-time area lighting brings compelling realism to games but remains limited by high computational costs — especially for physically based polygonal lights. While Linearly Transform Cosines (LTCs) enable efficient shading of such lights, they scale poorly in scenes with many light sources due to per-light computation overhead. In this work, we present a scalable, real-time solution using clustered forward rendering with both position and normal-space clustering. While normal cones in clustered shading were previously considered too expensive for practical use, we demonstrate that they can offer significant performance gains when applied to polygonal area lights by enabling aggressive and accurate light culling. Our method preserves high visual fidelity while supporting thousands of polygonal lights at interactive frame rates, making the use of LTC polygonal lights practical for modern real-time applications.

Acknowledgements

I would like to sincerely thank my supervisor, Dr Markus Billeter, for his guidance, expertise, and consistently insightful feedback throughout the course of this project.

I would also like to thank my assessor, Dr Julian Brooks, whose thoughtful critique and suggestions were invaluable in refining and improving this report.

Contents

Chapter 1

Introduction and Background Research

Beautiful real-time lighting is crucial to modern game rendering and directly impacts immersion in fictional worlds. In real life, light sources are never perfect points — they have physical size and emit light across a surface. This is why shadows in nature are soft, and why materials reflect their surroundings in complex ways. While modern games strive for realistic lighting, rendering accurate area lights in real-time remains a significant challenge. Traditional punctual light sources (point lights, spot lights, and directional lights) are computationally efficient but cast hard shadows and produce unnaturally sharp specular highlights due to their infinitesimal size. Unlike punctual lights, area light sources emit across a surface, illuminating scenes in a more familiar way — casting soft shadows, and producing realistic specular reflections.

Despite their computational cost, some modern game engines have started incorporating simpler area light shapes. CD Projekt RED's Cyberpunk 2077 is a state of the art example in real-time rendering, using sphere and capsule shaped area lights for neon business signs, emissive street ads, fluorescent office lighting, and much more (Sikachev et al., 2021). Guerrilla Games used spherical area lighting for the sun in Horizon: Zero Dawn (de Carpentier and Ishiyama, 2017), creating realistic sunset reflections on the water (Figure 1.1). Polygonal area lights are also desirable tools for environment artists, perfect for city billboards, screens, and large flat lights. However, whilst physically based polygonal lights have been viable in real-time since Heitz et al. (2016) with their Linearly Transformed Cosines (LTC) method, modern games are yet to incorporate them, as Sikachev et al. note they are still too computationally expensive to consider using.

A key limitation of LTC-based polygonal lights is poor scalability: each $\mathcal{N}$-gon light requires $\sim 10\mathcal{N}$ times more computation than a single point light. When multiple lights influence a single pixel, performance drops significantly, making real-time rendering infeasible. However, since area lights are inherently local (affecting only a subset of the scene), aggressive light culling would reduce the overall cost.

(a) Capsule area lights in *Cyberpunk 2077*.

(b) Spherical area light in *Horizon: Zero Dawn*.

Figure 1.1: Examples of area lighting in modern real-time games.

This project investigates efficient methods for rendering polygonal area lights in real-time applications. By extending clustered forward shading (Olsson et al., 2012), we aim to make large-scale usage of polygonal area lighting feasible without compromising performance. The proposed solution tightly culls area lights per cluster, ensuring that at any given pixel, only a small subset of relevant lights contribute to shading — keeping performance scalable even in scenes containing thousands of area lights, which typically do not heavily overlap. Special considerations are made to preserve specular highlights — an underexplored and uniquely important aspect of area light culling — essential to avoid graphical artefacts with even slightly glossy materials.

1.0.1 Deliverables

Along with this report, the proposed system has been implemented in a custom glTF 2.0 Physically Based Rendering (PBR) engine, developed in C with the OpenGL 4.6 graphics API for Windows and Linux. glTF 2.0 is a modern 3D scene file format designed for PBR pipelines and modern graphics APIs. A video demonstration (Appendix C) showcases the system in a scene composed of 2000 polygonal lights and sixteen instances of the UE4 Sun Temple, arranged in a grid to simulate a more expansive environment. The Sun Temple model was provided by Epic Games (2017) through the Open Research Content Archive (ORCA).

1.1 Foundational Concepts for Polygonal Area Lights

Modern real-time renderers simulate lighting by calculating how light interacts with surfaces at each pixel — a process called per-pixel lighting. This involves two key steps: first, converting 3D models into pixel fragments (rasterization), and second, evaluating the geometric and material data of each fragment with light sources using GPU-accelerated maths to compute the final pixel colours. The following sections introduce the core models behind these calculations, starting with how surfaces reflect light (BRDFs), how microscopic details affect reflections (microfacet theory), and finally how area lights differ from simpler light sources.

1.1.1 Bidirectional Reflectance Distribution Functions

A surface's appearance depends on how it converts incoming irradiance into outgoing radiance toward the camera — governed by its *Bidirectional Reflectance Distribution Function* (BRDF). Fred Nicodemus (1965) introduced BRDFs when modelling how surfaces scatter incoming radiance in different directions. Notably BRDFs came about from his realisation that the part of an opaque surface that governs its reflectance properties is its microscopic configuration (roughness) rather than the macroscopic configuration (curvature) in which our triangular meshes are defined. In rendering, we use material parameters stored in textures or vertex data to parameterise one BRDF for multiple kinds of materials. For instance, diffuse lighting gives objects their colour, and in asset pipelines this is stored in diffuse (or albedo) textures mapped

onto the 3D models. Other parameters that BRDFs use can also be stored in textures, such as metalness, roughness, emissivity, and ambient occlusion. This is the BRDF mathematically:

$$\rho(\vec{\omega}_i, \vec{\omega}_o) = \frac{dL(\vec{\omega}_o)}{dE(\vec{\omega}_i)} = \frac{dL(\vec{\omega}_o)}{L(\vec{\omega}_i)\cos\theta_i \, d\vec{\omega}_i}$$

In rendering, this simply means: if we shine more light on a surface from direction $\vec{\omega}_i$, how much of that light reflects towards the viewer. The terms express how much the outgoing radiance $L(\vec{\omega}_o)$ is increased by an increase in surface irradiance E via more radiance $L(\vec{\omega}_i)$. Perfectly diffuse reflectors scatter light in all directions uniformly. Assuming an opaque surface, the light must come from above the horizon of the hemisphere (we denote the direction vectors that form the unit hemisphere as Ω^+). The clamped-cosine/Lambertian BRDF is an energy conserving perfectly diffuse reflector:

$$\rho_{\,\text{lambert}}(\vec{\omega}_i, \vec{\omega}_o) = \frac{1}{\pi}, \qquad \vec{\omega}_i, \vec{\omega}_o \in \Omega^+$$

The normalizing factor $\frac{1}{\pi}$ ensures that the incoming irradiance E_i equals the radiant exitance E_o, and this can be computed exactly due to the simplicity of the BRDF's integration over all outgoing light directions. In the case of polygonal light shading, however, we instead integrate the BRDF over the incoming light directions, bounded by the polygon's projection onto the unit sphere — a process detailed in Section 1.1.3. The complexity of this integration depends on the chosen lighting model. In the following section, we explore the physically based models commonly used in modern rendering systems, which account for the complex behaviour of microscopic surface reflections. These models introduce challenges for area light culling, as specular reflections depend more on surface orientation than distance. Unlike diffuse lighting, which can often be culled with simple range-based heuristics, specular highlights peak at specific view-dependent angles, requiring more careful consideration during light assignment.

1.1.2 Microfacet Theory Used in Physically Based Rendering.

At the microscopic scale, each individual photon of visible light is either reflected, refracted, or absorbed by tiny components of the surface. Because of this, we can model BRDFs for rough surfaces by imagining them as composed of many interconnected, mirror-like microfacets (Figure 1.2). Microfacet modelling was first introduced by Torrance and Sparrow (1967) to derive BRDFs that better predicted the empirical readings that showed rough materials had specular peaks at grazing angles. Most microfacet based BRDFs follow the Cook-Torrance model with pluggable functions for F, D, and G (Cook and Torrance, 1982).

$$\rho(\vec{\omega}_o, \vec{\omega}_i) = \frac{F\,D\,G}{4(n \cdot \vec{\omega}_i)(n \cdot \vec{\omega}_o)}$$

The geometry factor G models how the microfacets occlude each other and is derived from masking-shadowing functions (Heitz, 2014), where microfacets mask each other from reflecting

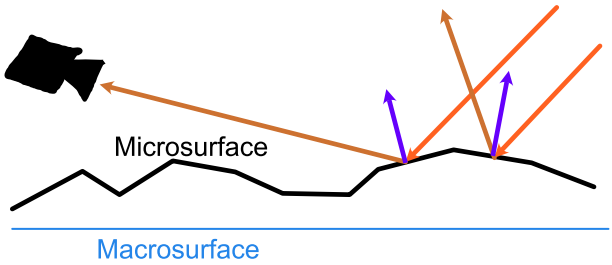

Figure 1.2: Microfacets scatter the incident light in different directions, the microfacet normal distribution describes the proportion of rays that reflect towards the camera.

radiance towards the viewer or shadow each other from receiving incoming light.

The Fresnel term F depends on incidence angle and wavelength, determining light reflection, refraction, and absorption at the medium. The full Fresnel equations are too computationally expensive and so we use Schlick's approximation (Schlick, 1994), which is only a function of incidence angle. Fresnel effects explain increased reflectance of each incident ray at grazing angles. The specular D term, however, is the microfacet normal distribution $D(\vec{m})$, explaining why more microfacet normals align with $\vec{m}$ at grazing angles. Both effects contribute to stronger specular highlights at oblique angles. The specular D term used here is the Trowbridge-Reitz (1975) distribution, also known as GGX (Walter et al., 2007), which is more physically accurate and thus preferred over traditional Phong (1975) and Blinn-Phong (Blinn, 1977) lighting models.

1.1.3 Computing the Irradiance of Area Lights

The contribution of a point light to a fragment simply requires evaluating the surface's BRDF for the single incoming and outgoing directions. Area lights are much more difficult, we cannot evaluate the contribution of the light at any specific direction, instead we must integrate the BRDF over all incoming directions for which the area light is visible from the shading location. For analytic integration we assume the radiance is uniform across the area light, but textured area lights are also possible. For a constant outgoing direction towards the viewer $\vec{\omega}_o$, we can define this integration over all incoming directions from the lights surface P as the following:

$$L(\vec{\omega}_o) = \int_P L(\vec{\omega}_i)\, \rho(\vec{\omega}_i, \vec{\omega}_o)\, \cos\theta\, d\vec{\omega}_i \tag{1.1}$$

Given a polygon with n vertices, the set P of unit vectors that point from the shading location to the n-gon is bounded by the polygon's projection onto the unit sphere around the shading location (Figure 1.3), effectively describing the directions where the polygon is visible. In physically based rendering, we want to calculate integral 1.1 using BRDFs like the Trowbridge-Reitz/GGX distribution ρ_{GGX}. These more complex models have no closed-form integral solutions over the various area light shapes and this is exactly why real-time PBR area lights have been far from straightforward. However, for perfectly diffuse reflectors, the Lambertian's integral over spherical polygons has been known since Lambert (1760):

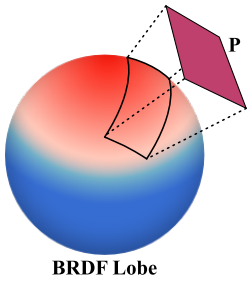

Figure 1.3: The BRDF lobe represents radiance toward the viewer for each incident light direction. Irradiance from a polygonal light is computed by integrating the BRDF over the directions bounded by the corresponding spherical polygon.

$$E_i = I(v_1, \ldots, v_n) = \frac{1}{2\pi} \sum_{i=1}^{n} \arccos(p_i \cdot p_j) \left(\frac{p_i \times p_j}{\|p_i \times p_j\|} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) \tag{1.2}$$

Heitz (2017) is a nice geometric derivation of this formula, and this can be directly expressed in shader code to render area lights in real-time for perfectly diffuse materials. Notably,
Equation 1.2 forms the core component of the LTC-based polygonal light formulation described in Section 2.1.

Chapter 2

Related Work

2.1 Real-Time Physically Based Polygonal Light Shading

Polygonal light shading with physically accurate lighting models like GGX has been tackled in various ways for different rendering pipelines. The most straightforward is Monte-Carlo integration, which approximates the irradiance from an area light by sampling a fixed number of random points on its surface. However, unless the number of samples is huge, the rendered image will exhibit significant noise, preventing this method from being viable for real-time rendering (Lecocq et al., 2017). As hardware raytracing becomes more available to consumers, applications involving real-time raytracing could implement polygonal lights using BRDF importance sampling of a linearly transformed cosine (Peters, 2021). Purely random Monte-Carlo converges too slow, and many lighting effects such as caustics are the result of very improbable chaining of specular reflections (Fan et al., 2023). More recent techniques such as Reservoir-based Spatiotemporal Importance Resampling (ReSTIR) (Bitterli et al., 2020) improve sample reuse across space and time, significantly increasing the efficiency of Monte Carlo sampling.

Historically, Monte-Carlo techniques solely belonged to the domain of offline rendering, which real-time methods have used as a reference to approximate. Notice that point lights are trivial because the BRDF can be evaluated directly at the shading location. We can use this principle for area lights by finding the most representative point light at a given shading location, which is approximated by the closest point of the area light (or intersecting point) to the reflected view ray. Karis (2013) introduced an energy conservation factor to better approximate the brightness of the reference implementation. They provide a simple formulation for spherical area lights, but for non-spherical lights, the point is less straightforward to find. Sikachev et al. (2021) needed to iterate on Karis's original capsule lights for Cyberpunk 2077 with a 100% analytic formulation instead. Representative point light has been used for rectangular lights as well (János, 2017) albeit quite a bit trickier than spheres lights, but for general polygons the formulation becomes much more complicated, especially in energy conservation when polygons clip the horizon. Considering that the LTC method, introduced shortly, not only provides far more accurate results, but has a simpler implementation that automatically handles horizon clipping, we do not consider using this method for polygonal lights, but recognise its prevailing importance in real-time rendering.

Lecocq et al. (2016) proposed analytic approximations for the non-occluded direct illumination of area lights. Their approach extends Arvo's irradiance tensor formulation, originally developed for Phong shading, to support microfacet BRDFs. By reformulating the 1D

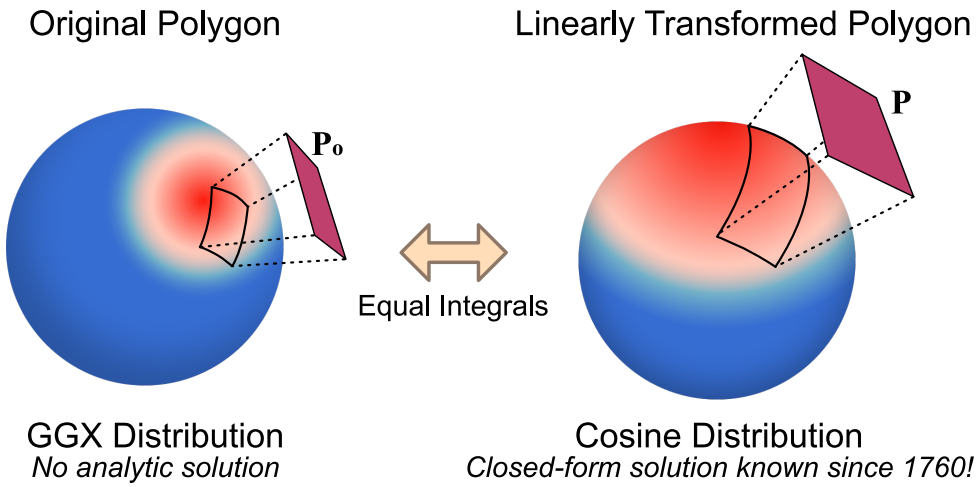

Figure 2.1: Vector graphic reimagination of Fig. 4 from Heitz et al. (2016). A spherical polygon (left) integrated over a GGX BRDF lobe is linearly transformed (right) such that the resulting polygon can be evaluated under the clamped cosine distribution. The two integrals are equivalent.

boundary edge integral with simple peak functions and introducing an edge-splitting strategy to handle spherical warping, they achieve realistic specular reflections in real-time. The framework also supports disc and spherical lights via a novel "polygon spinning" method, allowing flexible area light shapes and roughness levels without sacrificing speed.

The current state of the art PBR polygonal light shading technique by Heitz et al. (2016) introduces a family of spherical distribution called Linearly Transformed Cosines (LTC). This method extends the closed form solution for the clamped cosine BRDF defined in Equation 1.2 to microfacet BRDFs such as GGX by exploiting a clever mathematical equivalence:
For any given GGX configuration (roughness α, view angle θ_v), there exists a linear transformation $\mathbf{M}$ that warps the GGX lobe into an approximately cosine-shaped distribution:

$$\mathrm{GGX}(\vec{\omega}_o; \alpha, \theta_v) \approx \mathrm{cosine}(\mathbf{M} \cdot \vec{\omega}_o)$$

where $\mathbf{M}$ encodes a stretching and shearing to match the GGX lobe's anisotropy and width — Figure 2.1 demonstrates the equivalence of these lobes graphically. Recall that when integrating over the clamped cosine, the direction vectors are defined by a spherical polygon. Thus, if that linear transformation is applied to our spherical polygon then the irradiance of the transformed spherical polygon over the clamped cosine distribution is equal to the irradiance of the original spherical polygon over the GGX distribution. Intuitively, imagine stretching a rubber sheet (GGX lobe) until its shape matches a cosine distribution. The same stretch applied inversely to the light polygon "undoes" the distortion, letting us reuse the simpler cosine solution.

The linear transforms for each roughness and incidence angle are precomputed and stored in a small texture, which eliminates the performance penalty typically associated with using the most accurate BRDF available to us (GGX). LTC-based polygonal lights give highly accurate results with less artefacts than comparable methods like Lecocq et al. (2016), as shown by a comparison in the supplemental material of Heitz et al. (2016).

Finding the linear transformations (which are represented by 3×3 matrices) requires finding

the new polygon such that the two integral over different distributions are equal. Starting with the original GGX distribution $D_o(\vec{\omega}_o)$ over the original spherical polygon P_o. The equivalent integration of the clamped cosine $D(\vec{\omega})$ over transformed polygon $P = M \cdot P_o$ is:

$$\int_P D(\vec{\omega})\, d\vec{\omega} = \int_{P_o} D_o(\vec{\omega}_o)\, d\vec{\omega}_o$$

The left hand side is the clamped cosine integration that we have a closed form of. However, applying a linear transformation alters the differential solid angle $d\vec{\omega}$, meaning we cannot directly substitute $d\vec{\omega} = d\vec{\omega}_o$. Intuitively this is because after we apply a linear transformation (followed by normalization) to an infinitesimal patch on the original spherical polygon, the new infinitesimal patch is not the same size and this affects the result of integration. So instead, we must introduce a change of variables, which involves the Jacobian determinant of the transformation. Specifically, the transformation $\vec{\omega} = M\vec{\omega}_o / \|M\vec{\omega}_o\|$ rescales solid angles, requiring a correction factor:

$$\frac{\partial \vec{\omega}_o}{\partial \vec{\omega}} = \frac{|M^{-1}|}{\|M^{-1}\vec{\omega}\|^3}$$

$$\int_P D(\vec{\omega})\, d\vec{\omega} = \int_{P_o} D_o\left(\frac{M^{-1}\vec{\omega}}{\|M^{-1}\vec{\omega}\|}\right) \cdot \frac{\partial \vec{\omega}_o}{\partial \vec{\omega}}\, d\vec{\omega} = \int_{P_o} D_o(\vec{\omega}_o)\, d\vec{\omega}_o$$

This ensures the integral remains invariant under transformation. The Jacobian is accounted for in the shader with an additional precomputed factor — the inverse norm of the clamped LTC distribution. Thus, five parameters need to be looked-up from a precomputed 64×64 texture: the values a,b,c, and d in the matrix below, and the inverse norm of the corresponding clamped cosine distribution for energy conservation.

$$M = \begin{bmatrix} a & 0 & b \\ 0 & c & 0 \\ d & 0 & 1 \end{bmatrix}$$

Since the precomputed norm is for the horizon clamped LTCs, the stored distributions already encode the correct responses when the light moves below the horizon, and so the shader does not need to explicitly clip the polygon using a branch heavy algorithm like Sutherland-Hodgman or similar polygon clipping approaches. Instead, a simple texture lookup retrieves the corrected form factor, making it very efficient.

LTC polygonal lights are currently the simplest and most accurate real-time implementation, and so the obvious choice for most pipelines, including ours. LTCs even support textured area lights, although that is not important for our project. For use cases like low-end GPUs where the need for memory access to precomputed lookup tables is more expensive, Technicolor's completely analytic method — Lecocq et al. (2016) — may be one to consider instead, at the cost of more artefacts.

2.2 Light Culling

To maintain real-time performance in scenes with many light sources, modern rendering pipelines cull lights: a pre-shading step that rapidly discards lights with negligible contribution to a pixel, making lighting much more efficient. The goal is to quickly estimate whether a light has meaningful influence before evaluating its full contribution in the fragment shader. Efficient light culling is essential when rendering hundreds, thousands, or potentially millions of lights. However, most existing algorithms are tailored for punctual lights like point or spot lights, which can be easily bounded by spheres or cones. In contrast, polygonal lights have more complex spatial extents and directional characteristics, making them harder to cull efficiently.

In this section, we review common real-time light culling techniques and discuss the challenges in adapting them to polygonal lights. In chapter 4, we then propose modifications to clustered shading (Olsson et al., 2012) that enable aggressive culling of area lights while preserving visual fidelity — with important considerations for specular highlights. The methods we focus on are for dynamic lights which can move, change colour/intensity, and be spawned in and out. Not only does this extra flexibility over static lights allow for more dynamic and interesting scenes, but dynamic methods also allow faster artist workflows because there is no offline baking of light maps or culling data-structures required after every change to a scene.

In addition to light culling, deferred shading is another technique that helps optimise lighting by reducing redundant calculations. It initially renders the raw data of each object in the scene into a set of textures called G-buffers, which typically store attributes such as world-space positions, normals, albedo, and material properties. In a second screen-space pass, the fragment shader reads from these G-buffers and evaluates the lighting contribution of all relevant lights per fragment to produce the final shaded image. This deferred approach reduces redundant lighting calculations caused by overdraw, since lighting is computed once per screen pixel, rather than per-fragment during rasterization of each object like in forward shading.

Whilst deferred shading is an industry proven method, it does not replace forward shading because the intermediary G-buffers are large and bandwidth heavy, which can idle modern GPUs since their throughput is much more heavily bottlenecked by raw data transfer rates rather than parallel computing power (mobile GPUs struggle with G-buffers due to bandwidth too). Deferred shading also has no simple solution to transparency, and so transparent objects must be done in a separate forward pass. Maintaining two very different render passes for the same scene complicates the engine significantly. Forward shading not only has better support for transparency, but also for MSAA (Multi-Sample Anti-Aliasing) since in deferred shading this means storing 2 to 16 times the amount of G-buffer data. The overdraw in forward shading can be minimized using an early depth test to prepass the scene to a depth buffer (the overhead of this depends on the hardware) thus preventing unnecessary fragment shader invocations.

Tiled Shading (Olsson and Assarsson, 2011) builds upon the deferred shading pipeline — and forward shading via Forward+ (Harada et al., 2012) — by introducing a light culling step in screen space using compute shaders. The screen is divided into a grid of 2D tiles (typically

16×16 pixels), and a compute pass determines which lights affect each tile by performing light-bound intersection tests using depth information from the G-buffer. During the lighting pass, only the subset of lights relevant to a tile are evaluated for each fragment, significantly reducing the number of lighting calculations compared to considering all lights globally. This optimization is especially effective in scenes with many small or localized lights, as those lights only influence a few tiles.

With Tiled Shading leveraging the parallelism of compute shaders, it is well-suited for modern GPU architectures. However, it operates purely in 2D screen space, which can lead to inefficiencies in scenes with large depth discontinuities, causing foreground fragments to also process lights affecting background elements and vise versa. Also, assigning the specular component of light is challenging since tiles are 2D. Traditionally this was not an major issue, as the specular blips from point lights become invisibly narrow on glossy surfaces, and for rough surfaces they are broad and low-intensity, often blending into the diffuse response. As a result, their visual impact is minimal. Area lights on the other hand, maintain significant specular contributions even on rough surfaces due to their spatial extent — making them more important to consider during shading and culling.

In a GCD talk, Thomas (2015) discusses the many extensions to tiled shading that have come about for further light operation savings. For instance, the min and max depth of lights for each tile can be calculated via a min-max parallel reduction over the depth buffer per tile. The simplest way is to use atomic min-max operations but it can be made more efficient with a binary tree reduction to remove the need for atomics. Depth discontinuities can still arise in atomic-min-max but are addressed by 2.5D culling from Harada (2012) using two bit masks per tile where the bits represent Z-bins that split the depth buffer. The first depth bitmask has a 1 in the positions where geometry is, the second has a 1 in positions where lights are assigned. The bitwise AND retrieves the overlap, ignoring lights not overlapping with geometry.

Even with 2.5D tiling further reducing the number of contributing lights, we are still looping over lights per 2D screen-space tile, just with lights in empty Z slices not being added to the computation. The number of lights in 2D tiles is still not efficient enough to loop over with LTC-lights because of how much more computationally expensive they are than point lights. Not only that, tiled shading performance is unstable due to view dependence, and to assign the specular contribution of polygonal lights requires considering the potential reflection vectors for the lights in each tile, and without any depth or normal information about the geometry within the tile, a specular heuristic would have to assign camera facing lights to practically the entire screen to avoid specular artefacts from light under-assignment. So instead of tiled shading, we consider a clustered approach (full 3D tiled) by Olsson et al. (2012), which assigns lights to 3D clusters (voxels or froxels). We benefitted from a less common variant brought up in the original paper where each voxel/froxel is split into further clusters by grouping normal directions.

Clustered shading (Olsson et al., 2012) subdivides the view-frustum into a 3D grid. The grid can either be fitted by froxels (frustem elements) – i.e. the natural shape from dividing screenspace tiles on the Z-axis – or voxels defined by AABBs (Axis-Aligned Bounding Boxes) — the trade off in using voxels is they are defined by a simpler shape, but voxels slightly

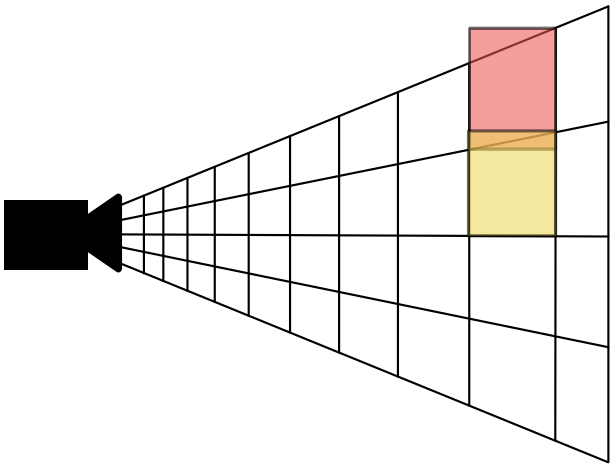

Figure 2.2: Voxel clusters overlap (orange region) when fitting to the froxel grid (top-down 2D illustration of the 3D grid).

overlap to fit the froxel grid (Figure 2.2). Each element of this grid is a cluster that – like a tile – stores an array of indices to lights to that affect that cluster. The light assignment can either be brute-forced assigned to clusters in $\mathcal{O}(\text{num lights} \times \text{num clusters})$ time, or accelerated hierarchically by first constructing a Bounding Volume Hierarchy (BVH) over the lights. resulting in $\mathcal{O}(\log(\text{num lights}) \times \text{num clusters})$ time. The BVH groups lights into progressively larger bounding volumes (AABBs or spheres), enabling efficient culling during assignment. In the fragment shader, the cluster index if found based on its quantized position. This gives the light list that the fragment iterates over. Point lights are culled using a precomputed sphere bounding volume by solving for the distance where light attenuation brings its radiance below some minimum perceivable threshold. When extending to hundreds of thousands or millions of point lights, this precomputation can be parallelised — either with multithreading on the CPU, or more scalably on the GPU with another compute shader.

A minor performance drawback of using per-cluster instead of per-tile light lists is reduced branch coherency in the fragment shader. In tiled shading each light list is typically shared for each 16x16 block of pixels, which automatically has good branch coherency because GPU thread-groups perform the same instruction as the other threads in the group (Single Instruction Multiple Data). Clustered shading does not govern that tiles in screen-space use the same light lists, giving the potential for diverging branches within thread-groups. This turns out to seldom appear as an actual problem however, and Olsson et al. (2012) shows that clustered shading greatly decreases the number of light operations over tiled shading, which is the main concerning factor that has made the expensive LTC lights appear so little in games.

Chapter 3

Methodology

3.1 Overview of Approach

Designing polygonal light assignment for clustered forward shading required complete control over the rendering pipeline, which ruled out extending an off-the-shelf renderer like Unity or Unreal Engine. These engines are highly optimized and abstracted, often making it difficult — if not infeasible — to modify low-level systems such as light clustering logic, GPU data layouts, or shader integration for custom BRDF approximations like Linearly Transformed Cosines (LTC). In contrast, implementing the renderer from scratch provided full visibility into each stage of the shading pipeline and the flexibility to prototype, debug, and validate novel assignment strategies. This level of control was essential given the exploratory and experimental nature of the project.

It is worth noting that Google's Filament renderer could have been a viable base, as it already uses clustered forward shading for punctual lights and lacks polygonal lights. However, I only became aware of it after completing this project, and whilst Filament is impressive and actively maintained, its maturity and complexity may have made it more challenging to iterate on experimental ideas that require changes across the entire pipeline. In retrospect, the existence of Filament reinforces the validity of the clustered forward shading approach I selected (as argued in section 2.2), especially considering it's high performance across so many platforms (mobile, web and desktop). Similarly to Filament, I opted to load scenes from glTF 2.0 files, a modern game-ready format designed specifically for real-time PBR pipelines, with ample support from industry-standard 3D tools.

It quickly became clear that an iterative development approach was essential, since both clustered shading and polygonal lights depend on having a fully working renderer as a foundation. This structured progression ensured each feature was robustly integrated into the system, so I began — and stuck to — the following development plan:

- Develop a glTF 2.0 PBR forward renderer (with fallback plan that I could use the simpler OBJ file format).
- Implement clustered forward shading for point lights.
- Implement LTC polygonal lights.
- Integrate LTC lights into clustered forward shading: Starting simple by only considering diffuse bounding volumes.

- Handling polygonal light assignment with glossy materials (initially a stretch-goal): Area light specular is not properly contained by tight diffuse bounding volumes.
- Test and iterate on design for assignment approaches, focusing on reducing light operations and assignment computational complexity.
- *Future Work (Post-Project):* Add hierarchical assignment step for real-world application *(crucial for deployment with hundreds of thousands of lights, but not included in our scope).*

Non iterative approaches like Waterfall would never work for this project. Early design decisions needed to be validated through experimentation — particularly in clustered light assignment, where I tested numerous heuristics. For example, culling strategies (AABBs, spheres, cones, normals) had to be prototyped and evaluated before selecting the most effective approach. These decisions could not have been made upfront without implementing and benchmarking real, working systems. Iteration enabled more agile responses to unexpected challenges — like needing to rework compute shaders for discrete GPUs or artefacts from specular underassignment — and allowed for gradual improvement through profiling and testing. It also allowed the project scale naturally: if I had not successfully rendered glTF 2.0 files within the first month, I planned to fallback to the simpler OBJ format, and stretch goals like improving specular assignment were only pursued after a functioning foundation.

Building a renderer from the ground up required selecting a graphics API to interface with the GPU. OpenGL 4.6 was chosen as a modern, crossplatform API that supports compute shaders — a solution requirement. My existing familiarity with OpenGL significantly accelerated development, which was necessary to complete the project within a few months. Vulkan, while offering more explicit control over the GPU, requires a considerable amount more boilerplate and infrastructure just to achieve basic rendering, which was unnecessary for the scope of this project. Additionally, OpenGL is far more cross-platform than DirectX (which is Windows-only) or Metal (which is macOS-only), making it a practical choice for portability and development flexibility. While the core features exploited for this project — such as compute shaders and the debug message callback — were introduced in OpenGL 4.3, I opted to target OpenGL 4.6, the latest version of the API. Targeting 4.6 ensures compatibility with the most recent driver optimizations and better cross-vendor behaviour.

This project is implemented in C rather than C++. While C++ is often used in game engine programming due to its abstraction features, I chose C for its minimalism, faster compilation, and closer alignment with low-level systems programming. By avoiding C++'s broader ecosystem and bloated paradigms, we retain the fine control over performance-critical code while reducing unnecessary toolchain overhead. I compiled with Mingw-w64 (Minimum GNU for Windows) which meant the equivalent Linux build script was almost identical, and after completing the majority of development on a Windows laptop, porting to Linux was completed in under half an hour, with minimal effort required beyond transferring the project files. To support development, several lightweight C libraries were used to avoid unnecessary boilerplate and ensure cross-platform compatibility. GLFW provided simple window and OpenGL context

creation, allowing quick deployment to both Windows and Linux systems. Scene loading was aided by `cgltf` for parsing glTF JSON into C structures, `stb_image.h` for JPG and PNG textures, and `cglm` for fast vector, matrix, and quaternion operations. A minimal runtime UI was added with Nuklear to adjust shading parameters interactively during testing. I added a simple shader metaprogramming system where small variations in shader code, such as lighting models or optimizations, could be toggled at runtime by injecting preprocessor defines and recompiling shaders on the fly, significantly accelerating experimentation. OpenGL function loading was automated using `glad`, avoiding tedious manual setup for each platform.

Debugging graphical applications is notoriously difficult, so care was taken to set up reliable error detection. OpenGL's debug callback mechanism was used to catch API misuse, and I adapted it to forcibly crash on severe errors, allowing use of gdb's `backtrace` to directly pinpoint faults. However, many issues like incorrect buffer formats do not trigger API errors. For deeper inspection, Renderdoc was invaluable: it allowed full frame captures showing vertex buffers, textures, shader inputs, and pipeline state, making subtle bugs far easier to diagnose.

Version control was handled with Git, using a GitHub remote for easy machine portability and backup. Large scene files, which are ill-suited to Git, were transferred separately via OneDrive. Blender was used throughout development to create small, controlled glTF test scenes with rough and glossy surfaces, and later to duplicate complex scenes like the UE4 Sun Temple efficiently using instancing, allowing for larger-scale testing without bloated asset files.

3.2 Renderer Architecture and Pipeline Design

Figure 3.1 shows the final renderer architecture, capable of rendering complex scenes with full support for physically based rendering (PBR) workflows, and includes timing features and light operation counting for profiling and benchmarking different techniques. The system handles all common material maps: base colour maps define the diffuse surface albedo; metallic-roughness maps distinguish between conductor and dielectric surfaces while encoding surface microfacet roughness; normal mapping perturbs surface normals to add detailed lighting variation without increasing geometric complexity, where tangent space data is available; emissive maps allow parts of materials to appear self-illuminated, independent of external lighting; and ambient occlusion maps precompute local occlusion to darken crevices and add a greater sense of depth. The architecture first precomputes bounding volumes on the CPU, then the GPU creates the cluster grid and assigns lights. Finally draw calls invoke the fragment shader which uses the cluster grid for more efficient shading. We will now discuss the design choices behind our clustered shading implementation.

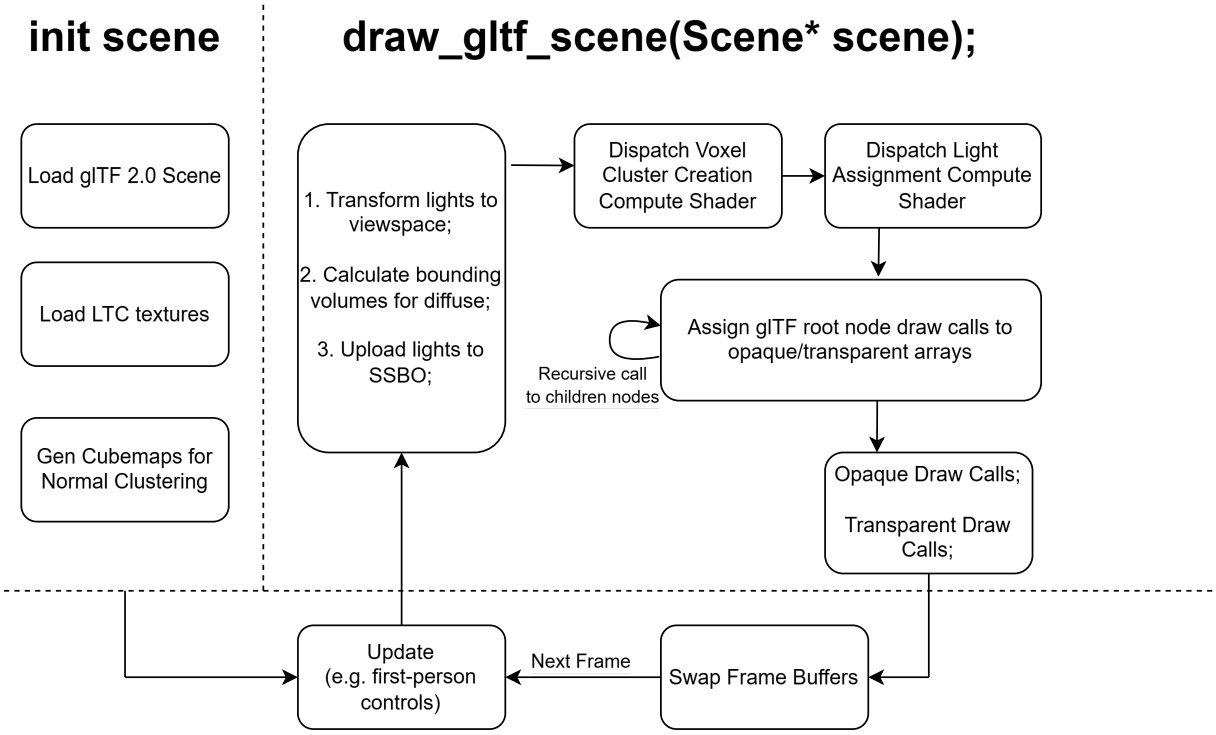

Figure 3.1: Final Renderer Architecture

3.3 Design for Clustered Shading

The optimal amount to subdivide the view frustum must come from empirical testing, after exploring many options on both integrated and discrete graphics card our final implementation performs best with $(16 \times 9 \times 24)$ clusters on all tested platforms. There was around a 15% decrease in light operations with 24 depth slices over 12, but doubling the depth slices doubles the assignment time. We used voxel clusters for light assignment because they are far simpler than froxels to reason about in relation to area lights, especially simplifying the implementation for specular light that also considers normals. Typically, whether lights are stored in view space or world space does not matter, however each voxel cluster is defined by an Axis-Aligned Bounding Box (AABB), which requires view space so that their axes align with, and tile our 3D grid properly. Lights must therefore be transformed into view space to perform correct light-cluster intersection tests.

The next choice in clustered shading is how to partition the depth-axis of the view frustum into clusters. The simplest approach of uniformly dividing the frustum in NDC space (Normalized Device Coordinates) is not ideal since the way the world is projected into NDC space is not linear and creates very thin slices towards the near plane, and excessively long slices towards the far plane. Linear depth slicing in world/view space is also rare since close geometry when projected occupies many more pixels, so closer clusters should really occupy much less world space volume. The best option is to slice exponentially in view space. The idea is that because the width and height of far clusters are much larger due to perspective distortion, it is better to divide the depth similarly, creating approximately cube shaped clusters, which exponential slicing does well. Given a view frustum defined by near and far clipping planes, an exponential

depth slicing defines the min and max z values for a cluster as follows:

$$\texttt{cluster}_{\texttt{near}} = \texttt{near} \times \left(\frac{\texttt{far}}{\texttt{near}}\right)^{\frac{\texttt{depth\_slice}}{\texttt{num\_depth\_slices}}}$$

$$\texttt{cluster}_{\texttt{far}} = \texttt{near} \times \left(\frac{\texttt{far}}{\texttt{near}}\right)^{\frac{\texttt{depth\_slice}+1}{\texttt{num\_depth\_slices}}}$$

To further improve performance during light assignment, we assign one workgroup per cluster instead of one thread per cluster. This enables efficient use of shared memory within each workgroup, dramatically reducing overhead from global memory accesses and thread divergence. In practice, this strategy yielded a 25× speed-up in cluster light assignment performance. (See Appendix F for details on how work is distributed across the workgroup.)

3.4 Polygonal Light Culling and Assignment

In our polygonal light assignment approach, we track diffuse and specular contributions separately, halving the light shading time in clusters affected by only one of diffuse or specular components. We first apply a half-space rejection to discard clusters fully behind single-sided polygonal lights. These lights emit only in the direction of their surface normal. Naively testing if the cluster centre is behind the light is fast but can lead to visible artefacts, as clusters may still be partially intersected by the light volume; checking each corner of the cluster fixes this but is not efficient. Instead, we test the furthest point in the cluster in the opposite direction of the light normal. If this point is also behind the light plane, we safely reject the cluster. This avoids both false positives and unnecessary lighting calculations (see Figure 3.2), while incurring only minimal additional cost over the naive approach. For lights that pass the half-space rejection, we proceed with diffuse and specular assignment, using separate culling strategies tailored to each component.

To accelerate diffuse polygonal light assignment, we precompute simple, tight bounding volumes that encapsulate each light's influence. Specifically, we use a two-stage approach: a fast AABB test, followed by a tighter bounding sphere test. This reduces the number of

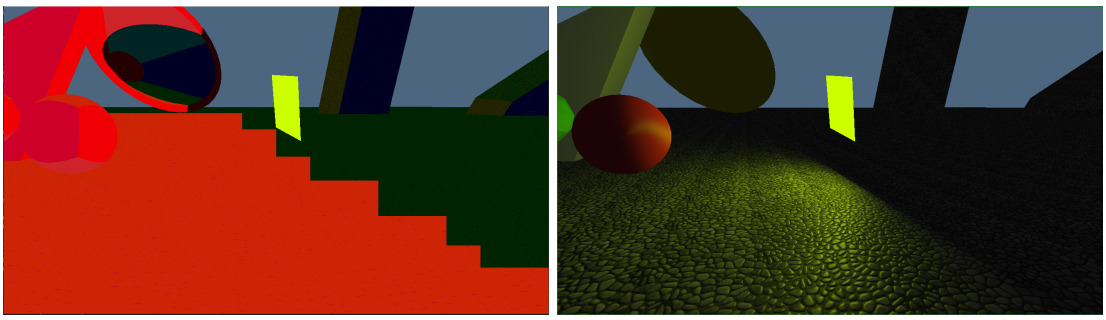

Figure 3.2: Left: Visualisation of clusters assigned by the furthest-point based half-space rejection test. Right: Only clusters fully behind the light are culled, avoiding artefacts along the light border.

expensive per-cluster light evaluations. The bounding sphere is centred at the light's polygon centroid and extended by an influence radius. This radius accounts for both the geometric size of the polygon and its luminous intensity, providing a consistent culling threshold across lights of different sizes. The AABB is derived from the polygon's bounds, expanded equally in all directions by the same influence radius. These bounds assume all lights are double-sided, since single-sided rejection is already handled by the half-space test. Although Oriented Bounding Boxes (OBBs) can provide tighter bounds for elongated lights, we opt for spheres and AABBs in our implementation for simplicity and efficiency. A promising alternative to long, thin polygon lights bounded with OBBs is to instead represent them as LTC line lights, as proposed by Heitz and Hill (2017). Line lights are more efficient to shade and can be culled effectively by testing the distance from the cluster centre to the light segment against the sum of the light's influence radius and the cluster's diagonal length.

Specular lighting is handled separately, using either an expanded bounding sphere or a more accurate light cone test based on normal clustering. The expanded sphere approach simply enlarges the diffuse bounding radius to better encompass regions where specular highlights might appear outside the diffuse bounds. This method is trivial to implement, requiring only a scale factor on the diffuse radius. Only specular contributions are assigned using the expanded region, so fragments in this outer zone skip evaluation of the expensive diffuse LTC. This makes the method an effective, position-only solution for reducing mild specular artefacts, but in glossy scenes or shading models with strong specular, the expanded sphere radius would need to be far too large for efficient culling — for instance, in the mirrored Sun Temple scene, a 1.5× specular radius increased light operations by 54%, and very glossy scenes would need far more than a 50% increase, causing too many lights to overlap for efficient shading.

To address these inefficiencies, we explore a more precise alternative based on normal clustering. By leveraging the orientation of surface normals, we can more tightly cull specular contributions, significantly reducing light overlap and improving performance in highly reflective scenes. This approach results in more accurate specular assignment and proves vastly more efficient than the expanded bounding sphere method, especially in scenes with strong specular response. I also experimented with enhancing position-only specular assignment using scene-defined roughness ranges and specular thresholding techniques inspired by Blinn-Phong pipelines (Källberg and Larsson, 2014). However, these methods yielded limited benefits without dynamic roughness information and introduced additional complexity to the scene setup. That said, they remain promising directions if roughness ranges can be computed dynamically per cluster — for example, via parallel reduction over the roughness buffer in a deferred rendering pipeline. More generally, especially accurate variants of clustered shading could emerge by performing per-tile reductions over the full G-buffer, incorporating not just roughness but other perceptual cues — for instance, dark coloured surfaces have a higher specular threshold than light coloured ones. Finally, I explored a simpler approximation by assuming a fixed upward-facing normal for each cluster. This idea draws from optimizations in glossy ray-traced global illumination, where the vertical resolution of reflections can be reduced (McGuire, 2019), given that most visible reflections originate from horizontal surfaces like water or floors. However, I found this technique to have limited generality and ultimately did

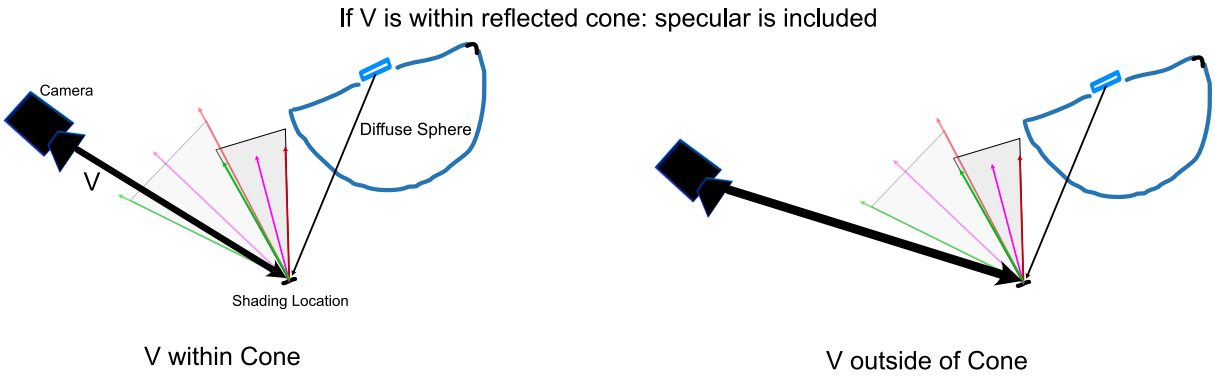

Figure 3.3: Illustration of reflected light cones (transparent) for rays reflecting off a perfect mirror with potential surface normals defined by the cluster's normal cone (opaque). Specular is culled if the view vector V (black) lies outside this cone.

not adopt it in our final implementation. Based on these insights, I developed a normal based specular assignment algorithm that balances accuracy with efficiency using cone volumes.

3.4.1 Normal-Based Specular Clustering

To more accurately assign specular lighting contributions from area lights, we define clusters not only for each spatial group (voxels) but also by groups of normal directions (normal cones). At each voxel in this model, there are multiple clusters for each quantized normal, forming a 4D cluster grid. A normal cone represents the angular spread of normals in a cluster, and reflected light will thus have a reflected cone of possible outgoing light directions — for perfect mirrors this is illustrated in Figure 3.3. The axis of the reflected light cone is the reflection of the incident vector over the normal cone axis, and it is trivial to show that the angular spread of the light cone for a perfect mirror is double the normal cone angle. For materials with microfacet BRDFs, the specular peak is elliptical instead of exact, requiring the cone angle to be widened based on the specular BRDF's max lobe width. We use a default value of 45° for this adjustment, though specific implementations may benefit from experimenting with smaller angles to achieve a balance between accuracy and performance.

Our current implementation leaves out cone widening due to the angular size of the light and clusters, which causes artefacts close to lights. To address this, we always include specular within the diffuse radius and only apply this method to the outer radius of more distant specular. I briefly tested a formulation that incorporated the angular sizes of clusters and lights, but it was too late in the project to fully explore — although it should be very doable. Chapter 5 demonstrates that once this behaviour is corrected near lights, the specular assignment becomes highly efficient by applying normal cones to the inner radius as well.

Chapter 4

Implementation and Validation

This chapter details the implementation of the core components of our renderer and provides visual validation of its results. We first present our glTF 2.0-compliant physically based renderer. Figure 4.1 demonstrates successful rendering of several key PBR features, including accurate area light reflections, correct interpretation of metal-roughness textures, support for normal mapping, and proper handling of glTF blend modes. Subsequent sections describe the implementation of polygonal lights using Linearly Transformed Cosines (LTC), as well as the light assignment strategy, including how to setup clustering based on surface normals.

4.1 LTC-Based Polygonal Lights

Separate LTC distributions are used for the diffuse and specular terms, with Lambertian and GGX BRDFs selected, respectively. The diffuse colour is obtained by multiplying the Lambertian irradiance — computed via LTC integration using the identity matrix — with the mesh's base colour. The specular component incorporates the GGX LTC matrix, which approximates the distribution term D of the Cook-Torrance BRDF. The Fresnel term F is handled using Schlick's approximation, modified with a shadowed F_{90} value sampled from the second precomputed texture. The geometry term G, which accounts for visibility and masking, is also taken from the texture and corresponds to the Smith masking function.

An alternative for the diffuse term could be Disney's BRDF (Burley, 2012), which yields slightly more realistic results than the Lambertian model. However, Lambertian diffuse does not require an LTC transformation, so we only need precomputed lookup textures for the specular component. We use the GGX LTC textures generated by Heitz et al. (2016), sampled via bilinear interpolation with the fragment's roughness and view angle.

Each frame, we upload view space polygonal lights to video memory by memory mapping a Shader Storage Buffer Object (SSBO). Implementation details related to the storage and struct packing of dynamic polygonal light lists are provided in Appendix E.

Figure 4.1: Validation of core rendering features (left to right): area light reflection shapes, metal-roughness maps, normal mapping, and glTF blend modes. Emissive and occlusion maps are also supported.

The precomputed matrices M^{-1} are calibrated for surfaces with normal $\mathbf{N} = (0, 0, 1)$. To evaluate LTCs for arbitrary surface normals, we construct a local tangent-space basis $\{\mathbf{T}_1, \mathbf{T}_2, \mathbf{N}\}$, where $\mathbf{T}_1$ lies in the plane formed by the view vector $\mathbf{V}$ and the surface normal. The matrix M^{-1} is then transformed into this local frame by multiplying it with the transpose of the basis matrix, aligning the lighting lobe appropriately with respect to the viewer and surface. Once reoriented, the polygonal light is transformed accordingly, and the clamped cosine irradiance is computed. This process is performed in the fragment shader for fragment position `P`, normal `N`, and view vector `V` as shown below:

```
vec3 T1 = normalize(V - N * dot(V, N));
vec3 T2 = cross(N, T1);
Minv = Minv * transpose(mat3(T1, T2, N));

// We construct our spherical polygon from this new matrix like so:
vec3 points_o[MAX_UNCLIPPED_NGON];
for (int i = 0; i < n; ++i)
{
    points_o[i] = normalize(Minv * (viewspace_points[i].xyz - P));
}
vec3 vsum = integrate_lambertian_hemisphere(points_o, n);

// Apply normalisation factor that also accounts for horizon clipping
return vec3(length(vsum) * horizon_clipped_scale);
```

The standard float32 `acos` function provided in GLSL, HLSL, and similar shading languages should not be used to compute the arccosine in `integrate_lambertian_hemisphere()` (Equation 1.2). This is because the default `acos` implementation lacks the high precision required in this context, resulting in severe visual artefacts with bright lights (Hill and Heitz, 2017). A more accurate approximation tailored for this use case is presented in Appendix G, which also keeps the result in an intermediary vector form to enable automatic horizon clipping via horizon clipped form factors that are combined with the precomputed LTC norm.

4.2 Polygonal Light Assignment

Each light is tested against each cluster using our GLSL function `test_arealight()`. The implementation of per-workgroup light assignment, which invokes this function across slices of the global light list per thread, is detailed in Appendix F.

This function performs three steps to assign polygonal lights, storing diffuse and specular contributions separately. These are encoded into the first two bits of a `uint` as follows: `none=0b00`, `diffuse=0b01`, `specular=0b10`, `both=0b11`.

```
uint test_arealight(uint i, Cluster cluster, NormalCone nc)
{
    // 1. Half space rejection for single sided area lights
    // 2. Diffuse test
    // 3. Specular test

```

```
    // Separate diffuse and specular light culling
    uint result = 0u;
    if (diffuse_passed)  result |= 0x1u;
    if (specular_passed) result |= 0x2u;
    return result;
}
```

Half-space rejection is implemented efficiently using GLSL's `lessThan()` function to select the correct min/max corners of the cluster AABB for evaluating the furthest point:

```
float plane_constant = -dot(light_normal, p0);
vec3 furthest_point = mix(
    cluster.max_point.xyz, cluster.min_point.xyz,
    lessThan(light_normal, vec3(0))
);
if (dot(light_normal, furthest_point) + plane_constant < 0)
{
    return 0u;
}
```

This test is only executed when the simpler centre test fails (see Appendix H.1 for full implementation). Figure 3.2 in Chapter 3 shows this rejection technique in action. Only clusters fully behind the light are culled, avoiding border artefacts.

We had not yet described how the influence radius of an area light is computed for diffuse bounding spheres and AABBs. A practical heuristic is employed, scaling with both the area and intensity of the light to yield consistent bounds. The `area` parameter is computed as the polygon's surface area, using the Shoelace formula after projecting the polygon onto the local XY-plane. The projection aligns the Z-axis with the polygon normal. Full implementation details are provided in Appendix H.2.

```
float calculate_area_light_influence_radius(AreaLight* al, float area, float
    min_perceivable)
{
    // 1. Radiant flux scales with area
    float flux = (al->color_rgb_intensity_a[0] +
        al->color_rgb_intensity_a[1] +
        al->color_rgb_intensity_a[2]) *
        al->color_rgb_intensity_a[3] * area;

    // 2. Empirically tuned scaling factor
    float effective_flux = flux * 4.4f;

    // 3. Compute radius for hemispherical irradiance: Find r in E = flux / (2
    PI r^2)
    return sqrt(effective_flux / (2.0f * M_PI * min_perceivable));
}
```

Listing 4.1: C code for computing the influence radius used in bounding volume calculations. The final radius includes the polygon's geometric radius.

This heuristic estimates the distance at which a perfectly aligned Lambertian surface — facing

the centre of the polygon — receives irradiance just above the minimum perceptible threshold. This provides a conservative bound. The scaling factor of 4.4 is an empirically derived constant, obtained through testing, which ensures that diffuse light is not prematurely attenuated.

To determine whether a cluster AABB intersects with the diffuse bounding sphere, a standard sphere-AABB test is used: compute the closest point on the AABB to the sphere's centre, and compare the squared distance to the squared radius to avoid unnecessary square roots.

```
bool sphere_aabb_intersection(vec3 center, float radius,
    vec3 aabb_min, vec3 aabb_max)
{
    // Closest point of AABB to the center of the sphere
    vec3 closest_point = clamp(center, aabb_min, aabb_max);
    closest_point -= center;
    float distance_squared = dot(closest_point, closest_point);
    return distance_squared <= radius * radius;
}
```

Listing 4.2: Sphere-AABB intersection test in GLSL.

The sphere test is only evaluated if the light's AABB intersects the cluster AABB. This AABB-AABB test uses vectorized boolean operations:

```
bool aabb_aabb_intersect(vec3 aabb1_min, vec3 aabb1_max,
                         vec3 aabb2_min, vec3 aabb2_max)
{
    return all(greaterThanEqual(aabb1_max, aabb2_min)) && all(lessThanEqual(
    aabb1_min, aabb2_max));
}
```

For the position only specular improvement from Chapter 3: multiply the diffuse influence radius by a constant factor (1.5 was used in our tests):

```
bool specular_passed = sphere_aabb_intersection(sphere.xyz, 1.5 * sphere.w,
    cluster.min_point.xyz, cluster.max_point.xyz);
```

4.2.1 Normal Clustered Shading Pipeline Implementation

Introducing quantized normal directions extends the cluster grid to four dimensions. To fit this into the GPU's 3D compute shader execution model, we fold both depth and normal indices onto the Z-axis. The compute dispatch for cluster creation reflects this adjustment:

```
glDispatchCompute(1, 1, CLUSTER_GRID_SIZE_Z * CLUSTER_NORMALS_COUNT);
```

```
layout (local_size_x = CLUSTER_GRID_SIZE_X, local_size_y = CLUSTER_GRID_SIZE_Y,
    local_size_z = 1) in;
```

Light assignment is performed on a 1D execution grid. The position and normal indices are decoupled using the following approach:

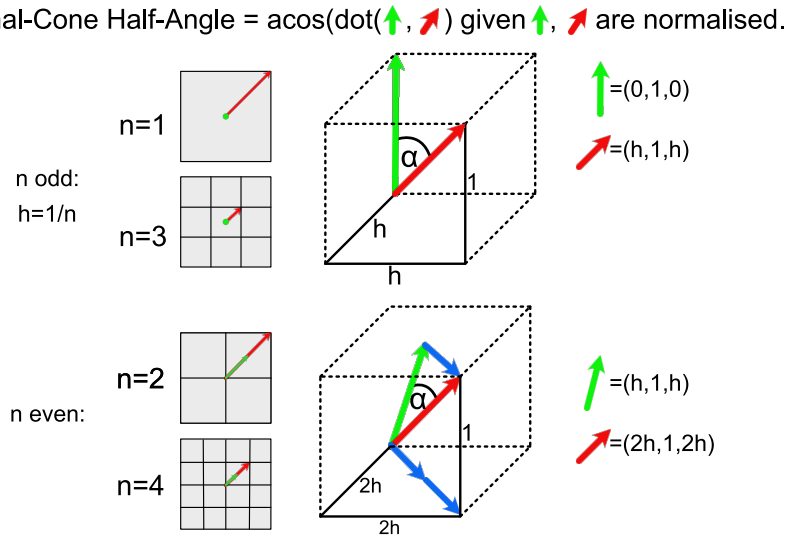

Figure 4.2: Deriving the normal cone half angle for normal counts of 6, 24, 54, etc.

```
// Load the cluster from global memory into shared memory.
uint index = gl_WorkGroupID.x;
if (gl_LocalInvocationID.x == 0)
{
    cluster = clusters[index];

    // Reset our light counts.
    cluster.point_count = 0u;
    cluster.area_count = 0u;
}
barrier();  // Ensure all threads see the loaded cluster

uint clusters_per_layer = CLUSTER_GRID_SIZE_X * CLUSTER_GRID_SIZE_Y;
uint combined_z = index / clusters_per_layer;
uint normal_bin = combined_z % CLUSTER_NORMALS_COUNT;
```

Normal directions are quantized by subdividing each face of a cube into $n \times n$ regions, resulting in a total of $6n^2$ normal cones. Each cone is centred on its corresponding cube face subdivision. The maximum half-angle of each cone is derived geometrically, as shown in Figure 4.2:

```
float n = sqrt(CLUSTER_NORMALS_COUNT/6);  // since CLUSTER_NORMALS_COUNT = 6n^2
float h = 1.0 / n;
if (int(n) % 2 == 0)
{
    normal_cone.half_angle = acos(dot(normalize(vec3(h, 1.0, h)), normalize(vec3
    (2.0*h, 1.0, 2.0*h))));
}
else
{
    normal_cone.half_angle = acos(normalize(vec3(h, 1.0, h).y));
}
normal_cone.cluster_normal = texture(representative_normals_texture, normal_bin
    / float(textureSize(representative_normals_texture, 0))).rgb;
```

This normal cone represents the range of surface normals in a given cluster and is passed to

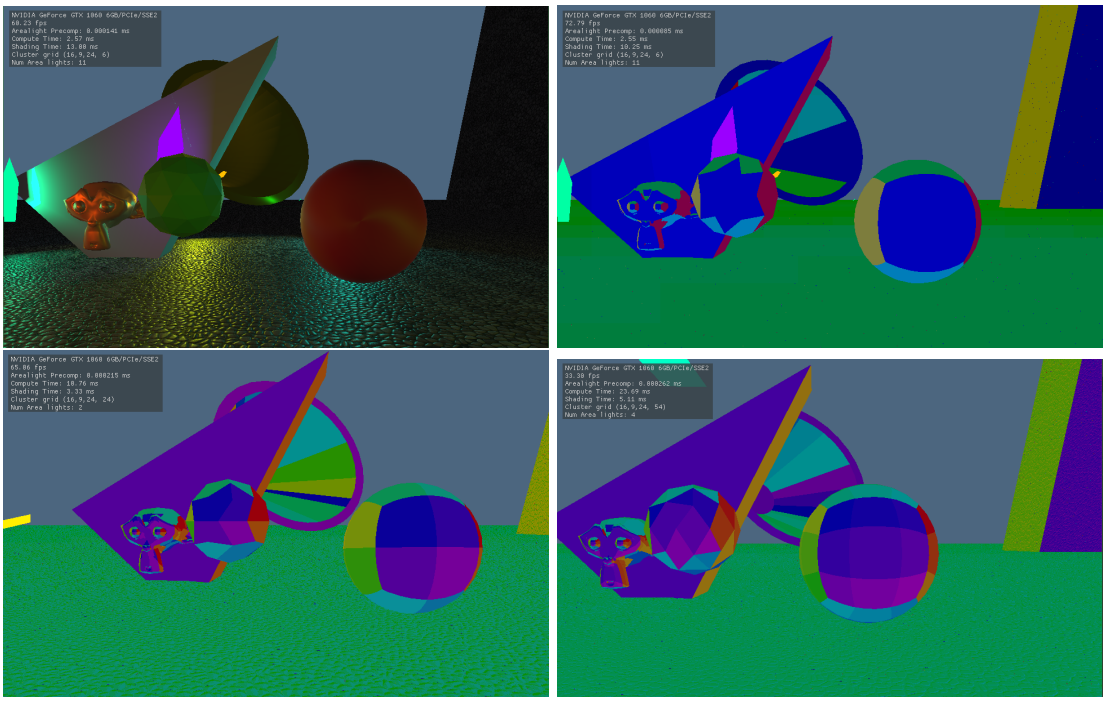

Figure 4.3: Visualizing quantized normal directions. Looking at how the sphere is quantized expectedly matches the grid from Figure 4.2: (a) $1 \times 1 \times 6 = 6$ normals, (b) $2 \times 2 \times 6 = 24$ normals, and (c) $3 \times 3 \times 6 = 54$ normals.

`test_arealight()` for directional culling. The cone axis for a normal bin is retrieved from a small representative normals texture. In the fragment shader, surface normals are snapped to their respective normal cones using a cubemap lookup — effectively the inverse of the representative normals texture. This texture and cubemap is generated in `src/main.c:init_empty_cluster_grid()`. If normal mapping is enabled, the TBN matrix is applied prior to quantization:

```
vec3 N;
if (is_normal_mapping_enabled == 1)
{
    vec3 normal_sample = texture(normal_texture, texcoord_0).rgb * 2.0 - vec3
    (1.0);
    N = normalize(tbn_matrix * normal_sample);
}
else
{
    N = normalize(tbn_matrix[2]);  // Set N to view_normal
}
uint normal_index = texture(cluster_normals_cubemap, N).r;
```

Figure 4.3 visualizes the results of this normal quantization. As shown, the sphere's surface is discretized in accordance with the expected cube face grid derived in Figure 4.2. With this implementation in place, the normal cones can be used to support the specular assignment method described in Chapter 3. The full implementation is provided in Appendix H.3.

Chapter 5

Results, Evaluation and Discussion

5.1 Results

A real-time flythrough showcasing the 2016-light diffuse bounding performance is available in [Demo Video 1, Appendix C]. The test scene contains approximately 606,000 triangles, comprising sixteen instances of the Sun Temple model arranged in a grid. This setup provides a representative mid-scale environment to evaluate rendering performance under dense area lighting — where the high concentration of area lights imposes substantial computational demands, even under ideal culling conditions. Without culling, the frame time is over 140× higher, highlighting the critical importance of effective light assignment.

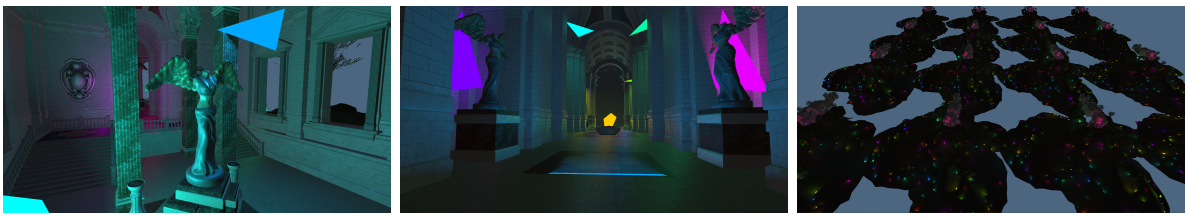

Figure 5.1: Scene perspective for results tables. Left to right: Location A, Location B, Location C.

Table 5.1: Performance on NVIDIA GTX 1060 at 1280x720 (cluster grid: $16 \times 9 \times 24 \times N$)

N	Light Ops	Compute (ms)	Fragment (ms)	Total (ms)
126 Lights – 16 Sun Temples Location A				
1	65936949	0.43	105.55	105.98
6	51355552	2.8	88.7	91.5
24	53645586	11.0	85.7	96.7
54	47590904	24.75	73.0	97.75
dif_1	42893462	0.43	59.6	60.04
2016 Lights – 16 Sun Temples Location A				
1	70101700	1.60	119.7	121.3
6	51519648	8.15	84.0	92.15
24	55694960	32.2	95.5	127.7
54	48230849	72.2	80.45	152.65
dif_1	42915180	1.5	65.45	66.95

Note: **dif**$_1$ rows are reference timings without normals where diffuse and specular are captured by only the diffuse without the 50% larger radius. The '1' rows are also position only culling but includes the 50% larger specular radius. (Full table for 3 more locations in Appendix D, results are similar)

Table 5.2: Predominantly close fragments (A and B) vs predominantly far fragments (C) shading differences. NVIDIA RTX 4070 at 1920x1027 (bordered fullscreen) (cluster grid: $16 \times 9 \times 24 \times N$)

2016 Lights – 16 Sun Temples					
Location	**N**	**Light Ops**	**Compute (ms)**	**Fragment (ms)**	**Total (ms)**
A	dif_1	84499416	0.35	16.09	16.44
B	dif_1	84565432	0.35	15.19	15.54
C	dif_1	268551545	0.35	82.33	82.68
A	6	98094873	1.61	19.91	21.52
B	6	96404152	1.71	18.25	19.96
C	6	273140899	1.58	86.49	88.07

5.2 Evaluation and Discussion

The primary goal of this project was to enable efficient real-time shading of scenes with many polygon area lights, by culling irrelevant lights through an extended clustered shading pipeline. Table 5.1 shows that the lighting approach scales effectively: a 16x increase in triangle and area light count only results in around a 10% increase fragment shading time. This indicates that our method's performance is primarily determined by area light density per pixel rather than total scene complexity, confirming that the culling strategy is effective.

To test generality, three distinct scene location with varying lighting conditions were evaluated. Results are detailed in Appendix D. Location B highlighted an artefact where purely diffuse culling was insufficient, while Locations A and C do not exhibit such artefacts. This reinforces that extending the specular culling radius beyond diffuse may only be necessary when strong distant specular highlights are present (such as scenes with water).

Normal cone culling was crucial for reducing the number of costly light evaluations in our extensions for improved specular. Even with a relatively high number of normal cones (e.g., 54 cones), total frame time was faster than with position-only culling, primarily because shading time was almost halved through more efficient culling of lights outside the diffuse radius. Among the tested configurations, using six normal cones provided a good balance between performance and visual quality across all scenes. To quantify the efficiency gains, we compared the number of specular light operations after culling with different numbers of normal cones against our position-only extended sphere heuristic. We focused this comparison specifically on the outer specular radius, where normal cone culling is applied. The calculation proceeds as follows:

To isolate the outer specular contribution, we subtract the diffuse radius operations (common to all cases):

$$\text{Diffuse radius ops} = 42,893,462$$

Outer specular light operations:

$$\text{Position Specular:} \quad 65,936,949 - 42,893,462 \quad = 23,043,487 \text{ ops} \tag{5.1}$$

$$\text{24 Normal Cones:} \quad 53,645,586 - 42,893,462 \quad = 10,752,124 \text{ ops} \tag{5.2}$$

$$\text{54 Normal Cones:} \quad 47,590,904 - 42,893,462 \quad = 4,697,442 \text{ ops} \tag{5.3}$$

From these results, we observe that using 54 normal cones reduced outer specular operations by approximately 80% relative to position-only culling, while using 24 normal cones achieved a 54% reduction. These savings directly translate into substantial shading time improvements. It was not meaningful to calculate the reduction in specular from 6 normal cones because the current light cone method only accurately tracks specular highlights with ≥ 24 normal cones. Using only 6 cones leads to visible underassignment near the sides of the screen. Additionally, compute timings indicate that using 24 normal cones increases light assignment cost by roughly $4\times$ compared to 6 cones. Thus, improving the fidelity of 6-cone tracking remains an important, very feasible goal for achieving both performance and visual quality.

Demo Video 2 (Appendix C) visualizes the cone artefacts close to lights and the necessity of range-limiting the normal cone method.

Another limitation is the reduced culling efficiency at far depths. Scenes dominated by distant fragments — such as outdoor environments (Location C in Table 5.2, Table D.1, and Figure 5.1) — suffer from high shading costs due to coarse clusterings at large distances. This suggests that the current exponential depth slicing approach needs refinement for open-world or area light dense scenes — such as special systems for far light sources similar to Persson (2015).

Finally, although normal cones were only applied to the outer radius in our implementation, we can estimate the potential savings if normal cone culling were extended across the entire light radius. Specifically, the total light operations could be approximated by:

$$\text{Diffuse ops} = \frac{\text{Inner radius ops}}{2} = \frac{42,893,462}{2} = \text{Inner specular ops} \tag{5.4}$$

$$\text{Total ops} = \text{Diffuse ops} + \frac{\text{Outer cone ops}}{\text{Outer position ops}} \times \text{Inner Specular ops} \tag{5.5}$$

yielding:

$$\text{24 Normal Cones:} \quad \frac{42,893,462}{2} \times \left(1 + \frac{10,752,124}{23,043,487}\right) \approx 31,453,806 \text{ ops} \tag{5.6}$$

$$\text{54 Normal Cones:} \quad \frac{42,893,462}{2} \times \left(1 + \frac{4,697,442}{23,043,487}\right) \approx 25,818,672 \text{ ops} \tag{5.7}$$

compared to perceptible operation counts of 21,447,781 and 20,653,387, respectively (see Appendix I for methodology counting perceptible ops). Thus, extending normal cone culling to the inner radius promises exceptional efficiency, with minimal light evaluations wasted on imperceptible contributions. Unfortunately, we did not complete the full implementation during the project timeline; we discuss this further in our Future Work section.

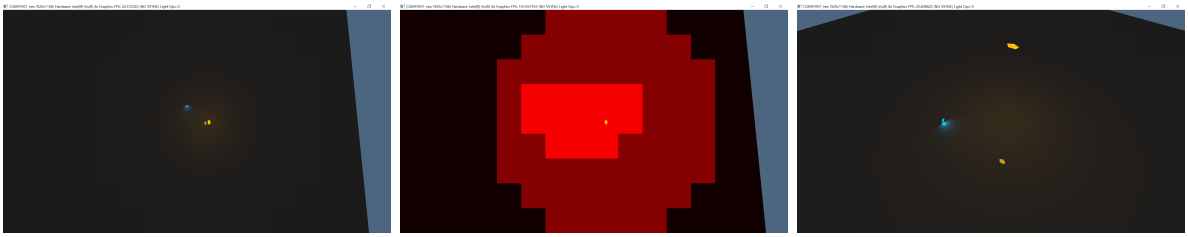

Figure 5.2: Diffuse light assignment extents for two area lights: one blue, oriented perpendicular to the floor, and one orange, angled downward. Top-down views (left, middle) and a side view (right) show that the blue light — aligned with surface normals — has a much smaller effective diffuse radius. However, due to position-only clustering, the red cluster assignments are overly large for the blue light, illustrating the need for normal-aware diffuse culling.

Moreover, the current diffuse bounds are overly conservative because they solely rely on position clustering, which assumes the worst-case falloff across all normal directions. As illustrated in Figure 5.2, horizontal surfaces exhibit much smaller falloff extents. To address this, per-normal-cone diffuse bounding volumes should be computed — adjusting sphere sizes based on the normal cone's orientation relative to the area light. This would enable much tighter and more accurate culling compared to our current singular diffuse bounding sphere approach.

5.3 Future Work

This section outlines both immediate refinements to our method and possible extensions to broader real-time rendering problems, including shadows and global illumination.

To correct cone-based artefacts near light sources, we currently define specular contribution as the union of the light cone and the diffuse volume. However, this introduces excessive coverage in close-range lighting. A better approach would remove the diffuse volume dependency entirely by incorporating the angular size of the polygon and the cluster into the reflected cone radius. I did not get this formulation working in time for this report, but it has the potential to significantly reduce the cost of specular evaluations within the diffuse radius.

Large far clusters contain huge numbers of lights, leading to performance degradation when the screen is dominated by distant fragments (Table 5.2). Linear depth slicing could reduce far cluster sizes but would require an impractically large number of slices, making compute time prohibitively expensive. A more viable solution would be to approximate distant area lights with cheaper representations, such as point or spot lights, or to introduce a Level-of-Detail (LOD) system for light assignment.

While LTC lights inherently exhibit inverse-square attenuation, the diffuse bounds could be made significantly tighter by adopting a modified quadratic attenuation curve, similar to those commonly used for point lights. A sharper falloff at the bounds would allow for tighter culling without introducing perceptible artefacts, while also mitigating the issue where many distant lights cumulatively become visible — an outcome that undermines culling strategies aiming to match the non-culled image.

Our current normal cone method does not trivially hierarchical-ise since each area light has separate light cones per cluster. If tight bounding volumes are instead found per normal cone globally (e.g. each light has 6/24/54 total volumes instead of 6/24/54 per position cluster) then these volumes can be precomputed and used to construct a BVH tree over the lights, and turn the serial part of our light assignment from $\mathcal{O}(\texttt{num\_lights})$ to $\mathcal{O}(\log \texttt{num\_lights})$, which would allow the method to scale to tens or hundreds of thousands of lights with minimal increase in assignment time.

Our renderer does not shadow the lights and this would hugely improve the visuals. There are real-time techniques for shadowing area lights such as raytraced shadows, albeit requiring powerful hardware. With shadowing, an occlusion test could be used against clusters to avoid assigning occluded lights to clusters. It would be interesting to compare whether this trade off of more complex culling reduces the overall render time.

While our current approach focuses on direct lighting, similar culling techniques could be adapted to reduce the cost of global illumination systems, particularly those that approximate indirect lighting in real-time, such as Dynamic Diffuse Global Illumination (DDGI) (Majercik et al., 2021). For example, cone-based or volume-based light assignment could be extended to identify regions that contribute minimally to indirect bounce lighting, allowing for efficient updating of irradiance probes.

5.4 Concluding Remarks

The results and discussion presented here demonstrate that clustered light assignment with normal cone culling is a highly scalable and effective solution for rendering scenes with dense area lighting. While the current implementation already achieves substantial performance gains, the proposed refinements — including inner-radius cone culling, tighter diffuse bounds, and hierarchical cone structures — offer a clear path toward dramatically greater shading efficiency. With these enhancements, area light rendering is poised to become not only viable, but efficient even under extremely high light counts.

References

Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A. and Jarosz, W. (2020). Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting *ACM Trans. Graph.* **39**(4).
URL: *https://doi.org/10.1145/3386569.3392481*

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures *SIGGRAPH Comput. Graph.* **11**(2), 192–198.
URL: *https://doi.org/10.1145/965141.563893*

Burley, B. (2012). Physically-based shading at disney *SIGGRAPH 2012* ACM ,.
URL:
https://media.disneyanimation.com/uploads/production/publication_ asset/48/asset/s2012_ pbs_ disney_ brdf

Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics *ACM Trans. Graph.* **1**(1), 7–24.
URL: *https://doi.org/10.1145/357290.357293*

de Carpentier, G. and Ishiyama, K. (2017). Decima engine: Advances in lighting and aa PowerPoint presentation. Slides from conference presentation.
URL: *https://advances.realtimerendering.com/s2017/DecimaSiggraph2017.pdf*

Epic Games (2017). Unreal engine sun temple, open research content archive (orca).
URL: *https://developer.nvidia.com/ue4-sun-temple*

Fan, Z., Hong, P., Guo, J., Zou, C., Guo, Y. and Yan, L.-Q. (2023). Manifold path guiding for importance sampling specular chains *ACM Trans. Graph.* **42**(6).
URL: *https://doi.org/10.1145/3618360*

Harada, T. (2012). A 2.5d culling for forward+ *SIGGRAPH Asia 2012 Technical Briefs* SA '12 Association for Computing Machinery New York, NY, USA ,.
URL: *https://doi.org/10.1145/2407746.2407764*

Harada, T., McKee, J. and Yang, J. C. (2012). Forward+: Bringing Deferred Lighting to the Next Level *in* C. Andujar and E. Puppo (eds)*Eurographics 2012 - Short Papers* The Eurographics Association ,.

Heitz, E. (2014). Understanding the masking-shadowing function in microfacet-based brdfs *Journal of Computer Graphics Techniques (JCGT)* **3**(2), 48–107.
URL: *http://jcgt.org/published/0003/02/03/*

Heitz, E. (2017). Geometric Derivation of the Irradiance of Polygonal Lights *Research report* Unity Technologies.
URL: *https://hal.science/hal-01458129*

Heitz, E., Dupuy, J., Hill, S. and Neubelt, D. (2016). Real-time polygonal-light shading with linearly transformed cosines *ACM Trans. Graph.* **35**(4).
URL: *https://doi.org/10.1145/2897824.2925895*

Heitz, E. and Hill, S. (2017). Linear-light shading with linearly transformed cosines *in* W. Engel (ed.)*GPU Zen: Advanced Rendering Techniques* Balboa Press ,.
URL: *https://hal.science/hal-02155101*

Hill, S. and Heitz, E. (2017). Real-time area lighting: A journey from research to production Presented at the Game Developers Conference (GDC). Updated: March 18th, 2017.
URL: *https://advances.realtimerendering.com/s2016/s2016_ltc_rnd.pdf*

János, T. (2017). Area lights. Accessed: 2025-04-02.
URL: *https://wickedengine.net/2017/09/area-lights/*

Källberg, L. and Larsson, T. (2014). Optimized phong and blinn-phong glossy highlights *Journal of Computer Graphics Techniques (JCGT)* **3**(3), 1–6.
URL: *http://jcgt.org/published/0003/03/01/*

Karis, B. (2013). Real shading in unreal engine 4 *Technical report* Epic Games.
URL: *http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf*

Khronos Group (2024). Interface block (glsl) - opengl wiki. Accessed: 2025-04-10.
URL: *https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)*

Lambert, J. H. (1760). *Photometria, sive de mensura et gradibus luminis, colorum et umbrae* Sumtibus Eberhardi Klett Augsburg.

Lecocq, P., Dufay, A., Sourimant, G. and Marvie, J.-E. (2016). Accurate analytic approximations for real-time specular area lighting *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* I3D '16 Association for Computing Machinery New York, NY, USA p. 113–120.
URL: *https://doi.org/10.1145/2856400.2856403*

Lecocq, P., Dufay, A., Sourimant, G. and Marvie, J.-E. (2017). Analytic approximations for real-time area light shading *IEEE Transactions on Visualization and Computer Graphics* **23**(5), 1428–1441.

Majercik, Z., Müller, T., Keller, A., Nowrouzezahrai, D. and McGuire, M. (2021). Dynamic diffuse global illumination resampling *Proceedings of High Performance Graphics* ,.
URL: *https://research.nvidia.com/publication/2021-12_dynamic-diffuse-global-illumination-resampling*

McGuire, M. (2019). Part 3: Ddgi overview – dynamic diffuse global illumination. Accessed: 2025-04-18.
URL: *https://morgan3d.github.io/articles/2019-04-01-ddgi/overview.html*

Nicodemus, F. E. (1965). Directional reflectance and emissivity of an opaque surface *Appl. Opt.* **4**(7), 767–775.
URL: *https://opg.optica.org/ao/abstract.cfm?URI=ao-4-7-767*

Olsson, O. and Assarsson, U. (2011). Tiled shading *Journal of Graphics* **GPU**, 235–251.

Olsson, O., Billeter, M. and Assarsson, U. (2012). Clustered deferred and forward shading *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* EGGH-HPG'12 Eurographics Association Goslar, DEU p. 87–96.

Persson, E. (2015). Practical clustered shading. GDC Talk, Avalanche Studios.
URL: *https://www.gamedevs.org/uploads/practical-clustered-shading.pdf*

Peters, C. (2021). Brdf importance sampling for polygonal lights *ACM Trans. Graph.* **40**(4).
URL: *https://doi.org/10.1145/3450626.3459672*

Phong, B. T. (1975). Illumination for computer generated pictures *Commun. ACM* **18**(6), 311–317.
URL: *https://doi.org/10.1145/360825.360839*

Schlick, C. (1994). An inexpensive brdf model for physically-based rendering *Computer Graphics Forum* **13**(3), 233–246.

Sikachev, P., De Francesco, G. D., Nowakowski, K. and Kowalczyk, K. (2021). Area light sources in cyberpunk 2077 *ACM SIGGRAPH 2021 Talks* SIGGRAPH '21 Association for Computing Machinery New York, NY, USA ,.
URL: *https://doi.org/10.1145/3450623.3464630*

Thomas, G. (2015). Advancements in tiled-based compute rendering Presented at the Game Developers Conference (GDC). Talk.
URL:
https://media.gdcvault.com/gdc2015/presentations/Thomas_ Gareth_ Advancements_ in_ Tile-Based.pdf

Torrance, K. E. and Sparrow, E. M. (1967). Theory for off-specular reflection from roughened surfaces∗ *J. Opt. Soc. Am.* **57**(9), 1105–1114.
URL: *https://opg.optica.org/abstract.cfm?URI=josa-57-9-1105*

Trowbridge, T. S. and Reitz, K. P. (1975). Average irregularity representation of a rough surface for ray reflection *J. Opt. Soc. Am.* **65**(5), 531–536.
URL: *https://opg.optica.org/abstract.cfm?URI=josa-65-5-531*

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

Appendix A

Self-appraisal

A.1 Critical self-evaluation

Much of the algorithmic work was developed late in the project's timeline. As a result, I found myself repeatedly regenerating results in the final two months, continually refining and rethinking my approach. A significant portion of the development effort went into building the renderer from scratch, which formed the foundation for the clustering algorithms. This left limited time which meant my future work section became overran with all my planned ideas that I did not have time to get working. Despite these constraints, I believe the results are solid and align well with the core goals I originally set.

I flooded my test scene densely with large area lights, which made the performance sometimes seem a bit underwhelming — given performance would still be limited here with perfect culling and instead would require faster light evaluation like LODing different approximations in further clusters. Not only should I probably have used smaller area lights throughout the scene, but I also should have programmed a way to spawn them in rather than what I actually did which was manually place 200 lights, save them and then mirror those sixteen times across the scene. I would have like to programmatically spawn more area lights in throughout the scene, with the area light size automatically decreasing as more were spawned in, allowing testing of more lights, with the same density, then I could have graphed more detailed results on the differences between small and large amounts of lights. Instead I only had the 200 and 2000 light scenes, which while demonstrated scalability, does not show the function that represents the complexity, although we can infer assignment time scales linearly because of my bruteforce implementation.

While I could have tried extending an off-the-shelf renderer, which may have gave me more time to develop clustering algorithms. The experience I gained in doing everything from scratch gave me a much greater intuition for the problem and techniques involved. This helped me develop the approaches I ended up using and allowed my write-up to be detailed, reasoned, and hopefully gives uninitiated readers a better intuition too, given the thorough background research that I ended up needing to do.

A.2 Personal reflection and lessons learned

The renderer I developed for this project functioned well overall, but its design was tightly coupled to the glTF format. While this approach was suitable for the project's scope and did

not hinder progress, my understanding has since evolved. In retrospect, I would prefer a renderer architecture more akin to a general-purpose game engine — where internal scene structures are better suited for dynamic objects and not tied to a specific file format. Such a design would facilitate broader asset compatibility, including formats like OBJ.

Before undertaking this project, I had no experience with compute shaders, physically based rendering, or handling large numbers of dynamic lights. Working through these challenges significantly deepened my understanding of renderer architecture. I learned the importance of system-level design decisions — such as draw call organization, the need for effective culling strategies, and the trade-offs involved in real-time performance.

One key insight was the value of designing for flexibility. Initially, the renderer executed draw calls immediately, which worked fine until I introduced transparency. Supporting it required storing draw calls in dynamic arrays for later sorting and execution, necessitating extensive changes across the codebase. This experience highlighted the benefit of designing with future extensibility in mind, even when aiming for a lean implementation. Looking back, the decision to implement only the features I immediately needed helped keep development focused. However, I now have a clearer sense of how to balance minimalism with foresight — structuring systems in a way that allows them to evolve without requiring large-scale rewrites. With the benefit of hindsight, I can now make more sustainable architectural choices from the outset.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

This project used several lightweight open-source libraries to assist with cross-platform support, scene loading, and UI. All libraries used were permissively licensed and suitable for both academic and commercial projects:

- **GLFW** — Licensed under the [zlib/libpng license](https://www.glfw.org/license.html), which permits free use, modification, and distribution.
- **glad** — Licensed under the [MIT License](https://github.com/Dav1dde/glad/blob/master/LICENSE), allowing unrestricted use and distribution, provided the license is included.
- **cgltf** — Licensed under the [MIT License](https://github.com/jkuhlmann/cgltf/blob/master/LICENSE), which is simple and permissive.
- **stb_image.h** — Part of the stb single-header library collection, under [public domain or MIT license](https://github.com/nothings/stb/blob/master/LICENSE), depending on jurisdiction.

- **cglm** — Licensed under the [MIT License](https://github.com/recp/cglm/blob/master/LICENSE), encouraging wide reuse and modification.
- **Nuklear** — Licensed under [public domain or MIT license](https://github.com/Immediate-Mode-UI/Nuklear/blob/master/LICENSE), suitable for both commercial and academic use.

No proprietary or closed-source software components were used in the final project. All reused materials were properly credited, and no license terms were violated.

A.3.2 Social issues

A notable social consideration of this project is its potential to support a more open and diverse game development ecosystem. Industry trends increasingly favour large, full-featured engines like Unreal Engine 5, which provide state-of-the-art rendering capabilities but often come with high complexity and limited flexibility for developers seeking low-level control.

By focusing on a lighting technique that is both efficient and relatively simple to implement — as detailed in this report and supported by supplemental source code — this project aims to make advanced rendering more accessible to developers building their own engines or working in lightweight environments. This is particularly valuable in educational contexts, for indie developers, and in research projects where large engines may be impractical or overly restrictive.

Encouraging simpler alternatives can help maintain diversity in the tools and approaches used in the real-time rendering space. In the long term, this may contribute to reducing monopolistic tendencies in the engine market, fostering innovation and user autonomy.

A.3.3 Ethical issues

The field of real-time rendering is often used systems like games, CAD, etc. that are deployed at a large scale with millions of users. High fidelity real-time rendering techniques stay up to date with the development of increasingly powerful hardware, and so the use of the highest fidelity techniques can encourage the user base to adopt more newer hardware — which has a higher environmental impact due to increased power consumption and E-waste (unless the old, replaced hardware is repurposed).

Our project however, is used to make an existing technique more accessible by making it more efficient. Although notably, the purpose was to deploy the technique en masse, reflecting a common occurrence for efficiency gains where we exploit the efficiency gains to render more area lights instead of reduce total energy consumption.

A.3.4 Professional issues

This project did not involve collaboration with external clients, users, or commercial stakeholders, so professional concerns such as client interaction, requirements gathering, or contractual obligations were not directly relevant.

However, professional standards were maintained throughout the development process for keeping the reference implementation understandable. The code includes various preprocessor flags, some involving shader metaprogramming, to toggle between different techniques and configurations for testing and experimentation. While components like light operation counting for benchmarking are not directly relevant to the final implementation, they played an important role in optimizing and evaluating performance during development. Despite the inclusion of these experimental elements, the core components required for a direct and efficient implementation of the lighting technique are well-documented and structured in a way that makes them easily understandable and adaptable for future applications.

Appendix B

External Material

- Sun Temple Model: Due to issues downloading Nvidia's Sun Temple model, I used an existing OBJ file provided by my supervisor, Markus Billeter, and converted it to glTF 2.0 using Blender.
- https://github.com/DaveH355/clustered-shading: This GitHub repo was very helpful to get up and running with clustered shading and OpenGL compute shaders — although my final implementation necessarily differs greatly.
- https://learnopengl.com/Guest-Articles/2022/Area-Lights: There were also issues downloading Heitz et al. (2016)'s provided materials. Luckily, this article had the fitted GGX LTC textures which I used in my solution. The article also mentioned later improvements that I likely would not have spotted like the vector form of the cubic approximation of arccos for horizon clipped form factors.

Appendix C

Demo Videos

Demo 1 was recorded with `ffmpeg`, which had some recording artefacts so the video occasional appears to drop frames.

- **Demo Video 1 (Main Showcase):** Real-time flythrough of 2016-light scene using diffuse-only bounds. [URL: `https://youtu.be/hqbCelGbMRc?si=f3NnEI9hrKSv9t3w`]
- **Demo Video 2:** Visual demonstration of normal cones, showing need for range limitations and associated artefacts. [URL: `https://youtu.be/m9sKT003kLM?si=kzoV1Hw1jvnT6j5O`]

Appendix D

Results Across Locations

Table D.1: Performance on NVIDIA GTX 1060 at 1280x720 (cluster grid: $16 \times 9 \times 24 \times N$)

N	Light Ops	Compute (ms)	Fragment (ms)	Total (ms)
126 Lights – Location A				
1	65936949	0.43	105.55	105.98
6	51355552	2.8	88.7	91.5
24	53645586	11.0	85.7	96.7
54	47590904	24.75	73.0	97.75
dif	42893462	0.43	59.6	60.04
2016 Lights – Location A				
1	70101700	1.60	119.7	121.3
6	51519648	8.15	84.0	92.15
24	55694960	32.2	95.5	127.7
54	48230849	72.2	80.45	152.65
dif	42915180	1.5	65.45	66.95
126 Lights – Location B				
1	60767215	0.43	84.3	84.73
6	56786818	2.8	77.0	79.8
24	58069784	10.0	81.16	91.16
54	54220412	26.6	76.0	102.6
dif	52208794	0.43	66.9	67.34
2016 Lights – Location B				
1	65528147	1.68	99.5	101.18
6	57228839	11.1	83.36	94.46
24	65528147	1.5	98.7	100.2
54	55243963	73.0	81.0	154.0
dif	52544704	1.52	73.29	74.81
2016 Lights – Location C				
1	N/A†	1.54	295.0	296.54
6	132055610	8.6	273.0	281.6
24	N/A†	32.5	328.0	360.5
54	131789980	71.4	303.5	374.9
dif	N/A†	1.61	260.0	261.61

†Light Ops unavailable due to GPU driver failure caused by excessive atomic counter pressure.
Note: **dif** rows are reference timings where diffuse and specular are captured by only the diffuse without the 50% larger radius.

Appendix E

Variable Sized Polygonal Lights in Video Memory

Polygonal lights can be differently sized polygons, but for constant time access they are stored in one video memory array, called a Shader Storage Buffer Object (SSBO), by storing a max number of vertices for all of them such as 10, and then storing a number n, for the size of each specific n-gon. On the CPU side, I store area lights in a dynamic array, and then simply copying the array verbatim to the SSBO each frame since we assume every light is dynamic, in OpenGL we copy via a mapped buffer. The only difference is on the GPU side, the lights are stored in viewspace rather than worldspace. The SSBO is reallocated whenever the number of lights changes.

```
// CPU side AreaLight in C:
typedef struct AreaLight
{
    vec4 color_rgb_intensity_a;
    int n;
    int is_double_sided;
    float _packing0, _packing1;

    // For clustered shading
    vec4 min_point;
    vec4 max_point;
    vec4 sphere_of_influence_center_xyz_radius_w;

    vec4 points_worldspace[MAX_UNCLIPPED_NGON];
}
AreaLight;

// GPU side AreaLight in glsl:
struct AreaLight
{
    vec4 color_rgb_intensity_a;
    int n;
    int is_double_sided;
    float _packing0, _packing1;
    vec4 aabb_min;
    vec4 aabb_max;
    vec4 sphere_of_influence_center_xyz_radius_w;
    vec4 points_viewspace[MAX_UNCLIPPED_NGON];  // 4th component unused, vec3[]
    would be packed the same way but vec3 is implemented wrong on some drivers
};
```

SSBO Packing Requirements When transferring data from the CPU to a GPU Shader Storage Buffer Object (SSBO), it's crucial that the layout and alignment of struct members match the GPU's expected memory layout. In GLSL, SSBOs typically use the `std430` layout, which imposes strict alignment rules — for example, textttvec4 types must be aligned to 16 bytes, and even smaller types like textttint or textttfloat may require padding to avoid misalignment.

Note that its common practice to avoid `vec3` types and use vec4 instead as its not always clear how they will be packed, here is an excerpt from the OpenGL's documentation about the `std140` layout, which may or may not apply to `std430` (Khronos Group, 2024):
Warning: Implementations sometimes get the std140 layout wrong for vec3 components. You are advised to manually pad your structures/arrays out and avoid using vec3 at all.

To ensure compatibility, I designed the AreaLight struct on the CPU to match the GLSL layout exactly. This is why you see padding fields (such as _packing0, _packing1) to account for gaps introduced by alignment rules. Without this alignment, the GPU may interpret fields incorrectly, leading to subtle and difficult-to-debug rendering issues. Knowing how your compiler will pad your C structures is necessary for correctness in this step. By carefully matching packing rules and aligning data consistently, I was able to directly memcpy the CPU-side struct to the GPU SSBO each frame, enabling efficient and error-free light data uploads.

This attention to alignment and packing not only ensures correctness but also avoids the performance penalties that can arise from misaligned memory access on the GPU.

Appendix F

Per Workgroup/Warp Light Assignment

Here is the approach that ended up being 25 times faster for bruteforce polygonal light assignment than one cluster per thread assignment:

In order to accelerate clustered light assignment — particularly under heavy area light loads — we implemented a per workgroup light assignment scheme using shared memory and cooperative testing. Instead of assigning one cluster per thread, each workgroup processes a single cluster, declared via:

```
layout (local_size_x = 32, local_size_y = 1, local_size_z = 1) in;

// Declare a shared instance of Cluster for the workgroup.
shared Cluster cluster;
```

Each thread in the workgroup processes a slice of the global lights array, using standard parallel techniques where local ID 0 initialises and finalises, and using memory barriers and atomics appropriately:

```
void main()
{
    // Load the cluster from global memory into shared memory.
    uint index = gl_WorkGroupID.x;
    if (gl_LocalInvocationID.x == 0)
    {
        cluster = clusters[index];

        // Reset our light counts.
        cluster.point_count = 0u;
        cluster.area_count = 0u;
    }
    barrier();  // Ensure all threads see the loaded cluster

    uint clusters_per_layer = CLUSTER_GRID_SIZE_X * CLUSTER_GRID_SIZE_Y;
    uint combined_z = index / clusters_per_layer;
    uint remainder = index % clusters_per_layer;
    uint normal_bin = combined_z % CLUSTER_NORMALS_COUNT;

    NormalCone normal_cone;
    // (...) create normal cone

    const uint max_point_lights = CLUSTER_MAX_LIGHTS/2;
    const uint max_area_lights = CLUSTER_MAX_LIGHTS/2;

    uint total_threads = gl_WorkGroupSize.x;
    uint local_thread_id = gl_LocalInvocationID.x;
```

```
28
29      // Each thread processes a slice of point lights.
30      for (uint i = local_thread_id; i < num_point_lights; i += total_threads)
31      {
32          if (test_sphere_aabb(i, cluster, normal_cone))
33          {
34              uint point_index = atomicAdd(cluster.point_count, 1u);
35              if (point_index < max_point_lights)
36              {
37                  cluster.point_indices[point_index] = i;
38              }
39          }
40      }
41
42      barrier();
43
44      // And then a slice of the area lights
45      for (uint i = local_thread_id; i < num_area_lights; i += total_threads)
46      {
47          uint contribution_flags = test_arealight(i, cluster, normal_cone);
48          if (contribution_flags != 0u)
49          {
50              uint area_index = atomicAdd(cluster.area_count, 1u);
51              if (area_index < max_area_lights)
52              {
53                  cluster.area_indices[area_index] = i;
54                  cluster.area_light_flags[area_index] = contribution_flags;
55              }
56          }
57      }
58
59      barrier();
60
61      if (gl_LocalInvocationID.x == 0)
62      {
63          clusters[index] = cluster;
64      }
65  }
```

Appendix G

Corrected acos Approximation for LTC evaluation

Here is the full integration of the Lambertian hemisphere with the necessary improved acos approximation (Hill and Heitz, 2017) used in the LTC irradiance evaluation.

```
vec3 integrate_edge_sector_vec(vec3 point_i, vec3 point_j)
{
    float x = dot(point_i, point_j);
    float y = abs(x);
    float a = 0.8543985 + (0.4965155 + 0.0145206 * y) * y;
    float b = 3.4175940 + (4.1616724 + y) * y;
    float v = a / b;
    float theta_sintheta;
    if (x > 0.0)
    {
        theta_sintheta = v;
    }
    else
    {
        theta_sintheta = 0.5 * inversesqrt(max(1.0 - x*x, 1e-7)) - v;
    }
    return cross(point_i, point_j) * theta_sintheta;
}

vec3 integrate_lambertian_hemisphere(vec3 points[MAX_UNCLIPPED_NGON], int n)
{
    vec3 vsum = integrate_edge_sector_vec(points[n-1], points[0]);  // Start
    with the wrap around pair (n-1, 0)
    for (int i = 0; i < n-1; ++i)
    {
        vsum += integrate_edge_sector_vec(points[i], points[i+1]);
    }
    return vsum;
}
```

Appendix H

Details for `test_arealight()`

H.1 Half-Space Rejection

The `lessThan` GLSL vector operation is used to efficiently choose the correct min/max corners for this projection. We skip this slightly more costly operation for the clusters where the naive centre test fails.

```
vec3 cluster_center = (cluster.min_point.xyz + cluster.max_point.xyz) * 0.5;

// Half space rejection for single sided area lights
if (area_lights[i].is_double_sided == 0)
{
    vec3 p0 = area_lights[i].points_viewspace[0].xyz;
    vec3 p1 = area_lights[i].points_viewspace[1].xyz;
    vec3 p2 = area_lights[i].points_viewspace[2].xyz;
    vec3 light_normal = cross(p1 - p0, p2 - p0);

    // Fast center test before exact check
    if (dot(cluster_center - p0, light_normal) < 0)
    {
        float plane_constant = -dot(light_normal, p0);
        vec3 furthest_point = mix(
            cluster.max_point.xyz, cluster.min_point.xyz,
            lessThan(light_normal, vec3(0))
        );
        if (dot(light_normal, furthest_point) + plane_constant < 0)
        {
            return 0u;
        }
    }
}
```

Listing H.1: Half-space early rejection, the first step of `test_arealight()`

H.2 Polygon Area Used for Diffuse Influence Radius

```
float polygon_area(AreaLight* al)
{
    if (al->n < 3) return 0.0f;

    // Find normal from first 3 points
    vec3 u, v, normal;
    glm_vec3_sub(al->points_worldspace[1], al->points_worldspace[0], u);
```

```
    glm_vec3_sub(al->points_worldspace[2], al->points_worldspace[0], v);
    glm_vec3_cross(u, v, normal);
    glm_vec3_normalize(normal);

    // Construct an orthonormal basis
    vec3 tangent, bitangent;
    if (fabsf(normal[0]) > fabsf(normal[1]))
    {
        glm_vec3_cross((vec3){0,1,0}, normal, tangent);  // Try y-up first
    }
    else
    {
        glm_vec3_cross((vec3){1,0,0}, normal, tangent);  // Otherwise, x-right
    }
    glm_vec3_normalize(tangent);
    glm_vec3_cross(normal, tangent, bitangent);

    // Project the polygon onto the new 2D basis
    vec2 projected[MAX_UNCLIPPED_NGON];
    for (int i = 0; i < al->n; ++i)
    {
        vec3 p;
        glm_vec3_sub(al->points_worldspace[i], al->points_worldspace[0], p);
        projected[i][0] = glm_vec3_dot(p, tangent);
        projected[i][1] = glm_vec3_dot(p, bitangent);
    }

    // Compute the area using the Shoelace theorem
    float area = 0.0f;
    for (int i = 0; i < al->n; ++i)
    {
        int j = (i + 1) % al->n;
        area += projected[i][0] * projected[j][1] - projected[j][0] * projected[
    i][1];
    }

    area = fabsf(area) * (al->is_double_sided ? 1.0f : 0.5f);

    return area;
}
```

Listing H.2: C code that projects the polygon to 2D along its normal and applies the Shoelace formual to find the area accurately.

H.3 Normal Cone Based Specular Testing

```
struct Cone
{
    vec3 axis;
    float angle;  // Half angle
};
```

```
6
7  Cone compute_specular_cone(vec3 cluster_center, Cone cluster_cone, vec3
       light_center, float light_geo_radius, float max_spec_angle)
8  {
9      Cone light_cone;
10     light_cone.axis = normalize(reflect(light_center - cluster_center,
       cluster_cone.axis));
11     light_cone.angle = max_spec_angle + cluster_cone.angle * 2.0;
12
13     return light_cone;
14 }
15
16 bool specular_visible(Cone light_cone, vec3 view_dir)
17 {
18     float a = acos(dot(light_cone.axis, normalize(-view_dir)));
19     return a <= light_cone.angle;
20 }
21
22 uint test_arealight(uint i, Cluster cluster, NormalCone nc)
23 {
24     // Half Space   (...)
25     // Diffuse Test (...)
26
27     // Specular test
28     bool specular_passed = false;
29
30     Cone cluster_cone;
31     cluster_cone.axis = nc.cluster_normal;
32     cluster_cone.angle = nc.half_angle;
33
34     Cone light_cone = compute_specular_cone(
35         cluster_center,
36         cluster_cone,
37         sphere.xyz,
38         radians(45.0)
39     );
40     specular_passed = specular_visible(light_cone, view_dir);
41
42     // Include diffuse term since this breaks down near the light otherwise
43     specular_passed = specular_passed || diffuse_passed;
44
45     // Sadly, we have to include a range limit too, otherwise we assign far too
       many distant clusters:
46     specular_passed = specular_passed && sphere_aabb_intersection(sphere.xyz,
       1.5 * sphere.w, cluster.min_point.xyz, cluster.max_point.xyz);
47
48     // (...)
49 }
```

Listing H.3: Specular cone test for area lights. The cone axis is a reflection of the cluster's normal axis. Angle includes BRDF specular lobe width and optional light/cluster angular size.

H.4 Diffuse Extra: Oriented Bounding Boxes

The oriented bounding box (OBB) can be created the same way as the AABB, except in its local coordinate system. To precompute an OBB you need the direction the polygon is most stretched along i.e. it's eigen vector, but instead of doing principle component analysis, simply find the longest edge (v_i, v_{i+1}) to get the primary axis $\vec{L} = \vec{v}_{i+1} - \vec{v}_i$. Then the other axis are the polygon normal — which you can compute with the cross product:
$\vec{N} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)$ — and then finally $\vec{L} \times \vec{N}$.

We only implemented spheres and AABBs, for full game engines, a good approach instead of long thin polygonal lights with OBBs would be to implement LTC line lights, solved by Heitz and Hill (2017). Then you could bound them by finding the closest point of the AABB to the line, and test it against the same influence radius.

Appendix I

Definition of a perceivable light operation to count redundant LTC light operations

We count diffuse and specular light operations separately as they are distinct `LTC_evaluate()` calls, so when only one of those components contribute to perceivability we increment by 1, if both do, we increment by 2. We ignore the effect of colour maps on perceivability and so we consider a light operation perceivable, if it's luminance transferred to sRGB is above 0.01.

```
vec3 white_specular_radiance = al.color_rgb_intensity_a.a * specular;
vec3 white_diffuse_radiance = al.color_rgb_intensity_a.a * vec3(1.0) * diffuse;
vec3 both = white_specular_radiance + white_diffuse_radiance;

float perceptual_luminance_spec = dot(pow(white_specular_radiance, vec3(
    INV_GAMMA)), vec3(0.2126, 0.7152, 0.0722));
float perceptual_luminance_diffuse = dot(pow(white_diffuse_radiance, vec3(
    INV_GAMMA)), vec3(0.2126, 0.7152, 0.0722));
float perceptual_luminance_both = dot(pow(both, vec3(INV_GAMMA)), vec3(0.2126,
    0.7152, 0.0722));

const float luminance_threshold = 0.005;  // half 0.01 as diffuse and specular
    sum together

if (perceptual_luminance_spec > luminance_threshold !=
    perceptual_luminance_diffuse > luminance_threshold)
{
    atomicCounterIncrement(light_ops_atomic_counter_buffer);
}
else if (perceptual_luminance_both > 2*luminance_threshold)
{
    atomicCounterIncrement(light_ops_atomic_counter_buffer);
    atomicCounterIncrement(light_ops_atomic_counter_buffer);
}
```

Perceptible light operations:

Specular 1 (Position-only):	65,936,949 ops	22,733,682 perceptible ops	(I.1)
24 Normal Cones:	53,645,586 ops	21,447,781 perceptible ops	(I.2)
54 Normal Cones:	47,590,904 ops	20,653,387 perceptible ops	(I.3)