Stereoscopic Clustered Light Shading

by

Mick van Gelderen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday 14 February 2020 at 16:00.

Student number	4091566	
Project duration	April 2018 - February 2020	
Thesis committee	Prof. Elmar Eisemann (supervisor)	CGV, TU Delft
	Dr. David Tax	PRB, TU Delft
	Dr. Markus Billeter (daily supervisor)	CGV, TU Delft

TU Delft

Abstract

Real-time realistic rendering requires the evaluation of the influence of many light sources. In case of dynamic geometry or light sources, this evaluation must be performed every frame. In this thesis I present enclosed clustering: an adaptation of Clustered Light Shading to stereoscopic rendering which cuts the per-eye cost of light assignment in half. To achieve this, clustering is performed once with a clustering camera frustum that encloses both of the eye camera frusta. Decoupling of the clustering camera from the rendering camera gives way to two alternative ways of constructing the clustering: orthographic clustering and displaced perspective clustering. Orthographic clustering uses a uniform world-space grid and has been traditionally dismissed. Displaced perspective clustering uses a decoupled clustering camera to reduce the number of small clusters near the camera. These techniques can be applied to both monoscopic and stereoscopic rendering.

Keywords: Real-Time Graphics, Clustered Light Shading, Stereoscopic Rendering, Virtual Reality, Light Culling.

Preface

Silly as it may be, completing this thesis was one of the hardest things I have done in my life. I consider myself lucky to be able to say my life has been smooth sailing, especially in terms of education. My master thesis however, was different. At the beginning I didn't even know where to start. I would be working on this project for almost a year and deciding on "the right" subject, when there are so many interesting subjects to pick, seemed important. Additionally, it felt almost impossible to produce a thesis which makes a proper scientific contribution, something I understood to be essential for a high-quality master thesis, as I was competing with developers and researchers with many years of experience. After drifting around in the sea of project proposals for some time, I finally found an understandable project with a clear goal thanks to my supervisor, Elmar Eisemann, and got really exited.

At the very start of the project, I decided I would develop my own renderer. I imagined myself knowing what I wanted to achieve, but unable to figure out how to actually do it in some existing rendering engine. I would much rather learn what goes into creating an engine, than to learn how to use one. Obviously, this decising was going to, and definitely did, cost me dearly in terms of time and effort. In the end however, I am happy I did. I am very proud of the renderer I wrote, as it is performant and has many useful features like automatic shader and configuration reloading, state recovery on restart, proper profiling, the ability to record and replay user input, fast model loading, and more. As the renderer is somewhat complex and developed over a year without previous experience, a lot of maintenance work was involved as I needed to refactor the code frequently to keep it performant and extensible. Of course, corners were cut and the resulting code has rough edges. For example, adding a new shader programs or a new shader variable requires changes in multiple places which creates an unnecessary barrier to experimentation. In the past year I gained a lot of experience and also a lot of appreciation for existing rendering engines like Unity and Unreal Engine.

Working on the renderer consistently was a challenge. I can take a set-back or two, but when my screen is completely black for the third time, my willpower is challenged. If I can't find the problem after re-reading both the code and the documentation multiple times, it becomes really easy to give up. Fortunately, one of the most knowledgeable and kind people I know, Markus Billeter, was there to support me when I needed it the most. I also want to thank Nestor Salamon, Leonardo Scandolo, Mijael Bueno Perez, Baran Usta, Ahmad Nasikun, and many others from the Computer Graphics and Visualisation group at TU Delft for their general support. I will always remember the fun and helpful conversations we have had and the reassurance you gave me. I also want to thank my friends Oliver Lee, Olivier Hokke, Diego Valdivia and my family for their support in rough times, I love you guys. Without these and many more people than reasonably fit on this page, this thesis would have never seen the light of day.

Speaking of light...

Mick van Gelderen Delft, February 2020

Contents

1

Introduction

Computer graphics is the science of generating images with the aid of computers. It has applications in many industries. For example: animators use computer graphics to create animated movies, doctors look at 3D colored visualisations generated from raw medical scan data, and realistic renders of products are used for marketing purposes. Because of the variety in applications, there are many different fields within computer graphics. Sections 1.1 through 1.3 introduce real-time graphics, rendering and virtual reality, guiding the reader to the area of interest for this thesis. Section 1.4 describes my contributions and the structure of this thesis.

1.1 Real-Time Graphics

We can categorize the time-frame within which an image needs to be produced into offline-, interactive-, and real-time methods. Offline methods are used in, for example, the movie industry. There exist so called *render farms*, where many computers together render high-quality images and animations. Interactive methods are capable of producing images in a few seconds. These are seen in visualizations in the medical domain and previews generated by 3D modelling software. Real-time methods is the most constrained category. The time in which an image must be produced is so small that we perceive it as instantaneous. This is important for many games, but also for immersive training simulations used in the training of, for example, firemen and doctors.

The work in this thesis is concerned with real-time graphics. Real-time means that we can generate a new image at the display's refresh rate, which has traditionally been around 60 times per second. Commercial displays with even higher frame rates are starting to show up, like 90Hz for the HTC Vive (where we have to generate two images) and 240Hz for some gaming displays. The higher the frame rate, the less time there is to compute a new image.

1.2 Physically-based Rendering

Physically-based rendering, within the context of computer graphics, is the act of generating an image that resembles reality. Without light we can not see anything. It follows naturally that when we render images of virtual worlds, light plays an important role. In fact, when we render an image, we are actually simulating light.

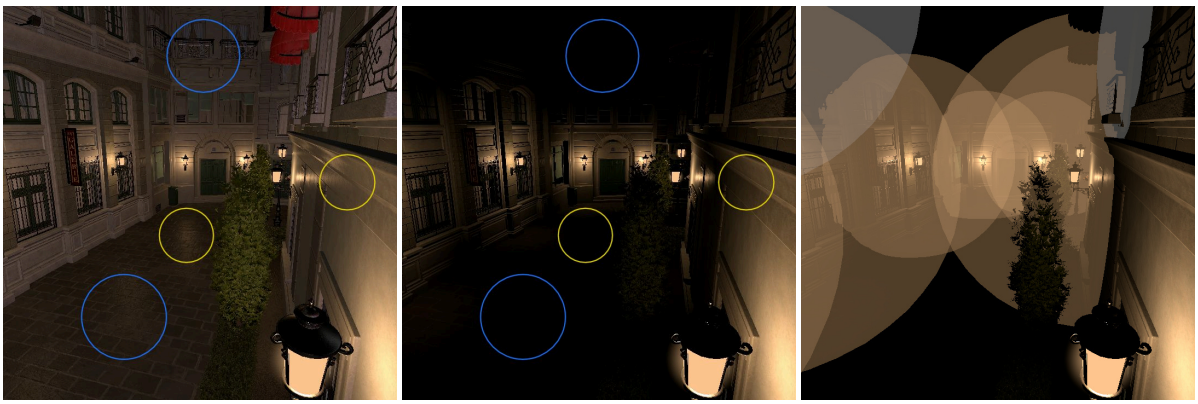
To render an image we need geometry, light sources and a camera. The geometry in a scene is usually modelled as a collection of triangles. The light sources illuminate the scene. The camera is a

description of an observer in the scene. It defines how each pixel in the to be rendered image is affected by incoming light.

There are two different approaches to rendering: ray-tracing and rasterization. Ray tracing is based on the direct simulation of light rays travelling through the scene. This has to be done intelligently because of the many possible paths light can take. Rasterization instead determines what geometry is visible and then computes the contribution of light. Rasterization has better performance but effects like shadows and reflections are far from trivial to compute. While ray-tracing is growing in popularity, the majority of real-time applications still use only rasterization or a combination of the two techniques. The work in this thesis is evaluated using a rasterization based renderer. However, the discussed techniques can also be applied to ray-tracing.

In this thesis we are especially interested in how we represent and efficiently compute the contribution of each light source on all visible surfaces. Imagine that we have a display of 1920 by 1080 pixels. We have to compute for every pixel what piece of geometry can be seen. For each visible piece of geometry we must compute how much light we can see bouncing off of it. This means that for each of the 2 million pixels we must compute the contribution of each light source. With enough light sources this computation becomes too slow for real-time applications, even when not considering light that is bouncing between scene elements (indirect illumination).

For static scenes and lights, the light contributions can be pre-computed for all the geometry in the scene. This has been explored in depth and yields great looking results. However, this so-called *static lighting* no longer works if we start to move the geometry or lights around because it relies on their positions always being the same. The number of dynamic light sources used in scenes has typically been very limited. The work in this thesis assumes that it must be possible to change the lights and geometry freely at any time.



(a) *Unlimited Influence* **(b)** *Limited Influence* **(c)** *Light Influence Volumes*

Figure 1.1: *By limiting the influence of of lights we can improve rendering times. However, when doing so light emitted from the light sources no longer travels as far as it should. Figure 1.1a shows what the scene looks like without limiting the light influence. In Figure 1.1b the contribution of each light is limited to a small sphere centered on the light. Figure 1.1c shows these spheres. The spheres were intentionally chosen to be smaller than usual to clearly illustrate the consequences of this approximation. Unfortunately, (specular) reflections of a light on surfaces outside of its influence volume vanish (yellow circles). Additionally, the combined contribution of many small lights on distant surfaces is lost (blue circles). Regardless of these consequences, this approximation is used in real-time applications to keep shading times low.*

Most light sources only contribute to a small region in the scene. Rather than evaluating the contribution of all light sources everywhere, we limit the influence of each light to an *influence volume* as illustrated in Figure 1.1. This volume is necessarily smaller than the real influence volume, which is theoretically unbounded. For point lights, which emit light equally in all directions, we choose the influence volume to be a sphere centered on the point light. When evaluating the contribution of light sources on a point of a surface (shading), we can now only consider the lights of which the influence volume covers the point on the surface. There are multiple ways in which this can be done as discussed in Chapter 2. The work in this thesis builds on a specific variant called Clustered Light Shading (CLS)

[OBA12]. The basic idea of CLS is to determine a list of relevant lights for groups of points on visible surfaces. To shade a surface points, the list of relevant lights is retrieved and processed than all lights in the scene. CLS will be explained in detail in Chapter 3.

1.3 Virtual Reality

Virtual Reality (VR) has applications in for example 3D modelling [RRS19], museum experiences [Woj+04] and, even back in 1993, the training of surgeons [Sat93]. Recently, VR has become more popular due to affordable consumer-ready Head-Mounted Displays (HMDs). To create a VR experience, we must at the minimum consider one specific property of the human visual system: *binocular vision*. Binocular vision is a type of vision where the brain of an organism is able to construct a perception of our three-dimensional environment by combining the images seen by its left and right eye. The images of the both eyes together are called a *stereoscopic* image. An example is shown in Figure 1.2. This means that for VR applications, we must render two images instead of one. The displays of VR headsets also require a higher-than-usual framerate for a comfortable viewing experience. Since we are already pushing the limits of our computers to generate a single image at the display's refresh rate, generating twice as many is difficult. Fortunately, the images of each eye are almost the same. This similarity opens up the possibility of intelligent re-use of rendering computations.

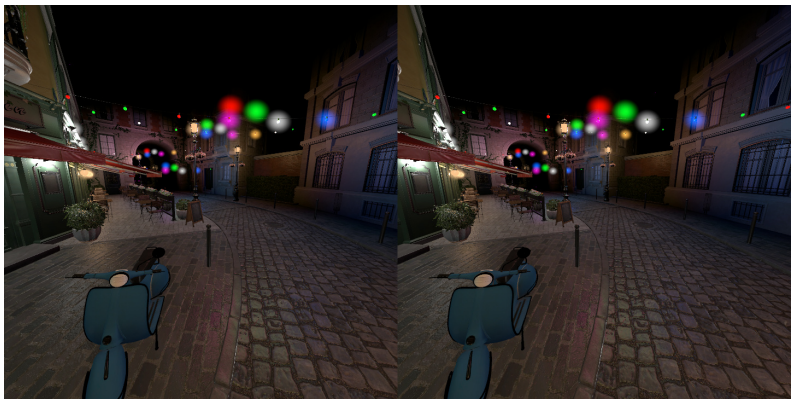


Figure 1.2: *The stereoscopic image pair of the left and right eye shown side by side. The similarity, or coherence, between the images of each eye can be exploited to accelerate rendering in various ways.*

1.4 Contributions and Structure

In this thesis I heavily build on CLS: a light culling technique presented in "Clustered Deferred and Forward Shading" [OBA12]. I propose *enclosed clustering*, which aims to reduce the cost of clustering for stereoscopic rendering. I achieve this performing light assignment (one of the steps involved in CLS) only once for both eyes, rather than for each eye separately. I expect this to save time because each eye covers mostly the same space.

Additionally, I propose two variations of the construction of the clustering grid: *orthographic clustering* and *displaced perspective clustering* which may improve rendering times for certain applications compared to the traditional *perspective clustering*. These alternative constructions can be combined with enclosed clustering.

In summary, I make the following contributions.

- I propose *enclosed clustering*, and show that it allows CLS to be used for stereoscopic rendering at close to half the cost,
- I present two variations on how the clusters are defined for CLS, *orthographic clustering* and *displaced perspective clustering*, and show that they can outperform traditional perspective clustering.

- I provide an in-depth comparison between the presented and traditional methods and insight on how to choose between them.

This thesis is structured as follows. Chapter 2 provides background information and introduces related work. Chapter 3 describes CLS in detail because it is central to the work in this thesis. Chapter 4 explains enclosed-, orthographic-, and displaced perspective clustering. Chapter 5 provides details on the implementation of these methods. Chapter 6 presents and discusses the results obtained through elaborate profiling. Chapter 7 presents the conclusions and provides some directions for future work.

Appendix A explores a number of light attenuation function and motivates the function used in the renderer. Appendix B shows the application of CLS to simulating global illumination using virtual point lights. This is more of an experimental result and therefore not covered in the main text. Appendix C contains results that did not fit in the main text. Appendix D explains a practical use of prefix-sums and how they are implemented on the GPU. Prefix-sums are used to implement CLS but also in other places in the renderer. Appendix E gives a little more detail on the uncommon parameterization of orthographic- and perspective projections used in this thesis which I think should be the standard.

2

Background and Related Work

The computation of light contributions is a computationally expensive step within the context of real-time rasterization, particularly when the number of light sources is large. In case of stereoscopic rendering, the rasterization work is increased compared to common monoscopic monitors due to a higher total pixel count and frame-rate. Fortunately, the views from the left and right eye are largely coherent allowing for the re-use of computations. This chapter briefly recapitulates virtual scene descriptions in Section 2.1 and the rasterization pipeline in Section 2.2. Forward and deferred rendering are covered in Sections 2.3 and 2.4. In Section 2.5 CLS and other real-time fully dynamic many-light rendering techniques are introduced. To cope with increased rendering work due to high pixel counts and frame rates imposed by virtual reality displays, techniques exploiting the coherency between stereoscopic images are discussed in Section 2.6. Finally, Section 2.7 relates clustering grid construction using a decoupled camera projection to other work.

2.1 Geometry, Lights and Cameras

This section briefly recapitulates scene and camera descriptions which are the inputs to the rendering process. For a complete introduction to the various parts of rendering I refer the reader to [Scr20].

A virtual scene commonly consists of a description of surfaces and of light sources. These surfaces are collectively called the *geometry* of the scene and are modeled by *geometric primitives*. The most common geometric primitive is a triangle, but lines and points are also used. Implicit surfaces can also be used when modeling a scene but, unlike with ray tracing, rasterization does not support them directly: they must be tesselated into geometric primitives. The vertices of geometric primitives have data associated with them, called *vertex attributes*, like their 3D position, the normal, binormal and tangent of the modelled surface, and texture coordinates. The vertex position is necessary to determine visibility and the other attributes are used in the computation of the surface color. The light sources in a virtual scene are commonly modeled by directional-, point-, or spotlights. A directional light eminates light in a parallel fashion while point- and spotlights eminate light from a single point. A spotlight is similar to a point light but additionally focuses light in a cone.

What is visible in each pixel of an image is determined by a camera model. The camera model defines a coordinate frame and a projection which maps 3D points onto the image plane. There are

two basic types of projection: *perspective projection* and *orthographic projection*. With perspective projection all view rays go through a single point and with orthographic projection all view rays have the exact same direction.

2.2 Rasterization Pipeline

The rasterization pipeline available on GPUs allows very fast rendering of scenes with a high geometric complexity. The pipeline consists of the following steps.

1. Transform all vertices into clip space (Vertex Shader).
2. Generate fragments (Rasterization).
3. Shade fragments (Fragment Shader).

In the first step, the vertices of the geometric primitives are transformed into camera clip space. Usually the vertex positions are defined local to the modeled object. They need to be transformed from the object's coordinate frame to the camera's coordinate frame, usually by going through a world coordinate frame which connects the two. Then the vertex positions are represented as homogeneous coordinates and projected into the camera clip space. The GPU then performs the perspective division yielding the vertex positions in normalized device coordinates. All geometric primitives falling in a by the driver pre-defined box are visible.

The second step clips all visible geometric primitives and generates a fragment for each sample position in each pixel covered by the clipped geometric primitives. A fragment therefore represents a point on a surface in the scene and corresponds to a single pixel. In case of multi-sampling there can be multiple fragments per pixel.

In the final step each fragment is processed by the fragment shader and produces a final color value. The fragment shader receives the vertex attributes (optionally) interpolated by the fragment's barycentric coordinate weights. These attributes are used to directly, or indirectly through texture maps, model properties of the surface like its albedo, roughness and metalness. Usually the fragment shader evaluates the contribution of light sources on the point on the surface that the fragment represents. The color values produced by all fragments in a pixel are stored in a framebuffer which is essentially a layered image.

On modern GPUs, the vertex- and fragment shader are programmable meaning they can be customized to perform any desired computation. The rasterization step however is still largely fixed. Given the rasterization pipeline, there are still multiple ways in which we can arrive at the final image. These ways can be loosely categorized as forward- or deferred rendering.

2.3 Forward Rendering

Forward rendering is, as the name suggests, only moving forward in terms of the rendering steps. The geometry is rendered, fragments are generated and each fragment passing a *depth test* is shaded. The depth test ensures only fragments closer to the camera than any previously generated fragments are shaded. The depth test maintains a *depth map* which contains the closest fragment depth value for every pixel. In case a fragment passes the depth test, the previous fragment is overwritten. The overwritten fragment therefore does not, and should not since it is occluded, contribute to the final image. The shading computations performed for fragments that are later overwritten are wasted. This unnecessary shading is called *overshading*. One way to prevent overshading is by rendering the scene twice: first to obtain the complete depth map and then a second time to shade only the actually visible fragments.

2.4 Deferred Rendering

Contrary to forward rendering, deferred rendering [ST90] defers shading until after visibility has been resolved. This is achieved by storing all surface properties required by the used shading model in a stack of images collectively called the Geometry Buffer (G-Buffer). Examples of surface properties are

albedo, metallness, roughness, position and normal. To compute the final image for each light a simple mesh is drawn covering its influence volume. The light volume mesh can be a low-poly 3D triangle mesh or a simple camera-aligned quad, as long as it generates a fragment everywhere the light is considered to have substantial influence. The generated fragments then read the G-Buffer, compute the contribution of the light on the surface and write the contribution to the output pixel. Since all surface information is available for each sample, only a single geometry pass is required to compute the final image. The benefits of deferred rendering are:

- the elimination of overshading,
- accurate light culling, and
- the evaluation of a single light at a time which allows memory re-use.

The downsides are:

- the large memory requirements of the G-Buffer,
- for each generated sample in a pixel the surface properties must be read and the result must be blended into the framebuffer resulting in high bandwidth requirements,
- only a single surface can be visible in each pixel which precludes the use of transparent materials.

2.5 Many-Light Rendering

To render realistic images we often need a large number of light sources. The evaluation of all light sources for all fragments is costly. There has been a lot of research on rendering many-light scenes in real-time and offline settings. I would like to refer the reader to [Dac+14] for an overview of these method up to 2014. The work in this thesis is restricted to fully dynamic real-time methods and this section gives a brief history of these methods.

2.5.1 Light Culling

Tile-Based Shading (TBS) [OA11] is a light culling technique which can be combined with either forward or deferred rendering. The method was originally meant to better utilize the Synergystic Processing Units in the PlayStation 3 [Swo09], but proved to be useful in a broader context. The same idea is also published in a short paper under the name *Forward+* [HMY12] but I refer to this technique as TBS as that paper is more complete. Every frame the camera frustum is divided into a grid of screen space aligned tiles. Naively, each tile would cover the entire camera depth range. The extend of each tile can be limited by computing the min and max depth of all fragments in the tile. These values can be found efficiently through a min and max reduction of the depth map. For each tile in the grid, the list of relevant lights is computed. A light is relevant to a tile when its influence volume intersects the tile. For each pixel, the tile that the pixel lies in is determined and the associated lights are used to compute its final color. When applied to deferred shading, the G-Buffer textures need to be read only once and the output is written to only once, solving the bandwidth problem. TBS is not robust with respect to large depth discontinuities. When fragments close to the camera and far away from the camera fall into the same tile, the resulting volume of the tile becomes very large. The tile is then likely intersect a larger number of lights, even though many of those lights may not actually affect any of the fragments in the tile. The inclusion of unnecessary lights negatively impacts the shading time.

A natural extension of TBS to overcome this problem is to additionally divide each tile along the depth dimension, which is exactly what CLS [OBA12] does. The fragments are *clustered* in the most basic sense by partitioning space into a grid of *clusters*. To deal with the potentially large number of clusters for which a light list needs to be computed, they first determine for each cluster if its light list will actually be used. For each of these *visible* clusters a light list is computed which requires intersection cluster volumes with light volumes. A Bounding Volume Hierarchy (BVH) is built over the light volumes and used to accelerate the light assignment. CLS can be used with both forward rendering, allowing the rendering of transparent geometry, and with deferred rendering and its benefits. One potential advantage of deferred rendering is that the lights are evaluated one by one, allowing

for example the re-use of the memory for a shadow map texture. This potential advantage is lost when using CLS, because the shading of a sample iterates over all relevant lights in a single shader invocation and the data associated each light must therefore be available simultaneously. GPUs process pixels in small groups. We speak of *execution divergence* when the pixels in these groups must execute different code paths due to data-dependent conditional branches. Since CLS can lead to different lights being processed in a single wavefront, execution divergence may occur. To reduce execution divergence, binning and sorting strategies can be applied to TBS [SG16; Dro17] which may prove better than using CLS.

[Ört15] performs the light assignment for all clusters using conservative rasterization. The influence volume of each light is rasterized, storing the min and max depth per tile. For each light, for each tile, for each cluster in the tile covered by the light, the light index is appended to a linked list of lights stored per cluster. To take advantage of the flexibility in representing light influence volumes as meshes, Drobot suggests generating the light meshes from their shadow map so as not to assign lights to occluded clusters [Dro17].

2.5.2 Global Illumination

Real-time global illumination remains a challenge because we need to compute not only the direct illumination from light sources, but also the indirect illumination caused by light bouncing off all surfaces. Instead of simulating light bounces directly, many-light methods approximate the indirect illumination by placing many virtual light sources and computing their direct illumination instead [Kel97]. Later techniques focus on keeping the virtual light sources temporally stable to avoid artifacts in animations and so that visibility information can be reused [DS06; Lai+07; HKL16]. Unfortunately, to get good results many virtual light sources that individually have a small effect are needed. Together they should contribute to far away surfaces but since CLS assigns a small influence volume to lights with a small intensity, the far away contributions are lost. Using a larger radius is inefficient since that drastically increases the number of lights per sample. Instead, clusters of virtual light sources can be replaced by representative lights when shading far away surfaces [Lau+16]. In Appendix B I show my results of using CLS to render global illumination effects through virtual point lights.

2.6 Utilizing Coherency in Stereoscopic Rendering

A recent method allows the rendering of a scene from many viewpoints in real-time [Kol+19]. Sublinear scaling in terms of view and scene complexity is achieved through shared rendering of samples generated by iterating over pairs of nodes from two trees (one tree over the views and one tree over the geometry), pruning branches where no more detail is required. The idea of shared rendering is far from new and has been applied to stereoscopic rendering [Neh+07]. The shaded samples from one eye are reprojected to the other eye. Missing information and view dependent effect like specular reflections introduce error and need to be dealt with. In this thesis I exploit the coherence between views to perform the light assignment step of CLS only once rather than per eye.

2.7 Alternative Projections

Performing the clustering once requires the decoupling the clustering camera frustum from the rendering camera frustum. This decoupling opens up possibilities in the construction of the clustering grid. I present a technique called *displaced perspective clustering* which gives finer control over the shape and distribution of clusters. A similar idea is used in [WSP04] where the geometry gets transformed by an additional perspective projection to achieve a more even distribution of shadow map resolution in screen space.

3

Clustered Light Shading

This chapter explains CLS as described in "Clustered Deferred and Forward Shading" by Olsson, Billeter, and Assarsson. CLS can be divided into four steps: 1) determine the clustering space, 2) compute cluster visibility, 3) assign lights to visible clusters and 4) shade fragments using these light lists. Sections 3.1 - 3.4 go over these steps in order. The first three steps together will be referred to as *performing clustering* and the resulting data as *a clustering*.

3.1 Cluster Space Construction

As was mentioned in the introduction, many real-time applications limit the influence of light sources to a relatively small region where the light is considered to contribute significantly. Outside of this *influence volume*, the light is ignored which saves lighting computations at the loss of (specular) reflections of far-away lights and combined contributions of multiple far-away lights. To make use of these bounded light sources, CLS groups shading samples into *clusters* and computes the relevant lights per cluster. The relevant lights are computed for groups of samples rather than per sample for various reasons. One of the reasons is that we would have to store a list of lights for each sample which even for a small number of lights would require a large amount of memory. Another reason is that the time gained over naively evaluating the contribution of all lights must outweigh the time spent pre-computing which would be difficult to achieve when computing a list per sample. This section describes how the samples are clustered in [OBA12].

Clustering Scheme

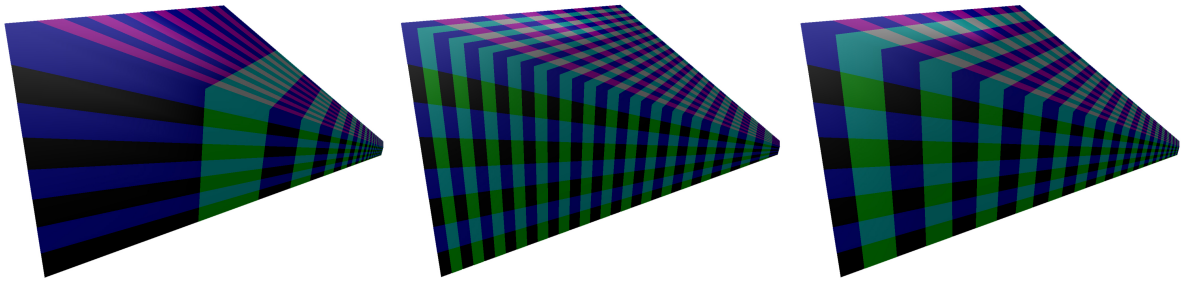
The clustering scheme must be very efficient because we have a large number of samples and the clustering is performed every frame. CLS employs a simple space partitioning scheme as this is both fast and it provides predictable cluster sizes. This partitioning can be performed by 3D position but also by normal direction. I only discuss and explore position-based partitioning to limit the scope of this thesis.

A common way to partition space is to use a simple uniform world-space grid [GL10]. This partitioning is dismissed in [OBA12] for the following reasons:

1. selecting the size of the grid cells requires manual tweaking,
2. depending on the size of the grid, the cluster index may require a very large number of bits, and

3. clusters far away from camera become very small under perspective projection which leads to excessive light list computations.

Instead, the rendering camera frustum is divided, or *quantized*, into a grid of sub-frusta. The grid divides the frustum in post-projective space evenly along the X- and Y-axis, and exponentially along the Z-axis as shown in Figure 3.1c. The exponential subdivision stems from the desire to make the clusters as cubical as possible [Llo+06].



(a) *Uniform Z Post-Projective* **(b)** *Uniform Z Pre-Projective* **(c)** *Exponential Z Pre-Projective*

Figure 3.1: *Distributing clusters linearly in post-projective space (a) or pre-projective space(b) leads to a undesirable distribution of clusters. Instead we make clusters as cubical as possible by distributing their depths exponentionally in pre-projective space (c). Images from [OBA12].*

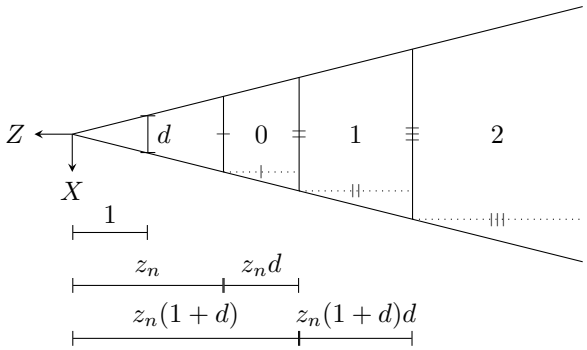


Figure 3.2: *Three clusters with Z-indices 0, 1 and 2 are shown. The width and depth of each cluster are kept equal leading to a geometric sequence and thus an exponential distribution of their near plane Z-coordinates. The parameter d is the length of the line intersecting the frustum at $Z = -1$.*

We desire the clusters to be as cubical as possible because that will give us a similar cluster resolution in all directions in world space, which is likely to effectively cull the light sources. We cannot make each cluster perfectly cubical, but we can get close by forcing each cluster to be as long as it is wide and high. Given the length d, the clustering camera near plane z_n and the previous constraints as visualized in Figure 3.2, we can derive a sequence z_i giving the Z-coordinate of the near plane of a cluster with Z-index i:

$$z_0 = -z_n \tag{3.1}$$

$$\begin{aligned} z_i &= z_{i-1} + z_{i-1}d \\ &= z_{i-1}(1+d) \end{aligned} \tag{3.2}$$

$$z_i = -z_n(1+d)^i \tag{3.3}$$

Given the angles α_l and α_r made by the camera frustum's left and right side planes with the $X = 0$ plane and the dimensions of the cluster grid D where D_x is the number of clusters along the X-axis of

the grid, we can compute d:

$$\tan(\alpha_l) = \frac{x_0}{-1} \qquad \tan(\alpha_r) = \frac{x_1}{-1} \qquad \Delta x = x_1 - x_0$$

$$d = \frac{\Delta x}{D_x}$$

The equations above only consider the XZ plane, but we are working in three dimensions. When the camera frustum and screen have the same aspect ratio, and the desired pixels per cluster is square ($P_x = P_y$), the value found for d in the XZ plane will be the same as the value for d in the YZ plane. In practice, we can settle on a value for d by choosing either or taking the average:

$$d = \frac{1}{2}\left(\frac{\Delta x}{D_x} + \frac{\Delta y}{D_y}\right). \tag{3.4}$$

Cluster Grid Dimensions

Since we are using a grid, we need to somehow determine its dimensions D. We determine the dimensions D_x and D_y from the screen dimensions S and a desired number of pixels per cluster P:

$$D_a = \left\lceil \frac{S_a}{P_a} \right\rceil \text{ for } a \in \{x, y\} \tag{3.5}$$

Using Equation 3.3, we can determine the dimensions D_z from d and the camera's depth range (z_n, z_f):

$$D_z = \left\lceil \ln_{1+d} \frac{z_f}{z_n} \right\rceil \tag{3.6}$$

Computing a Cluster Index

To compute a cluster index for a point p_{cam} in camera (view) space, we compute the position of this point in *cluster space*. Cluster space is the space in which the cluster grid is defined. The XY-coordinates of the cluster space position p_{clu} are computed using the clustering camera's perspective projection mapped to the range $(0, D)$ rather than the usual clip-space range. The Z-coordinate of p_{clu} is computed using Equation 3.3 rewritten as follows:

$$p_{clu,z} = \ln\left(\frac{p_{cam,z}}{-z_n}\right) \tag{3.7}$$

The cluster index is then:

$$(i_x, i_y, i_z) = \lfloor p_{clu} \rfloor \tag{3.8}$$

3.2 Cluster Visibility

The idea of CLS is to compute a list of relevant lights per cluster to be used during shading. While we could compute a light list for all clusters, doing so would be wasteful. Only the light lists of the clusters that have *visible fragments* in them are actually used, where a visible fragment is a fragment that contributes to the final color of its pixel. I refer to clusters containing visible fragments as *visible clusters*.

To mark clusters as visible, we have to determine the cluster indices of the visible fragments. We need 1 bit to represent whether a cluster is visible or not. In [OBA12], two strategies are presented. The first strategy computes the cluster index for each pixel, then computes the unique cluster indices for each screen space tile of clusters and finally computes a global list of all unique cluster indices. The second strategy virtually allocates a bit for each cluster in the cluster grid using page tables as the virtual allocation method. In this thesis I omit the page table scheme and simply allocate enough

memory for the entire grid. A page table scheme would allow the use of a higher grid resolution, but the need for such a fine grid did not arise.

For opaque geometry, where only one fragment is visible per pixel, this can be done by performing a depth pre-pass. The fragment positions and therefore cluster indices can be reconstructed from the depth value at each pixel. The depth buffer can be re-used during subsequent rendering.

At this point, we know for each cluster if it is visible or not. To facilitate performing computations for the visible clusters only, we compute a contiguous list of indices of the visible clusters. This *compaction* of visible cluster indices is computed using the parallel prefix sum described in Appendix D. The light list meta-data will be computed and stored for visible clusters only, so we also need to store the reverse mapping; for each cluster we store its visible cluster index incremented by one if it is visible and zero otherwise.

3.3 Light Assignment

At this point we have a list of visible clusters. Each visible cluster knows its cluster index which, given the cluster frustum and dimensions, tells us the volume that it describes in cluster camera space. The number of lights ending up in each cluster can vary, so we count how many lights each visible cluster intersects. We then compute for each cluster, an offset into a global list of light indices. These offsets are found by computing the prefix sum over the light counts. A second pass intersecting each visible cluster with the lights then writes the light indices into the global list using the computed offsets.

Cluster-Light Intersections

To determine if a cluster is affected by a light, we must compute if the cluster volume and the light's influence volume intersect. We can perform the intersection test before or after perspective projection [MM12]. Before perspective projection volume of a cluster is described by a frustum and, since I have limited myself to point lights, the influence volume of a light is described by a sphere. Instead of a frustum, an axis-aligned bounding box or sphere can be used to describe the volume of a cluster [Wro17]. This may be helpful for certain kinds of light like spotlights which are described by cones. By performing perspective projection frusta become axis-aligned boxes and spheres become ellipsoids. I only consider computing frustum-sphere intersections in pre-projective space because they are easier to reason about and exploring post-projective is outside of the scope of this thesis.

Accurate frustum-sphere intersection tests require analyzing many different cases. In practice an enlarged frustum-point test is used instead. The frustum is enlarged by moving the frustum planes along their normals scaled by the sphere's radius. We then test if the sphere's center lies within this enlarged frustum. Because the frustum has the same origin and orientation as the clustering camera, we can simplify the enlarged frustum-point.

The enlarged frustum-point is conservative and therefore has a false positive volume. The false positive volume equal to the difference between enlarged-frustum and the Minkowski sum of the frustum and sphere. Usually, for example when doing mesh culling, the frustum is a lot larger than the sphere under test. In this case the false-positive volume is relatively small. However, for us the frusta are small compared to the light influence spheres [Per13]. This leads to a large false-positive volume and thus to lights being included in light lists where they should not.

The intersection tests are performed in the light counting and light assignment passes of CLS. The performance of these passes are partially determined by the used intersection test. For conservative intersection tests the light count may be higher than strictly necessary. This has direct influence on the performance of the shading pass. Selecting an appropriate intersection test is orthogonal to the work in this thesis and therefore I only consider the point in enlarged-frustum test. Note however that a more efficient or accurate intersection test decreases the light assignment and shading time respectively, and this should be kept in mind when viewing the results presented in this thesis. I expect however that the chosen simple intersection test is common in practice and represents a reasonable amount of work and culls efficiently enough.

3.4 Shading

With the pre-computation out of the way, CLS is easy to integrate. A fragment shader needs to take the following additional steps:

1. Compute the fragment position in cluster space.
2. Discretize the fragment position to obtain its 3D index in the cluster grid.
3. Compute the 1D cluster index from the 3D index using the cluster grid dimensions.
4. Look up the potential visible cluster index associated with the cluster index.
5. Assuming the visible cluster index exists, look up the light list offset and count.

After obtaining the list of relevant lights, the shader computes the sample's color by evaluating the contribution of the relevant lights only, rather than all lights in the scene.

4

Methods

The real-time rendering of realistic images with many light sources requires efficient light culling strategies such as CLS which was explained in the previous chapter. When rendering stereoscopic images for display in VR headsets, the amount of rendering work is increased substantially compared to an average monoscopic desktop display. While CLS can be applied to both the left and the right eye individually, this thesis presents *enclosed clustering* in Section 4.1: a method which seeks to reduce the computational cost of CLS for stereoscopic rendering. Specifically, I make use of the fact that the space covered by the left and right eye largely overlap and attempt to perform the light assignment step only once rather than twice.

To realize the single light assignment, the clustering camera has to be decoupled from the rendering camera. This decoupling opens a window to various ways of constructing the clustering camera in a stereoscopic, but also in a monoscopic setting. In Section 4.2 I explore *orthographic clustering* where the clustering camera uses a orthographic projection. This leads to the originally dismissed [OBA12] idea of using a uniform grid of clusters in a pre-projective space, for example world- or camera space. The main reason for the dismissal of this idea is that far-away clusters become very small when viewed under perspective projection which in the worst case means we compute a light list per pixel. While this is a valid concern, the traditional *perspective clustering* leads to very large clusters which likely contain many lights leading to long shading times. Rather than making a prediction, I explore orthographic clustering regardless of the mentioned concerns and evaluate how both strategies compare in practice.

A second alternative clustering strategy for mono- and stereoscopic rendering uses the traditional perspective clustering, but moves the origin of the clustering camera backwards. This *displaced perspective clustering*, discussed in Section 4.3, provides a mix between orthographic and perspective clustering based on the displaced distance. In practice, the first layers of clusters closest to the camera may be smaller than necessary. As a solution, distribution of of clusters along the Z-axis is modeled as a piecewise function [Per13] ending with the exponential distribution from [OBA12] where the first few depths are specified by hand. Displaced perspective clustering solves the same problem, albeit at the cost of decoupling the clustering camera from the rendering camera.

The idea of performing the light assignment once for multiple cameras can be extended to generic multi-view rendering. Multi-view rendering simply means a scene is rendered multiple times from different perspectives. Examples of multi-view rendering occur in the rendering of reflections, physically correct depth of field and motion blur [Rag+11; LD12; Vai+12], and global illumination [Kol+19]. While we can combine the light assignment for any number of cameras, any benefits must come from the fact that at least some of the cameras cover the same space. The application of CLS to multi-view rendering is discussed in Section 4.4.

4.1 Enclosed Clustering

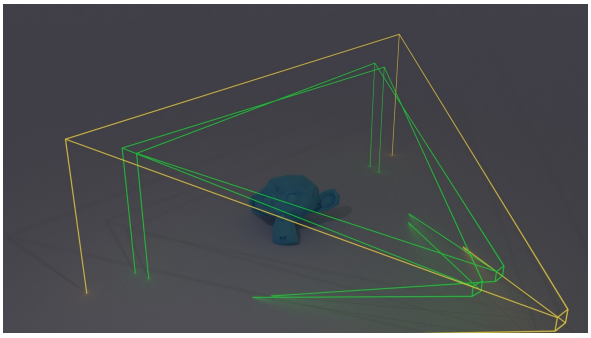


Figure 4.1: *Illustration of a possible clustering camera frustum (yellow) enclosing the two rendering camera frusta (green) of the left and right eye. In practice, we would like the enclosing camera frustum to be as tight as possible to make the best use of the clustering grid resolution.*

As mentioned in the introduction of this chapter, enclosed clustering seeks to reduce the cost of the light assignment step of CLS in a stereoscopic rendering setting. When CLS is applied to each eye individually, most of the light assignment work is done twice unnecessarily as the clusters from both cameras occupy mostly the same space. What enclosed clustering proposes is that we construct a clustering camera from the rendering cameras in such a way that the rendering camera frusta are completely enclosed by the clustering camera frustum as shown in Figure 4.1. Samples from the left and right eye can be transformed to the clustering camera space to determine cluster visibility and later to retrieve the relevant light list. The differences in the rendering pipeline between performing CLS per-camera and the proposed enclosed clustering are shown in Figure 4.2.

1. Construct clustering camera (both)
2. Mark visible clusters (both)
3. Assign lights to visible clusters (both)
4. Shade samples (both)

(a) *Per-camera clustering*

1. Construct enclosing clustering camera (once)
2. Mark visible clusters* (both)
3. Assign lights to visible clusters (once)
4. Shade samples* (both)

(b) *Enclosed clustering*

Figure 4.2: *The difference between enclosed clustering and per-camera clustering. We save computation time by performing the light assignment once instead of twice. The computational cost of construction of the clustering camera is insignificant. The steps marked by the * symbol are slightly modified to now reproject samples into the enclosed clustering camera space instead of the rendering camera space. Note that with per-camera clustering, the rendering process is completely independent allowing memory re-use. For enclosed clustering however, some information, like the depth buffers used to determine cluster visibility, may need to be kept for usage in further rendering and therefore must be stored simultaneously increasing the memory required by the rendering pipeline.*

Section 4.1.1 explains how I construct the enclosing camera. Because we change the clustering space, the computation of the cluster index changes. The revised cluster index computation is detailed in Section 4.1.2.

4.1.1 Enclosed clustering camera construction

The enclosing clustering camera should tightly enclose both rendering camera frusta. Based on common virtual reality headsets, we can make assumptions on how the left and right eye cameras are set up. This narrows the amount of variation we have to consider when constructing an enclosing frustum. In the presented construction, I assume that the collection of far-plane corners of the to-be-enclosed

camera frusta and the collection of near-plane corners can be separated by a plane orthogonal to the Z-axis.

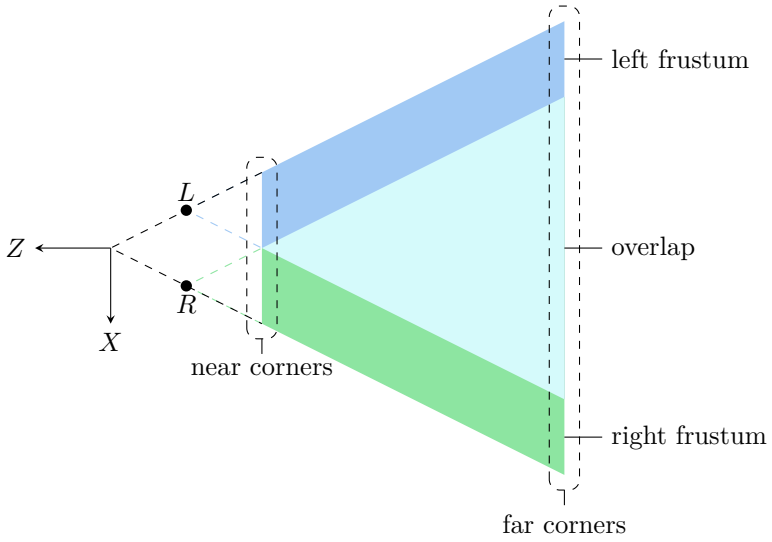


Figure 4.3: *A left and right camera frusta with origins L and R drawn in cluster camera space. The near corners set contains all near-plane corners from all cameras, and similar for the far corners set. In this illustration, no rotations have been applied to the left and right eye. The left and right frusta are also symmetric. In reality slight rotations and assymetric frusta may occur. The proposed method is robust under these variations.*

A simple camera configuration is given in Figure 4.3. To construct the enclosing camera I first transform the near and far corners of both frusta into *head space*. The head space is defined by the average origin and orientation of the left and right eye. Then I iterate over all possible combinations near and far corners. I project the line between each combination of corners onto the XZ and YZ planes. For each line where all other corners are on only one side of the line, I calculate its intersection with the Z-axis. The intersection with the largest z-component becomes the new origin. Using the new origin, I compute the minimum and maximum tangents that the far plane corners make with the new origin in the XY and YZ planes. The minimum and maximum tangent in the XZ and YZ planes define the left, right, bottom and top side planes of the new frustum respectively. The displacement of the origin along the Z-axis is subtracted from the old near and far plane depths to obtain the new near and far planes.

For perspective clustering, the cluster grid dimensions D are parameterized by a desired width and height in pixels P in screen space, the screen dimensions S and the camera depth-range (n, f). However, the enclosing camera does not have a screen associated with it, so we cannot compute the cluster dimensions from the desired number of pixels per cluster. There are multiple options for how to determine the cluster grid dimensions D for the enclosing clustering. We need to somehow combine the dimensions that the left and right eye would have, for example by taking the average. Taking the average however leads to stretched clusters. Instead, I compute the new cluster grid dimensions from a desired average cluster volume V_c. The desired V_c is calculated as the average of: the average cluster volumes of both eyes that would be used in case of per-camera clustering.

Definition of the average cluster volume

We can compute the average volume of a single cluster cluster V_c by dividing the volume of the frustum V_F by the total number of clusters $N = D_x \cdot D_y \cdot D_z$. In order to derive the formula for the volume of a perspective projection frustum, we first define the frustum paramters.

Definition 4.1. The frustum of a perspective projection camera can be described by the matrix F:

$$F = \begin{bmatrix} \frac{l}{n} & \frac{r}{n} \\ \frac{b}{n} & \frac{t}{n} \\ -f & -n \end{bmatrix}$$

The parameters l, r, b, t, n and f are the conventional left, right, bottom, top, near and far values. We divide l, r, b and t by n so that they are independent from n which simplifies many equations. These values are equal to the tangent of the angle between the respective side plane with the Z-axis divided by -1. Each row describes the parameters for a different axis. The entries of the first row referred to as $F_{x,0}$ and $F_{x,1}$ and similar for the other rows. The shorthand $F_{\Delta x}$ is equal to $F_{x,1} - F_{x,0}$. When clear from the context the entries may simply be referred to as x_0, x_1, Δx, et cetera to reduce notational clutter.

The formula for V_F is found by subtracting the volume of the pyramid described by the origin and the near plane from the pyramid described by the origin and the far plane. The length along the X-axis at depth z is given by $-z\Delta x$ and similar for Y. The formula for the pyramid at depth z is therefore:

$$V(z) = \frac{1}{3}(-z\Delta x)(-z\Delta y)(-z) = -\frac{1}{3}z^3\Delta x\Delta y \tag{4.1}$$

We can now compute the volume of the frustum with:

$$\begin{aligned} V_F &= V(z_0) - V(z_1) \\ V_F &= \frac{1}{3}\Delta x\Delta y(z_1^3 - z_0^3) = NV_c \end{aligned} \tag{4.2}$$

Computing dimensions from a desired average cluster volume

The cluster grid dimensions D_x and D_y can be computed from a desired cluster width and height d at $Z = -1$ and Δx and Δy from the clustering camera frustum F. The depth dimension D_z can be found using Equation 3.6 which requires d, z_0 and z_1 from the clustering camera frustum F. To keep the clusters as cubical as possible, we use the same value d for the width, height and depth giving the following constraints:

$$D_x = \frac{\Delta x}{d} \qquad D_y = \frac{\Delta y}{d} \qquad D_z = \ln_{1+d}\left(\frac{z_0}{z_1}\right) \tag{4.3}$$

We can find a value for d by combining Equations 4.2 and 4.3 from the clustering camera frustum F and the desired average cluster volume V_c:

$$\begin{aligned} V_F &= NV_c \\ &= D_xD_yD_zV_c \\ \frac{1}{3}\Delta x\Delta y(z_1^3 - z_0^3) &= \frac{\Delta x}{d}\frac{\Delta y}{d}\ln_{1+d}(z_0/z_1)\,V_c \\ d^2\ln(1+d) &= \frac{3\ln(z_0/z_1)\,V_c}{z_1^3 - z_0^3} \end{aligned} \tag{4.4}$$

I estimate d in Equation 4.4 numerically using a small number of Newton-Raphson iterations. To compute the initial guess for d, I substitute $ln\,(1+d)$ with d in Equation 4.4 and solve. The computation should converge quickly since the function is monotonically increasing and the initial guess positive.

The cluster dimensions are then found by computing and ceiling D_x, D_y and D_z using equation 4.3. After ceiling, we can optionally recompute d as d' to fit the clusters exactly into the frustum along the depth dimension using Equation 4.5:

$$\begin{aligned} z_0 &= z_1(1+d')^{\lceil D_z \rceil} \\ d' &= \sqrt[\lceil D_z \rceil]{\frac{z_0}{z_1}} - 1 \end{aligned} \tag{4.5}$$

4.1.2 Revised cluster index computation

Decoupling the clustering camera from the rendering camera changes the computation of the cluster index from a sample position. In traditional perspective clustering, the clusters nicely tile the screen in the XY-plane. The X- and Y cluster index can therefore be given by the screen space position of a sample. Because we necessarily moved the origin of the clustering camera with respect to the rendering camera (Figure 4.3), such a tiling becomes impossible as shown in Figure 4.4.

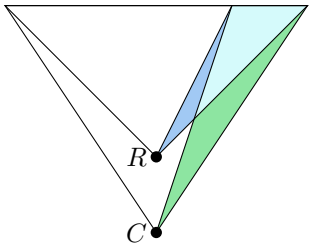


Figure 4.4: *By moving the origin of the clustering camera from R to C, the screen space tiles of C (green) no longer tile the screen space of the rendering camera R (blue).*

To find the cluster index we must compute the sample position in clustering camera space which can be done in the vertex shader or in the fragment shader. After doing so, we find the cluster index by transforming it into the clustering camera space and projecting by the clustering camera projection. The Z-coordinate is found from the clustering camera space through the exponential distribution described by Equation 3.3. The computation of the Z-coordinate is non-linear and therefore must be computed per sample; it can not be linearly interpolated from values computed per vertex.

4.2 Orthographic Clustering

The arguably simplest partitioning of 3D space is to divide it into a grid of cubes. This is exactly what *orthographic clustering* does. This idea is not new, in fact it was intentionally avoided for a couple of reasons. The most important reason is that orthographic clusters can become very small when viewed under perspective projection. When we have many small clusters we must compute many light lists. The computation of the light list may not outweigh the shadings in saving time if the list is only used by a very small number of pixels. Additionally, it is likely detrimental to performance when neighbouring pixels access different light lists due to potential cache misses and execution divergence. On the flip side, the large far-away clusters produced by traditional perspective clustering may not cull lights effectively. Far-away samples may need to consider many more lights than strictly necessary.

To construct a grid of cubes we first need to choose a coordinate frame. Note that because the grid is uniform in pre-projective space, the orientation of the grid does not matter much. Regardless, I have chosen to do construct the grid in camera space as shown in Figure 4.5. By constructing the grid in camera space, we can make the grid fit the camera frustum so that "only" $\frac{2}{3}$ of the clustering space is wasted. Instead of using the rendering camera space, we could consider using world space. Using world space would eliminate the need to compute light positions in cluster camera space.

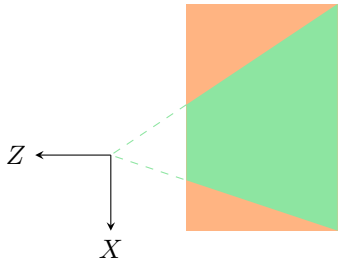


Figure 4.5: *The clustering grid is established using the bounding box (orange) of the rendering camera (green).*

To find the dimensions D of the cluster grid I first determine the side lengths s_x, s_y and s_z of a single cluster. Instead of specifying the side lengths individually, I assume we want the cluster to be a cube and compute a common side length $s = s_x = s_y = s_z$ from a desired cluster volume V_c by solving:

$$s_x s_y s_z = s^3 = V_c \tag{4.6}$$

$$s = \sqrt[3]{V_c} \tag{4.7}$$

Then, I compute bounding box B of the rendering camera frustum in clustering camera space. The dimensions D are then found by dividing the lengths of the sides of the bounding box $B_{\Delta x}$, $B_{\Delta y}$ and $B_{\Delta z}$ by the lenghts of the sides of a cluster and ceiling the result:

$$D_x = \left\lceil \frac{B_{\Delta x}}{s_x} \right\rceil \qquad D_y = \left\lceil \frac{B_{\Delta y}}{s_y} \right\rceil \qquad D_z = \left\lceil \frac{B_{\Delta z}}{s_z} \right\rceil \tag{4.8}$$

To make the clusters divide up the bounding box perfectly, the actual lengths of the sides of a cluster s'_x, s'_y and s'_z can be recomputed from the found dimensions and the bounding box B. When doing so the clusters likely end up not being perfect cubes.

This construction works for any number of cameras as long as we can settle on a common coordinate frame like world space. Orthographic clustering is therefore easily applied to enclosed clustering and even to multi-view rendering. In multi-view rendering there may not be a common orientation between the cameras which makes the construction of an enclosing perspective projection-based frustum difficult.

To find the cluster index of a sample we simply linearly interpolate the sample's position in the bounding box B to the range $(0, D_a)$ along each axis a. This is of course an orthographic projection which can be combined with any other transformations and executed in the vertex shader.

Each cluster in the clustering grid is an axis-aligned box in the cluster camera space. Because the clusters are axis-aligned boxes instead of frusta, we can use the more efficient axis-aligned box versus sphere intersection test. This test does not generate false positives like the point in enlarged frustum test.

4.3 Displaced Perspective Clustering

The goal of traditional perspective clustering is to keep the amount of clustering work per pixel consistent across the camera depth. As a result, the number of clusters close to the camera may be larger than necessary. Persson suggests using a manually tweaked distribution of clusters close to the camera, rather than the geometric distribution. Instead of a piecewise function, I suggest displacing the clustering camera origin as shown in Figure 4.6. This modification comes for free in case of enclosed clustering since the clustering camera is already decoupled from the rendering camera.

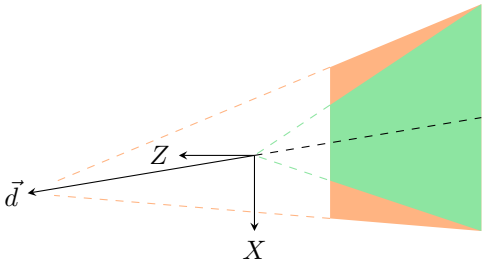


Figure 4.6: *We displace the origin along the central Z axis of the frustum. By increasing the displacement the clustering becomes more like ortographic clustering.*

In order to construct the displaced camera frustum we simply translate the origin. Then we compute new tangents in such a way that the displaced camera's far plane matches up with the rendering camera's far plane. As we increase the displacement, the clusters become more cube-like. However, if the frustum is assymetric the clusters become skewed cubes. To solve this, we could re-orient the frustum so that it is symmetric. Re-orienting the clustering camera frustum is not explored further in this thesis.

Displaced perspective clustering decouples the clustering camera from the rendering camera. Therefore I use the same method as described in Section 4.1.2 to compute the cluster index from the sample position.

The ideal cluster-light intersection test depends on the final shape of the clusters. For a large displacement and a symmetric frustum, the clusters become cubes. When the clusters are similar to cubes, an enlarged axis-aligned box versus sphere test can be used. Otherwise, the usual enlarged frustum versus point test from perspective clustering can be used.

4.4 Multi-View Rendering

The idea of re-using the clustering for stereoscopic rendering can be extended to any number of views. In fact, stereoscopic rendering is just one of many applications of multi-view rendering. Other applications include rendering soft-shadows, realistic motion and defocus blur, reflections and indirect illumination [Kol+19]. The implementation for enclosed clustering can, apart from the construction of the enclosing clustering camera which is a small part and probably happens on the CPU, be used as-is for multi-view rendering. Doing so is only helpful when the rendering frusta overlap, otherwise we might as well perform the clustering per camera as there are no duplicated light assignment calculations.

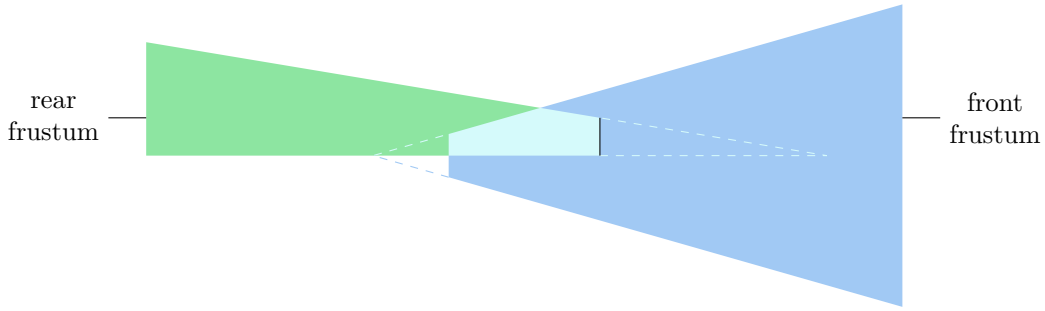
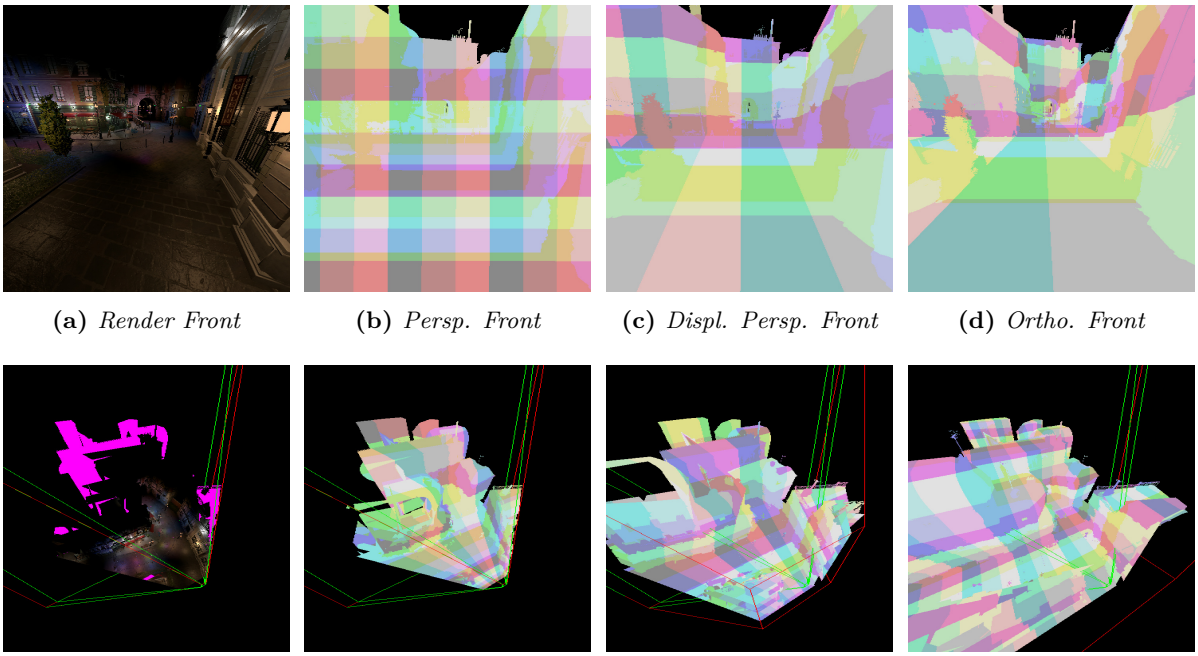


Figure 4.7: *Enclosed clustering for a front and rear view calls for orthographic clustering. The shape of the enclosing frustum becomes like a box and we would want to divide the depth linearly in camera space.*

For some of the applications, like reflections and indirect illumination, there is little coherency between the orientation of the views. Figure 4.7 illustrates such a scenario. The mirror camera faces in the opposite direction of the main camera. It is not clear how to choose an appropriate orientation for the construction of an enclosing perspective clustering camera. Unlike perspective clustering, orthographic clustering does not require a common direction and is therefore more easily applied to CLS for multi-view rendering.

4.5 Summary

In this chapter I presented three methods: enclosed clustering, orthographic clustering and displaced perspective clustering. Enclosed clustering performs the light assignment step of CLS once instead of twice when rendering a stereoscopic image by choosing a common enclosing clustering camera in which the clustering grid is constructed. Orthographic clustering and displaced perspective clustering are two alternative clustering grid construction methods which can be applied to both monoscopic and stereoscopic rendering. The different clustering camera construction techniques applied to monoscopic rendering are shown in Figure 4.8.



(a) *Render Front* **(b)** *Persp. Front* **(c)** *Displ. Persp. Front* **(d)** *Ortho. Front*

(e) *Render Bird's* **(f)** *Persp. Bird's* **(g)** *Displ. Persp. Bird's* **(h)** *Ortho. Bird's*

Figure 4.8: *The Bistro Exterior scene rendered with different clustering strategies. The front view determines the clustering camera. The bird's eye view simply takes a different perspective and does not affect the clustering camera. The clustering strategies are visualized by using the 3D cluster index modulo 3 mapped to each color channel for each pixel. The magenta color in the renders indicate that no cluster is available for that pixel which can happen in the bird's eye view. Fragments outside of the clustering space were discarded. The rendering camera frusta are drawn in green. Clustering camera frusta are drawn in red. The clusters produced by displaced perspective- and orthographic clustering do not nicely tile the screen like traditional perspective clustering does because the clustering and rendering are decoupled and do not share their origin. Notice how displaced perspective clustering is a mix between perspective- and orthographic clustering.*

5

Implementation

I implemented the CLS variations described in Section 4 in a renderer built specifically to support the work presented in this thesis. The renderer is implemented in Rust and makes use of OpenGL and some basic utility libraries like `cgmath` for 3D math, `image` for image processing and `rand` for random number generation. I built the render from scratch, which I knew would be a time consuming endeavour, to gain a good understanding of the entire rendering pipeline and to appreciate the work put into modern game engines. The complete source code is publicly available at github.com/mickvangelderen/clustered-light-shading. An overview of the implemented variations is given in Table 5.1.

Name	**Description**
`Indi`	*Individual clustering:* The whole clustering is performed for each rendering camera individually.
`Encl`	*Enclosed clustering:* Multiple rendering cameras re-use a single clustering. The clustering is computed using a clustering camera that encloses all the rendering cameras.
`Persp` P	*Perspective clustering:* Traditional CLS where the clustering camera and the rendering camera are the same. The desired width and height of clusters in screen space is given by P.
`Ortho` S	*Orthographic clustering:* Clusters are given by a uniform grid in camera (view) space. Each cluster is a cube with side length S.
`Displ` P D	*Displaced perspective clustering:* The clustering camera decoupled from the rendering camera by displacing the origin backwards by D along the frustum's central forward axis. The clustering grid is constructed with using the traditional perspective based method. The cluster grid dimensions are determined from a desired average cluster volume V_c. The desired cluster volume V_c is found by computing the volume that would occur in case of Persp P.

Table 5.1: *An overview of the CLS variations considered in this thesis. The name is the short-hand used to reference each variant. The parameters P, S and D are omitted when irrelevant.* `Indi` *or* `Encl` *can be combined with any of* `Persp`, `Ortho` *or* `Displ`.

In [OBA12], various alterations of CLS are compared with traditional deferred shading, and deferred- and forward tiled-based shading. The overall best performing CLS variation is *ClusteredDeferredPt* and the runner-up is *ClusteredForward*. The base CLS implementation I used is most similar to Clustered-

Forward. Forward shading gives use maximum flexibility because it allows shader changes in between draw calls, transparency and Multi-Sampling Anti-Aliasing (MSAA). Other variations from [OBA12]: computing visible clusters by sorting and compacting a screen space buffer of cluster indices instead of using page tables, partitioning based on normals and computing explicit cluster bounds did not yield increases in overal rendering time and are therefore not used.

5.1 Cluster Space Construction

The clustering camera and cluster grid dimensions are determined as described in Chapter 4 on the CPU where all the parameters are available. This computation is linear in the number of cameras per cluster, which very low, and its cost is therefore mostly insignificant. It is possible to have multiple clusterings per frame, for example in case of stereoscopic rendering with `Indi` clustering.

For perspective clustering, the transformation between camera and cluster grid space is non-linear for the z-component. I pre-compute all required coefficients for the forward and reverse transformation and upload them to the GPU so that. For orthographic clustering the transformation from object to cluster index space can be merged into a single matrix.

5.2 Cluster Visibility

One step in the CLS involves determining which clusters are visible so that we need to perform the light assignment only for a small fraction of the clusters in the clustering grid. The original paper describes two methods:

1. Render the cluster index per sample to a buffer and then compute a contiguous array of all unique indices in the buffer.

2. First (virtually) allocate a bit per cluster. For each visible sample mark its corresponding cluster as visible. Compact the indices of visible clusters into a contiguous array.

Rather than finding unique values globally, the first method makes use of the fact that cluster indices are already unique per screen space XY tile and finds the unique values locally per tile before compaction. Since this assumption is no longer valid when the clustering and rendering cameras are decoupled, which is true when using any of the variations presented in this thesis, I use the second method.

Because I am not partitioning based on normals, the required memory for a reasonably sized clustering grid can be allocated at once. For example, a grid with 128 clusters per side using 1-bit per cluster requires only 2MiB of memory. The relatively low memory requirements eliminate the need for a paging scheme. My implementation however uses 32-bits per cluster instead of 1-bit. For the previous example the required memory is then 64MiB which is non-trivial but acceptable for a modern desktop computer. The reason for doing so is that we also need to store the index in the list of visible clusters (if present) for each cluster so that we may look up the light lists during shading. Since the cluster indices are 32-bits in my implementation, we require 32-bits per cluster. The same memory can be used for visibility marking and subsequently to store the mapping since they are required at different times. Since we have 32-bits available per cluster, instead of marking whether or not a cluster is visible, I instead count the number of visible fragments per cluster. The visible fragment counts per cluster were helpful during debugging and are used to generate a histogram when profiling. The histogram gives a sense of the distribution of fragment counts per cluster which can help explain obtained timings for the various stages of the rendering pipeline.

Under certain circumstances, for example on more memory constrained computers like consoles or when using a larger number of clusters, a page table may become necessary. The implementation of a page table may also decrease the cost of visible cluster indices compaction as entire pages of clusters may be skipped, however the compaction has a very low computational cost. The lack of utilization of page tables or another method that creates a hierarchy of clusters is a limitation in the work presented in this thesis.

Since the number of visible clusters is a lot smaller than the total number of clusters, we can more easily associate data with them. Because determining cluster visibility and the compaction happens on the GPU, we can not know how many visible clusters a frame will contain on the CPU side without

synchronization. I use a fixed maximum capacity for data associated visible clusters. On the CPU side, implementations should monitor the number of visible clusters asynchronously and report it when there is insufficient capacity. As usual, the GPU side should be careful not to write past buffer boundaries.

5.2.1 Precision Considerations

We need the cluster index, which is computed from the sample's position in cluster space, to be exactly the same during visibility marking and during shading. During the visibility pass, the fragment position is reconstructed from the depth buffer generated by a depth pre-pass. This means that during shading, we should also use the sample's depth, which is given by `gl_FragCoord.z`, to compute the cluster index. Initially I tried to compute the cluster index from the sample position in cluster camera space which can be computed in the vertex shader and interpolated by the hardware for the sample. However, computing the cluster index this way sometimes resulted in a different cluster index due to limited precision as shown in Figure 5.1. Computing the cluster index using the sample depth in both the visibility pass and when shading makes this a non-issue.

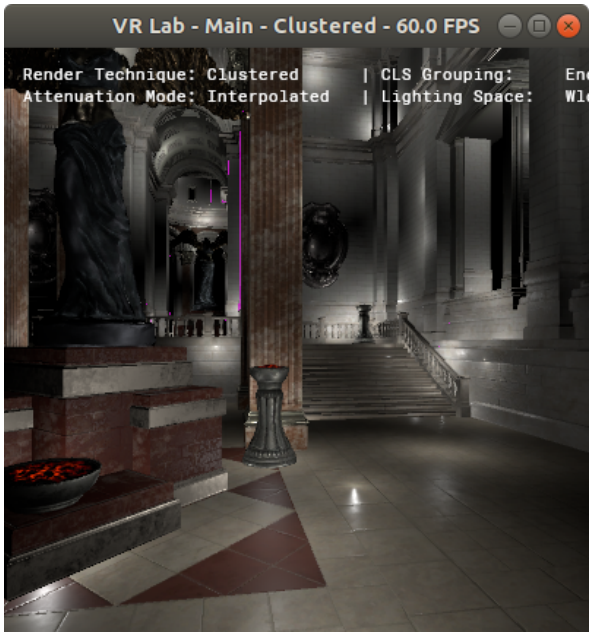


Figure 5.1: *Pixels for which the corresponding visible cluster index is missing are colored magenta. This is true for some samples in the hallway to the left of the center pillar.*

5.2.2 Transparency

In the presence of transparent geometry, more than one sample can be visible per pixel. However, a standard depth buffer can only hold one depth value per pixel. This means that we cannot reconstruct the sample positions of transparent geometry from the depth buffer to mark clusters as visible. To support transparent geometry, I first fill the depth buffer by rendering all opaque and masked geometry, and then I render all transparant geometry. For each generated transparent geometry sample, I mark its corresponding cluster as visible directly from the fragment shader.

5.2.3 Multi-Sample Anti-Aliasing

MSAA is an anti-aliasing technique which decouples visibility sampling from shading. Visibility is sampled at multiple locations within each pixel. If any of the samples within a pixel are covered by the triangle, all these samples together are shaded *once* as shown in Figure 5.2. This is different from super-sampling, which simply increases the sampling resolution.

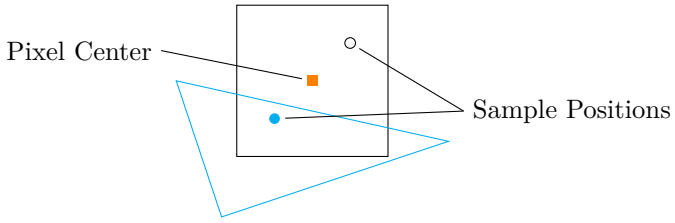


Figure 5.2: *A pixel (black) with sampling locations (circles) partially covered by a triangle (blue). In this scenerio the fragment shader would be invoked once and the resulting color applied only to the bottom left sample.*

As before, to correctly determine the visible clusters using the depth buffer, we must be sure that the computed sample positions are the same during the cluster visibility pass and during shading. For a multi-sampled depth buffer we need to know that the depth value stored for each sample is the same as the `gl_FragCoord.z` passed to the fragment shader during the shading pass. Unfortunately, the OpenGL specification is a bit loose with regards to how MSAA needs to work exactly. This is likely intentional so that GPU vendors have some flexibility in their implementation but it makes it hard to guarantee that the computation of the cluster index is using the same data during the visibility pass and during shading in a cross-GPU-vendor manner. The GLSL qualifier `centroid` can help by forcing the evaluation of vertex attributes at the pixel center $(\frac{1}{2}, \frac{1}{2})$.

The performance of the visibility marking pass is directly related to the sample count. For 16x multi-sampling we read 16 depth values per pixel and perform 16 atomic writes. We could try to deduplicate the number of atomic writes by finding unique depth values per pixel or even per group of pixels. Alternatively we could determine which pixels are *simple*, in the sense that they can be represented by a single sample, or *complex* [NVi19]. Then we process simple and complex pixels in two separate passes. The exploration of further improvements to this method falls outside of the scope of this thesis.

The original motivation for reconstructing visible samples from the depth buffer was that doing so is faster than rendering a second full geometry pass after filling the depth buffer. For a multi-sampled depth buffer, doing a second geometry pass may actually be faster because the hardware then invokes the fragment shader only once for multiple samples. In order to not have the GPU disable the early fragment test in the presence of atomic operations, we must declare `layout(early_fragment_tests) in;` in our fragment shader. Normally, forcing early-z is problematic for masked geometry is problematic because depth and stencil values will be written regardless of a `discard` in the fragment shader. However, since we already have the depth buffer available we can disable depth writes with `glDepthMask(GL_FALSE)`.

5.3 Light Assignment

We must compute for each visible cluster the list of lights of which the influence volumes intersect the cluster's volume. I have implemented this with a two-intersection-passes method. In the first intersection pass, I compute for each visible cluster how many lights intersect it. The partial sums of these lengths are then computed with a parallel prefix-sum yielding starting offsets for each list. A second intersection pass then uses these offsets to actually write the lists of light indices.

Unlike Olsson, Billeter, and Assarsson, I do not use a bounding volume hierarchy over the light sources to accelerate the cluster-light intersection tests. Unfortunately accelerating the light assignment was not implemented due to time constraints. This is an important limitation of this thesis as the cost of light assignment grows linearly with the number of lights in my implementation, while sub-linear scaling is possible. A large light assignment cost means that more time is saved with enclosed clustering, thereby showing larger benefits than would occur in a complete CLS implementation.

5.4 Shading

The renderer uses a simple Physically-Based Rendering shading model adopted from [Vri20]. The implementation has not been particularly optimized, but should be sufficient to represent the amount

of work performed in production-ready applications.

Since CLS requires limiting the influence region of lights, the light attenuation function must be modified so that it smoothly transitions to zero at the boundaries of this region. Appendix A explores various options for the attenuation function of point lights. The chosen attenuation function, parameterized by a light intensity I, an intensity cut-off I_0 and the distance d is:

$$a(d) = \frac{I}{d^2} - I_0^2 \frac{d^2}{I} \tag{5.1}$$

6

Results and Discussion

In the previous chapters I have introduced enclosed clustering, orthographic clustering and displaced perspective clustering. Table 5.1 gives an overview of the implemented methods along with the names used to refer to them. This chapter describes the evaluation method in Section 6.1. Then I proceed to find good parameters for `Persp`, `Ortho` and `Displ` clustering by considering them individually in Section 6.2. These parameters are established so that we may use them when comparing `Indi` with `Encl` clustering in Section 6.3 and when comparing `Persp`, `Ortho` and `Displ` clustering with each other in Section 6.4.

6.1 Evaluation Method

As mentioned in Chapter 5, I developed a OpenGL based renderer in Rust with the specific goal of evaluating the work in this thesis. This section describes the exact method of evaluation and common configuration values used to obtain the results presented in the remainder of this chapter.

6.1.1 Profiling

The renderer can record the initial camera state and all subsequent user input. I obtain profiling results by replaying these recordings and taking various measurements every frame. These measurements can be either time sensitive or not. Time sensitive, or *temporal*, measurements require a different approach than *atemporal* measurements.

Temporal Profiling

For the evaluation of the computational performance I record the time elapsed on both the CPU and GPU for various sections of the rendering process. Because the obtained values are dependent on time, I call this type of profiling *temporal profiling*. The only CLS computation happening on the CPU is the construction of the clustering space. The time spent doing so per frame is insignificant. All significant CLS computations happen on the GPU and all timings shown in this chapter represent time elapsed on the GPU. Since the profiling was executed on a time-sharing operating system, our renderer does not have exclusive access to the GPU. This means that we may measure time spans that include not only our computations, but also work caused by other applications, making the time span larger. The replayed simulations are deterministic, which allows replaying the same capture multiple times to obtain

multiple measurements of the same computation. I take the minimum of each sample across multiple runs to estimate the best-case timings.

Atemporal Profiling

There are a number of useful measurements we can take that depend only on the simulation and not on the execution time. For example, the sequence of rendered frames should be the same for any run because the simulation and rendering is deterministic. To record this information I perform an single *atemporal profiling* run for each profiling configuration. Atemporal measurements are disbled during temporal profiling runs so as not to influence temporal measurements.

6.1.2 Scenes

Two different scenes were used for profiling: the Amazon Lumberyard Bistro Exterior scene [Lum17] and the UE4 Sun Temple scene [Gam17]. Table 6.1 gives an overview of the scenes and some of their details. The results of both scenes ended up being very similar so I have chosen to only show the results from the `Bistro` scene in line with the text. The `Bistro` scene is the more demanding scene. The results gathered using the `Suntem` scene are included in Appendix C.

Name	Triangles	Transparency
`Bistro`	3.0M	yes
`Suntem`	1.6M	no

Table 6.1: *Overview of the scenes used in profiling.*

6.1.3 Camera Configuration

The far plane is set to 100 units for the rendering camera. For monoscopic rendering the field of view is set to 90 deg. For stereoscopic rendering I use the frustum tangents provided by the HTC-Vive. The camera far plane and field of view are not varied in the following experiments. The effects are these parameters are clear:

- A larger depth range leads to more clusters for any of the described methods.
- A larger field of view leads to more clusters in case of `Ortho` and `Displ`, but not in case of `Persp`.

Having a larger cluster grid means more clusters may be visible which means more light assignment work. Care must be taken, especially for `Ortho` as it is very sensitive to the depth range and field of view, that the dimensionality of the grid doesn't become so large that we can no longer allocate 32-bits per cluster.

6.1.4 Lighting Conditions

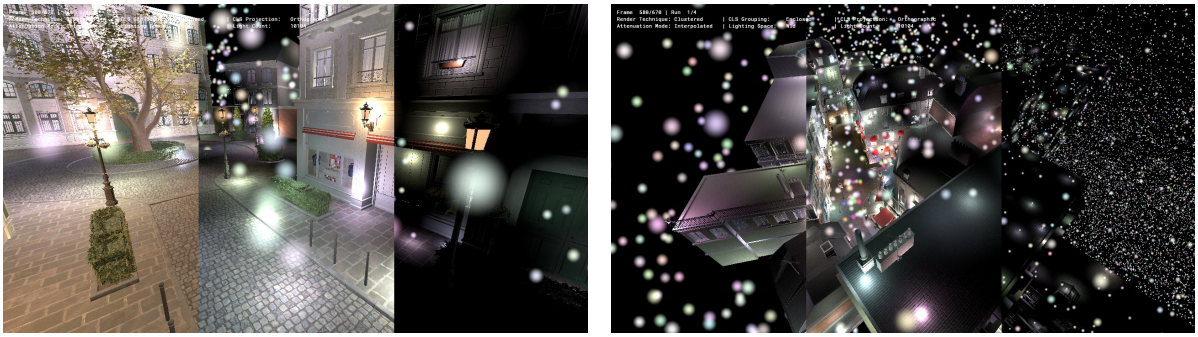
Applications have varying requirements on the number of lights that need to be supported. To get an idea of how the number of light affect the performance of various methods I use three different light configurations: 1000, 10.000 and 100.000 lights. These lights are randomly placed in the bounding box of the scene. Each light is simulated to fall down in a manner similar to rain, except no collisions occur. Lights reaching the bottom of the bounding box are recreated randomly at the top of the bounding box. Regardless of the total number of lights in the scene, the number of lights that can be considered at most per pixel is very much determined by the processing power of the used hardware. To keep the frame times reasonable, I use chose the light volume radius (denoted by R_1) in such a way that the number of lights per pixel are similar between the light configurations.

6.1.5 Machine

The profiling was performed on a desktop computer sporting a GeForce GTX 1070 Ti. The rendering resolution was 1280x720. The operating system was the linux-based Ubuntu.

6.2 Cluster Construction Method Parameters

The exploration space of the proposed methods and their parameters is very large. To reduce the exploration space, I determine close-to-optimal parameters for each cluster construction technique individually. This is done for all scenes and lighting conditions, as the differences in geometry and lighting may require different parameters.



(a) *Frame 300* **(b)** *Frame 580*

Figure 6.1: *Two frames from the Bistro scene replay. From left to right: 1000, 10000 and 100000 lights. The reason that the images get increasingly dark despite a higher light count is that the light radii were chosen to be smaller to keep the average number of lights per pixel similar. The radii are shrunk by decreasing the light intensity. The larger light count but diminished intensity together produce a smaller total illumination due to the used attenuation function. Note that in frame 580, most of the the visible geometry is fairly far from the camera. For* `Ortho`*, this leads to a large number of visible clusters. For* `Persp`*, this leads to large clusters with many lights.*

From a shading perspective, we would like to iterate over the smallest possible number of lights. To achieve this, we need high clustering resolution (a large number of clusters in the clustering grid) so that we precisely cull the lights. From a clustering perspective however, a fine clustering resolution means that we must compute the light lists for many clusters. In this section we essentially try to establish a cluster size for each method such that the total frame computation time is minimal. There is no reason to find exact optima as small changes to the scene, the lighting conditions, or even the replay can change the optimum number dramatically. As long as we are largely in the right ballpark, we can make reasonable comparisions. Figures 6.2, 6.3 and 6.4 show profiling results for `Persp`, `Ortho` and `Displ` clustering gathered with the `Bistro` scene replay. The plots show the total frame time along with various temporal and atemporal profiling measurements for each frame of the replay. By inspecting the plots visually, I judged that `Persp` 64, `Ortho` 4 and `Displ` 64 32 had the best overall frame-time performance.

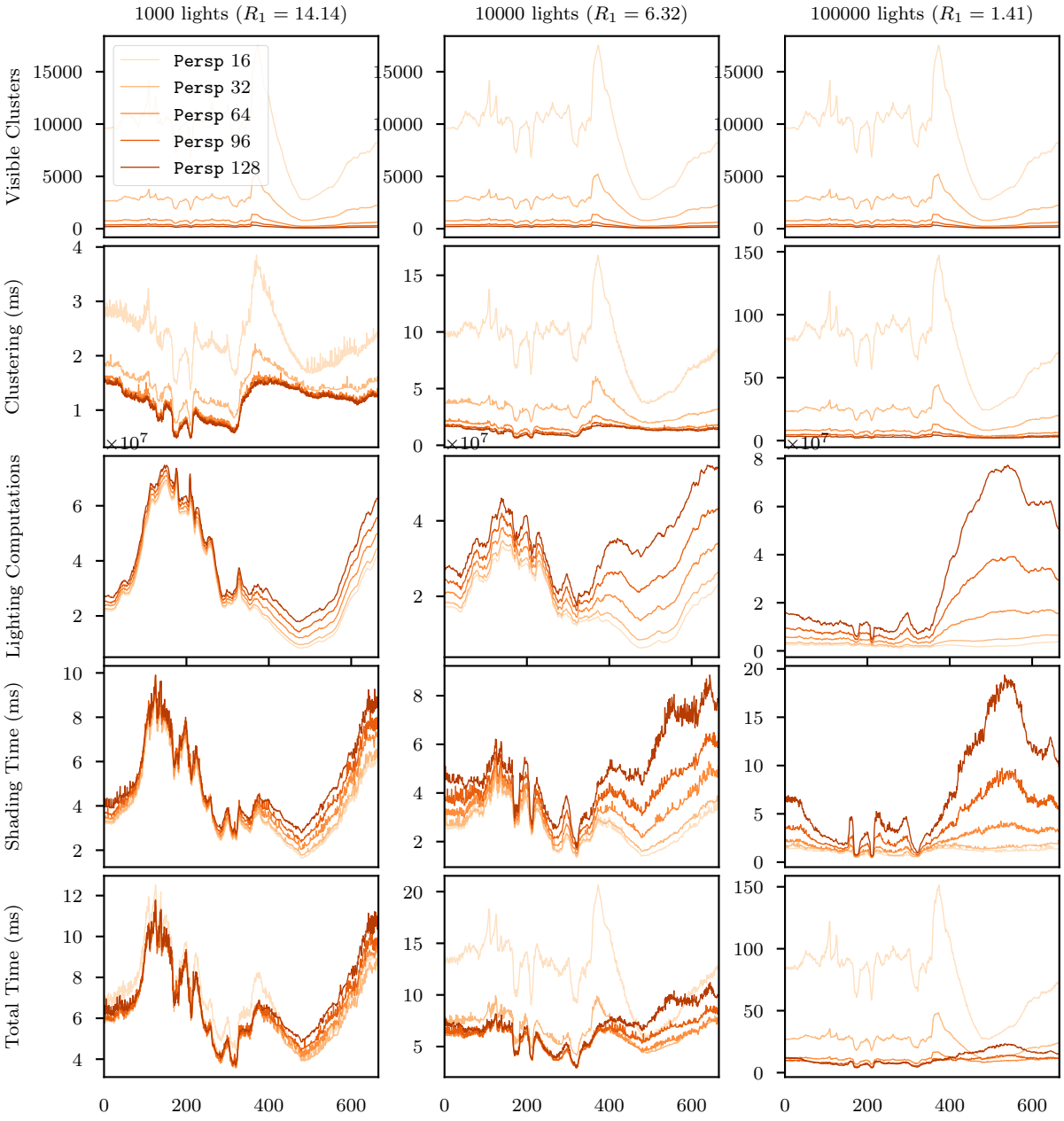


Figure 6.2: *The profiling results of the* `Bistro` *scene using* `Persp` *clustering. Through visual inspection, I judge* `Persp` *64 to perform the best among the tested sizes. Around frame 580 only a small part of the render contains visible geometry and the geometry is far away from the camera origin. This, as is to be expected, leads to only a small number of clusters being marked as visible. The low visible cluster count consequently explains the low clustering time. Each of these clusters, however, covers a large amount of space. Since the lights were distributed uniformly over the bounding box of the scene, more lights are assigned to each cluster. Even through there are less fragments being rendered around frame 580 (because more of the black background is visible), the overall shading time is increased due to a large number of lights per cluster.*

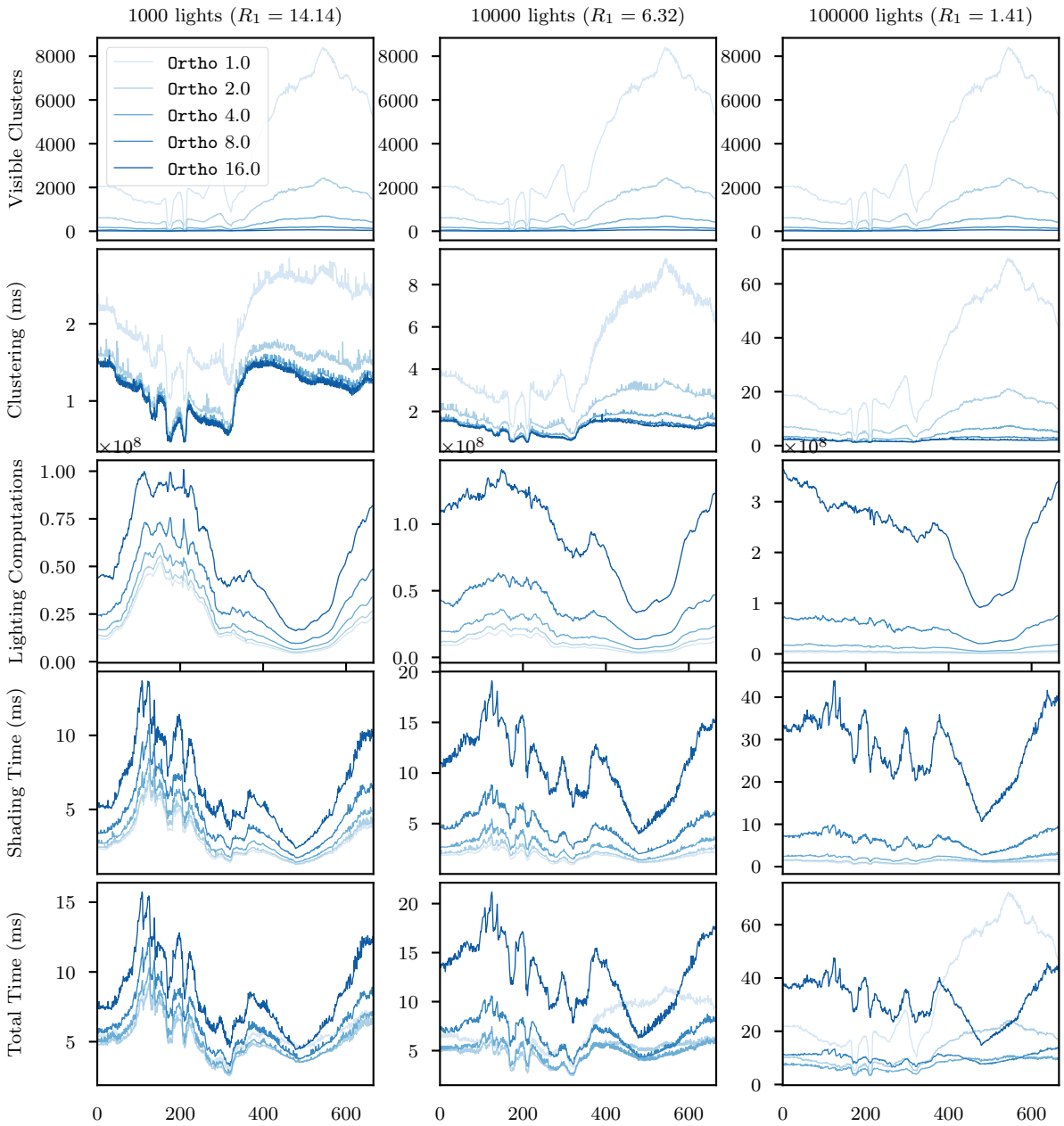


Figure 6.3: *The profiling results of the* `Bistro` *scene using* `Ortho` *clustering. Through visual inspection, I judge* `Ortho` *4.0 to perform the best among the tested sizes. As is to be expected, the clustering time is the largest around frame 580 where the visible geometry is far away from the camera, leading to a large number of visible clusters. The shading time actually decreases around that time because a large part of the render doesn't contain any geometry at all: only the black background is showing.*

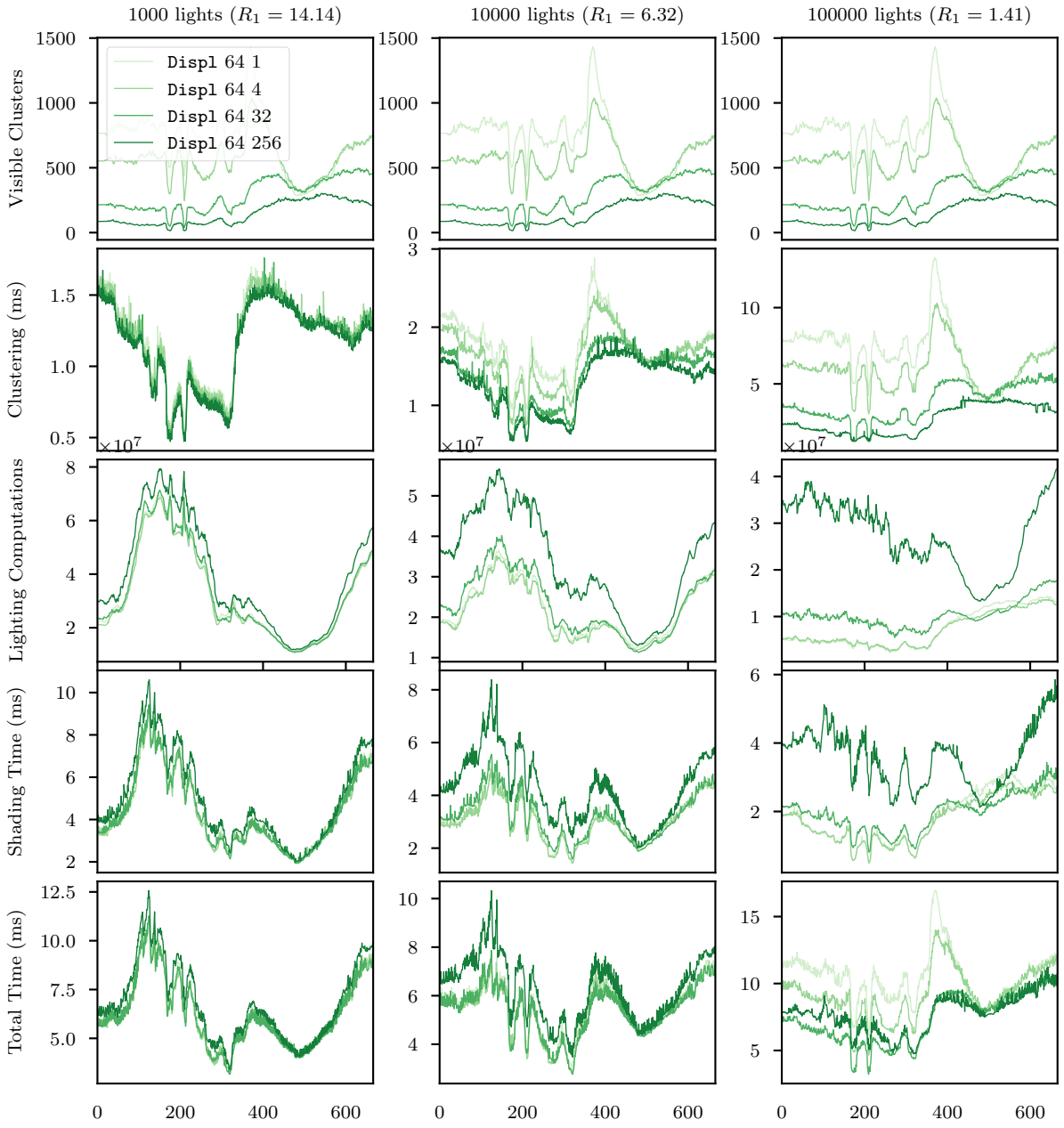


Figure 6.4: *The profiling results of the* `Bistro` *scene using* `Displ` *clustering. Through visual inspection, I judge* `Persp` *64 32 to perform the best among the tested sizes.*

6.3 Enclosed Clustering

CLS can be used for stereoscopic rendering by simply applying the technique to both eyes individually (`Indi`). Enclosed clustering (`Encl`) adapts CLS to stereoscopic rendering and performs the light assignment step of CLS only once for both eyes together, instead of twice. This requires decoupling the clustering camera from the rendering camera which incurs extra computational costs. In this section I show that the time gained by reducing the number of light assignments by almost half easily outweighs the time lost due to this decoupling.

Figure 6.5 compares the profiling results for `Indi` and `Encl` using `Persp` and `Ortho` with the parameters found in the previous section. The `Displ` clustering method was left out since it is a mix between `Persp` and `Ortho` and would only clutter the graphs. As described in Section 4.1.1, the dimensions of the cluster grid are determined using the desired cluster volume. Because the enclosing frustum covers more space than either the left or right eye's camera frustum alone, the dimensions of the grid are also larger. The total number of clusters in the grid is therefore not exactly halved. However, because we keep the resolution of the clustering grid somewhat constant by increasing the dimensions of the grid based on the desired cluster volume, the total number of visible clusters *should* be close to halved.

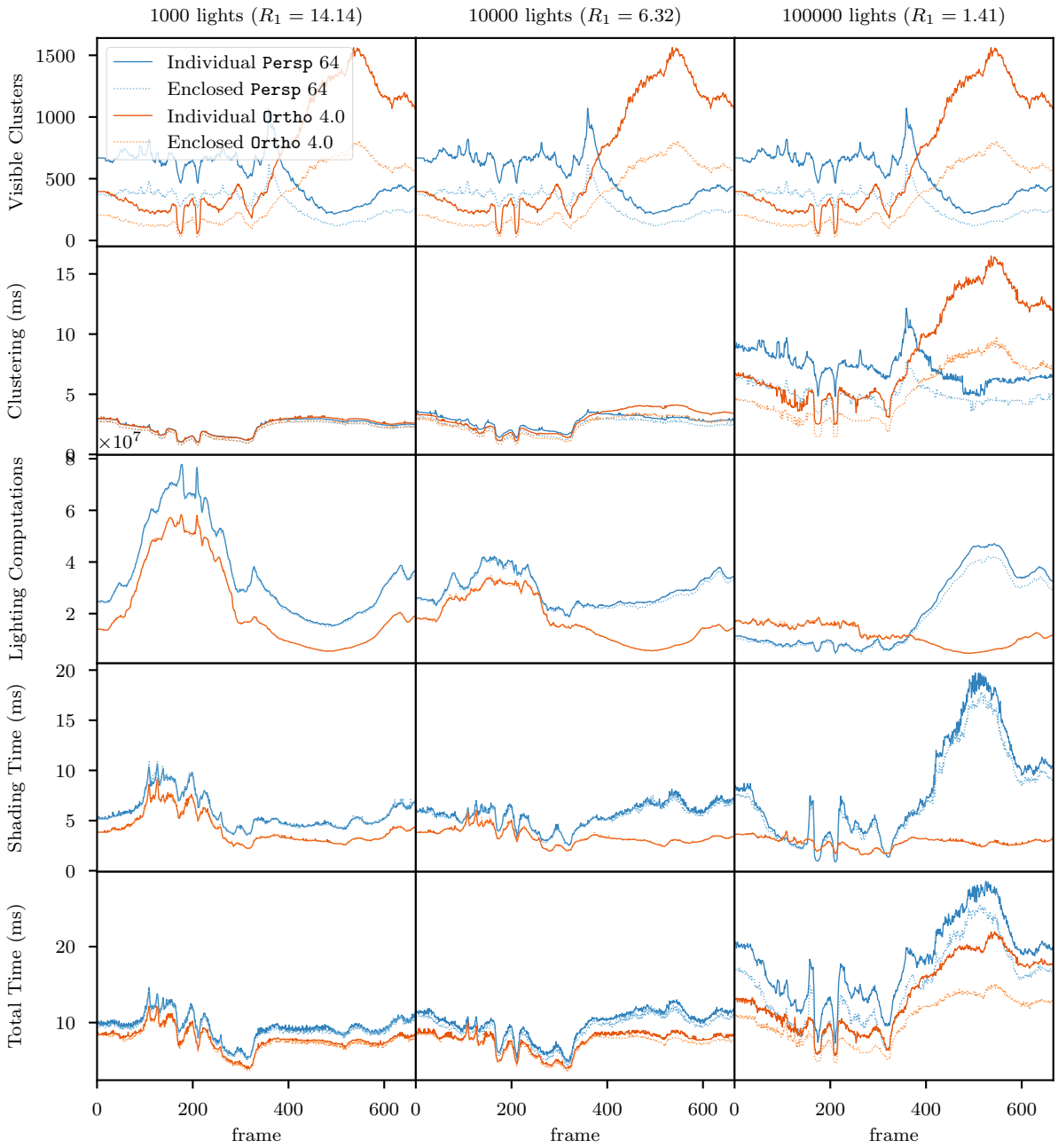


Figure 6.5: *Profiling results of* `Indi` *and* `Encl` *clustering for the* `Bistro` *scene. The total visible cluster count is halved as expected. This translates into a lower clustering time, especially in case of many lights (100.000 and lack of acceleration structure) or clusters (*`Ortho` *with far-away geometry around frame 580). When put in perspective of the total frame time, the performance gains are marginal because the light assignment is already a relatively small step in the entire rendering process. Interestingly, the number of lighting computations seems to actually decrease for* `Persp`*. This observation actually inspired the* `Displ` *method. What I expect is happening is that due to the origin of the enclosing camera being moved backwards, the clusters near the camera become bigger. The distribution of the density, or resolution, of clusters in camera space essentially shifts from being very focused at the near plane, to a more equal spread over the depth range. Having a smaller number of very small clusters near the camera benefits the clustering, and having more resolution far away from the camera benefits the culling and therefore shading.*

6.4 Orthographic- and Displaced Perspective Clustering

The development of `Encl` clustering opened up a path to alternative ways of constructing the clustering camera and grid. In this section we compare traditional perspective based clustering (`Persp`) with, the traditionally dismissed, uniform world-space grid clustering (`Ortho`) and with, the in this thesis introduced, diplaced perspective clustering (`Displ`). Decent cluster size parameters were found for each method individually in Section 6.2. Figure 6.6 contains the profiling results obtained for the tuned methods with the `Bistro` scene. More detailed timing information on is presented in Figures 6.7 and 6.8.

The distribution of clusters with respect to the number of lights they intersect and the number of fragments they contain are presented in Figures 6.9 and Figure 6.10 respectively. The distributions are recorded as histograms for every frame and visualized as a heatmap.

It is difficult to discuss the potential of the presented methods for real applications as the light assignment step is not accelerated. In other work they actively try to retain the tiling of TBS in CLS approaches to minimize execution divergence. Decoupling the clustering camera from the rendering camera, as we must for `Ortho` and `Displ`, makes such a screen space tiling impossible. Regardless, there seems to be some wasted precision close to the near plane for `Persp`. When rendering scenes with uniform light distribution, a relatively small camera depth range, and large depth discontinuities within this range, `Ortho` and particularly `Displ` are worth exploring. Once the decoupling of the clustering camera from the rendering camera is implemented, supporting all of the variations explored in this thesis requires little additional implementation work.

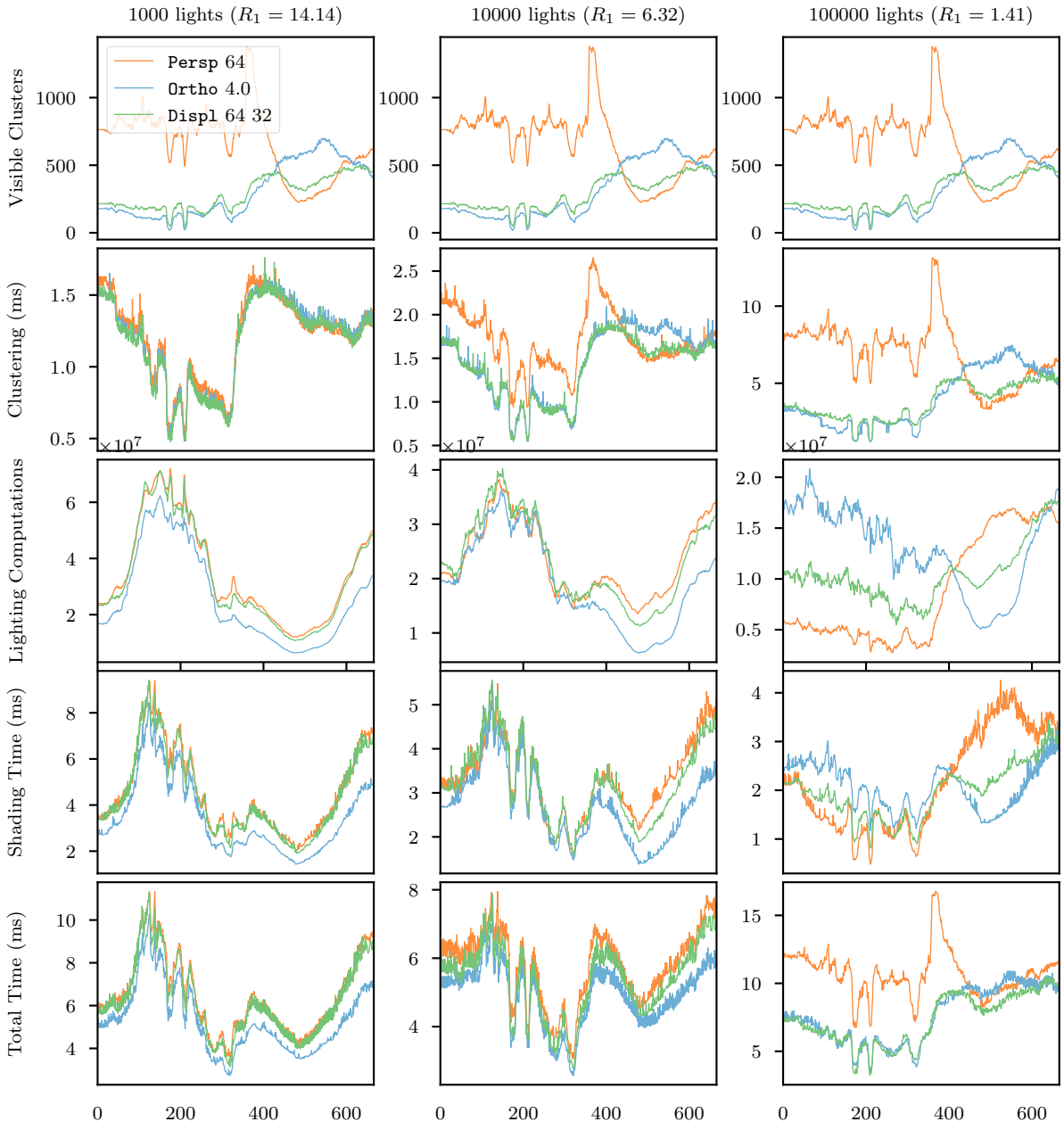


Figure 6.6: *The profiling results of the* `Bistro` *scene using the tuned* `Persp` *64,* `Ortho` *4, and* `Displ` *64 32.* `Displ` *seems to deliver a more stable frame time than either* `Persp` *or* `Ortho` *alone. It is able to deal with the visible geometry being both close-by and far-away. However, these results are specific to the scene and lighting conditions. When far-away lights are fewer with larger radii,* `Persp` *may be a much better choice still. On the other hand,* `Ortho` *will benefit greatly from accelerating the light assignment [OBA12].*

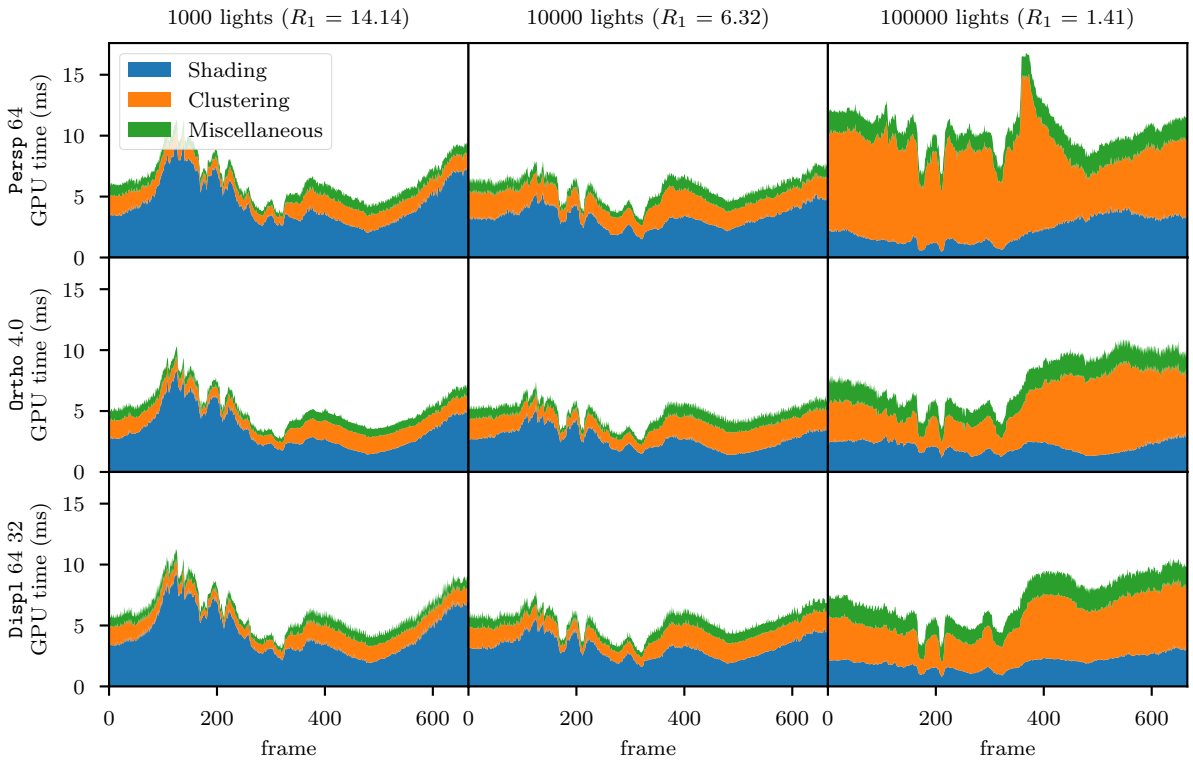


Figure 6.7: *Frame breakdown, Bistro scene. The miscellaneous category captures GPU work unrelated to either clustering or shading. Examples of this work are culling, filling indirect draw buffers, light simulation, and framebuffer blitting.*

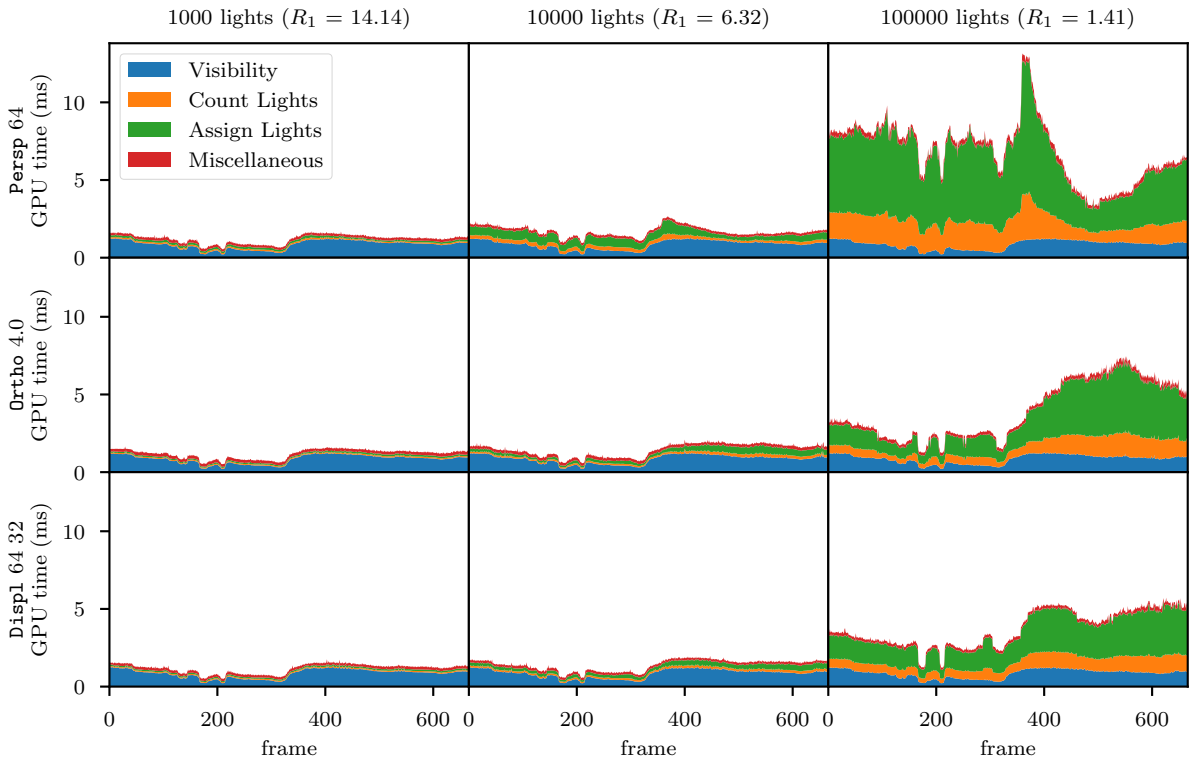


Figure 6.8: *Clustering breakdown, Bistro scene. The Visibility category includes the depth pass and counting visible fragments. The miscellaneous category captures uploading and transforming lights, computing prefix sums and other small work.*

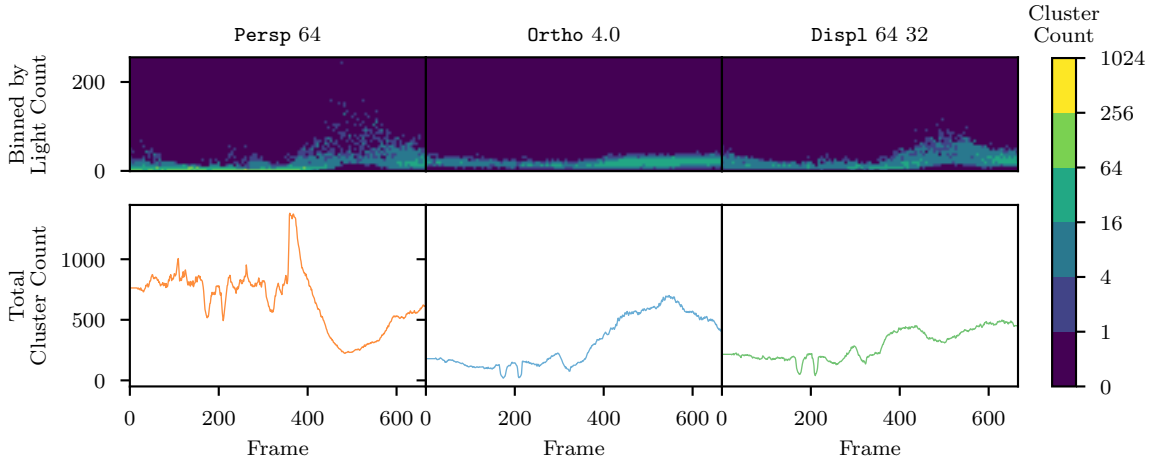


Figure 6.9: *The distribution of clusters with a certain light count. As we move the camera away from the visible geometry (frames 400 - 600), the visible clusters become bigger for* `Persp` *and as a result contain more lights. For* `Ortho` *however, all clusters are the same size but more of them are visible.*

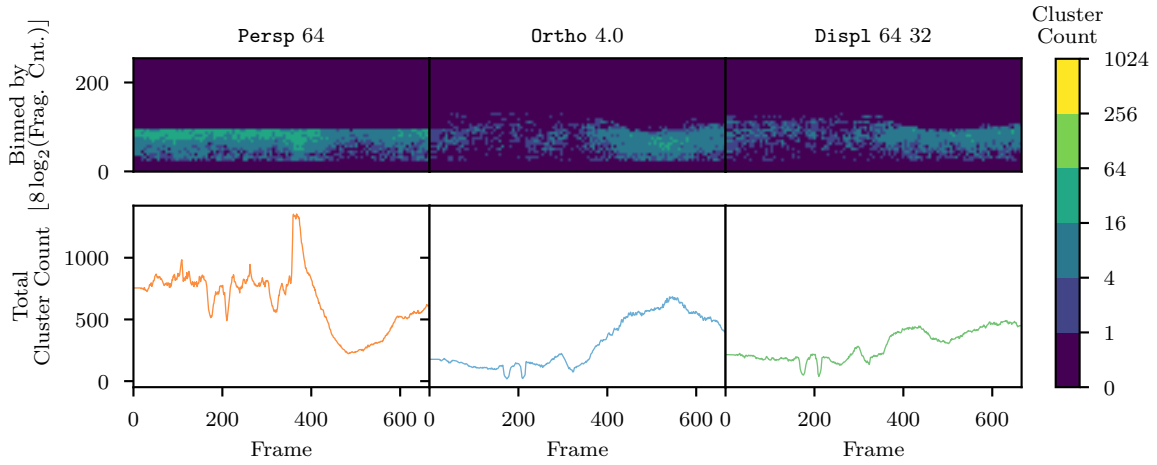


Figure 6.10: *The distribution of clusters with a certain fragment count. We see that a cluster can contain at most* 64^2 *fragments per cluster for* `Persp` *64 as we only take 1 sample per pixel.*

7

Conclusion

The main contribution of this thesis is enclosed clustering (`Encl`) which improves CLS in a stereoscopic rendering context by reducing the amount of light assignment work. Besides enclosed clustering I presented orthographic (`Ortho`) and displaced perspective clustering (`Displ`) which construct the clustering camera and grid in a slightly different way. These methods and their implementation were explained in detail in Chapters 4 and 5. In Chapter 6 I presented and discussed temporal and atemporal performance metrics obtained through extensive and careful profiling of the presented, and existing, methods. This chapter briefly discusses the results of each presented method and closes with possible directions for future work.

7.1 Enclosed Clustering

Enclosed clustering should always be used for stereoscopic rendering. The decreased total number of light lists to be computed outweighs the costs introduced by decoupling the clustering from the rendering camera. For unaligned views, orthographic clustering can be used. This enables using more lights in scenes with mirrors or other reflections where the same space is viewed from multiple directions.

7.2 Orthographic Clustering

Orthographic clustering is a competitive way of defining your clusters. It lends itself well to enclosed clustering, even if the views are not aligned. Implementation is simple because clusters are axis aligned bounding boxes. It is possible to use the world space orientation for this clustering which may eliminate computations such as transforming the lights into cluster camera space.

The use of orthographic clustering can lead to a large number of visible clusters. This is problematic because light assignment has to be performed for every visible cluster. Using a large camera depth range or field of view is therefore a challenge. Decreasing the cost of light assignment greatly benefits this technique.

7.3 Displaced Perspective Clustering

Perspective clustering is an established technique. However, the importance of clusters being formed by dividing screen space XY tiles along the Z-axis and the resulting screen space alignment may be less

important today. Letting go of this desire allows us to deal with an unnecessarily high cluster resolution close to the camera near plane by moving the origin of the clustering camera backwards. Doing so allows us to get the good parts of both ortographic and perspective, clustering. This technique is more robust with respect to the expected frame time when varying the distance to visible geometry than either technique alone. For enclosed clustering, the clustering camera is already decoupled from the rendering camera making this technique come at no additional cost.

7.4 Future Work

The renderer written to support the work in this thesis has, although the performance is good, a number of shortcomings due to time constraints. The most obvious and painful one being the lack of acceleration of the light assignment. This section describes some of these practical improvements and some more open directions for future work.

Reoriented displaced perspective clustering

In Section 4.3 I displace the camera by the central axis of the frustum. Doing so with a large displacement leaves the resulting frusta looking like skewed boxes when the camera frustum is assymetric. To make the boxes symmetric, we can re-orient the enclosing frustum to be symmetric. This is similar to the reorientation of shadow mapping camera frusta in [WSP04].

Per-tile clustering

Following the previous suggestions, we could eliminate the skew of clusters around the edges of the clustering camera frustum by using a frustum per XY tile of clusters. We would need to figure out how to efficiently implement the frustum-light intersections. Increasing the cluster grid resolution to infinity, we would be looking at a sort of spherical projection, rather than perspective projection.

Support more kinds of light sources

Currently I have described only point light sources. To support more light sources, we have to find appropriate frustum-light intersection tests and an attenuation function satisfying the requirements set in Appendix A.

Cascaded or Hierarchical Orthographic Clustering

For aligned views, we could use a cascade of clustering sizes. For unaligned views, we could use a hierarchical data structure. Efficiently implementing visibilty marking for a hierarchical data structure is challenging because we probably cannot allocate enough memory to hold an implicit tree. A practical implementation would involve dynamic memory allocation.

By using a hierarchical data structure, it would also be possible to aggregate far-away light contributions by merging point lights. Because of the cut-off intensity I_0, many small lights do not contribute to far-away surfaces individually. When merged together, the larger intensity increases the influence radius. Blending between the contribution of the individual light sources and the merged representation will be challenging. Several techniques for merginging lights are described in [Dac+14].

Adaptive Cluster Size

Stability is important for real-time graphics. It would be interesting to try and figure out a way to find the optimal clustering technique and parameters at run time. There are many parameters that can be varied. For example the displacement distance of displacement perspective clustering or the cut-off intensity I_0 from the light attenuation.

Accelerate Light Assignment

Orthographic and displaced perspective clustering should be re-evaluated after accelerating the light counting and assignment. In the case of many clusters and many lights, it may be useful to employ a dual tree algorithm like the one described in [Kol+19].

Improving Specular Highlights

Specular highlights are visible far outside of the light's influence radius. Currently they are diminished because of the attenuation function. To increase realism, it would be interesting to try to use cluster normals [OBA12] and view direction to include lights in the specular lobe. Finding a method that is temporally stable will be challenging.

A

Choosing a light attenuation function

What is commonly known as the *brightness* or *intensity* of a light source is more accurately defined by the radiant intensity $I_{e,\Omega}$ in watt per steradian (W/sr). You can think of it as the amount of energy emitted or received per second over $\frac{1}{4\pi}$th of the surface of the unit sphere.

Point lights emit light in all directions equally. The total power emitted does not change as we move further away from the light, but the surface of the sphere it is spread over does. In fact, if we define I to be the power received by the unit sphere. Then the power received per area by a sphere with radius d is $I \cdot d^{-2}$, because the area grows by a factor d^2.

$$a_{phy} = \frac{I}{d^2} \tag{A.1}$$

The function a_{phy} has two issues. First of all $\lim_{d \to 0} a_{phy} = \infty$. When the distance to the light source becomes very small, the intensity becomes very large. Very large intensities lead to precision problems and require a high-dynamic range rendering pipeline.

Secondly, we want lights to have a limited radius of influence. This means that we need the attenuation to be 0 at this radius. However, $a_{phy} \neq 0$ for finite d. We would like the light's radius to be as small as possible so that we reduce the number of lights we need to compute the contribution for.

This chapter explains how I arrived at the attenuation function used in the renderer. I start by defining intuitive parameters. Using these parameters I define a variety of candidate attenuation functions. These functions are compared visually. Finally, I motivate the selection of the attenuation function used in the remainder of this thesis.

A.1 Function parameters

The attenuation function given in Equation A.2 is commonly seen in computer graphics, for example in the 3D modelling software Blender. Apart from the intensity I, the function is parameterized by three coefficients (c_0, c_1 and c_2).

$$a_{clq} = \frac{I}{c_0 + c_1 d + c_2 d^2} \tag{A.2}$$

With the right coefficients a_{clq} can be made smooth and reasonable. Tuning these three coefficients is not intuitive, so we take a step back to try and find more intuitive parameters.

To limit the maximum attenuation, we define a near-radius R_0 which we use as the minimum d. If we take $R_0 = 1$ then the maximum intensity should be I. Choosing the right R_0 depends on the scene and the light that is being modeled.

Rather than directly defining the light's outer radius R_1, we parameterize our equations by a cut-off intensity I_0. We then find R_1 by solving $a_{phy}(R_1) = I_0$ which gives $R_1 = \sqrt{\frac{I}{I_0}}$. Choosing a smaller I_0 leads to a larger R_1. The cut-off I_0 directly determines how accurately you want to approximate a_{phy} and is independent of the light intensity I.

A.2 Considered functions

We want the attenuation to be zero at R_1. There are many ways of achieving this. The simplest would be to simply subtract I_0 from a_{phy}. We call this the "reduced" attenuation function a_{red}.

$$a_{red} = a_{phy} - I_0$$

To ensure a smooth transition from lit to unlit, we can force both the attenuation and its derivative to zero. We solve for two coefficients c_0 and c_1.

$$a_{smo}(d) = a_{phy} + c_0 \cdot d + c_1 \qquad a_{smo}(R_1) = a'_{smo}(R_1) = 0$$
$$a_{smo}(d) = a_{phy} + \frac{2I}{R_1^3} \cdot d - \frac{3I}{R_1^2} \qquad a_{smo}(d) = a_{phy} + \frac{2I_0}{R_1} \cdot d - 3I_0$$

We can interpolate between attenuation functions to generate more options. We do this using t_1 and t_2 as defined below. If only the squared distance is available, t_2 has an advantage over t_1 since it does not require computing d.

$$t_1 = \frac{d}{R_1} \qquad t_2 = \frac{d^2}{R_1^2}$$
$$a_{i,j,n} = (1 - t_n) \cdot a_i + t_n \cdot a_j$$

We can use $t_n = t_1^n$ to create an even sharper transition between the two interpolated functions. Sharp transitions can introduce an inflection point in the attenuation function which is undesirable. For this reason I only consider interpolation u sing t_1 and t_2.

Since a_{red} and a_{smo} incorporate a_{phy}, we can simplify the resulting functions. Lets say $a_j = a_i + a_j^*$, then we can simplify $a_{i,j,n}$.

$$\begin{aligned} a_{i,j,n} &= (1 - t_n)a_i + t_n(a_i + a_j^*) \\ &= a_i - t_n a_i + t_n a_i + t_n a_j^* \\ &= a_i + t_n a_j^* \end{aligned}$$

$$a_{phy,red,1} = \frac{I}{d^2} - I_0 \frac{d}{R_1} \qquad a_{phy,smo,1} = \frac{I}{d^2} + 2I_0 \frac{d^2}{R_1^2} - 3I0 \frac{d}{R_1}$$
$$a_{phy,red,2} = \frac{I}{d^2} - I_0 \frac{d^2}{R_1^2} \qquad a_{phy,smo,2} = \frac{I}{d^2} + 2I_0 \frac{d^3}{R_1^3} - 3I0 \frac{d^2}{R_1^2}$$

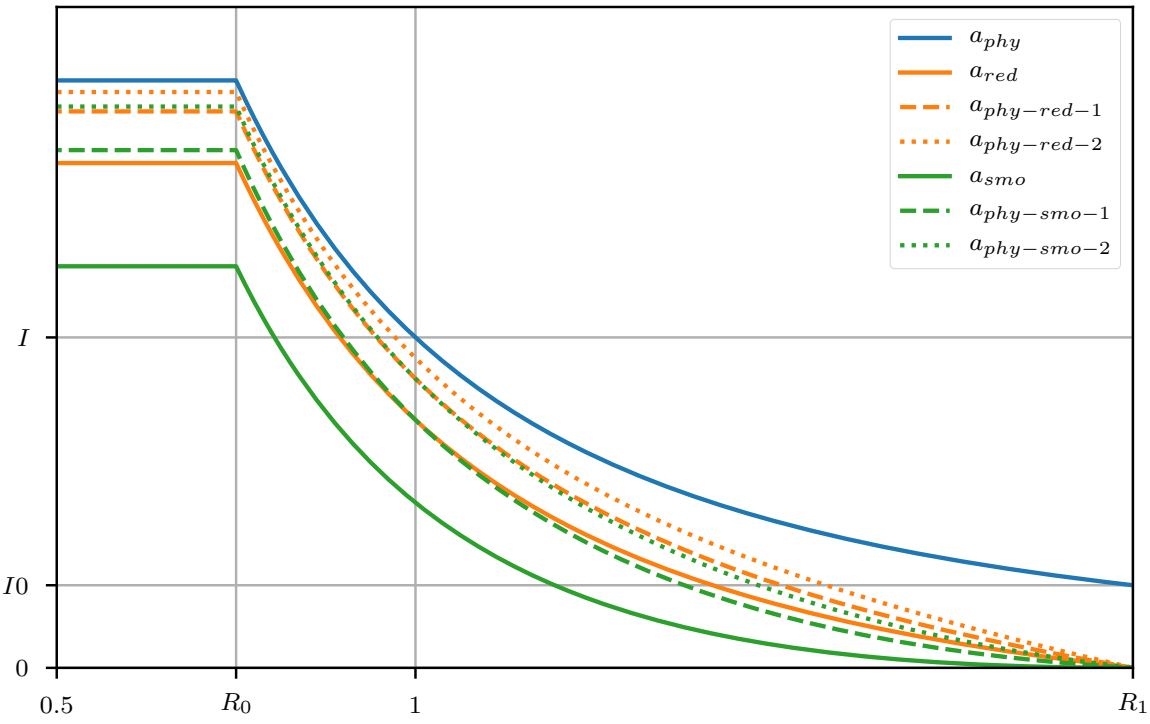


Figure A.1: *Basic attenuation functions for different light intensities.* $R_0 = 0.75$, $I_0 = 0.25, I = 1.00$. *This figure shows that we must make a trade-off between a smooth transition to zero at* R_1*, and staying true to* a_{phy}.

The attenuation functions we have defined have the desired property that $\lim_{I_0 \to 0} a = a_phy$. This means that the smaller we make I_0, the closer we get to the correct attenuation. However, R_1 simultaneously grows to infinity which means all clusters will contain all lights, defeating the purpose of clustered light shading.

For completeness, we also define a linear attenuation function a_{lin} and a step attenuation function a_{stp}. These functions are useful for debugging and illustration.

$$a_{lin} = I * \frac{R_1 - d}{R_1}$$
$$a_{stp} = I * step(d, R_1)$$

A.3 Evaluation

We want each light that we do calculations for to contribute significantly to the scene. Figure A.2 shows the results for a_{lin}. While this function does make every light contribute a lot to the scene, I think that with physically based rendering and current hardware the attenuation function of choice should resemble a_{phy} as much as possible.

In Figure A.3 we see that the effect of "cutting off" our light influence volumes at R_1 is very noticable. When I_0 is high, a lot of brightness in the scene is lost and far-from-the-light specular reflections are lost.

While the idea of having a smooth attenuation function is reasonable, the light contribution is much less than that of a_{red} for example. Figure A.4 compares three functions which we consider good representatives of all functions we have considered.

For the remainder of this thesis, I use $a_{phy,red,2}$ because I think it looks good and it is cheap to

(a) a_{lin} *with* $I_0 = 1.00$ **(b)** a_{stp} *with* $I_0 = 1.00$ **(c)** a_{stp} *with* $I_0 = 0.25$

Figure A.2: *Linear and step attenuation. The limited light radius is especially clear when we look at the specular reflections. Notice how* I_0 *affects the light radii. Using* $I_0 = 0$ *would result in physical attenuation.*

compute. The attenuation function needs to be evaluated for each directional light source and therefore affects the shading time. Studying this effect is out of the scope of this thesis.

(a) a_{phy} *with* $I_0 = 0.00$ **(b)** a_{phy} *with* $I_0 = 1.00$ **(c)** *Debug view*

Figure A.3: *Physical attenuation with different influence radii. In A.3b we use clustered light shading. The light assignment was done using the light radii* R_1*. Since* $a_{phy}(R_1) \neq 0$*, the cluster boundaries are visible.*

(a) a_{red} *with* $I_0 = 0.25$ **(b)** a_{smo} *with* $I_0 = 0.25$ **(c)** $a_{phy,red,2}$ *with* $I_0 = 0.25$

Figure A.4: *Comparing a selection of our limited radius attenuation functions. All functions decrease the overall brightness compared to* a_phy*. Out of them,* $a_{phy-red-2}$ *maintains most of the brighness without any noticable edges.*

B

Global Illumination

As mentioned in [Dac+14], CLS can be used to accelerate rendering with virtual point lights.

Figure B.1: *Bistro scene with 5000 virtual point lights. Many of the lights do not contribute significantly.*

I attenuate the point lights by $\max(0.0, \dot{(}light\_normal, frag\_to\_light)^{\frac{1}{2}})$ and use a fairly large minimum distance of 1.0 for the point light attenuation. The VPL's are generated by rendering a RSM cube map and sampling it in pseudo random directions.

Figure B.2: *Color bleeding is visible near the vegetation and red objects.*

Figure B.3: *Sun Temple scene with 1000 VPLs.*

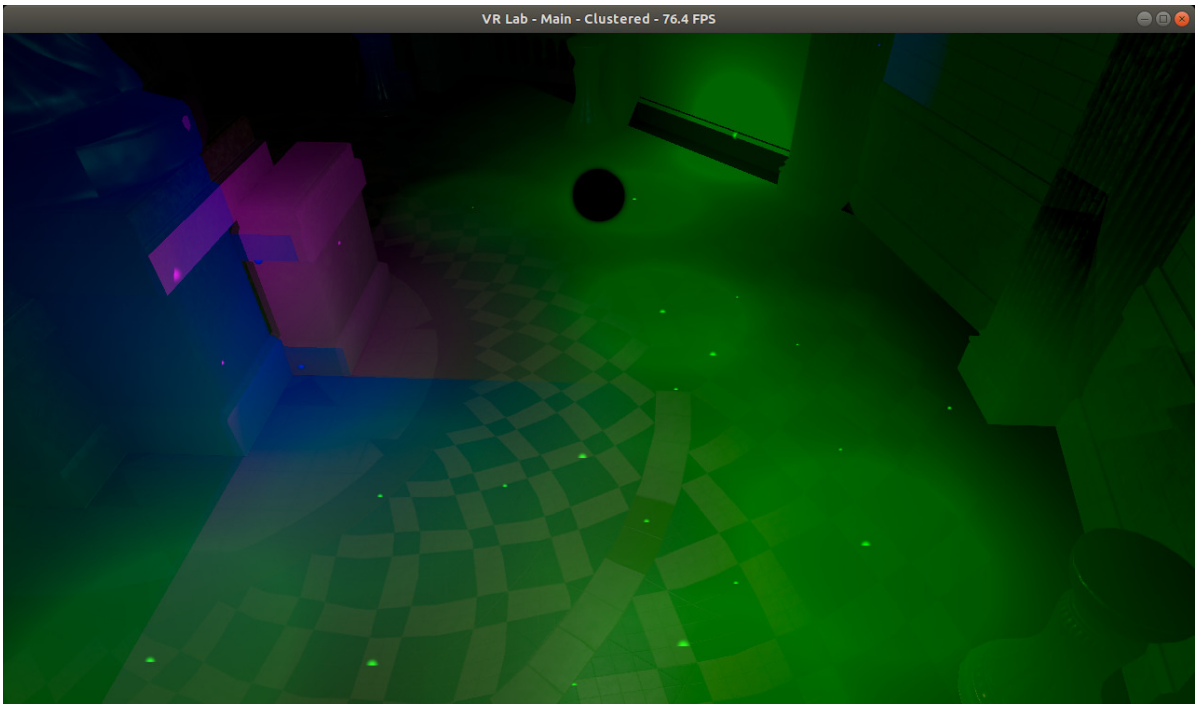


Figure B.4: *Virtual point lights as hemispherical lights, using the normal for color. Hemispherical lights create hard edges. Some of the green VPLs unintenionally illumintate the surface they are on. This can be fixed by offsetting the light positions and/or using a small zero-influence radius.*

C

Sun Temple results

Figures C.1 and C.2 show frame 250 and 680 of a fly-through of the Sun Temple scene under various lighting conditions.

Figure C.1: *Frame 250 of the Sun Temple scene replay. From left to right: 1000, 10000 and 100000 lights.*

Figures C.3 and C.4 show that the Sun Temple scene leads to the same preferred sizes for orthographic and perspective clustering as the Bistro scene.

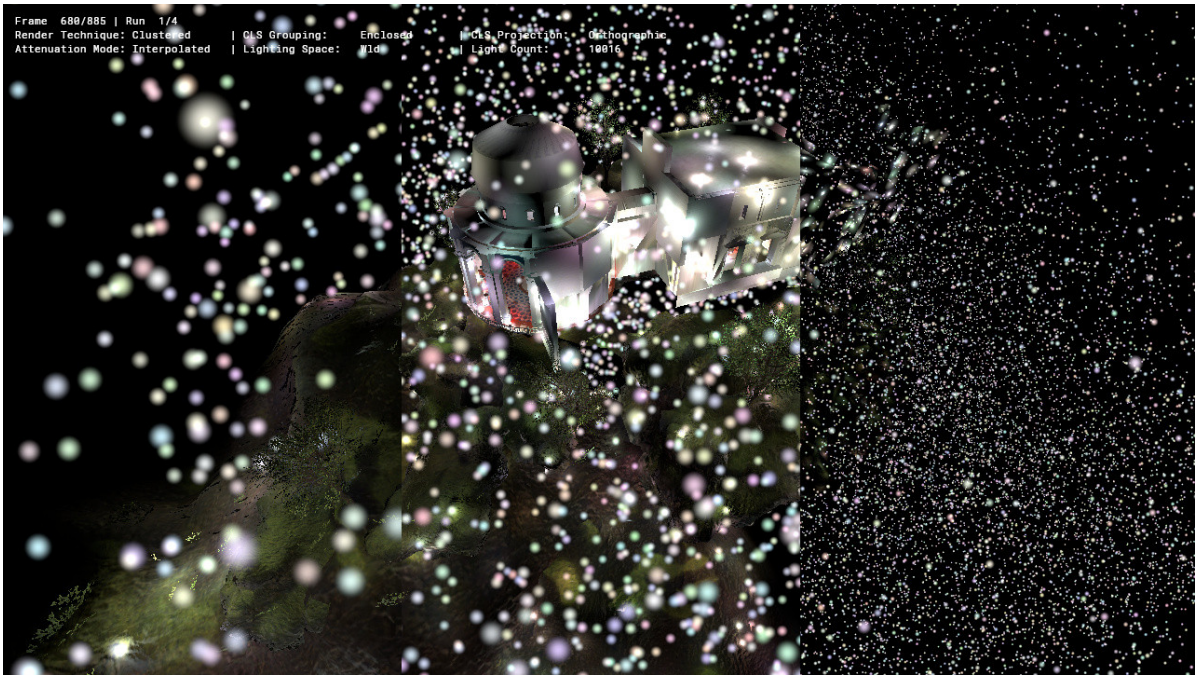


Figure C.2: *Frame 680 of the Sun Temple scene replay. From left to right: 1000, 10000 and 100000 lights.*

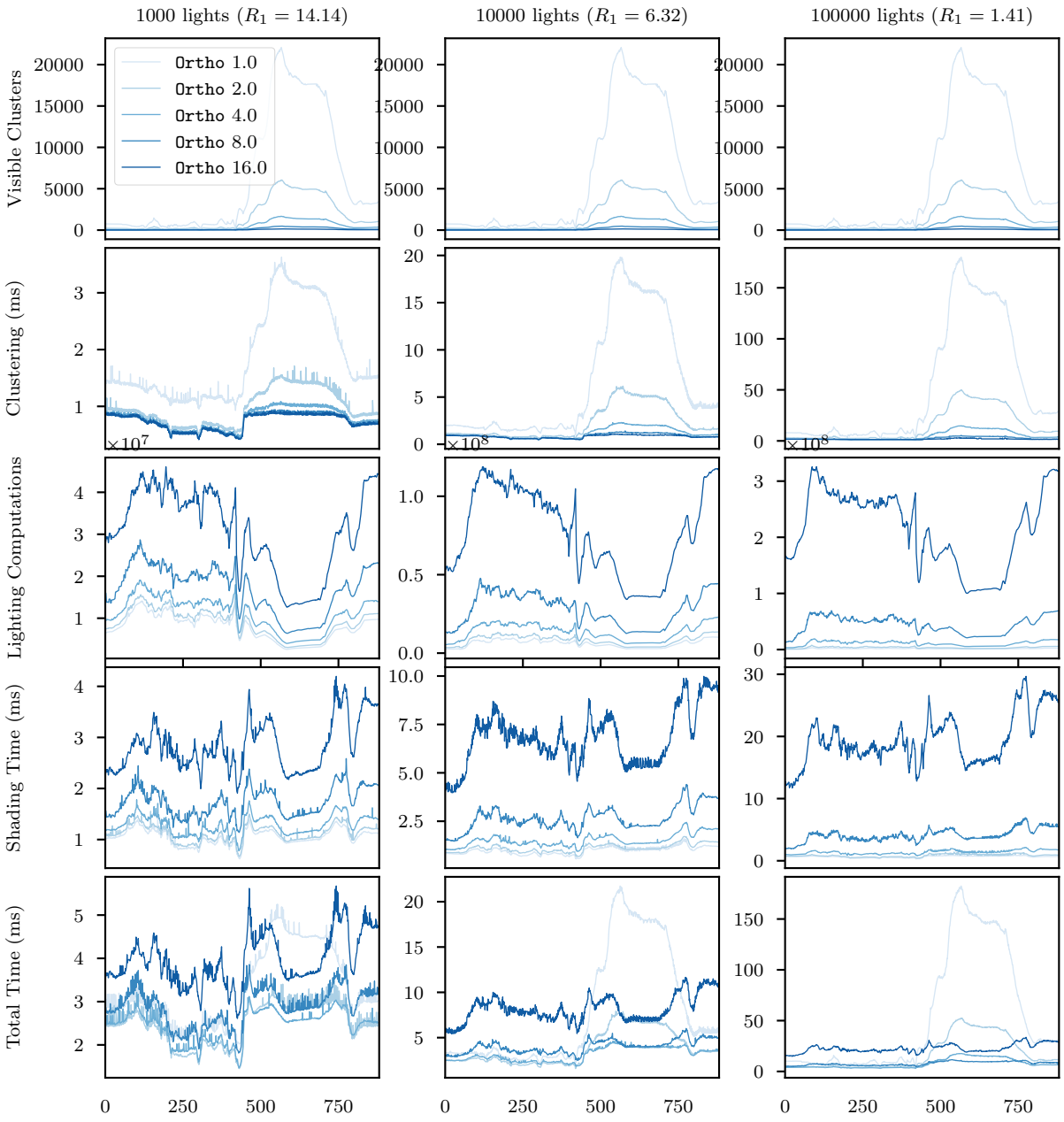


Figure C.3: *Orthographic clustering, Sun Temple scene.*

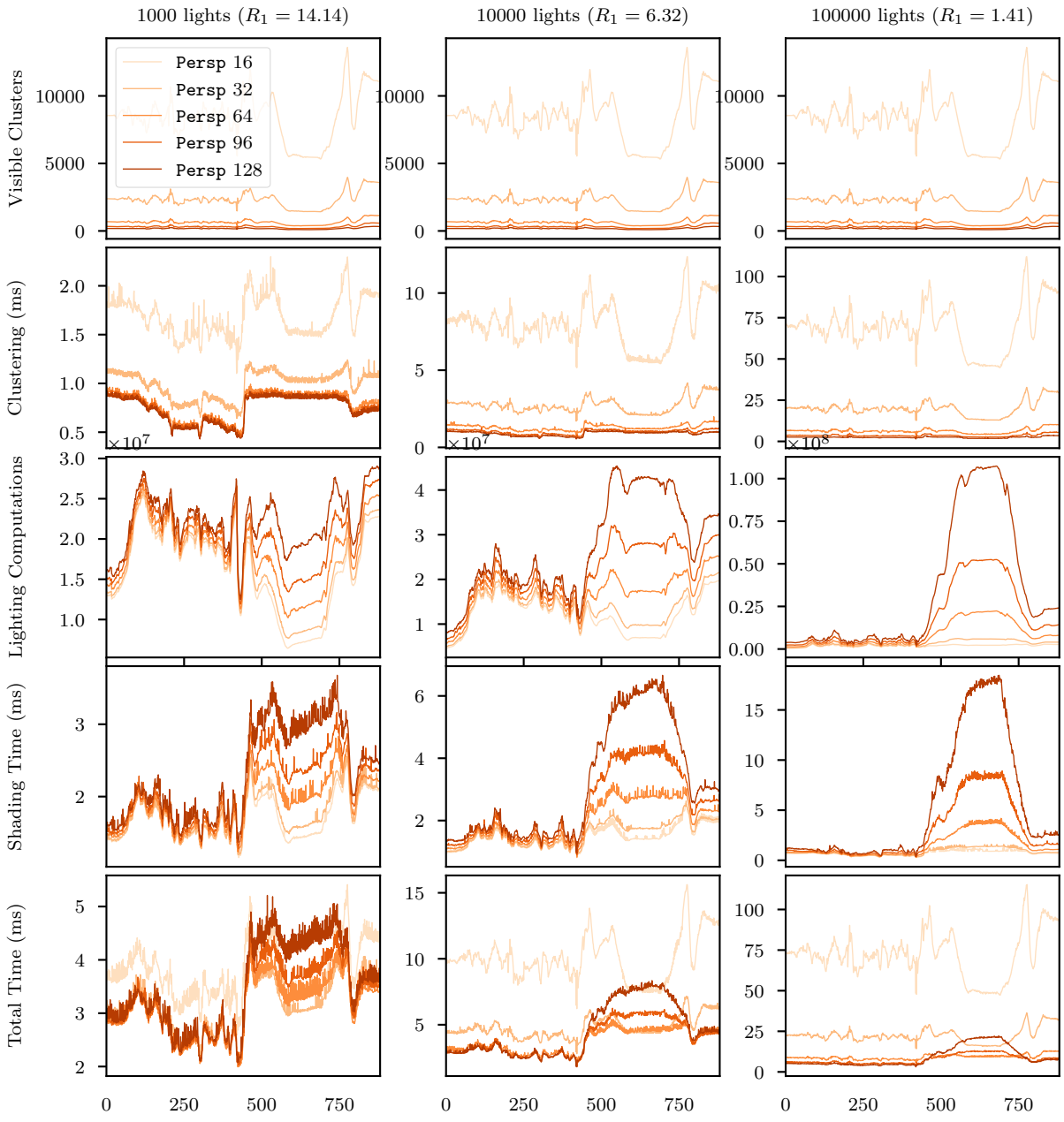


Figure C.4: *Perspective clustering, Sun Temple scene.*

D

Prefix Sum

The prefix-sum [Ble89] is a useful tool in computer science that can help construct a compact representation of nested data with a dynamic size. I discuss the need for such a representation in sections D.1. The basic prefix-sum operation is described in Section D.2 and Section D.3 covers the paralellized computation.

D.1 Motivation

Let's say you are writing some cards to your friends. There is this one friend which you have not spoken to in a long time. For his card, you have more text to write than there is space on the card. You will probably try to compress your message or maybe send multiple or a bigger card. Having an unknown number of lines, or more abstractly *items*, is a common problem in computer science. One way to tackle it is by allocating *probably enough* space, which is what the card manufacturer did. Determining what is probably enough is only guesswork, a safe choice is likely to reserve a lot of unused space.

The common way of dealing with this is to have what is called a *dynamic memory allocator* which reserves large chunks of the available memory and intelligently hands out smaller pieces to applications based on their current need. This allocator also needs to be able to reclaim pieces of memory that are no longer needed. The repeated acquisition and releasing of memory can lead to fragmentation and designing an efficient generic allocator is hard [BZM02; Mic04] .

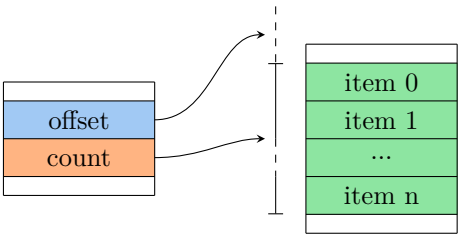


Figure D.1: *A dynamically sized array. The left depicts the array's meta-data. The right shows the items in the array which are laid out contiguously in memory.*

A dynamically sized array needs to keep track of where its items are located and how many there are (see Figure D.1). Instead of allocating exactly the number of items required, implementations

usually over-allocate slightly. This helps to reduce the number of re-allocations which usually involves copying all current data to a different location in memory. These implementations have a third variable called the *capacity* which defines the maximum number of items that could fit in the memory that was allocated.

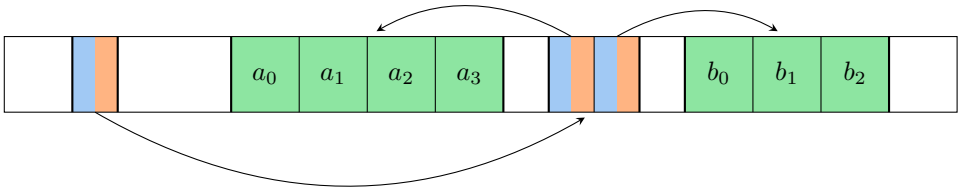


Figure D.2: *Possible memory layout of a ragged array.*

We can now represent our data with an array of arrays, a so called *ragged array*, as shown in Figure D.2. While this is conceptually simple, it is not always an appropriate choice. To read a specific item from the nested array, we need to first look up the address of the nested array, then the address of the item and then the item. The indirection causes an extra memory read, which are slow compared to computations. Processors try to hide the latency of memory reads but having multiple loads in succession makes that difficult. Additionally, the items of the nested arrays can be anywhere in memory, leading to bad data locality when iterating over all the nested arrays in sequence. Bad data locality means that techniques like pre-fetching and caching are less effective.

D.2 Stream compaction

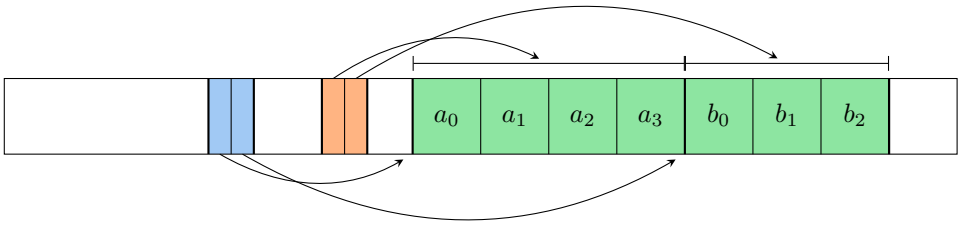


Figure D.3: *Compact layout. The array meta-data for the offsets and counts are omitted.*

We can rearrange the data from Figure D.2 to be more compact as shown in Figure D.3. To build the data in this way, we first have to determine count_i for all nested arrays i. Then, the offset offset_i of nested array i is equal to the total number of items in the nested arrays preceding it.

$$\text{offset}_i = \sum_{j=0}^{i-1} \text{count}_j$$

Computing the offsets for all i can be done efficiently by realizing that we can re-use the previous offset (Equation D.1) as illustrated in Figure D.4.

$$\text{offset}_i = \text{offset}_{i-1} + \text{count}_{i-1} \tag{D.1}$$

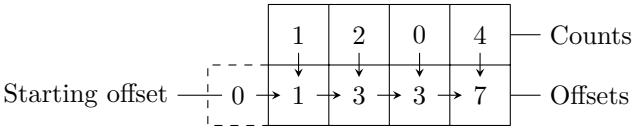
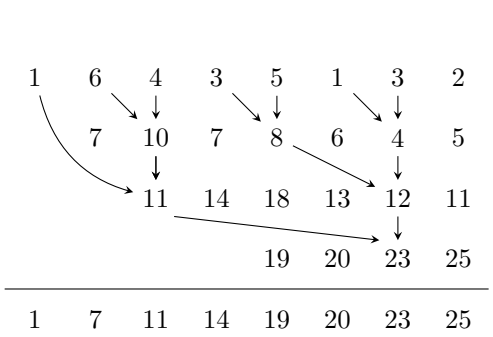


Figure D.4: *Sequential prefix-sum*

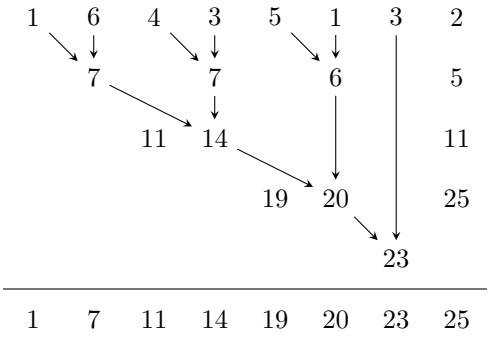
To actually allocate and write the data in the desired manner, we need to first determine count_i for all i. Then we can compute the offsets offset_i. With the offsets available, we iterate over our input again. This time we know we should write each item in nested array i at offset $\text{offset}_i + \text{current\_count}_i$. At the cost of a second pass over our input to compute the counts, we have a compact data layout.

D.3 Parallel prefix-sum

To create ragged arrays on the GPU, we could use a generic memory allocator. Using a generic memory allocator on the GPU does not make much sense if we know exactly what we are going to do. In our case, we know that we can use the prefix-sum to calculate the offsets of each nested array. To compute a prefix-sum on the GPU efficiently, we must try to utilize its many processors in parallel.



(a) *Reduction efficient parallel prefix-sum. The total number of additions is 17 and they happen in 3 reduction steps.*



(b) *Work efficient parallel prefix-sum. The total number of additions is 11 and they happen in 4 reduction steps.*

Figure D.5: *Work efficient and reduction efficient parallel prefix-sum. The arrows indicate the sources for the additions involved in the computation of the 7th value (23).*

Given as many processors as there are elements to process, we can optimize the computation in two ways. We can minimize the number of addition operations that are performed. Alternatively we can minimize the number of reduction steps that need to be taken. Both methods are shown in Figure D.5. Having fewer reduction steps is more important to us than minimizing the work, so we will use the reduction efficient variant.

We have a fixed number of processors over which we would like to distribute the work. We divide the elements over the processors and have them compute a local prefix sum giving the local offsets. We then sum the local offsets to obtain global offsets. Finally, we compute a local prefix sum again but this time we have the global offsets available. This method is illustrated in Figure D.6.

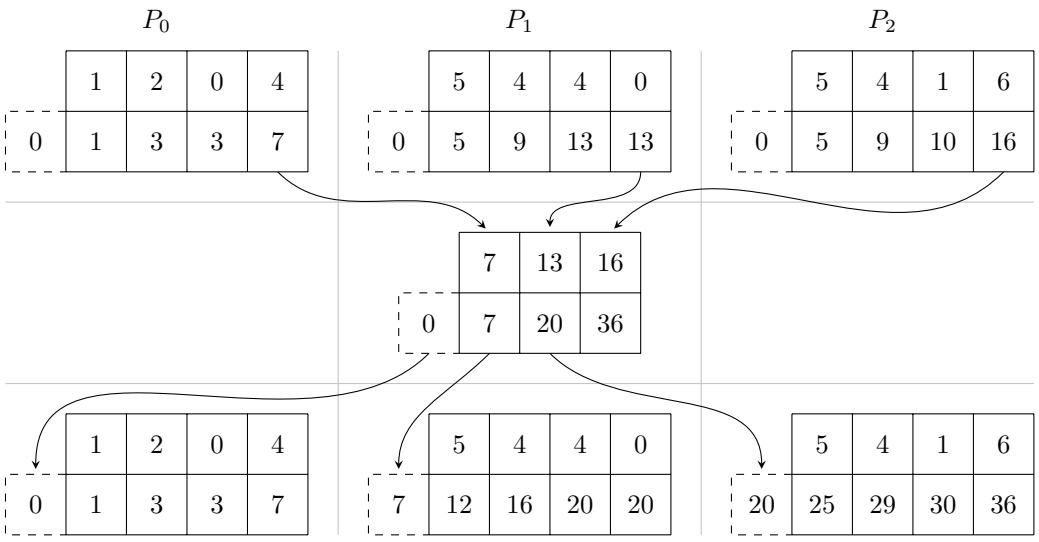


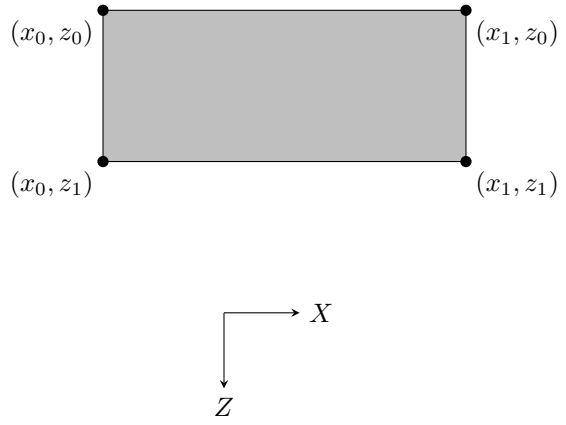
Figure D.6: *Parallel prefix-sum using (left to right) three processors P_0, P_1 and P_2. The three passes (top to bottom), are implemented in terms of sequential prefix-sums. The first two passes compute the* starting offset *(Figure D.4) for the third pass. The number of starting offsets computed in the second pass is related to the number of available processors and therefore usually relatively small. The first and last block need only be computed once but this is not shown because we do not utilize this optimization on the GPU.*

E

Orthographic and Perspective Projection

Figure E.1: *Ortographic and perspective projection frusta in 3D.*

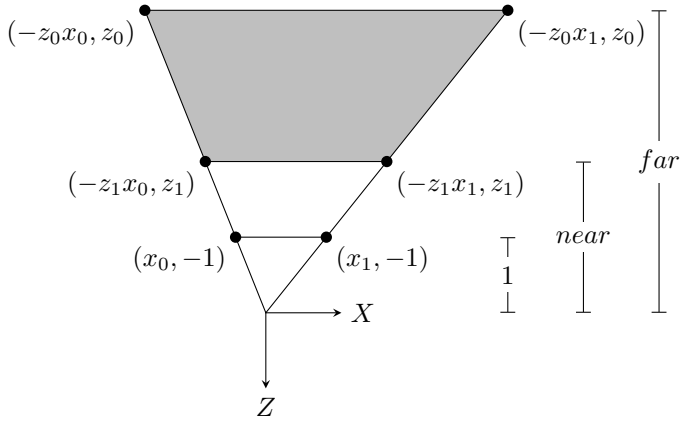
In this thesis I am interested in *orthographic* and *perspective* camera frusta. This section describes both frusta and describes the definitions of their parameters used throughout this thesis. Figures E.2 and E.3 illustrate the orthographic and perspective projection frusta shown from above. When necessary I use $f_{x,0}$ to refer to x_0 of f and similar for the other parameters.



$$z_0 = -far \qquad z_1 = -near$$
$$x_0 = -left \qquad x_1 = right$$
$$y_0 = -bottom \qquad y_1 = top$$

$$f = \begin{bmatrix} x_0 & x_1 \\ y_0 & y_1 \\ z_0 & z_1 \end{bmatrix}$$

Figure E.2: *The orthographic projection camera frustum is actually an axis-aligned box. I therefore use the same definition for axis-aligned boxes throughout this thesis.*



$$z_0 = -far \qquad z_1 = -near$$
$$x_0 = -\frac{left}{near} \qquad x_1 = \frac{right}{near}$$
$$y_0 = -\frac{bottom}{near} \qquad y_1 = \frac{top}{near}$$

$$f = \begin{bmatrix} x_0 & x_1 \\ y_0 & y_1 \\ z_0 & z_1 \end{bmatrix}$$

Figure E.3: *The perspective projection camera frustum is a frustum with a near and far plane orthogonal to the Z-axis, a left and right side plane containing the Y-axis, and a bottom and top side plane containing the X-axis. I define a camera frustum in terms of the signed tangents of the side planes and a near and far plane. I find having the tangents available directly convenient.*

E.0.1 Projection Matrices

In my experience, the derivation of the projection matrices is usually unnecessarily hard. The above definitions lead to a very symmetric and simple derivation, which is why I have included them. I do not explain projection here as it is not essential to understanding the work in this thesis, but in essence we map each point in the frustum f to an axis-aligned box r. When projecting to normalized device coordinates for OpenGL, we would use the range $r_i = (-1, 1)$. Given frustum f and the range r, we calculate the linear interpolation coefficients a_i and b_i for $i \in \{x, y, z\}$ with Equation E.1 where $\Delta r_i = r_{i,1} - r_{i,0}$ and analogous for Δf_i.

$$a_i = \frac{\Delta r_i}{\Delta f_i} \qquad b_i = \frac{r_{i,0} \cdot f_{i,1} - r_{i,1} \cdot f_{i,0}}{\Delta f_i} \tag{E.1}$$

For the perspective projection, the z-component is computed by linearly interpolating $\frac{1}{-z}$ from $\frac{1}{-z_0}$ to $\frac{1}{-z_1}$ instead. Through simplification we arrive at Equation E.2. for these coefficients.

$$a'_z = \frac{\Delta r_z \cdot f_{z,0} \cdot f_{z,1}}{\Delta f_z} \qquad b'_z = \frac{r_{z,1} \cdot f_{z,1} - r_{z,0} \cdot f_{z,0}}{\Delta f_z} \tag{E.2}$$

Using the computed interpolation coefficients, we can easily define the orthographic and perspective projection matrices as shown in Equation E.3. These matrices expect homogeneous coordinates.

$$P_{\text{ortho}} = \begin{bmatrix} a_x & & & b_x \\ & a_y & & b_y \\ & & a_z & b_z \\ & & & 1 \end{bmatrix} \qquad P_{\text{persp}} = \begin{bmatrix} a_x & & -b_x & \\ & a_y & -b_y & \\ & & -b'_z & a'_z \\ & & -1 & \end{bmatrix} \tag{E.3}$$

Bibliography

[Ble89] G. E. Blelloch. “Scans as Primitive Parallel Operations”. In: *IEEE Trans. Comput.* 38.11 (Nov. 1989), pp. 1526–1538. ISSN: 0018-9340. DOI: 10.1109/12.42122.

[ST90] Takafumi Saito and Tokiichiro Takahashi. “Comprehensible Rendering of 3-D Shapes”. In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’90. Dallas, TX, USA: Association for Computing Machinery, 1990, pp. 197–206. ISBN: 0897913442. DOI: 10.1145/97879.97901.

[Sat93] Richard M Satava. “Virtual reality surgical simulator”. In: *Surgical endoscopy* 7.3 (1993), pp. 203–205.

[Kel97] Alexander Keller. “Instant Radiosity”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56. ISBN: 0897918967. DOI: 10.1145/258734.258769.

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. “Reconsidering Custom Memory Allocation”. In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’02. Seattle, Washington, USA: Association for Computing Machinery, 2002, pp. 1–12. ISBN: 1581134711. DOI: 10.1145/582419.582421.

[Mic04] Maged M. Michael. “Scalable Lock-Free Dynamic Memory Allocation”. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. PLDI ’04. Washington DC, USA: Association for Computing Machinery, 2004, pp. 35–46. ISBN: 1581138075. DOI: 10.1145/996841.996848.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. “Light space perspective shadow maps”. In: *Rendering Techniques* 2004 (2004), 15th.

[Woj+04] Rafal Wojciechowski et al. “Building virtual and augmented reality museum exhibitions”. In: *Proceedings of the ninth international conference on 3D Web technology*. 2004, pp. 135–144.

[DS06] Carsten Dachsbacher and Marc Stamminger. “Splatting Indirect Illumination”. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D ’06. Redwood City, California: Association for Computing Machinery, 2006, pp. 93–100. ISBN: 159593295X. DOI: 10.1145/1111411.1111428.

[Llo+06] D. Brandon Lloyd et al. “Warping and partitioning for low error shadow maps”. In: *Proceedings of the Eurographics Workshop/Symposium on Rendering, EGSR*. Ed. by Tomas Akenine-Möller and Wolfgang Heidrich. Nikosia, Cyprus: Eurographics Association, June 2006, pp. 215–226.

[Lai+07] Samuli Laine et al. “Incremental Instant Radiosity for Real-Time Indirect Illumination”. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR’07. Grenoble, France: Eurographics Association, 2007, pp. 277–286. ISBN: 9783905673524.

[Neh+07] Diego Nehab et al. “Accelerating Real-Time Shading with Reverse Reprojection Caching”. In: *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. GH ’07. San Diego, California: Eurographics Association, 2007, pp. 25–35. ISBN: 9781595936257.

[Swo09] Matt Swoboda. “Deferred lighting and post processing on playstation 3”. In: *Game Developer Conference*. 2009.

[GL10] Kirill Garanzha and Charles Loop. “Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing”. In: *Comput. Graph. Forum* 29 (May 2010), pp. 289–298. DOI: 10.1111/j.1467-8659.2009.01598.x.

[OA11] Ola Olsson and Ulf Assarsson. “Tiled shading”. In: *Journal of Graphics, GPU, and Game Tools* 15.4 (2011), pp. 235–251.

[Rag+11] Jonathan Ragan-Kelley et al. “Decoupled Sampling for Graphics Pipelines”. In: *ACM Trans. Graph.* 30.3 (2011). ISSN: 0730-0301. DOI: 10.1145/1966394.1966396.

[HMY12] Takahiro Harada, Jay McKee, and Jason C. Yang. “Forward+: Bringing Deferred Lighting to the Next Level”. In: *Eurographics 2012 - Short Papers*. Ed. by Carlos Andujar and Enrico Puppo. The Eurographics Association, 2012. DOI: 10.2312/conf/EG2012/short/005-008.

[LD12] Gábor Liktor and Carsten Dachsbacher. “Decoupled Deferred Shading for Hardware Rasterization”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’12. Costa Mesa, California: Association for Computing Machinery, 2012, pp. 143–150. ISBN: 9781450311946. DOI: 10.1145/2159616.2159640.

[MM12] Michael Mara and Morgan McGuire. “2D polyhedral bounds of a clipped, perspective-projected 3D sphere”. In: *JCGT. in submission* 5 (2012).

[OBA12] Ola Olsson, Markus Billeter, and Ulf Assarsson. “Clustered Deferred and Forward Shading”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG’12. Paris, France: Eurographics Association, 2012, pp. 87–96. ISBN: 9783905674415.

[Vai+12] Karthik Vaidyanathan et al. “Adaptive Image Space Shading for Motion and Defocus Blur”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG’12. Paris, France: Eurographics Association, 2012, pp. 13–21. ISBN: 9783905674415.

[Per13] Emil Persson. “Practical clustered shading”. In: *SIGGRAPH Course: Advances in Real-Time Rendering in Games* (2013).

[Dac+14] Carsten Dachsbacher et al. “Scalable Realistic Rendering with Many-Light Methods”. In: *Comput. Graph. Forum* 33.1 (Feb. 2014), pp. 88–104. ISSN: 0167-7055. DOI: 10.1111/cgf.12256.

[Ört15] Kevin Örtegren. *Clustered Shading: Assigning arbitrarily shaped convex light volumes using conservative rasterization*. 2015.

[HKL16] Peter Hedman, Tero Karras, and Jaakko Lehtinen. “Sequential Monte Carlo Instant Radiosity”. In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’16. Redmond, Washington: Association for Computing Machinery, 2016, pp. 121–128. ISBN: 9781450340434. DOI: 10.1145/2856400.2856406.

[Lau+16] Gilles Laurent et al. “Forward Light Cuts: A Scalable Approach to Real-Time Global Illumination”. In: *Computer Graphics Forum* 35.4 (2016), pp. 79–88. DOI: 10.1111/cgf.12951.

[SG16] Tiago Sousa and Jean Geffroy. *The devil is in the details: idTech 666*. 2016. URL: https://advances.realtimerendering.com/s2016/Siggraph2016_idTech6.pdf.

[Dro17] Michal Drobot. *Improved Culling for Tiled and Clustered Rendering*. 2017. URL: https://advances.realtimerendering.com/s2017/2017_Sig_Improved_Culling_final.pdf.

[Gam17] Epic Games. *Unreal Engine Sun Temple, Open Research Content Archive (ORCA)*. Oct. 2017. URL: http://developer.nvidia.com/orca/epic-games-sun-temple.

[Lum17] Amazon Lumberyard. *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)*. July 2017. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro.

[Wro17] Bart Wronski. *Cull that cone! Improved cone/spotlight visibility tests for tiled and clustered lighting*. 2017. URL: https://bartwronski.com/2017/04/13/cull-that-cone/.

[Kol+19] Timothy R. Kol et al. “MegaViews: Scalable Many-View Rendering with Concurrent Scene-View Hierarchy Traversal”. In: *Computer Graphics Forum* 38.1 (2019), pp. 235–247.

[NVi19] NVidia. *Antialiased Deferred Rendering*. 2019. URL: https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/d3d_samples/antialiaseddeferredrendering.htm.

[RRS19] Enrique Rosales, Jafet Rodriguez, and ALLA SHEFFER. “SurfaceBrush: From Virtual Reality Drawings to Manifold Surfaces”. In: *ACM Trans. Graph.* 38.4 (July 2019). ISSN: 0730-0301. DOI: 10.1145/3306346.3322970.

[Scr20] Scratchapixel. *Learn Computer Graphics From Scratch!* 2020. URL: https://www.scratchapixel.com/.

[Vri20] Joey de Vries. *Learn OpenGL: Physically-Based Rendering*. 2020. URL: https://learnopengl.com/PBR/Theory.