UTRECHT UNIVERSITY

MASTER THESIS

Volume Tiled Forward Shading

Author:
Jeremiah VAN OOSTEN

Supervisor:
Dr. Jacco BIKKER

*A thesis submitted in fulfillment of the requirements
for the degree of Master of Science*

in the

Game and Media Technology
Department of Information and Computing Sciences

July 18, 2017

Declaration of Authorship

I, Jeremiah VAN OOSTEN, declare that this thesis titled, "Volume Tiled Forward Shading" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ________________________________

Date: ________________________________

"Perfection is not attainable, but if we chase perfection we can catch excellence."

Vince Lombardi

UTRECHT UNIVERSITY

Abstract

Faculty of Science
Department of Information and Computing Sciences

Master of Science

Volume Tiled Forward Shading

by Jeremiah VAN OOSTEN

In order to achieve a greater degree of visual fidelity, graphics programmers are constantly seeking methods to push the limits of what is possible in graphics hardware. In this thesis, we describe Volume Tiled Forward Shading, a new lighting technique based on Tiled Forward and Clustered Forward Shading from Ola Olsson et. al. By constructing a Bounding Volume Hierarchy over the lights in the scene, we are able to demonstrate that a scene containing millions of light sources can be rendered in real-time using Volume Tiled Forward Shading. Volume Tiled Forward Shading proves to be a viable technique to achieve real-time frame rates in scenes containing many light sources.

Acknowledgements

I would like to thank my thesis supervisor Dr. Jacco Bikker for guiding me through the process of writing this document. His own pursuit of perfection through graphics programming is the inspiration of many.

I would also like to thank Bert Heesakkers for supporting me when I encountered a road block or when I just needed someone to bounce ideas off of. His patience and feedback was crucial to the success of this project.

I would also like to thank my girlfriend Suzanne Dingemans and our three boys Lucas, Thomas and Daniel for their patience and understanding over the last few years when, at times, I needed to prioritize this master project over them.

I would also like to recognize my team coordinator Robert Grigg for providing a balance between my work and master's schedule.

I would also like to thank the NVIDIA Corporation for providing the GPU resources that made creating the experiment for this thesis possible.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AABB	**A**xis-**A**ligned **B**ounding **B**ox
API	**A**pplication **P**rogramming **I**nterface
ARB	**A**rchitecture **R**eview **B**oard
BVH	**B**ounding **V**olume **H**ierarchy
CPU	**C**entral **P**rocessing **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DSV	**D**epth-**S**tencil **V**iew
DX	**D**irect**X**
FXAA	**F**ast **A**pproximate **A**nti-**A**liasing
GPU	**G**raphics **P**rocessing **U**nit
LSD	**L**east **S**ignificant **D**igit
MSAA	**M**ulti-**S**ample **A**nti-**A**liasing
MSD	**M**ost **S**ignificant **D**igit
NDC	**N**ormalized **D**evice **C**oordinates
OGL	**O**pen **G**raphics **L**ibrary
SDK	**S**oftware **D**evelopment **K**it
RTV	**R**ender **T**arget **V**iew
SM	**S**treaming **M**ultiprocessor
SRV	**S**hader **R**esource **V**iew
UAV	**U**niform **A**ccess **V**iew

Dedicated to my children Lucas, Thomas, and Daniel.
May your lives be a continuous journey of learning.

Chapter 1

Introduction

1.1 Motivation

In the eternal pursuit of achieving photo-realistic computer generated images, graphics programmers are continually trying to find improved rendering techniques to produce realistic images at "real-time" frame rates. Since the advent of the first viable commercial hardware-accelerated 3D graphics adapters in 1996 (Singer, 2013) there have been many advances in rendering techniques that improve upon previous techniques in one or more ways.

The first graphics adapters that supported the OpenGL 1.0 API featured a fixed-function rendering pipeline that performed per-vertex lighting calculations in the *Texturing and Lighting* stage. The lighting equation that was used to compute the vertex color was specified in the specification document (Segal and Akeley, 1994) and could not be modified by the graphics programmer. The OpenGL specification required that vendors supported a minimum of eight hardware lights but the number of lights that could actually be enabled in the scene was dependent on the scene complexity and the performance of the GPU hardware.

The common term used to refer to the rendering technique described in the OpenGL 1.0 API specification is *Forward Rendering*. *Forward Rendering* is the process of pushing geometry *forward* through the rendering pipeline and applying the same stages of the rendering pipeline to each geometric object in order to produce the final image. For each object pushed through the rendering pipeline, the *Forward Rendering* technique applies the lighting equation for each active light in the scene regardless of the lights contribution to the final color of the pixel being rendered. Without the ability to programmatically modify the *Texturing and Lighting* stage of the fixed-function pipeline, it was difficult to further optimize this technique.

Programmable vertex shaders were not available until November 2000 when Microsoft released the first vertex shader profile (vs_1_1) together with the DirectX 8.0 SDK (Oosten, 2014). Although the number of shader instructions was limited to 128, it was the first time the graphics programmer could bypass the *Texturing and Lighting* stage of the fixed-function pipeline and implement their own lighting functions.

Two years after the release of DirectX 8.0, the DirectX 9.0 SDK was released. In addition to programmable vertex shaders, DirectX 9.0 introduced the programmable pixel shader profile (ps_2_0). That same year the vertex and fragment program extensions were added to the list of standard extensions accepted by the OpenGL Architecture Review Board (ARB). The vertex and fragment program extension were added to the OpenGL 2.0 standard on October 22, 2004 (Segal and Akeley, 2004).

With the introduction of programmable shaders, which occurred at the turn of the 20th century, the graphics programmer was free to implement the vertex and pixel shader stages of the rendering pipeline. But even with the power to change the way shading was performed in the rendering pipeline, GPUs were still limited by their computational performance. Programmable pixel shaders (fragment programs in OpenGL) made per-pixel lighting calculations possible. Per-pixel lighting allows for the lighting equation to be applied to every visible pixel. Traditional *Forward Rendering* uses a brute-force approach to compute the shading of a pixel by considering every active light in the scene. Using this technique, the number of active lights in a scene was limited by the performance of the GPU hardware. For this reason, various shading techniques were introduced in an effort to increase lighting complexity than was previously possible using traditional *Forward Rendering*.

One such technique that attempts to increase the number of active lights in the scene is called *Deferred Shading* (Saito and Takahashi, 1990; Geldreich and Pritchard, 2004; Shishkovtsov, 2006; Leeuw, 2007; Mittring, 2009). *Deferred Shading* is a technique that uses several image buffers to store geometric information that is used to compute the lighting information in a later pass. The types of information that are stored in the image buffers are:

- material diffuse color
- material specular color (RBG) and specular power (A)
- surface normals
- screen space depth values

The combination of all of these buffers is referred to as the *Geometric Buffer* or simply the *G-buffer* (Saito and Takahashi, 1990).

In the second pass of *Deferred Shading* all of the dynamic lights in the scene are rendered as geometric objects. Point lights are rendered as spheres, spot lights as cones, and directional lights are rendered as full-screen quads. The depth buffer and stencil buffer are used to discard fragments that are not affected by any light sources.

One of the benefits of *Deferred Shading* compared to *Forward Shading* is that the expensive lighting calculations are only computed for fragments that are influenced by a light source. There is also less overdraw on fragments that are occluded by other geometry in the scene which reduces the amount of redundant lighting calculations that must be performed.

The disadvantage of *Deferred Shading* is that only opaque objects can be rasterized into the G-buffer. Objects with transparent materials can cover (but not occlude) both opaque and other transparent objects, but the multiple layers required to describe transparent objects cannot be represented in the 2D image buffers that compose the G-buffer. Therefore rendering pipelines that use the *Deferred Shading* technique must also rely on an additional *Forward Rendering* pass to render transparent objects.

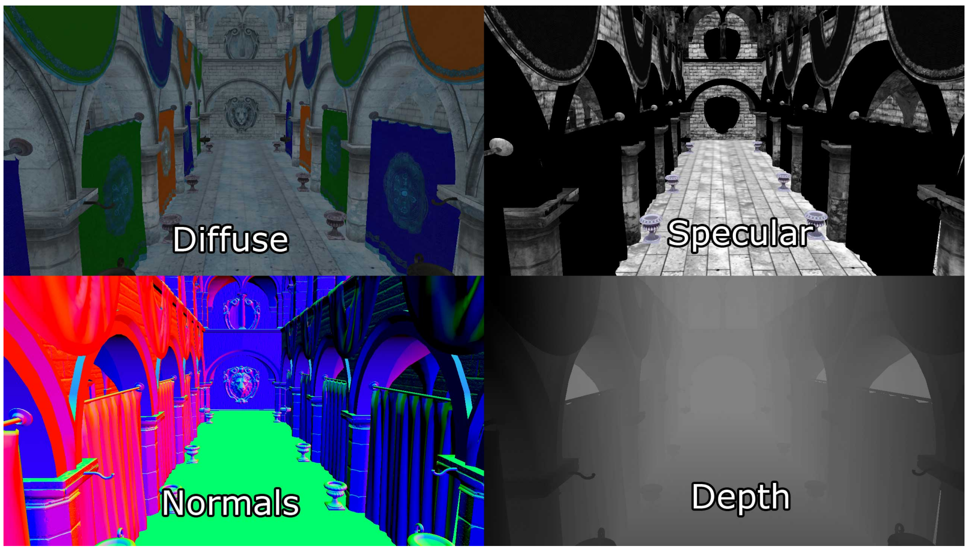

FIGURE 1.1: Deferred shading: Several buffers are used to describe the G-buffer. Diffuse (top-left), specular (top-right), normals (bottom-left), and depth (bottom-right).

Another disadvantage of *Deferred Shading* is that only a single lighting model can be simulated in the lighting pass. Since only a single pixel shader can be bound to the rendering pipeline when the light geometry is rendered, it is not possible to switch lighting models depending on the object that is being lit by the invocation of the pixel shader.

Another disadvantage of *Deferred Shading* is that *Multi-Sample Anit-Aliasing* (MSAA) (Microsoft, 2017) is not supported. MSAA works by invoking the pixel shader several times at various sub-pixel offsets and the final pixel color is determined by blending the results based on the pixel coverage. Since the lighting pass can only sample from a single fragment in the G-buffer, MSAA cannot be used to produce the G-buffer targets. Enabling MSAA for the lighting pass will not produce the correct results since only the geometry representing the light source is being rendered and MSAA will only affect pixels that are contained within the bounds of the light source. If anti-aliasing is required in the deferred shading pipeline, another anti-aliasing technique such as *Fast Approximate Anti-Aliasing* (FXAA) (Lottes, 2009) must be used. FXAA requires an additional post-process pass which may not be feasible in some applications.

Tiled Forward Shading (Olsson and Assarsson, 2011), also known as *Forward+* (Harada, McKee, and Yang, 2012; McKee, 2012), is a rendering technique that divides the screen into a grid of uniform tiles. In an initial pass, the active lights in the scene are sorted into the screen space tiles. In the shading pass, only the lights that are contained in the same screen space tile as the shaded fragment need to be considered in the lighting calculations. This technique can be further optimized by using the depth buffer to compute the minimum and maximum depth bounds within the tile. Any light that is not contained within the frustum formed by the tile edges and the minimum and maximum depth planes can be discarded.

Tiled Forward Shading allows for many more dynamic lights in the scene compared to forward and *Deferred Shading* but it is not without drawbacks.

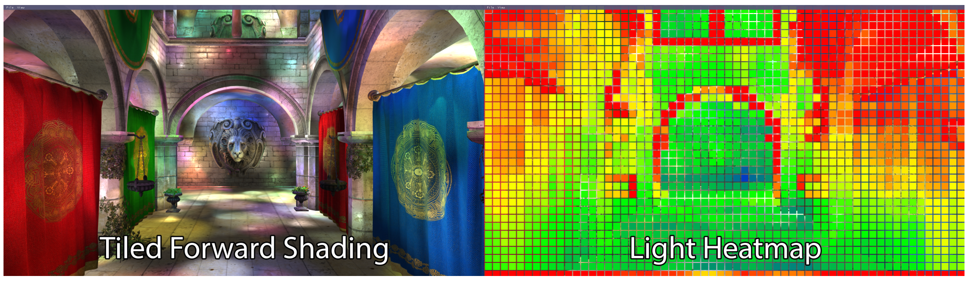

FIGURE 1.2: Tiled Forward Shading: Scene lit with 10,000 dynamic lights (left), light heatmap: blue: 1 - 10, green: 10 - 30, yellow: 30 - 40, orange: 40 - 50, red: 50+ lights (right).

Taking the minimum and maximum depth bounds to form the tile frustum is a reasonable optimization when rendering opaque geometry but cannot be applied when rendering transparent objects. The reason for this is that transparent geometry cannot be rendered during the depth pre-pass and therefore cannot be used to constrain the tile frustum. The solution to this problem is to build two light lists, one that can be used while rendering opaque geometry, and another that is used while rendering transparent geometry.

Another disadvantage of *Tiled Forward Shading* is that tiles which have a large depth disparity (minimum and maximum depth values within the tile are far apart) will result in a large tile frustum and may include lights that do not contribute to the final shading of the fragment (false positives). Harada (Harada, 2012) suggests splitting the tile frustum into multiple sections along the depth of the tile and disregarding lights that only intersect with empty sections. Harada is able to show that this technique was effective at reducing the false positives in the case of tiles with a large depth disparity. However, the added complexity in the light culling stage resulted in a 10% overhead of the entire technique.

The light culling algorithm in *Tiled Forward Shading* uses a frustum that is created from the tile edges and the minimum and maximum depth values in the tile. Frustum culling is inherently inaccurate because it relies on performing plane-intersection tests with the geometry of the light volume. Only light volumes that are fully contained in the negative half-space of the plane can be disregarded. This technique to perform light culling results in many tiles accepting the light when it should be disregarded as can be seen in Figure 1.3.

Figure 1.3 shows the outline of a point light as a red circle. The blue tiles represent tiles that have determined that the light is covering the tile. The tiles that are fully (or partially) contained within the red circle are correctly detecting the light is contained in the tile (true positives). The tiles that are fully outside of the red circle are incorrectly detecting the light is contained within that tile (false positives). This occurs because there is no single plane in the frustum of the tile that can reject the light.

The false positives resulting from the frustum culling technique could possibly be reduced using the Separating Axis Theorem (SAT) (dyn4j, 2017) but this was not explored in the context of this thesis.

In a previous study (Oosten, 2015) it was shown that although the *Tiled*

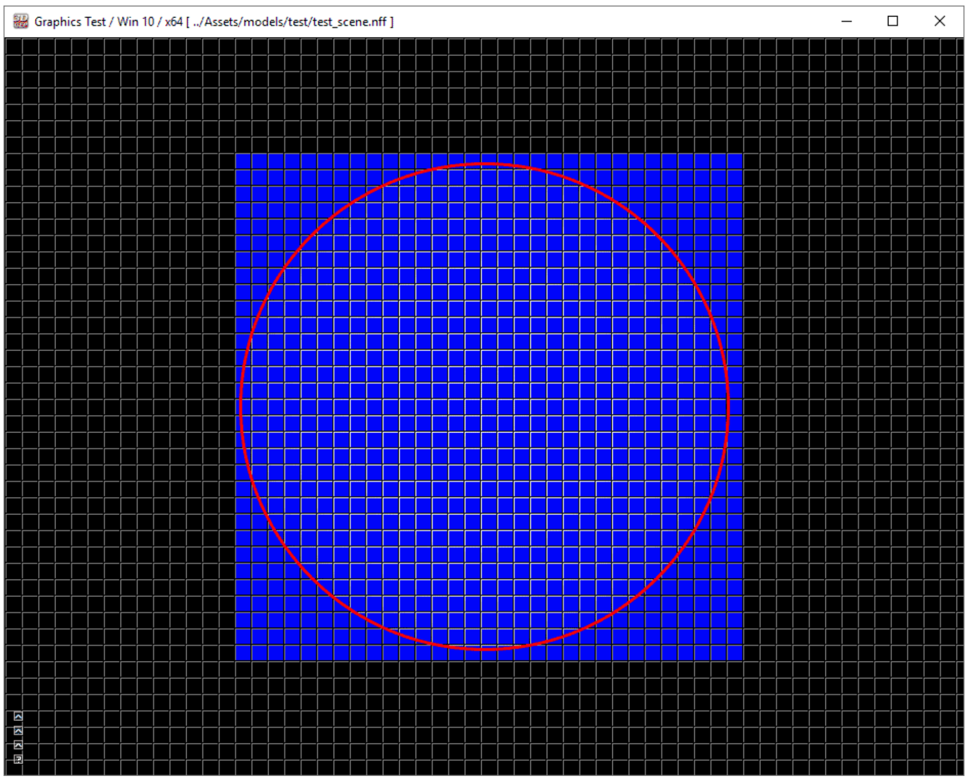
Graphics Test / Win 10 / x64 [../Assets/models/test/test_scene.nff]

FIGURE 1.3: Culling a point-light against the tile frustum results in false positives. The blue tiles outside of the red circle are incorrectly including the point light (Oosten, 2015).

Forward Shading technique outperformed both the *Forward* and *Deferred Shading* techniques, the primary bottleneck of *Tiled Forward Shading* was the light culling stage. That study showed that the light culling stage of the *Tiled Forward Shading* technique had a $\mathcal{O}(n^2)$ runtime complexity which limits the number of lights that could be present in the scene.

In this thesis we introduce *Volume Tiled Forward Shading*. *Volume Tiled Forward Shading* is similar to *Tiled Forward Shading* but instead of using a 2D grid of uniform tiles in screen space, a 3D grid of volume tiles is constructed in view space. A single volume tile covers a uniform size in screen space, but the size of volume tiles further away from the viewer increase logarithmically to maintain self-similar dimensions, that is, volume tiles maintain a cubic shape regardless of their distance from the view plane.

In this thesis we will show that *Volume Tiled Forward Shading* performs better than *Tiled Forward Shading* in the average case. Unlike *Deferred Shading*, *Volume Tiled Forward Shading* has support for multiple lighting models and MSAA is also natively supported. Unlike *Tiled Forward Shading*, *Volume Tiled Forward Shading* supports both opaque and transparent objects with a single light list. Because *Volume Tiled Forward Shading* uses Axis-Aligned Bounding Boxes (AABB) to define a volume tile, the light culling stage of the volume tiled forward shading technique produces less false positives than *Tiled Forward Shading* when using frustum culling.

We also introduce an optimization to the *Volume Tiled Forward Shading* technique that reduces the runtime complexity of the light culling stage to $\mathcal{O}(\log_{32} n)$ allowing for a virtually unlimited number of lights to be active in the scene.

1.2 Research Question

In this thesis, we will attempt to answer the following question:

"Can the performance of Tiled Forward Shading be improved by using Volume Tiled Forward Shading?"

To answer this question, we will present an experiment that demonstrates both *Tiled Forward Shading* and *Volume Tiled Forward Shading*. We will show that on average, *Volume Tiled Forward Shading* performs better than *Tiled Forward Shading*. We will also present a technique that sorts the light sources into a *Bounding Volume Hierarchy* (BVH) before the light culling stage, allowing for millions of light sources to be active in the scene while still providing "real-time" frame rates. In the context of this thesis, anything higher than 30 Frames Per Second (FPS) is considered "real-time".

Chapter 2

Background

2.1 Previous Work

Several rendering techniques have been developed in the past which contribute to the *Volume Tiled Forward Shading* technique described in this thesis. *Forward Rendering* was traditionally one of the most pervasive techniques applied in fixed-function (non-programmable) dedicated GPU hardware. Forward Rendering applies both rasterization and shading in a single pass through the rendering pipeline. *Deferred Shading* is a rendering technique that decouples the rasterization and shading passes to minimize redundant lighting computations. *Tiled Forward Shading* sorts lights into screen space tiles in order to minimize the number of lights that needs to be considered during shading. *Clustered Shading* extends upon the 2D screen space tiles of Tiled Forward Shading and divides the 2D tiles along the view-space depth into 3D clusters.

Forward Rendering is a traditional rendering technique that was generally implemented in fixed-function dedicated graphics processors. Although currently less pervasive on desktop applications, Forward Rendering is still commonly used in mobile applications (Olsson, 2015). Forward Rendering works by rasterizing and shading each geometric object in the scene in a single pass. Each shaded pixel considers the lighting contribution of every light in the scene even if the contribution of that light to the final color of the pixel is negligible.

The steps of the per-pixel *Forward Rendering* technique are shown in Algorithm 2.1.

Algorithm 2.1 Forward Rendering.

```
Require: $G$ is a list of geometric objects.
Require: $L$ is a list of lights.
Require: $S$ is a framebuffer that stores the final rendered image.
 1: function FORWARDRENDERING($G$,$L$)
 2:     for $g$ in $G$ do
 3:         $F \leftarrow$ RASTERIZE($g$)
 4:         for $f$ in $F$ do
 5:             $c \leftarrow 0$
 6:             for $l$ in $L$ do
 7:                 $c \leftarrow c+$ COMPUTELIGHTING($l$, $f$)
 8:             end for
 9:             $S[f] \leftarrow c$
10:         end for
11:     end for
12: end function
```

The *Forward Rendering* pseudo-code shown in Algorithm 2.1 shows an extremely simplified version of how the GPU generates the rendered image. The *Rasterize* function on line 3 generates a list of fragments that must be shaded for a given geometric object. The *ComputeLighting* function on line 7 applies the lighting function for light l and visible fragment f.

It is easy to see that this technique contains a triple-nested loop with run time complexity of $\mathcal{O}(gfl)$ where g is the number of geometric objects that need to be rendered, f is the number of visible fragments, and l is the number of lights that contribute to the final shading of the object. Clearly this is not an ideal technique for performing high-resolution rendering of scenes that contain many geometric objects and many lights. There are several obvious optimizations that can be applied to this technique:

1. Reduce the number of geometric objects
2. Reduce the number of visible fragments
3. Reduce the number of lights

Reducing the number of geometric objects that need to be rendered can be achieved by implementing geometric culling techniques such as back-face culling or camera frustum culling (Clark, 1976). More advanced occlusion culling techniques exist such as z-buffer optimization techniques (Catmull, 1974) and hierarchical occlusion maps (Zhang et al., 1997). Occlusion culling techniques will not be discussed in the context of this thesis.

Reducing the number of visible fragments that need to be shaded is achieved by reducing (or eliminating) *overdraw*. Overdraw occurs when a previously shaded fragment in the color buffer is replaced by a fragment that appears in-front, but is rendered after the previous fragment. Sorting geometric objects based on their distance to the camera can help to reduce overdraw if the objects are sorted from nearest to farthest from the camera. If geometric objects are sorted before rendering then z-buffer optimizations can be applied which help to reduce the number of redundantly shaded fragments.

Reducing the number of lights can be achieved by determining exactly which lights will effect the shaded fragment before shading actually takes place. If an assumption can be made about the contribution of the light based on the distance the light is to the point being shaded then lights that are sufficiently far away can be disregarded during shading. This can be achieved by assigning a maximum range to the light. Lights that are farther away from the point being shaded than the maximum range will not contribute to the final color of the fragment and do not need to be considered during shading.

Deferred Shading is a rendering technique that focuses on reducing both the number of visible fragments that must be shaded and reducing the number of lights that contribute to the final color of the light. Deferred Shading is a two-pass technique that works by first rasterizing each of the geometric objects in the scene into a set of 2D image buffers. These buffers are used to store the geometric information that is required to perform the lighting calculations in a later pass. The image buffers that store the geometric information is commonly referred to as Geometric Buffers (G-Buffers). Each G-buffer contains a single geometric property (such as color, depth, or surface normal) (Saito and Takahashi, 1990). The G-buffers are then used in the shading pass to compute the final lighting contribution.

In the second pass of the Deferred Shading technique, geometric volumes that represent the light sources in the scene are rasterized. Point lights are represented by spheres, spot lights as cones, and directional lights as full-screen quads (Hargreaves and Harris, 2004). Only the visible fragments that are contained within the light volumes are shaded by the light. The depth buffer from the geometric pass is used to determine which fragments are contained in the light volume and the geometric information stored in the G-buffers is used to compute the final shading for the light source.

Tiled Forward Shading is a rendering technique that focuses on reducing the number of lights that must be considered during shading. It achieves this by first assigning the lights in the scene to the cells of a uniform 2D grid that is defined in screen space. During shading, only the lights that are contained within the grid cell of the currenlty shaded fragment must be considered (Olsson and Assarsson, 2011).

Clustered Shading is a rendering technique developed by Ola Olsson, Markus Billeter, and Ulf Assarsson (Olsson, Billeter, and Assarsson, 2012). Clustered Shading extends Tiled Forward Shading into three-dimensional space by dividing the 2D screen-space tiles into 3D clusters. Clustered Shading improves on Tiled Forward Shading by reducing false positives caused by large depth discontinuities in tiles containing geometric boundaries. Clustered Shading uses a 2D image buffer to store the cluster keys which restricts the algorithm to opaque geometry. A method to apply clustered shading to transparent geometry is not described in their paper.

2.1.1 Forward Rendering

Forward Rendering is the traditional rendering technique that is most commonly used in dedicated fixed-function rendering hardware. Forward Rendering works by rendering each geometric object in the scene and shading each visible fragment by computing the lighting contributions of all lights in the scene. At the most basic level, Forward Rendering makes no attempt to eliminate lights in the scene that provide negligible contribution to the final color of the pixels.

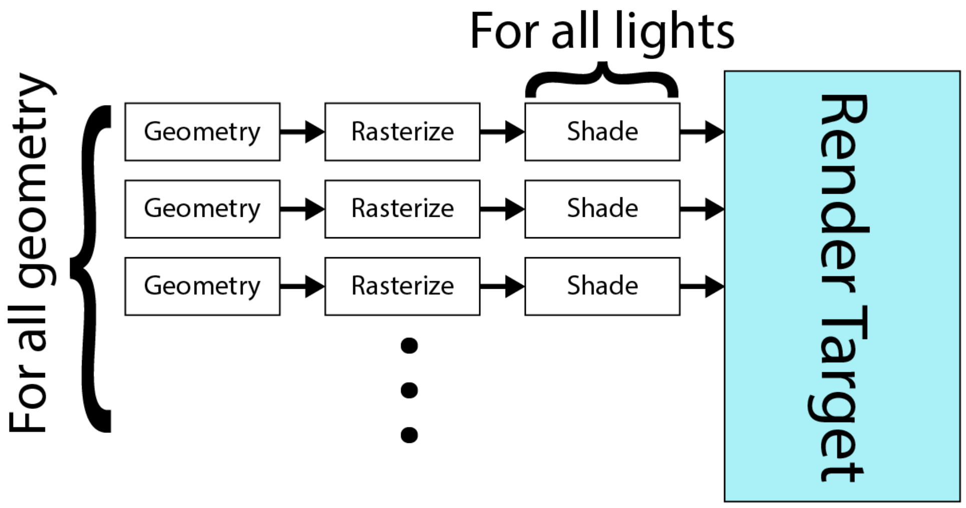

FIGURE 2.1: Forward shading considers all lights to shade every geometric object in the scene.

Although trivial to implement, Forward Rendering is not well suited for performing high-resolution rendering of scenes that contain many geometric objects and many lights. Since the introduction of the programmable shader pipeline, it is possible to implement a wide variety of rendering techniques that perform better than Forward Rendering under certain conditions. Deferred Shading, Tile Forward Shading, and Clustered Shading are a few of the techniques that perform better than traditional Forward Rendering when rendering scenes with many geometric objects and many lights. These techniques will be described in the next sections.

2.1.2 Deferred Shading

Deferred Shading is a rendering technique that decouples geometric rasterization and shading into separate passes (Deering et al., 1988; Hargreaves and Harris, 2004). In the first pass of the Deferred Shading technique, geometric attributes of the objects in the scene are rasterized into several full-screen textures called the *Geometry Buffers* or *G-Buffers* (Saito and Takahashi, 1990). In the second pass, volumes that represent the lights in the scene are rasterized. The geometric attributes are read from the G-Buffers in order to compute the final lighting contribution.

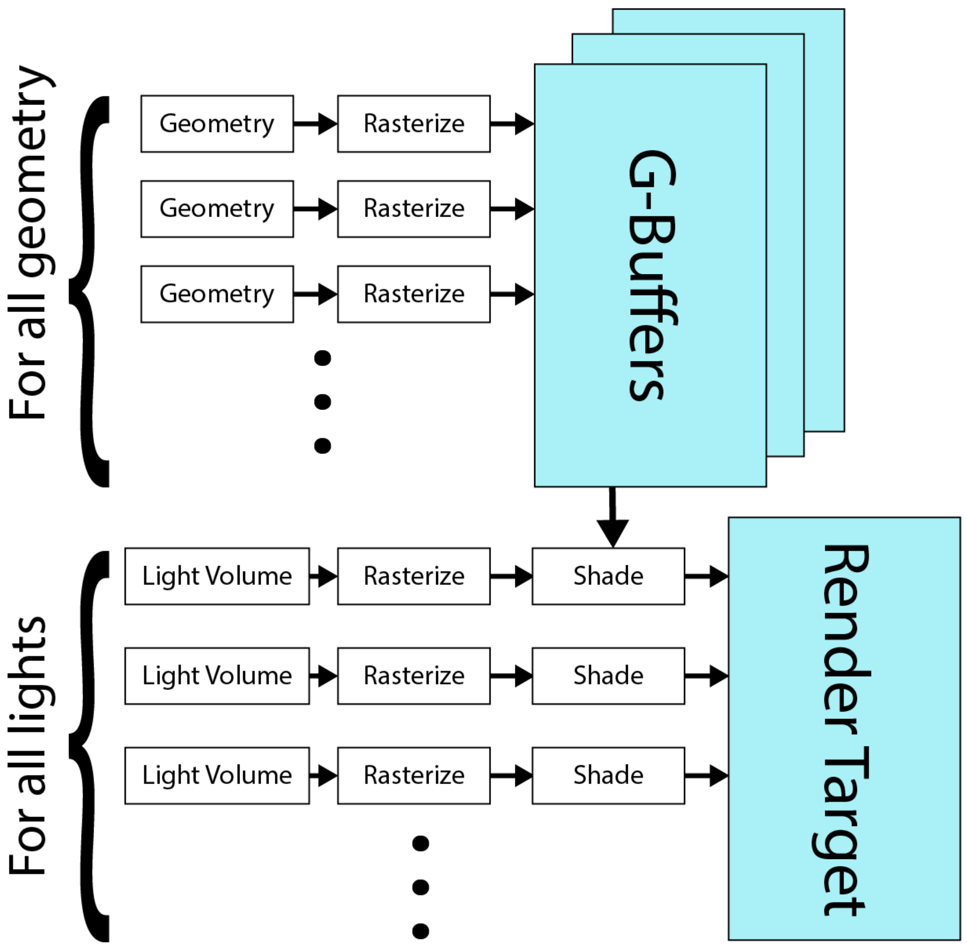

FIGURE 2.2: Deferred shading operates in two passes: Geometry pass and the lighting pass.

Geometry Pass

In the first pass of the Deferred Shading technique the G-buffers are generated. Any type of geometric information can be stored in the G-buffers but the most common attributes to store in the G-buffers are (Hargreaves and Harris, 2004; Shishkovtsov, 2006; Leeuw, 2007):

1. Depth/Stencil
2. Ambient & Emissive (Light Accumulation)
3. Normals
4. Specular
5. Diffuse

The *Depth/Stencil* buffer is most commonly stored as a 32 bit per pixel texture format where 24 bit are used to store the depth and 8 bit are used to store the stencil value. The position of the fragment can be reconstructed in the lighting pass from the depth value and the x and y screen position of the current fragment.

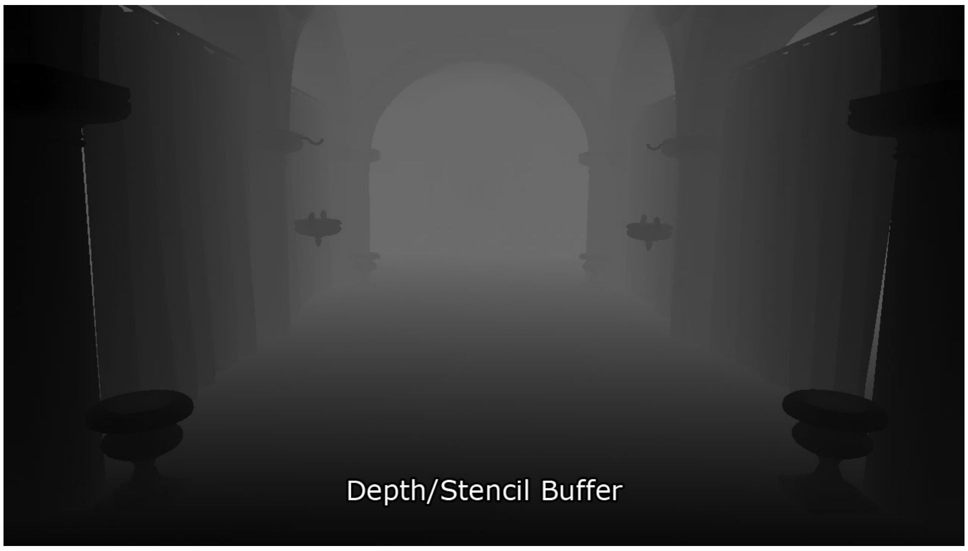

FIGURE 2.3: The output of the depth/stencil buffer (Oosten, 2015).

The *Light Accumulation* buffer stores the ambient and emissive contributions of the geometry. The Light Accumulation buffer is generally stored as a 32 bit per pixel texture where 8 bit are used to represent the red, green, and blue channels. The alpha channel can be used to store either the specular intensity (Hargreaves and Harris, 2004) or the luminance (Leeuw, 2007).

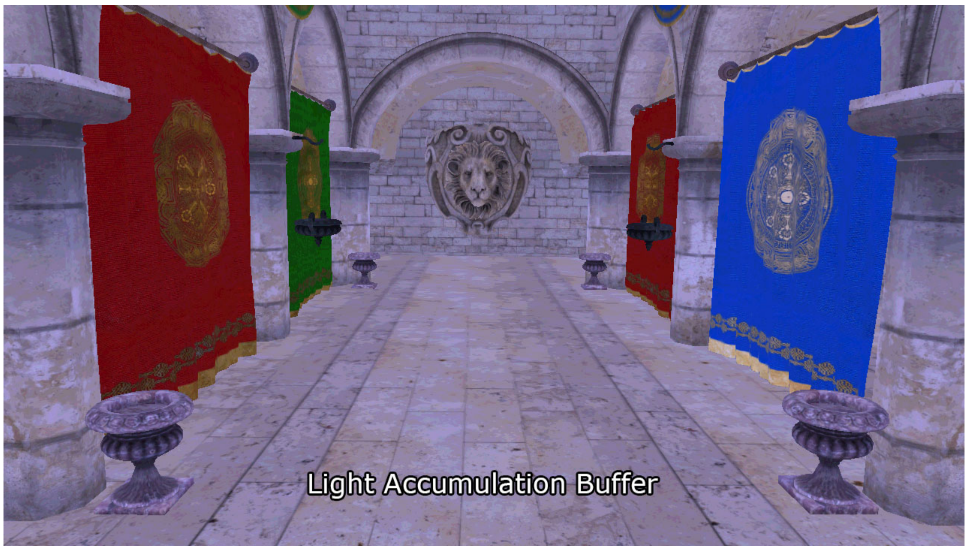

FIGURE 2.4: The output of the light accumulation buffer. The image has been brightened to improve visibility (Oosten, 2015).

The Normal buffer stores the surface normals for the geometry in view space. The normal buffer is usually compressed into a 32 bit where the

x and y components of the view space normal are compressed into 16 bit floating-point values. The z component of the view space normal is recomputed in the lighting pass using Equation 2.1 (Hargreaves and Harris, 2004; Leeuw, 2007).

$$z = \sqrt{1 - x^2 - y^2} \tag{2.1}$$

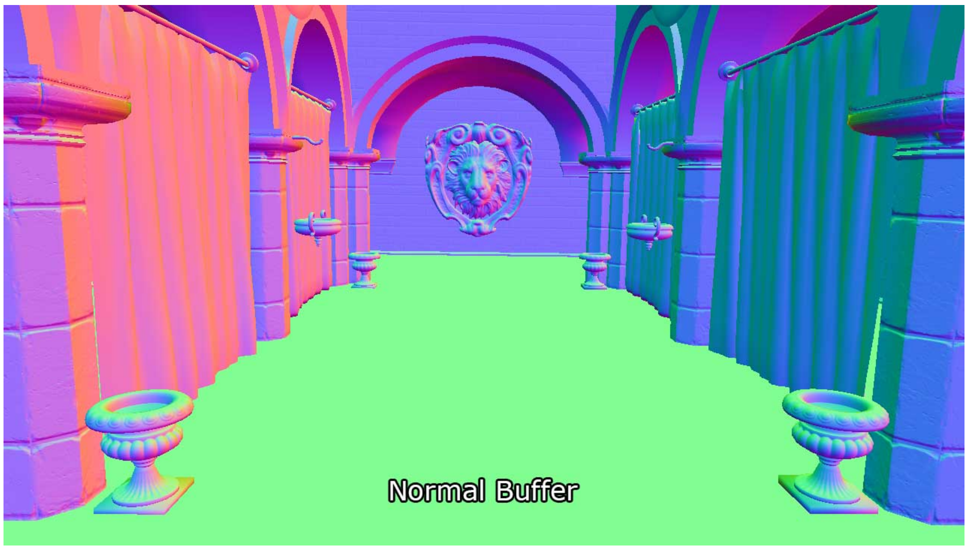

FIGURE 2.5: The Normal buffer stores the view space normal of the geometry. In this image, all three normal components are visualized (Oosten, 2015).

The Specular buffer stores the specular color and specular power. The specular color is stored in the red, green, and blue channels and the specular power is converted in the range $[0 \cdots 1]$ using Equation 2.2 and stored in the alpha channel as an 8 bit unsigned normalized value (Leeuw, 2007).

$$\alpha' = \frac{\log_2(\alpha)}{10.5} \tag{2.2}$$

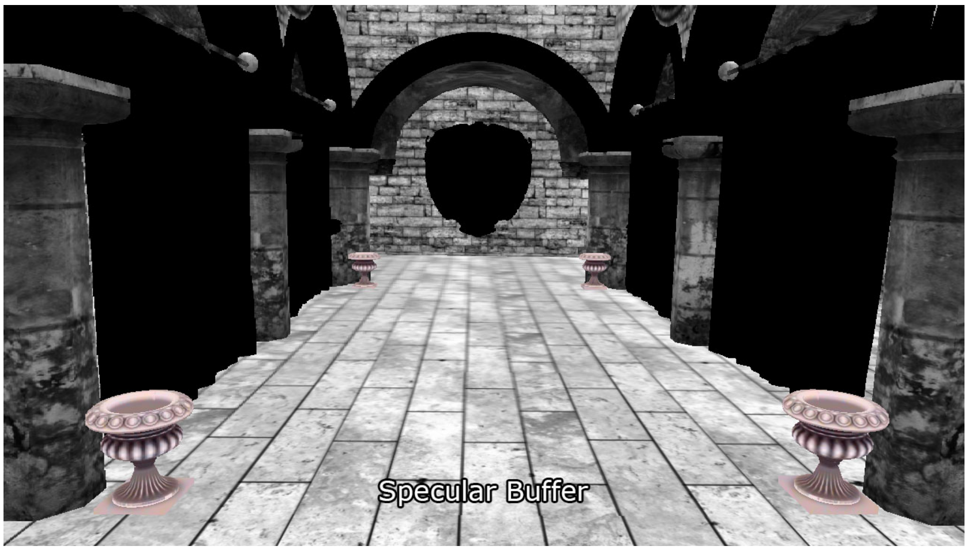

FIGURE 2.6: The speuclar buffer stores the specular color and specular power (Oosten, 2015).

The Diffuse lighting buffer stores the diffuse contribution of the geometry. The diffuse buffer is usually stored as a 32 bit per pixel buffer where each the red, green, and blue channels are 8 bit each. The alpha channel can be used for other purposes. In some cases it is used to store the pre-rendered static sun shadows (Leeuw, 2007) or is left completely unused (Hargreaves and Harris, 2004).

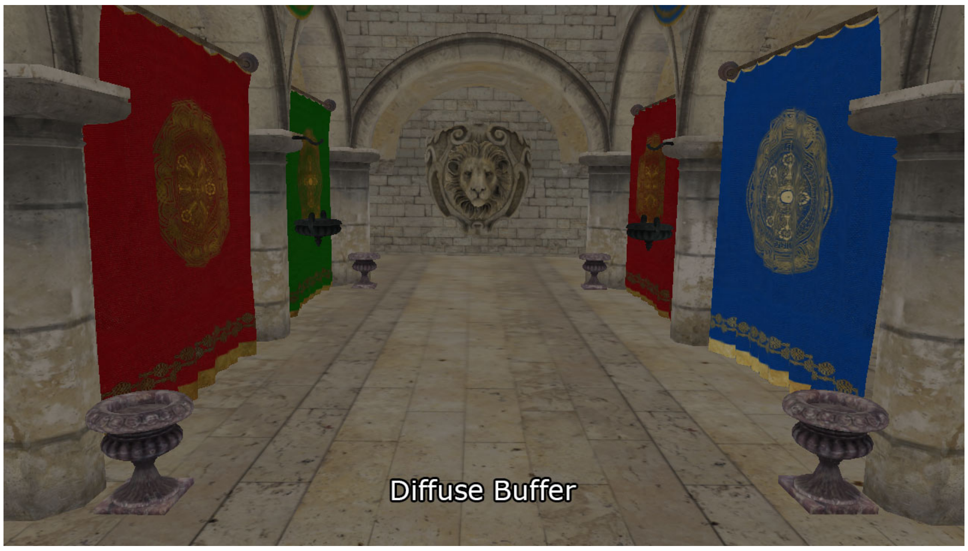

FIGURE 2.7: The diffuse buffer stores the diffuse contribution (Oosten, 2015).

Lighting Pass

The Lighting pass is the second pass of the Deferred Shading technique. In the Lighting pass geometric volumes that represent the lights in the scene are rasterized. The attributes that were written to the G-Buffers in the Geometry pass are used as inputs to compute the final lighting contribution of the pixels.

In order to determine which fragments are affected by the lights, the volume that represents the light source is rasterized; spheres for point lights, cones for spot lights, and full-screen quadrilaterals for directional lights. Fragments that are contained within the volume of the light are shaded by the light source. According to Michiel van der Leeuw (Leeuw, 2007) the phases required to shade the lit pixels are:

- For each light
  - Find and mark visible lit pixels
  - Shade lit pixels and add to framebuffer

To find the visible lit pixels, fragments that are in front of the far light boundary of the light volume are marked in the stencil buffer. This is illustrated in Figure 2.8.

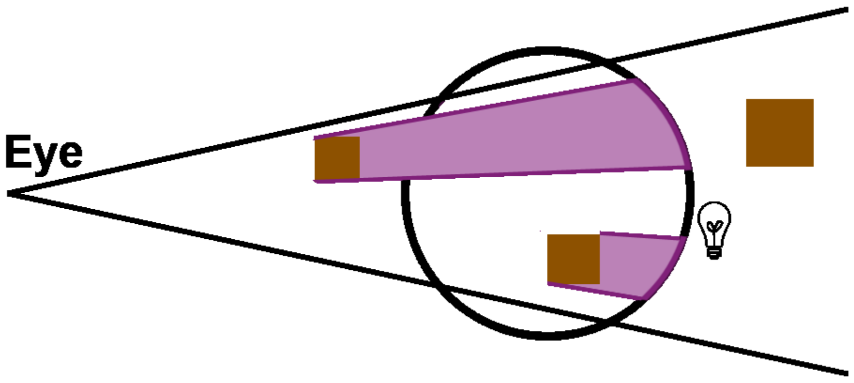

FIGURE 2.8: Mark pixels in front of the far light boundary.

Then the front faces of the light volume are rendered and any fragments that are behind the near light boundary and have been marked in the previous step are lit.

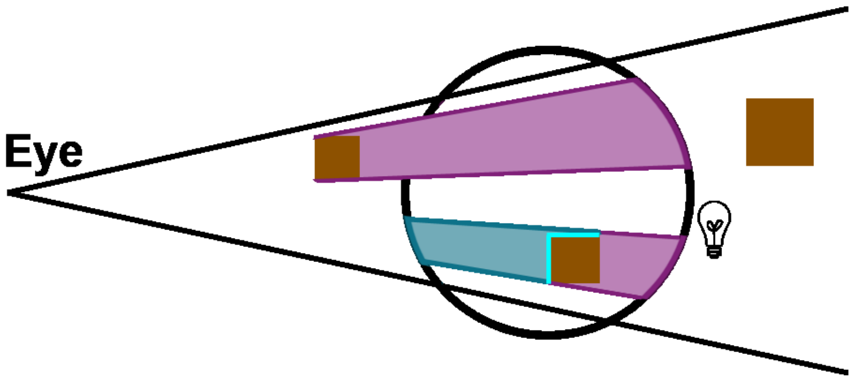

FIGURE 2.9: Find pixels inside the light volume and compute shading.

To compute the final shading of the pixels, the attributes contained in the G-Buffers are read and unpacked and the final lighting contribution is added to the Light Accumulation buffer using additive blending.

2.1.3 Tiled Forward Shading

Tiled Forward Shading (Olsson and Assarsson, 2011) works by dividing the screen into a uniform grid of tiles. The size of a tile is chosen in order to balance the trade off between memory usage and computational efficiency. Small tiles (8x8 pixels) will result in many screen space tiles, increasing the memory footprint, but reduces false positives at geometric boundaries. Large tiles (32x32 pixels) reduces the number screen space tiles, decreasing the memory footprint, but results in more false positives at geometric boundaries.

Tiled Forward Shading consists primarily of these passes:

1. Cull lights
2. Shade samples

Cull Lights

The light culling pass of the Tiled Forward Shading technique uses a uniform grid of tiles to assign each active scene light to tiles in the grid. This pass is usually executed using a compute shader which is invoked with one thread group for each tile in the grid. The size of the thread group is based on the size of the tile. For example, 8x8 tiles will result in 8x8 thread groups (64 threads per thread group), 16x16 tiles will result in 16x16 thread groups (256 threads per thread group), and 32x32 tiles will result in 32x32 thread groups (1,024 threads per group).

The frustum for the current tile can either be precomputed in an initialization phase or recomputed in the light culling pass. The view space frustum of the tile is used to perform light culling for the tile. The frustum for a tile in the light grid is visualized in Figure 2.10.

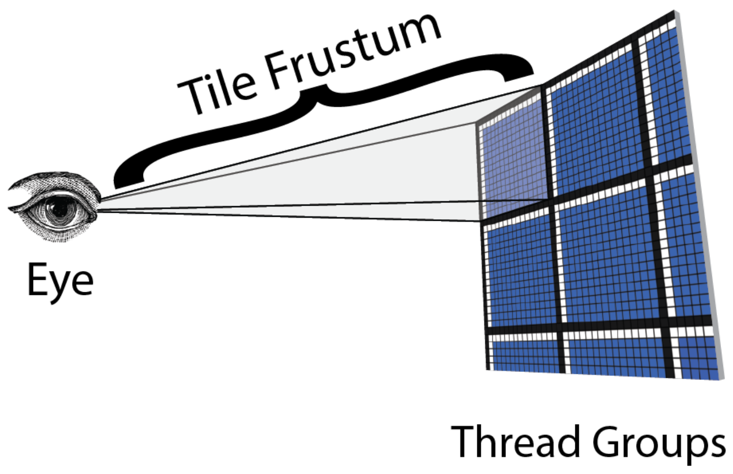

FIGURE 2.10: The view space frustum for a tile (Oosten, 2015).

The frustum for the tile is computed from the tile's screen space corners and the camera's near and far clipping plane. The tile's frustum is used to cull all of the active lights in the scene. If the light intersects the frustum for the tile, its index is added to a local *light index list* for the thread group.

Once all the lights in the scene have been culled against the view space frustum of the tile, the local light index list is copied to the global light index list. Each tile needs to store both the *offset* in the global light index list and the *number of lights* overlapping the tile. The offset and light counts for the tiles are stored in a 2D texture called the *light grid* where each texel corresponds to a tile in the grid. Figure 2.12 shows the data structures that are used to define which lights overlap the tiles.

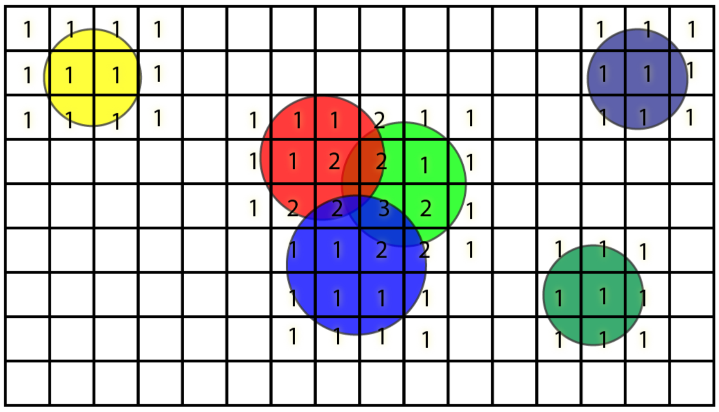

FIGURE 2.11: Lights overlapping tiles in the tile grid.

The pseudo code for the light culling algorithm is shown in Algorithm 2.2.

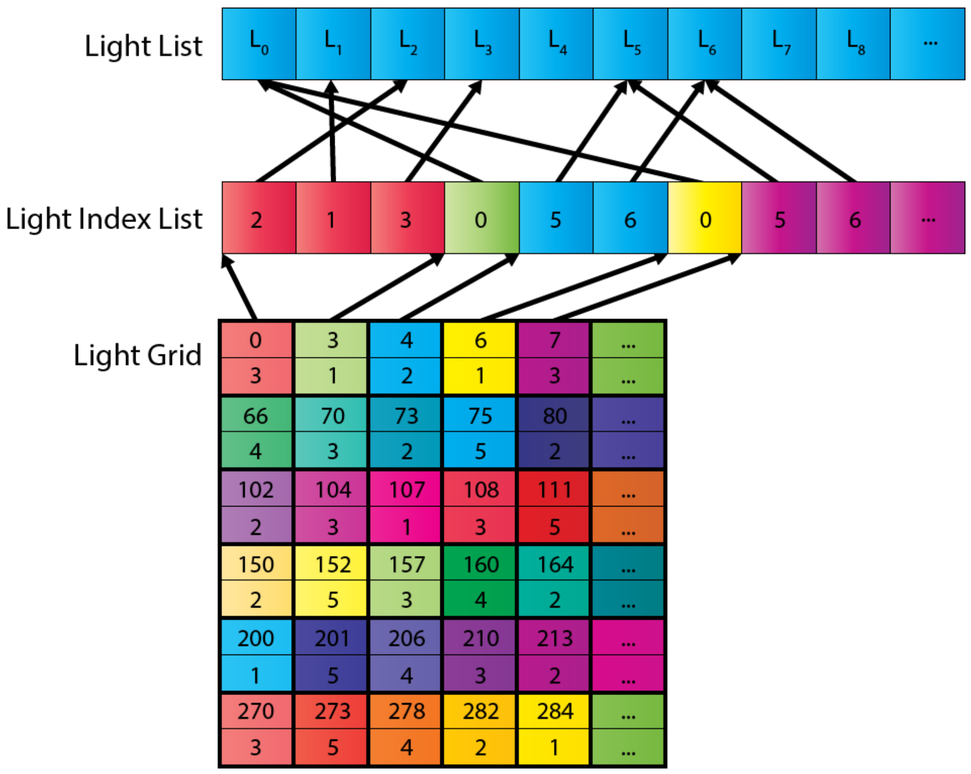

FIGURE 2.12: The data structures that are used to store the per tile light lists. The *Light Grid* stores the offset and the number of lights in the global *Light Index List* for each tile.

Algorithm 2.2 Cull lights algorithm

Require: L is a list of n lights.
Require: C is the current index in the global light index list.
Require: I is the global light index list.
Require: G is the 2D grid storing the index and light count into the global light index list.
Require: tid is the 2D index of the current thread within the dispatch.
Require: B is the 2D size of a tile.
Ensure: G is updated with the offset and light count of the current tile.

```
1:  function CULLLIGHTS($tid$)
2:      $t \leftarrow \lceil \frac{tid}{B} \rceil$
3:      $i \leftarrow \{0\}$
4:      $f \leftarrow$ FRUSTUM($t$)
5:      for $l$ in $L$ do
6:          if CULL( $l, f$ ) then
7:              APPENDLIGHT($l, i$)
8:          end if
9:      end for
10:     $c \leftarrow$ ATOMICINC($C, i.count$)
11:     $G(t) \leftarrow (c, i.count)$
12:     $I(c) \leftarrow i$
13: end function
```

The *Frustum* function on line 4 of Algorithm 2.2 retrieves the frustum for tile at index t. The tile frustum can either be computed on-the-fly or retrieved from list of precomputed frusta. The *Cull* function on line 6 checks to see if the light l is contained within the tile frustum f. This function returns $true$ if the light l is contained within the frustum f. The *AppendLight* function on line 7 appends the light l to the local light index list i. The *AtomicInc* function increments the global light counter C based on the number of lights that have been appended to the local light index list i.

Shade Samples

After the light culling pass is finished, the pixel shader takes the resulting light grid and the light index list to determine which lights affect the geometry inside each tile. The lighting models used by a traditional Forward Rendering technique can be applied without modification to the Tiled Forward Shading technique. The only difference in Tiled Forward Shading is that the light grid stores the offset and light counts into the global light index list. This means that only lights overlapping the current pixel's tile need to be considered during the lighting computation.

Depth Prepass

In the Tiled Forward Shading technique described by Olsson and Assarsson (Olsson and Assarsson, 2011), a depth pre-pass is an optional optimization pass which is executed before the *Cull Lights* pass. The depth information of the current frame can be used to constrain the clipping planes of the tile's view space frustum during light culling. The maximum depth value in the tile is used to define the far clipping plane and the minimum depth value in the tile is used to define the near clipping plane for the tile's view space frustum.

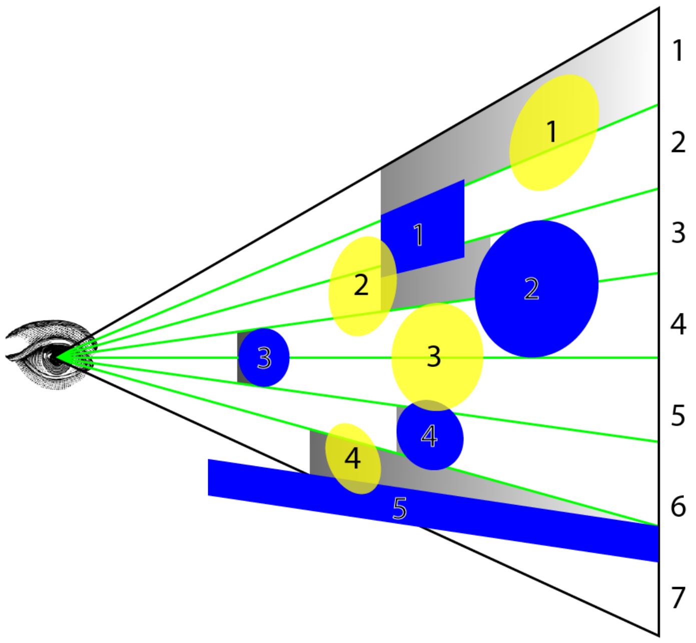

FIGURE 2.13: The blue objects in the image represent opaque scene objects. The yellow spheres represent light sources, and the gray shanded areas represent the depth range of the tile's view space frustum. Light 1 is incorrecty included in the frustum for Object 1 because the object partially covers the first tile creating a depth discontinuity in the tile (Oosten, 2015).

If the light grid was constructed without the depth pre-pass optimization then the same light index list and light grid can be used for both opaque and transparent geometry. Without the depth pre-pass optimization, many lights which may not effect the lighting result of the shaded pixels will be included in the light index list. False positives are not ideal and should be avoided. One way to avoid redundant lights in the light grid for opaque geometry is to construct two light grids, one for shading opaque geometry and another for shading transparent geometry. It is trivial to adapt the light culling algorithm shown in Algorithm 2.2 to account for transparent geometry. The light is first culled by the frustum whose far clipping plane is at the maximum depth value in the tile and the near clipping plane is at the position of the camera's near clipping plane. If the light is contained within the first frustum, it is added to the light list for transparent geometry. The frustum is further constrained to the minimum and maximum depth values within the tile and culled again. If the light is contained within the second frustum, it is added to the light list for opaque geometry. Figure 2.14 shows the depth bounds used to construct the tile frustums for opaque and transparent geometry.

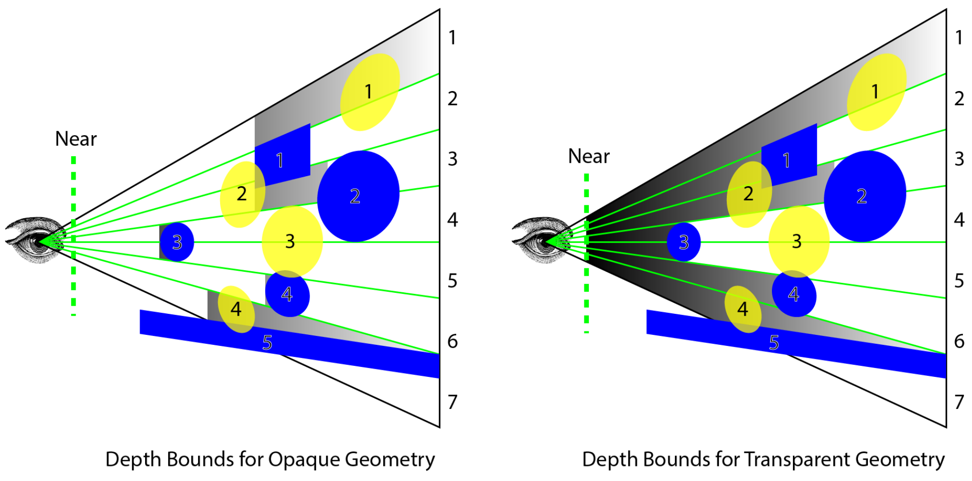
Near
1
2
3
4
5
6
7
1
2
3
4
5
Depth Bounds for Opaque Geometry
Depth Bounds for Transparent Geometry

FIGURE 2.14: The culling frusutm for opaque geometry can be derived from the minimum and maximum depth bounds within the tile (left). The culling frustum for transparent geometry uses the maximum depth value and the camera's near clipping plane (right).(Oosten, 2015).

2.1.4 Clustered Shading

Clustered Shading is a rendering technique that is similar to Tiled Shading that divides the 2D screen tiles into 3D clusters (Olsson, Billeter, and Assarsson, 2012). Clustered Shading consists of the following passes:

1. Cluster assignment
2. Find unique clusters
3. Assign lights to clusters
4. Shade samples

Cluster Assignment

In the context of Clustered Shading, a *cluster* is is a grouping of view samples. In Olsson et al.'s paper, samples are clustered based on the position and quantized normal of the sample. The sample position and normal are chosen for clustering because these attributes will most likely result in clusters that will be affected by the same set of lights.

A *cluster key* for each sample is computed by quantizing the sample's position in view space. The cluster grid uses a uniform subdivision in the width and height of the screen and an exponential subdivision in the depth. An exponential subdivision in the depth is chosen so that clusters remain as cubic as possible within the grid. Figure 2.15 shows an example of the exponential spacing used for the cluster subdivision.

The cluster key for a sample is an (i, j, k) tuple that is computed from the sample's screen-space coordinates (x_{ss}, y_{ss}) and the view space depth z_{vs}. For example, for clusters with a screen space tile size of (t_x, t_y), $(i, j) = (\lfloor x_{ss}/t_x \rfloor, \lfloor y_{ss}/t_y \rfloor)$.

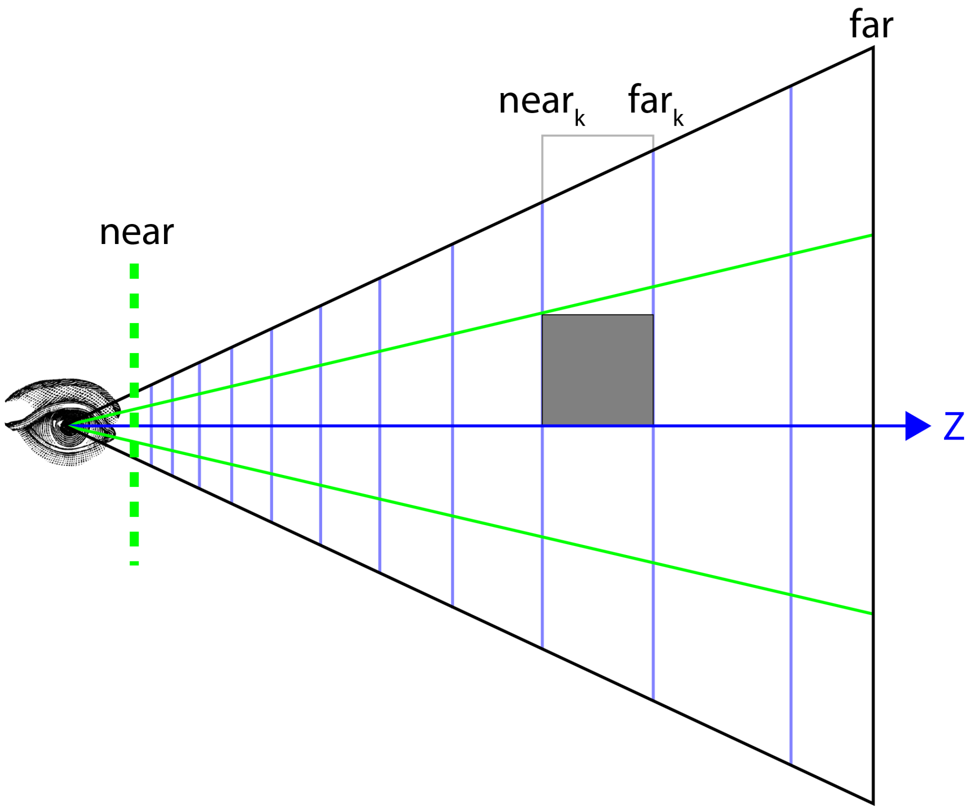

FIGURE 2.15: The cluster grid uses exponential spacing in the depth in order to keep the clusters as cubic as possible.

For a given field of view of 2θ and a number of subdivisions in the Y direction (S_y), k is computed using the following equation:

$$k = \left\lfloor \frac{\log\left(-z_{vs}/near\right)}{\log\left(1+\frac{2\theta}{S_y}\right)} \right\rfloor \tag{2.3}$$

Six bits in the cluster key are used to encode the quantized normal direction of the sample. The normals are quantized by mapping the direction of the normal to a cell of a 3D grid ($3 \times 3 \times 6 = 54$) mapped over a cube. Figure 2.16 shows a example of quantizing the normal on the cube. According to Olsson et al., clustering the normals improves light culling.

The cluster key is packed into a 32-bit integer. Eight bits are used for each of the i and j components. Eight bits is sufficient for a 8192×8192 screen buffer (assuming a tile size of 32×32 pixels). Ten bits are used to store the k component, and the last six bits are used to store the quantized normals. Figure 2.17 shows the per-component mapping to the cluster key. The cluster keys for each sample are written to a 2D screen space texture.

Find Unique Clusters

The next step of the Clustered Shading technique is to find the unique clusters in the cluster key buffer. Since many samples can be grouped into the same cluster, it is necessary to find only the unique clusters that are needed to perform the light assignment step. Unique clusters are identified by first

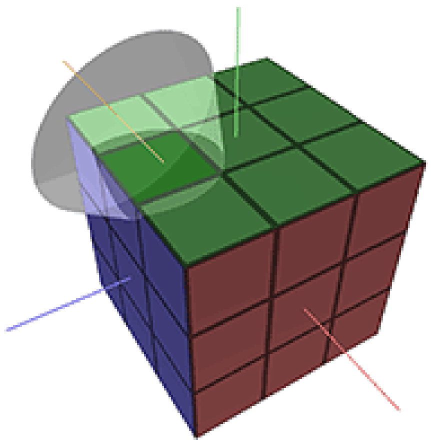

FIGURE 2.16: Normals are quantized to a 2D grid mapped over a cube (Olsson, Billeter, and Assarsson, 2012).

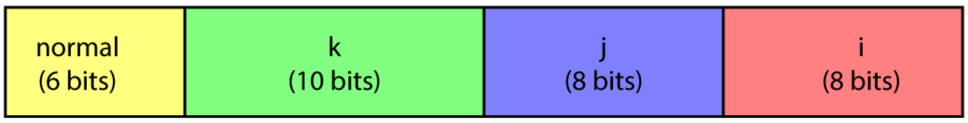

FIGURE 2.17: Cluster Key: 32-bits are used to encode the cluster key. 8 bits for the i and j components each, 10 bits for the k component, and 6 bits for the quantized normal.

sorting the cluster ID's within the screen space tiles. A parallel compaction algorithm is then run over the sorted cluster keys to identify the unique clusters. Figure 2.18 shows an example of the sorting and compaction of the cluster key buffer.

Assign Lights

To perform the light assignment step, a BVH is built over the light sources in the scene. The BVH is built by first sorting the light sources according to their Z-order (Morton Code) in the scene. The Z-order of the light source is determined by the descretized center position of the light source. The lowest level of the BVH is constructed by grouping 32 lights from the sorted list to form the leaf nodes of the BVH. This process is repeated to form the upper levels of the BVH until only a single root node remains.

For each unique cluster, the BVH is traversed testing the cluster's AABB against the AABB of the nodes of the BVH. At the leaf nodes, the bounding sphere of the light sources are used to test against the cluster's AABB. If the normals are available for the clusters, this is further used to reject the light if it will not effect any samples in the cluster.

Shade Samples

During shading, the cluster index is retrieved for each pixel being shaded, the light list for the cluster is retrieved and the sample is shaded normally similar to the shading pass of the standard forward rendering technique.

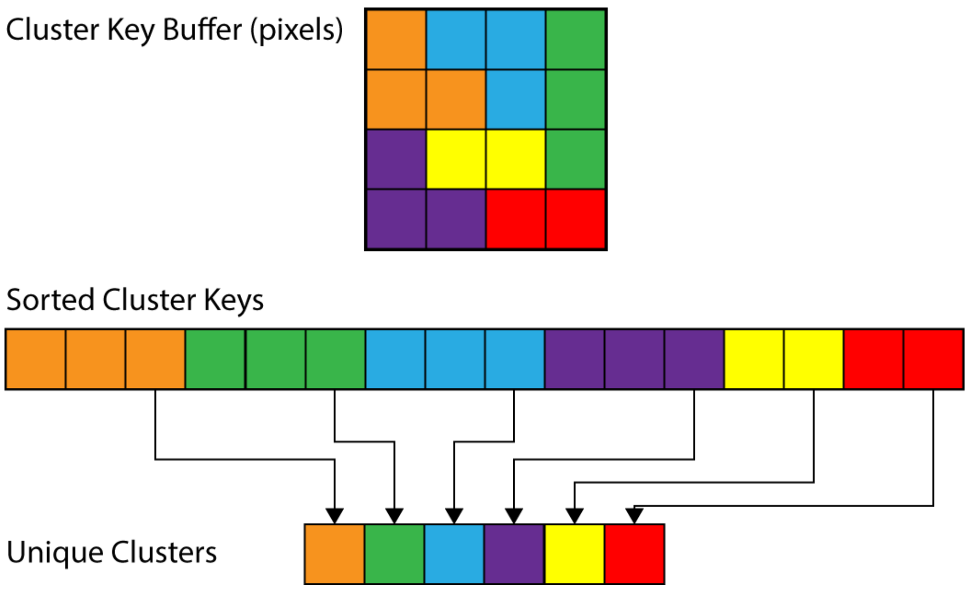

FIGURE 2.18: Unique Clusters: the cluster key buffer is sorted and compacted to find the unique clusters.

2.2 Summary

Forward Rendering rasterizes and shades geometric scene objects in a single pass through the rendering pipeline. *Forward Rendering* can be optimized either by minimizing the number of geometric objects that must be rendered, reducing the number of fragments that must be shaded, or by reducing the number of lighting calculations that must be performed.

Deferred Shading minimizes both the number of fragments that must be shaded and the number of lights that effect the screen pixels. *Deferred Shading* accomplishes this by decoupling the rasterization of geometry and computing the lighting into different passes. Geometric information is written to the G-Buffers in the Geometry pass. In the lighting pass, the light volumes are rasterized and the geometric attributes stored in the G-buffers are used to compute the final lighting contribution for the pixels contained in the light volume. Because *Deferred Shading* uses 2D screen space buffers to store the geometric attributes, it is not possible to support transparent materials.

Tiled Forward Shading first assigns the lights to 2D screen space tiles before rendering the scene geometry. During shading, the light lists are read from the light grid and only the lights that are overlapping with the tile for the current pixel need to be considered for lighting. *Tiled Forward Shading* suffers from false positives during the light assignment phase due to large depth disparities at geometric boundaries within a tile.

Clustered Shading improves upon *Tiled Forward Shading* because false positives resulting from large depth disparities at geometric boundaries within a tile are reduced but similar to *Deferred Shading*, *Clustered Shading* does not provide a solution for rendering transparent objects. *Clustered Shading* uses a 2D screen buffer for storing the per-pixel cluster keys. Since only a single cluster key can be stored for each screen pixel, transparent objects cannot be represented in this buffer.

Before the implementation of the *Volume Tiled Forward Shading* technique is described, it is important to provide a brief description of modern GPU architecture. The justifications for the choices that were made during the implementation of the *Volume Tiled Forward Rendering* technique require a basic understanding of how the GPU stores and transfers data, and how a program is executed on the GPU. In the next chapter, a brief survey of modern graphics hardware is provided.

Chapter 3

GPU Architecture

3.1 Introduction

In order to justify the choices that were made during the implementation of the *Volume Tiled Forward Rendering* technique, it is important to have a basic understanding of modern GPU architecture. This chapter provides a brief description of the high-level constructs that are available to the GPU programmer. These constructs include a *dispatch*, *thread group*, *warp*, and *thread*. A *dispatch* describes the execution domain for a compute shader program. The *dispatch* is further subdivided into *thread groups*. A *thread group* represents an autonomous collection of work that is capable of sharing memory and performing syncronization within the *dispatch*. A *warp* represents the maximum amount of work that can be executed in synchronized lock-step. In the context of an nVidia GPU a warp consists of 32 work units (NVIDIA, 2016a) and in the context of an AMD GPU a SIMD unit consists of 16 work units (AMD, 2012). A *thread* is considered the smallest level of execution in the context of a dispatch. A *thread* represents a single unit of work and produces an individual result. The remainder of this paper will be primarily concerned with the architecture of a modern NVidia GPU. The exact model of the GPU will be discussed in the performance and analysis chapter of this paper.

It is also important to understand optimized memory access patterns in order to comprehend the choices that were made during the development of the *Volume Tiled Forward Rendering* technique. Two optimization concepts are discussed in this chapter: coalesced access to global memory, and avoiding shared memory bank conflicts. Coalesced reads and writes to global memory ensures minimal transactions to global memory. Avoiding shared memory bank conflicts also improves memory throughput for shared memory. Adhering to these memory optimization practices ensures maximum memory throughput and improves overall performance of the GPU application.

3.2 Thread Dispatch

Work is executed on the GPU by issuing a *dispatch*. A dispatch consists of a number of *thread groups*. The dispatch must be executed with enough thread groups to compute the results for the problem domain. Each thread group consists of a number of *threads*. The number of threads in a thread group must be carefully chosen to make optimal use of the resources available to the *Streaming Multiprocessor* (SM). The SM is the processing unit that executes a thread group on the GPU. The SM executes 32 threads from the thread group, called a *warp*, in synchronous lock-step. Figure 3.1 shows a theoretical example of the layout of a dispatch. The image shows an example of a two-dimensional dispatch but the dispatch can be either one, two, or three-dimensional.

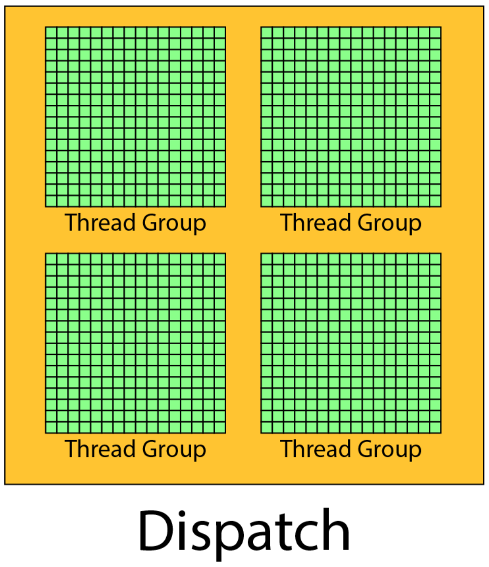

FIGURE 3.1: The dispatch consists of thread groups. Each thread group consists of a number of threads. This image shows a two-dimensional dispatch but the dispatch can be either one, two, or three-dimensional.

nVidia's Pascal GPU architecture (GP104) has 20 SM's, each SM contains 128 CUDA cores and 96 kB of shared memory (NVIDIA, 2016b). Each SM can schedule 4 thread groups at a time. Each thread group has access to a minimum of 16 kB of shared memory but to maintain maximum occupancy, one must be careful not to exceed 1/4 of the maximum amount of shared memory available to the SM per thread group (24 kB on Pascal GPU architecture). When designing the compute shader, for portability reasons, it is important not to exceed 16 kB of shared memory per thread group. If a thread group exceeds 1/4 of the available shared memory, the thread scheduler will reduce the number of simultaneous thread groups until the requested shared memory per thread group can be achieved. Exceeding 1/4 of the maximum amount of shared memory per thread group will result in reduced thread occupancy and the GPU will not be fully utilized.

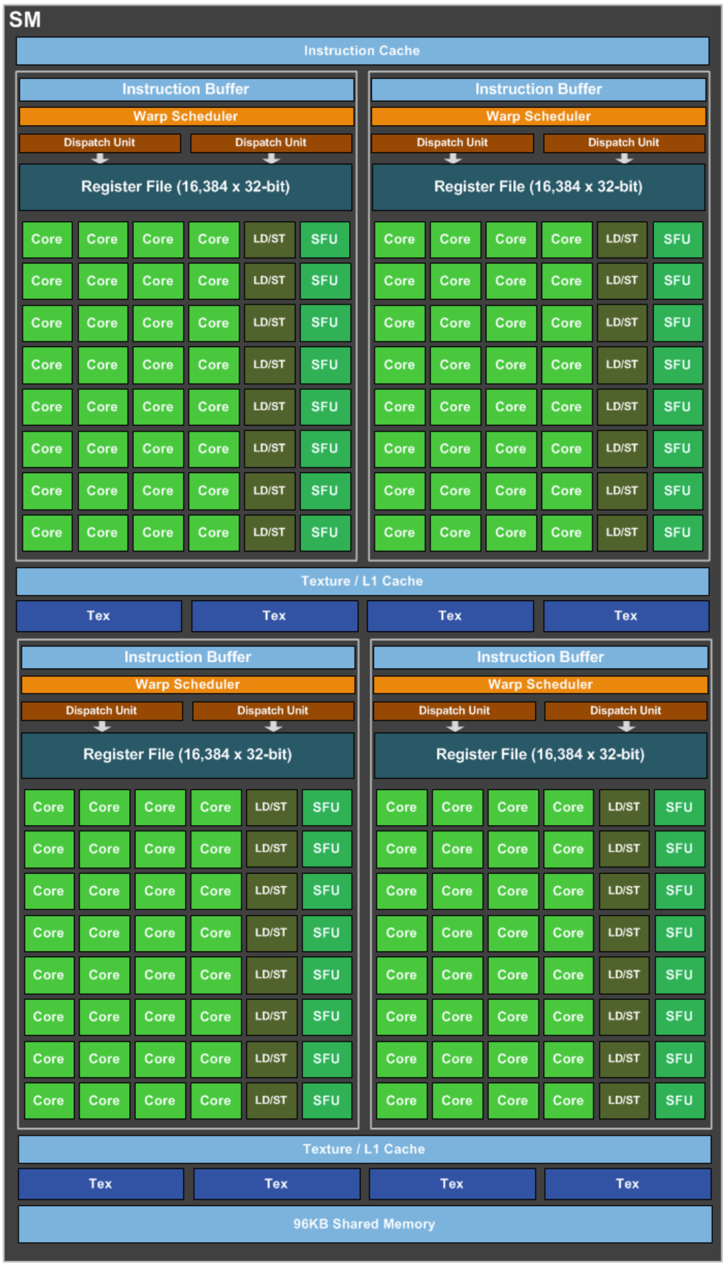

FIGURE 3.2: Pascal GP104 Streaming Multiprocessor architecture.

3.3 Coalesced Access to Global Memory

In order to optimize memory throughput, accesses to global memory should be coalesced within a warp. Coalesced memory access reduces the number of fetches required for a warp.

Global memory is accessed via 32 B, 64 B, or 128 B memory segments. The size of the memory segment is dependent on the size of the word accessed by each thread in a warp.

1. 32 B for 1 B words (8 bit values)
2. 64 B for 2 B words (16 bit values)

3. 128 B for 4 B, 8 B, and 16 B words (32 bit, 64 bit, and 128 bit values)

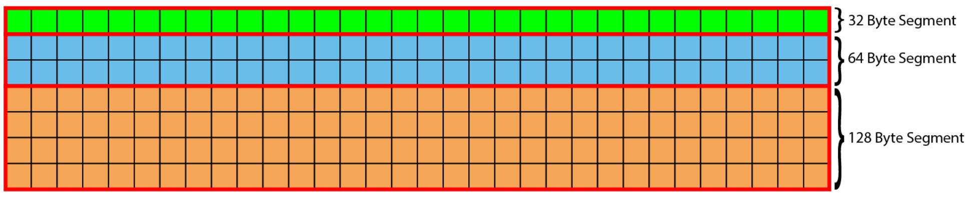

FIGURE 3.3: Global memory segments are 32 B for 1 B words, 64 B for 2 B words, and 128 B for 4 , 8 , and 16 B words.

Coalescing will occur when the kth thread in a warp accesses the kth word in a memory segment. If each thread in the warp sequentially accesses a 1 B value from global memory, this will result a single memory transaction of 32-bytes. If each thread in the warp accesses a 16 B (4-component floating-point) value from global memory, this will result in 4 128 B memory transactions (one for each quarter-warp)(NVIDIA, 2016a).

See Figure 3.4 for an example of each thread in a warp accessing a 16 B word from global memory.

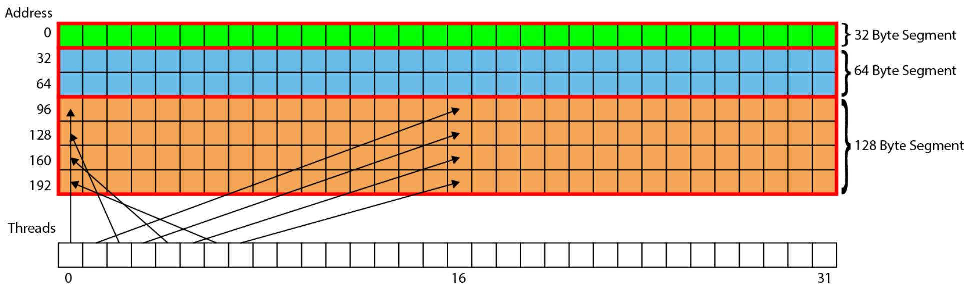

FIGURE 3.4: Each thread in a warp accesses a 16 B (4-component floating-point value) from global memory. This will result in 4 128 B memory transactions.

If memory access is misaligned or straddles a 128 B memory segment, then more memory transactions will be required to fulfil the request.

3.4 Avoid Bank Conflicts

When designing a GPU compute algorithm, it is important to be aware of how shared memory is accessed when a load or store operation is executed. Shared memory is stored in 32 banks of 32 bit words. Consecutive 32 bit words are interleved across the memory banks. If multiple threads in a warp access different 32 bit addresses that map to the same bank of shared memory, a *bank conflict* will occur and memory accesses will be serialized (Figure 3.5). If every thread in a warp accesses the same 32 bit address that maps to a single bank then the result will be broadcast to all threads in the warp (Figure 3.6). If every thread in a warp reads from a different memory bank then no bank conflict occurs and all reads can be performed simultaneously (Figure 3.7) (Oosten, 2011; NVIDIA, 2016c).

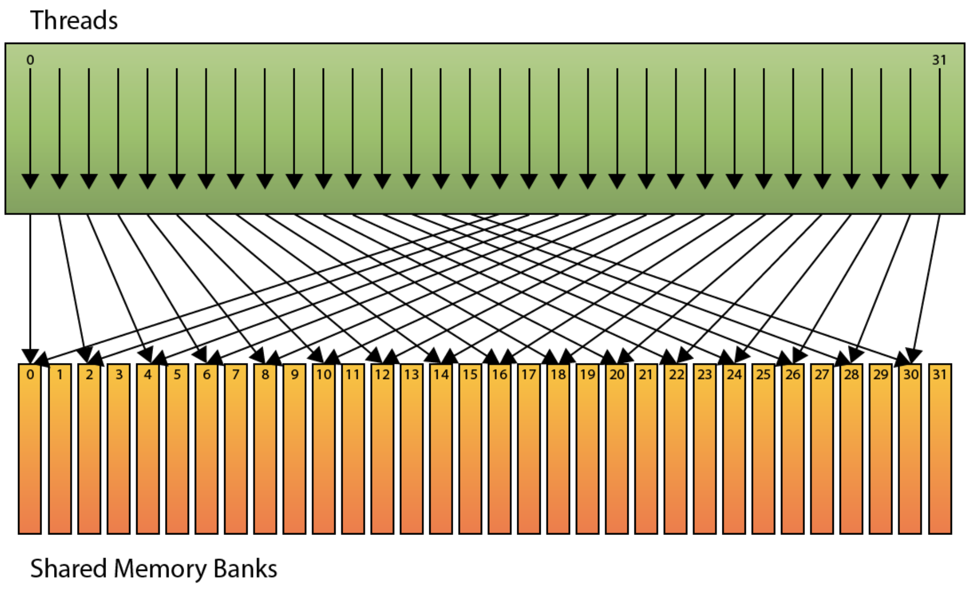

FIGURE 3.5: The shared memory is accessed by each thread with a stride of two. In this case, a 2-way bank conflict occurs. This will result in 2 serialized reads from shared memory.

.

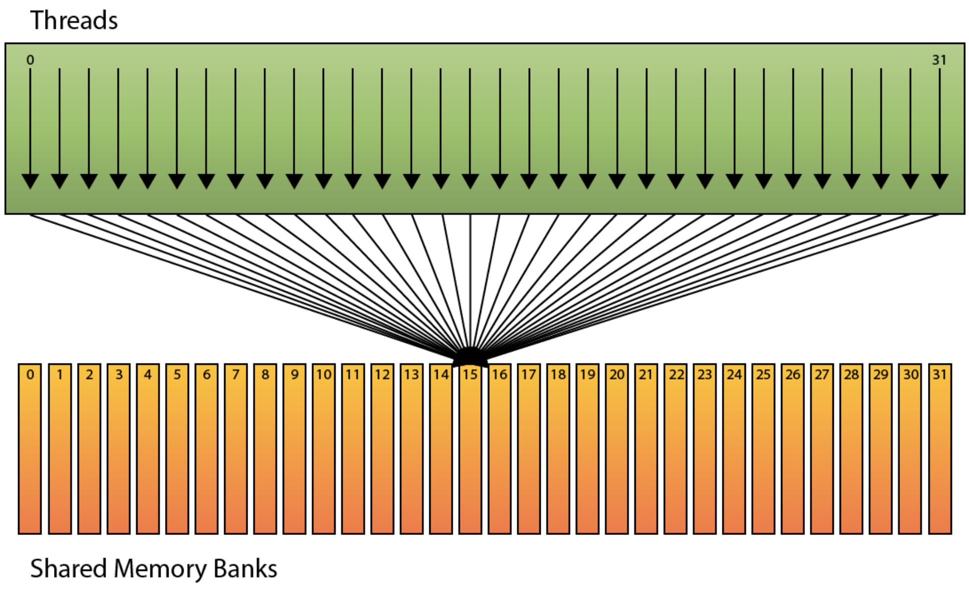

FIGURE 3.6: If every thread in a warp accesses the same address of a shared memory bank then the value is broadcast to all threads. In this case, no bank conflict occurs and all reads can be performed simutaniously (Oosten, 2011).

.

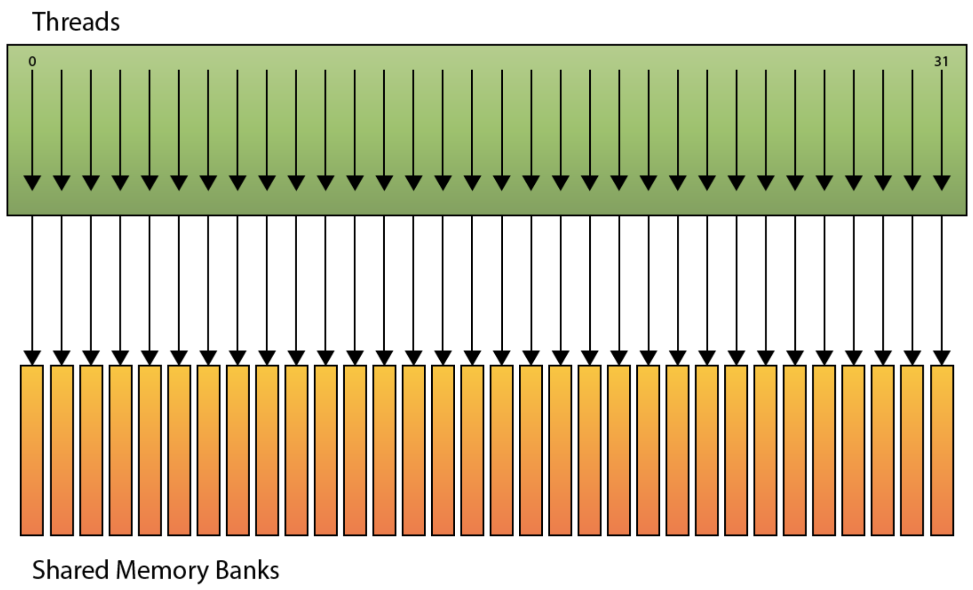

FIGURE 3.7: This image shows an example of linear addressing. If each thread in a warp accesses a different shared memory bank, then no bank conflict occurs.

.

Chapter 4

Parallel Primitives

4.1 Introduction

Parallel primitives form the building blocks for implementing parallel algorithms such as radix and merge sorting, determining minimum and maximum values, summing a set of values, to name just a few. This chapter describes two such parallel primitives: a *reduction* and a *scan*.

Reduction is a parallel primitive that reduces a set of n values to a single value by applying a binary associative operator $\oplus$ over the set. For example, a sum is an example of an operation that can be implemented as a reduction.

Scan is another parallel primitive that takes a set of n values $[a_0, a_1, \cdots, a_{n-1}]$ and applies the binary associative operator $\oplus$ to produce the set $[a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$ in the case of an *inclusive scan* or $[i_\oplus, a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2})]$ in the case of an *exclusive scan* where $i_\oplus$ is the identity of the set such that it does not effect the result of any value in the set when the $\oplus$ operator is applied. For example, $i_\oplus$ is a 0 for addition and a 1 for multiplication (Blelloch, 1989; Wilt, 2013).

4.2 Reduction

A parallel reduction is a useful building block for various GPU algorithms such as finding minimum or maximum values in a set, or summing a set of data in parallel. The parallel reduction technique described here is derived from *The CUDA Handbook* (Wilt, 2013) and *DirectCompute Optimizations and Best Practices* (Young, 2010).

The parallel reduction technique described here is a 2-pass reduction. In the first pass, a maximum of p thread groups are dispatched. Each thread group reduces to a single value that is written to global memory. The maximum number of thread groups (p) should be chosen so that only a single thread group is required to perform the final reduction in the second pass. The number of threads per thread group (t) should be chosen so that the reduction can be performed using group shared memory without exceeding group shared memory limits (Section 3.2). The second pass dispatches a single thread group that reduces the remaining p values into a single value.

The reduction requires two passes because the first pass writes a maximum of p values to global memory. Since there are no synchronization primitives for global memory access within a dispatch, invoking a second dispatch is the only way to guarantee that all of the thread groups from the first pass have finished writing their value to global memory.

There are several methods that can be used to implement a reduction (Wilt, 2013).

1. Serial
2. Pair-wise Log-Step
3. Interleaved Log-Step

In simplified terms, a reduction applies a binary associative operator $\oplus$ over a data set of n values reducing to a single value.

$$\sum_{i=0}^{n} a_i = a_0 \oplus a_1 \oplus a_2 \oplus a_3 \cdots \oplus a_n \tag{4.1}$$

As the name implies, a *serial* reduction (Figure 4.1) applies the binary operator to two operands per pass and requires $(n-1)$ passes to reduce to a single value. This technique is commonly used when the reduction is performed on a single thread of execution.

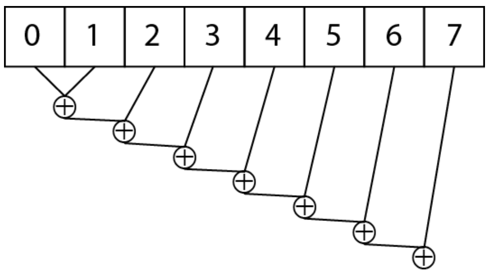

FIGURE 4.1: Serial reduction applied over an array of eight values.

$$(((((((a_0 \oplus a_1) \oplus a_2) \oplus a_3) \oplus a_4) \oplus a_5) \oplus a_6) \oplus a_7) \tag{4.2}$$

The *pair-wise* log-step reduction is performed in $\mathcal{O}(\log_2 n)$ steps. This method performs poorly on the GPU because when a single thread accesses adjacent memory locations in global memory, uncoalesced memory transactions will occur. As mentioned in Section 3.3 coalescing will occur when the k$^{\text{th}}$ thread in a warp accesses the k$^{\text{th}}$ word in a memory segment. In this case, the k$^{\text{th}}$ thread in a warp is accessing the 2k$^{\text{th}}$ word in a memory segment causing uncoalasced memory transactions to occur.

Pair-wise log-step reduction will also cause 2-way bank conflicts in shared memory to occur when this access pattern is used.

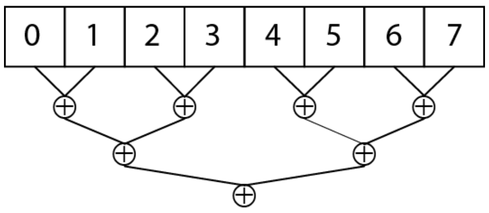

FIGURE 4.2: Pair-wise log-step reduction. This method does not make optimal use of memory access patterns in GPU memory.

$$(((a_0 \oplus a_1) \oplus (a_2 \oplus a_3)) \oplus ((a_4 \oplus a_5) \oplus (a_6 \oplus a_7))) \tag{4.3}$$

The log-step reduction algorithm performs best when each thread accesses global memory by interleaving addresses by a multiple of $(t \times p)$ where t is the number of threads per thread group and p is the number of thread groups in the dispatch.

The *interleaved* log-step reduction (Figure 4.3) performs better than the *pair-wise* reduction because bank conflicts are avoided when the addresses are accessed in an interleaved pattern (Figure 4.4).

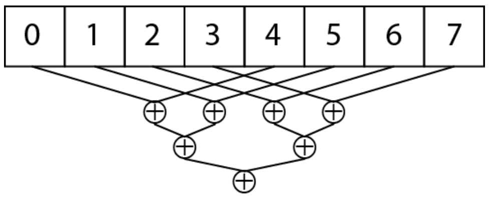

FIGURE 4.3: Interleaved log-step reduction. Accessing both global and shared memory is optimized.

$$(((a_0 \oplus a_4) \oplus (a_1 \oplus a_5)) \oplus ((a_2 \oplus a_6) \oplus (a_3 \oplus a_7))) \tag{4.4}$$

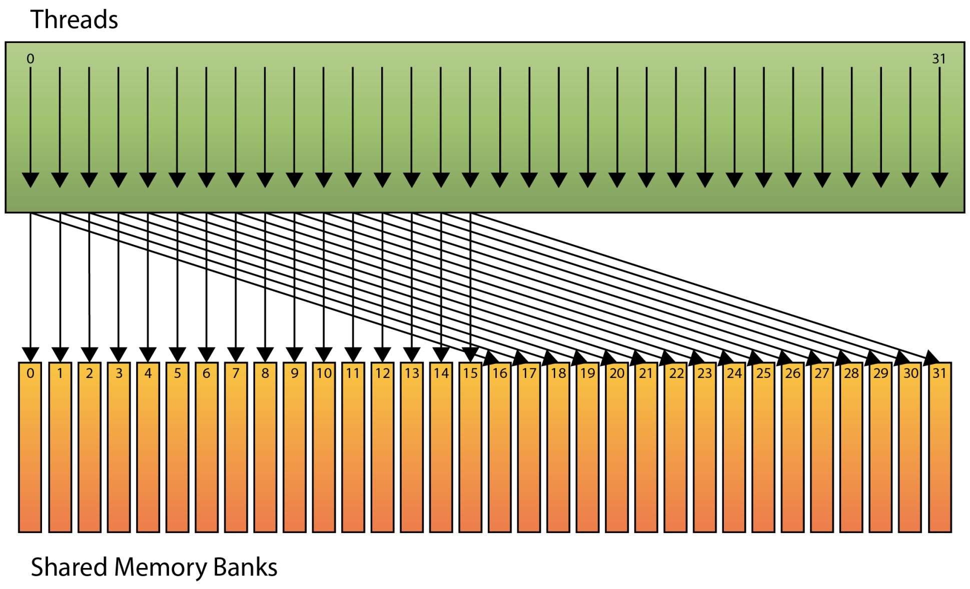

FIGURE 4.4: Bank conflicts are avoided when using an interleaved access pattern.

As previously mentioned, the parallel reduction algorithm operates in two passes. The first pass executes a dispatch of a maximum of p thread groups. Each thread group consists of t threads. In the first pass, each thread will reduce $n/(t \times p)$ values where n is the number of values to be reduced. Each thread writes its reduced value to group shared memory. The reduction continues by halving the number of active threads and reducing the two values at tid and $tid + (t/2^i)$ where tid is the thread index within the thread group, t is the number of threads in the thread group, and i is the current iteration of the reduction. The reduction step is repeated until $\lfloor t/2^i \rfloor < 1$. Algorithm 4.1 shows the pseudo code for the interleaved log-step reduction function.

Algorithm 4.1 Interleaved log-step parallel reduction.

```
Require: $x$ is a list of $n$ input values in group shared memory.
Require: $y$ stores the result of the reduction in global memory.
Require: $gid$ is the index of the current thread group.
Require: $tid$ is the index of the thread within the thread group.
Require: $p$ is the number of thread groups.
Require: $t$ is the number of threads per thread group.
1: function LogStepReduction($gid$,$tid$,$\oplus$)
2:     $k \leftarrow t/2$
3:     while $k > 0$ do
4:         if $tid < k$ then
5:             $x[tid] \leftarrow x[tid] \oplus x[tid + k]$
6:         end if
7:         $k \leftarrow \lfloor k/2 \rfloor$
8:     end while
9:     if $tid = 0$ then
10:        $y[gid] \leftarrow x[tid]$
11:    end if
12: end function
```

It is interesting to note that the *LogStepReduction* function shown in Algorithm 4.1 does not show the group shared memory barrier that is required to guarantee all threads in a thread group have finished reading or writing to shared memory. The group shared memory barriers should be inserted after line 6 in Algorithm 4.1.

The *LogStepReduction* function shown in Algorithm 4.1 can be optimized in the case the thread count (k) is less than or equal to the size of a warp. In this case, when k is 32 or less then the threads in the thread group execute in warp-synchronise lock-step and group shared memory barriers are no longer required. According to Young, unrolling the last warp and removing group shared memory barrier results in a 45% increase in performance (Young, 2010). Algorithm 4.2 shows the *LogStepReduction* function with the warp-synchronous optimization applied. In this case the group shared memory barrier is added to the algorithm for clarity.

Algorithm 4.2 Interleaved log-step parallel reduction with warp-synchronous optimization.

```
Require: $x$ is a list of $n$ values in group shared memory.
Require: $y$ stores the result of the reduction in global memory.
Require: $gid$ is the index of the current thread group.
Require: $tid$ is the index of the thread within the thread group.
Require: $p$ is the number of thread groups.
Require: $t$ is the number of threads per thread group.
function LogStepReduction($gid$,$tid$,$\oplus$)
    $k \leftarrow t/2$
    while $k > 32$ do
        if $tid < k$ then
            $x[tid] \leftarrow x[tid] \oplus x[tid + k]$
        end if
        GroupSharedMemoryBarrier
        $k \leftarrow \lfloor k/2 \rfloor$
    end while
    if $tid < 32$ then
        while $k > 0$ do
            $x[tid] \leftarrow x[tid] \oplus x[tid + k]$
            $k \leftarrow \lfloor k/2 \rfloor$
        end while
    end if
    if $tid = 0$ then
        $y[gid] \leftarrow x[tid]$
    end if
end function
```

The result of the first pass is p values written to global memory, where p is the number of thread groups dispatched in the first pass. In the second pass, a single thread group is executed to reduce the final p values from the first pass. The algorithm for the second pass is identical to that of the first pass. The only difference between the first and second passes of the reduction algorithm is the number of resulting values.

4.3 Scan

According to Blelloch, a scan operation takes a binary operator $\oplus$ with the identity $i_\oplus$ and an ordered set $[a_0, a_1, \cdots, a_{n-1}]$ of n elements and returns the ordered set $[i_\oplus, a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2})]$ where $i_\oplus$ is the identity value for the $\oplus$ operator (0 for addition and 1 for multiplication) (Blelloch, 1989).

If the $\oplus$ operator is addition then the scan operation applied to the array

$$[2, 1, 2, 3, 4, 8, 13, 21] \tag{4.5}$$

would produce

$$[0, 2, 3, 5, 8, 12, 20, 33] \tag{4.6}$$

The pseudo code for a sequential scan is shown in Algorithm 4.3.

Algorithm 4.3 Sequential scan.

```
Require: $x$ is a list of $n$ values.
Ensure: $y$ contains the result of the scan operation.
function SequentialScan($x$,$i_\oplus$,$\oplus$)
    $y[0] \leftarrow i_\oplus$
    for $i = 1$ to $n$ do
        $y[i] \leftarrow y[i-1] \oplus x[i-1]$
    end for
    return $y$
end function
```

The sequential scan algorithm operates in $\mathcal{O}(n)$ time in a single thread of execution. The algorithm can be parallelized to operate in $\mathcal{O}(\log_2 n)$ steps. For each step i of the parallel scan algorithm, if the thread id (tid) of the current thread is greater than 2^i then the value of $x[tid]$ and $x[tid - 2^i]$ is summed and stores the result at $x[tid]$.

The pseudo code for the parallel scan is shown in Algorithm 4.4. This algorithm is based on the naive parallel scan operation presented by Mark Harris, Shubhabrata Sengupta, and John D. Owens in chapter 39 of *GPU Gems 3* (Harris, Sengupta, and Owens, 2008).

Algorithm 4.4 Parallel scan.

```
Require: $x$ is a list of $n$ values.
Require: $I_\oplus$ is the identity.
Require: $tid$ is the ID of the thread in the thread group.
Ensure: $y$ contains the result of the scan operation.
function ParallelScan($x$,$tid$,$i_\oplus$,$\oplus$)
    if $tid = 0$ then
        $y[0] \leftarrow i_\oplus$
    else
        $y[tid] \leftarrow x[tid - 1]$
    end if
    for $i \leftarrow 0$ to $\log_2 n - 1$ do
        $t \leftarrow y[tid]$
        if $tid > 2^i$ then
            $t \leftarrow t \oplus y[tid - 2^i]$
        end if
        $y[tid] \leftarrow t$
    end for
    return $y$
end function
```

In the first step, the output array y is primed by copying all of the elements from the input array (x) to the output array (y) shifted one index to the right. A loop is iterated $log_2 n$ times and for each thread whose thread ID is greater than 2^i where i is the loop iteration counter, the sum of $y[tid] \oplus y[tid - 2^i]$ is computed and stored at $y[tid]$ where tid is the ID of the thread in the thread group. Figure 4.5 shows a graphical implementation of performing the parallel scan over a set of eight values. Each thread operates on a single index in the set.

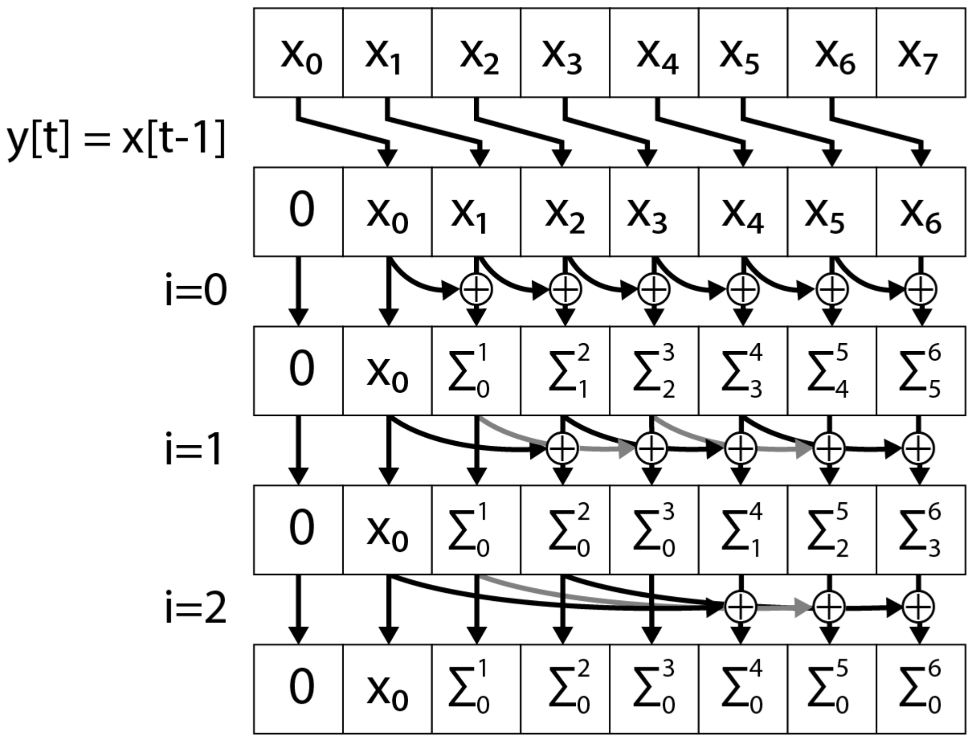
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
y[t] = x[t-1]
0 x_0 x_1 x_2 x_3 x_4 x_5 x_6
i=0
0 x_0 Σ_0^1 Σ_1^2 Σ_2^3 Σ_3^4 Σ_4^5 Σ_5^6
i=1
0 x_0 Σ_0^1 Σ_0^2 Σ_0^3 Σ_1^4 Σ_2^5 Σ_3^6
i=2
0 x_0 Σ_0^1 Σ_0^2 Σ_0^3 Σ_0^4 Σ_0^5 Σ_0^6

FIGURE 4.5: Parallel scan. For each iteration i of the parallel scan, each thread t larger than 2^i computes $x[t] \oplus x[t-2^i]$ and stores the result at index t.

Chapter 5

Sorting

5.1 Introduction

Sorting is an important building block for the construction of spatial data structures such as a *Bounding Volume Hierarchies* (BVH). As computer architectures evolve towards an increasingly parallel model, it is becoming progressively more important to understand parallel algorithm design. This chapter deals with the problem of sorting using the parallel power of the GPU. Two sorting algorithms are described; radix sort and merge sort.

Radix sort is a sorting algorithm that sorts a set of integer keys by only considering a single digit starting at the least significant digit (LSD) and proceeding to the most significant digit (MSD). Because radix sort is a *stable* sort (if two items are equal, their order doesn't change) the ordering of the values is guaranteed to be correct after sorting the individual digits. Radix sort has a runtime complexity of $\mathcal{O}(wn)$ where w is the size of the key (number of significant digits) and n is the number of keys to be sorted.

Merge sort is a sorting algorithm that merges two sorted lists by comparing values from each list according to some sort condition. The value that adheres to the sort condition (less than, less than or equal, greater or equal, greater) is chosen and inserted into the resulting list. Every iteration of merge sort creates a resulting list that is twice the size of the of the original lists. To sort a completely random list of n values requires $\log_2 n$ iterations and has a runtime complexity of $\mathcal{O}(n \log_2 n)$.

In order to make optimial use of GPU hardware, a hybrid sorting approach is applied that first uses radix sort to sort chunks of 256 keys from the input. A parallel merge sort is then used to merge the 256 key chunks to generate the final sorted list. Limiting the radix sort to 256 keys per chunk ensures the sorting can be performed entirely in high-speed on-chip shared memory without exceeding the $16\,\mathrm{kB}$ shared memory limit described in Section 3.2. A parallel merge sort was chosen to sort the resulting chunks because a very efficient parallel merge sort algorithm exists that minimizes gather operations from global memory, and evenly partitions the workload to make optimal use of the GPU hardware (Green, McColl, and Bader, 2012; Harris, Sengupta, and Owens, 2008).

5.2 Radix Sort

The first pass of the hybrid sorting algorithm uses a radix sort that produces sorted chunks of 256 keys. Radix sort works by considering a single bit from the sort key and placing all keys with a 0 in that bit before all keys with a 1. The algorithm starts at the least-significant bit of the key and the process is

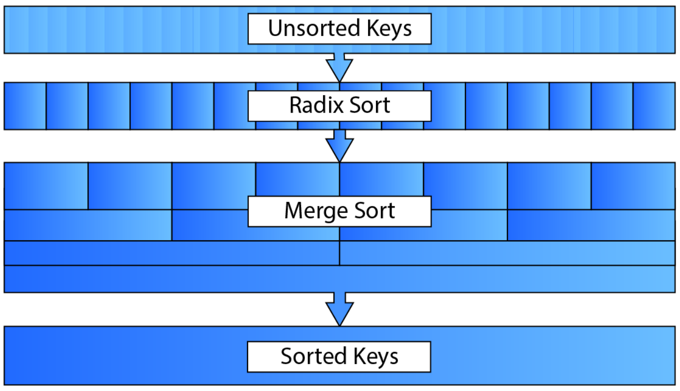

FIGURE 5.1: In order to sort the keys, a hybrid sorting approach is used. The unsorted keys are first sorted into chunks of 256 keys using a parallel radix sort. A merge sort is repeatedly applied to the sorted chunks to produce the final sorted list.

repeated until the most-significant bit. The radix sort algorithm shown here is based on the algorithm described in Chapter 39 of GPU Gems 3 (Harris, Sengupta, and Owens, 2008).

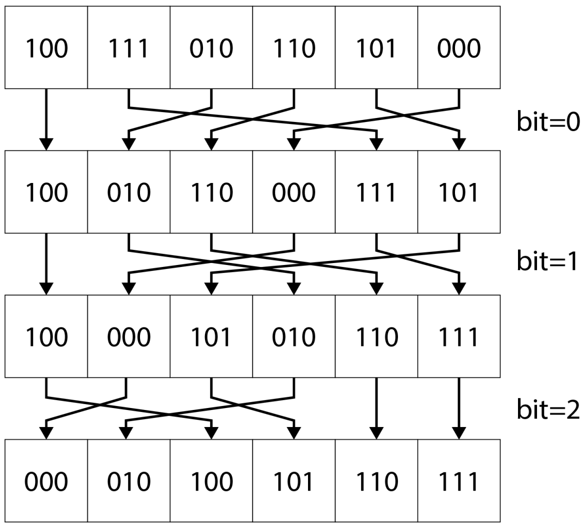

FIGURE 5.2: Radix sort loops over the bits of the key starting at the least-significant bit. All keys with a 0 in the bit are placed before keys with a 1. The process is repeated for each bit resulting in a sorted list.

The radix sort algorithm uses a parallel scan operation (Section 4.3) in order to determine the number of sort keys that contain a 0 in the current bit.

The radix sort algorithm starts by reading the keys from global memory into shared memory. Performing the radix sort in shared memory ensures that only a single read and a single write to global memory is performed per thread.

For the sake of the following algorithm, sort keys with a 1 in the current bit will be referred to as a *true* sort key sort keys with a 0 in the current bit

Algorithm 5.1 Parallel radix sort.

```
Require: $x$ is an unsorted list of $n$ keys.
Require: $numBits$ is the number of bits of the sort key.
Require: $tid$ is the ID of the thread in the thread group.
Require: $e$ stores a 0 for true sort keys, and a 1 for false sort keys.
Require: $f$ contains the destination index of all false sort keys.
Require: $d$ is the destination index for the keys.
Ensure: $y$ contains the sorted keys.
 1: function RADIXSORT($x$,$tid$)
 2:     for $B \leftarrow 0$ to $numBits$ do
 3:         $b \leftarrow$ BITMASK($x[tid]$,$B$)
 4:         if $b = 0$ then
 5:             $e[tid] \leftarrow 1$
 6:         else
 7:             $e[tid] \leftarrow 0$
 8:         end if
 9:         $f \leftarrow$ PARALLELSCAN($e$,$tid$,0,+)
10:         $totalFalses \leftarrow e[n-1] + f[n-1]$
11:         $t[tid] \leftarrow tid - f[tid] + totalFalses$
12:         if $b = 1$ then
13:             $d[tid] \leftarrow t[tid]$
14:         else
15:             $d[tid] \leftarrow f[tid]$
16:         end if
17:         $x[tid] \leftarrow x[d[tid]]$
18:     end for
19:     $y[tid] \leftarrow x[tid]$
20:     return $y$
21: end function
```

will be referred to as a *false* sort key.

The following steps are then repeated for each bit of the key:

1. Using a temporary array (e) stored in shared memory, write a 1 for all false sort keys ($b = 0$) and a 0 for all true sort keys ($b = 1$).

2. Perform a parallel prefix scan over array e and store the result in another array (f). f now contains the destination index of all false sort keys.

3. The last element of array e plus the last element of array f contains the total number of false sort keys. This value is written to a shared variable called $totalFalses$.

4. The destination index d for a true sort key at index i is $d = i - f[i] + totalFalses$. The destination index d for false sort keys is $d = f[i]$

5. The original sort keys are written to the keys array in shared memory according the destination index d and step 1 is repeated for the next increasing significant bit.

After all of the bits of the sort keys have been processed, the results are copied to global memory.

The pseudo-code for the radix sort is shown in Algorithm 5.1.

Figure 5.3 shows an example of the radix sort algorithm being applied to the least significant bit of the input.

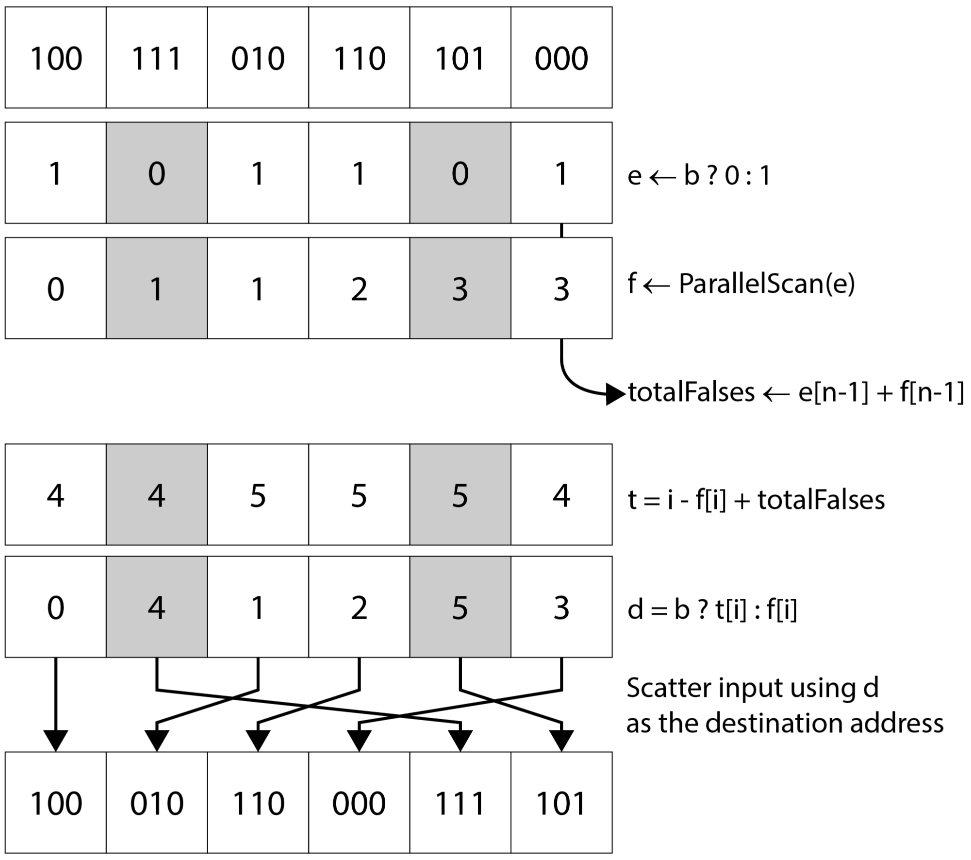

FIGURE 5.3: The radix sort algorithm applied to the least significant bit of the input.

After the chunks have been sorted using the radix sort algorithm the sorting pass continues by performing repeated applications of merge sort until a final sorted list of keys remains. In the next section the merge sort algorithm is described.

5.3 Merge Sort

The resulting 256 key chunks from the radix sort need to be merged to produce the final list of sorted keys. Radix sort is no longer a viable sorting algorithm because the larger chunks would not fit into group shared memory requiring more fetches from global memory. The technique used to merge the chunks is called *Merge Path* (Green, McColl, and Bader, 2012).

It is necessary to first describe the serial merge function since the parallel merge uses serial merge to perform the actual merging of the values. The serial merge function operates on two sorted lists A and B of size $|A|$ and $|B|$ and produces a third list C of size $|C| = |A| + |B|$. The merge iterates over the elements of A and B copying the smallest value from either A or B into C. If either of the input lists are exhausted then the remaining elements from the other list are copied to C. Algorithm 5.2 shows the pseudo-code for the serial merge.

Algorithm 5.2 Serial merge sort.

```
Require: $A$ is a sorted list.
Require: $B$ is a sorted list.
Ensure: $C$ is a sorted list of size $|A| + |B|$.
 1: function SERIALMERGE($A$,$B$)
 2:     $a \leftarrow 0$
 3:     $b \leftarrow 0$
 4:     for $i \leftarrow 0$ to $|A| + |B|$ do
 5:         if $a < |A|$ and $A[a] < B[b]$ or $b \geq |B|$ then
 6:             $C[i] \leftarrow A[a]$
 7:             $a \leftarrow a + 1$
 8:         else
 9:             $C[i] \leftarrow B[b]$
10:             $b \leftarrow b + 1$
11:         end if
12:     end for
13:     return $C$
14: end function
```

The merge sorting algorithm can be visualized by placing the elements of list A and B in a grid where the elements of A are placed in the columns and the elements of B are placed in the rows of the grid. Then the sequential merge can be visualized as a path that moves from the top-left corner of the grid to the bottom-right corner of the grid. The path moves right to the next column when the current element in A is less than the current element in B and it moves down to the next row when the current element in B is less than the current element in A. The path that is formed through the grid is called the *merge path*. Figure 5.4 shows an example of the merge path through the virtual grid.

If the merge path through the virtual grid can be found without sorting the lists then we can perform the merge sort algorithm in parallel by splitting the work based on the number of values to be sorted per thread group. In order to make optimal use of the GPU resources and minimize gathers and scatters to global memory, the merge sort should also be performed in group shared memory. The amount of work performed by each thread should also be the same so that each thread finishes its merge sort at the same time reducing the chance of idle threads.

To parallelize the merge sort algorithm, the input lists are split based on the number of elements that should be sorted by each thread. A diagonal line which represents the splitting of the input lists can be drawn through the virtual grid. The point at which the diagonal line intersects with the merge path indicates the point at which the two lists are split. Figure 5.5 shows an example of the diagonal split through the virtual grid. The point that the diagonal crosses the merge path determines which values from A and B will be merged by each thread.

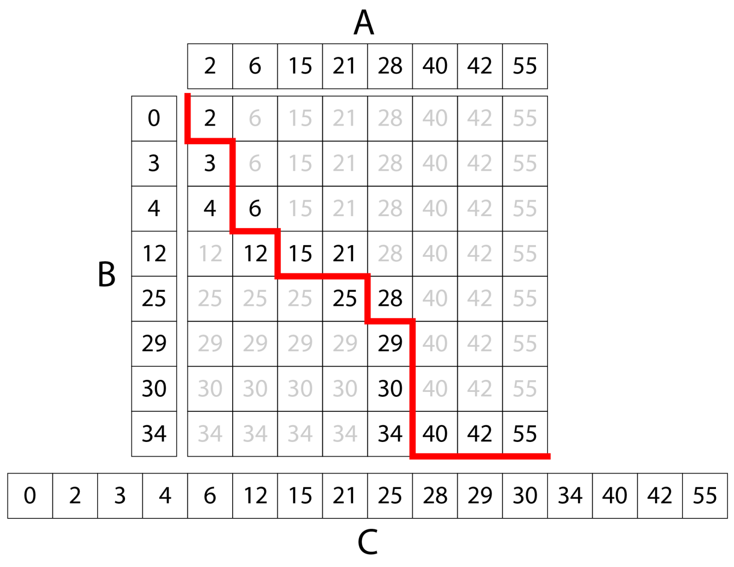

FIGURE 5.4: The serial merge can be visualized as a grid that is formed by placing the elements of A in the columns of the grid and the elements of B in the rows of the grid. The red line in represent the merge path that is the result of merging the elements of A and B to form the sorted list C.

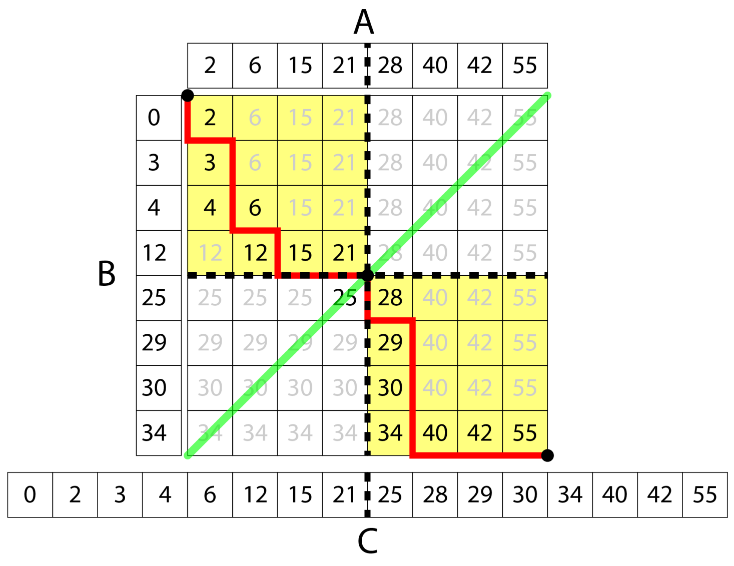

FIGURE 5.5: The green diagonal line represents the split that is made to parallelize the merge sort function. Where the diagonal line intersects the merge path indicates the values from A and B that will be sorted by each thread. In this example, the diagonal split occurs every 8 values. In this case 4 values from A and 4 values from B will be merged by each thread.

The point at which the diagonal split crosses the merge path is called the *merge path partition*. To find the point in list A and list B to begin the merge, a binary search is executed over the lists. The binary search starts at the midpoint between the start of the lists and the diagonal point. If the value of A at the midpoint is less than the value of B at the diagonal minus the midpoint then the starting point is set to the midpoint otherwise the end point is set to the midpoint and the search starts again. Algorithm 5.3 shows the pseudo-code for the binary search that finds the merge path partition point. The binary search performs in $\mathcal{O}(\log_2 n)$ time where n is the size of the diagonal and requires at most $\mathcal{O}(\log_2 n)$ reads from global memory.

Algorithm 5.3 Parallel merge path partition.

```
Require: $A$ is a sorted list.
Require: $B$ is a sorted list.
Require: $diag$ is the diagonal split.
 1: function MERGEPATHPARTITION($A$,$B$,$diag$)
 2:     $begin \leftarrow$ MAX(0,$diag - |B|$)
 3:     $end \leftarrow$ MIN($diag$,$|A|$)
 4:     while $begin < end$ do
 5:         $mid \leftarrow \lfloor (begin + end)/2 \rfloor$
 6:         if $A[mid] < B[diag - 1 - mid]$ then
 7:             $begin \leftarrow mid + 1$
 8:         else
 9:             $end \leftarrow mid$
10:         end if
11:     end while
12:     return $begin$
13: end function
```

To reduce reads from global memory during the serial merge operation, it is ideal to first store the values from both A and B in shared memory. The merge path partition algorithm can be applied to the entire thread group to determine the merge path partition for each thread group. When the merge path partitions for each thread group are known, the sub array of A and B can be loaded into shared memory. Algorithm 5.4 shows the pseudo-code for the parallel merge. It is assumed that the sub arrays of A and B have been loaded into shared memory based on the merge path partitions of the thread group.

Algorithm 5.4 Parallel merge sort.

```
Require: $A$ is a sorted sub array stored in shared memory.
Require: $B$ is a sorted sub array stored in shared memory.
Require: $vt$ is the number of values to sort per thread.
Require: $tid$ is the ID of the thread within the thread group.
Ensure: $C$ stores the result of sorting $A$ and $B$.
 1: function ParallelMerge($A$,$B$,$vt$,$tid$)
 2:     $diag_0 \leftarrow vt * tid$
 3:     $diag_1 \leftarrow vt * (tid + 1)$
 4:     $a_0 \leftarrow$ MergePathPartition($A$,$B$,$diag_0$)
 5:     $a_1 \leftarrow$ MergePathPartition($A$,$B$,$diag_1$)
 6:     $b_0 \leftarrow diag_0 - a_0$
 7:     $b_1 \leftarrow diag_1 - a_1$
 8:     $C[diag_0 \cdots diag_1] \leftarrow$ SerialMerge($A[a_0 \cdots a_1]$,$B[b_0 \cdots b_1]$)
 9:     return $C$
10: end function
```

The parallel merge function shown in Algorithm 5.4 first computes the diagonals for the current thread. The merge path partitions are computed based on the diagonals and the ranges of A and B that will be merged serially for this thread are determined. Each thread then performs a serial merge over the sub-arrays of A and B and stores the result in C.

The parallel merge algorithm shown here is a simplified version of the final parallel merge function and does not show the steps required to move the sub-arrays of A and B into shared memory. The algorithm also does not account for the case when the number of threads required to sort A and B is less than the number of available threads.

Chapter 6

Morton Code

6.1 Introduction

A Morton code (Morton, 1966) is a scalar integer code that represents a multidimensional position in space. The Morton code is guaranteed to preserve the locality of the points in space based on the Z-order of the points (Figure 6.1).

The Morton code can be considered a spatial *sort key* for the object it represents. If the Morton codes for all of the objects in a scene are sorted, the order of the objects in the sorted list will follow a space-filling curve called the Z-order curve (Figure 6.1). This type of space-filling curve is ideal for constructing the leaf nodes of a Bounding Volume Hierarchy (which is the subject of the next chapter).

The first step to finding the Morton code for the objects in the scene is to compute the minimum bounding volume that fully encloses all of the objects in the scene. The minimum bounding volume is used to normalize the positional components of the objects into the range $[0 \ldots 1]$.

With the bounding volume known, the Morton codes can be computed by normalizing the positional components of the scene objects into the range $[0 \ldots 1]$ then scaling them by $2^k - 1$ where k is the number of bits for each positional component. For example, to convert a 3D position into a $32\,\mathrm{bit}$ Morton code, the maximum value for k is 10 because each of the 3 components uses 10 bits in the Morton code for a total of 30 bits. In this case, two of the 32 bits would be unused.

The final Morton code is produced by interleaving the bits of the normalized and scaled positional values.

6.2 Minimum Bounding Volume

The first step to generating the Morton codes for the objects in the scene is to normalize the position of the objects in the scene into the range $[0 \ldots 1]$. This is accomplished by determining the minimum Axis-Aligned Bounding Box (AABB) that encloses all of the objects in the scene. The AABB over the objects is used to shift and scale the position of the objects so that the minimum point of the bounding volume is at $(0, 0, 0)$ and the maximum point of the bounding volume is at $(1, 1, 1)$.

In order to create the bounding box for the objects, the *parallel reduction* algorithm described in Section 4.2 is used to find the minimum and maximum points that encloses all of the objects in the scene. It is assumed that the AABB for a single object can be determined either directly or by deriving it from some properties of the object. For example, to compute the AABB for a triangle, the minimum and maximum of all three vertices is used. For more complex scene objects, it may be reasonable to store the AABB as a property of the object.

The method to reduce the AABB over the lights uses a 2-pass approach. In the first pass, A maximum of p thread groups are dispatched to reduce the n values. Each thread group produces a single value that is written to global memory. Each thread group reduces $n/(t \times p)$ values where n is the total number of values to be reduced and t is the number of threads per thread group.

In the first pass of the reduction, each thread reduces $n/(t \times p)$ objects serially and stores the reduced value in thread local storage (registers). The locally reduced AABB for the objects is written to group shared memory and the *LogStepReduction* function is invoked to perform the final reduction to produce a single value for the thread group. The pseudo code for the first pass of the parallel reduction is shown in Algorithm 6.1.

Algorithm 6.1 First pass of the parallel reduction.

```
Require: $O$ is a list of $n$ objects.
Require: $AABB$ is a list of AABBs in group shared memory.
Require: $gid$ is the index of the current thread group.
Require: $tid$ is the index of the thread within the thread group.
Require: $dtid$ is the index of the thread within the dispatch.
Require: $p$ is the number of thread groups.
Require: $t$ is the number of threads per thread group.
 1: function REDUCE1($gid$,$tid$)
 2:     $aabb_{min} \leftarrow FLT\_MAX$
 3:     $aabb_{max} \leftarrow -FLT\_MAX$
 4:     for $i \leftarrow dtid$ to $n$ step $(t \times p)$ do
 5:         $o \leftarrow O[i]$
 6:         $aabb_{min} \leftarrow$ MIN($aabb_{min}$,$o_{pos} - o_{range}$)
 7:         $aabb_{max} \leftarrow$ MAX($aabb_{max}$,$o_{pos} + o_{range}$)
 8:     end for
 9:     $AABB[tid] \leftarrow aabb$
10:     LOGSTEPREDUCTION($gid$,$tid$)
11: end function
```

The result of the first pass of the reduction is a maximum of p values, one value per thread group, reduced and stored in global memory. The second pass of the reduction dispatches only a single thread group of p threads to perform the final reduction. The algorithm for the second pass is similar to the first pass, the primary difference being that the resulting AABB's from the first pass are being used as input for the reduction instead of the objects position and range. The result of the second pass is a single value written to global memory that defines the AABB that encompasses all of the objects in the scene.

Given the minimum AABB that encloses all of the objects in the scene, the Morton codes for each object can be computed.

6.3 Compute Morton Codes

Morton code or Morton-order is a function which maps multi-dimensional data into 1-dimensional space while preserving the locality of the data points (Morton, 1966). For a k-bit Morton code, each component of the spatial coordinate is converted to its k-bit binary representation. The Morton code is computed by interleaving the k-bits of the coordinate components. The result is a dk-bit integer value, where d is the dimensionality of the coordinate space, that when sorted will arrange the points in the set according to the z-order of the points. The recursively z-shaped curve that is produced from the sorted Morton codes is shown in Figure 6.1.

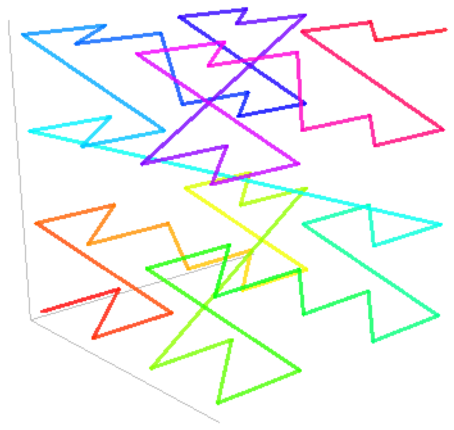

FIGURE 6.1: Z-order curve (Dickau, 2008).

Before the Morton code can be computed, the floating-point representation of the points in 3D space need to be converted to their integer representation. The floating-point components are converted to integers by first normalizing the points within the AABB of the data set and then scaling each component by $2^k - 1$. The bits of the resulting integer components are interleaved producing the final Morton code. Figure 6.2 shows an example of computing the Morton code for a 3-component vector.

Algorithm 6.2 shows the function to compute the k-bit Morton code m from a quantized value q. The Morton code is computed by combining the bits resulting by masking the b^{th} bit of the x, y, and z components of the quantized value q and shifting by s bits to the left.

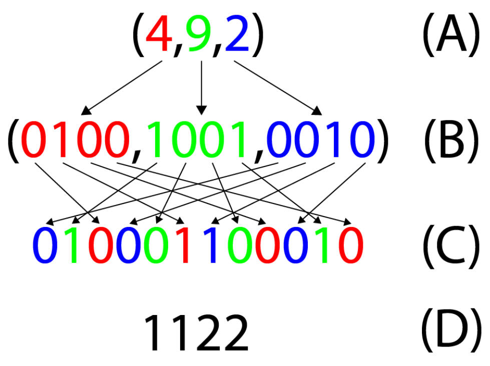

FIGURE 6.2: The integer representation of a coordinate in 3D space (A); The 4-bit binary representation of the coordinates (B); The result of interleaving the bits of the coordinate components (C); The resulting 12-bit Morton code in decimal representation (D).

Algorithm 6.2 Compute the k-bit Morton code for quantized coordinate c.

```
Require: $q$ is the quantized coordinate.
Require: $k$ is the number of bits of each coordinate.
Ensure: $m$ is the $dk$-bit Morton code.
 1: function MORTONCODE($q$,$k$)
 2:     $m \leftarrow 0$
 3:     $s \leftarrow 0$
 4:     $b \leftarrow 1$
 5:     while $b < 2^k$ do
 6:         $m \leftarrow m \vee$ SHIFTLEFT($q_x \wedge b$,$s + 0$)
 7:         $m \leftarrow m \vee$ SHIFTLEFT($q_y \wedge b$,$s + 1$)
 8:         $m \leftarrow m \vee$ SHIFTLEFT($q_z \wedge b$,$s + 2$)
 9:         $b \leftarrow$ SHIFTLEFT($b$,1)
10:         $s \leftarrow s + 2$
11:     end while
12:     return $m$
13: end function
```

Algorithm 6.3 shows the kernel function to compute the Morton codes for the lights in the scene. The light's position is normalized by shifting the light's position by the minimum point of the AABB and scaling it by the range of the AABB. The quantized position of the light q is computed by multiplying each component of the normalized light position by $2^k - 1$. The resulting quantized position is a 3-component vector where each component is in the range $[0 \cdots 2^k)$.

Algorithm 6.3 Compute k-bit Morton codes for the lights.

```
Require: $O$ is a list of $n$ objects.
Require: $AABB$ is the minimum AABB over the objects.
Require: $k$ is the number of bits of each coordinate of the Morton code.
Require: $dtid$ is the index of the thread within the dispatch.
Ensure: $M$ will contain the $n$ Morton codes for the objects.
1: function ComputeMortonCodes($dtid$,$k$)
2:     $r \leftarrow AABB_{max} - AABB_{min}$
3:     if $dtid < n$ then
4:         $o \leftarrow O[dtid]$
5:         $q \leftarrow \left(\frac{o_{pos} - AABB_{min}}{r}\right) \times (2^k - 1)$
6:         $M[dtid] \leftarrow$ MortonCode($q$,$k$)
7:     end if
8: end function
```

With the Morton codes generated for all of the objects in the scene, the next step is to sort the Morton codes using the hybrid sorting technique described in Chapter 5. After the Morton codes have been sorted, the BVH over the scene objects can be constructed. BVH construction and traversal is the subject of Chapter 7.

Chapter 7

Bounding Volume Hierarchy

7.1 Introduction

A *Bounding Volume Hierarchy* (BVH) is a tree-like data structure that allows for quickly determining if two or more scene primitives are overlapping with each other. The leaf nodes of the BVH are constructed by considering the smallest primitive in the scene. For physics optimizations, the smallest primitive may be the physics objects in the scene, or for ray-tracing application, the smallest primitive would be the individual triangles that compose the geometry in the scene. The Axis-Aligned Bounding Box (AABB) that minimally encloses the primitive is determined by taking the primitives's minimum and maximum points in space and the AABB is used to construct the leaf nodes of the BVH. The upper nodes of the BVH are constructed by taking the leaf nodes and building an AABB that minimally encloses the child nodes. A BVH can be constructed by taking 2 or more leaf nodes to construct the upper nodes of the BVH. The number of child nodes used to construct the upper nodes of the BVH is called the *degree* of the BVH. An example of a 2-degree BVH is shown in Figure 7.1.

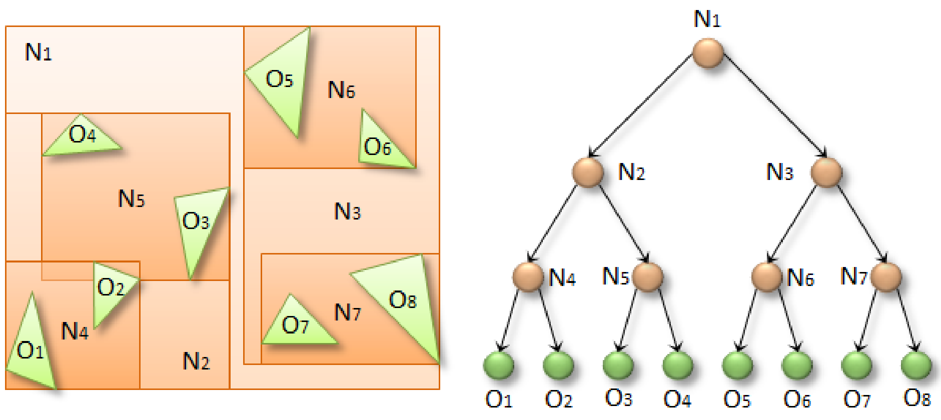

FIGURE 7.1: A Bounding Volume Hierarchy built over several primitives in 2D space (Karras, 2012).

This chapter describes the construction and traversal of a 32-degree BVH. A 32-degree BVH was chosen so that the BVH can be constructed and traversed by 32 threads using warp-synchronous lock-step traversal. This eliminates the need for thread group synchronization barriers during BVH construction and traversal providing an opportunity to exploit a performance improvement.

7.2 BVH Construction

The method to construct the BVH described here uses a bottom-up approach. The leaf nodes of the BVH are implicitly derived from the AABB of the scene primitives and are not directly stored in the BVH. The AABB of the primitives are used to construct the lowest level child nodes of the BVH. To construct the child nodes, 32 consecutive primitives are read from the sorted list. In order to ensure spatial locality, the primitives are sorted using their Morton code using the technique described in Chapter 6 and sorting is described in Chapter 5.

The BVH is constructed in two phases. Two phases are required to construct the BVH because the results of the first phase are written to global memory. The only way to synchronize writes to global memory that occur across thread group boundaries is by invoking a separate dispatch (as described in Chapter 3).

In the first phase, the AABBs for the lowest level child nodes of the BVH are computed. This is done by reading the AABB of 32 primitives and reducing them to a single AABB that encloses all 32 primitives.

In the second phase of the BVH construction, the upper nodes of the BVH are built. The upper nodes of the BVH are constructed in a similar manner to the child nodes. The primary difference between these two phases is the source of the AABB in the first phase is derived from the scene primitives and in the second phase the AABBs are derived from the lower child nodes of the BVH. Figure 7.2 shows an example of the two phases that are required to construct the BVH.

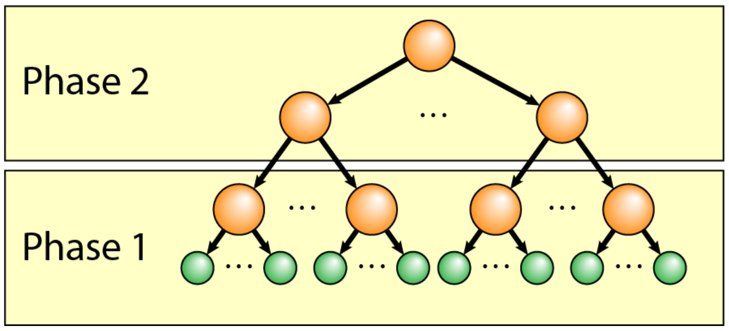

FIGURE 7.2: In the first phase of the BVH construction, the AABB of the child nodes of the last level of the BVH tree are computed from the sorted scene primitives. In the second phase of the BVH construction, the upper nodes are computed.

7.2.1 Build Leaf Nodes

The leaf nodes of the BVH are constructed by taking 32 primitives from the sorted list and building the AABB over the primitives. The AABB is then written to the first child at the lowest level of the BVH tree. The process continues for the next set of 32 primitives until all of the primitives have been processed. Since the leaf nodes of the BVH are the AABB of the primitives themselves they are not explicitly stored in the BVH. The indices of the primitives in the sorted list can be derived from the ID of the child nodes.

The choice of 32 primitives for each node of the BVH was chosen so that the AABB for the primitives can be computed in warp-synchronous lock-step. Each thread group of the compute shader computes 32 child nodes of the BVH and there is no need to perform thread group synchronization barriers.

Each thread reads a single AABB from the sorted list and stores the AABB in group shared memory. A parallel log-step reduction (Section 4.2) is performed over the 32 AABBs using the warp-synchronous optimization shown in Algorithm 4.2. The first thread of each warp writes the reduced AABB to the child node of the BVH. The pseudo-code for the construction of the leaf nodes of the BVH is shown in Algorithm 7.1.

Algorithm 7.1 Build the leaf nodes of the BVH.

```
Require: $O$ is a sorted list of $n$ primitives.
Require: $tid$ is the thread ID in the thread group.
Require: $dtid$ is the thread ID in the dispatch.
Require: $AABB$ is the AABB of the primitives stored in group shared
    memory.
Ensure: $BVH$ is the BVH data structure stored in global memory.
1: function BUILDLEAFNODES($O$,$dtid$,$tid$)
2:     $o \leftarrow O[dtid]$
3:     $AABB[tid] \leftarrow o_{AABB}$
4:     LOGSTEPREDUCTION($AABB$,$tid$)
5:     if MOD($tid$,32)$= 0$ then
6:         $i \leftarrow$ FIRSTCHILDINDEX($n$) $+\frac{dtid}{32}$
7:         $BVH[i] = AABB[tid]$
8:     end if
9: end function
```

In the next phase of the BVH construction, the upper nodes of the BVH are constructed.

7.2.2 Build Upper Nodes

The algorithm to build the upper nodes of the BVH is very similar to that of the leaf nodes shown in Algorithm 7.1. The primary difference being that the AABB of the leaf nodes are read directly from the BVH instead of being constructed from the scene primitives. The compute shader to compute the AABBs for the upper nodes is invoked for each level of the BVH above the leaf nodes. This means that the compute shader to compute the upper levels of the BVH will be invoked at most $\lceil \log_{32}(n) \rceil - 1$ times. If the AABB over the scene primitives that was used to compute the Morton codes as described in Chapter 6 is available, then it can be used directly for the root node of the BVH and does not need to be reconstructed.

In the next section, the traversal of the BVH is described.

7.3 BVH Traversal

BVH traversal is used to quickly determine which leaf nodes of the BVH are contained within a certain area of the scene. For example, if the scene was split into a regular grid (a voxel grid) then an overlap test is performed with each node of the BVH against the AABB for the cell of the voxel grid. Any leaves of the BVH that are contained in the child node of the BVH that overlaps the AABB of the cell of the voxel grid is considered to be contained within the cell. Optionally, each leaf node can also be checked for intersection against the voxel cell to achieve a more refined result. In this section, the term *cell* is used to refer to the area in the scene that is being checked for overlap with the nodes of the BVH.

The method to traverse the BVH uses a stack to push the index of the child node in the BVH if the AABB of the child node overlaps with the cell. The technique to traverse the BVH was inspired by Tero Karra (Karras, 2012).

A thread group of 32 threads is dispatched for each cell in the scene. Each thread checks a child node of the BVH. If the AABB of the BVH node overlaps with the AABB of the cell, the index of the BVH node is pushed onto the stack. If the traversal reaches the leaf nodes, then the AABB of the primitives are checked if they intersect with the AABB of the cell. The first thread of each warp pops a node off the stack and the process continues until the stack is empty.

Algorithm 7.2 Traverse the BVH and append overlapping lights to the light list.

```
Require: $O$ is a sorted list of $n$ primitives.
Require: $pIdx$ is the index of the parent node in the BVH.
Require: $tid$ is the thread ID in the thread group.
Require: $AABB$ is the AABB of the current cell.
 1: function TRAVERSEBVH($O$,$tid$)
 2:     if $tid = 0$ then
 3:         $pIdx \leftarrow 0$
 4:         PUSHNODE(0)
 5:     end if
 6:     repeat
 7:         $i \leftarrow$ FIRSTCHILDINDEX($pIdx$) + $tid$
 8:         if ISLEAFNODE($i$) then
 9:             $oIdx \leftarrow$ GETINDEXOFPRIMITIVE($i$)
10:             $o \leftarrow O[oIdx]$
11:             if AABBINTERSECTAABB($o_{AABB}$,$AABB$) then
12:                 APPENDTOLIST($o$)
13:             end if
14:         else if AABBINTERSECTAABB($AABB$,$BVH[i]$) then
15:             PUSHNODE($i$)
16:         end if
17:         if $tid = 0$ then
18:             $pIdx \leftarrow$ POPNODE
19:         end if
20:     until $pIdx = 0$
21: end function
```

Algorithm 7.2 shows the traversal of the BVH. First the index of the root node (index 0) is pushed onto the stack. It isn't necessary to check the intersection of the root node's AABB with the AABB of the cell because it is just as fast to check the AABB of all 32 child nodes. If none of the first child nodes overlap with the AABB of the volume tile then neither does the root and traversal will end.

If the traversal reaches a leaf node, the index of the primitive is computed from the index of the child node and the intersection test is performed against the AABB of the primitive. If the AABB of the primitive overlaps with the AABB of the cell, it is appended to a list of primitives for that cell.

The result of the BVH traversal is a list of primitive ID's for each cell in scene. The resulting list can then be used in a later stage of the technique to optimize expensive operations (for example, collision detection, ray tracing, or shading).

Chapter 8 describes the implementation of the Volume Tiled Forward Shading technique. An optimized version of the technique uses the BVH construction method described in this chapter to build a BVH over the dynamic lights in the scene. The resulting light list is used to minimize the number of lights that must be considered during the shading pass of the Volume Tiled Forward Shading technique.

Chapter 8

Implementation

8.1 Introduction

In this chapter, the implementation of the Volume Tiled Forward Shading technique is described. Volume Tiled Forward shading builds upon the Tiled Forward Shading technique (Olsson and Assarsson, 2011) by dividing the 2D screen space tiles into 3D volume tiles. Volume Tiled Forward Shading uses the same log-space partitioning method used to divide clusters used in the Clustered Shading technique (Olsson, Billeter, and Assarsson, 2012) but unlike the Clustered Shading technique described by Olsson, Billeter, and Assarsson, Volume Tiled Forward Shading provides native support for transparent geometry.

Similar to Tiled and Clustered Shading, Volume Tiled Forward Shading is a rendering technique that minimizes the number of lights that must be considered during shading. It accomplishes this by first assigning the lights in the scene to the 3D volume tiles by performing an Axis-Aligned Bounding Box (AABB) test against the lights and the volume tile. Only lights that are contained within the same volume tile as the current fragment need to be considered during shading.

Assigning the lights to the volume tiles using a brute-force approach reveals a performance bottleneck in the Volume Tiled Shading technique. In order to alleviate the performance bottleneck of the light assignment phase of the Volume Tiled Forward Shading technique, a Bounding Volume Hierarchy (BVH) can be constructed over the lights in the scene. During the light assignment phase, the BVH is traversed to find the subset of lights overlapping the current volume tile. Only the lights that are contained within the nodes of the BVH that are overlapping with the volume tile need to be checked for intersection with the volume tile.

8.2 Volume Tiled Forward Shading

The Volume Tile Forward Shading technique consists of two main phases:

1. Initialization
2. Update

In the initialization phase, the AABBs for the volume tiles are generated. In this phase, the size of the volume grid is computed and based on the subdivision of the volume grid, the AABBs for each tile are computed.

The initialization phase is executed when the application is started or if the dimensions of the framebuffer are changed, for example if the screen size is changed. Changing the field of view of the camera will also require the volume gird to be rebuilt since the number of subdivisions in the depth is dependent on the field of view of the camera. The purpose of the initialization phase is to compute the dimensions of the grid and to precompute an AABB for each volume tile.

In the update phase, the light assignment to volume tiles is made. This phase uses the AABBs of the volume tiles computed in the initialization phase to perform intersection tests between the lights in the scene and the volume tiles. During the shading pass, only the lights that are contained within the volume tile of the current fragment need to be considered.

The update phase is executed each frame that either the camera moves, an object in the scene changes, or the position or size of a light in the scene is modified. Since any of these events can occur during a frame, the update phase is executed each frame regardless of the previous state of the scene.

8.2.1 Initialization

The purpose of the initialization phase of the Volume Tiled Forward Shading technique is to compute the size of the volume grid and to compute the AABBs for each volume tile.

Compute Grid Size

Volume Tiled Forward Shading uses a 3D grid of AABBs. Since the AABBs are stored in view space, it is only necessary to build the volume grid when the screen is resized or the field of view of the camera is changed.

The dimensions of the volume grid are based on the screen dimensions and the width and height of a tile of the volume grid in screen pixels. For a given tile size (t_x, t_y) and screen dimensions (w, h), the number of subdivisions in the volume grid (S_x, S_y) is computed as

$$(S_x, S_y) = \left(\left\lceil \frac{w}{t_x} \right\rceil, \left\lceil \frac{h}{t_y} \right\rceil \right) \tag{8.1}$$

The number of subdivisions in the depth of the volume grid is dependent on the vertical field of view of the camera (2θ), the distance to the near clipping plane (z_{near}), the distance to the far clipping plane (z_{far}), and the number of vertical subdivisions (S_y).

$$S_z = \left\lfloor \frac{\log\left(z_{far}/z_{near}\right)}{\log\left(1 + \frac{2\tan\theta}{S_y}\right)} \right\rfloor \tag{8.2}$$

Compute AABBs

With the size of the volume grid known, the AABB for each volume tile is computed. Algorithm 8.1 describes how to compute the AABB for the volume tile at index (i, j, k).

On lines 2, and 3 of Algorithm 8.1, the distance in view space to the planes of the volume tiles at index k and $k+1$ is computed.

On lines 4, and 5 the screen space points of the top-left and bottom-right corners of the volume tile are computed. On lines 6, and 7 the screen space points are converted to view space.

Algorithm 8.1 Compute Volume Tile AABBs

```
Require: $S_x$ is the number of subdivisions in the width.
Require: $S_y$ is the number of subdivisions in the height.
Require: $z_{near}$ is the distance to the near clipping plane.
Require: $z_{far}$ is the distance to the far clipping plane.
Ensure: $AABB$ will contain the AABB for volume tile at index $(i, j, k)$.
 1: function COMPUTEAABB($i$,$j$,$k$)
 2:     $k_{near} \leftarrow z_{near}\left(1 + \frac{2\tan\theta}{S_y}\right)^{k}$
 3:     $k_{far} \leftarrow z_{near}\left(1 + \frac{2\tan\theta}{S_y}\right)^{k+1}$
 4:     $p_{min} \leftarrow \{i \cdot S_x, j \cdot S_y, 1\}$
 5:     $p_{max} \leftarrow \{(i+1) \cdot S_x, (j+1) \cdot S_y, 1\}$
 6:     $p_{min} \leftarrow$ SCREENTOVIEW($p_{min}$)
 7:     $p_{max} \leftarrow$ SCREENTOVIEW($p_{max}$)
 8:     $p_{min,near} \leftarrow$ LINEINTERSECTPLANE($\mathbf{0}$,$p_{min}$,$k_{near}$)
 9:     $p_{min,far} \leftarrow$ LINEINTERSECTPLANE($\mathbf{0}$,$p_{min}$,$k_{far}$)
10:     $p_{max,near} \leftarrow$ LINEINTERSECTPLANE($\mathbf{0}$,$p_{max}$,$k_{near}$)
11:     $p_{max,far} \leftarrow$ LINEINTERSECTPLANE($\mathbf{0}$,$p_{max}$,$k_{far}$)
12:     $AABB_{min} \leftarrow$ MIN($p_{min,near}$,$p_{min,far}$,$p_{max,near}$,$p_{max,far}$)
13:     $AABB_{max} \leftarrow$ MAX($p_{min,near}$,$p_{min,far}$,$p_{max,near}$,$p_{max,far}$)
14: end function
```

On lines 8-11, the four corner points of the AABB are computed by performing a line-plane intersection test.

The minimum point of the AABB is the minimum of all of the corner points of the AABB and the maximum point of the AABB is the maximum of all of the corner points of the AABB. This will result in neighbouring AABBs overlapping slightly but this is intended behaviour because the volume created by the volume tiles are not perfect cubes but actually more resembling the shape of a frustum. The AABB for the volume tile is the minimum bound box that encloses that frustum (see Figure 8.1).

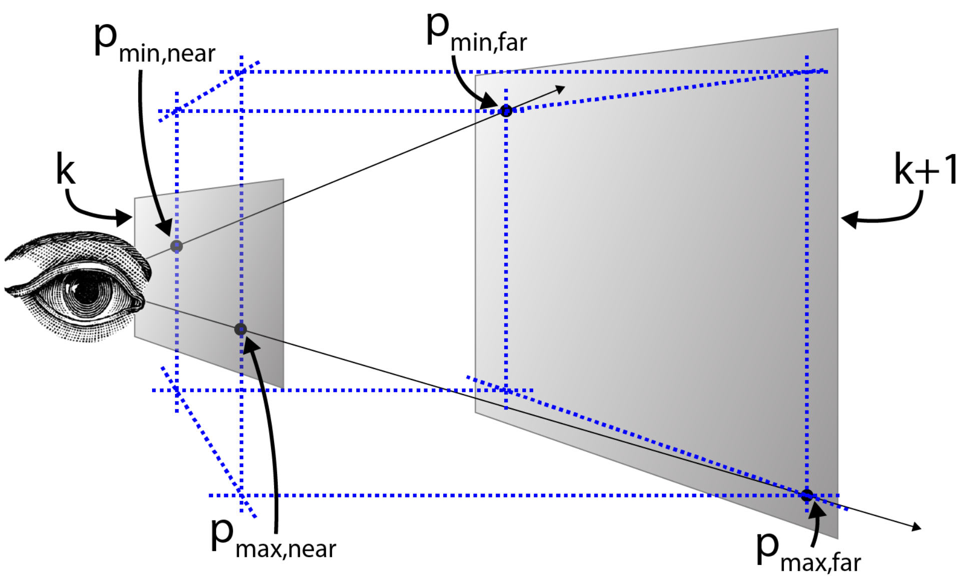

FIGURE 8.1: The AABB for the volume tile.

8.2.2 Update

The update phase of the Volume Tiled Rendering technique is performed each frame that the scene needs to be rendered. The main purpose of the update phase is to assign the lights in the scene to the volume tiles. It is important that the light assignment is only computed for volume tiles that actually contain a visible fragment in 3D space. Only fragments that are not occluded by opaque geometry should be activated. To ensure that only the volume tiles that contain visible fragments that are not occluded by opaque geometry are activated, a depth pre-pass is executed by rendering the opaque objects in the scene and updating the depth buffer. The active tiles are then determined by rendering both opaque and transparent objects in the scene and activating the tiles that contain a fragment during rendering.

The light assignment phase is executed for all active volume tiles. Using a naive approach, each active volume tile performs a brute-force search by testing every light in the scene for intersection. An optimization for the light assignment pass is described in Section 8.2.3.

The update phase consists of several passes:

1. Depth pre-pass
2. Mark active tiles
3. Build tile list
4. Assign lights to tiles
5. Shade samples

The depth pre-pass ensures all opaque geometry is recorded into the depth buffer so that only volume tiles containing visible fragments are activated in the next pass.

In the Mark active tiles pass, both opaque and transparent geometry are rendered. Any volume tile that contains a visible sample is flagged in the pixel shader. By using the depth buffer from the depth pre-pass, and relying on the GPU's ability to perform early depth testing, no volume tiles that contain overdrawn samples will be flagged in this pass.

After all of the volume tiles containing visible samples have been flagged, a list of unique volume tiles is created.

In the light assignment pass, the light index list and light grid are generated by intersecting the the lights in the scene against the active volume tiles. Each light that is partially contained in an active volume tile is added to the light list for that volume tile.

The opaque and transparent rendering passes are performed in the same way as tiled rendering. Only the lights that are contained within the same volume tile as the sample is considered during shading.

Depth Pre-pass

The purpose of the depth pre-pass is to record all of the opaque objects to the depth buffer. This ensures that the pixel shader for the next pass is only invoked for visible samples. Any samples that will be overdrawn by samples closer to the viewer will not cause the volume tile to be activated in the next pass. The result of the depth pre-pass is shown in Figure 8.2.

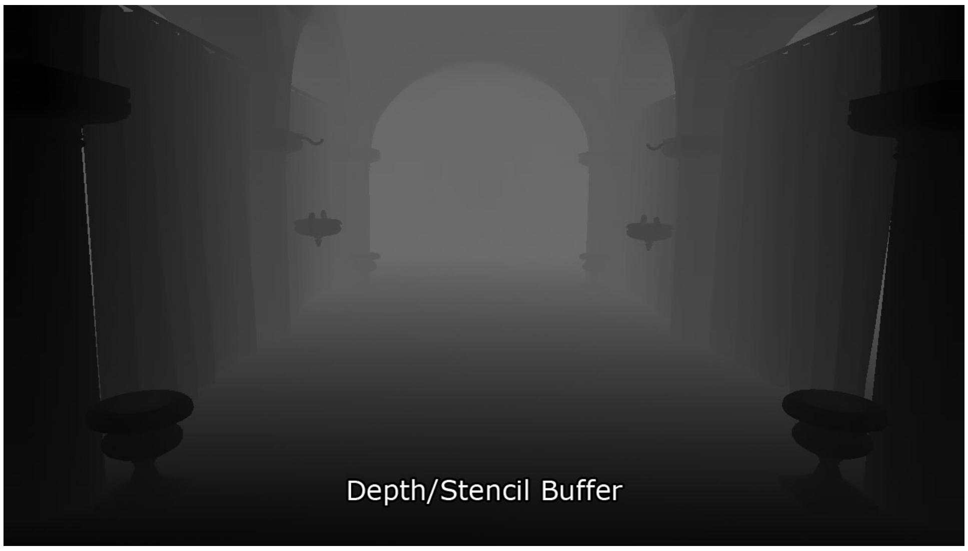

FIGURE 8.2: The result of the depth pre-pass.

Mark Active Tiles

After the depth pre-pass, the scene is rendered again but instead of rendering only opaque objects, both opaque and transparent geometry are rendered.

The pixel shader for this pass is very simple. First, the index of the volume tile is computed from the pixel's screen space position and the view space depth. Then the volume tile for the pixel is marked as active. To mark the current volume tile as active, a list of boolean flags is updated in the pixel shader. Each entry of the list represents one volume tile in the grid. The algorithm for the pixel shader is shown in Algorithm 8.2.

Algorithm 8.2 Mark Active Tiles

```
Require: $x$ is the pixel's x coordinate in screen space.
Require: $y$ is the pixel's y coordinate in screen space.
Require: $z$ is the pixel's z coordinate in view space.
Require: $TileFlags$ is a list of boolean flags for each tile.
1: function MARKTILE($x$,$y$,$z$)
2:     $i \leftarrow$ COMPUTETILEINDEX($x$,$y$,$z$)
3:     $TileFlags[i] \leftarrow true$
4: end function
```

Build Tile List

The sparse list of active tiles generated from the previous pass needs to be compressed into a dense list of unique tile indices. To generate the dense list of tile indices, a compute shader is invoked over the list of tile flags. Each thread of the compute shader checks a single entry in the tile flags array. If the tile was flagged (true) in the previous pass, the index of that tile is written to a list of unique tile ID's. The algorithm for the compute shader to build the unique list of tile ID's is shown in Algorithm 8.3.

Algorithm 8.3 Build Tile List

```
Require: $tid$ is the ID of the thread in the dispatch.
Require: $TileFlags$ is a sparse list of boolean flags for each tile.
Ensure: $ActiveTiles$ is a dense list of active volume tile ID's.
1: function BUILDTILELIST($tid$)
2:     if $TileFlags[tid]$ then
3:         ATOMICAPPEND($ActiveTiles$,$tid$)
4:     end if
5: end function
```

The result of the *Build Tile List* pass is a dense list of active volume tile ID's. This is similar to the result of the *Find Unique Clusters* pass of the *Clustered Shading* technique described by Olsson et al. but does not require the sorting and compaction steps described in their paper (Olsson, Billeter, and Assarsson, 2012).

Assign Lights to Tiles

The *Assign Lights to Tiles* pass is similar to the *Light Culling* pass of the *Tiled Forward Shading* technique described in Section 2.1.3. A compute shader is executed over the active volume tiles, one thread group per tile. For each active volume tile, all of the lights in the scene are tested against the AABB of the volume tile. Lights that intersect with the AABB of the volume tile are added to a local light index list. After all of the active scene lights have been tested, the local light index list is copied to a global light index list. The algorithm to assign lights to the volume tiles is shown in Algorithm 8.4.

Algorithm 8.4 Assign lights to tiles.

```
Require: $L$ is a list of $n$ lights.
Require: $C$ is the current index in the global light index list.
Require: $I$ is the global light index list.
Require: $G$ is the 3D grid storing the light count and offset into the global
    light index list.
Require: $gid$ is the 3D index of the current thread group.
Ensure: $G$ is updated with the offset and light count of the current tile.
 1: function CULLLIGHTS($gid$)
 2:     $i \leftarrow \{0\}$
 3:     $AABB \leftarrow TileAABB[gid]$
 4:     for $l$ in $L$ do
 5:         if SPHEREINTERSECTAABB( $l$, $AABB$ ) then
 6:             APPENDLIGHT($l, i$)
 7:         end if
 8:     end for
 9:     $c \leftarrow$ ATOMICINC($C, i.count$)
10:     $G(gid) \leftarrow (c, i.count)$
11:     $I(c) \leftarrow i$
12: end function
```

A light grid is used to store the light count and offset into the global light index list for each volume tile. This data structure is identical to that used by the *Tiled Forward Shading* technique but the light grid for the *Volume Tiled Forward Shading* technique can be visualized as a 3D light grid instead of a 2D light grid. A single slice of the volume light grid is shown in Figure 8.3.

Shade Samples

The shading pass of the *Volume Tiled Forward Shading* technique is similar to that of the *Tiled Forward Shading* technique. The tile ID for the current sample is computed from the x and y screen space position and the view space depth (z) of the sample. The light count and the offset into the global light index list is retrieved from the volume light grid that was generated in the previous pass and only the lights that overlap the current volume tile are used to compute the final shading for the sample.

This process is identical for both the opaque and transparent passes. The same volume tile light grid is used for both opaque and transparent geometry since any volume tile that contained a sample was activated in the *Mark Active Tiles* pass of the technique.

An example of the shaded scene is shown in Figure 8.4.

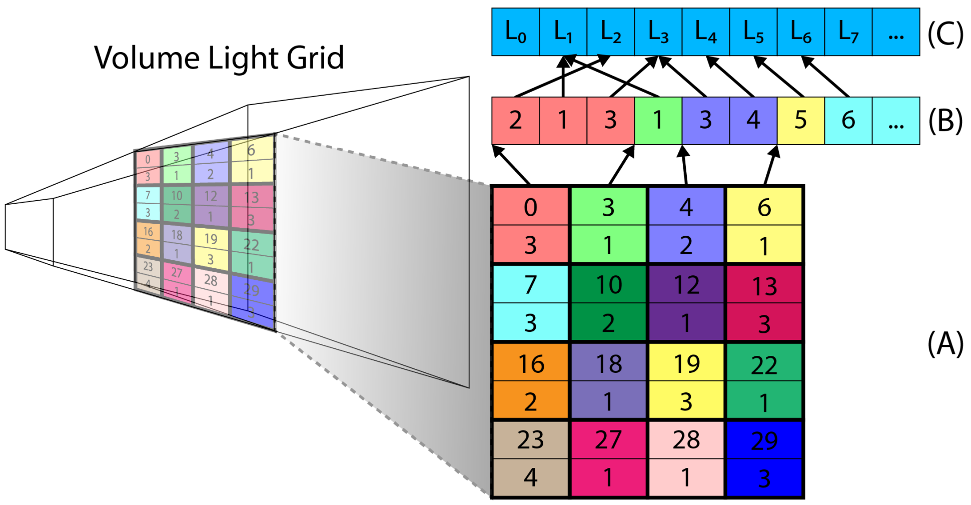

FIGURE 8.3: A single slice of the volume light grid (A). The light grid stores the light count and an offset into the light index list (B). The light index list stores the index of the light source in the light list (C).

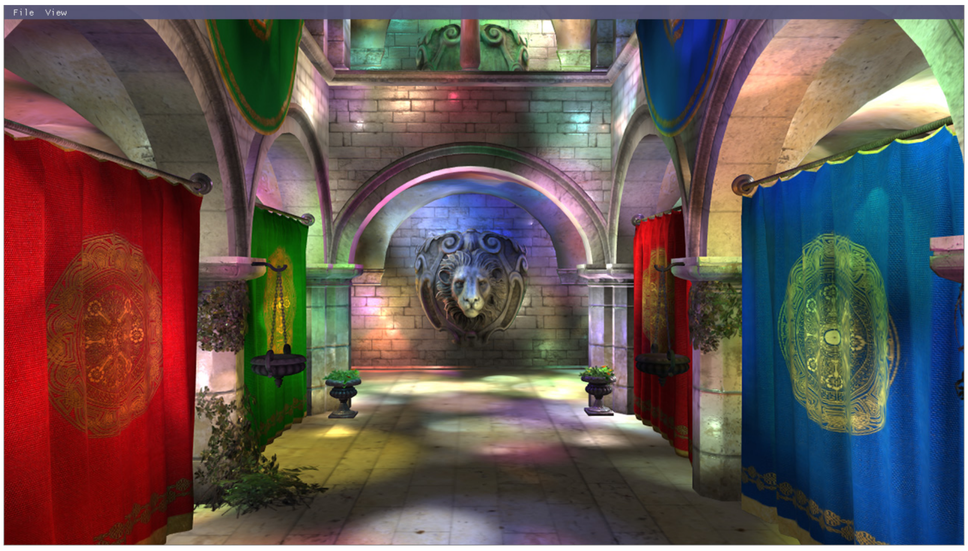

FIGURE 8.4: The Crytek Sponza scene rendered using *Volume Tiled Forward Shading*.

8.2.3 Optimization

The technique described in this section is the basic technique that implements *Volume Tiled Forward Shading*. The performance profiling results will be shown later in Chapter 10 where the performance of the *Assign Lights to Tiles* pass will show a performance bottleneck limiting the maximum number of lights that can be used in the scene.

To mitigate the performance overhead of the *Assign Lights to Tiles* pass, a *Bounding Volume Hierarchy* (BVH) can be built over the lights in the scene according to the technique described in Chapter 7. Using a BVH greatly improves the light assignment pass of the *Volume Tiled Forward Shading* technique because instead of performing a brute-force check for every light in the scene against the AABB of the volume tile, only the lights that are contained in the BVH node that also overlap with the volume tile need to be checked. This effectively reduces the asymptotic running time of the *Assign Lights to Tiles* pass from $\mathcal{O}(mn)$ to $\mathcal{O}(m \log_{32} n)$ where m is the number of active volume tiles and n is the number of active lights in the scene.

8.3 Summary

In this chapter the steps of the *Volume Tiled Forward Shading* technique were described. In the first pass, a depth pre-pass was performed by rendering only the opaque objects in the scene. In the next pass, both opaque and transparent objects are rendered using the depth buffer from the previous pass to ensure only visible samples are rendered. For each visible sample, the volume tile that contains the sample is activated. In order to generate a dense list of active tile IDs, a compute shader pass is executed that writes the active tile IDs to a contiguous list. The lights in the scene are then checked against the active volume tiles and a list of overlapping lights for each active tile is stored in a global light index list. The global light index list is then used to perform final shading.

An optimization technique using a BVH over the lights in the scene is described. Using the BVH optimization greatly improves the performance of the *Assign Lights to Tiles* pass as will be shown in Chapter 10.

Chapter 9

Experiment Setup

9.1 Introduction

This chapter describes how the experiment is created and how the performance results are collected. How the application is created, what graphics API and the GPU hardware that is used, the scenes that are used for rendering, and which tests are conducted are all important factors to understand when analyzing the performance data.

9.2 Application

Since graphics rendering is a highly demanding task for the GPU, a lot of consideration for performance and optimization was made. The application was written in C++ using some features of the C++11 standard, such as the multi-threading and async-tasks libraries were used to ensure the rendering thread could run without interference from the main thread where the windows message loop was being handled.

9.3 Graphics API

The graphics engine was created using the DirectX 12 graphics API. DirectX 12 has several advantages over previous DirectX APIs such as DirectX 11. One major advantage of DirectX 12 is its ability to bind more than eight *Uniform Access Views* (UAV). The *Light Culling* computer shader requires 13 UAVs to be simultaneously bound. The disadvantage of using DirectX 12 is that it is only working with the Microsoft Windows operating system restricting the platform that the experiment can be run on to Windows. The Vulkan API would have been a possible alternative however the Vulkan API was not yet available to the public when the research for this paper was started.

9.4 GPU Hardware

In order to capture the rendering performance of the various techniques, an *NVidia GeForce GTX TITAN X* was used for all experiments. This GPU was chosen to capture the performance analysis because NVidia was kind enough to donate it for the purpose of these experiments. No other GPU hardware was used since the most important conclusion that can be drawn from the performance statistics are the relative performance difference between various rendering techniques. Performance scaling across various

GPU types can be estimated if the relative performance characteristics of the GPUs are known. As mentioned in Section 9.3 the experiment utilizes the DirectX 12 rendering API in order to implement the experiment. This implies that the GPU used to test the experiment must have support for this rendering API.

9.5 Scenes

Several scene files were acquired from Morgan McGuire's computer graphics archive of meshes (McGuire, 2017). Among these scene files is the *Sponza Atrium* scene which is shown in Figure 9.1. This scene was originally created by Marko Dabrovic in 2002 (Crytek, 2010) and quickly became a popular scene for use in demonstrating rendering algorithms. This scene has become popular because it is an elegant scene that contains vibrant colors and the open ceiling provides a realistic scene for demonstrating environment lighting effects and global illumination effects. The Sponza scene was chosen for this experiment not only for its visual appeal but also because it seems representative of a typical scene that may be found in a modern video game.

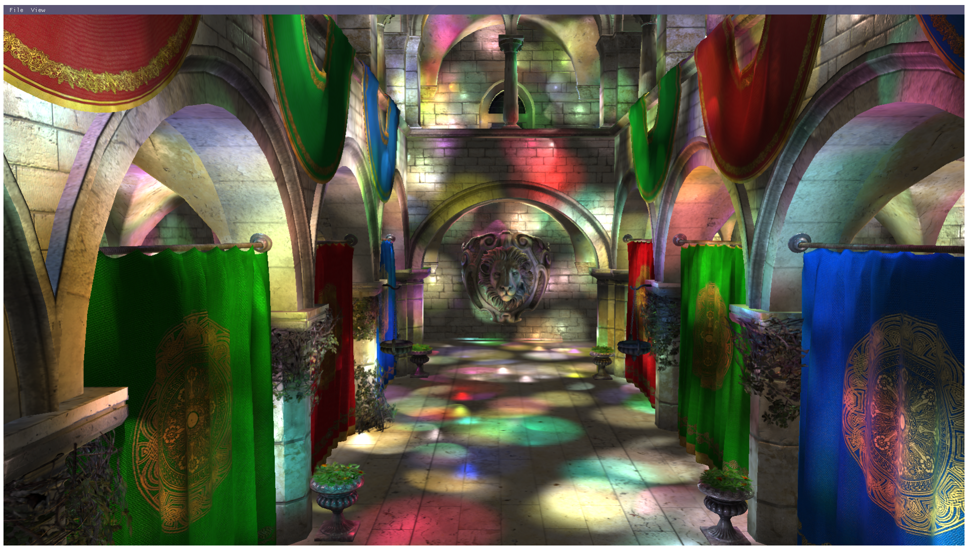

FIGURE 9.1: The Sponza Atrium scene (Crytek, 2010)

The *San Miguel* scene shown in Figure 9.2 was originally created by Guillermo M. Leal Llaguno and is based on a hacienda that he visited in San Miguel de Allende, Mexico (McGuire, 2017). This scene was chosen as a test scene because of the large number of transparent geometry in the scene activates a lot of volume tiles in the scene and pushes the limits of the *Volume Tiled Forward Shading* algorithm. Similar to the Sponza Atrium scene, the San Miguel scene is also representative of a scene that may be found in a modern video game.

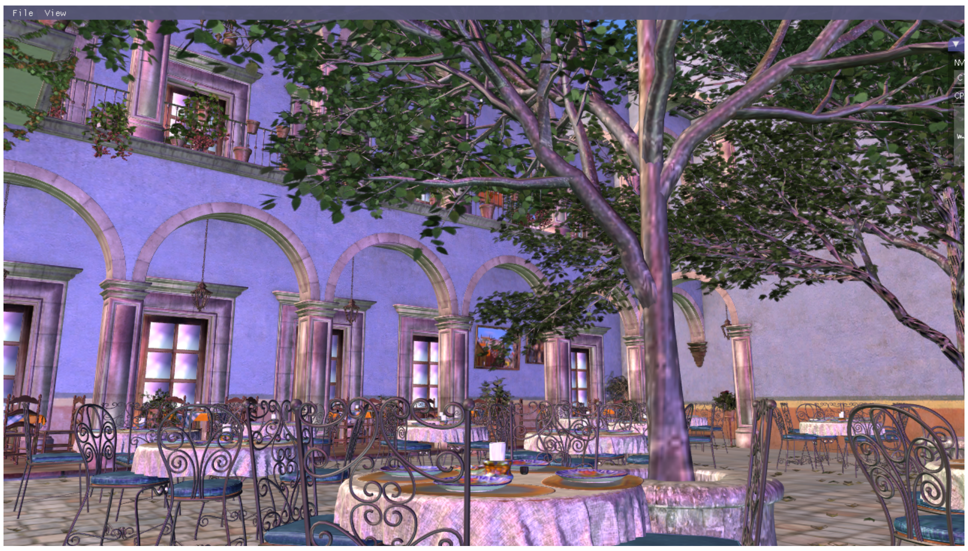

FIGURE 9.2: The San Migule hacienda (McGuire, 2017)

9.6 Algorithms

Several rendering algorithms were implemented for this experiment. *Forward Rendering* was implemented in order to establish a ground-truth representation of the rendered scene and to be used as a benchmark to determine the expected rendering quality of the test scene. A performance analysis of Forward Rendering is provided in Chapter 10 and establishes a performance benchmark that can be used to determine a relative performance improvement when compared to other rendering algorithms.

Tiled Forward Shading described in Chapter 2 was also implemented since this lighting algorithm forms the basis of the *Volume Tiled Forward Shading* algorithm.

Two versions of the *Volume Tiled Forward Shading* algorithm were implemented in the experiment. The first version implements a naive approach to the *Volume Tiled Forward Shading* technique that is described in Chapter 8. The second version creates a *Bounding Volume Hierarchy* (BVH) over the lights in the scene in order to improve the performance of the *Assign Lights to Tiles* pass of the *Volume Tiled Forward Shading* technique.

9.7 Profiling

GPU profiling data is captured using GPU timestamp queries in real-time while the application was running. No external profiling tools are used to capture the performance information. A timestamp query is performed at the beginning of a block of passes of the rendering technique and again at the end of the block of passes. The number of *ticks* between the two timestamps is computed and converted to milliseconds for visualization purposes.

9.8 Tests

For each rendering algorithm described in Section 9.6 the scene is rendered with an increasing number of lights. The scene is rendered from a stationary camera position while the profiler captures a minimum of 500 frames and the average of the timings is read from the statistic data.

Initial tests are executed with an increasing number of lights while maintaining a constant volume wherein the lights are randomly placed. This results in the density of the lights within the scene to increase linearly which has a proportional impact on the performance of the opaque and transparent rendering passes. The performance results deceivingly show poor rendering performance due to the overhead caused by the opaque and transparent rendering passes. Even if only a single pixel in the scene needs to consider one thousand lights, the performance of the entire rendering algorithm will suffer.

Since an extremely high light density is not representative of a typical scenario, the experiments were executed again with a constant light density of 1 $light/unit^3$. The area in which the lights are randomly placed was increased to ensure a random distribution with a density of 1 $light/unit^3$. This resulted in more accurate representation of the performance results of the Tiled and Volume Tiled Forward Shading techniques. The performance of the Forward Shading technique is not sensitive to the density of the lights in the scene since that technique always considers every light in the scene for every rasterized pixel.

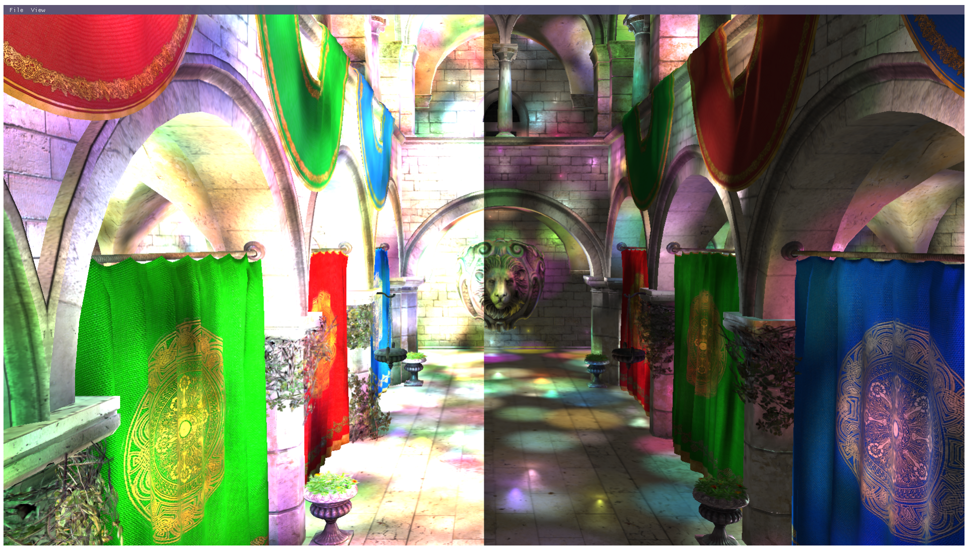

FIGURE 9.3: The scene contains 65,536 lights. The image on the left shows an average light density of 4.85 $light/unit^3$ while the image on the right shows an average light density of 1 $light/unit^3$.

Chapter 10

Results

10.1 Introduction

In this chapter, the performance of the various rendering technique described in Section 9.6 is analysed. The performance characteristics of traditional *Forward Rendering* described in Section 2.1.1, *Tiled Forward Shading* described in Section 2.1.3, *Volume Tiled Forward Shading* described in Section 8.2, and *Volume Tiled Forward Shading with BVH Optimisation* described in Section 8.2.3 are compared.

The performance results for the tests described in Chapter 9 are shown and the performance results are analysed. All tests are performed at 1920x1080 resolution using NVidia GTX Titan X GPU. Timings shown are in milliseconds unless otherwise stated.

10.2 Forward Rendering

The *Forward Rendering* technique was run with both the *Sponza Atrium* (Crytek, 2010) scene and the *San Miguel* (McGuire, 2017) scene mentioned in Section 9.5. Since Forward Rendering is not sensitive to the light density of the scene, the experiment was only executed with an increasing number of lights but the area in which the lights were spawned was not adjusted resulting in an increasing light density.

The following passes of the *Forward Rendering* technique were captured:

1. Depth Prepass
2. Opaque Pass
3. Transparent Pass
4. Total Rendering Time

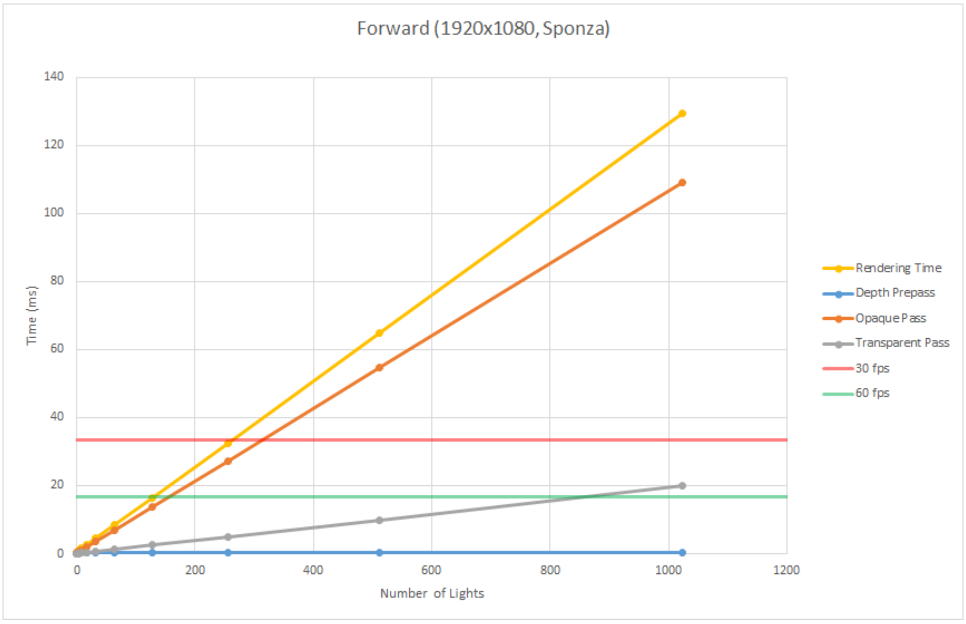

FIGURE 10.1: Sponza Atrium scene using *Forward Rendering*.

TABLE 10.1: Timings for rendering the Sponza Atrium scene using Forward Rendering.

Num Lights	Depth Prepass	Opaque Pass	Transparent Pass	Frame Time
0	0.148	0.237	0.065	0.450
2	0.163	0.425	0.103	0.691
4	0.168	0.656	0.151	0.975
8	0.169	1.069	0.237	1.475
16	0.170	1.903	0.385	2.459
32	0.171	3.577	0.668	4.416
64	0.171	6.905	1.281	8.356
128	0.171	13.619	2.514	16.304
256	0.171	27.286	4.991	32.449
512	0.171	54.673	9.966	64.810
1024	0.171	109.139	19.915	129.225

From these results, it can be observed that the *Forward Rendering* technique exceeds the 60 FPS threshold after 128 lights and exceeds the 30 FPS threshold at just over 256 lights and the performance decreases linearly as the light count increases. In the case of the *Sponza Atrium* scene the opaque rendering pass is the primary bottleneck for rendering while the transparent pass also degrades linearly but at a slower rate. This is expected behaviour for this scene since there are fewer transparent objects in the scene than opaque objects.

In the next experiment, the San Miguel scene is loaded and the number of lights in the scene is increased and the statistics captured.

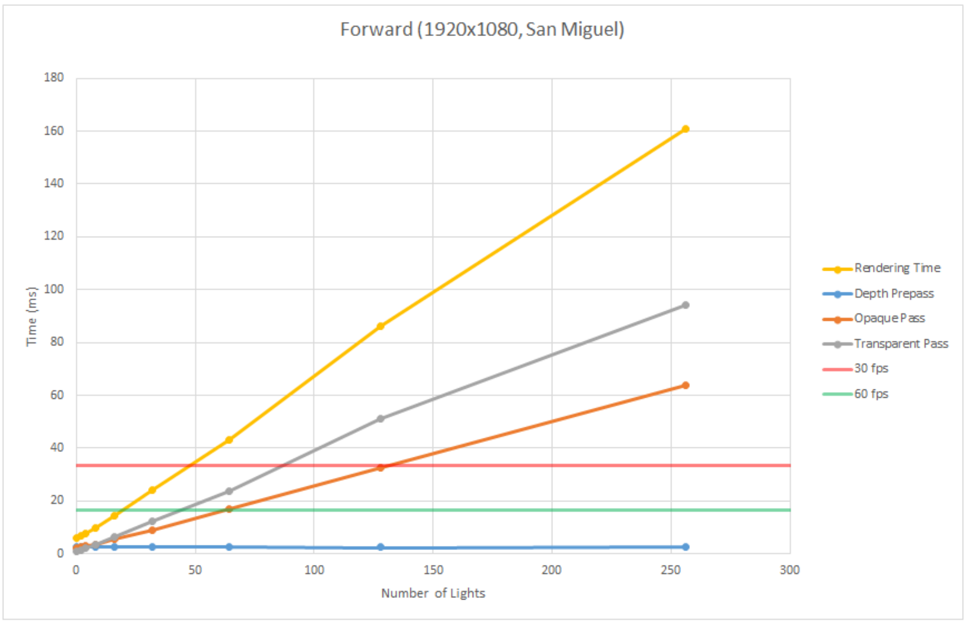

FIGURE 10.2: San Miguel scene using Forward Rendering.

TABLE 10.2: Timings for rendering the San Miguel scene using Forward Rendering.

Num Lights	Depth Prepass	Opaque Pass	Transparent Pass	Frame Time
0	2.534	2.444	1.163	6.141
2	2.630	2.640	1.463	6.733
4	2.635	2.909	2.138	7.681
8	2.631	3.666	3.522	9.819
16	2.635	5.412	6.378	14.425
32	2.625	9.156	12.224	24.005
64	2.621	16.829	23.885	43.335
128	2.529	32.557	51.023	86.109
256	2.617	63.760	94.295	160.672

The San Miguel scene is clearly more complex than the Sponza scene and contains much more geometry. It also contains a considerable amount of transparent objects compared to the Sponza scene primarily due to the large trees in the middle of the hacienda as can be seen in Figure 10.3. Due to the large amount of transparent objects in this scene, the the transparent pass of the *Forward Rendering* technique becomes the bottleneck of the rendering time. This is due to the large amount of overdraw that is caused by the transparent objects.

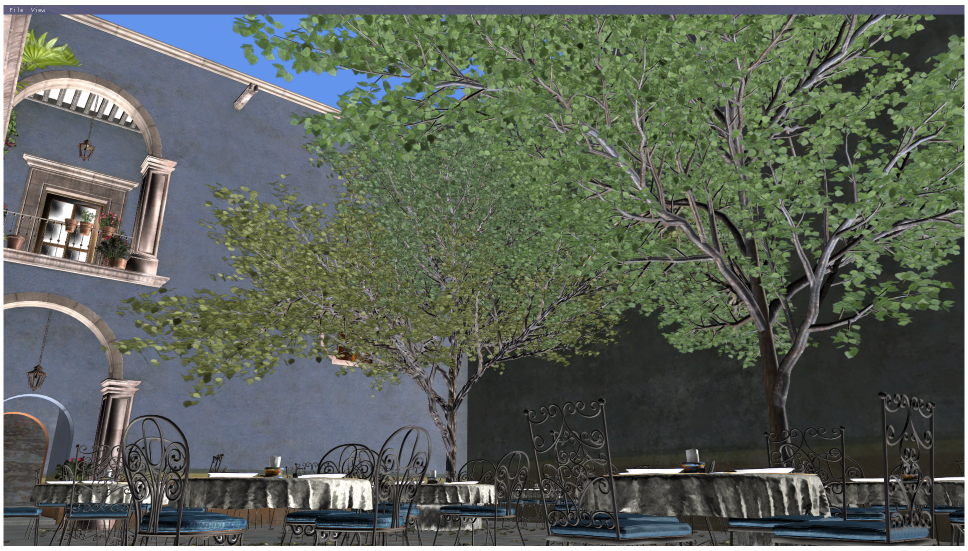

FIGURE 10.3: San Miguel scene showing the two large trees in the middle of the hacienda. Each leaf on the tree is a transparent quad causing a lot of overdraw.

10.3 Tiled Forward Shading

In the next experiment, the *Tiled Forward Shading* technique is tested. In the first test, the Sponza scene is loaded again and the lights are increased and the statistics collected. In the first case, the number of lights are increased while the size of the volume in which the lights are placed remain constant. In this case, the density of the lights increased linearly as the number of lights increased.

The following passes of the *Tiled Forward Shading* technique were captured:

1. Depth Prepass
2. Light Culling
3. Opaque Pass
4. Transparent Pass
5. Total Rendering Time

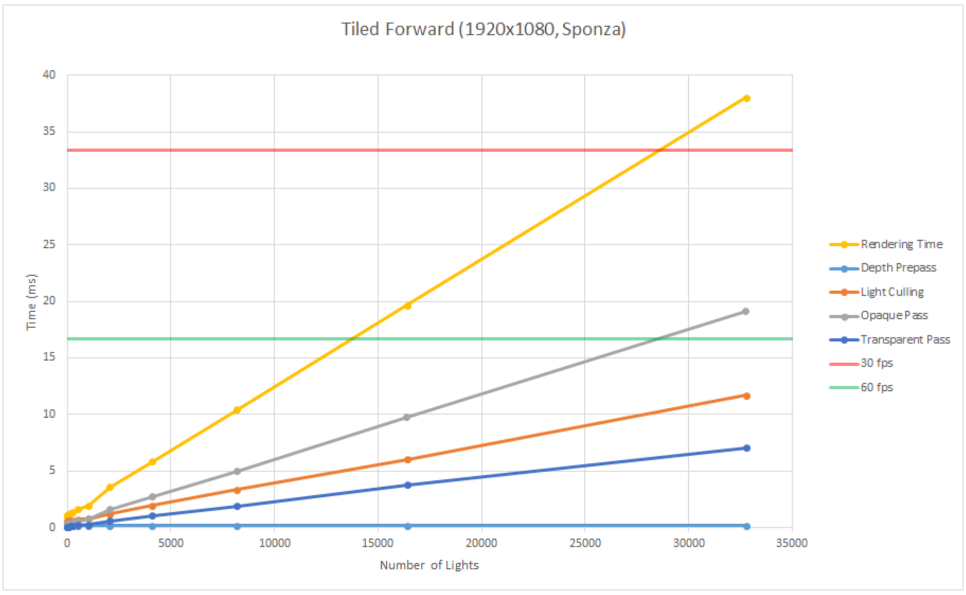

FIGURE 10.4: Chart showing performance of rendering the Sponza scene using *Tiled Forward Shading* with increasing light density.

TABLE 10.3: Timings for rendering the Sponza scene using *Tiled Forward Shading* with increasing light density.

Num Lights	**Depth Prepass**	**Light Culling**	**Opaque Pass**	**Transparent Pass**	**Frame Time**
0	0.153	0.394	0.290	0.069	0.906
2	0.160	0.525	0.289	0.069	1.043
4	0.160	0.514	0.289	0.069	1.032
8	0.160	0.530	0.290	0.069	1.049
16	0.159	0.549	0.292	0.073	1.072
32	0.157	0.568	0.296	0.080	1.100
64	0.157	0.598	0.317	0.088	1.160
128	0.157	0.618	0.367	0.115	1.257
256	0.157	0.637	0.446	0.148	1.388
512	0.163	0.711	0.565	0.195	1.633
1024	0.163	0.768	0.781	0.239	1.951
2048	0.168	1.202	1.596	0.597	3.563
4096	0.170	1.922	2.681	1.041	5.814
8192	0.170	3.348	5.010	1.904	10.433
16384	0.169	6.026	9.725	3.747	19.666
32768	0.169	11.679	19.123	7.055	38.026

The results reveal that the performance of the *Tiled Forward Shading* technique improves on that of the *Forward Rendering* technique shown in the previous section. Even with increasing light density, the *Tiled Forward Shading* technique can handle approximately 14,300 dynamic lights before crossing the 60 FPS threshold and approximately 28,700 dynamic lights before cross the 30 FPS threshold.

As expected with increasing light density, the rendering time increases linearly as the number of lights increases. As with *Forward Rendering*, the opaque pass increases at the highest rate compared to the transparent pass. The light culling technique also increases linearly as the number of lights increases but at a slower rate than the opaque rendering pass.

In the next test, the same scene was used but the light density is not more than 1 $light/unit^3$.

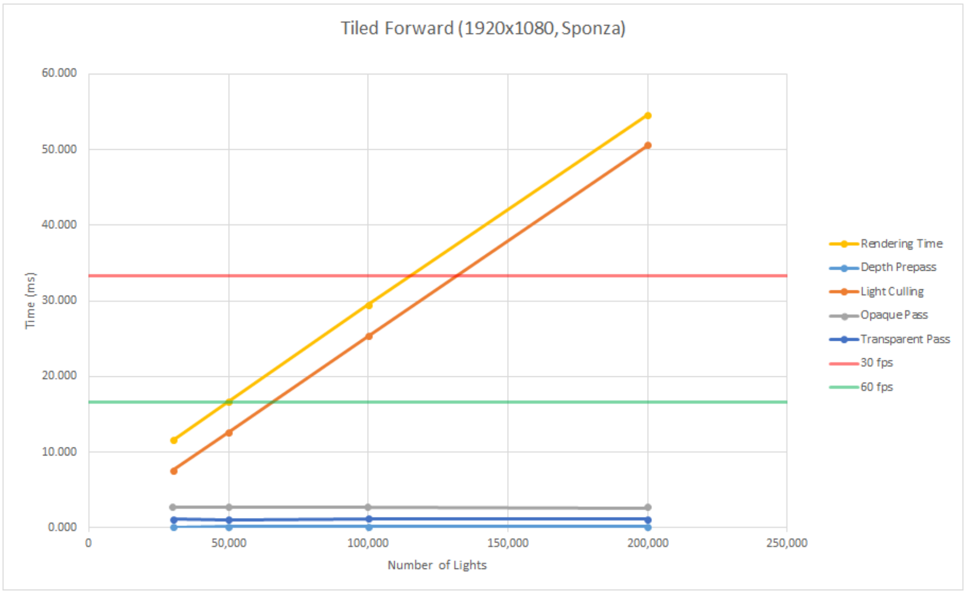

FIGURE 10.5: Chart showing performance of rendering the Sponza scene using *Tiled Forward Shading* with maximum light density of 1 $light/unit^3$.

TABLE 10.4: Timings for rendering the Sponza scene using *Tiled Forward Shading* with maximum light density of 1 $light/unit^3$.

Num Lights	**Depth Prepass**	**Light Culling**	**Opaque Pass**	**Transparent Pass**	**Frame Time**
30,000	0.164	7.555	2.712	1.140	11.571
50,000	0.166	12.609	2.768	1.139	16.683
100,000	0.166	25.375	2.767	1.212	29.520
200,000	0.166	50.633	2.684	1.155	54.637

In this case the time required for the opaque and transparent rendering passes remains constant. This shows that the *Tiled Forward Shading* rendering technique is effective at improving the performance of the shading passes of the rendering technique when the light density remains constant. The major bottleneck of this rendering technique is the light culling pass, increasing linearly as the number of lights increases. This is an indication that in order to improve the performance of this rendering technique, the most obvious area to invest effort is the light culling pass.

10.4 Volume Tiled Forward Shading

The *Volume Tiled Forward Shading* technique was tested with the Sponza scene. During the first test, the number of lights was increased without modifying the volume wherein lights were randomly placed. In the second test, the number of lights was increased while maintaining a maximum light density of $1\ light/unit^3$. The rendering technique was also tested using the San Miguel scene while maintaining a maximum light density of $1\ light/unit^3$.

Statistics were collected for the following passes of the rendering algorithm:

1. Depth Prepass
2. Mark Active Tiles
3. Build Tile List
4. Assign Lights
5. Opaque Pass
6. Transparent Pass
7. Total Rendering Time

The *Mark Active Tiles* pass will "activate" any volume tiles that contain a sample while the *Build Tile List* pass creates a dense list of volume tile ID's that are activated. The *Assign Lights* pass assigns lights to active tiles that are intersecting with the volume tiles. These passes are described in detail in Chapter 8 but are mentioned here for clarity.

In the first test, the Sponza scene was loaded and the number of lights was increased while keeping the volume the lights were placed in a constant size resulting in an increasing light density.

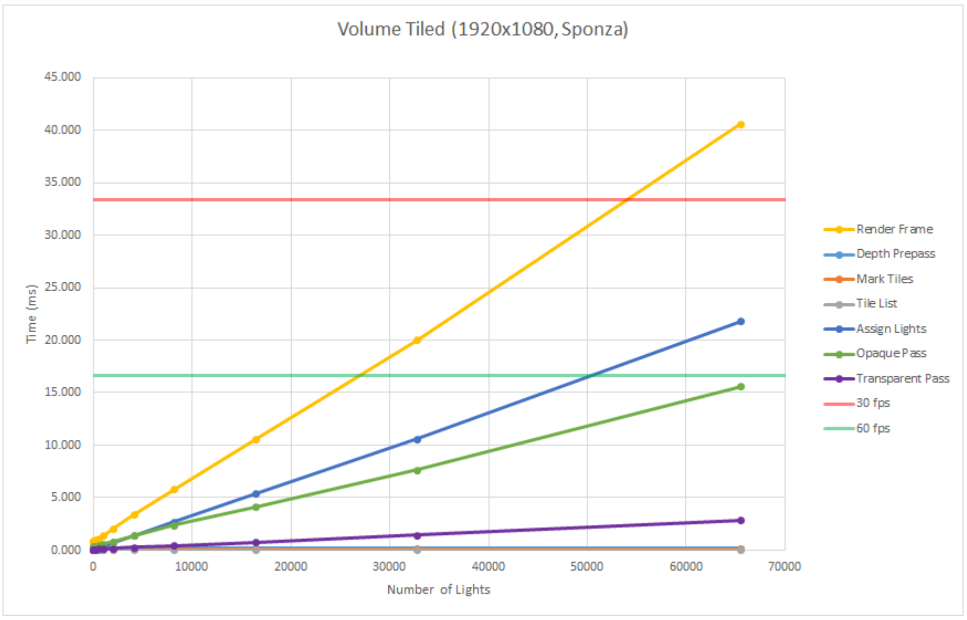

FIGURE 10.6: Chart showing performance of rendering the Sponza scene using *Volume Tiled Forward Shading* with increasing light density.

TABLE 10.5: Timings for rendering the Sponza scene using *Volume Tiled Forward Shading* with increasing light density. The *Depth Prepass*, *Mark Active Tiles*, and *Build Tile List* data is omitted from this table to conserve space.

Num Lights	**Assign Lights**	**Opaque Pass**	**Transparent Pass**	**Frame Time**
0	0.136	0.296	0.078	0.798
2	0.164	0.296	0.078	0.833
4	0.164	0.296	0.078	0.833
8	0.168	0.308	0.081	0.852
16	0.169	0.303	0.080	0.849
32	0.173	0.323	0.083	0.875
64	0.172	0.309	0.081	0.858
128	0.181	0.355	0.091	0.923
256	0.197	0.394	0.100	0.992
512	0.215	0.444	0.104	1.064
1024	0.353	0.599	0.121	1.378
2048	0.769	0.832	0.183	2.090
4096	1.409	1.368	0.282	3.366
8192	2.695	2.378	0.453	5.833
16384	5.369	4.108	0.741	10.527
32768	10.611	7.644	1.447	20.010
65536	21.804	15.618	2.873	40.607

The *Volume Tiled Forward Shading* rendering technique shows a significant performance increase over *Tiled Forward Shading* even when the light density is increased. This is primarily due to the better light assignment that can be achieved by segmenting the light tiles in the depth. Since fewer false positives are generated by the volume tiled culling pass, the overall performance of the rendering technique is improved. It can be observed from these results that the time for the *transparent* and *opaque* passes does increase linearly, while the time required for the *light assignment* stage increases at a faster rate.

In the next test, the *Sponza* scene is rendered again using the *Volume Tiled Forward Shading* technique. In this case, the light density does not exceed $1\ light/unit^3$.

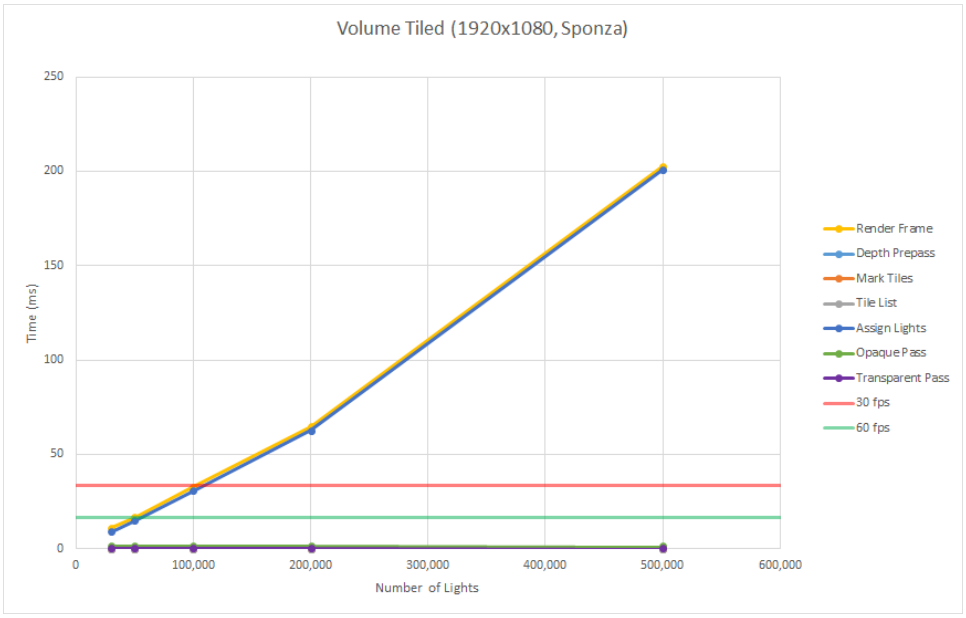

FIGURE 10.7: Chart showing performance of rendering the Sponza scene using *Volume Tiled Forward Shading* with maximum light density of $1\ light/unit^3$.

TABLE 10.6: Timings for rendering the Sponza scene using *Volume Tiled Forward Shading* with maximum light density of $1\ light/unit^3$. The *Depth Prepass*, *Mark Active Tiles*, and *Build Tile List* timings are omitted from this table to conserve space.

Num Lights	**Assign Lights**	**Opaque Pass**	**Transparent Pass**	**Frame Time**
30,000	9.135	1.419	0.279	11.114
50,000	14.838	1.421	0.271	16.810
100,000	30.901	1.440	0.266	32.894
200,000	62.746	1.399	0.261	64.692
500,000	200.852	1.356	0.252	202.739

While the average density of the lights in the scene does not exceed 1 $light/unit^3$, the performance of the *opaque* and *transparent* shading passes remains almost insignificant to the overall performance of the rendering algorithm and it is clear from these results that the *light assignment* pass consumes nearly 100% of the total rendering time.

In the next test, the *San Miguel* scene was rendered using the *Volume Tiled Forward Shading* technique while maintaining a maximum light density of 1 $light/unit^3$.

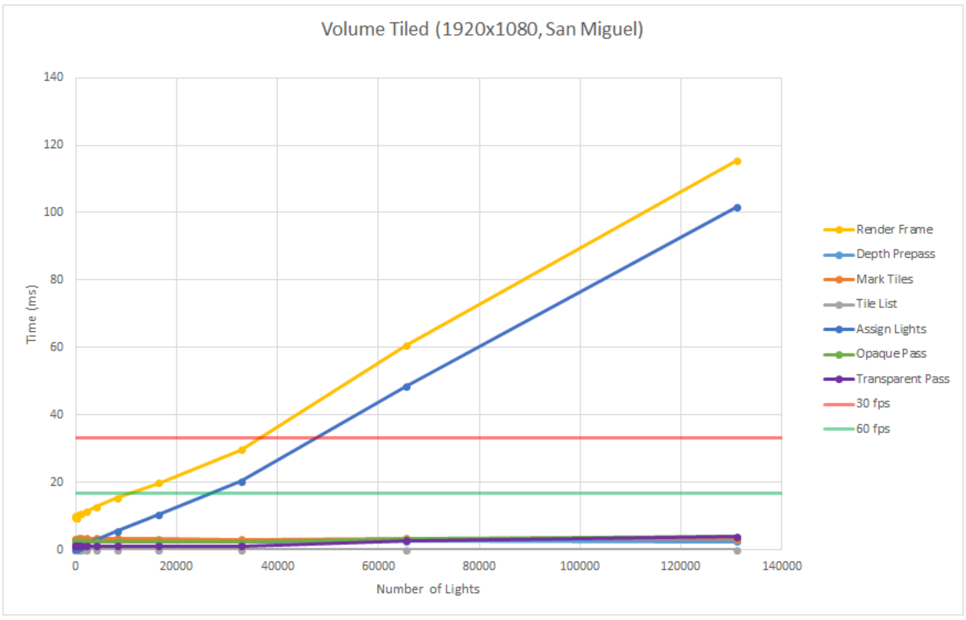

FIGURE 10.8: Chart showing performance of rendering the San Miguel scene using *Volume Tiled Forward Shading* with maximum light density of 1 $light/unit^3$.

TABLE 10.7: Timings for rendering the San Miguel scene using *Volume Tiled Forward Shading* with maximum light density of 1 $light/unit^3$. The *Depth Prepass*, *Mark Active Tiles*, and *Build Tile List* timings are omitted from this table to conserve space.

Num Lights	Assign Lights	Opaque Pass	Transparent Pass	Frame Time
0	0.257142	2.46628	1.1836	9.7806148
2	0.325061	2.46049	1.17391	9.8930485
4	0.330718	2.47328	1.19382	9.9534927
8	0.325999	2.46624	1.18611	9.9100997
16	0.320079	2.43724	1.17058	9.7923808
32	0.310132	2.39869	1.14573	9.627357
64	0.303072	2.34331	1.13876	9.4462456
128	0.327978	2.25679	1.1598	9.4033059
256	0.328362	2.33662	1.12413	9.4245832
512	0.439239	2.54218	1.2427	10.3517244
1024	0.764081	2.54836	1.23193	10.6700791
2048	1.60282	2.50297	1.23063	11.3737761
4096	2.96093	2.5123	1.22029	12.7359512
8192	5.65014	2.50619	1.21972	15.3887541
16384	10.5104	2.41291	1.17965	19.8892828
32768	20.4104	2.39271	1.21395	29.7193525
65536	48.5804	3.29217	2.76885	60.7786017
131072	101.763	3.92572	3.90272	115.4876544

Due to the increased geometric complexity of the *San Miguel* scene, the timings of the *opaque* and *transparent* shading passes are increased compared to that of the *Sponza* scene but still remain relatively insignificant compared to the increasing timings of the *light assignment* phase. Although the total rendering performance of *Volume Tiled Forward Shading* technique is improved compared to that of the *Tiled Forward Shading* technique, there is still plenty of room for improvement in the *light assignment* phase.

In the next section a variation of the *Volume Tiled Forward Shading* technique is tested. In this case, a Bounding Volume Hierarchy (BVH) is constructed over the lights in the scene before performing the light assignment pass.

10.5 Volume Tiled Forward Shading (BVH)

An optimized version of the *Volume Tiled Forward Shading* technique was tested where a BVH was constructed over the lights before performing the *light assignment* phase. Before the BVH can be constructed, the lights are sorted according to the Z-order of the lights as explained in Chapter 6 and 5. After constructing the BVH over the lights in the scene, the performance of the light assignment phase is significantly improved.

The passes of the Volume Tiled Forward Shading technique with BVH optimization are:

1. Reduce Lights
2. Compute Morton Codes
3. Sort
4. Build BVH
5. Depth Prepass
6. Mark Active Tiles
7. Build Tile List
8. Assign Lights
9. Opaque Pass
10. Transparent Pass
11. Total Rendering Time

The first four stages are required to build the BVH over the lights in the scene and are unique to the *Volume Tiled Forward Shading with BVH* technique. The last six stages of the *Volume Tiled Forward Shading with BVH* technique are identical to that of the naïve *Volume Tiled Forward Shading* technique.

The first test using the *Volume Tiled Forward Shading with BVH* technique renders the *Sponza* scene with an increasing number of lights while maintaining a constant volume to position the lights. This results in an increasing light density.

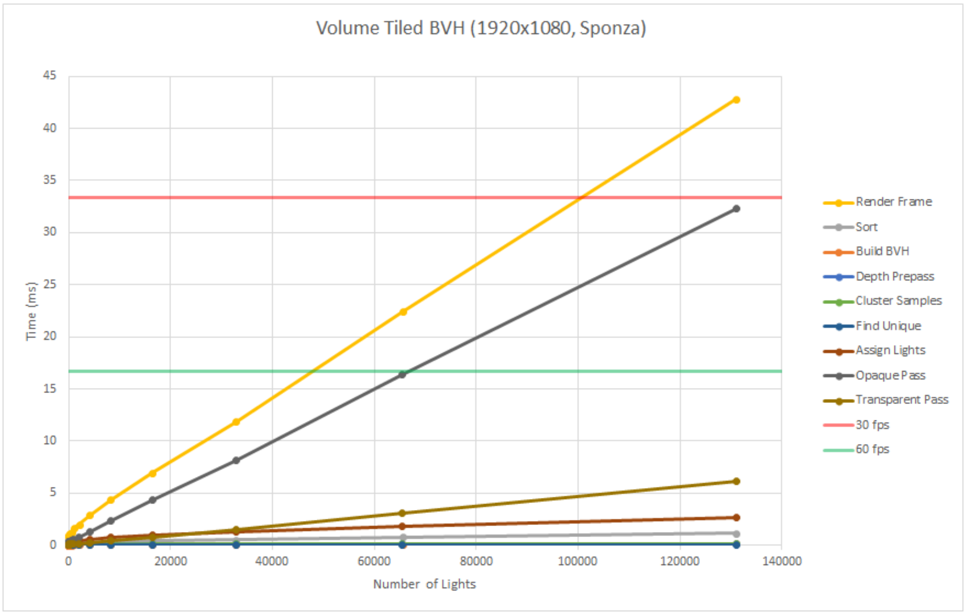

FIGURE 10.9: Chart showing performance of rendering the Sponza scene using *Volume Tiled Forward Shading with BVH* with increasing light density.

TABLE 10.8: Timings for rendering the Sponza scene using *Volume Tiled Forward Shading with BVH* with increasing light density. The *Reduce Lights*, *Compute Morton Codes*, *Depth Prepass*, *Mark Active Tiles*, and *Build Tile List* timings are omitted from this table to conserve space.

Num Lights	**Sort**	**Build BVH**	**Assign Lights**	**Opaque Pass**	**Transparent Pass**	**Frame Time**
1024	0.177	0.035	0.361	0.593	0.132	1.639
2048	0.224	0.034	0.447	0.782	0.167	1.996
4096	0.290	0.050	0.609	1.333	0.278	2.900
8192	0.365	0.052	0.798	2.350	0.476	4.391
16384	0.448	0.051	1.000	4.325	0.812	6.985
32768	0.535	0.055	1.281	8.117	1.541	11.878
65536	0.774	0.072	1.791	16.334	3.082	22.413
131072	1.188	0.166	2.692	32.294	6.114	42.843

As expected with increasing light density, the *opaque* shading pass consumes a significant portion of the rendering time. What is interesting is that the timings for the *sorting* and the *light assignment* phases remains almost constant during this test.

In the next test, the Sponza scene was rendered again this time maintaining a maximum light density of 1 $light/unit^3$.

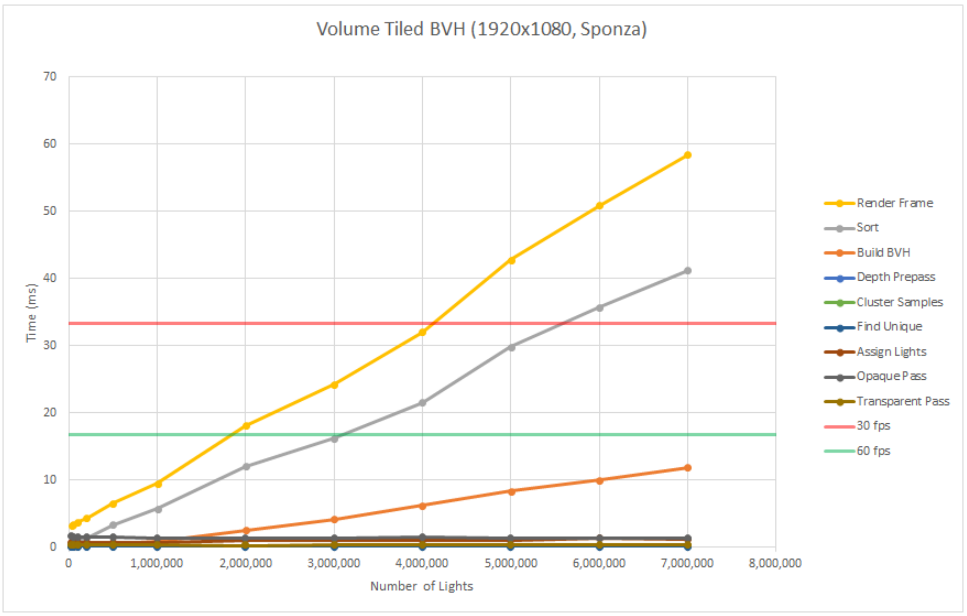

FIGURE 10.10: Chart showing performance of rendering the Sponza scene using *Volume Tiled Forward Shading with BVH* with maximum light density of 1 $light/unit^3$.

TABLE 10.9: Timings for rendering the Sponza scene using *Volume Tiled Forward Shading with BVH* with maximum light density of 1 $light/unit^3$. The *Reduce Lights*, *Compute Morton Codes*, *Depth Prepass*, *Mark Active Tiles*, and *Build Tile List* timings are omitted from this table to conserve space.

Num Lights	**Sort**	**Build BVH**	**Assign Lights**	**Opaque Pass**	**Transparent Pass**	**Frame Time**
30,000	0.342	0.053	0.617	1.579	0.322	3.232
50,000	0.474	0.100	0.599	1.515	0.289	3.299
100,000	0.825	0.158	0.596	1.496	0.273	3.690
200,000	1.361	0.230	0.643	1.448	0.273	4.327
500,000	3.310	0.468	0.608	1.388	0.259	6.501
1,000,000	5.691	0.840	0.681	1.385	0.264	9.501
2,000,000	12.018	2.476	1.014	1.309	0.248	18.055
3,000,000	16.199	4.160	0.911	1.354	0.270	24.246
4,000,000	21.492	6.203	0.968	1.406	0.267	32.037
5,000,000	29.818	8.297	0.937	1.376	0.273	42.754
6,000,000	35.701	9.981	1.242	1.277	0.260	50.875
7,000,000	41.238	11.817	1.069	1.311	0.255	58.452

In this test, the timings for the *opaque* and *transparent* shading passes remains constant and the primary overhead of the technique is consumed by the sorting and BVH construction phases.

In the next test, the *San Miguel* scene is rendered using the *Volume Tiled Forward Shading with BVH* technique while maintaining a maximum light density of 1 $light/unit^3$.

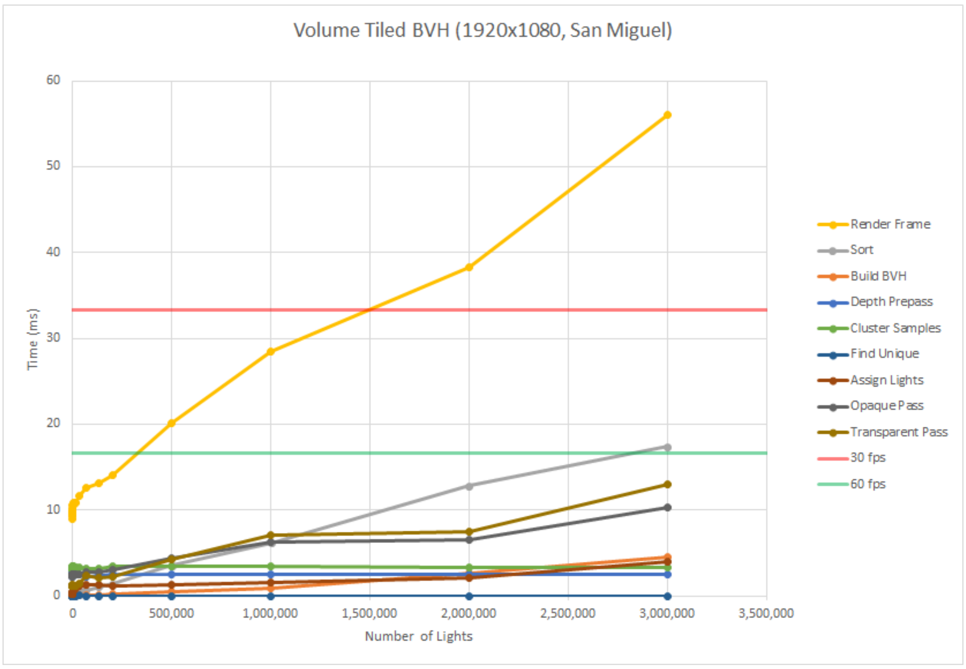

FIGURE 10.11: Chart showing performance of rendering the San Miguel scene using *Volume Tiled Forward Shading with BVH* with maximum light density of 1 $light/unit^3$.

TABLE 10.10: Timings for rendering the San Miguel scene using *Volume Tiled Forward Shading with BVH* with maximum light density of 1 $light/unit^3$. The *Reduce Lights*, *Compute Morton Codes*, *Depth Prepass*, *Mark Active Tiles*, and *Build Tile List* timings are omitted from this table to conserve space.

Num Lights	Sort	Build BVH	Assign Lights	Opaque Pass	Transparent Pass	Frame Time
500000	3.573	0.500	1.229	4.328	4.277	20.102
1000000	6.157	0.899	1.607	6.301	7.106	28.410
2000000	12.804	2.650	2.151	6.573	7.506	38.293
3000000	17.382	4.456	3.981	10.297	12.952	56.069

In this case, the sorting continues to be the most expensive phase of the technique but the overhead of the opaque and transparent shading passes has a more noticeable effect with this scene. The fluctuations in the timings are primarily due to the difficulty in creating a good test scenario with this scene. The San Miguel scene is much larger than the Sponza scene. While trying to maintain a maximum light density, it was often the case that the lights were created outside of view of the camera, resulting in a light density of 0 $light/unit^3$. The area in which the lights was created was adjusted and the timings were measured again but it was difficult to maintain consistent results due to fluctuations in the density of the lights. Since it is possible to determine a trend from these results, these fluctuations are deemed acceptable for this test.

What can be observed from these results is that although the performance of the *Volume Tiled Forward Shading with BVH* technique still outperforms that of the naïve *Volume Tiled Forward Shading* technique, the total frame time exceeds the acceptable limit of 33.3 ms at approximately 1.5 million light sources. This is primarily due to the increased number of volume tiles that are active in this scene due to the high number of transparent samples caused by the trees in the middle of the hacienda as can be seen in Figure 10.3.

10.6 Techniques Compared

In order to derive a general impression of the relative performance of each of the techniques discussed in this thesis, comparisons were made showing the performance of each technique in a single graph. The relative performance of rendering the Sponza scene with increasing light density is shown first.

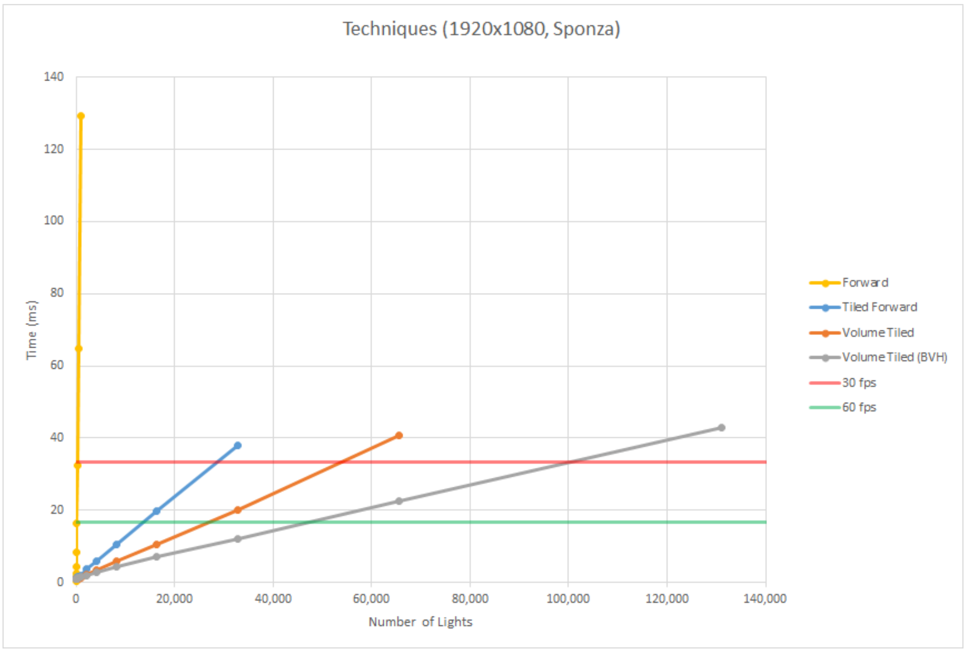

FIGURE 10.12: Chart showing relative performance of rendering the Sponza scene using *Forward*, *Tiled Forward*, *Volume Tiled Forward*, and *Volume Tiled Forward Shading with BVH* with an increasing light density.

TABLE 10.11: Timings showing relative performance of rendering the Sponza scene using *Forward, Tiled Forward, Volume Tiled Forward,* and *Volume Tiled Forward Shading with BVH* with an increasing light density.

Num Lights	Forward	Tiled Forward	Volume Tiled Forward	Volume Tiled Forward (BVH)
0	0.450	0.906	0.798	0.808
2	0.691	1.043	0.833	0.951
4	0.975	1.032	0.833	0.954
8	1.475	1.049	0.852	0.955
16	2.459	1.072	0.849	0.956
32	4.416	1.100	0.875	0.975
64	8.356	1.160	0.858	0.976
128	16.304	1.257	0.923	1.067
256	32.449	1.388	0.992	1.095
512	64.810	1.633	1.064	1.227
1024	129.225	1.951	1.378	1.639
2048		3.563	2.090	1.996
4096		5.814	3.366	2.900
8192		10.433	5.833	4.391
16384		19.666	10.527	6.985
32768		38.026	20.010	11.878
65536			40.607	22.413
131072				42.843

Even with increasing light density, both the naïve and optimized *Volume Tiled Forward Shading* rendering techniques outperform *Tiled Forward Shading*. This is primarily due to the improved light culling resulting in reduced false positives at geometric boundaries within a tile. It can be observed that even with a relatively low number of lights, the *Volume Tiled Forward Shading* techniques outperforms *Tiled Forward Shading*.

Next, the relative performance of the different rendering technique is compared when rendering the Sponza scene again but with a maximum light density of $1\ light/unit^3$.

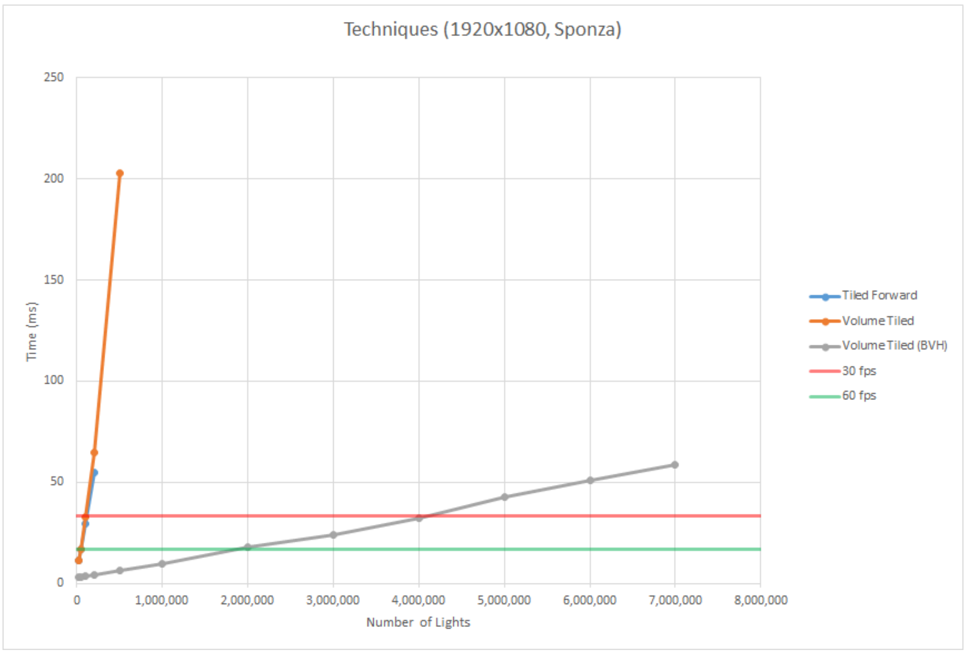

FIGURE 10.13: Chart showing relative performance of rendering the Sponza scene using *Tiled Forward*, *Volume Tiled Forward*, and *Volume Tiled Forward Shading with BVH* with a maximum light density of $1\ light/unit^3$. The performance of traditional *Forward Rendering* is omitted from this chart because the timings start at 30,000 light sources which is already too many lights to gather any useful timings.

TABLE 10.12: Timings showing relative performance of rendering the Sponza scene using *Tiled Forward*, *Volume Tiled Forward*, and *Volume Tiled Forward Shading with BVH* with a maximum light density of $1\ light/unit^3$.

Num Lights	**Tiled Forward**	**Volume Tiled Forward**	**Volume Tiled Forward (BVH)**
30000	11.571	11.114	3.232
50000	16.683	16.810	3.299
100000	29.520	32.894	3.690
200000	54.637	64.692	4.327
500000		202.739	6.501
1000000			9.501
2000000			18.055
3000000			24.246
4000000			32.037
5000000			42.754
6000000			50.875
7000000			58.452

While maintaining a maximum light density of $1\ light/unit^3$ the rendering performance of the *Volume Tiled Foward Shading with BVH* outperforms

Tiled Forward and naïve *Volume Tiled Forward Shading* by far. What may be unexpected from these results is that the performance of the *Tiled Forward Shading* technique outperforms that of the naïve *Volume Tiled Forward Shading* technique. This is likely caused by the limited density of the lights in the scene resulting in fewer false positives at geometric boundaries mitigating the overhead of these false positives.

Next the performance characteristics of the various rendering techniques is analyzed while rendering the San Miguel scene.

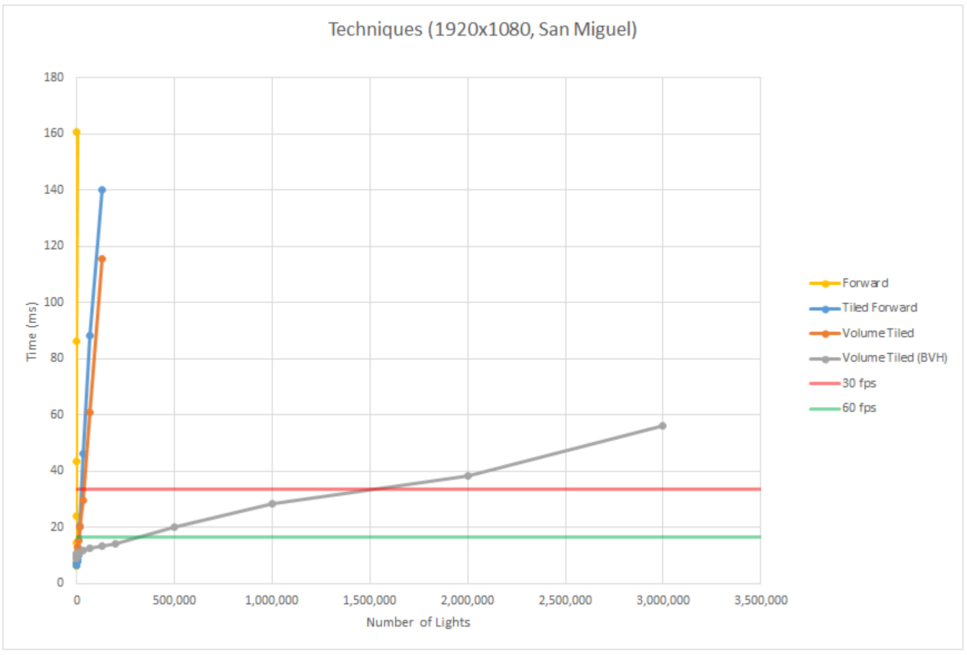

FIGURE 10.14: Chart showing relative performance of rendering the San Miguel scene using *Forward*, *Tiled Forward*, *Volume Tiled Forward*, and *Volume Tiled Forward Shading with BVH* with a maximum light density of 1 $light/unit^3$.

TABLE 10.13: Timings showing relative performance of rendering the San Miguel scene using *Forward*, *Tiled Forward*, *Volume Tiled Forward*, and *Volume Tiled Forward Shading with BVH* with a maximum light density of $1\ light/unit^3$.

Num Lights	Forward	Tiled Forward	Volume Tiled Forward	Volume Tiled Forward (BVH)
0	6.141	6.450	9.781	8.925
2	6.733	7.032	9.893	9.942
4	7.681	7.045	9.953	10.240
8	9.819	7.030	9.910	9.756
16	14.425	7.032	9.792	9.537
32	24.005	7.036	9.627	9.595
64	43.335	6.975	9.446	9.526
128	86.109	6.713	9.403	9.420
256	160.672	6.758	9.425	9.397
512		6.759	10.352	10.606
1024		7.139	10.670	10.539
2048		7.604	11.374	10.663
4096		8.495	12.736	10.837
8192		10.041	15.389	10.781
16384		20.531	19.889	10.893
32768		46.051	29.719	11.644
65536		88.118	60.779	12.631
131072		139.965	115.488	13.154
200000				14.050
500000				20.102
1000000				28.410
2000000				38.293
3000000				56.069

In this case the performance of the *Volume Tiled Forward Shading with BVH* technique clearly outperforms any of the other techniques. An unexpected results of this test is that the performance of the *Volume Tiled Forward Shading* technique exceeds that of *Tiled Forward Shading* only after more than 16,384 active light sources in the scene. The improved performance of the *Tiled Forward Shading* technique with less than 16,384 lights sources is due to the reduced cost of the *light assignment* phase of the *Tiled Forward Shading* technique that is a result of relatively fewer screen tiles than active volume tiles in the case of *Volume Tiled Forward Shading*. At more than 16,384 lights, the benefits of the improved light culling of *Volume Tiled Forward Shading* on the *opaque* and *transparent* rendering passes outweighs that of the *Tiled Forward Shading* technique.

10.7 Rate of Increase

In order to provide an overall impression of the performance difference between the various techniques, the rate of increase (r) is computed using equation 10.1.

$$r = 100 \left(\frac{t_{max} - t_{min}}{l_{max} - l_{min}} \right) \tag{10.1}$$

where t_{max} is the maximum measured time, t_{min} is the minimum measured time, l_{max} is the maximum number of lights tested, and l_{min} is the minimum number of lights tested. The rate of increase is multiplied by 100 to shift the values into a meaningful range.

TABLE 10.14: Relative rate of increase (r) for *Forward Rendering* (**FR**), *Tiled Forward Shading* (**TFS**), *Volume Tiled Forward Shading* (**VTFS**), and *Volume Tiled Forward Shading with BVH* (**VTFSBVH**).

Technique	**Sponza**	**Sponza (1 $light/unit^3$)**	**San Miguel (1 $light/unit^3$)**
FR	12.576	-	60.363
TFS	0.113	0.025	0.102
VTFS	0.061	0.041	0.081
VTFSBVH	0.032	0.00079	0.00157

10.8 Summary

In summary, the performance of the *Volume Tiled Forward Shading with BVH* technique outperforms all other techniques tested in this experiment when the number of lights in the scene exceeds 16,384, regardless of the light density and number of transparent objects in the scene. For scenes with less than 512 light sources, a simpler technique such as *Tiled Forward Rendering* exhibits decent performance characteristics, but if there is a requirement to support a large number of active light sources in the scene, then it is recommended to invest the time and effort required to implement a rendering technique that optimizes the light assignment phase such as the *Volume Tiled Forward Shading with BVH* technique described in this thesis.

Chapter 11

Conclusion & Future Work

11.1 Summary

As shown in Chapter 10, the primary bottleneck of the *Volume Tiled Forward Shading* technique is the *Assign light to tiles* pass providing an obvious opportunity to optimize this technique. To optimize the *Assign lights to tiles* pass, a Bounding Volume Hierarchy (BVH) is constructed over the lights in the scene. By traversing the BVH in the *Assign lights to tiles* pass, the complexity of that pass is effectively reduced from $\mathcal{O}(mn)$ to $\mathcal{O}(m \log_{32} n)$ where m is the number of active volume tiles and n is the number of active lights in the scene.

The construction of the BVH requires the lights in the scene to be sorted by their Z-order (Dickau, 2008). The time required to sort the lights within the scene imposes a performance overhead of $\mathcal{O}(n \log_2 n)$. Although the additional performance overhead of sorting does not outweigh the performance gains of the *Assign lights to tiles* pass, it does provide an opportunity to improve the performance of the sorting algorithm.

In the next section, several areas where the Volume Tiled Forward Shading technique can be improved are explored.

11.2 Future Work

Similar to any experimental rendering technique, the *Volume Tiled Forward Shading* technique can be improved in several areas. For example, the *Volume Tiled Forward Shading* technique utilises several render passes over the scene geometry; *Depth pre-pass*, *Mark active tiles*, and the *shading* passes each require the geometric objects in the scene to be rendered. These rendering passes invoke expensive draw calls from the rendering API. Several options to mitigate the CPU overhead imposed by the rendering API are suggested.

Another area for improvement of the *Volume Tiled Forward Shading* technique is in the volume of the volume tiles that are further away from the camera. Volume tiles close to the camera are small but volume tiles further away from the camera become large. The larger volume tiles can potentially contain many lights that do not necessarily contribute to the final shading of the geometry contained within the volume tile. A method that makes better use of the depth buffer during the *Assign lights to title* phase (similar to that used by the *Tiled Forward Shading*) is considered.

Sorting the lights is a required step before the BVH can be constructed over the lights but sorting also imposes a new limiting factor for further improving the *Volume Tiled Forward Shading* rendering technique. We explore several techniques that may help to mitigate the performance overhead of the sorting phase.

11.2.1 Reducing Draw Calls

The *Volume Tiled Forward Shading* rendering technique utilizes several render passes in order to determine the active volume tiles in the scene and to perform final shading. Each of these passes requires the visible objects in the scene to be drawn using one of the draw calls from the rendering API. For rendering static geometry, the overhead of invoking the draw function in the rendering API may be minimal but if the scene contains a lot of animated objects, the cost of computing vertex, geometry, or tessellation passes may prove to be expensive.

One way to reduce the overhead of invoking the draw calls on the API is by using *indirect draw* or *indirect compute* (or more generally referred to as *execute indirect*). Execute indirect works by creating a structured buffer on the GPU that contains all of the arguments, draw parameters, or dispatch parameters that need to be invoked in order to render the scene. When all of the scene objects need to be drawn, only a single execute indirect function needs to be called through the rendering API and thousands of scene objects can be rendered, avoiding the cost of invoking those draw calls on the CPU.

If the scene contains a lot of animated objects, it may be expensive to perform the vertex transformations required by (for example), a skeletal animated mesh, or to invoke the tessellation stages that are used to create highly detailed terrain, or ocean simulation. In this case, the tessellated and transformed vertices can be stored in a feedback buffer that can be reused for later stages thus eliminating the need to re-transform all of the vertex data several times per frame.

11.2.2 Self-Similar Volume Tiles

Similar to *Tiled Forward Shading*, *Volume Tiled Forward Shading* defines square tiles in screen space that when extruded actually form frusta in view space. When using a perspective projection matrix, the frustums become larger further away from the camera and thus cover a larger volume in space. Volume tiles closer to the camera are small and may only cover a few lights in the scene, but the volume tiles further away from the camera grow exponentially in volume. The larger the volume tile is, the more lights will fit within the tile and therefore the more lights must be considered when shading all of the samples that are contained within that volume tile.

The speed of the rendering technique is dependent on the number of lights that must be considered during shading. As can be seen from the performance results in Chapter 10, the *Forward Rendering* technique shows the poorest performance profile. The goal of both the *Tiled Forward Shading*, and *Volume Tiled Forward Shading* rendering techniques is to minimize the number of lights that need to be considered during shading. If even a single screen pixel needs to consider many lights during shading, then the total performance benefit of *Tiled*, or *Volume Tiled Forward Rendering* is lost.

One possible method to reduce the shading cost per volume tile, is to minimize the area of the volume tile that contains visible samples. A technique similar to *Tiled Forward Rendering* that uses the depth buffer to constrain the minimum and maximum depth values during light assignment could be used. This would require an additional buffer that contains the minimum and maximum depth values per volume tile to be used. The minimum and maximum values of the tile could be determined during the *Mark active tiles* pass. It should be noted that both the opaque and transparent objects in the scene must be considered when determining the minimum and maximum depth values within a tile. During the *Assign lights to tiles* pass, the AABB of the volume tile can be adjusted to account for the minimum and maximum depth values thus restricting the bounds of the tile to minimum bounding volume of the visible samples, instead of the entire volume tile.

11.2.3 Improved Sorting

Sorting is a difficult problem to solve efficiently. The techniques presented in this thesis offers favourable performance results but there is always room for improvement. The sorting stage is executed directly on the GPU in order to take advantage of the massive parallelism of the GPU and to avoid the need to transfer the sorted light indices from CPU to GPU but there might be an advantage to performing the sorting on the CPU. If the application is not CPU-bound then it might be a good idea to utilize the CPU to perform sorting at the same time the scene is being rendered on the GPU. Performing the sorting on the CPU would result in the sorted light indices being a frame behind but the 1-frame delay may not be that noticeable when rendering can be achieved at 60 or 90 FPS.

Radix sort and merge sort were the only sorting techniques explored during the creation of this thesis. Not all possible optimizations for radix sort and merge sort were explored. For example, Chapter 4 only describes a naïve approach of the parallel scan operation. A more efficient technique for performing a parallel prefix sum is described in Chapter 39 of GPU Gems 3 (Harris, Sengupta, and Owens, 2008) but this technique was not implemented in this experiment.

When sorting was implemented for the experiment, an assumption was made about the overhead of performing a merge sort on the GPU with only 2 values. For this reason, the radix sort algorithm was used to sort chunks of 256 values. The chunks of 256 values are merged using several iterations of the merge sort algorithm. If an efficient technique for merge sorting small chunks on the GPU can be created then it may perform better if only the merge sort algorithm described in this thesis is used to sort all of the values.

11.3 Conclusion

In this thesis the *Volume Tiled Forward Shading* technique is introduced. *Volume Tiled Forward Shading* technique extends upon the *Tiled Forward Shading* technique by dividing along the depth of the uniform scree-space tiles of the *Tiled Forward Shading* technique into self-similar 3D volume tiles. The geometry of the scene is rendered and any volume tile that contains a visible sample is activated and the lights in the scene are assigned to the active tiles. In the shading pass, only the lights that are contained in the same volume tile as the sample need to be considered. This technique proves to be very efficient at reducing the false positives that are generated by the elongated frustums created by *Tiled Forward Shading* technique and shows a consistent performance gain in the opaque and shading passes compared to *Tiled Forward Shading*. On average, the basic implementation of the *Volume Tiled Forward Shading* performs better than *Tiled Forward Shading* when the number of lights in the scene exceeds 16,384.

The performance of the *Volume Tiled Forward Shading* technique can be further improved by building a Bounding Volume Hierarchy (BVH) over the lights before the light assignment to volume tiles is performed.

The goal of this experiment was to support 1 million active dynamic scene lights while still maintaining real-time frame rates on commodity desktop graphics hardware. This goal (and beyond) has been achived as can be seen in the performance results shown in Chapter 10. It is safe to say that the results of this experiment have been successfully demonstrated.

Bibliography

AMD (2012). *AMD Graphics Cores Next (GCN) Architecture*. Advanced Micro Devices Inc. URL: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

Blelloch, G.E. (1989). "Scans as primitive parallel operations". In: *IEEE Transactions on Computers* 38.11, pp. 1526–1538. DOI: 10.1109/12.42122. (Visited on 02/17/2017).

Catmull, Edwin (1974). "A Subdivision Algorithm for Computer Display of Curved Surfaces". PhD thesis. University of Utah.

Clark, James H. (1976). "Hierarchical geometric models for visible surface algorithms". In: *Communications of the ACM* 19.10, pp. 547–554. DOI: 10.1145/360349.360354.

Crytek (2010). URL: http://www.crytek.com/cryengine/cryengine3/downloads (visited on 01/04/2017).

Deering, Michael et al. (1988). "The triangle processor and normal vector shader". In: *ACM SIGGRAPH Computer Graphics* 22.4, pp. 21–30. DOI: 10.1145/378456.378468.

Dickau, Robert (2008). *Lebesgue 3D curve, iteration 2*. URL: https://commons.wikimedia.org/wiki/File:Lebesgue-3d-step2.png (visited on 01/31/2017).

dyn4j (2017). URL: http://www.dyn4j.org/2010/01/sat/ (visited on 07/10/2017).

Geldreich, Rich and Matt Pritchard (2004). *GDC Vault - Deferred Shading on DX9 Class Hardware and the Xbox*. URL: http://www.gdcvault.com/play/1015172/Deferred-Shading-on-DX9-Class (visited on 09/27/2016).

Green, Oded, Robert McColl, and David A. Bader (2012). "GPU merge path". In: *Proceedings of the 26th ACM international conference on Supercomputing - ICS '12*. DOI: 10.1145/2304576.2304621. (Visited on 02/17/2017).

Harada, Takahiro (2012). "A 2.5D Culling for Forward+". In: *SIGGRAPH Asia 2012 Technical Briefs*. SA '12. Singapore, Singapore: ACM, 18:1–18:4. ISBN: 978-1-4503-1915-7. DOI: 10.1145/2407746.2407764. URL: http://doi.acm.org/10.1145/2407746.2407764.

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

Hargreaves, Shawn and Mark Harris (2004). *Deferred Shading*.

Harris, Mark, Shubhabrata Sengupta, and John D. Owens (2008). "Parallel Prefix Sum (Scan) with CUDA". In: *GPU Gems 3*. Ed. by Hubert Nguyen. 1st ed. Addison-Wesley, pp. 871–873. (Visited on 02/17/2017).

Karras, Tero (2012). *Thinking Parallel, Part II: Tree Traversal on the GPU*. URL: https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/ (visited on 01/05/2017).

Leeuw, Michiel van der (2007). *Deferred Rendering in Killzone 2*.

Lottes, Timothy (2009). *FXAA*. NVIDIA Corporation. URL: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf (visited on 07/10/2017).

McGuire, Morgan (2017). *Computer Graphics Archive*. https://casual-effects.com/data. URL: https://casual-effects.com/data.

McKee, Jay (2012). *Technology Behind AMD's "Leo Demo"*.

Microsoft (2017). URL: https://msdn.microsoft.com/en-us/library/windows/desktop/cc627092(v=vs.85).aspx#Multisample (visited on 07/10/2017).

Mittring, Martin (2009). *A bit more deferred - CryEngine 3*.

Morton, G. M. (1966). *A computer oriented geodetic data base and a new technique in file sequencing*. 1st ed. International Business Machines Co.

NVIDIA (2016a). URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 01/13/2017).

— (2016b). 1st ed. NVIDIA Corporation. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (visited on 01/06/2017).

— (2016c). 1st ed. NVIDIA Corporation. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (visited on 01/06/2017).

Olsson, Ola (2015). *Introduction to Real-Time Shading with Many Lights*.

Olsson, Ola and Ulf Assarsson (2011). "Tiled Shading". In: *Journal of Graphics, GPU, and Game Tools* 15.4, pp. 235–251. DOI: 10.1080/2151237x.2011.621761.

Olsson, Ola, Markus Billeter, and Ulf Assarsson (2012). "Clustered Deferred and Forward Shading". In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association. URL: http://dx.doi.org/10.2312/EGGH/HPG12/087-096 (visited on 10/13/2016).

Oosten, Jeremiah van (2011). *Optimizing CUDA Applications - 3D Game Engine Programming*. URL: http://www.3dgep.com/optimizing-cuda-applications/ (visited on 01/06/2017).

— (2014). *Introduction to DirectX 11*. URL: http://www.3dgep.com/introduction-to-directx-11 (visited on 09/21/2016).

— (2015). *Forward vs Deferred vs Forward+ Rendering with DirectX 11*. URL: http://www.3dgep.com/forward-plus (visited on 09/29/2016).

Saito, Takafumi and Tokiichiro Takahashi (1990). "Comprehensible rendering of 3-D shapes". In: *ACM SIGGRAPH Computer Graphics* 24.4, pp. 197–206. DOI: 10.1145/97880.97901.

Segal, Mark and Kurt Akeley (1994). *The OpenGL Graphics System: A Specification*. 1st ed. Silicon Graphics, Inc. URL: https://www.opengl.org/registry/doc/glspec10.pdf (visited on 09/21/2016).

— (2004). *The OpenGL Graphics System: A Specification*. 2nd ed. Silicon Graphics Inc. URL: https://www.opengl.org/registry/doc/glspec20.20041022.pdf (visited on 09/23/2016).

Shishkovtsov, Oles (2006). "Deferred Shading in S.T.A.L.K.E.R." In: *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Ed. by Randima Pharr MattFernando. 3rd ed. Pearson Addison Wesley Prof. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html (visited on 09/27/2016).

Singer, Graham (2013). *The History of the Modern Graphics Processor*. URL: `http://www.techspot.com/article/650-history-of-the-gpu` (visited on 09/21/2016).

Wilt, Nicholas (2013). *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. 1st ed. Addison-Wesley, pp. 365–383.

Young, Eric (2010). *DirectCompute Optimizations and Best Practices*. URL: `http://on-demand.gputechconf.com/gtc/2010/presentations/S12312-DirectCompute-Pre-Conference-Tutorial.pdf` (visited on 01/20/2017).

Zhang, Hansong et al. (1997). "Visibility culling using hierarchical occlusion maps". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*. DOI: `10.1145/258734.258781`.