BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS
AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

# LARGE-SCALE RENDERING USING SHADOWMAPS

ZOBRAZENÍ ROZSÁHLÉ SCÉNY POMOCÍ STÍNOVÝCH MAP

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR JÁN SPIŠIAK
AUTOR PRÁCE

SUPERVISOR Ing. LUKÁŠ POLOK,
VEDOUCÍ PRÁCE

BRNO 2016

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

# Zadání bakalářské práce

Řešitel: **Spišiak Ján**

Obor: Informační technologie

Téma: **Zobrazení rozsáhlé scény pomocí stínových map**
**Large-Scale Rendering Using Shadowmaps**

Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte knihovnu OpenGL a její rozšíření.
2. Nastudujte a popište metodu shadow mapping.
3. Seznamte se s existujícími implementacemi, používajícími shadow mapping pro zobrazení rozsáhlé 3D scény, a popište je.
4. Navrhněte metodu pro vykreslování scény s použitím stínových map. Zaměřte se na optimalizace nehybných světel a možnost neomezeného počtu světel ve scéně.
5. Implementujte jednoduchou aplikaci, demonstrující výpočet osvětlení pomocí navržené metody ve spojení s vhodným lokálním osvětlovacím modelem.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti dalšího vývoje.
7. Vytvořte video s prezentací projektu.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, experimenty vedoucí k bodu 4.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Polok Lukáš, Ing.**, UPGM FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

[illegible]

doc. Dr. Ing. Jan Černocký
*vedoucí ústavu*

## Abstract

Shadow mapping is the most widely used method in real-time 3D graphics for producing shadows in local light models. This thesis step-by-step explains the process of creating shadow maps. Depth biasing as well as filtering methods are analysed, then the calculation of normal offset bias for variable sized kernels is derived. We describe the process of efficiently fitting stable cascade frustums to view frustum. Also shown is how to use modern OpenGL to reduce performance overhead.

## Abstrakt

Shadow mapovanie je najpoužívanejšia metóda ktorá sa využíva v real-time 3D grafike na tvorbu tieňov v lokálnych osvetlovacích modeloch. Táto práca krok-za-krokom vysvetľuje proces vytvárania shadow máp. Porovnané su metódy výpočtu hĺbkovej odchylky ako aj filtrovacie metódy, a zároveň je odvodený výpočet normálovej odchýlky pre filtrovacie kernely s premennou veľkosťou. Taktiež popíšeme proces ako efektívne obaliť frustum kamery kaskádovým frustumom. Popri tom vysvetlíme ako využiť moderné OpenGL API na zníženie výkonnostných nedostatkov.

## Keywords

Shadow mapping, OpenGL, depth bias, normal bias, percentage-closer filtering, exponential shadow maps, cascade shadow maps, shadow stabilization, split shadow maps, layered shadow maps.

## Klíčová slova

Shadow mapy, OpenGL, hĺbková odchylka, normálová odchylka, percentage-closer filtrovanie, exponenciálne shadow mapy, kaskádové shadow mapy, stabilizácia tieňu, split shadow mapy, layered shadow mapy.

## Reference

SPIŠIAK, Ján. *Large-Scale Rendering Using Shadowmaps.* Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Polok Lukáš.

# Large-Scale Rendering Using Shadowmaps

## Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Lukáš Polok. In the thesis I use the first person plural *we*. This is done for more natural and easier readability. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . . . . . . . . . . . . . . . . . . . . .
Ján Spišiak
May 18, 2016

## Acknowledgements

Thanks to Ing. Lukáš Polok for supervising me, Faculty of information technology Brno university of technology for giving me chance to study and my family for support. Also thanks to all people working in computer graphics, for sharing their knowledge, often above and beyond of what would be required.

# Contents

**1 Introduction** **2**

**2 Shadow Mapping** **3**
- 2.1 Model . . . 3
- 2.2 Precision . . . 5
- 2.3 Biasing . . . 5
- 2.4 Filtering . . . 6
  - 2.4.1 PCF . . . 6
  - 2.4.2 Exponential SM . . . 8
  - 2.4.3 Variance SM . . . 8
- 2.5 Coverage . . . 9

**3 OpenGL** **12**
- 3.1 Pipeline overview . . . 12
- 3.2 Buffer objects . . . 13

**4 Implementation** **15**
- 4.1 Shadow map generation . . . 15
  - 4.1.1 Initialization . . . 15
  - 4.1.2 Frustum culling . . . 16
  - 4.1.3 Buffer updating . . . 17
  - 4.1.4 Drawing loop . . . 17
- 4.2 Tile-based rendering . . . 17
  - 4.2.1 Position reconstruction . . . 18
  - 4.2.2 Light model . . . 18
- 4.3 Light culling . . . 19
- 4.4 Shadow map sampling . . . 19

**5 Results** **22**
- 5.1 Future work . . . 22

**Bibliography** **24**

**Appendices** **26**
- List of Appendices . . . 27

**A Figures** **28**

**B CD contents** **30**

# Chapter 1

# Introduction

The need for real-time shadows in computer graphics has resulted in several approaches over the years; the most widely used being shadow mapping. Thanks to its flexibility it can be used to implement simple hard shadows, but its basis is also used in various advanced shadowing techniques, capable of producing both hard and soft shadows. What is shadow mapping and how do we achieve it is discussed in chapter 2. The biggest problem of shadow mapping is that it requires sampling the scene from light's view. When we render the scene from camera's view the sampling resolution is going to mismatch with shadow map resolution creating aliasing. The various approaches of removing aliasing will be discussed.

In chapter 3 we discuss working with OpenGL API. This library helps us offload most of the hard work in computer graphics to fast graphics card specialized in parallel computations. In order to get the maximum performance we need to be well versed with the specification, but significant role also plays the driver. Drivers responsibility is to implement the API and communicate with graphics card, submitting jobs the programmer prepares. This all has to happen in parallel, with CPU submitting work fast enough, otherwise the GPU stalls and we waste computing power. Hardware architecture plays a big role as well, with newer GPUs pushing the boundaries of computing power, and with vendor support of new extensions.

How to design a software architecture used in our application is part of chapter 4. We will step through important parts of pipeline, explore possible implementations, and judge how to choose between them. Taking into account multi-core design of modern processors, we need to try and design algorithms with parallelism in mind. From the OpenGL 4.3 we now have the possibility to use compute shaders, allowing us to easily tap into GPU's processing power even outside of classic rasterization pipeline. This allows us to efficiently compute lighting for hundreds of lights.

The result, application written in C++, will be then used to compare chosen methods. Some of the screenshots from the application will be shown. Video is available on CD. When measuring the GPU timings, care needs to be taken to properly query the card through OpenGL. We will also use two cards, one older NVIDIA Kepler architecture, and newer Maxwell architecture to see how different they behave. Details on this are written in chapter 5

# Chapter 2

# Shadow Mapping

Before discussing shadows we first need to define lighting model. In real-time graphics, most widely used lighting model is direct (also called local) illumination. As the name implies, it only takes into account light that comes directly from light source, the fragment's attributes and computes the reflected light. This is in contrast with global illumination, which models rays reflected off objects in scene multiple times, before hitting the point from where light reflects off into our camera. In global illumination model, the shadow is simply fewer light rays hitting surface (similar to real life). But in direct illumination, it is up to us to to find how much direct light from light source makes it onto a surface. One of these methods is shadow mapping.

There is also difference between light sources. While global illumination has no problems dealing with area lights, in direct illumination these require fast approximation of integral over half-sphere summing incoming light. And so most used light sources are simple with no surface area. These include point, directional and spot lights. The shadows between non-area and area lights also differ, while non-area lights should theoretically produce very sharp shadows, are lights will create soft shadows with penumbra size dependent on the area of light that the surface is illuminated by. Variable penumbra size is achieved with soft shadow mapping, but is more expensive than hard shadow mapping for non-area lights. Hard shadow mapping instead uses filtering/softening to either anti-alias or simulate penumbra, however such penumbra is usually fixed size.

In order compute shadow term in direct illumination, we need to know if there is a occluder between receiver and light. To find these occluders we sample the scene from light's view-projection space. Then the shadow map is sampled from our camera's rasterized view of scene. This however suffers from projection aliasing. Other techniques like shadow volumes extend the occluder silhouette to create a shadow boundaries defining volume. These shadows are very clear and don't suffer form projection aliasing, since the rasterization happens only once and in camera's view-projection space. This method has fell out of favor, since it requires additional geometry data, performance is of concern. This chapter will discuss the problems of shadow mapping and the methods used to solve them. We won't go over all implementation details, that will be discussed in chapter 4 instead.

## 2.1 Model

So let's analyze a simple case for an analytical point light $l$. The amount of light coming from The simplest space to compute occlusion is space based on lights view space. In implementation this usually ends up being texture space of a light view projection NDC space. We can get this space by multiplying world transformation with light view-projection transformation and hardware takes care of the rest. A point $r$ (as in receiver) is in shadow if there exists a surface point $o$ (as in occluder), such that $r_{xy} = o_{xy} \wedge o_z < r_z$. For directional light the situation is the same, except since our rays are parallel the light projection matrix will be different.

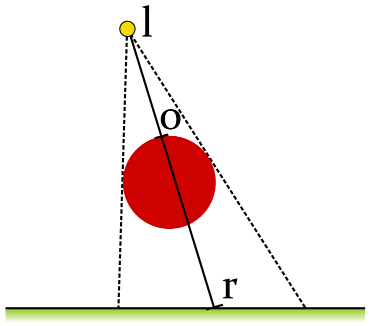


Figure 2.1: Hard shadow occlusion

For area lights size of penumbra depends on light size and distances between light, blocker and receiver. This approximation is used by Percentage Closer Soft Shadows, which enables us to create soft shadows even with single shadow map. In reality to be accurate we would have to render scene from multiple points on, and indeed some techniques utilize multiple shadow maps to improve appearance. This technique is increasing in popularity, as newer hardware architecture capabilities can significantly lower its cost. Using triangle similarity we can solve for penumbra width $p_s$, with light width $l_s$, occluder depth $o_z$ and receiver depth $r_z$:

$$\frac{p_s}{l_s} = \frac{r_z - o_z}{o_z}$$
$$p_s = \frac{(r_z - o_z) \cdot l_s}{o_z} \tag{2.1}$$

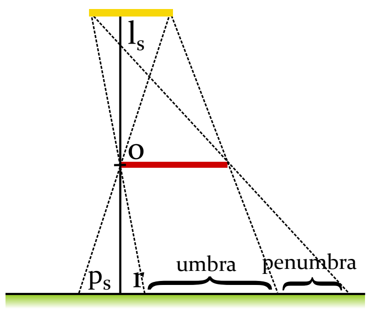


Figure 2.2: Soft shadow occlusion

## 2.2 Precision

For storing our sampled results, we need to decide the format and encoding. If our platform supports it, we can simply use the implicitly written depth buffer from rasterization. However, care should be taken when constructing perspective projection matrix, in OpenGL it usually is:

$$P = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.2}$$

This, together with hardware normalization, transforms view space $z_v$ into normalized device coordinates (NDC) space $z_n$:

$$z_n = \frac{f+n}{f-n} + \frac{1}{z_v}\frac{-2fn}{f-n} \tag{2.3}$$

The problem with this rational function is that just after first $n$ (ie. $z_v = n + n$) distance into view frustum we've used $\frac{n}{f-n}$ (approximately half) of NDC space. This is very wasteful, and can result in insufficient resolution in far distance. We can either increase near plane distance, or use a combination of reversed near/far planes with floating point depth buffer. Floating points have higher precision near zero, thus reversing the mapping with far plane near zero should increase precision in distance. While in directional lights with orthogonal mapping we won't have this problem since it's simple linear function as shown in equation 2.4. In this case the precision is uniformly distributed and we don't have to compensate.

$$z_n = 2\frac{z_v - n}{f - n} - 1 \tag{2.4}$$

## 2.3 Biasing

Since the scene in shadow map is rendered from light's view-projection, in most cases it will end up being under-sampled, meaning that single shadow map texel covers multiple window space texels. This projective aliasing, possibly combined with precision errors, creates *shadow acne* or speckled patches of self-shadowed areas in a similar fashion to moiré pattern. The problem becomes exacerbated on slopes, where single shadow texel could cover long strip of pixels.

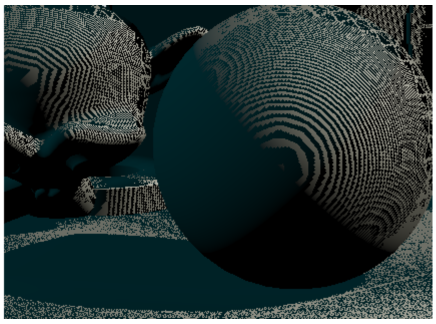


Figure 2.3: Self-shadowing errors

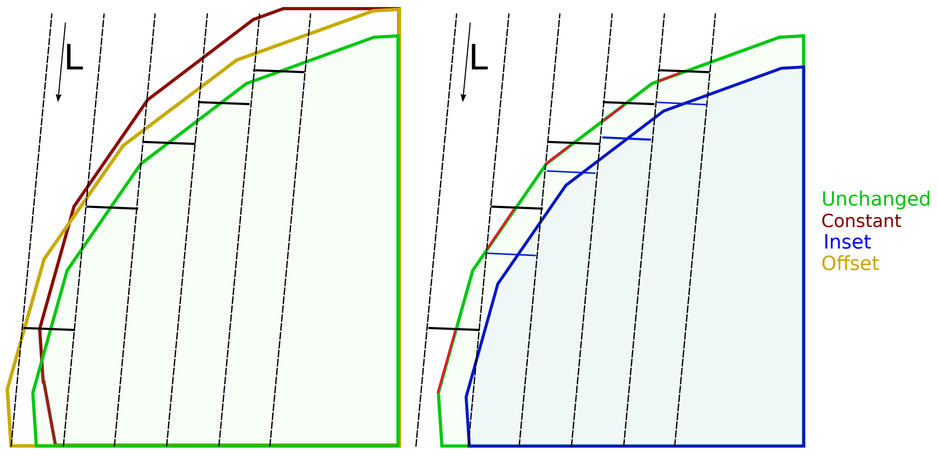


Figure 2.4: Biasing methods

The simplest approach is to simply offset by constant depth in light space (either by adding to occluder or subtracting from receiver depth position). This unfortunately solves only small sampling errors [5], and increasing this value causes *peter-panning* effect, where the object and its shadow gets separated (giving a floating appearance). Another way, would be to inset vertices into mesh using their normal during shadow map rasterization. However, this could create visible disconnects in parts of mesh where it's not continuous surface. Also insetting by normal requires normal information per vertex during shadow pass. This could have an impact on performance, since it would mean double the vertex size. More performant variant of insetting, is to offset during shadow coordinates calculation, show also in figure 2.5, where normal information is available (in vertex, or texture). This has a drawback of visually moving the shadow out of place, but the effect can be minimized down to displacement of few pixels. Research by Dou et al. [7] has tried to calculate optimal adaptive $\varepsilon$ bias per sample. Ehm et al. [8] optimized and improved the method, but found that the calculation needs a constant $K$ fitted to scale scene factor. However, both of these methods cannot be used with hardware filtering. Speed of these techniques, varies on the pipeline used, but generally is negligible compared to other parts of shadow mapping.

## 2.4 Filtering

While a simple depth test results in a point either being fully in shadow or not, this effect is rarely wanted. Using hardware shadow interpolation somewhat removes extreme crispiness, but depending on resolution, still leaves visible shadow map texels. The biggest difference between these methods is that some (ESM, VSM) can be filtered in separate pass, while others (PCF, Poisson) cannot. Filtering in separate pass allows us to make use of separable kernel, lowering the number of samples. Though on other hand also means, we also filter texels that might end up not being sampled.

### 2.4.1 PCF

The simplest approach to softening a shadow boundary is Percentage Closer Filtering. Reeves et al. [14] proposed that instead of having a step function, the shadow strength is based on ratio of occluded versus unoccluded samples. We can control the softness of the shadow by increasing or decreasing kernel size. We can combine this with hardware shadow filter function to reduce the number of samples. Using `sampler*Shadow` samplers, we supply additional coordinate that will be compared to four samples of the shadow map, and returns normalized number of how many passed. Increasing softness needs bigger kernels, and this can cause problems. Our biasing solution will need to make sure to either individually offset samples, or offset the whole kernel by large enough bias otherwise bigger kernels will bring self-shadowing back. This problem is most visible if our scene has long stretches of geometry lit at low angle.

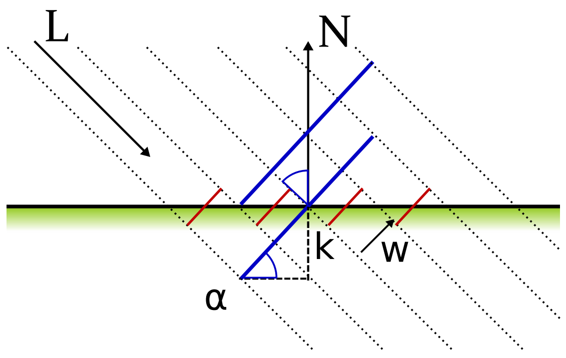


Figure 2.5: PCF kernel self-shadow problem

We can calculate the normal offset $k$ needed to push kernel out of shadow from figure 2.5. We notice the right triangle with hypotenuse $2w$ and angle $\alpha$. We can use sinus to calulate offset from these two variables. To quickly calculate $\alpha$ we should take advantage of dot product between two normalized vectors, light and normal vectors, yielding $\cos(\alpha)$. The width of light texel depends on light's projection matrix and shadow map's resolution. We can precalculate these as they as they don't change per fragment. To generalize we will use $r$ as kernel radius, i.e. half of sample count.

$$\begin{aligned} k &= r \cdot w \cdot \sin(\alpha), \\ sin(\alpha) &= \sqrt{1 - \cos(\alpha)^2}, \\ w &= \frac{t-b}{S_y} \end{aligned} \tag{2.5}$$

Using the sinus equality we can avoid two trigonometric functions in a exchange for squaring and square root. The $t$ and $b$, used to calaculate texel width, can be taken from light's ortographic matrix $P_{1,1}$ where they are in form of $\frac{2}{t-b}$. We assumed symmetric shadow map, but if they are not we can apply similar principle in horizontal direction. This solution applied single offset to whole kernel, nonetheless we could calculate offset for each sample, albeit at bigger cost. We should then preferably transform normal into light space and apply normal offset per sample. Does this method preclude using normal maps? We presumed having vertex normal available, however in deferred renderer that is rarely possible, if using normal maps. Artifacts could happen if fragment's normal pushes kernel into shadow. Note that the offset amount is modulated by $\sin(\alpha)$, so the offset increases if the light is almost perpendicular to normal direction, however such fragment also should reflect less light since it's facing away from the light. Another way is to write depth derivatives into gbuferr and reconstruct normal from depth and derivatives. Calculating derivatives from depth buffer will cause discontinuities at edges.

Very popular variant to PCF is Poisson disc sampling. We first need to generate sample offsets. Work by Bridson R. [4] shows how to implement this in general for $n$-samples for radius $r$. The points for offsets fulfill simple rule, that they cannot closer to others than radius $r$. Limiting the points to be lay inside a circle, we make the filter size rotationally invariant. Another advantage is that we can use lower number samples. This introduces noise in result, but can still produce acceptable visual quality. The filter can be rotated per pixel or per frame, to increase dithering. However we should be careful make this pattern stable when camera is not moving, or we could get flickering shadow boundaries.

### 2.4.2 Exponential SM

By replacing shadow simple step shadow function $r_z < o_z$ with exponential function $exp(k \cdot (r_z - o_z))$ we get a transition between lit area and shadow are where the depth difference is positive. We can use $k$ darkening factor to strengthen shadow transition. When blur is used to smooth out depth differences, together with the test function produces continuous transition. Blurring can be done using separable kernel, whether it's Gauss or box filter. Using such depth function also means that thin objects close to receiver will produce lighter shadows, which strengthen away from object casting them, the opposite of the effect we perceive in real life.

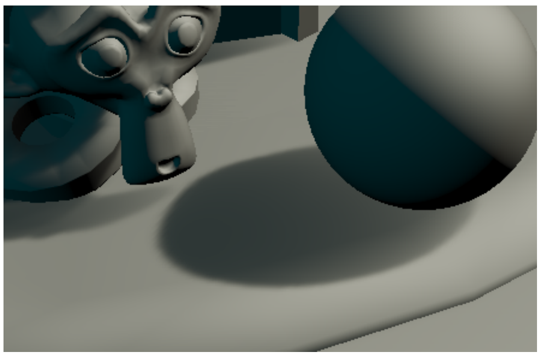


Figure 2.6: ESM bleeding with low darkening factor

### 2.4.3 Variance SM

Another test function we can use is based on approximating the depth $o_z$ to be first moment $M_1$ and depth squared to be second moment $M_2$ of a depth distribution. From moments we can calculate mean $\mu$ and variance $\sigma^2$. The paper [6] then uses Chebyshev's inequality to express attenuation of light hitting the fragment at depth $r_z$.

$$\begin{aligned} \mu &= M_1 = o_z \\ M_2 &= o_z^2 \\ \sigma^2 &= M_2 - M_1^2 \\ p(r_z) &= \frac{\sigma^2}{\sigma^2 + (t-\mu)^2} \end{aligned} \tag{2.6}$$

This technique is susceptible to light-bleeding which occurs when two occluders shade same point, the penumbra from first object then bleeds through seconds object shadow.

Later variations try to solve this problem, such as Layered VSM [11], or combination of exponential and variance SM called EVSM.

## 2.5 Coverage

If we are rendering rendering anything bigger than very small test scenes we quickly run into coverage inefficiencies. Using a regular single shadow map for most scenes results in extreme undersampling near the near plane of our frustum, and oversampling near the far plane. Increasing shadow map resolution has negative impact on both memory and bandwidth, while not solving oversampling in far distances. Using mip-maps with basic depth test is not recommended, it could introduce biasing problems 2.3 or moving boundaries, and while high anisotropy can smooth the errors, it has high performance impact [3]. Some depth test functions as VSM and ESM are capable of mip-mapping.

The nature of this method is to solve coverage inefficiency by splitting the view frustum into multiple parts, and for each a unique shadow map is created. The basis of this technique is relatively old, but it is still primary choice for many developers, thanks to its robustness and quality increase. The differences are how we fit the shadow map onto or into view frustum. One way is to split frustum with planes parallel to near and far plane, and the cover these splits by fitting the light frustum around them. This was used in parallel split shadow maps by Zhang et al. [17]. Other algorithms like cascade shadow maps, create volumes inside view frustum and fit shadow maps around them. This can have higher coverage efficiency, however accessing the shadow maps becomes more complex.

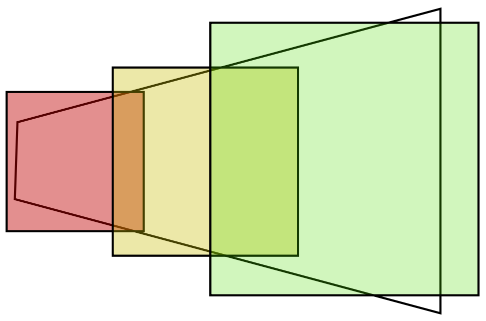


Figure 2.7: Same-sized cascades fitted to view frustum

Important factor when it comes to fitting the light's frustum around view frustum split is that we don't change its size. If we would fit a bounding box in light space around the split, its size would change depending on camera orientation. This would result in shadow map texels changing size, creating flickering on shadow boundaries, more pronounced depending on how mis-matched is the sampling rate. One way to ensure that ensure the size will be stable is to create a bounding sphere around split. Sphere has the same sized bounding box no matter the light's view matrix. To calculate a frustum split bounding sphere we make use of the frustum being symmetric this way we can reduce the problem to 2D. We will need a plane splitting the frustum in half going through opposite corners. The sphere center will lay on intersection of two symmetry planes, which is across the middle of the frustum. We will define the split as two planes at fraction of a distance between near and far plane. The real distance is easily calculated with $near + s \cdot (far - near)$, this way we are independent of actual values of near and far planes. Reducing the problem to 2D we get a bounding circle around isosceles trapezoid as seen in figure 2.8. It should be noted that

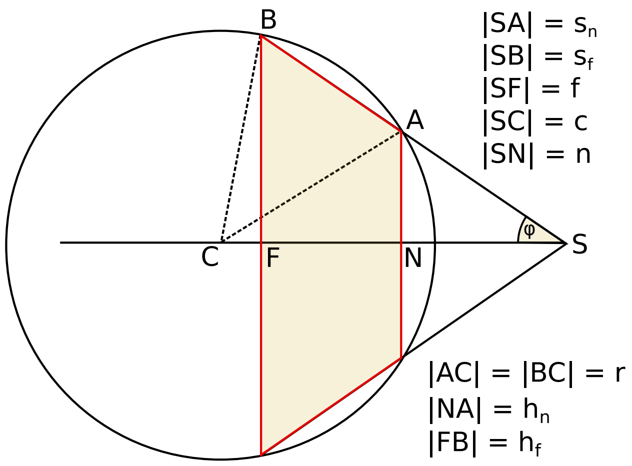


Figure 2.8: Frustum split bounding sphere

the center of the sphere doesn't have to be inside this isosceles, if the distance between the near split and far split is short enough. This method is independent of whether the center lies inside or not. If we would insist for some reason, for the center to lie inside the split we could limit it to far split distance. We can see that there is lot of coverage outside the split, this can still be useful as we will show in section 4.4.

So we need two things, the distance $c$ from origin to center of sphere, and it's radius $r$. We make use of knowing that distances $|AC|$ and $|BC|$ are same, with length $r$. These segments are also hypotenuses of triangles $CAN$ and $CBA$ respectively. We will use triangle similarity between $NSA$ and $FSB$ to calculate $s_f$ from $s_n$. The $h_n$ is a half of near plane diagonal which we need to calculate from our camera's properties defined by vertical field of view $\phi$ and ratio between height and width $y$.

$$\begin{aligned}
(c-n)^2 + h_n^2 &= (c-f)^2 + h_f^2 \\
2fc - 2cn &= h_f^2 + f^2 - (h_n^2 + n^2) \\
c &= \frac{s_f^2 - s_n^2}{2(f-n)} \\
c &= \frac{s_n^2(\frac{f}{n} - 1)}{2(f-n)}, \\
s_n &= h_n^2 + n^2, \\
h_n^2 &= (n \cdot \tan(\frac{\phi}{2}))^2 + (y \cdot n \cdot \tan(\frac{\phi}{2}))^2
\end{aligned} \tag{2.7}$$

When we move the camera the shadow map origin also moves. This offsets the sampling points, even if by a small amount, which can cause texels on boundaries to miss or hit triangles. This will cause flickering, unless we fix the shadow map origin to a grid. We can fixate the shadow map by calculating the offset from the grid and then moving the origin back. The grid size depends on the map resolution. The pseudo-code is shown in listing 2.1.

Another approach called perspective shadow maps samples a scene from a post-projection space. This space is setup up so that objects closer to camera have higher sampling resolution. However certain situations may arise, such as shadow casters being behind camera

```
vec4 origin = lproj * lview * vec4(0, 0, 0, 1)
origin *= map_size / 2
offset = (round(origin) - origin) * (2 / map_size)
lproj[3][0] += offset.x
lproj[3][1] += offset.y
```

Listing 2.1: Shadow map stabilization (square)

view frustum, that require special handling. Later successors such as Light Space Perspective SM, try to solve its limitations by modifying the post-projection transform. This method hasn't really caught on in industry. One reason might be that the split shadow maps provide simpler implementation with similar or better results. Research by Rosen [15] explored the use of warping shadow maps. As the name suggests, we sample the scene non-uniformly based on parameters as distance to camera, object boundaries, etc. We do this by rendering shadow map with non-linear transformations on scene geometry, using tessellation to mitigate artifacts of rasterization resulting from non-linearity. This novel approach can dynamically change sampling frequency of shadow map, however the requirement for tessellating the geometry can hinder performance during shadow map generation.

# Chapter 3

# OpenGL

OpenGL is a specification of an application programming interface (API) designed around 3D graphics. This API is implemented by driver vendor. Open-source implementations such as Mesa3D do exist, but because of the competitive market, the GPU vendors usually supply their optimized drivers as proprietary software. OpenGL changed a lot in last years, stepping away and abandoning fixed function pipeline. The vendors now recommend trying to adhere with Almost Zero Driver Overhead (AZDO) principle. Simply stating, the less we have to talk with driver and try to batch work together, the lower the chance that we stall the GPU pipeline losing processing power. These techniques can be implemented either by extensions or core function in newer versions.

## 3.1 Pipeline overview

We will take a short look at a minimal pipeline needed to draw triangle on screen. Majority of data in OpenGL is managed by two objects, buffers and textures. Buffers are linear memory storage blocks where user writes his data to be read by OpenGL or vice versa. Internal format of a buffer depends on a buffer target binding. Textures on the other hand can be up to three-dimensional, but their internal format is strictly set, and used whenever accessing or writing to texture. OpenGL has a large number of switches and options controlling the pipeline, but most of the work is done in programmable stages called shaders. Shaders are written in OpenGL Shading Language (GLSL). They have to be compiled and linked by the driver at run-time, in order to run on GPU.

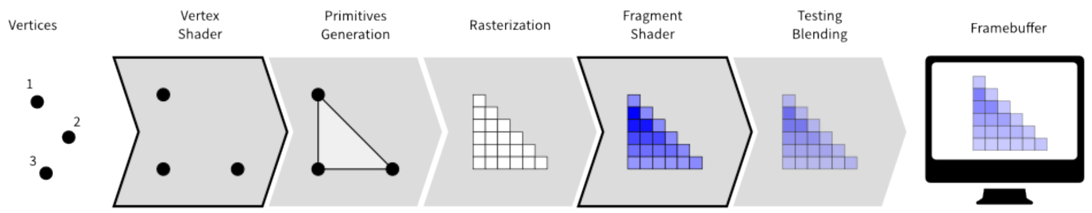


Figure 3.1: Simplified overview of pipeline from [1]

The vertex data is supplied in a buffer bound to `GL_ARRAY_BUFFER` and is also called vertex buffer object (VBO). When the vertex data repeats itself (i.e. a vertex is shared by

[1]Glumpy. Modern OpenGL, 2016. https://glumpy.github.io/modern-gl.html. Licensed under CC BY 3.0. Modified. [Online, accessed 12.5.2016]

multiple primitives), we can cut down the buffer size by using index buffer object (IBO) bound to `GL_ELEMENT_ARRAY_BUFFER`. In this buffer we specify the primitives as indexes into VBO, creating a level of indirection, but cutting the VBO size significantly. The GPUs nowadays have vertex caches, and seem to handle this kind of indexing very well. We describe the structure of a VBO in a vertex array object (VAO). We tell OpenGL how many attributes (position, normal, uv, etc.) we have by enabling them, and then specifying their type, stride and offset where they start in VBO.

When a draw call is issued the vertex data is sent to a vertex shader. Here we have to specify the location of the vertex attributes, relative to each other. If we don't, the driver is allowed change the order and we would have to query it, so it's best to explicitly state the attribute location with layout directive: `layout(location = 0)`. This way we make sure that different shaders will share the vertex location. After the vertex is processed by the vertex pipeline (starting with vertex and ending with geometry shader), by specifying the `gl_Position` output variable we tell the opengl where in normalized device coordinates (NDC) space the vertex is. The fragment is then rasterized depending on the draw target resolution. If we have depth testing enabled, it also tested against the depth buffer. Remaining fragments are sent to fragment shader. The shader processes the fragment and outputs its values, which get blended (if enabled) into the bound color buffer.

## 3.2 Buffer objects

The main distinction between buffers is in the way they connect to the pipeline. This is specified by the *target* parameter during binding. There are four types that we will use the most: `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER` for vertex data and indices, and `GL_UNIFORM_BUFFER`, `GL_SHADER_STORAGE_BUFFER` for data that we need to present to shader. So how exactly do we create and submit data? There are two ways, and we will briefly look over both of them.

### Mutable

We start by creating buffer with `glBufferData()`. The parameters are self-documenting, except for *usage* parameter. This is just a hint for driver, whether we are going to read or write, and how often will the buffer content change. For example, for a VBO, if our geometry is static, we would use `GL_STATIC_DRAW`. These hints are not restricting access in any way, as per specification chapter 6.2 [16].

We can either populate the buffer during allocation, or we can use `glBufferSubData()` with *offset* parameter to copy into buffer region. This however raises a question, when is the data copied? OpenGL has to copy the data before the function returns, since we are not required to hold the data afterwards. But what happens if the previous data is still needed for drawing? The driver in this case will mostly likely force synchronization and wait until the previous data is used. There is a workaround, by reallocating the buffer with same size, we tell the driver that we don't need the old data, also called "orphaning". If the data is still needed, we get a new allocation, thus not blocking, and if the old data is not needed, then new allocation happens over the old data.

### Immutable

Immutable buffers are allocated with `glBufferStorage()`. Immutable in this context means, that we promise to not change its size. However the contents are readable/write-able depending on the flags we specify. What is different from mutable buffers, is that we can call `glMapBufferRange` on immutable storage, to get a pointer to driver's memory. This way we when we generate data in our application we can write it straight into the buffer. When the buffer is mapped, the OpenGL cannot do any operations with it. After we are done we will unmap the buffer, the pointer is invalidated, and the data is available to OpenGL again. The buffer data can we invalidated either by mapping with `GL_MAP_INVALIDATE_BUFFER_BIT` or by `glInvalidateBufferData`. If the previous data is still used invalidating the buffer will make sure that mapping will not block, because we get a pointer to new memory. More advanced buffer streaming can be done with persistent buffers, which stay mapped the whole time. We would have to implement triple buffering (one part is used, another prepared, and one being written to), and explicit synchronization.

# Chapter 4

# Implementation

Application is written in C++ against c++11 standard. The compiler used was part of Microsoft Visual Studio 2015 on Windows 8.1. All of the libraries used, as well as source code, are multi-platform. However a multi-platform build system for the dependencies and application is not included. One of the reasons for developing on Windows, instead of Linux, was to avoid potential issues with less supported driver on Linux.

### Context creation

To use OpenGL we need a valid OpenGL context first. Context creation is not a part of standard, and is specific to operating system. We also need a window from window manager which is also platform dependent. SDL2 (Simple Directmedia library) offers abstraction for both of these things, and supports multiple platforms. We have to set window properties, and context settings such as version, default buffer options, etc. We will try to use some of the new functions in 4.5 core profile to utilize faster pathways in driver. On windows we would need to dynamically load function pointers, instead we can make use of GLEW (OpenGL Extension Wrangler) library to do it for us with simple call to `glewInit()`[1].

## 4.1 Shadow map generation

### 4.1.1 Initialization

In order to draw off-screen we need to setup a framebuffer. Framebuffer object (FBO) is just a set of attachment points for textures (or renderbuffers) to attach into. We can categorize these points into four types: color, depth, stencil, depth-stencil. If our platforms allows using texture as depth buffer, we should consider doing so. This way we can render without color buffer, thus reducing the framebuffer bandwidth. Without color buffer we don't have to explicitly write to any output in fragment shader, since fragment's depth is implicitly written (unless `glDepthMask()` is set to false).

When creating textures used for shadow maps, we should try and use layered textures. We can bind these to framebuffer using `glFramebufferTexture()` to create layered framebuffer or `glFramebufferTextureLayer()` to bind single layer of a texture. Layered framebuffers are

[1]If we are using core profile, we also need to set glewExperimental to true, otherwise GLEW won't correctly query driver, resulting in invalid function pointers. This is a result of GLEW being slightly outdated, and trying to use deprecated parameters. There are other, but also more complex, alternatives such as glbinding library.

used together with `gl_Layer` variable in shaders to control which layer the primitive will be rasterized into. When binding single layer we proceed as with normal non-layered texture.

There are numerous ways we can store vertices in VBOs, but generally engines use interleaved attributes. What this means is that attributes are stored sequentially and we specify offset from start of vertex using `glVertexAttribPointer()`. Stride in this case is size of vertex. Since the only information needed to generate shadow maps is depth, we can take advantage of this to reduce vertex size down to position only. So we have to consider the vertex size we use for geometry pass, if it's big our vertex shader would have to skip the unused attributes between positions. In this case it might be better idea to split position attribute into separate buffer.

### 4.1.2 Frustum culling

Before we submit our draw calls we need to decide which objects should cast shadow. Drawing objects that won't end up on screen can be costly, we might pay for submitting vertices, and also for vertex processing cost, even though no fragments will be rasterized. We can use frustum culling to eliminate objects outside of shadow map. The frustum characteristics depend on light type, especially what projection matrix it uses. For directional lights it is oriented bounding box, while for spotlight it's pyramid shape due to perspective projection and in the case of point light it is cube centered around source.

First we need to extract frustum planes. Gribb et al. [10] show how to extract planes from projection matrix. We can use this on view-projection matrix, to extract planes in world space. Since there are many ways to do frustum culling, we aren't limited to world space. Some approaches use clip space or local space, depends on the algorithm that best suits us. In worlds space we can use axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB) or bounding spheres, or even combination since for example bounding spheres first, since it's fast and then test OBB after. There are additional optimizations possible to use with frustum culling shown in work by Assarsson et al. [2]. Simplest bounding shape to cull is sphere, in pseudocode shown in listing. For AABB we can test with two dot products, first the center, then extent with absolute value of plane normal 4.1. Quick way to calculate world space, shown by Arvo [1], is to AABB from local space AABB (which is constant for given mesh) is to transform center using model matrix, and then transform extent as a normal with absolute value of model matrix. This AABB is not going to have best fit, but is very fast compared to calculating from mesh data.

```
result = IN;
for each plane
   d = dot(center, plane_normal);
   r = dot(extent, abs(plane_normal));
   if d + r < - plane_d
      return OUT
   else if d - r < - plane_d
      result = PARTIAL
return result
```

Listing 4.1: Fast axis-aligned bounding box frustum cull

### 4.1.3 Buffer updating

In order to properly transform our vertex data, the vertex shader needs to know the model, view and projection (MVP) matrices. These are usually combined into one MVP matrix on CPU, therefore saving GPU core cycles that would be wasted doing the matrix multiplication for each vertex. One way of making this data available to shader is with Uniform Buffer Objects (UBO). Each shader has a number of uniform block binding indices that buffer ranges can be bound to. We can set the buffer index explicitly from shader, this way we won't have to query it after shader linking. Also it ensures that if we change shader, we won't have to bind UBOs again, since the binding indexes stay the same. Generally two kinds of information are updated this way, per pass (such as geometry pass, lighting pass, etc.) and per object (matrices, materials, etc.).

### 4.1.4 Drawing loop

After we make sure that OpenGL state is correctly set, and our VAO, textures, shader and UBOs/SSBOs are bound we can issue draw command. This is where we find out if our rendering technique is designed adequately. The general rule is fewer state changes, whether it's binding objects or context states, means less driver involvement and higher throughput we can get from OpenGL. There are draw commands specifically designed for joining draws together, such as the `glDraw*Instanced*()` and `glMultiDraw*()` functions. However multi-draw requires putting geometry is shared VBO, so if the models don't fit into single vbo, we need to split and manage residency.

```
layout(location = 0) in vec3 vert_pos;
layout(location = 1) in uint draw_id;
layout(binding = 0) buffer Object {
   mat4 mvp[];
   uint layer;
};

void main() {
   #ifdef GL_ARB_shader_viewport_layer_array
      gl_Layer = layer;
   #endif
   gl_Position = mvp[draw_id] * vec4(vert_pos, 1.0);
}
```

Listing 4.2: Multi-draw vertex shader for depth pass

## 4.2 Tile-based rendering

Lighting pass accumulates the light for each pixel, and this is where we need to have shadow maps ready to sample. Shading techniques approach this step differently: in forward shading we shade the geometry at the same time we submit it, in deferred we defer, or delay, the lighting pass after we have accumulated geometry data in special framebuffer (also called gbuffer). Forward shading comes with multiple problems: expensive overdraw, object shaded by multiple lights, lack of buffers needed for post-process. Deferred also comes with its problems: fixed material attributes and big gbuffer taking memory and bandwidth resources. When increasing number of lights both of these start under-performing, forward because of having to shade unnecessarily, deferred because of state switches needed to render light's bounds. Olsson et al. [12] introduced tile-based light culling method, which produces per-tile light information used to shade tile's portion of screen. This method can be used with forward shading also called forward+, as well as with deferred. Compute shaders are usually used to build tile data, and can be used to compute lighting in deferred rendering as well.

We also need to decide the space we are going to do shading in. The two most sensible options are view or world space. In forward shading it's easy to use either, but in deferred we have to decide what space store the normals in. Storing normals in view space is more efficient, since we can reconstruct the z-direction and need to store just two components. However some post-processing effects need world space normals so it's up to us to decide which space is more beneficial. Our application uses world space normals, and therefore shading happens in world space as well.

### 4.2.1 Position reconstruction

Fragments position is a important information used for shading, shadow map sampling, and light culling in tiled renderer. In forward+ shading it is necessary to do a depth pre-pass to build light list. This can be expensive depending on geometry complexity. During the second pass we can get fragments position by interpolating the position from vertex shader. In deferred however, we have to store this position in gbuffer. Storing the position in view/world space as three vector component is not good idea, we would need lot of precision and it would make for a large gbuffer, wasting lot of bandwidth. Instead we can reconstruct position from hardware depth buffer. There are two ways, depending on whether we are operating in fragment shader or using compute shader. When using the rasterization pipeline we can interpolate the direction of rays going to frustum corners in desired space. The resulting direction is multiplied by linearized depth and added camera position gives us fragments position in world space. To linearize depth we need to convert it from texture space $z_t$ to view space $z_v$. We need to do inverse of depth transform of view to NDC space, using 2.3 we get result shown in 4.1. In compute shader easiest choice is to construct texel's position in screen space and then use inverse transform of view-projection. We also need to apply perspective divide.

$$\begin{aligned} z_n &= 2 \cdot z_t - 1 \\ z_v &= \frac{2nf}{f + n - z_n(f - n)} \\ z_v &= \frac{A}{1 - z_t B}, A = n, B = \frac{f - n}{f} \end{aligned} \tag{4.1}$$

### 4.2.2 Light model

For local light model I've chosen normalized Blinn-Phong shading model for specular and Lambertian for diffuse reflections. Blinn-Phong is similar to Phong shading model, but produces longer specular highlights at grazing angles. Normalization in this context tries to ensure that the specular exponent should have no effect on the total reflectance. This results in less intense reflections with low specular factor, and stronger reflections with high factor.

$$RDF = (N \cdot H)^n (N \cdot L) \frac{n+6}{8\pi} \tag{4.2}$$

For point lights we also need attenuation model. In real life the point light intensity falls of with squared distance. However, with this model, the luminous intensity never reaches zero. To keep the number of shaded computations down, we try to limit the point light radius, so we can skip the computation for pixels further than this radius. We can force the intensity to zero using cutoff variable as shown in 4.3. Alternative is to use more slightly more linear model shown in 4.4.

$$k = \frac{1}{1+r^2},$$
$$Att_1(x) = \begin{cases} \frac{1-k}{1+x^2} - \frac{k}{1-k} & \text{if } x < r \\ 0 & \text{if } x > r \end{cases} \tag{4.3}$$

$$Att_2(x) = \begin{cases} (1 - \frac{x}{r})^2 & \text{if } x < r \\ 0 & \text{if } x > r \end{cases} \tag{4.4}$$

## 4.3 Light culling

First we need to calculate frustum. Top, bottom, left and right planes can be calculated from the projection matrix and tile offset. For near and far planes we have to find minimum and maximum depth of a tile. One way to do this is for each thread do atomic min/max over texel depth. If the depth is float we need to first convert it with `floatBitsToUint()` and then do atomic operations, since compute shaders don't support atomic float operations yet. Then we wait for all threads to finish. This way of finding min/max depth is sequential and doesn't really take advantage of parallel execution. One possible optimization is to do parallel reduction in a separate pass. Once we have min/max depth we can construct frustum. Only a single thread needs to do this, and then use shared variable for other threads in tile to read. Once we have frustum, we synchronize and we can start culling. It's easier to do in view space, so we can pre-compute lights position in view space or do it in shader. We will construct a loop so that each thread will cull single light per loop. With a work-group size 16 times 16 that's 256 lights culled per loop. If the light is in frustum we add it to shared light list. We wait for all threads to finish and then we can start shading. Each thread will shade single pixel, looping over the lights and accumulating color, and then storing the color in output image.

Presentation by Gareth T. [9], shows culling optimizations. One problem is that on depth discontinuities single frustum will span long distances, producing false positives for lights in the middle which might not shade any pixel. Calculating two min/max intervals, one for each split, will produce two frustums we can test against, with the lights in middle properly culled away. This should lower number of lights and bring the shading costs down. Constructing AABB around frustum and testing against light radius before frustum culling was also shown as advantageous.

## 4.4 Shadow map sampling

In order to sample shadow map we need to know position of currently shaded fragment in shadow map texture space. In classic forward shading we can do calculations in vertex shader, and pass the interpolated result to fragment shader. This could be done by calculating the world space position and the transforming to shadow textures space. The complexity depends on vertex count of our scene, which may or may not be desirable. With forward+ we have to calculate per fragment, since lights change per tile. Per vertex calculation is not possible in deferred shading, but for special cases (such as sun light) we can create another texture in gbuffer, output shadow term and use it as a mask for light intensity. Otherwise we have to calculate shadow space position per fragment. This can be done by reconstructing world space position of fragment (shown in 4.2.1) and then transforming from world space to shadow texture space. The sample GLSL code in 4.4 shows how to calculate shadow space coordinates from world space, using the normal offset bias.

```
vec4 shadow_space(vec3 pos_w, vec3 normal_w, vec3 light_dir_w, float ↩
    texel_s, mat4 view_proj)
{
   float cos_alpha = max(dot(normal_w, light_dir_w), 0);
   float sin_alpha = sqrt(1 - cos_alpha * cos_alpha);
   float off = texel_s * max(sin_alpha, 0.5);
   return view_proj * vec4(pos_w + normal_w * off, 1.0);
}
```

Listing 4.3: Shadow map coordinates calculation function

If we are using cascade shadow maps, we will first need to find the cascade the fragment is in. This is usually computed by stepping linearized depth from depth buffer. We then compute the lowest split index that the fragment is ensured to be in. We can then test whether fragment is in lower split, if it is we sample the lower split. This has a positive effect on quality, while the cost is low and constant as we test only single lower split. The code shown in 4.4 shows how to do this.

```
float casc_shadow_term(vec3 pos_w, vec3 normal_w, vec3 light_dir_w, float cam_depth)
{
   float sp = 0;
   for (int i = 0; i <= sun_light.max_split; ++i)
      sp += (step(sun_light.splits[i], cam_depth));

   // Try lower split
   int spi = int(max(sp - 1, 0));
   vec4 frag_shadc = shadow_space(pos_w, normal_w, light_dir_w, smpl_count[spi] * sun_light.scales[spi], sun_light.vp[spi]);
   // so it doesn't interfere with tests
   frag_shadc.w = 0.5;

   vec4 bounds = vec4(0.5 - 0.5*smpl_count[spi]/split_size);
   vec4 dist = abs(frag_shadc - vec4(0.5));
   if (any(greaterThan(dist, bounds))) {
      spi = spi + 1; // we didn't fit in lower split
      if (spi > sun_light.max_split) // outside of last split
         return 1;
      // Recalculate coords for next split
      frag_shadc = shadow_space(pos_w, normal_w, light_dir_w, smpl_count[spi] * sun_light.scales[spi], sun_light.vp[spi]);
   }

   return shadow_pcf(frag_shadc.xyz, spi);
}
```

Listing 4.4: Shadow split calculation function with lower split test

# Chapter 5

# Results

The task was to investigate techniques used in shadow mapping. We have analyzed problems resulting from projective aliasing, developed model to solve them and implement the methods in application. We showed how to solve depth biasing using normal offsets for variable sized kernels. Softening of shadows was implemented using percentage-closer filtering with hardware filtering, and simple pre-filterable approach was implemented with exponential shadow maps. Large scene was used to show advantages of cascade shadow mapping. The scene features dynamic level of detail terrain using tessellation. Deferred tile-based light rendering was used to cull and shade lights. However shadow maps were not computed for these. Rendering high number of shadow maps with standard textures costs a lot of bandwidth. Possible approaches are discussed in section 5.1. Modern OpenGL functionality was used when applicable, limiting the overhead of driver-application communication.

When testing percentage closer filtering produced better visual results in cases where lights comes from higher angle, but struggles when the angle is low. Normal offset method worked very well, even with variable sized kernels, however wasn't tested together with normal mapping. Exponential shadow maps easily deal with self-shadowing problem, however produce light bleeding when occluder and receiver are close. Using geometry shaders together with layered framebuffers, was not found to be faster than non-layered framebuffers. Geometry shaders seem to slow down rasterization pipeline even when outputting same number of primitives as their input. Using `GL_ARB_shader_viewport_layer_array` was not found to be faster than non-layered framebuffer. One of the reasons for this may be that the cost of object rendering was too low to make a difference. More complex scene with more dynamic geometry would be better to show difference.

## 5.1 Future work

The biggest difficulty when dealing with high number of shadow maps is memory and bandwidth limitations. Olsson et al. [13] in their work show how sparse textures are used to implement virtual shadow maps. It requires state-of-the-art light render called clustered shading. It is an extension of tile-based shading, but instead of 2D tiles works with 3D clusters. This allows them precisely cull lights, but more important, partially allocate shadow map storage as needed. In terms of filtering, hybrid frustum traced shadows use combination of shadow maps and ray tracing to produce very precise shadows. These techniques use newest hardware capabilities such as conservative rasterization. Soft-shadow mapping can together with area lights create near realistic lighting conditions.

To improve cascade shadow mapping, compute shader could be used to calculate depth bounds from depth buffer, and calculate the split distribution. This should be combined with compute shader frustum culling, so we don't have to read-back the values, creating implicit synchronization. Compute shaders can also be used for fast blurring if we choose pre-filterable method.

# Bibliography

[1] Arvo, J.: Graphics Gems. chapter Transforming Axis-aligned Bounding Boxes. San Diego, CA, USA: Academic Press Professional, Inc.. 1990. ISBN 0-12-286169-5. pp. 548–550.
URL http://dl.acm.org/citation.cfm?id=90767.90922

[2] Assarsson, U.; Moller, T.: Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*. vol. 5, no. 1. 2000: pp. 9–22.

[3] Barczak, J.: Shadow Maps Don't (Always) Need Mips. 2014. [Online, accessed 15.4.2016].
URL http://www.joshbarczak.com/blog/?p=396

[4] Bridson, R.: Fast Poisson Disk Sampling in Arbitrary Dimensions. In *ACM SIGGRAPH 2007 Sketches*. SIGGRAPH '07. New York, NY, USA: ACM. 2007. ISBN 978-1-4503-4726-6. doi: 10.1145/1278780.1278807.
URL http://doi.acm.org/10.1145/1278780.1278807

[5] DigitalRune: Shadow Acne. 2014. [Online, accessed 10.4.2016].
URL http://www.digitalrune.com/Blog/Post/1765/Shadow-Acne

[6] Donnelly, W.; Lauritzen, A.: Variance Shadow Maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. ACM. 2006. ISBN 1-59593-295-X. pp. 161–165. doi: 10.1145/1111411.1111440.
URL http://doi.acm.org/10.1145/1111411.1111440

[7] Dou, H.; Yan, Y.; Kerzner, E.; et al.: Adaptive depth bias for shadow maps. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM. 2014. pp. 97–102.

[8] Ehm, A.; Ederer, A.; Klein, A.; et al.: Adaptive Depth Bias for Soft Shadows. 2015.
URL https://pdfs.semanticscholar.org/a467/b51b107040ee517b70b735f8e607818dd71c.pdf

[9] Gareth, T.: Advancements in Tiled-Based Compute Rendering. 2015. [Game Developers Conference 2015, San Francisco, CA].
URL http://www.gdcvault.com/play/1021764/Advanced-Visual-Effects-With-DirectX

[10] Gribb, G.; Hartmann, K.: Fast Extraction of Viewing Frustum Planes from the WorldView-Projection Matrix. 2012. [Online, accessed 20.4.2016].
URL http://www.cs.otago.ac.nz/postgrads/alexis/planeExtraction.pdf

[11] Lauritzen, A.; McCool, M.: Layered Variance Shadow Maps. In *Proceedings of Graphics Interface 2008*. GI '08. Toronto, Ont., Canada, Canada: Canadian Information Processing Society. 2008. ISBN 978-1-56881-423-0. pp. 139–146.
URL http://dl.acm.org/citation.cfm?id=1375714.1375739

[12] Olsson, O.; Assarsson, U.: Tiled Shading. *Journal of Graphics, GPU, and Game Tools*. vol. vol. 15, no. issue 4. 2011-11-08: pp. 235–251. ISSN 2151-237x. doi: 10.1080/2151237X.2011.621761.
URL http://www.tandfonline.com/doi/abs/10.1080/2151237X.2011.621761

[13] Olsson, O.; Persson, E.; Billeter, M.: Real-time Many-light Management and Shadows with Clustered Shading. In *ACM SIGGRAPH 2015 Courses*. SIGGRAPH '15. New York, NY, USA: ACM. 2015. ISBN 978-1-4503-3634-5. pp. 12:1–12:398. doi: 10.1145/2776880.2792712.
URL http://doi.acm.org/10.1145/2776880.2792712

[14] Reeves, W. T.; Salesin, D. H.; Cook, R. L.: Rendering Antialiased Shadows with Depth Maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM. 1987. ISBN 0-89791-227-6. pp. 283–291. doi: 10.1145/37401.37435.
URL http://doi.acm.org/10.1145/37401.37435

[15] Rosen, P.: Rectilinear Texture Warping for Fast Adaptive Shadow Mapping. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. March 2012. ISBN 978-1-4503-1194-6. pp. 151–158. doi: 10.1145/2159616.2159641.
URL http://doi.acm.org/10.1145/2159616.2159641

[16] Segal, M.; Akeley, K.: The OpenGL Graphics System: A Specification (Version 4.5 (Core Profile) - May 28, 2015). 2015.
URL https://www.opengl.org/registry/doc/glspec45.core.pdf

[17] Zhang, F.; Sun, H.; Xu, L.; et al.: Parallel-split shadow maps for large-scale virtual environments. In *Proc. VRCIA 2006*. 2006. pp. 311–318.

# Appendices

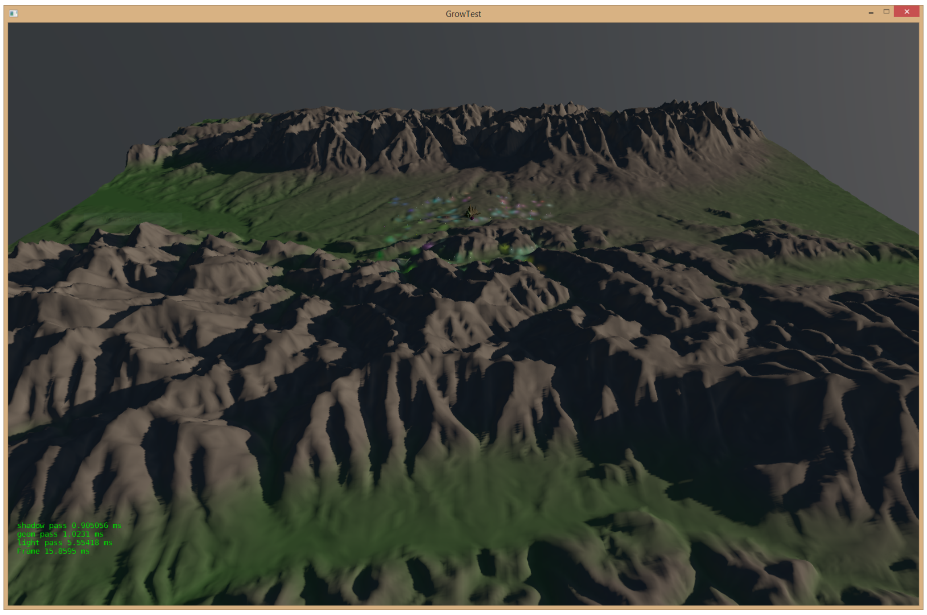
## List of Appendices

**A Figures** **28**

**B CD contents** **30**

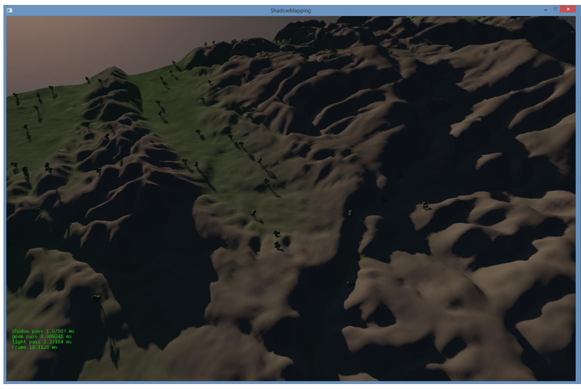
# Appendix A

# Figures



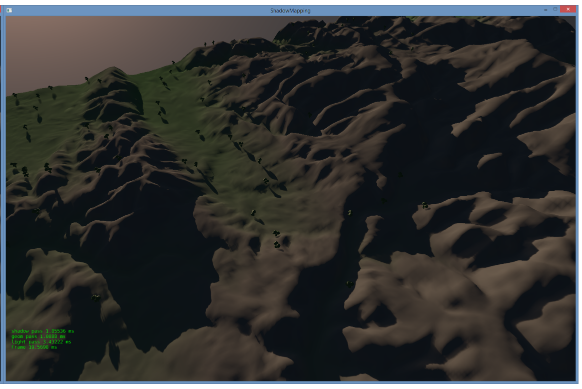
```
shadow pass 0.905056 ms
geom pass 1.0231 ms
light pass 5.55418 ms
Frame 15.8595 ms
```

Figure A.1: Rendered large-scale scene



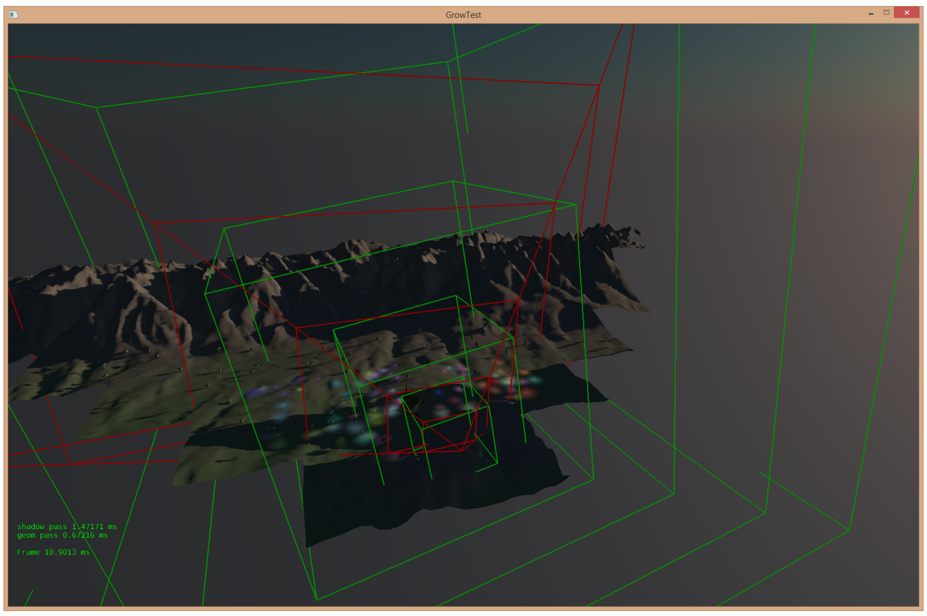
```
shadow pass 1.67507 ms
geom pass 0.909248 ms
light pass 2.37104 ms
Frame 18.1128 ms
```

Figure A.2: PCF filtering



```
shadow pass 1.85536 ms
geom pass 1.0088 ms
light pass 3.43222 ms
Frame 19.5698 ms
```

Figure A.3: Exponential shadow maps



```
shadow pass 1.47171 ms
geom pass 0.67216 ms

Frame 10.9013 ms
```

Figure A.4: Split shadow map frustums visualized

# Appendix B

# CD contents

CD contains this thesis in PDF form and latex sources used to build it are in documents/ folder. The application source codes are in source/ folder together with libraries needed to build the codes. Also included is Microsoft Visual 2015 build solution file. Run-time binaries are in binaries/ folder together with the built libraries. The root directory contains file LICENSE.txt which contains licenses of all software and 3D models used in this project. It also contains the license of this work. In the file USAGE.txt is short explanation of keyboard controls used to control the program. The video file in root directory contains short presentation of the work.