TÉCNICO LISBOA

Shadow rendering techniques for mobile devices

Henrique da Câmara Martins Araújo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Professor João António Madeiras Pereira

Examination Committee:

Chairperson: Professor Paolo Romano
Supervisor: Professor João António Madeiras Pereira
Member of the Committee: Professor Adriano Martins Lopes

January 2021

Acknowledgments

First I'd like to thank my supervisor Prof. João Madeiras Pereira for giving me the chance to develop this thesis in a topic that I so much desired, and providing me with the means necessary for it's research and development.

I'd also like to thank Samsung for providing me with the equipment necessary for the development of this thesis.

Finally I'd like to thank my family and friends for supporting me during this journey and for pushing me when needed, including my girlfriend Ana and my boss Pedro who both pushed me a lot during hard times and my parents who enabled me to pursuit a master's degree.

Abstract

With an increase in demand for games in smartphones, efficiency in rendering techniques for mobile devices is becoming more and more important. Shadows play an important role in the rendering of a scene, making it more believable, but rendering them can take a big toll in the device resources.

In this document, we present multiple solutions of rendering hard shadows. These solutions were then compared to conclude which is the most suitable algorithm for shadow rendering in a mobile environment and use it for further development.

With the base algorithm chosen, we again compared multiple techniques to improve the visual quality of the shadow produced, by introducing a penumbra to the shadow and even a variable one.

Some of these solutions were chosen, from the results of previous studies, to be developed into a mobile app that would allow to further test and verify if one or more of these solutions could viably produce a realistic shadow while having an acceptable performance in a mobile environment.

From these solutions we noted that Percentage-Closer Filtering in junction with Percentage Closer Soft Shadows were the most viable solution, being able to provide variable soft shadows with a good performance, while also being easily adaptable into different scenes.

Keywords: Real-Time Shadows; Mobile Shadows; OpenGL ES; Shadow Maps; Variable Soft Shadows; Mobile Environment.

Contents

Acknowledgments **2**

Abstract **4**

List of Tables **9**

List of Figures **11**

Acronyms **13**

1 Introduction **15**
1.1 Objectives 15
1.2 Contributions 16
1.3 Document Structure 16

2 Related Work **18**
2.1 Understanding Shadows 18
2.1.1 Umbra and Penumbra 18
2.1.2 Hard and Soft Shadows 19
2.2 Shadow rendering techniques 20
2.2.1 Shadow Mapping 20
2.2.2 Shadow Volumes 21
2.2.3 Ray Tracing 22
2.2.4 Shadow Volumes vs. Shadow Maps vs. Ray Tracing 23
2.3 Improving shadow mapping 24
2.3.1 Mid-point shadow mapping 24
2.3.2 Warping shadow maps 25
2.3.3 Z-Partitioning shadow maps 26
2.3.4 Adaptive Partitioning for shadow maps 27
2.4 Shadow Mapping for Soft Shadows with fixed penumbra 28
2.4.1 Percentage-Closer Filtering 28
2.4.2 Variance Shadow Maps 29
2.4.3 Convolution Shadow Maps 30
2.4.4 Exponential Shadow Maps 31
2.4.5 VSM vs. CSM vs. ESM 31
2.5 Shadow Mapping for Soft Shadows with variable penumbra 32
2.5.1 Percentage Closer Soft Shadows 32
2.5.2 Summed Area Tables 32
2.5.3 Mipmapping 33
2.6 Mobile environment 34
2.6.1 Shared memory 34
2.6.2 Dynamic clock and voltage scaling 34
2.6.3 CPU big.LITTLE Architecture 34
2.6.4 Tile-based Rendering 34
2.6.5 Other mobile technologies 35

3 Implementation **37**
3.1 Development Environment 37
3.2 Shadow Mapping Implementation 39
3.2.1 Shadow Mapping 39
3.2.2 Percentage-Closer Filtering 42
3.2.3 Variance Shadow Maps 43
3.2.4 Exponential Shadow Maps 45
3.2.5 Percentage Closer Soft Shadows 46

4 Evaluation and Results **49**
4.1 Evaluation Methodology 49
4.1.1 Tools and metrics 49
4.1.2 Test Scenes and Testing Methodology 51
4.2 Results 53
4.2.1 Preliminary Test 53
4.2.2 Performance comparison 55
4.2.3 Testing VSM 58
4.2.4 PCF vs ESM 61

5 Conclusions and Future Work **66**
5.1 Major Contributions 66
5.2 Future Work 67

List of Tables

List of Tables

1 Average FPS with PCSS (7x7 tap), 1080*2220 map 53
2 Average FPS with VSM (5x5 filter), 1080*2220 map 59
3 Average FPS with VSM (5x5 filter) and a 3x3 PCSS tap 59
4 Average FPS with VSM, ESM and PCF with different settings 60
5 Average FPS with different shadow map sizes 62
6 Average FPS with different shadow map sizes 64

List of Figures

List of Figures

1 Shadow representation 18
2 Representation of umbra and penumbra 18
3 Representation of a hard shadow (top image) and a soft shadow (bottom image) 19
4 Inner workings of Shadow Mapping. 20
5 Surface acne. 20
6 Light leakage. 20
7 Shadow Volume demonstration. 21
8 Shadow Volume pass using depth test. 21
9 Improving shadow volumes performance. 22
10 Ray tracing demonstration. 22
11 Midpoint working example. 24
12 Example of Midpoint Shadow Map working and the problem near the silhouette. 24
13 Aliasing which occurs close to the Camera. 25
14 Perspective Shadow Maps working example. 25
15 LisPSM with optimal n. 26
16 Logarithmic Z-partitioning with fitted depth bounds. 26
17 Using Warping and Z-Partitioning. 27
18 Adaptive Shadow Map. 27
19 PCF with regular (left) and irregular (right) sampling. 28
20 PCF bleeding for large PCF kernel. 28
21 VSM rounding demonstration. 29
22 VSM with and without shadow receivers rendered into shadow map. 29
23 VSM with a threshold of 0 and 0,2. 29
24 Normal VSM vs. Layered VSM. 30
25 CSM rounding demonstration. [3] 30
26 CSM Ringing effect with M=2 (left) and M=8 (right). [3] 30
27 ESM rounding demonstration. [3] 31
28 VSM vs. CSM vs. ESM. 31
29 Inner workings of PCSS. 32
30 Representation of the developed app. 37
31 Render with simple shadow mapping 42
32 Render with PCF 43
33 Render with VSM 45
34 Renders using PCSS 47
35 sponza.obj illustration 51
36 dragon.obj illustration 51
37 bunny.obj illustration 52
38 Memory information 55
39 Shader resources usage 56
40 ALUs and EFUs per fragment 56
41 CPU utilization 57
42 Sponza render using VSM with 7x7 box filter, 7x7 PCSS tap and 1080*2220 map 58
43 Renders using 7x7 PCSS tap and 1080*2220 map 58
44 Renders using 7x7 PCSS tap and 1080*2220 map 59
45 Renders using 7x7 PCSS tap 3x3 PCSS tap 59
46 Image quality comparison between solutions 60
47 Image comparison between PCF and ESM in Sponza 61
48 Image quality comparison between PCF with diferent PCSS taps 61
49 Image quality comparison between PCF with diferent shadow map sizes 62
50 Image quality comparison between PCF and ESM 63
51 ESM with different shadow map sizes 63
52 Comparing ESM with different settings 64

Acronyms

ALU Arithmetic Logic Unit. 11, 50, 56, 57

CPU Central Processing Unit. 6, 11, 34, 49, 57

CSM Convolution Shadow Maps. 6, 11, 30–33, 66

CSV Comma Separated Values. 49

EFU Elementary Function Unit. 11, 50, 56, 57

ESM Exponential Shadow Maps. 6, 7, 9, 11, 16, 31, 32, 37, 38, 45, 47, 53–56, 59–61, 63, 64, 66

FBO Frame Buffer Object. 39, 40

FOV Field of View. 51, 52

FPS Frames per Second. 9, 16, 49, 50, 53, 54, 59–64, 66

GDDR Graphics Double Data Rate. 34

GPU Graphics Processing Unit. 34, 35, 49, 50, 57

IP Semiconductor intellectual property core. 34

LiSPSM Light-Space Perspective Shadow Mapping. 26

LPDDR Low-Power Double Data Rate. 34

PCF Percentage-Closer Filtering. 4, 6, 7, 9, 11, 16, 28, 32, 37, 38, 42–44, 46, 47, 53–56, 59–63, 66

PCSS Percentage Closer Soft Shadows. 4, 6, 9, 11, 16, 32, 33, 37, 38, 46, 47, 53, 58–63, 66

PSM Perspective Shadow Maps. 11, 25, 26

RAM Random Access Memory. 50

SAT Summed Area Tables. 6, 32, 33

SoC System on a chip. 49

VSM Variance Shadow Maps. 6, 7, 9, 11, 16, 29–32, 37, 38, 43, 45, 47, 53, 55, 56, 58–61, 66

1. Introduction

Since the development of the first mobile devices, there has been a huge increase in their usage. These devices started out as simple means for communication, but they became more and more sophisticated as the demand for newer and better devices rose.

At some point they became a necessity in our lives, allowing us to become closer to our friends and family, no matter the distance, to manage our bank accounts, to be able to immediately buy something we need or to search for anything we want to, and to entertain us, with an ever so increasing offer of mobile games.

As the market for games in mobile devices increases, so does the need to improve the image quality in these games, to make them more realistic, good looking, and ultimately more appealing.

One way we can improve the image quality is by drawing shadows as realistically as possible. To achieve this, we need to find the best algorithm for shadow rendering that is both the most accurate possible, but also not to demanding to the mobile device, in order to preserve the frame rate of the displayed scene and the device's battery.

1.1 Objectives

The main objective of this research is to compare multiple existing base algorithms for achieving shadow rendering in a mobile environment, choose the most suitable one, and compare it's multiple adaptations to achieve realistic shadows by comparing the measurements of their performance and realism, to conclude which would be the best solution in a mobile environment. The goal is to answer the following questions:

- Which basic shadow rendering algorithm is best to be adapted into rendering realistic shadows for mobile phones?
- Which adaptations are there and which should we use?
- Which of the chosen adaptations perform the best?
- What is the reason behind the performance of each chosen adaptation?
- Is any of the solutions performance acceptable for real time rendering?

To the author's knowledge, the majority of the research being done are focusing on improvements to the already existing solutions, or development of new techniques to render better and faster shadows. There is not much work on importing the already existing solutions into the mobile environment to improve the shadows rendered on it.

As such we will be developing these solutions in the mobile environment and test them, as to gain more information about them.

1.2 Contributions

To try to reach the objectives defined, a mobile app for Android was developed, in partnership with Samsung Research UK.

After studying the related work on the subject and finding out what is the current state of the art in shadow rendering, some of the solutions were chosen for development, namely, Shadow Mapping for the base solutions, together with Percentage-Closer Filtering, Variance Shadow Maps, Exponential Shadow Maps and Percentage Closer Soft Shadows.

These solutions were further tested and compared with each other, since the main objective of this study is to find which of the current used solutions is the most suitable for a mobile environment. Some performance improvement was implemented to these solutions but the focus was guided towards comparing each of them.

In these tests multiple metrics were gathered, like Frames per Second, which was recorded by the developed app, or others like the memory write total (in bytes) or percentage of time in which the shaders were working, these available through the Snapdragon Profiler application developed by Qualcomm.

From the data gathered, we could do extensive analysis to be able to reach conclusions about the following topics:

- Preliminary testing - The first tests done recorded the average framerate that each solution achieved under different settings and in different scenes. This gave us valuable insight in which scenes were more or less demanding and which solutions could perform better under what circumstances.
- Performance testing - After the preliminary test, a more extensive performance test using the Snapdragon Profiler app was done, using the most demanding scene and under demanding settings, so that a more in depth analysis could be done to understand why each solution performed as it did.
- Variance Shadow Maps testing - Testing VSM, which is one of the solutions implemented, was necessary to understand if this solution could achieve a good quality shadow under a reasonable performance, and later visually compare it with other solutions at a same performance level to understand if it was a viable solution or not.
- Comparing Exponential Shadow Maps against Percentage-Closer Filtering - The final testing done compared ESM with PCF to understand which of these solution was in fact more suitable and under which circumstances.

1.3 Document Structure

In this document, multiple steps were presented to reach the end goal, correlating to the research development and testing done along side the writing of the document.

The starting point, in section 2 we explain the concept of shadows, present the already existing solutions for simple shadow rendering, analyze them and choose one for development. From the one chosen we look at ways to improve the shadow into a more realistic one, and choose some of these solutions to implement.

After learning all the information needed, in section 3, the implementations of the different solutions chosen for development are described.

The different tools used and metrics needed are defined in section 4.1, and the results obtained from those are presented, compared and discussed in section 4.

After comparing and discussing the results in section 4, the conclusion obtained in it were presented in section 5.

2. Related Work

To be able to viably produce realistic shadows for the mobile environment we first need to understand how shadows work, then compare the multiple existing solutions for basic shadow rendering, already used in desktop environments, and choose one for development.

Some mobile environment intricacies also need to be discussed to better understand it's abilities and needs, so that we can understand the adaptations that should be done to our solutions

2.1 Understanding Shadows

A shadow can be defined by a point in which the direct path to the light is partially or fully obstructed by an object, so the area in shadow is the entire set of points with an obstructed direct path to the light, as represented in figure 1.

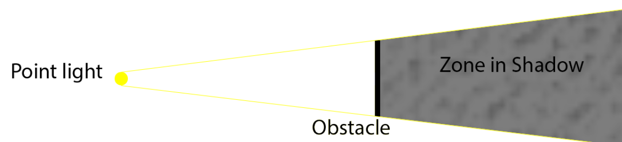


Figure 1: Shadow representation

Any part of a surface that intersects with this volume is thus in shadow, since any point in the surface has it's path to the light obstructed by the object.

2.1.1 Umbra and Penumbra

In real life, shadows can have a soft edge which is a zone of the shadow where a point is only partially occluded from the light source, when the light source is an area and not just a point. We call this zone the penumbra (light grey part of the shadow in figure 2).

Similarly, the zone of the shadow in which a point is fully occluded from the source is called the umbra (dark grey part of the shadow in figure 2).

Together, the umbra and the penumbra make up the entirety of the shadow.

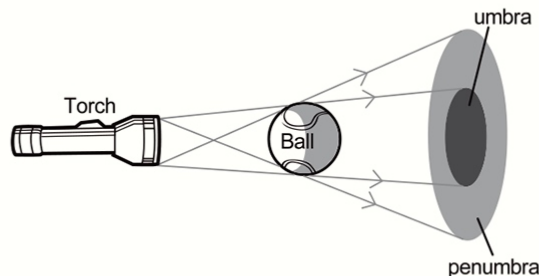


Figure 2: Representation of umbra and penumbra

2.1.2 Hard and Soft Shadows

Depending on the light source and the distance between the surface, the obstructing object and the light source, the projected shadows can have different penumbra sizes.

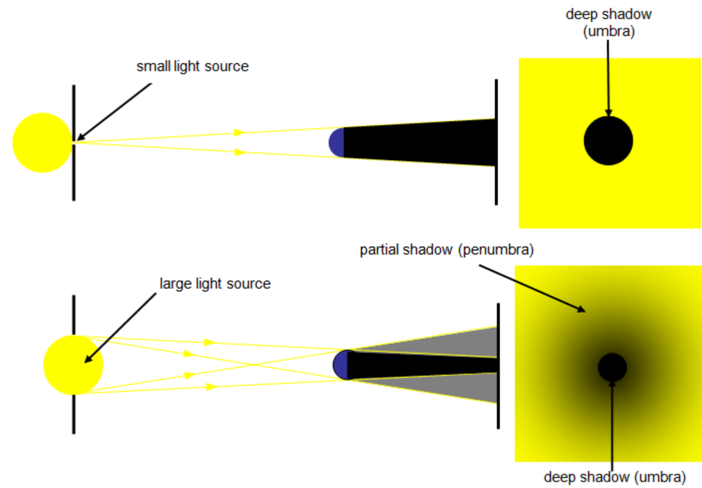


Figure 3: Representation of a hard shadow (top image) and a soft shadow (bottom image)

If the light source is far enough and the obstructing object close enough to the surface of the shadow, the shadow has no penumbra. We call these Hard Shadow.

On the other hand we have a Soft Shadow, which is a shadow with a visible penumbra. This type of shadows has a soft edge, where the shadow gets progressively lighter the farther from the umbra it is.

2.2 Shadow rendering techniques

There are multiple ways in which we can render shadows in a scene.
Between those, the three most commonly used are Shadow mapping, shadow volumes and ray tracing, due to their efficiency, reliability or accuracy.

2.2.1 Shadow Mapping

Shadow Mapping[28] consists in rendering the Scene from the point of view of the light source. We only need to store the depth for each pixel, since we only need information that tells us if a given point is in shadow or not. The generated image represents the depth of the lit points. If any given point has a higher depth than the generated image, that point is in shadow, as demonstrated in figure 4 [20].

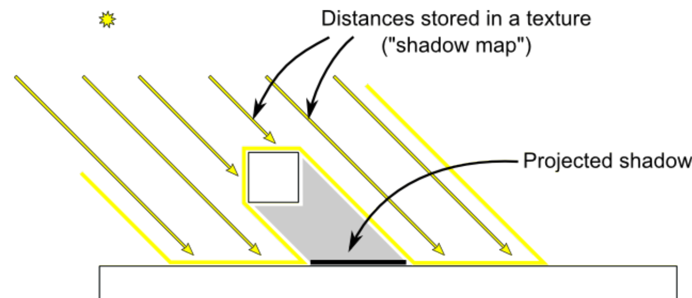


Figure 4: Inner workings of Shadow Mapping.

There is a need to introduce a tolerance threshold for the depth test which does not guarantee a perfect representation of the shadow.

If the threshold is too small there will be a self-shadowing problem (usually called z-fighting), in which parts of a surface that in the shadow map are represented as one pixel will get progressively more shadow the deeper they are until it reaches the next pixel, creating a line pattern across the surface. An example of this can be seen in figure 5.

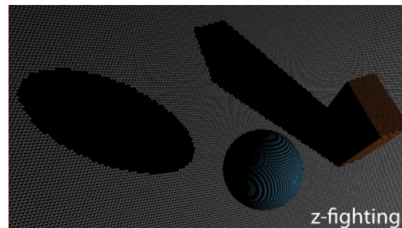


Figure 5: Surface acne.

On the other hand, if the threshold is too high some parts of the shadow wont be correctly shadowed and will be regarded as a point in light, i.e. light leakage occurs (example in figure 6).

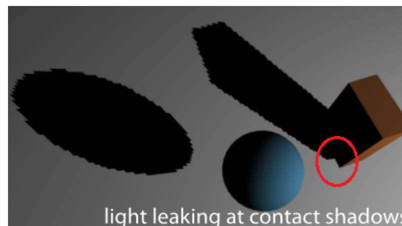


Figure 6: Light leakage.

This forces us to fine tune the tolerance threshold for each scene, while still getting some shadowing problems.

The advantage that the Shadow Mapping brings us is the adaptability of the base algorithm, which allows us to implement solutions that can, at least, mitigate the shadow mapping problems.

2.2.2 Shadow Volumes

The shadow volumes technique[5] creates a space in shadow. For each triangle, it traces lines with each vertex and the point of light creating a pyramid shaped volume that represents the volume in which all the points are in the shadow of the triangle.

To test if a given point is in shadow, assuming the eye is located in a non shadowed zone, we shoot a ray from the eye to the point. If the ray enters more shadow volumes than it leaves the point is in shadow, the point should be lit otherwise, as demonstrated in figure 7 [8].

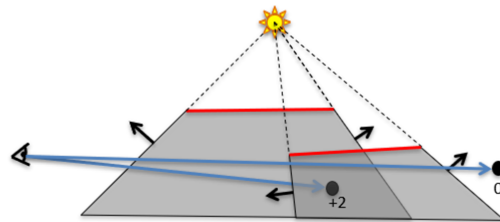


Figure 7: Shadow Volume demonstration.

This can be achieved without ray tracing, by using the normal rasterization. To do this, we render the triangles of the shadow volume to the stencil buffer. First, for the front facing triangle we increment the stencil buffer in case the object in a pixel is deeper than the triangle. Otherwise we decrement the stencil buffer for the back facing triangles (figure 8 [8]).

This way if an object is inside the shadow volume, the buffer will be incremented since the z-test will pass for the front-facing triangles and will not be decremented since it fails the z-test for the back-facing triangles. Then all the pixels with an incremented value will be shaded.

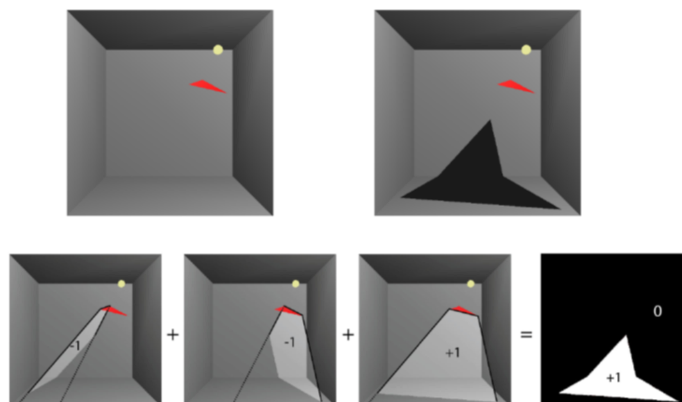


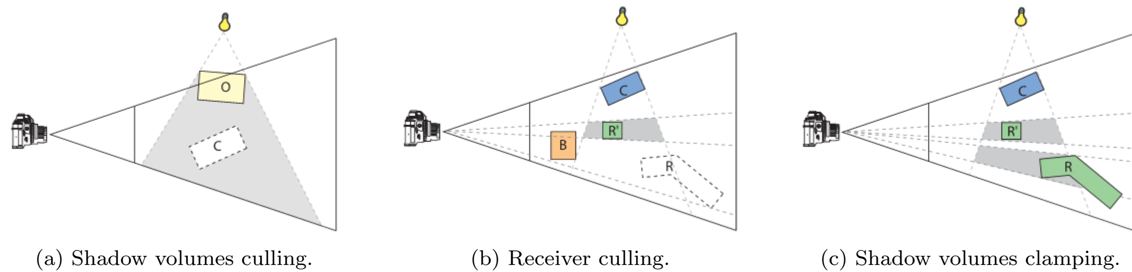
Figure 8: Shadow Volume pass using depth test.

This solution presents itself with a problem. If the eye is located inside a shadow volume the stencil buffer will not be correctly incremented/decremented,since one or more of the faces of the shadow volume are not visible.

To solve this we can use the z-fail algorithm. In this algorithm we invert the z-test and the incrementing and decrementing, so for the front-facing triangles instead of incrementing the stencil buffer if the depth test passes we decrement it if fails,and for the back-facing triangles we increment it if the depth test passes. This way we achieve the same end result, except now the scene is correctly shadowed in case the eye is located inside any shadow volume.

The Shadow Volumes algorithm also has it's problems, it's not as easy to work upon. It makes transforming the resulting shadow into a soft shadow harder. It is also quite costly in performance, since we create a volume for each of the scene's triangles.

We can use solutions like shadow volumes culling (fig. 9(a) [8]), shadow volumes clamping (fig. 9(b) [8]) or receiver culling (fig. 9(c) [8]) to better the performance.



(a) Shadow volumes culling. (b) Receiver culling. (c) Shadow volumes clamping.

Figure 9: Improving shadow volumes performance.

2.2.3 Ray Tracing

Using ray tracing to draw shadows consists of shooting a ray for each pixel.

When a ray is shot, we check for the first collision with an object of the scene and from the point of intersection between the ray and the object we shoot anew ray called a shadow feeler to the direction of the light. If the ray intersects with another object than the pixel should be shadowed.

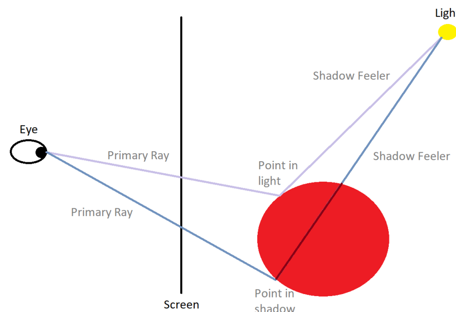


Figure 10: Ray tracing demonstration.

This algorithm is a very close representation of real life light behaviour, being very accurate, but extremely demanding.

Since there are usually millions of pixels in the screen, which acutes to the same amount of primary rays shot which will each check for intersections with the objects of the scene and might end up with a shadow feeler being shot, which in turn will also check for intersections with the objects of the scene. This makes it a very heavy application, in regards to performance.

2.2.4 Shadow Volumes vs. Shadow Maps vs. Ray Tracing

Ray tracing is too much demanding. For this reason we can rule out ray tracing as a viable solution to draw shadows using mobile devices currently on the market.

Fidelity wise, shadow volumes produce a better result since it does not have problems like light leakage or shadow acne, and it has pixel perfect quality, as opposed to shadow mapping which a pixel in a shadow map might not correspond to a pixel from the camera view, leading to some imperfections.

In terms of efficiency, shadow mapping is faster than shadow volumes, since it only has to render the distance from the light to the objects of the scene, opposed to creating multiple shadow volumes that must be checked multiple times and add much more complexity to the scene.

Shadow maps are also more versatile than shadow volumes, since we have an image in which we can work on, as opposed to volumes, which add a layer of complexity to be able to adapt and achieve soft shadows.

For this reason, the use of shadow volumes was discarded in favor of using shadow mapping with some adaptations.

2.3 Improving shadow mapping

As we discussed before, the focus of the solutions presented will be Shadow Mapping, which can be worked upon and be improved to deliver better performing and better looking shadows.

2.3.1 Mid-point shadow mapping

Mid-point shadow mapping [31] consists of storing both the first and second level depths on the shadow map, i.e. the depth to the first intersection and for the second one.

With both these values we can have multiple biases instead of a fixed one, a bias for each pixel of the shadow map represented by the middle point of the first level depth value and the second one.

We can see how this solution works in figure 11 [8].

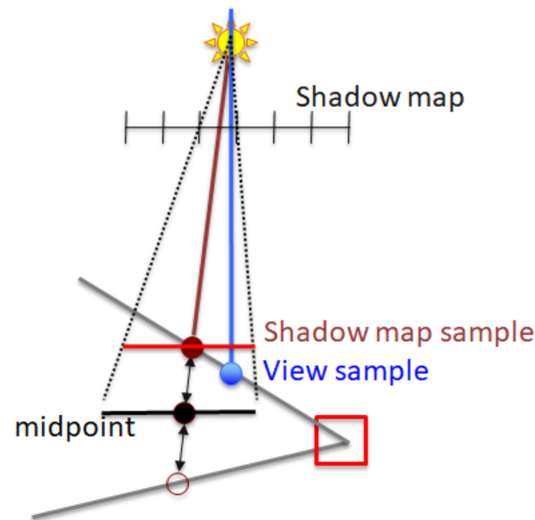


Figure 11: Midpoint working example.

This however, won't completely solve the problem, with problems still appearing near silhouettes, as seen in figure 12 [8].

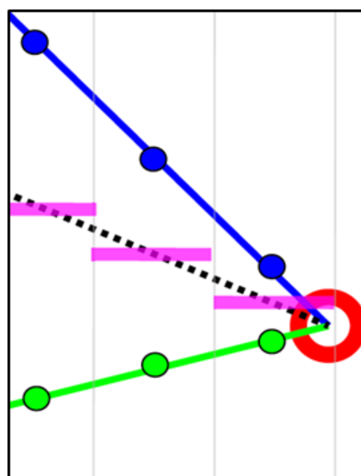


Figure 12: Example of Midpoint Shadow Map working and the problem near the silhouette.

2.3.2 Warping shadow maps

As the representation of the shadow is discrete, shadow mapping is susceptible to jagged shadows (which depends on the resolution used on the image rendered). This introduces aliasing in shadows close to the observer (as seen in figure 13 [29]) and might have too much detail for shadows that are far from it.

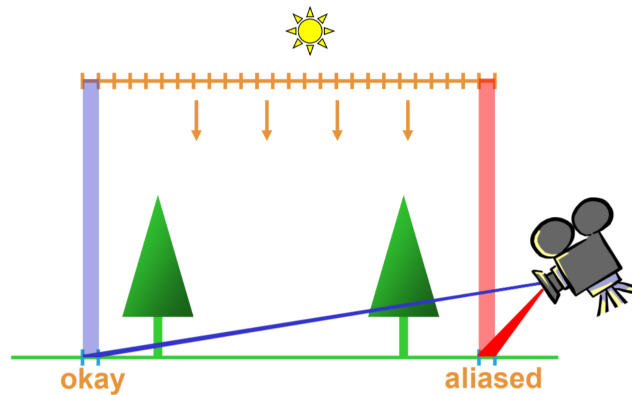


Figure 13: Aliasing which occurs close to the Camera.

To solve this problem we can redistribute the values in a shadow map to give more detail to parts of the shadow that are closer to the camera and less detail to shadows that are further away.

Perspective Shadow Maps [23] warps the shadow map to match the warpness of the scene. To do this we transform the scene to post-perspective space with the camera matrix including our light source, then we proceed with our shadow mapping generation (figure 14 [29]).

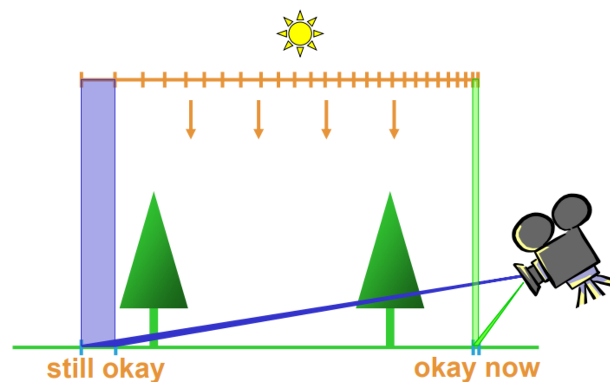


Figure 14: Perspective Shadow Maps working example.

But this comes with it's problems, points behind the camera sometimes need to be included since they can cast shadows in the viewing frustum but with the space being warped, the points are projected beyond the infinity plane.

Generating two shadow maps can solve this, generating the first one as described before and a new one that looks beyond the infinity plane, virtually moving the camera view point backwards so that the points that produce a shadow in the viewing frustum are all inside the transformed camera frustum, but we reintroduce perspective aliasing with this.

Another severe problem is the uneven z-distribution, which might introduce weird artifacts in shadows far away from the camera.

Light-Space Perspective Shadow Mapping [30] gets rid of PSM problems by defining the perspective frustum relative to the light space. LiSPSM finds a near and far planes for the scene and chooses a warping strength (n). The n is found according to the following equation:

$$n_{opt} = z_n + \sqrt{z_f z_n} \tag{1}$$

This results in a nice balance between PSM and a normal shadow map (figure 15 [29]), with more fidelity than a normal shadow map close to the camera and than a PSM from further away.

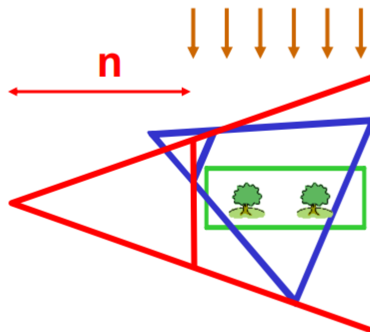


Figure 15: LisPSM with optimal n.

Warping the shadow map still has problems dueling with frusta case and only works if the z-range visible from light is big enough.

2.3.3 Z-Partitioning shadow maps

We can also fight undersampling by using partitions of the camera space [32] instead of treating it like a whole.

Z-Partitioning divides the view frustum into multiple sub-frusta, then we can calculate a shadow map for each of those subdivisions. This solution works where warping fails, since it works similarly to a normal shadow map. NVIDIA ShadowWorks [18] Cascaded Shadow Maps use this technique.

The size of the divisions need to be chosen. The optimal way of choosing the size is to use logarithmic sizes, as seen by Andrew Lauritzen et al. [15], but we might get empty areas. For this we need to adapt depth bounds to the scene by analysing it and getting the min and max Z (figure 16 [29]).

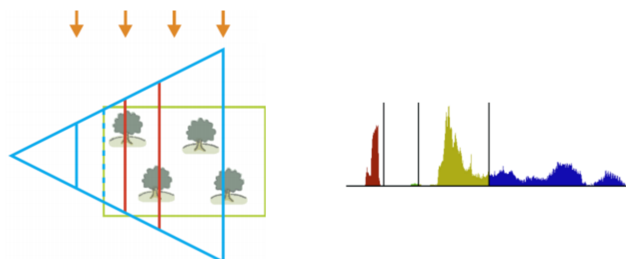


Figure 16: Logarithmic Z-partitioning with fitted depth bounds.

We can additionally use warping and z-partitioning together to achieve the best results possible (figure 17 [29]).

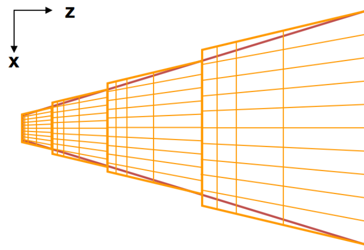


Figure 17: Using Warping and Z-Partitioning.

2.3.4 Adaptive Partitioning for shadow maps

Adaptive Partitioning [10] solves projection aliasing by adaptively splitting the shadow map using edge split using a quad tree, since we only need a high resolution at the edges of the shadow map.

We can see an example of Adaptive partitioning dividing the shadow map in figure 18 [29].

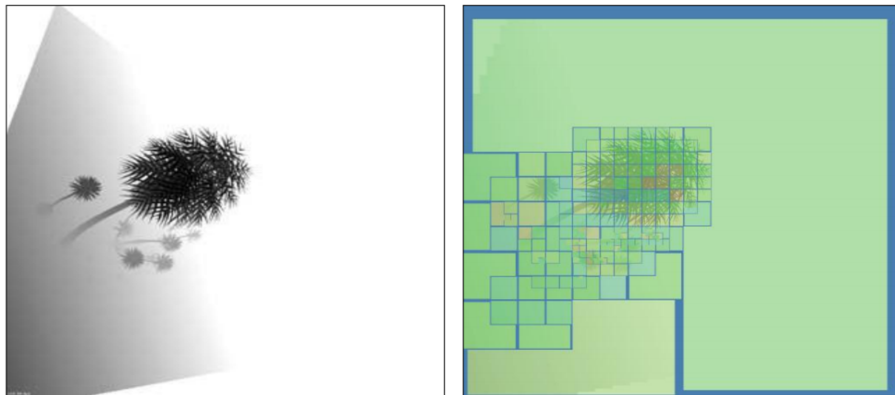


Figure 18: Adaptive Shadow Map.

2.4 Shadow Mapping for Soft Shadows with fixed penumbra

Due to Shadow Mapping versatility, the shadow map generated can be adapted and changed so it is possible to have a shadow with a penumbra in a given scene.

The solutions for this are many, with different degrees of realism and performance, usually one being the trade off of the other.

2.4.1 Percentage-Closer Filtering

PCF achieves a soft shadow by sampling multiple points of the shadow map instead of one, getting the values of a grid, calculating the shadow that each of those pixels would result and averaging these results.

Since the only locations of the shadow that benefit from PCF is the penumbra, we can use edge detection or check the difference of all the obtained pixels to apply PCF only if there is an edge or if the discrepancy is big enough, respectively.

To get rid of some banding we can trade the regular sampling (left of figure 19 [3]) for irregular sampling (right of figure 19 [3]), randomly picking the samples in the defined grid of the shadow map. This way we trade the banding problem for noise.

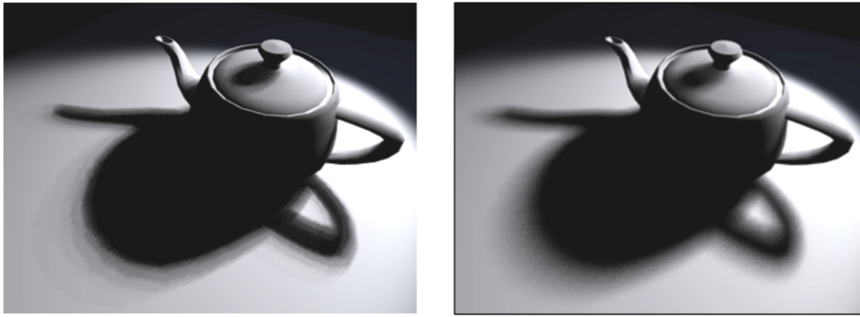


Figure 19: PCF with regular (left) and irregular (right) sampling.

PCF also accentuates the self-shadowing issue. There are three ways we can solve this, by using a depth gradient, by rendering the midpoints into the shadow map which still requires a depth bias for thin objects and render back faces into shadow maps, which only works for closed objects and has some light bleeding for large PCF kernels (figure 20 [3]).

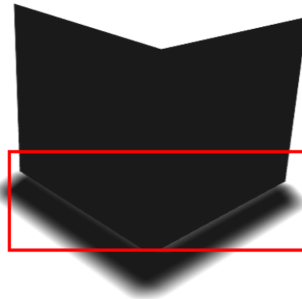


Figure 20: PCF bleeding for large PCF kernel.

2.4.2 Variance Shadow Maps

VSM [17] changes the way we store the shadow map. Instead of storing only depth value, it store the depth value together with it's square value. The shado map is then filtered so we can fetch a position on the VSM and then use Chebyshev's inequality (2) to determine the percentage of which the pixel is in shadow.

$$P(d < z) <= max(\sigma^2/(\sigma^2 + (d - \mu)^2, (d < \mu))) \tag{2}$$

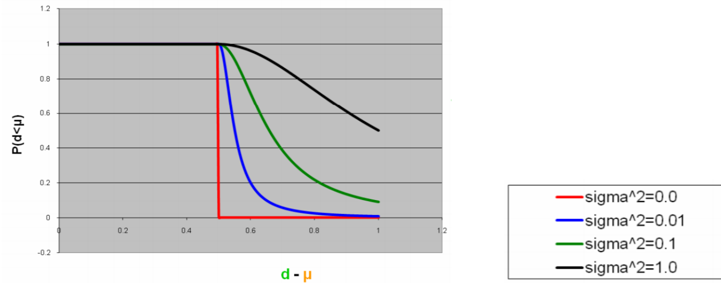


Figure 21: VSM rounding demonstration.

We can handle self shadowing problems by clamping σ^2 to a small minimum variance parameter, so if the variance is too small and wavery this minimum value will make it constant.

To avoid bad shadowing when blurring the shadow map, we need to render every object into the shadow map, since there can be some light leakage and the shadows might appear to thin and light leakage occurs (like demonstrated in figure 22 [3]).

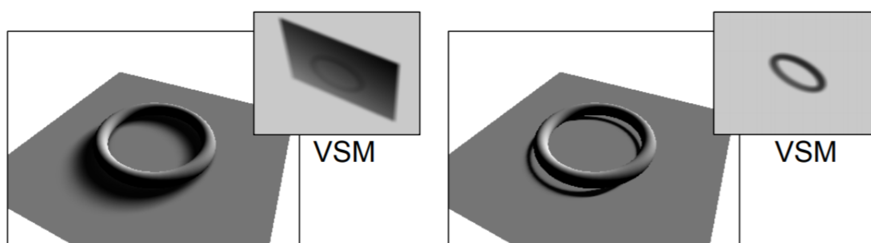


Figure 22: VSM with and without shadow receivers rendered into shadow map.

Light bleeding might also appear if 2 overlapping occluders have a big distance between them (left image of figure 23 [3]). The use of a threshold to remap shadow intensity can be used to mitigate this problem (right image of figure 23 [3]).

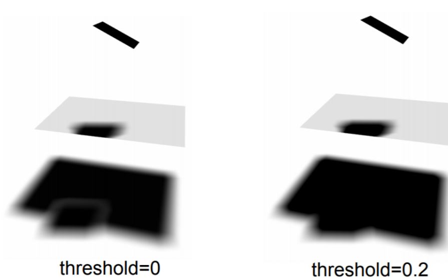


Figure 23: VSM with a threshold of 0 and 0,2.

Layered VSM [14] also tackles this problem by partitioning the shadow map into multiple layers defined by depth ranges and using clamping to each range, like in figure 24 [3].

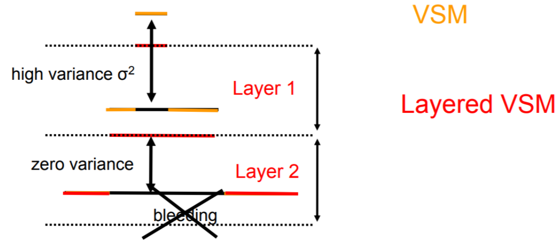


Figure 24: Normal VSM vs. Layered VSM.

2.4.3 Convolution Shadow Maps

CSM [1] approximates the depth values of a normal shadow map by transforming it into a wave function so that we can use Fourier (3) to deconstruct it and return a blurred map when rendering the shadows.

$$f(d,z) \approx \frac{1}{2} + 2\sum_{k=1}^{M} \frac{1}{ck} cos(ck.d) sin(ck.z) - 2\sum_{k=1}^{M} \frac{1}{ck} sin(ck.d) cos(ck.z) \tag{3}$$

It assumes that for each pixel of the shadow map there is a linear depth z, which needs to be normalized to be [0,1], since this is needed for Fourier to work.

CSM can have light bleeding issues but the higher the number of passes (higher M) the lower is the light bleeding problem.

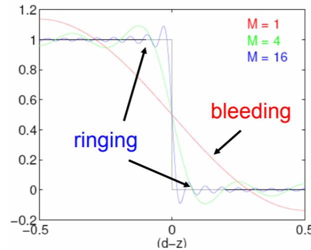


Figure 25: CSM rounding demonstration. [3]

Another problem present is the ringing effect (figure 26 left) which can be mitigated by multiplying each k-th sum by $exp(-a(k/M)^2)$, flattening the rings generated by Fourier, lowering the number and brightness of the rings (figure 26 right).

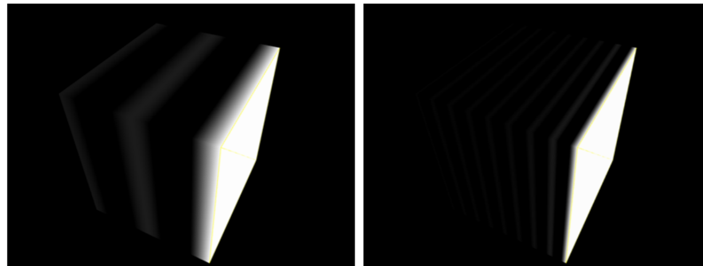


Figure 26: CSM Ringing effect with M=2 (left) and M=8 (right). [3]

2.4.4 Exponential Shadow Maps

ESM [2], similarly to CSM, approximates the depth values of a normal shadow map but using an equation. In ESMs case it uses an exponential approximation seen in equation 4.

$$exp(k * (z - d)) = exp(k * z) * exp(-k * d) \quad (4)$$

We can see an example of ESM approximation in figure 27.

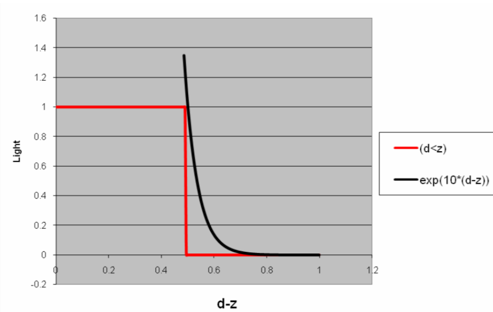


Figure 27: ESM rounding demonstration. [3]

We should tune the value K to achieve the desired shadow results. A smaller value has more of a blur, but the shadow might get less dark in result. On the contrary, by using a higher k, the shadow gets darker, but it has less of a blur.

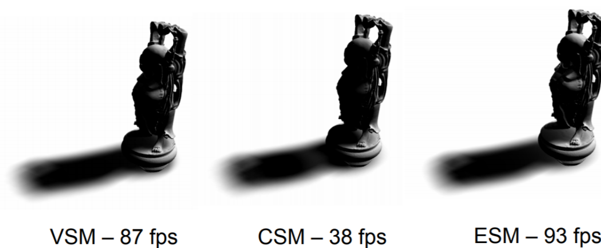
We can counter it's minor artifacts by over darkening the resulting shadow.

ESM also has a problem with light bleeding that cannot be avoided.

2.4.5 VSM vs. CSM vs. ESM

Generally speaking, the solution which requires less memory tends to be the faster at pre-filtering. Since ESM only stores the scale factor (R32 value), as opposed to VSM storing Minimum Variance and the Bleeding Reduction Factor (R32G32 value) and CSM storing multiple textures and the Absorption Factor (N * R8G8B8A8 value), ESM is the best performing solution.

In figure 28 [3], we can see the shadows produced by each of the algorithms and their corresponding performance.



VSM – 87 fps CSM – 38 fps ESM – 93 fps

Figure 28: VSM vs. CSM vs. ESM.

CSM produces the best looking shadow, but it can't achieve a good enough performance so that it should be tested in a mobile environment.

Both VSM and ESM also produce a good quality shadow and at a much better performance, thus these solutions will be the ones developed to achieve a soft shadow map.

2.5 Shadow Mapping for Soft Shadows with variable penumbra

2.5.1 Percentage Closer Soft Shadows

PCSS [9] is the current state of the art for drawing shadows, present in the NVIDIA ShadowWorks library used by computer games like Far Cry 4 and Assassin's Creed IV: Black Flag, as can be seen on their website [18].

PCSS adaptively blurs the shadow map according to the distance between the light and the occluder and the distance between the occluder and the desired point to shadow in the shadow receiver, as demonstrated in figure 29 [9].

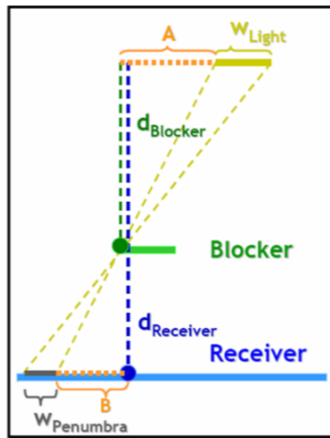


Figure 29: Inner workings of PCSS.

First step is the blocker search. Here we check for the occluders in a grid space of the shadow map and average their depth.

Then we estimate the width of the penumbra by using the rule of three between the width's of the light and the desired shadow penumbra and the distance from the light to the occluder and the occluder to the shadow receiver:

$$w_{penumbra} = \frac{p_s^z - z_{avg}}{z_{avg}} w_{light} \tag{5}$$

We can finally proceed with a normal shadow map filtering, using any of the soft shadow approaches that were discussed before (PCF, VSM, CSM or ESM) and tweak their blur to to achieve a harder or softer shadow according to the penumbra size calculated.

To support fast blurs of variable size we can use other pre-filtering methods to the shadow maps.

2.5.2 Summed Area Tables

SAT is a table generated from the shadow map in which each pixel (i,j) of the SAT correspond to the sum of all the pixels above and to the left of the pixel (i,j) of the shadow map, including it.

With this table we can get the sum of a given rectangular region as follows :

$$s = t[x_{max}, y_{max}] - t[x_{max}, y_{min}] - t[x_{min}, y_{max}] + t[x_{min}, y_{min}] \tag{6}$$

We can use this for PCSS because we can adaptively change the size of the area that we want to get using the formula above to get the sum of the area and dividing it for the number of pixels in that area which would yield its average.

The disadvantage of this solution is that it can only do box filtering.

CSM is not feasible with SAT's since it would have to generate multiple tables which could take a while to generate and worsen performance.

2.5.3 Mipmapping

We can use Mipmapping with CSM instead of using SAT's, a MIPCSM.

A mipmap stores an image multiple times with different increasing sizes. We can build one on top of a CSM, using a large box filter to generate each level of detail and doubling the resolution for the next level. We pick the LOD according to the size of the penumbra calculated from PCSS.

We can access the CSM and build the mipmap using trilinear filtering.
We should also do a pre-pass blur to reduce some issues to discretized blurs.

Mipmapping with CSM uses less memory and its overall faster to build than using a SAT.

2.6 Mobile environment

As Andrew Gruber[13] states, although having some similarities, mobile devices have different needs and applications than desktop or even laptop computers, as such we need to take into consideration those needs and adapt our algorithm to the device.

Smartphones are smaller and lighter than normal computers and have an expected long battery duration, this leads to a necessity for smaller and weaker hardware that wont overheat as much nor consume as much energy.

As we want to have a good battery duration we also need to reduce the power usage of the device while running our application.

2.6.1 Shared memory

Current desktop GPUs have dedicated GDDR memory with a high capacity and bandwidth. Mobile GPUs on the other hand, share LPDDR memory with other IPs of the device, are given less priority and have a considerably lower bandwidth. This lowers the performance of the device compared to a desktop computer, but by using shared LPDDR memory instead of using dedicated memory, the device is much more power efficient.

Another pro of using shared memory is that we can directly map memory which can lead to a beneficial split of work between the GPU and the CPU, together with buffering.

2.6.2 Dynamic clock and voltage scaling

The devices also have an aggressive dynamic clock and voltage scaling to correspond to the current usage, this way the efficiency is increased by lowering/raising the performance to the exact demand of the device, only using the necessary power at a given moment.

2.6.3 CPU big.LITTLE Architecture

Another way that a mobile device differs from a normal computer is the CPU architecture. Mobile phones nowadays employ a big.LITTLE architecture were instead of using multiple similar cores, the colors of the processor differ to better adapt to the different workloads of the phone.

If the phone is not using a lot of processing power then it uses the weaker but more efficient cores. But if there is a sudden need for more processing power than the more powerful cores kick in and help out in the workload.

2.6.4 Tile-based Rendering

In a mobile device we also use tile based rendering. This consists in dividing up the frame into multiple tiles and resolve each one at a time to the final frame buffer.

Each pass on a tile stores the resulting values on GMEM, which is a fast local memory present in nowadays mobile GPUs. The bigger the GMEM size, the fewer tiles we need for our scene, since we can store more information. Since GMEM is physically closer to the GPU and has higher bandwidth, this makes the rendering process both more efficient and perform better.

When a tile is finished rendering we get the values stored on GMEM and send it to the frame buffer in the system memory and start processing the next tile.

2.6.5 Other mobile technologies

Because the market for mobile gaming is increasing, the support for various technologies is increasing, i.e. technologies like Vulkan and Direct-X 12 are already widely supported and with an increase in features being ported to the mobile environment, with a higher priority for features that are power saving. Mobile GPU's also support OpenGL computing.

3. Implementation

After defining our environment and the solutions which will be implemented, we can now develop our application for a mobile environment. Here we look at which technologies we used for the development of the application.

We also discussed the implementation of each of the solutions used in the application, as well as some adaptations done to those solutions.

3.1 Development Environment

This work is developed in partnership with Samsung, which provided a Samsung Galaxy Note 9, thus the implementation is focused to run on Android operating systems.

Since the application will be developed for android, Android SDK was chosen for it's development, and, although generally coded in Java, since Android runs on a JVM, the App will be developed mostly in C/C++ through the JNI.

The base architecture of the application is shown in figure 30.

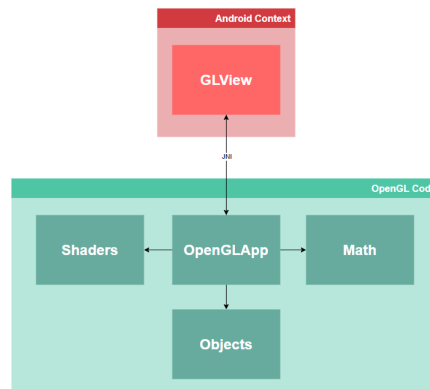


Figure 30: Representation of the developed app.

The OpenGLApp is where the actual application is developed. Everything related to the actual rendering engine of the scene, together with the graphics settings such as the camera settings, light settings and others, is maintained here.

The shaders module contains the shaders used in our application, the vertex shader and the fragment shader.

The math module contains the libraries used, in our case it only has the tinyobjloader library.

The Objects module contain the .obj files that where used in the project.

As discussed previously, the focus will be on an implementation using Shadow Mapping. Considering the related work, the shadow mapping implementations will be focused on PCF, VSM and ESM to achieve a soft shadow, together with PCSS to enable variable sized penumbrae.

Considering all of the prepositions defined for our application, an already existing project was chosen to work upon and implement our shadow rendering improvements. The project used was OpenGL ES

SDK for Android from ARM [19], since it already had a simple Perspective Shadow Mapping example, which it was used to implement PCF, VSM, ESM and PCSS.

To note that any shading calculations besides the calculation of the shadow were removed from the project. This was done to be able to visualize the shadows better.

The tinyObjLoader library [24] was also used to import .obj files to the application, since it allowed to more easily add objects to the app and use different scenes which would allow to compare the performance across scenes with different levels of detail and complexity.

3.2 Shadow Mapping Implementation

3.2.1 Shadow Mapping

The basic version of Shadow Mapping was already implemented in the base project, which was latter adapted to fit the needs of the improved algorithms.

To create and use a shadow map, the following steps were done.

Creation and configuration of a buffer to hold Shadow Map values

```
void createShadowMapTexture() {
        /* Generate and configure shadow map texture to hold depth values. */
        glGenTextures(1, &shadowMap.textureName);
        glBindTexture(GL_TEXTURE_2D, shadowMap.textureName);

        glTexStorage2D(
            GL_TEXTURE_2D,
            1,
            GL_DEPTH_COMPONENT24,
            shadowMap.width,
            shadowMap.height
        );

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_NONE);

        /* Attach texture to depth attachment point of a framebuffer object. */
        glGenFramebuffers(1, &shadowMap.framebufferObjectName);
        glBindFramebuffer(GL_FRAMEBUFFER, shadowMap.framebufferObjectName);

        glFramebufferTexture2D(
            GL_FRAMEBUFFER,
            GL_DEPTH_ATTACHMENT,
            GL_TEXTURE_2D,
            shadowMap.textureName,
            0
        );
};
```

To use a Shadow Map inside the vertex and fragment shaders, we first have to create a buffer in which we will write the shadow map values so that these values can be passed down to them. To do this, a texture is generated (line 1) and bound to change it's configuration (line 2).

The next step is to prepare the texture to store the depth values from the rendering pass into it (lines 6 to 12). This defined that the only value needed is the depth value, by using GL_DEPTH_COMPONENT24, and it also defines the chosen size for the shadow map.

Next, the parameters of the texture are defined, as seen in the code from line 14 to 18. Initially GL_TEXTURE_COMPARE_MODE was defined to GL_COMPARE_REF_TO_TEXTURE, so the access to the shadow map would return either a 0 or a 1, depending on the attribute position being behind of the point in the shadow map or not. This value was changed to GL_NONE, since the value needed was the actual depth of the point.

The minifaction and magnification filters are defined to nearest in the simple shadow map, and the texture is clamped to the edge.

Finally, the Frame Buffer Object is generated and bound, lines 21 and 22 respectively, and the texture is bound to it (lines 24 to 30) Since we are storing the depth values, the attachment is defined as GL_DEPTH_ATTACHMENT.

Creation of the Shadow Map

```
void createShadowMap()
{
    /* Bind framebuffer object.*/
    glBindFramebuffer(GL_FRAMEBUFFER, shadowMap.framebufferObjectName);

    /* Set the view port to size of shadow map texture. */
    glViewport(0, 0, shadowMap.width, shadowMap.height);

    /* Enable depth test to do comparison of depth values. */
    glEnable(GL_DEPTH_TEST);
    /* Disable writing of each frame buffer color component. */
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

    /* Update the lookAt matrix that we use for view matrix (to look at scene from the
    light's point of view). */
    calculateLookAtMatrix();

    draw(false);
}
```

First step to create the actual shadow map is to bind to the framebuffer that was created (line 4), and then the viewport is adjusted to match the texture map size (line 7).

The depth test is enabled, since we need it to get the depth values, and the color mask is set to disable the color writing in the FBO (line 12), since we don't need to store the colors. The scene is then drawn from the light's point of view.

In the vertex shader, on the pass to render the shadow map, the *modelViewProjectionMatrix* is calculated using the light position instead of the camera position.
The fragment shader is not used, since the color mask was set to not write any color.

At the end of this rendering pass the shadow map is generated and stored in the FBO.

Use the Shadow Map in the normal rendering pass

After the shadow map generation, the next step is to pass it to the shader and do a normal rendering pass.
In the vertex shader, the matrix to transform a point from the camera space to the light space needs to be calculated as follows:

```
outputViewToTextureMatrix = biasMatrix * lightProjectionMatrix
                          * lightViewMatrix * inverse(cameraViewMatrix);
```

where the *biasMatrix* is the matrix to pass the values from eye space coordinates [-1,1] to texture coordinates [0,1].

The actual shadow calculation is done in the fragment shader.
For this calculation we need the position of the light and the output positions, which are passed from the vertex shader as *outputLightPosition* and *outputPosition* respectively, the *outputViewToTextureMatrix* calculated in the vertex shader, and the shadow map which is a sampler2D uniform passed as *shadowMap*.

The first step is to get depth value from the shadow map:

```
float[2] getDepths(vec4 position) {
    /* Position of the vertex translated to texture space. */
    vec4 vertexPositionInTexture = outputViewToTextureMatrix * position;
    /* Normalized position of the vertex translated to texture space. */
    vec4 normalizedVertexPositionInTexture = vec4(vertexPositionInTexture.x /
    vertexPositionInTexture.w,
    vertexPositionInTexture.y / vertexPositionInTexture.w,
    vertexPositionInTexture.z / vertexPositionInTexture.w,
    1.0);

    float[2] depths;
    /* Depth value retrieved from the shadow map. */
    depths[0] = texture(shadowMap, normalizedVertexPositionInTexture.xy).r;
    /* Depth value retrieved from drawn model. */
    depths[1] = normalizedVertexPositionInTexture.z; /* shadowDepth */

    return depths;
}
```

To get the position of the shadow map needed, the position needs to be transformed to the light space (line 3) and then normalized (line 5 to 7).
This value can then be used to get the desired depth from the shadow map (line 12).

The depth value on the shadow map (*shadowMapDepth*) and the depth of the position (*modelDepth*) are then returned and the shadow calculation proceeds:

```
vec4 shadowcalc(vec4 oldColor, SpotLight spotLight, vec4 position) {
    float[2] depths = getDepths(position);
    float shadowMapDepth = depths[0];
    float modelDepth = depths[1];

    const float shadowMapBias = 0.0002;

    if (modelDepth - shadowMapBias > shadowMapDepth)
    {
        vec4 newColor = oldColor;

        /* Calculate colour for spot lighting.
         * Scale the colour by 0.5 to make the shadows more obvious. */
        newColor = newColor * 0.5;

        return newColor;
    }

    return oldColor;
}
```

To check if the position is in shadow or not, the *shadowMapDepth* and the *modelDepth* are compared (line 8). If the *modelDepth* is bigger than *shadowMapDepth* then the point is in shadow and the color is multiplied by 0.5 to darken it (lines 10 and 12). Otherwise the point is not in shadow and the color is not changed.

The *shadowMapBias* is introduced (line 6) to mitigate the self-shadowing issue, and reduces the *modelDepth* value (inside the if in line 8) so a point that would be considered as being in shadow, by a small margin, changes to being lit.

After this calculation, the scene is rendered with hard shadows:

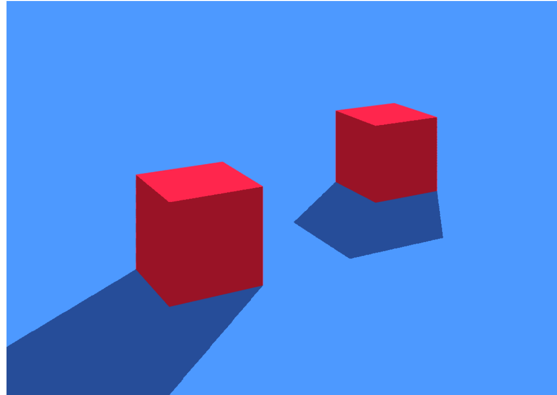


Figure 31: Render with simple shadow mapping

3.2.2 Percentage-Closer Filtering

The PCF also uses the shadow map generated by a basic Shadow Map solution, the difference is present in the fragment shader, where the shadow map is accessed multiple times and the average is calculated:

```
vec4 pcf(int tap, vec4 oldColor, SpotLight spotLight, vec4 position) {
    float count = 0.0f;
    float shadowTotal = 0.0;

    float step = spotLight.area / float(tap + 1);

    for (int i=-(tap/2); i<=tap/2; i++){
        for (int j=-(tap/2); j<=tap/2; j++){
            shadowTotal = shadowValCalc(spotLight, pos);
            count++;
        }
    }

    return oldColor * shadowTotal/count;
}
```

In this solution, the *tap* represents the number of accesses that will be done to the shadow map for the average calculation, the *spotLight* provides the desired area of the light to control the size of the desired penumbra, and the position represents the position of the point to be shadowed.

Inside the for loops, the i and j values represent the shift to the position in the shadow map. This will access the values inside a square in the shadow map (line 9), with the position at the middle of the square, starting to access it from the bottom left corner (when both i and j are -tap/2) and moving to the left and then up (until i and j are both tap/2). After each access, the value returned is added to the total value and the count is increased (line 10).

The *shadowValCalc* (used in line 9) was also created, which is an adaptation of *shadowCalc* functions, which returns either 0.5 or 1.0, depending if it is in shadow or not, instead of returning the resulting color.

After the loops are completed, the average is calculated by simply dividing the total values by the count (line 14), which is then multiplied by the color and thus resulting in a soft shadow.

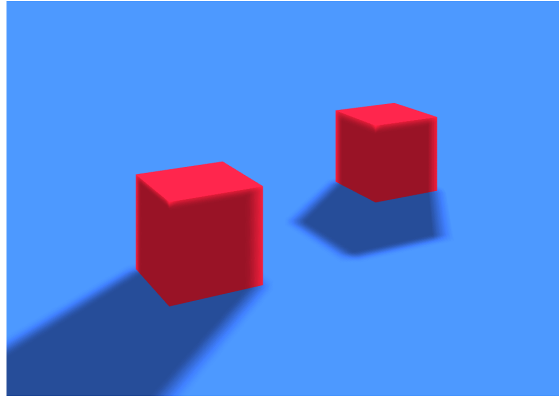


Figure 32: Render with PCF

3.2.3 Variance Shadow Maps

Since VSM stores two values instead of just one value, the shadow map texture needs to be adapted.

Adaptations to texture creation

```
glGenTextures(1, &shadowMap.colorTextureName);
glBindTexture(GL_TEXTURE_2D, shadowMap.colorTextureName);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_NONE);

glTexImage2D( GL_TEXTURE_2D, 0, GL_RG32F, shadowMap.width, shadowMap.height, 0, GL_RG,
    GL_FLOAT, 0);

glGenRenderbuffers(1, &shadowMap.textureName);
glBindRenderbuffer(GL_RENDERBUFFER, shadowMap.textureName);

glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32F, shadowMap.width, shadowMap.
    height);

glGenFramebuffers(1, &shadowMap.framebufferObjectName);
glBindFramebuffer(GL_FRAMEBUFFER, shadowMap.framebufferObjectName);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, shadowMap.
    colorTextureName, 0);
```

As before, we create and bind the texture that will old the shadow map (lines 1 and 2). The first difference is in the filtering, which is changed to GL_LINEAR (lines 4 and 5) so the shadow map is less aliased.

The next difference is in the stored value, instead of storing the depth value, the values stored will be the red and green values (one component for each moment for variance calculation). So instead of using a glTexStorage2D with GL_DEPTH_COMPONENT24, glTexImage2D is used with GL_RG32F, as seen in line 12.

A render buffer is also generated and bound so the shadow map is created correctly (lines 14 to 17).

Finally, both the texture and the render buffer are bound (line 22) to a generated frame buffer (lines 19 and 20), with GL_COLOR_ATTACHMENT0 and

GL_DEPTH_ATTACHMENT respectively.

Shadow Map rendering adaptations

Before the calculation of the shadow map, the color mask is set to write on the red and green values:

```
glColorMask(GL_TRUE, GL_TRUE, GL_FALSE, GL_FALSE);
```

The creation of the shadow map can now proceeds. The vertex shader is mostly unchanged, passing only isToRenderSM value to the fragment shader, in case it is doing the shadow map creation pass.

In the fragment shader the moments are calculated and stored:

```
if(useVSM && isToRenderSM == 1.0) {
    float depth = gl_FragCoord.z ;

    float moment2 = depth * depth;

    float dx = dFdx(depth);
    float dy = dFdy(depth);
    moment2 += 0.25*(dx*dx+dy*dy);

    color = vec4( depth, moment2, 0.0, 0.0 );
}
```

gl_FragCoord.z can be used to get the depth at that point (line 2), thus having the first moment. The second moment is the square of the first moment, calculated in line 4, which is then adjusted using partial derivative to give it a bias per pixel, in lines 6 to 8, and reduce the shadow map problems like light leakage.

Both of the moments are stored in the color value and the shadow map is created after this pass (line 10).

Normal rendering pass adaptations

Finally the normal rendering pass proceeds. In the fragment shader, the access to the shadow map will be done as in PCF, as to do a box filter to the shadow map, which will allow the calculation of the variance to soften the final shadow result:

```
vec2 boxfilter(vec4 position, float distance, int tap) {
    // Position of the vertex translated to texture space.
    vec4 vertexPositionInTexture = outputViewToTextureMatrix * position;
    // Normalized position of the vertex translated to texture space.
    vec2 normalizedVertexPositionInTexture = vec2(vertexPositionInTexture.x /
    vertexPositionInTexture.w,
    vertexPositionInTexture.y / vertexPositionInTexture.w);

    float count  = 0.0;
    float totalX = 0.0;
    float totalY = 0.0;

    float step = distance / float(tap + 1);
    step *= 0.01;

    for (int i=-(tap/2); i<=tap/2; i++){
        for (int j=-(tap/2); j<=tap/2; j++){
            vec2 pos = normalizedVertexPositionInTexture + vec2(float(i)*step, float(j)*
    step );
            vec2 shadow = texture(shadowMap, pos).rg;
            totalX += shadow.x;
            totalY += shadow.y;
            count++;
        }
    }

    return vec2(totalX/count, totalY/count);
}
```

This filter will return the average moments in an area, which will be used by the Chebyshev's upper bound to calculate the likelihood of which the point is in shadow which is translated to how much the point should be shadowed.

```
vec4 VSM(int tap, vec4 oldColor, SpotLight spotLight ,vec4 position)
{
    float[3] depths = getDepths(position);
    float shadowMapDepth = depths[0];
    float modelDepth = depths[1];

    // We retrieve the two moments previously stored (depth and depth*depth)
    vec2 moments = boxfilter(position, 1.0, tap);

    // The fragment is either in shadow or penumbra. We now use chebyshev's upperBound to
     check
    // How likely this pixel is to be lit (p_max)
    float variance = moments.y - (moments.x*moments.x);
    variance = max(variance,0.0000001);

    float d = modelDepth - moments.x;
    float p_max = smoothstep(0.0, 1.0, variance / (variance + d*d));

    return oldColor * (p_max * 0.5 + 0.5);
}
```

The variance is calculated with the moments in line 12. A minimum value of the variance is defined to remove some artifacts that can occur (line 13).

Finally, the distance between the point and the occluder is calculated (line 15) and used with the variance to calculate the percentage to shade the point (line 16). The smoothstep function is used so that lapping shadows experience no light bleeding.

This percentage is multiplied and added by 0.5 as to set it as a minimum shadow value of 0.5 (line 18).

When the pass finishes, a frame with a VSM shadow is generated.

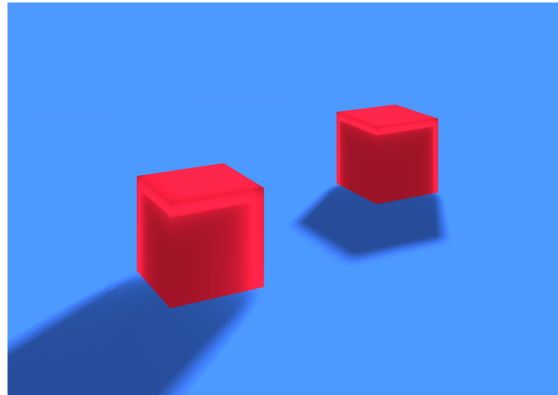


Figure 33: Render with VSM

3.2.4 Exponential Shadow Maps

The ESM implementation can be easily implemented with what has already been done, the filter used is the same as the one used by VSM, and since ESM uses a depth value from the shadow map, it only needs the x value returned from the access to the shadow map.

Thus, in the fragment shader, the shadow value is calculated as follows:

```
vec4 esm(int tap, vec4 oldColor, SpotLight spotLight, vec4 position) {
    float[3] depths = getDepths(position);
    float shadowMapDepth = depths[0];
    float modelDepth = depths[1];

    if (depths[2] < 0.0) return oldColor;

    float c = 100.0;

    float smDepth = boxfilter(position, spotLight.area * 50.0, tap).x;

    if (smDepth == -1.0) return oldColor;

    if ((modelDepth - smDepth) < 0.000015) return oldColor;

    float shadow = clamp( exp( -c * (modelDepth - smDepth)), 0.5, 1.0 );

    return oldColor * shadow;
}
```

The c value (defined in line 8) changes the penumbra size of the shadow, calculated in line 16. The exponential is then clamped between 0.5 and 1 (totally in shadow and totally in light respectively), for when the exponential falls over those values.

The bias here is checked before the calculation of the shadow, in line 14, by not shadowing a point if the difference between the depth of the point in the light perspective and the average depth values in the shadow map is smaller than the bias, which in this case is 0.000015.

3.2.5 Percentage Closer Soft Shadows

As previously established, PCSS is composed by three steps.

Blocker search

The first step is the blocker search, were, similarly to PCF, the values of an area in the shadow map will be accessed and averaged:

```
float pcssBlockerSearch(int tap, SpotLight spotLight, vec4 position) {
    float step = spotLight.area / float(tap + 1);
    float count = 0.0;
    float totalBlocker = 0.0;
    float modelDepth = getDepths(position)[1];

    for (int i=-(tap/2); i<=tap/2; i++){
        for (int j=-(tap/2); j<=tap/2; j++){
            float shadowMapDepth = getDepths(position + vec4(float(i)*step,0,float(j)*
    step,0))[0];

            if(modelDepth > shadowMapDepth) {
                totalBlocker += shadowMapDepth;
                count += 1.0;
            }
        }
    }

    if (count <= 0.0f) {
        return -1.0f;
    }

    return totalBlocker/count;
}
```

The difference is that the values averaged are the values returned from the shadow map and only the values where the point is in shadow are needed for the average, so if the point accessed is not in shadow, it is discarded (lines 11 to 14). The average is then calculated and returned, except if there were no points in shadow, in which -1.0 is returned and the shadow calculation is bypassed (lines 18 to 20), since

the point is fully lit.

Penumbra width calculation

After getting the average depth, the width of the penumbra needs to be calculated:

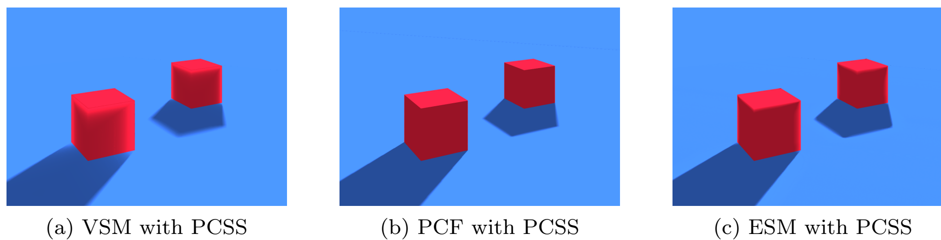
```
float pcss(int tap, SpotLight spotLight, vec4 position) {
    float blocker = pcssBlockerSearch(tap, spotLight, position);

    if (blocker == -1.0f) {
        return -1.0f;
    }

    float[3] depths = getDepths(position);
    float modelDepth = depths[1];
    return (modelDepth - blocker) / blocker;
}
```

In this step, the distance between the position and the block is needed as well as the average depth returned in the blocker (*modelDepth* and *blocker* respectively). With these values, the ratio between the distance from the point to the blocker and the blocker to the light is returned.

The returned value is later used to get the desired penumbra size:

```
float shadowBlurWidth = spotLight.area * pcss * 50.0;
```

Here, the value 50.0 is the adjustment made to the value returned from the **??**.



(a) VSM with PCSS (b) PCF with PCSS (c) ESM with PCSS

Figure 34: Renders using PCSS

In each of the soft shadows solution, *shadowBlurWidth* was used in different ways to change the penumbra size.

In PCF, the size of the penumbra is set by changing the size of the area accessed in the shadow map. In VSM, the solution is similar, the area of the filter blur is changed to control the size of the penumbra.

Finally, because of the way it calculates the shadow, ESM and PCSS do not produce a very accurate shadow, since when the shadow is closer there is a good amount of light leaking with ESM which enters in conflict with PCSS creating a hard shadow. This makes adapting the shadow results to a scene harder, specially in complex scenes with many shadows.

4. Evaluation and Results

After the development of the application, each of our solutions need to be evaluated so that we can reach a conclusion about which are the most suitable ones under which circumstances. For this, multiple data points need to be gathered and compared, so a conclusion can be drawn.

4.1 Evaluation Methodology

To be able to gather multiple metrics to measure the performance of each solution in multiple scenes, the Snapdragon Profiler, developed by Qualcomm [21], was used. This program can show multiple data points available from a smartphone which uses a snapdragon processor.

The particular model of the Samsung Galaxy Note 9, provided by Samsung, has a Qualcomm Snapdragon 845 SoC and a Qualcomm Adreno 630 GPU.

4.1.1 Tools and metrics

Framerate

Framerate is the measurement defined as the number of frames that are presented in a second, designated as Frames per Second. It is one of, if not the most widely used measurement to determine the performance of a graphical application.

The framerate is an important measure due to multiple reasons, for once, the more the FPS an application has, the smoother is the experience for the user, with an acceptable mark for real-time rendering being 30 FPS (an average lower than that can be described as too clunky), but generally needing an average of 60 to 90 FPS too actually feel smooth, due to the flicker fusion threshold of the human body [11].

It is also important, especially in the competitive video games area, for the amount of information that it is presented to the user, for example, in a competitive online shooter, being presented with more frames in a second equates to the on-screen information being available quicker, while also allowing the player to make a more correct and quicker assertion on the placement of his crosshair, which gives the player an advantage.

The application measures the average FPS by using a timer and by counting each frame. In the rendering loop, the timer is checked to see if one minute has passed, if it didn't the frame count is incremented, otherwise the ratio between the frame counter and the timer is logged and the timer is restarted. The value logged represents the average FPS in each minute.

Snapdragon Profiler

The Snapdragon Profiler is installed in the computer, which is then connected to the smartphone via USB, giving access to multiple measurements of the device. This profiling tool also allows for data to be recorded and exported as a CSV file, which can be processed and used to better visualize the data recorded.

From all the metrics available, the following were chosen to be recorded and compared:

- CPU
  - CPU Utilization %: percentage of CPU time the process is active

- Memory
  - Memory Usage: Memory (RAM) used by the process in bytes
- GPU General
  - GPU % Bus Busy: Approximate percentage of time the GPU's bus to system memory is busy
  - Clocks / Second: Number of GPU clocks per second
- GPU Memory Stats
  - Read Total (Bytes/sec): Total number of bytes read by the GPU from memory, per second.
  - Texture Memory Read (Bytes/sec): Bytes of texture data from memory per second.
  - Write Total (Bytes/sec): Total number of bytes written by the GPU to memory, per second.
- GPU Shader Processing
  - % Shaders Busy: Percentage of time that all Shaders are busy.
  - % Shader ALU Capacity Utilized: Percentage of maximum shader capacity utilized.
  - % Time ALUs Working: Percentage of time the ALUs are working while the Shaders are busy.
  - % Time Shading Fragments: Amount of time shading fragments compared to the total time spent shading everything.
  - ALU / Fragment: Average number of scalar fragment shader ALU instructions issued per shaded fragment.
  - EFU / Fragment: Average number of scalar fragment shader EFU instructions issued per shaded fragment.
- GPU Stalls
  - % Stalled on System Memory: Percentage of cycles the L2 cache is stalled waiting for data from system memory.
  - % Texture Fetch Stall: Percentage of clock cycles where the shader processors cannot make any more requests for texture data.
  - % Vertex Fetch Stall: Percentage of clock cycles where the GPU cannot make any more requests for vertex data.
- Thermal
  - Temperature: Die temperature in degrees Celsius.

The metrics were chosen to verify the different system demands of the app, as well as to compare and analyze the efficiency of the application in regards to memory sharing and utilization.
The metrics can be compared between the different scenes and the different solutions while using different options for each one of them.

These metrics were obtained in the context of the process, meaning that the measurements of data are recording only the data in the application context.
The Snapdragon profiler, besides providing these measurements, also provides a snapshot tool, to take screenshots of the smartphone screen.

Android Studio
Besides for developing the application, Android Studio was also used to check the Logcat, where the average FPS for each minute was printed.
Android Studio was also used to take screenshots of the smartphone.

4.1.2 Test Scenes and Testing Methodology

To better compare the different solutions, a set of 3 scenes, each with it's own .obj file, were used for testing.
These obj filters were downloaded from the McGuire Computer Graphics Archive [16].

The first scene uses the sponza.obj [6]. The general settings used on this scene were the following:

- Camera position: 0.0, 10.0, 45.0 (x, y, z);
- Camera rotation: 1.3 radians;
- Light position: -10.0, 20.0, 40.0 (x, y, z);
- Light look at: 0.0, 10.0, 0.0 (x, y, z);
- Light FOV: 135.0º;

This scene is not too demanding, with 228462 vertices. It is a good representation of what a scene in a game app might be, thus providing good results to compare with a real life scenario.

Figure 35: sponza.obj illustration

The next next scene uses the dragon.obj [25]. The general settings used on this scene were the following:

- Camera position: 0.0, 10.0, 45.0 (x, y, z);
- Camera rotation: 0.5 radians;
- Light position: 6.0, 7.0, -9.0 (x, y, z);
- Light look at: 0.0, 0.0, 0.0 (x, y, z);
- Light FOV: 90.0º;

This scene has 2613918 vertices, which by it's own makes it very demanding, serving the purpose of evaluating each solution under an already demanding scene.

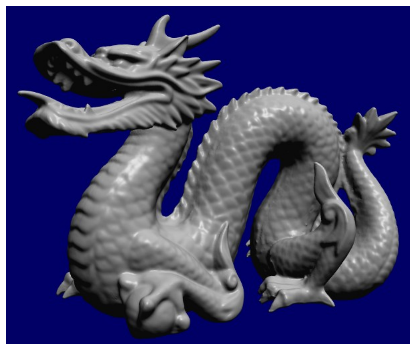


Figure 36: dragon.obj illustration

The final scene uses the bunny.obj [26]. The general settings used on this scene were the following:

- Camera position: 0.0, 10.0, 45.0 (x, y, z);
- Camera rotation: 1.3 radians;
- Light position: -10.0, 20.0, 25.0 (x, y, z);
- Light look at: 0.0, 0.0, 0.0 (x, y, z);
- Light FOV: 90.0º;

With an object with 432138 vertices, this scene is a good compromise between a simple scene with an acceptable complexity, which makes it good to compare with the other scenes.

Figure 37: bunny.obj illustration

The different settings of each of the algorithms, as well as general settings like the size of the shadow map, were also tested with different values.

To notice that these scenes where all tested using one fixed point of view.

4.2 Results

With our evaluation methods defined, we can now record and compare our data points.

Since there are multiple endpoints for tweaking parameters, as well as multiple scenes and solutions, preemptive, although extensive test was done to take some preliminary conclusions as well as to conduct other, more specific tests were conducted to confirm some of the conclusions drawn before.

Some parameters were tweaked and maintained as is for all of the tests performed, like some of the camera and light settings or the colors of the objects, and others were tweaked for each scene but are not referenced since they were presumed to have little to no impact on the performance or assumed to be a part of tweaking the object to fit the scene, like changing the light position, adapting the positioning and rotation of the object or tweaking the bias values.

4.2.1 Preliminary Test

This preliminary test was focused on comparing the framerate performance of each algorithm, using 3x3, 5x5 or 7x7 tap/filtering of the shadow map.

These algorithms were used together with PCSS since the main objective of this study is to conclude if soft shadows in real time rendering in a mobile app is achievable and viable. A 7x7 tap was used to introduce some resource demands and check how well each solution would perform under it.

The size of the shadow map was defined to be 1080 by 2220 (equal to the size of the screen). This value was defined as to be a good compromise between performance and visual quality.

The results of these tests are presented in table 1.

Algorithm	Sponza	Dragon	Bunny
PCF 3x3 tap	47.0	18.0	39.2
PCF 5x5 tap	35.1	15.0	31.3
PCF 7x7 tap	21.8	10.3	21.7
VSM 3x3 filtering	24.2	14.6	26.8
VSM 5x5 filtering	17.8	11.8	20.6
VSM 7x7 filtering	12.8	8.0	15.3
ESM 3x3 filtering	50.3	18.5	42.0
ESM 5x5 filtering	44.2	17.9	38.7
ESM 7x7 filtering	38.4	16.5	34.8

Table 1: Average FPS with PCSS (7x7 tap), 1080*2220 map

There is some valuable insight obtained by comparing these results.

For once, the fact that VSM is the most demanding is confirmed, although it was more demanding than previously though, since it did not reach an average of 30 FPS in any of the tests, and it even fell bellow 10 FPS in the Dragon scene, with a 7x7 box filtering.

This could be due to the fact that VSM uses two color values from the shadow map, which it has to both write and read from (instead of just the depth value, which is also easier to store) as well as potentially lacking some improvements that could enhance it's efficiency.

VSM also was the only solution where it performed worse in Sponza than in Bunny, presumably because it benefited more from avoiding calculations when these were not needed (calculations outside the penumbra), calculations which were less avoidable in the Sponza scene.

Besides bunny running VSM, it was expectable to get the best performance from Sponza, then bunny and with the Dragon scene having the lowest performances.

ESM consistently got better results than PCF, especially the bigger the tap was. ESM also had a lower impact when increasing the tap in the box filtering compared to PCF. This makes sense, although ESM and PCF both have the same amount of accesses to the shadow map, what each of those solutions

do with those values in between is different.

PCF after each tap evaluates if that point is in shadow or not so that the average of the shadow values can be determined, on the other hand ESM only averages the values that it got from the shadow map and later uses that average to calculate the shadow value.

Since for each access to the shadow map, PCF has more instructions than ESM, it is expectable that ESM has a better performance than PCF and a difference which is ever more noticeable the more the amount of accesses.

Finally, it also shows that a variable penumbra shadow in real time rendering is achievable and, with further tweaking it can even be achieved at 60 FPS.

4.2.2 Performance comparison

Guided by the preliminary tests, the next tests were done using the dragon scene using the same settings together with 5x5 filtering/tap of the shadow map, to compare each solution using the data available in the Snapdragon Profiler.

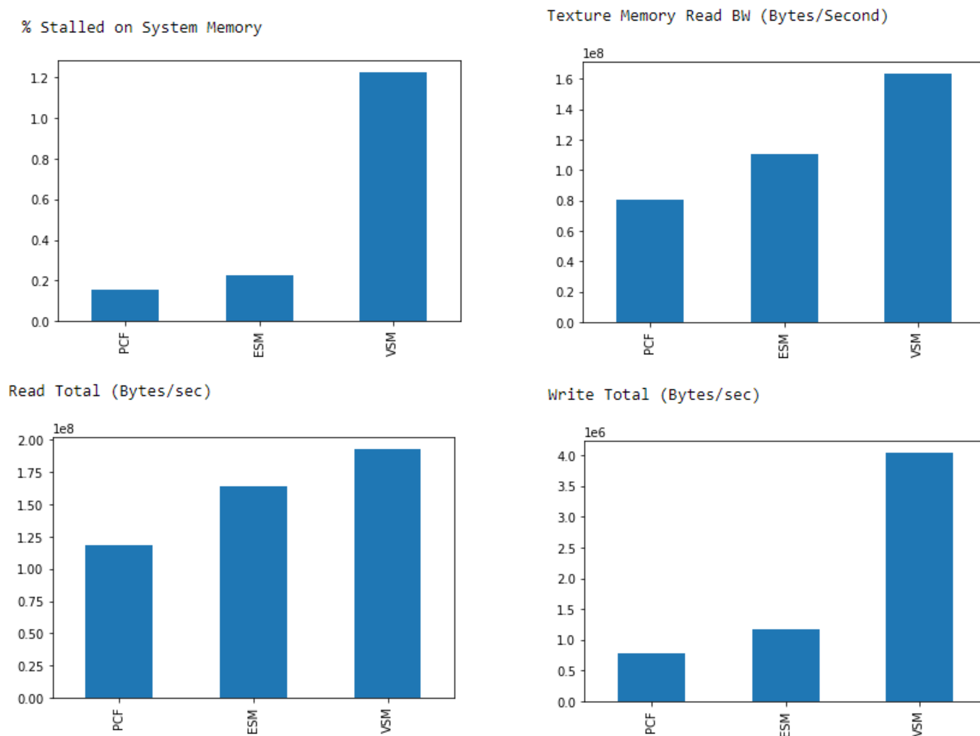


Figure 38: Memory information

As expected, figure 38 shows that VSM has a higher percentage of system memory stalls, as well as higher read total, texture memory read, write total and memory usage. This is due to the fact that VSM has to both read and write more data to the shadow map, using both the red and green channels, instead of a single depth channel. This difference is more noticeable in writing data, since the write total is around four times higher than ESM and PCF, showing that using the depth component does in fact only store one float value, as opposed to storing four values (RGBA) using the normal color attachment. Although blue and alpha values were disabled, they were probably filled with a predefined value.

This also explains why VSM has a higher performance loss against PCF and ESM in a mobile environment versus a desktop computer, since the memory is slower, the bottleneck that memory represents to VSM is bigger.

ESM has a slight more memory usage than PCF. This is expectable, since ESM finishes a pass in the fragment shader quicker than PCF which translates to more frequent memory reads to access the shadow map values. The filtering function used by ESM is also the same used by VSM which will unnecessarily read an extra value from the shadow map, although this does not seem to pose a big performance loss.

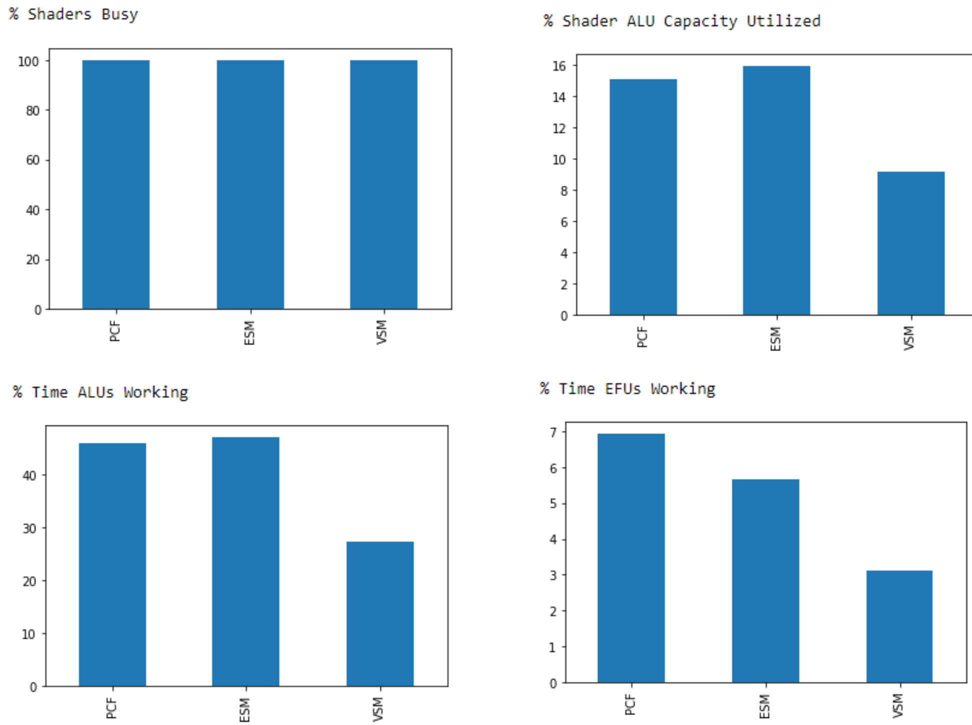


Figure 39: Shader resources usage

Comparing the usage of the shader resources (figure 39), VSM naturally has a lower usage, since it is being stalled on system memory and has more complex instructions between shading calculations it ends up using less of the shaders resources.

ESM and PCF have similar usages, with PCF using a bit more EFUs than ESM, while ESM uses a bit more ALUs. This should be due to the difference in the shadow calculation.

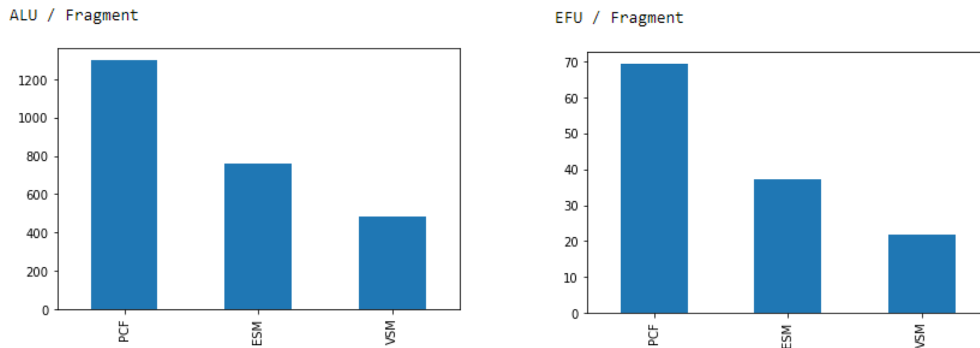


Figure 40: ALUs and EFUs per fragment

By analyzing figure 40, we confirm that PCF does in fact have more ALUs and EFUs intructions for each fragment than ESM, which explains why it has a lower performance.

Comparing this with figure 39, we can see that, although ESM has less instructions, the usage of these units is similar to PCF, since it is a time measure and, during that time the amount of fragments shaded by ESM is higher to that of PCF. Figure 40 also shows that the ALU/Fragment difference between PCF and ESM is lower to that of EFU/Fragment, which explains why the ESM usage of EFUs is lower than PCF, while the ALU usage is higher.

VSM has a lower number of ALU and EFU instructions, presumably it makes use of other units to process it's data, since the variance calculation is more intensive, so a parallel between ALU and EFU instructions and performance cannot be done.

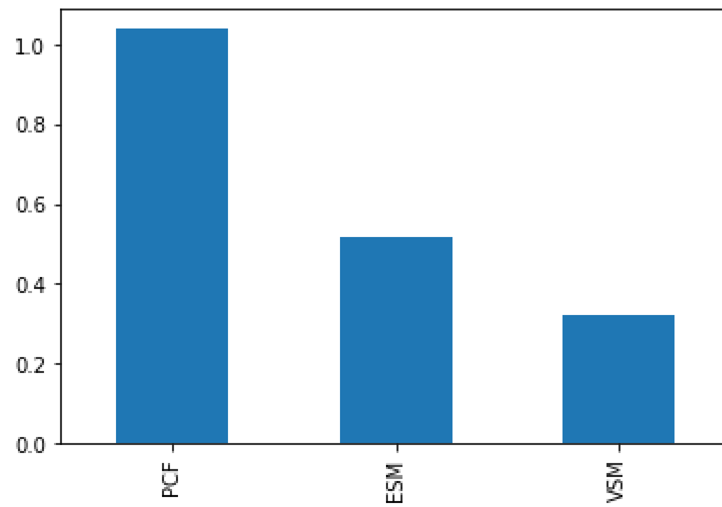


Figure 41: CPU utilization

Checking the CPU utilization (figure 41) confirms that the application is not CPU intensive, but instead is GPU intensive.

The cpu usage is also similar to the ALUs and EFUs per fragment, which can be due to some of these instructions needing CPU.

4.2.3 Testing VSM

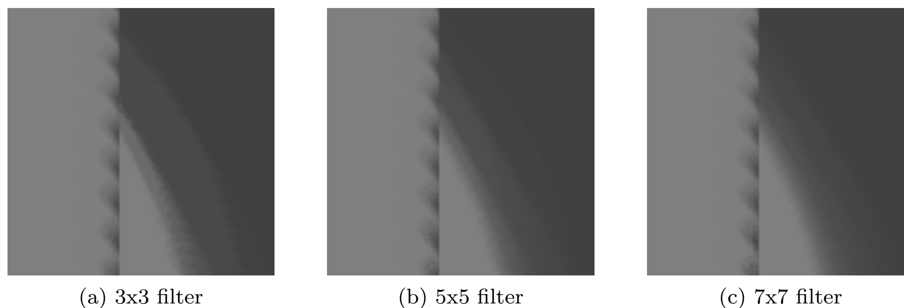
Since VSM can achieve the best visual results, multiple settings were changed to check if a good VSM result can be achieved with an acceptable level of performance. Data was also analyzed to verify the impact of this solution on the hardware resources.

In this test sponza scene was used, for it's good representation of a likely scenario where a solution like VSM would be used, as well as providing a good scene where artifacts can be more noticeable, depending on the light settings.

From figure 42, which is using demanding settings, some shadow artifacts are present. Since some of the artifacts are due to lack of better tweaking of some of the settings, the focus will be on the artifacts present on the actual shadows, and not shadow acne or light leaking.

Figure 42: Sponza render using VSM with 7x7 box filter, 7x7 PCSS tap and 1080*2220 map

By comparing the image results from 7x7, 5x5 and 3x3 box filter, seen in figure 43, the loss in image quality becomes quite noticeable from using a 5x5 filter to a 3x3 filter, with 5x5 filter resulting in a good quality render, although not as good as a 7x7 one.



(a) 3x3 filter (b) 5x5 filter (c) 7x7 filter

Figure 43: Renders using 7x7 PCSS tap and 1080*2220 map

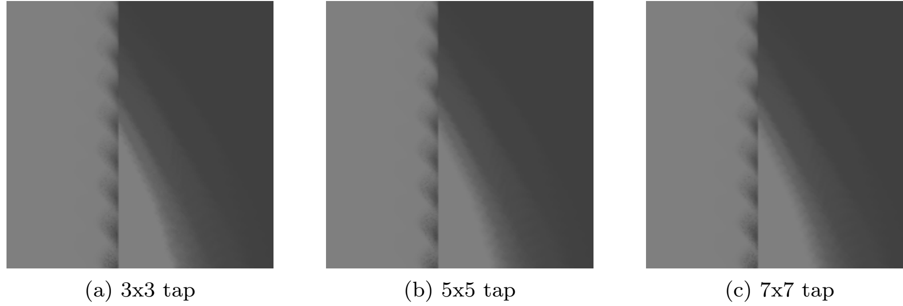
Since 3x3 filtering does not produce a good enough result and 7x7 is too demanding, the next tests will be done using 5x5 filter.

The next step is to compare different taps during the PCSS calculation, namely compare a 3x3, 5x5 and 7x7 tap. The results of these tests are presented in table 1.

PCSS tap	3x3	5x5	7x7
Framerate	29.9	23.6	17.8

Table 2: Average FPS with VSM (5x5 filter), 1080*2220 map

Comparing the results in table 2, diminishing the PCSS tap greatly increases the performance, almost hitting an average of 30 FPS whit a 3x3 tap.



(a) 3x3 tap (b) 5x5 tap (c) 7x7 tap

Figure 44: Renders using 7x7 PCSS tap and 1080*2220 map

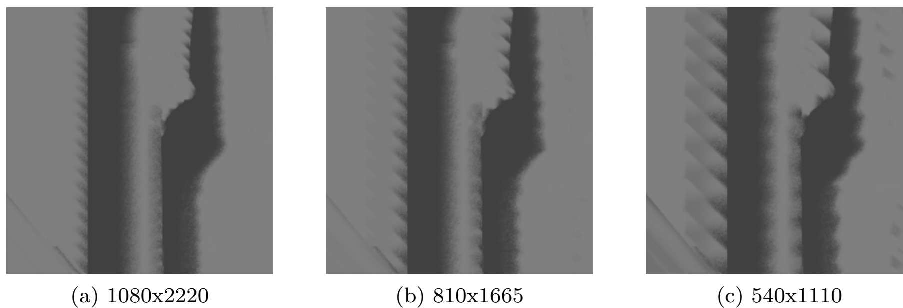
Figure 44 shows that the quality impact of lowering PCSS tap is not very noticeable, turning the ringing artifact slightly more noticeable in the 3x3 tap and introducing some very small noise in the penumbra, while the change is barely noticeable from the 7x7 to the 5x5 tap.
Since the there is a big performance benefit from lowering the PCSS tap, which comes with a small quality loss, it came as an easy decision to continue using the 3x3 tap PCSS for the next steps.

The next change to be done was to the size of the shadow map.

Shadow Map size	1080x2220	810x1665	540x1110
Framerate	29.9	32.5	34.0

Table 3: Average FPS with VSM (5x5 filter) and a 3x3 PCSS tap

Diminishing the size of the shadow map had some small performance improvements, enough to be able to achieve an average higher than 30 FPS.



(a) 1080x2220 (b) 810x1665 (c) 540x1110

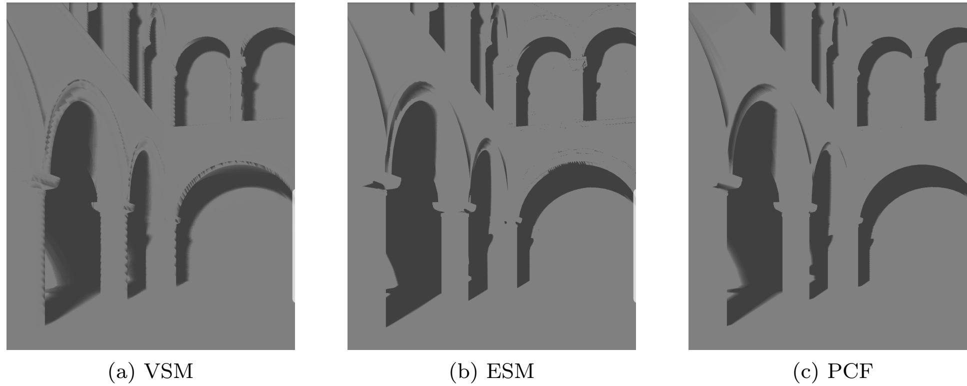
Figure 45: Renders using 7x7 PCSS tap 3x3 PCSS tap

On the other hand, by reducing the shadow map size, the quality of the shadow is moderately impacted, introducing visible and more prominent aliased shadows and other shadow artifacts, making the benefits on performance not good enough to compensate the loss in visual quality, since that with this quality, other solutions may produce a better result with a better performance.

Finally, VSM can be compared with ESM and PCF at a similar performance level to compare it's visual performance with these other solutions.

Algorithm	tap/filtering size	Shadow Map size	PCSS tap	Framerate
VSM	5x5	810x1665	3x3	32.5
PCF	5x5	4320x8880	5x5	36.2
ESM	7x7	4320x8880	5x5	35.5

Table 4: Average FPS with VSM, ESM and PCF with different settings



(a) VSM (b) ESM (c) PCF

Figure 46: Image quality comparison between solutions

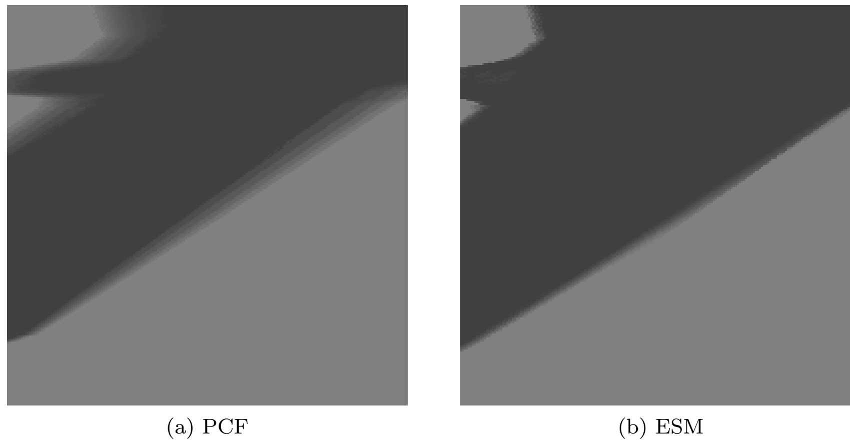
Visually speaking, PCF has a better shadow quality while also achieving a higher performance. ESM on this scene did not produce very consistent soft shadows, since the shadows had an small and inconsistent penumbra, which proves to be a worse candidate than VSM.

This test allowed the conclusion that VSM is not yet suited for a mobile environment, since the devices lack the performance needed for a good quality result and for less demanding settings other solutions become better.

4.2.4 PCF vs ESM

Since VSM was discarded as a viable solution, the next step is to compare PCF and ESM in such a way as to conclude which of these solutions is the best to be used and for which circumstances.

As seen previously in figure 46 and as discussed before, ESM has a hard time producing a good variable soft shadow, especially for complex scenes with many shadows. This means that although ESM has a better performance than PCF, for a complex enough scene, the best solution available for producing a variable soft shadow is PCF.



(a) PCF (b) ESM

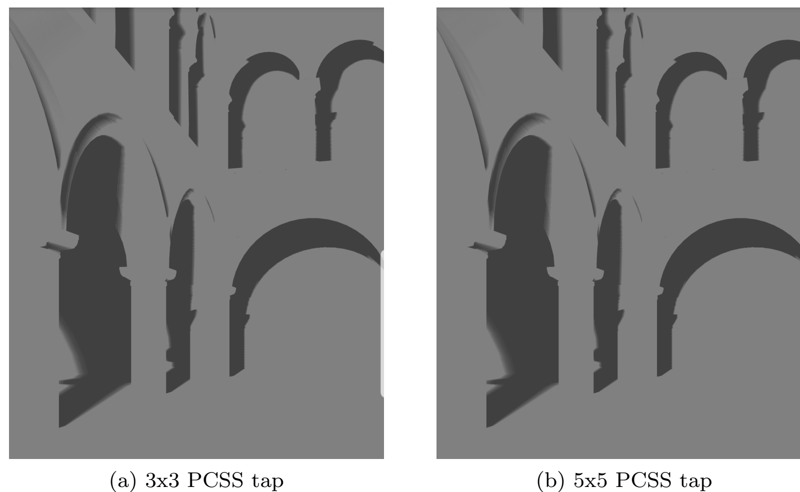
Figure 47: Image comparison between PCF and ESM in Sponza

Taking a closer look to the images previously shown in figure 46, by taking a closer look to the shadows of the first column on the left, shown in figure 47, we can see how ESM has a hard time with the soft shadows, since it begins with a hard shadow that starts becoming soft, but then it regresses and starts becoming an hard shadow again.

Some of the other shadows are produced correctly, but ESM is inconsistent.

Increasing PCF performance

By using PCF with the sponza scene, some of the settings can be tweaked to try and achieve stable framerate of 60 FPS while producing a good quality shadow.



(a) 3x3 PCSS tap (b) 5x5 PCSS tap

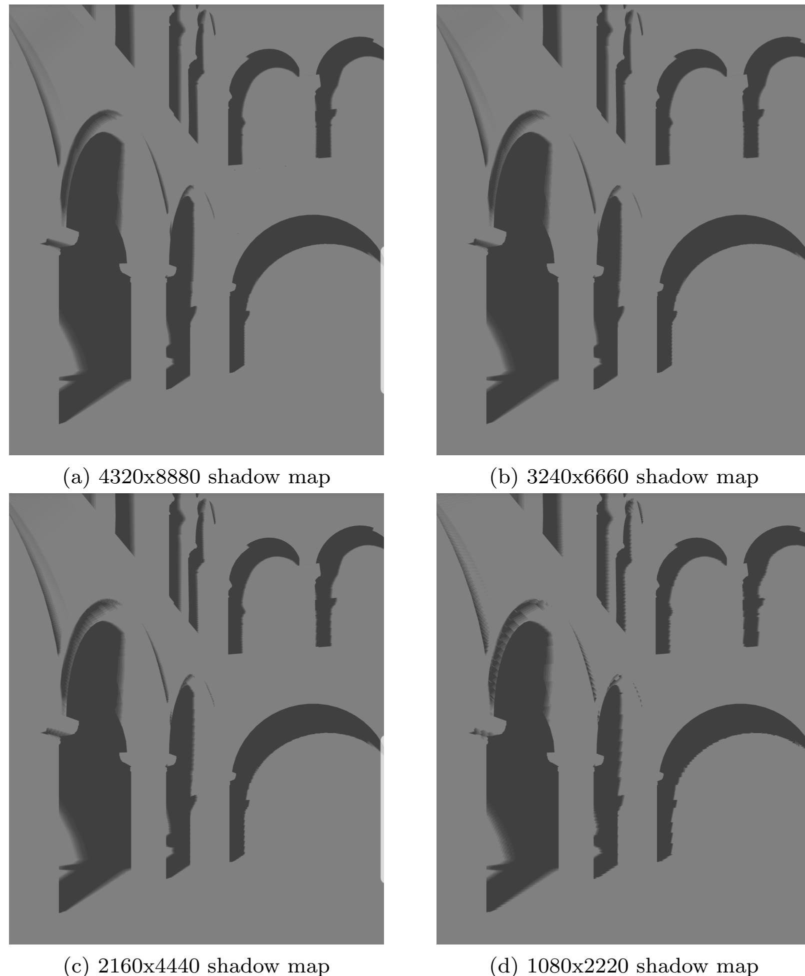
Figure 48: Image quality comparison between PCF with diferent PCSS taps

We started by reducing the PCSS tap from 5x5 to 3x3, which increased the framerate from 36.2 FPS to 47.8 FPS.

From figure 48, changing the PCSS tap did not seem to take a noticeable impact on the quality of the image produced, thus this change was maintained in the following steps.

Shadow Map size	4320x8880	3240x6660	2160x4440	1080x2220
Framerate	47.8	54.6	58.7	59.7

Table 5: Average FPS with different shadow map sizes



(a) 4320x8880 shadow map (b) 3240x6660 shadow map

(c) 2160x4440 shadow map (d) 1080x2220 shadow map

Figure 49: Image quality comparison between PCF with diferent shadow map sizes

By looking at table 6 we can see that a framerate of 60 FPS is achievable by using a shadow map size of 1080x2220, taking into account that the frame rate is capped at 60 FPS due to synchronization with the screen frequency, which explains why it is neither more than 60 FPS.

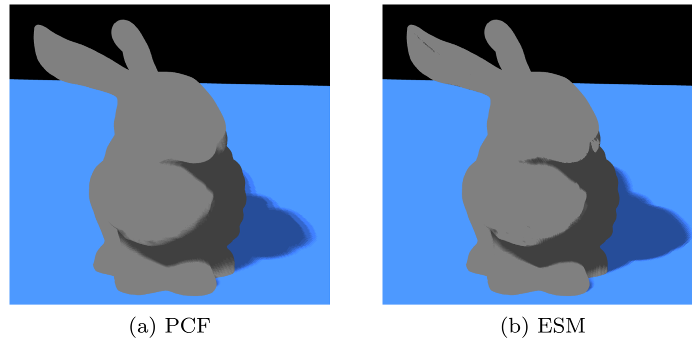
A 1080x2220 shadow map, although performing at a frame rate equal to the device output, saw the image quality reduced, where some aliasing started to become noticeable.

Choosing a shadow map with a size of 2160x4440 pixels or even 3240x6660 pixels might be a better choice, since both perform above 50 FPS, with the first almost hitting 60 FPS, while significantly reducing the aliasing shown on screen. A 3240x6660p shadow map almost gets rid of this problem.

ESM and PCF in a simple scene

If on the other hand, the scene is less complex, ESM might be a better option than PCF. To confirm this, the bunny scene was used to compare both of these solutions.

With the results from the preliminary test it makes sense to start comparing PCF using a 3x3 tap with ESM using a 5x5 filter, since they both perform similarly. The PCSS tap can be lowered, since, as seen before, the performance gained by lowering it far outweighs it's quality loss. After this change, both PCF and ESM had an average framerate of 59.6 FPS, which means both hit the FPS cap.



(a) PCF (b) ESM

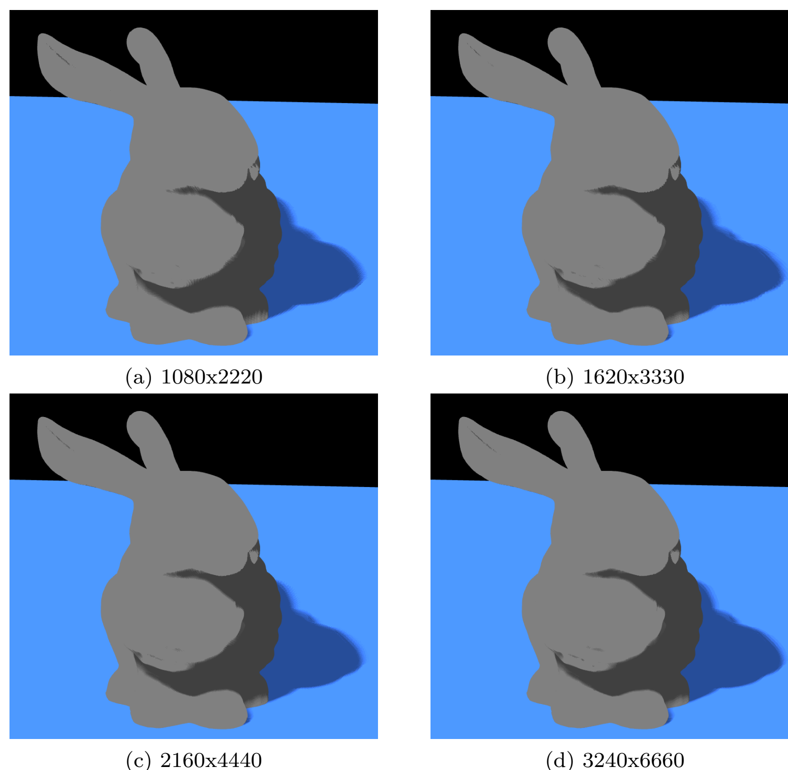
Figure 50: Image quality comparison between PCF and ESM

The resulting images shown in figure 50 tell us that ESM, in fact, performs better in a simpler scene. It correctly produced the bunny shadow on the plane and with less artifacts than PCF, since the latter presented a shadow with noticeable banding, due to the low tap used.

PCF, on the other hand, had less shadow artifacts present in the bunny surface, although the difference here was less noticeable.

With this in mind, ESM proves to be a better fit for simple scenes, since it provides a good resulting variable soft shadow at a good performance, being able to achieve a framerate of 60 FPS.

We can now push ESM settings to increase it's quality until the framerate stops being capped.



(a) 1080x2220 (b) 1620x3330

(c) 2160x4440 (d) 3240x6660

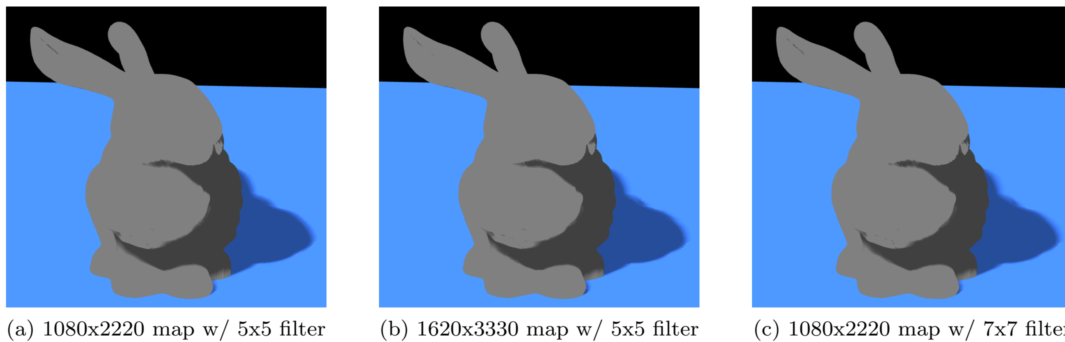
Figure 51: ESM with different shadow map sizes

Shadow Map size	1080x2220	1620x3330	2160x4440	3240x6660
Framerate	59.7	59.7	55.7	50.9

Table 6: Average FPS with different shadow map sizes

As seen on figure 51, by increasing the shadow map size, the shadow artifacts became less noticeable, specially the ones seen before in the bunny surface.
Both 1080x2220p and 1620x3330p shadow map sizes were capped at 60 FPS, the remaining two did not have a high enough framerate to be capped, although both achieved an FPS average higher than 50 FPS.

By still using a 1080x2220p sized shadow map and changing the filter size from 5x5 to 7x7, the average framerate dropped to 56.2 FPS, falling bellow the FPS cap.



(a) 1080x2220 map w/ 5x5 filter (b) 1620x3330 map w/ 5x5 filter (c) 1080x2220 map w/ 7x7 filter

Figure 52: Comparing ESM with different settings

By comparing the resulting images seen in figure 52, we see that increasing the filtering size improved the shadow that was cast on the plane, while increasing the shadow map size diminished the amount of artifacts present on the bunny surface.

Overall there was some space for improvement in ESM, which proved to be capable of providing a good quality shadow at a good performance for simple scenes.

5. Conclusions and Future Work

The objective of this study was to contribute to the knowledge of rendering shadows on a mobile device, using state of the art techniques already present in other platforms, in such a way as to mimic the shadow behaviour as realistic as possible, while achieving an acceptable performance.

There were multiple solutions available and many paths to take, so a choice of developing and further study one solution was the most viable solution. After evaluating various solutions available, shadow mapping seemed to be the best solution to use.

Even by reducing our scope to just one solution for shadow rendering, the number of different techniques available to improve this solution were many. Some techniques for performance improvements were presented, but not implemented. The focus was to use techniques that enabled to improve the quality of the shadow as to achieve variable soft shadows.

5.1 Major Contributions

This study improved our knowledge using state of the art techniques for rendering realistic shadows in real-time on a mobile environment. A mobile app was successfully developed for this purpose, with an already implemented basic shadow mapping, we were able to implement different techniques that improved the shadows produced by rendering soft shadows and even variable soft shadows, the latter being our main focus.

Three solutions to achieve a soft shadow were chosen to be developed and tested, PCF, VSM and ESM, since these were regarded as being the most viable solutions available. CSM was also proposed but ultimately it was regarded as being to demanding to be considered for development.

Along with these solutions, PCSS was implemented so a variable soft shadow could be implemented, since our main focus was to achieve this. We were able to combine this with the implemented soft shadow techniques, although ESM proved to be more difficult to implement it, and providing a final shadow that can have some problems, depending on the complexity of the scene.

From the multiple tests conducted to each of these solutions with different scenes and settings, we were able to take multiple conclusions.

For once, without being further improved, VSM as it was implemented did not achieve a viable solution, since it had bad performance when achieving a good quality shadow, and when settings were changed to increase performance, other solutions could produce a better looking shadow while also performing better.

ESM, due to it's difficulty in producing variable soft shadows, only produced the best solution in a simple scene, where the settings could be fine tuned to render a good quality shadow, that could even be better than one produced by PCF for around the same performance.

Finally, PCF proved to be the most viable solution, providing a good quality shadow, although with some artifacts present, depending on the settings used, while also being able to perform well, being able to produce a soft shadow map at the so much desired 60 FPS, at the cost of some quality.

We can conclude that shadows with a variable penumbra can actually be rendered in real-time in a mobile environment, at least with a powerful enough device.

5.2 Future Work

During the development of the application, some compromises had to be done, either because of time constraints or to avoid making the application and this study to complex.

Thus there are some paths that can further be researched and/or improved. These can be:

- Improvements to the base shadow map can be implemented and studied. Although these were talked about in the related work sector, they were not implemented due to time constraints. These solutions can each be implemented to verify if they can either improve the performance of our solution or improve its visual quality and allow us to change the settings of the implemented solutions to bump up the performance.
- Caching shadow map. Another improvement that can be done is caching our shadow map and determining which pixels can be reused and which need to be re-rendered. This was not implemented due to the sheer complexity of this solution, which could possibly be a study on it's own.
- Improvements to our implementation. Our implementation can also be improved, since the time available was not enough to polish the code, there are some rough edges that can be polished in it. Some of the code was recycled, and parts of the code can possibly be implemented in a more efficient way, possibly improving the performance of our solutions.
- Evaluate suitability of shadow volumes. A choice between implementing shadow mapping and shadow volumes was done, to complement this study, a study on shadow volumes can be researched and it can then be compared with the results available in our study.

5. References

[1] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. "Convolution Shadow Maps." In: *Rendering Techniques* 18 (2007).

[2] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. "Exponential shadow maps." In: *Graphics Interface*. ACM Press. 2008.

[3] Louis Bavoil. "Advanced soft shadow mapping techniques". In: *Presentation at the game developers conference*. Vol. 2008. 2008.

[4] R. Benjamim. "Real-Time Global Illumination on Mobile Devices using Reflective Shadow Maps". Master's thesis. Instituto Superior Técnico, Nov. 2019.

[5] Franklin C Crow. "Shadow algorithms for computer graphics". In: *Acm siggraph computer graphics* 11.2 (1977).

[6] M. Dabrovic. *Sponza*. 2002. URL: `http://hdri.cgtechniques.com/~sponza/files/` (visited on 11/2020).

[7] Elmar Eisemann, Ulf Assarsson, Michael Schwarz, Michal Valient, and Michael Wimmer. "Efficient real-time shadows". In: *ACM SIGGRAPH 2012 Courses*. 2012.

[8] Elmar Eisemann, Ulf Assarsson, Michael Schwarz, Michal Valient, and Michael Wimmer. "Efficient real-time shadows". In: *ACM SIGGRAPH 2013 Courses*. 2013.

[9] Randima Fernando. "Percentage-closer soft shadows". In: *ACM SIGGRAPH 2005 Sketches*. 2005.

[10] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P Greenberg. "Adaptive shadow maps". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001.

[11] *Flicker fusion threshold*. URL: `https://en.wikipedia.org/wiki/Flicker_fusion_threshold` (visited on 08/2020).

[12] *GPU Gems 3*. URL: `https://developer.nvidia.com/gpugems/gpugems3` (visited on 05/2020).

[13] A. Gruber. *Mobile GPU approaches to power efficiency*. Tech. rep. Qualcomm, 2019.

[14] Andrew Lauritzen and Michael McCool. "Layered variance shadow maps." In: *Graphics Interface*. Vol. 8. Citeseer. 2008.

[15] Andrew Lauritzen, Marco Salvi, and Aaron Lefohn. "Sample distribution shadow maps". In: *Symposium on Interactive 3D Graphics and Games*. 2011.

[16] M. McGuire. *Computer Graphics Archive*. July 2017. URL: `https://casual-effects.com/data` (visited on 11/2020).

[17] Kevin Myers. "Variance Shadow Mapping". In: *Retrieved May* 11 (2007).

[18] *NVIDIA ShadowWorks*. URL: `https://developer.nvidia.com/shadowworks` (visited on 05/2020).

[19] *OpenGL ES SDK for Android*. URL: `https://github.com/ARM-software/opengl-es-sdk-for-android` (visited on 07/2020).

[20] *Opengl-tutorial, Tutorial 16 : Shadow mapping*. URL: `http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/` (visited on 01/2021).

[21] *Snapdragon Profiler*. URL: `https://developer.qualcomm.com/software/snapdragon-profiler` (visited on 11/2020).

[22] N. Sousa. "Ray Tracing Acceleration Structures on Mobile Environments". Master's thesis. Instituto Superior Técnico, May 2018.

[23] Marc Stamminger and George Drettakis. "Perspective shadow maps". In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002.

[24] *Tinyobjloader*. URL: https://github.com/tinyobjloader/tinyobjloader (visited on 08/2020).

[25] Stanford University. *Dragon*. 1996. (Visited on 11/2020).

[26] Stanford University. *Stanford Bunny*. 1996. (Visited on 11/2020).

[27] *Variance shadow maps*. 2014. URL: https://dontnormalize.me/2014/01/19/variance-shadow-maps/ (visited on 10/2020).

[28] Lance Williams. "Casting curved shadows on curved surfaces". In: *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. 1978.

[29] Michael Wimmer. "Hard Shadows Aliasing and Remedies". In: *ACM SIGGRAPH 2013 Courses*. 2013.

[30] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. "Light space perspective shadow maps". In: *Rendering Techniques* 2004 (2004).

[31] Andrew Woo. "The shadow depth map revisited". In: *Graphics Gems III (IBM Version)*. Elsevier, 1992.

[32] Fan Zhang, Hanqiu Sun, Leilei Xu, and Kitlun Lee. "Hardware-accelerated parallel-split shadow maps". In: *International Journal of Image and Graphics* 8.02 (2008), pp. 223–241.