Shadow rendering techniques for mobile devices

Henrique Araújo
henriquecmaraujo@tecnico.ulisboa.pt

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.

Multiple solutions of rendering hard shadows, which could later be improved to render soft shadows, were presented. After comparing these solutions we concluded that the most suitable algorithm for shadow rendering in a mobile environment is Shadow Mapping due to its efficiency and adaptability.

With the base algorithm chosen, 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 than 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.

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

CCS Concepts
•Computing methodologies → Rendering; •Human-centered computing → Mobile devices;

INTRODUCTION

Since the development of the first mobile devices, there as 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, achieving a point were it is 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.

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 variable soft 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 variable soft 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?

RELATED WORK

Mobile environment

As Andrew Gruber[7] 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.

Some of these differences may include:

- Graphics Processing Unit (GPU) sharing memory with other device components, as opposed to a desktop computer which has it's own dedicated video memory.

- Dynamic clock and voltage scaling.

- Central Processing Unit (CPU) big.LITTLE Architecture, where the size, power and efficiency of the CPU cores is asymmetrical, usually four more powerful ones and four more efficient.

- Tile-based Rendering, where the screen-space is divided into multiple tiles according to the Graphics Memory (GMEM) size for more efficient rendering.

- Other technologies adapted to the mobile environment, for example, OpenGL ES.

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.

Shadow Mapping

Shadow Mapping[12] 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 represent the depth of the lit points. If any given point has a higher depth than the generated image, that point is in shadow.

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

Shadow Volumes

The shadow Volumes technique[4] 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.

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.

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.

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.

This algorithm is a very close representation of real life light behaviour, being very accurate. The downside is that it is extremely demanding.

Choosing between Shadow Volumes, Shadow Maps and 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.

Achieving 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.

Percentage-Closer Filtering (PCF)

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.

The shadow that PCF renders comes with banding issues. Irregular sampling of the shadow map can be used to trade-off banding problems for some shadow noise.

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.

Variance Shadow Maps (VSM)

VSM [9] 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 (1) to determine the percentage of which the pixel is in shadow.

$$P(d < z) <= max(\sigma^2/(\sigma^2 + (d - \mu)^2, (d < \mu))) \quad (1)$$

To avoid some self shadowing problems, σ^2 is clamped to a small minimum variance parameter, so if the variance is too small and wavery this minimum value will make it constant.

Another consideration to take is that every object of the scene should be rendered to the shadow map, to avoid bad shadow results.

Light bleeding might also appear if 2 overlapping occluders have a big distance between them. The use of a threshold to remap shadow intensity can be used to mitigate this problem.

Convolution Shadow Maps (CSM)

CSM [1] approximates the depth values of a normal shadow map by transforming it into a wave function so that we can use Fourier (2) to deconstruct it and return a blurred map when rendering the shadows.

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

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

Another problem present is the ringing effect, 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.

Exponential Shadow Maps (ESM)

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 3.

$$exp(k * (z - d)) = exp(k * z) * exp(-k * d) \quad (3)$$

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. This can be countered by over darkening the resulting shadow.

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

Comparing VSM, CSM and 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.

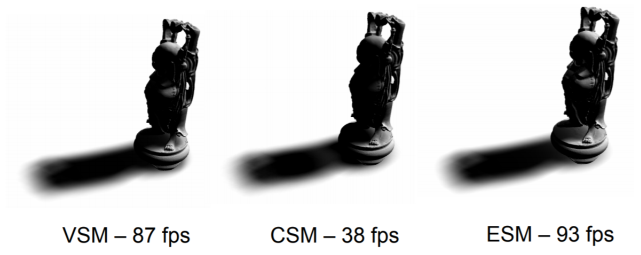


Figure 1. VSM, CSM and ESM compared.

In figure 1 [3], we can see the shadows produced by each of the algorithms and their corresponding performance.

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.

Achieving shadows with variable penumbra

Percentage Closer Soft Shadows (PCSS) [6] is the current State of the art for drawing shadows. It 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.

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, as demonstrated in equation 4.

$$w_{penumbra} = \frac{p_s^z - z_{avg}}{z_{avg}} w_{light} \quad (4)$$

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

IMPLEMENTATION

This work was 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 was developed for android, Android Software Development Kit (Android SDK) was chosen for it's development, and, although generally coded in Java (since Android runs on a Java virtual machine), the App was developed mostly in C/C++ through the Java Native Interface (JNI).

Considering the related work, the shadow mapping implementations were focused on PCF, VSM and ESM to achieve a soft shadow, together with PCSS to enable variable sized penumbrae.

With 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 (available here: https://github.com/ARM-software/opengl-es-sdk-for-android), since it already had a simple Perspective Shadow Mapping example, which it was used to implement PCF, VSM, ESM and PCSS.

Algorithm Implementation

Shadow Mapping

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

To create a shadow map a a texture was created to hold it's values. This texture was defined to hold only the depth values (GL_DEPTH_COMPONENT24). The minification and magnification filters were set to use nearest filtering and the wrapping method used was clamping.

The base project had GL_TEXTURE_COMPARE_MODE set to GL_COMPARE_REF_TO which would diretly compare the depth of the point accessing the shadow map to the value of the shadow map and return either 0 or 1, if it was in shadow or not. This value was changed to GL_NONE to be able to adapt it.

A Frame Buffer Object (FBO) was created and the texture was bound to it.

With the texture to hold the shadow map values created, the shadow map can now do a rendering pass from the position of the camera. This is done by binding the FBO with the shadow map texture, and doing a rendering pass. The color values to be written can all be disabled, since the rendering pass will only record the depth values.

After the values are written into the shadow map, we pass it to the normal rendering pass.

In the fragment shader, each position will be transformed into the light space and then will access the shadow map. The returned value is then compared to the transformed position depth, if the value in the texture is lower, the point is in shadow, otherwise it is in light. In the comparison a threshold is inserted to mitigate the shadow acne problem. This value needs to be adjust for different scenes since it also introduces light bleeding problems.

PCF implementation

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. The percentage in which the point is in shadow is determined by the average calculated.

This is done by offsetting the transformed position and accessing the shadow map with the offset value, multiple times while summing the values returned from the comparison of the value in the shadow map and the depth of the transformed position (which is either 0 or 1), and then returning the sum divided by the total amount of accesses.

The amount of accesses is defined by the tap, for example, a tap of 5 corresponds to 5x5 accesses since they are accessed within a square. The offset is calculated with the positions of this square, which has the original position ate the middle.

VSM implementation

Due to how VSM works, there were some changes that had to be done to our implementation.

As before, we created and bound the texture that would old the shadow map. The first difference is in the filtering, which was changed to GL_LINEAR since we could take advantage of this so our shadow map could be 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 GL_DEPTH_COMPONENT24, glTexImage2D was used with GL_RG32F.

A render buffer was also generated and bound so the shadow map was created correctly.

These were then bound to one FBO, as done previously.

To render the shadow map we disabled the blue and alpha colors using the color mask, since it only needs two components.

The fragment shader pass of the VSM had to be changed too.

Listing 1. Moments calculation

```
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);
```

As seen in listing 1, we got the gl_FragCoord.z, which corresponds to the depth value on that fragment, and stored it in the red value. In the green value we store the square of the depth, adjusted using partial derivative to give it a bias per pixel.

In the normal rendering pass, the fragment shader also has to be adapted.

Instead of getting only the depth value from the shadow map, we now get two values, the two moments (depth and depth squared) needed to calculate the variance, as demonstrated in listing 2.

Listing 2. Variance calculation

```
float variance = moment.y
               - (moments.x * moments.x);
variance = max(variance, 0.00005);
```

The minimum value of the variance is set to mitigate some shadowing problems.

With the variance calculated we can then calculate the percentage in which the point is in shadow, as shown in listing 3.

Listing 3. Shadow calculation using variance

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

The smoothstep function is used to mitigate the light leaking problem with overlapping shadows.

The p_max value is then returned and multiplied by the color to return the point shadowed.

ESM implementation

The ESM implementation was easily implemented with what had already been done. The filter used by ESM 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, after getting the filtered value the shadow can be calculated.

Listing 4. Shadow calculation using exponential

```
 float c = 100.0;
 float shadow = clamp(
    exp( -c * (modelDepth - smDepth)),
    0.5,
    1.0
);
```

The shadow calculation is show in listing 4. The c value changes the penumbra size of the shadow. 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.

PCSS implementation

As previously established, PCSS is composed by three steps.

The first step is the blocker search, where, similarly to PCF, the values of an area in the shadow map will be accessed but averaging the depths returned instead of the comparison between depths. The points that are not in light are discarded from the average, while the ones that are in shadow are counted in.

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, since the point is fully lit.

After getting the average depth, the width of the penumbra needs to be calculated. In this step, the distance between the position and the block is needed as well as the average depth returned in the blocker. 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, changing the values used by the soft shadow solutions.

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 (available here: `https://developer.qualcomm.com/software/snapdragon-profiler`), 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 and a Qualcomm Adreno 630.

The Snapdragon Profiler is installed in the computer, which is then connected to the smartphone via USB. 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 the multitude of metrics available in the profiler, a special importance was given to to metrics recording CPU, memory and GPU usage, since these are the most important parts for our application to run smoothly and to verify where the most stress is located.

Framerate is the measurement defined as the number of frames that are presented in a second, designated as Frames per Second (FPS). It is one of, if not the most widely used measurement to determine the performance of a graphical application. For this reason it was measured and used as one of the metrics for the tests. It was calculated in the app by counting each render pass and dividing it by a timer.

Android Studio was also used to check the Logcat, where the average FPS for each minute was printed, and also for taking snapshots of the device's screen.

We also had multiple scenes for testing, each with it's own object, from the McGuire Computer Graphics Archive [8], imported using tinyObjLoader library, available at: `https://github.com/tinyobjloader/tinyobjloader`.

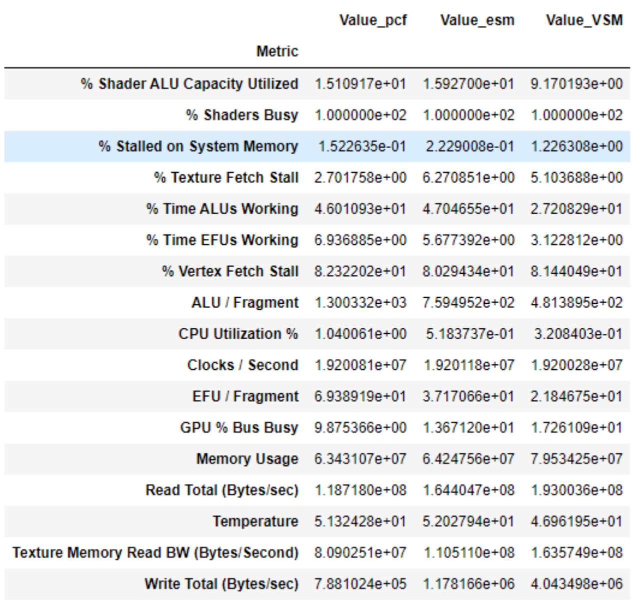
These objects were bunny.obj [11], dragon.obj [10] and sponza.obj [5].

RESULTS

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.

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 7x7 PCSS tap, 1080*2220 map



Metric	Value_pcf	Value_esm	Value_VSM
% Shader ALU Capacity Utilized	1.510917e+01	1.592700e+01	9.170193e+00
% Shaders Busy	1.000000e+02	1.000000e+02	1.000000e+02
% Stalled on System Memory	1.522635e-01	2.229008e-01	1.226308e+00
% Texture Fetch Stall	2.701758e+00	6.270851e+00	5.103688e+00
% Time ALUs Working	4.601093e+01	4.704655e+01	2.720829e+01
% Time EFUs Working	6.936885e+00	5.677392e+00	3.122812e+00
% Vertex Fetch Stall	8.232202e+01	8.029434e+01	8.144049e+01
ALU / Fragment	1.300332e+03	7.594952e+02	4.813895e+02
CPU Utilization %	1.040061e+00	5.183737e-01	3.208403e-01
Clocks / Second	1.920081e+07	1.920118e+07	1.920028e+07
EFU / Fragment	6.938919e+01	3.717066e+01	2.184675e+01
GPU % Bus Busy	9.875366e+00	1.367120e+01	1.726109e+01
Memory Usage	6.343107e+07	6.424756e+07	7.953425e+07
Read Total (Bytes/sec)	1.187180e+08	1.644047e+08	1.930036e+08
Temperature	5.132428e+01	5.202794e+01	4.696195e+01
Texture Memory Read BW (Bytes/Second)	8.090251e+07	1.105110e+08	1.635749e+08
Write Total (Bytes/sec)	7.881024e+05	1.178166e+06	4.043498e+06

Figure 2. VSM, CSM and ESM compared.

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.

There is some valuable insight obtained by comparing these results.

The fact that VSM is the most demanding solution is confirmed, although it was more demanding than previously though. This could be due to the fact that VSM uses two color values from the shadow map instead of one depth value.

ESM consistently got better results than PCF. This was increasingly noticeable the bigger the tap/filtering used. 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 ESM does less calculations between accesses this behavior is expectable.

It was also shown that a variable penumbra shadow in real time rendering is achievable and, with further tweaking, it was possible to achieve a framerate of 60 FPS.

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.

As previously though, figure 2 shows us that VSM has a bigger toll on memory usage in all aspects, from reading to writing and to total amount used. This is due to the fact that VSM has to both read and write more data to the shadow map.

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.

Comparing the usage of the shader resources, 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, except for some minor differences.

We could also confirm that the application was not CPU intensive, although there were some minor differences between each solution.

Testing VSM

Since VSM can achieve the best visual results, multiple settings were changed to check if a good VSM result could be achieved with an acceptable level of performance. Sponza scene was used due to it's realistic representation and good performance, while also being more susceptible to artifacts, which would be easier to notice.

Starting out with the settings used in the preliminary test for Sponza using VSM, we compared the visual quality that each tap presented. A 3x3 tap resulted in shadow map with too many noticeable artifacts. A 5x5 had some minor artifacts compared to 7x7 which barely had any, but the performance that a 5x5 tap brought led us to keep using it in further testing.

Next, the PCSS taps were changed. From this test we discovered that decreasing the PCSS tap had a big performance improvement while having little impact on the quality of the image. A 3x3 PCSS tap was used onward, which could, at the moment achieve an average of around 30 FPS.

The next change was the size of the shadow map, comparing 1080x2220 size with 810x1665 and 540x1110. The increase in performance by diminishing the size of the shadow map was not substantial, while the decrease in image quality was noticeable. Because of this the size of the shadow map was maintained as it was (1080x2220).

Finally, we compared VSM with these settings with similarly performing ESM and PCF settings.

ESM was used with a 7x7 filtering size, a 4320x8880 shadow map size and a 3x3 PCSS tap, achieving 35.5 FPS. PCF was used with a 5x5 tap, a 4320x8880 shadow map size and a 3x3 PCSS tap, achieving 36.2 FPS.

By comparing these solutions visually, the conclusion was that PCF was the better pick for a scene like this. Comparing it with VSM, it was both better looking and better performing, and ESM had some issues with the shadows due to it's difficulty in working alongside PCSS.

Comparing ESM with VSM

The previous test showed us that ESM was not a good solution for a scene like Sponza. We concluded that PCF should be used with a scene with many shadows and visually complex.

Still on the Sponza scene with the settings used from the comparison with VSM, we tried to improve the PCF performance to try to achieve 60 FPS, while still having a good quality shadow.

The first obvious change was the PCSS step, since, as seen before, it increases the performance a lot without impacting the shadow quality too much. This increased our average framerate to 47.8 FPS.

Then the size of the shadow maps was tuned down. Different sizes were tested and 1080x2220 was able to achieve 60 FPS, being capped at 60 due to synchronization with the screen frequency. Also to notice that 2160x4440 almost got to 60 FPS, averaging around 58.7 FPS, with better visual quality, since the shadow map size proved to be important for the quality of the shadow, although in this case it was not too much of a difference, since these resolutions are already pretty high.

With this we proved that variable soft shadows could be achieved in a mobile environment with actual real-time performance.

We also compared ESM and PCF in the bunny scene, to check if ESM would have more of a chance in a visually simpler scene, which in fact did.

Again using the tests from the preliminary tests, we determined that we should start with PCF using a 3x3 tap and ESM using a 5x5 filter tap, since both performed similarly. The PCSS was then lowered to 3x3, which made both solutions achieve 60 FPS.

The resulting images showed that, in this case ESM could behave good enough as to provide a good quality shadow, so good in fact as to be better than the one PCF produced.

Further testing the bunny scene using ESM and changing the shadow map size showed that we could increase the size to 1620x3330 while still achieve 60 FPS. Only by increasing to 2160x4440 did we fall bellow 60 FPS, achieving 55.7 FPS. To mention that these increases got rid of some artifacts present in the bunny surface, so it might be better to use a slightly higher resolution with a lower performance.

CONCLUSION

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.

PCSS was also implemented so a variable soft shadow could be implemented. 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.

VSM was not able to provide a good quality shadow at a low performance, since other solutions could have better looking shadows at a similar performance. When VSM produced a good shadow, it was at some high performance costs, making it not viable for mobile environment.

ESM as some inconsistency and bad shadowing problems when used with PCSS, this makes it only acceptable to use in certain scenes, when it can be better than PCF.

PCF proved to be the best solution to use overall, being able to achieve good quality shadows with good performance on a mobile device.

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.

REFERENCES

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

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

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

[4] Franklin C Crow. 1977. Shadow algorithms for computer graphics. *Acm siggraph computer graphics* 11, 2 (1977).

[5] M. Dabrovic. 2002. Sponza. (2002). `http://hdri.cgtechniques.com/~sponza/files/`

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

[7] A. Gruber. 2019. *Mobile GPU approaches to power efficiency*. Technical Report. Qualcomm.

[8] M. McGuire. 2017. Computer Graphics Archive. (July 2017). `https://casual-effects.com/data`

[9] Kevin Myers. 2007. Variance Shadow Mapping. *Retrieved May* 11 (2007).

[10] Stanford University. 1996a. Dragon. (1996).

[11] Stanford University. 1996b. Stanford Bunny. (1996).

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