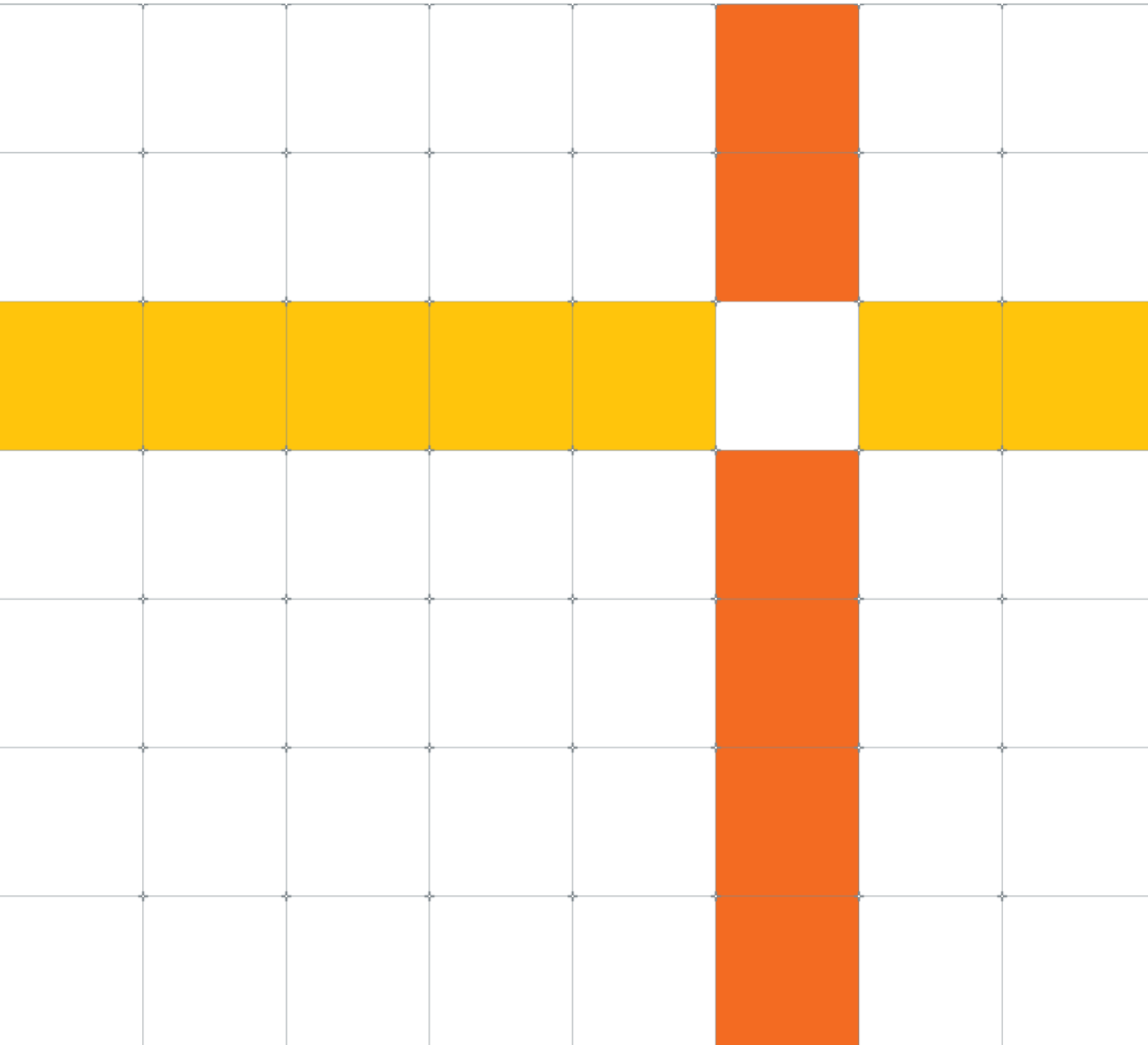
Foveated Rendering
Current and Future Technologies for Virtual Reality

Daniele Di Donato – Staff Software Engineer – Arm

November 2017



Non-Confidential Proprietary Notice

LES-PRE-20349

Contents

1 Introduction

Virtual Reality (VR) is becoming increasingly popular due to its ability to immerse the user into an experience. Those experiences can vary from watching a movie in a simulated theatre, having a look at your personal pictures as though they were paintings in a museum or finding yourself in front row seats of a huge sporting event. These specific experiences don't stress the device hardware to its maximum limit, and are usually less demanding compared to the other mainstream VR experience; gaming. The "new" era of VR is born with gaming as the main driving force, and that is reflected by the number of announcements made about VR at gaming and graphics conferences such as GDC and E3. The revolution started for desktops computers with the first developer kits, and gradually expanded to affect the mobile. With the introduction of the Samsung GearVR and Google Daydream consumer headsets, which simplified and expanded the VR adoption thanks to cable-free usage, lower prices and a focus on mass market users saw a distinct advantage of mobile compared to similar desktop counterparts.

Gaming is still the biggest driver of VR even on mobile, and that puts pressure on the performance of the whole system. It is worth remembering that the performance of a VR application is an important factor to its usability, since low or varying performance will cause the typical negative impacts of VR on a person, such as dizziness and motion sickness.

The high-performance requirements are:

- The need to render the scene twice since we have two different points of view to represent. This will double the amount of vertex processing executed and the time spent executing draw calls in the CPU.
- Having high resolution scene and screen framebuffers (>=1024p). This is needed since the lenses on the HMD magnify the pixels in the headset screen making them more noticeable. High resolution framebuffers increase the amount of fragment processing executed each frame.
- Having a high refresh rate (>=60 Hz). This is needed to reduce the motion to photon latency and avoid dizziness and motion sickness caused when motion doesn't trigger the expected visual response in the expected time.

All these requirements combine themselves in a VR application and new algorithms are under development to achieve higher performance while doing more work.

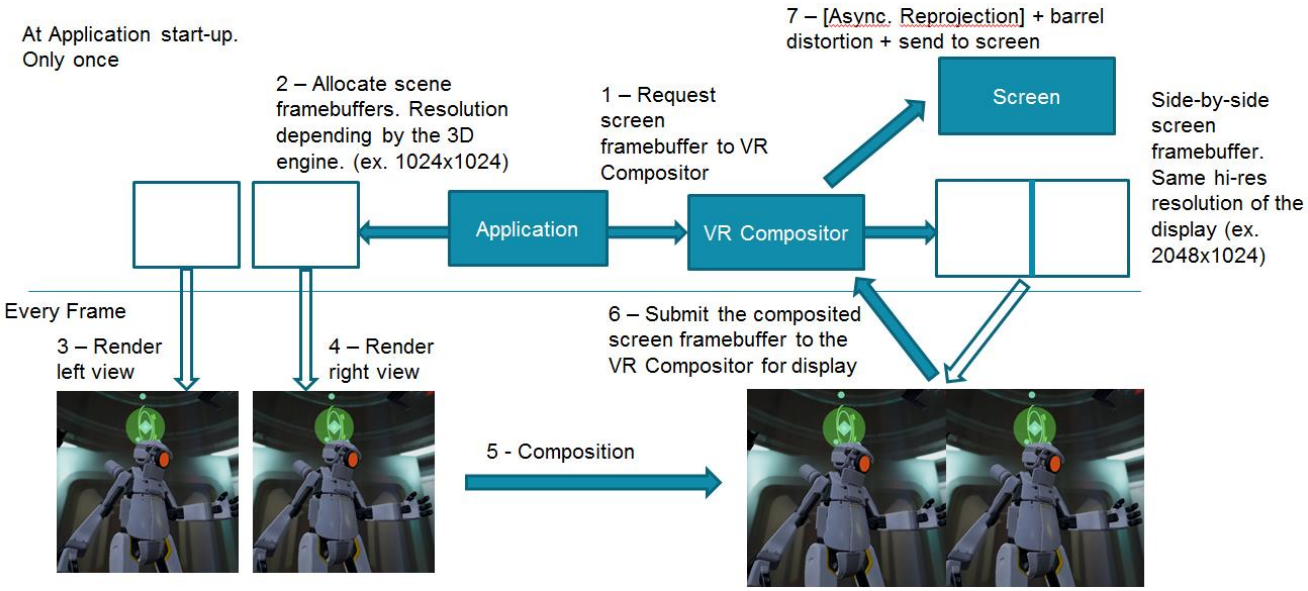


Figure 1: Typical VR Stereo Rendering flow.

To reduce the impact of rendering the scene twice, Mali GPUs support the Multiview extensions[1] . These extensions have the effect of reducing CPU load and optimizing vertex load and will be described in the "API extensions needed" section.

To reduce the latency and have a high refresh rate VR compositors typically use an Asynchronous Reprojection algorithm which is capable of re-project a previous frame according to the user head movement if the rendering of the new frame is taking too long. This algorithm can be considered as a safety net since it works if the application is capable of sustaining high refresh rates (>=60Hz) for most of its execution. If the application is not capable of maintaining that performance, the Asynchronous Reprojection algorithm will not be able to cope with the amount of head movement the user can do between frames.

The other requirement poses huge pressure on fragment processing but fortunately, two important considerations come into play when experiencing VR. The first of these considerations is that the lenses used in VR HMDs produce a pincushion distortion that needs to be handled by the VR compositor which applies a counter-acting barrel distortion to the screen framebuffer. Applying this distortion causes the center area of the scene framebuffer to be sampled with high resolution while the surrounding area is under-sampled and that means the GPU spent time rendering pixels that are not used.

[1] The Multiview extensions supported are:

GL_OVR_multiview: **https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview.txt**
GL_OVR_multiview2: **https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview2.txt**
GL_OVR_multiview_multisampled_render_to_texture:
https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview_multisampled_render_to_texture.txt

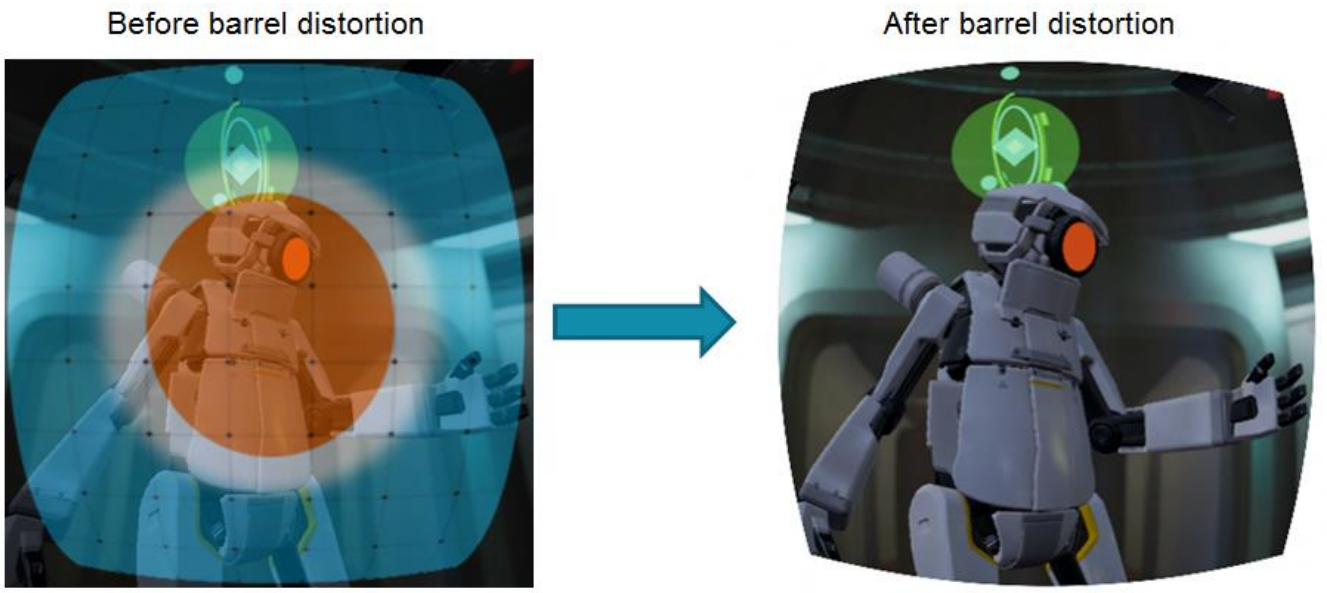


Figure 2 : Effect of barrel distortion. Pixels in the red area are sampled with higher frequency compared to the ones in the blue area.

This information is useful for variable-resolution shading algorithms that improve the resolution in the center of the Field of View (FoV), which is sampled with high frequency by the barrel distortion (red area in Figure 2), while reducing it on the sides to save fragment shading resources (blue area in Figure 2). The second consideration is about human physiology. Human eyes don't retain the same sharpness of detail in all the FoV and this can be exploited to improve performance of VR applications. If we are able to track your eye location, it can be defined which areas of the image require more resolution and which areas don't, thus improving performance without as much strain on the device.

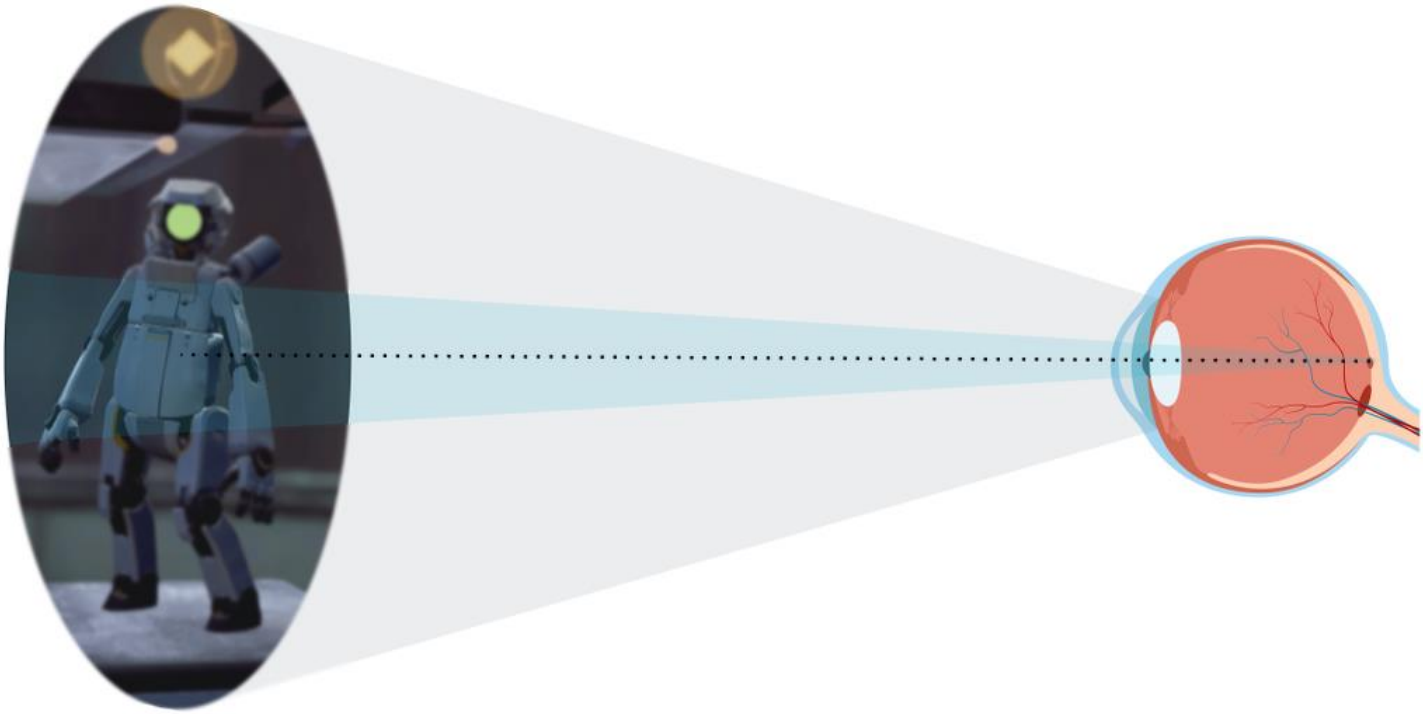


Figure 3: Variation of sharpness of details perceived by the human eye based on field of view angle. Perceived sharpness decreases when moving away from the center line of vision.

This idea is known as Foveated Rendering. The term Foveated comes from “fovea” which is the part of the human eye that has the ability to see with the highest resolution. The “periphery” is the area surrounding the fovea which has lower resolution thus less capable of detecting high frequency details.

Foveated rendering techniques reduce the fragment shading load on the GPU compared to variable-resolution shading, therefore improving performance and battery life of devices. There are various methods to achieve this and we are going to talk about some of them in this whitepaper. We will show what is achievable on current devices using the Multiview extensions to create a Variable-resolution shading algorithm and then further improve it with eye tracking to develop a fully functioning Foveated rendering algorithm.

2 Challenges of VR

In this section, we will explain the challenges that VR poses to the rendering system (CPU, GPU, etc). We will start by introducing CircuitVR, our internal demo specifically created to demonstrate the efficiency of the techniques described. We will start with a description of the Multiview extensions as they are the foundation of all the other techniques that will follow. We then describe the lens-matched shading implementation and its evolution to Foveated Rendering by the means of eye-tracking. Lastly, various optimizations are described to further improve the efficiency of these techniques.

2.1. Use case: CircuitVR

Before explaining the various technologies that solve some of the challenges in VR, we would like to give a brief introduction to the CircuitVR demo since most of the techniques and measurements have been done while building it.

Figure 4: Circuit VR Demo

CircuitVR is a demo developed internally within Arm using a customized version of Unreal Engine that has been modified to support the Foveated Rendering algorithm using the Multiview extensions available on current devices that use Mali-T6xx onwards. The demo was tailored to take advantage of the Foveated Rendering algorithm, and had some specific requirements to be sure we could have benefits from the algorithm since Foveated Rendering gives the best result for fragment bound applications. We execute around 60-70 draw calls

each frame (depending on the camera pose) and a maximum of ~55k triangles per frame. Before we dive into the specifics of the algorithm, it is good to have an overall introduction to the algorithm and each of its steps.

2.2. Variable-resolution shading and Foveated Rendering

Variable-resolution shading and foveated rendering can be implemented in current Mali GPUs, such as Mali-T8xx, Mali-G7x and Mali-G5x series GPUs, using the Multiview extensions. These solutions can be used to reduce the GPU load and improve performance of your application. Of course, these techniques are no silver bullets, and need to be applied with parsimony only when the application would really benefit from it. To give the expected performance benefit, the application needs to be fragment bound and have some spare vertex load to be used. This is needed because, as explained in further detail later, we are going to add more work for the vertex processing and reduce it for the fragment. In typical applications, fragment processing is by far the more demanding load, but there can be cases where vertex load is not too dissimilar, and applying this technique will not have the expected results.

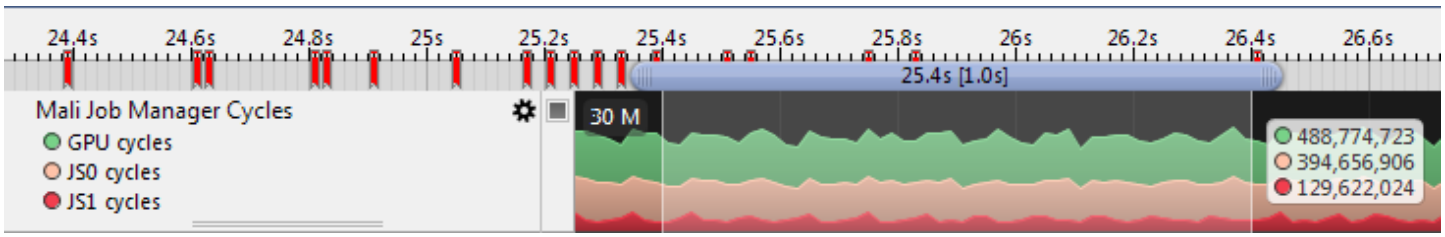


Figure 5: CircuitVR example of GPU load when rendering using Stereo Rendering. JS1 cycles represent the vertex load while JS0 cycles represent the fragment load for 1 second time interval. Fragment load is more than 3 times the vertex load.

Rendering pipelines that support these techniques have to undergo some small modifications. Usual VR applications render the scene twice, once for each eye. This is what is typically called Stereo Rendering. The image for each eye is rendered separately executing the same draw calls twice with only some small modification to cope with the different view of each eye. Using the Multiview extensions, we can avoid issuing the same set of draw calls twice which in turn helps improving both application and driver performance. Multiview is fundamental for implementing these techniques as without it, the application can easily become CPU bound. Mali implementation of the Multiview extensions supports rendering up to 4 simultaneous views at the same time, this is key to implementing Variable-resolution shading and Foveated rendering. The 4 views will have a lower resolution compared to the classic Stereo rendering (which only uses 2 views) and this is where the main benefit for fragment bounded applications comes from. The 4 views have a particularity in addition to be low-resolution, they represent different views of the same scene. Two of them have the same point of view as the classic Stereo Rendering case and represent the whole scene as visualized by the left and right eyes. The other two views have a lower field of view and same orientation and will represent a zoomed version of the image.

Figure 6: Left eye periphery – Low resolution framebuffer (for example 366x366). FoV: 90 degrees

Figure 7: Left eye fovea – Low resolution framebuffer (for example 366x366). FoV: 38.5 degrees

Since the resolution of the 4 views is the same, this means the images with smaller FoV will be able to represent only a small part of the whole scene seen by the user. When the zoomed area is at the center of the image, we are implementing a Variable-resolution shading algorithm since the resolution of the center will be the highest possible (dense pixel grid in the picture below) while the surrounding area will have a lower resolution (sparse pixel grid) defined by the size of the framebuffers used.

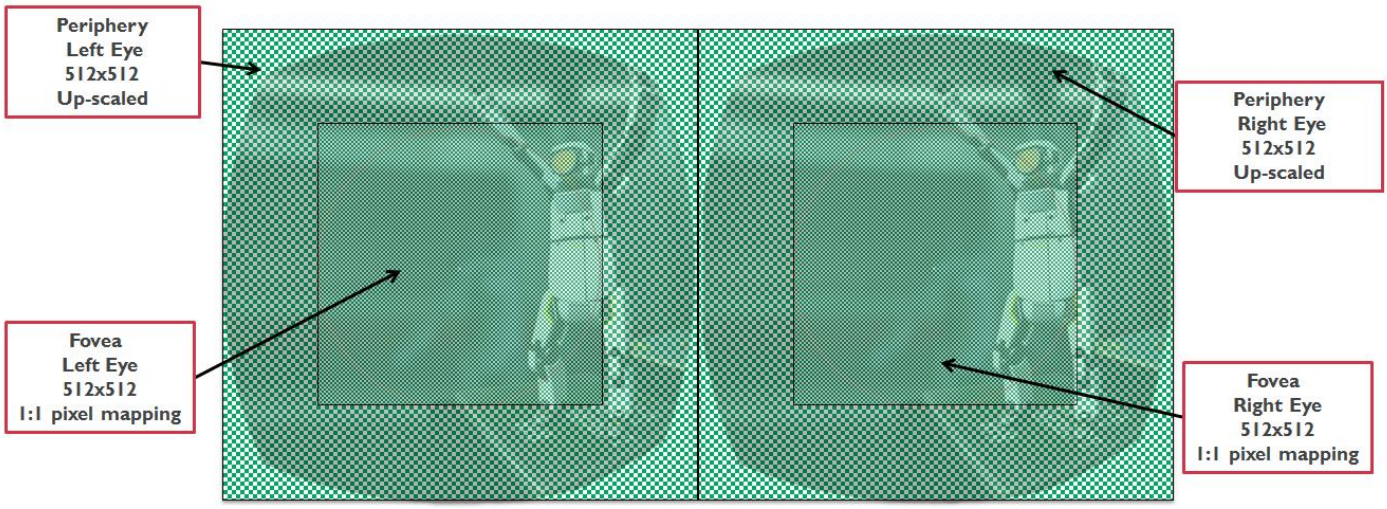


Figure 8: Variable-resolution shading algorithm

One problem with this technique is that the user can see the difference in resolution if we reduce the resolution of the surrounding area too much, so this technique limits the developer's options when it comes to the framebuffer resolution that they can use.

That's where Foveated Rendering with eye tracking comes into play. If we are able to sample where the eye of the user is looking at, we can move the "zoomed" fovea image in front of the eye giving the illusion of constant higher resolution. This can be achieved with even lower framebuffer resolutions since the user's fovea will always see the high-resolution image.

After we render the 4 views we will combine them into the screen framebuffer (Side-by-Side texture below) using some blending to improve the quality. Other improvements and optimization can also take place and we will describe some of those techniques (See Blurring offset/decrease contrast and Composition).

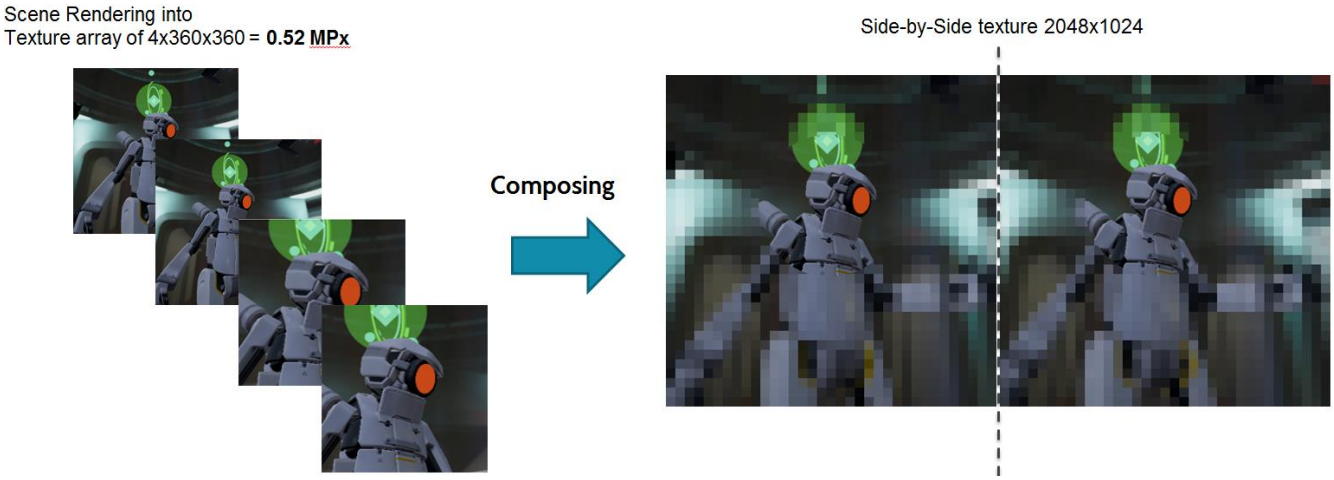


Figure 9: Combined screen framebuffer – High resolution framebuffer (for example 2048x1024)

3 Implementing Variable-resolution shading and Foveated Rendering

3.1. Prerequisite: Multiview

Multiview is the technology that currently allows implementing Variable-resolution shading and Foveated Rendering in mobile devices. The extension allows you to render into a texture array with a single draw call giving the shaders access to the index of each element of the array that the GPU is writing into. This allows each view to have a different output depending on the index of the array we are writing. This is great for VR applications which can often suffer from being CPU bound due to the many draw calls executed. The application can simply specify vertex attributes and uniforms in form of arrays and use the gl_ViewID_OVR index to access the attributes used to render the left and right eyes. In the code below we are rendering 2 views as defined by the layout identifier. The application code has been modified to provide the modelViewProjection matrix for each view as an array and the render target is backed by a GL_TEXTURE_2D_ARRAY of size 2.

Table 1: Example shader using Multiview

```
#version 300 es
#extension GL_OVR_multiview : enable
layout(num_views = 2) in;
in vec3 vertexPosition;
uniform mat4 modelViewProjection[2];
void main()
{
        gl_Position = modelViewProjection[gl_ViewID] * vec4(vertexPosition, 1.0);
}
```

Below are two graphs from the Arm Streamline profiling tool which refer to the initial version of the CircuitVR demo, which used classic Stereo Rendering. In this graph, we were capturing a scene with ~140 drawcalls per-view. Enabling Multiview allowed us to reduce the amount of time spent on the CPU by the render thread (the longest thread running each frame) by almost 22%. In theory, draw call execution time is reduced by 50% because we halve the number of draw calls issued. Of course, in practice, the overall CPU savings depend on the amount of draw calls executed. The Multiview extensions are also able to reduce the amount of vertex load by optimizing the vertex shaders that use the extension by separating the parts that don't depend on gl_ViewID_OVR and execute those just once.

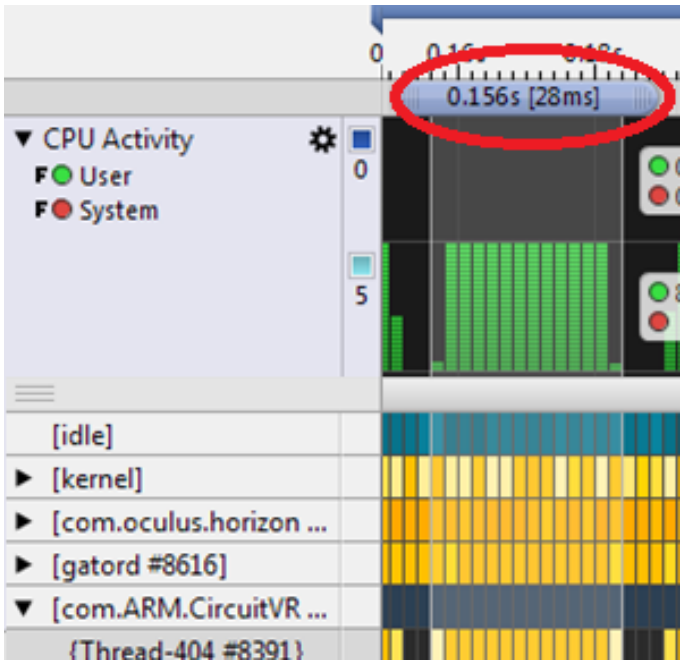


Figure 10: CPU time for renderer thread without Multiview

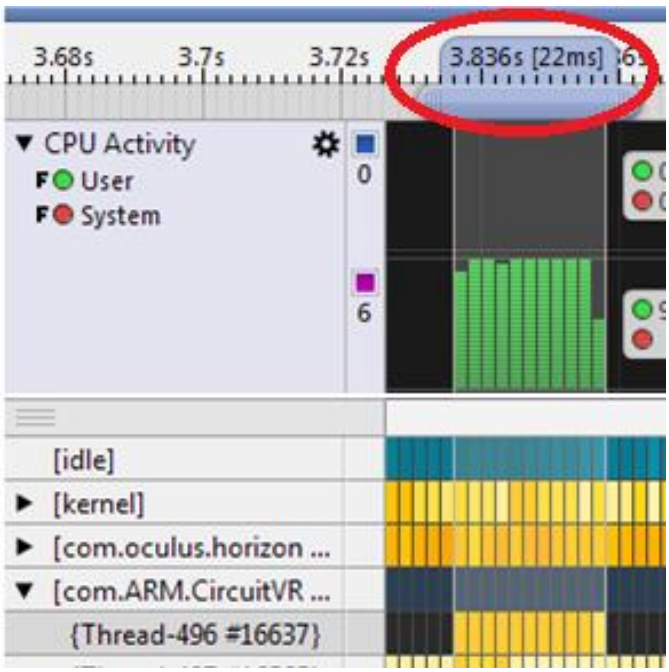


Figure 11: CPU Time for renderer thread with Multiview

3.2. Variable-resolution shading

For Variable-resolution shading we extend this concept and instead of rendering 2 views, we render 4 views. Two of them will render the scene as we will do for normal VR application for the left and right eye. The other two views will have a reduced FoV that will make the view looked like we are zooming in onto the scene.

Figure 12: Left and Right Eye

Figure 13: Left and Right eye with decreased FoV

Rendering 4 views seems counterintuitive to just rendering 2 views. The trick to improve performance is to render the 4 views with much lower resolution and then combine them into the screen framebuffer which has higher resolution.

The resolution we want to use is directly related with the FoV by the formula:

$$FOV_{ins} = \tan^{-1}(\tan\left(\frac{FOV_{ext}}{2}\right) * ratio\ ) * 2$$

FOV_{ins} is the FoV angle to be used for fovea views. FOV_{ext} is the current FoV used when rendering the whole scene for the periphery and $ratio$ is the percentage of the new resolution we want to use compared to the classic VR resolution we would have used. For example, in our demo we wanted a 35% reduction in the framebuffer size compared to the classic case which uses 1024x1024 pixels per-view. This would give us a render target of around 358 pixels (we then rounded it to 366). If we have a FOV_{ext} of 90 degrees the formula will give us a FOV_{ins} of 38.580 degrees.

In our demo, instead of rendering the scene at 1024x1024 pixel per-view (2 for classic VR) and then copy those images into a 2048x1024 screen framebuffer we render using 366x366 pixel per-view (4 for Foveated rendering) and then composite those images into the 2048x1024 screen framebuffer. Variable-resolution shading will therefore render 75% less pixels compared to classic VR rendering.

At the end of the rendering, the 2 views with decreased FoV will be copied directly to the 2048x1024 screen framebuffer with a 1:1-pixel ratio between source and destination targets. This means that the quality for that 366x366 pixel area in the screen framebuffer will be the same as if we were rendering at full resolution. The remaining pixels in the screen framebuffer will sampled from the other 2 views. In this case the pixel ratio is 1:X since a single pixel in the 366x366 source image will cover a bigger number of pixels in the 2048x1024 image. In our demo, we implemented this composition as a separate render pass, but an optimized implementation may leave this composition to the VR compositor used if it exposes this possibility.

Figure 14: Final combined Image for the left eye (pixilation effect added to better show the two areas).

Variable-resolution matched shading provides a good balance for existing devices and HMD as it reduces the amount of fragment shading executed on current generation devices. One problem with this technique is that the resolution used for main rendering needs to be high enough (we suggest 512x512 per-eye) as reducing the resolution too much can be visible for the user. However, the reduction in the number of fragment rendered is high enough (see below for more info) to make this technique effective. It needs to be noted that rendering 4 views causes GPU vertex load increases since we have 2 more views to render compared to classic Stereo Rendering. Vertex load should be considered before applying Variable-resolution shading since it can cause the GPU to become vertex bounded.

3.3. Composition

Once the 4 views have been rendered, it's time to combine them into the screen framebuffer. Our demo uses Unreal Engine 4.14, which executes a composition stage of the rendered scene into the final framebuffer provided by GearVR. This is where we can plug our composition code. Recently, Oculus Mobile SDK allowed developer to write into a texture array which can be drawn directly with Multiview. This improves the performance of classic VR application since it removes the need for the composition stage which is delegated to the Oculus GearVR library and can be executed at the last stage as part of the Asynchronous Reprojection algorithm.

The math for the composition is pretty simple, for each eye we render a quad that covers the whole eye in the screen framebuffer image and for each pixel we calculate its distance from the center of the quad (d in Figure 11). If the distance is less than the resolution of the fovea view, we use it to sample data from the fovea view image. If it's more, we sample from the periphery image. It must be noted that a scaling factor need to be applied to the texture coordinates used by the quad when sampling the fovea image, and this factor is based on the resolution used to render the fovea view. When sampling from the periphery, no change needs to be applied to the texture coordinates.

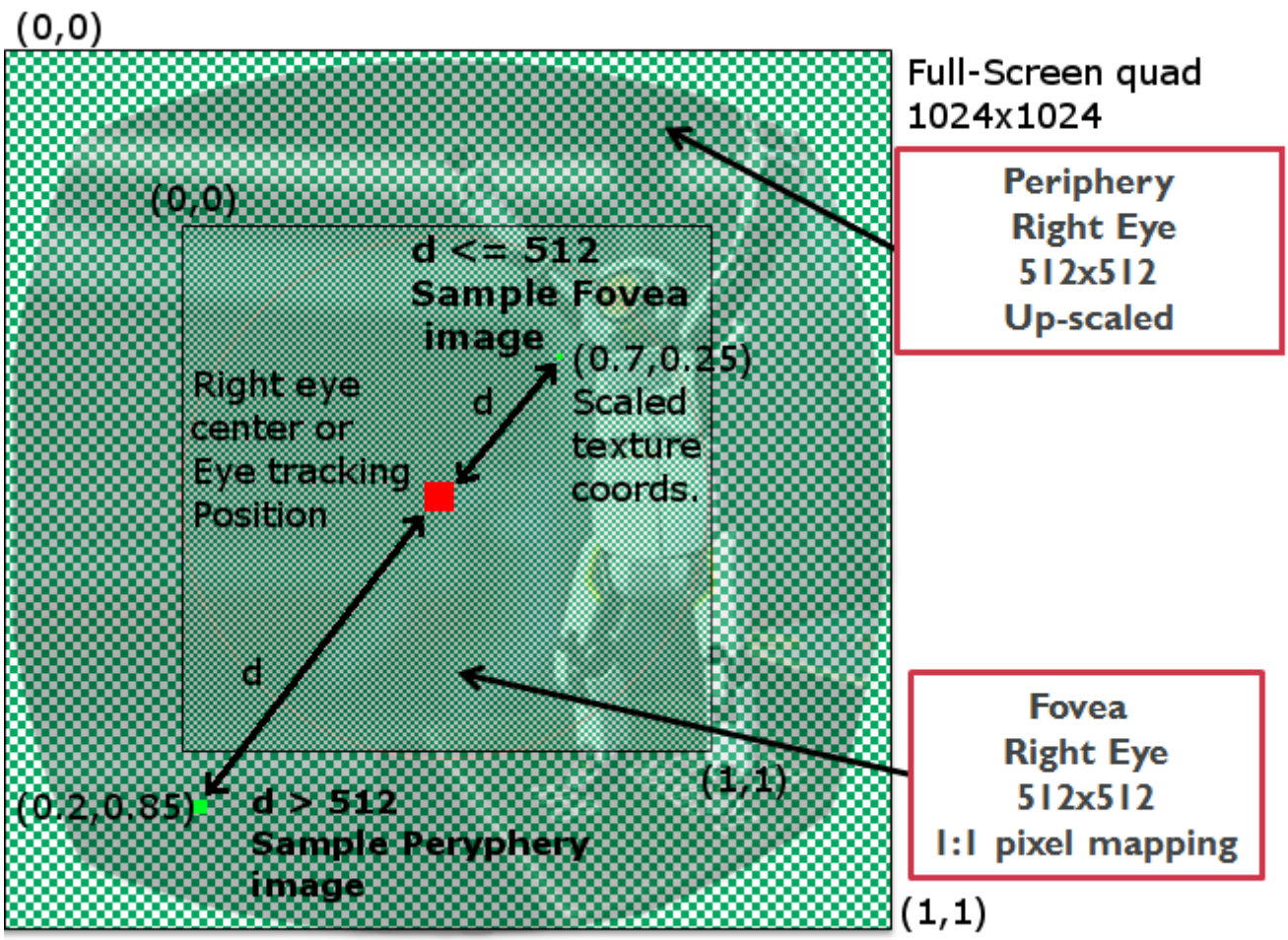


Figure 15: Composition phase for the right eye.
The screen framebuffer resolution is 1024x1024 per-eye. Fovea and periphery are rendered with 512x512 resolution.

3.4. Foveated Rendering using eye tracking

As a person's eyes move around the scene, the area that needs to be shown in higher detail constantly changes. This makes eye tracking a requirement for this implementation of Foveated Rendering as the algorithm needs to precisely track the eye location to always show the highest detailed picture in the center of the Field of View. The eye tracking system we used in the demo feeds the application with the location of where on the screen the eye is looking at. This position is then used in our rendering pipeline in two places, at rendering time and at composition. During rendering, the 2 fovea views (which have lower FoV) will need to move their view center to match the location the eye is currently looking and this is achieved by using the eye tracking information as a translation offset for the projection matrix of the 2 fovea views. This works best since it maintains the same camera center for both fovea and periphery views. At composition, the same eye tracking values are used to understand where the fovea view maps in the screen framebuffer and where the periphery view fills the rest. The change is pretty simple since instead of measuring the distance from the center of the final image we measure it from the eye tracking position. The improvement compared to variable-resolution shading is not only visual, it's hard to see the difference in resolution for the user, but also in the performance that can be achieved since we can now use smaller FoV for the fovea images that allows using lower resolution framebuffer targets.

For example, using 512x512 when testing variable-shading rendering gave us a good balance between quality and performance improvements. When moving to Foveated rendering we were able to lower the resolution down further to 366x366 maintaining the same quality or even improving it since it was harder for the user to see the lower resolution area thanks to the eye tracking which was always showing the best picture in the fovea region.

Latency of the information provided by the eye tracking device should be also considered. Having a high-performance eye tracker is crucial to avoid any latency which will cause the fovea image to "chase" the eye movement in the screen framebuffer. We need to add to this latency the application rendering time (both CPU and GPU), the VR compositor lens corrections and the scan-out time needed for the display to actually change the pixel color. It's important that the eye tracking data is sampled as late as possible during the rendering to help reducing the CPU rendering latency (the first block in Figure 12). Having a low latency eye-tracking system allows to reduce the delay between the acquisition of the eye positions from the eye tracker and the moment the application request this information (this is the distance between the 2 red dots in Figure 12). The resolution used also reduces the latency since the GPU rendering time will be shorter resulting in a smaller latency and faster feedback. Future methods will allow further reduction of the latency by moving the sampling inside the GPU rendering.

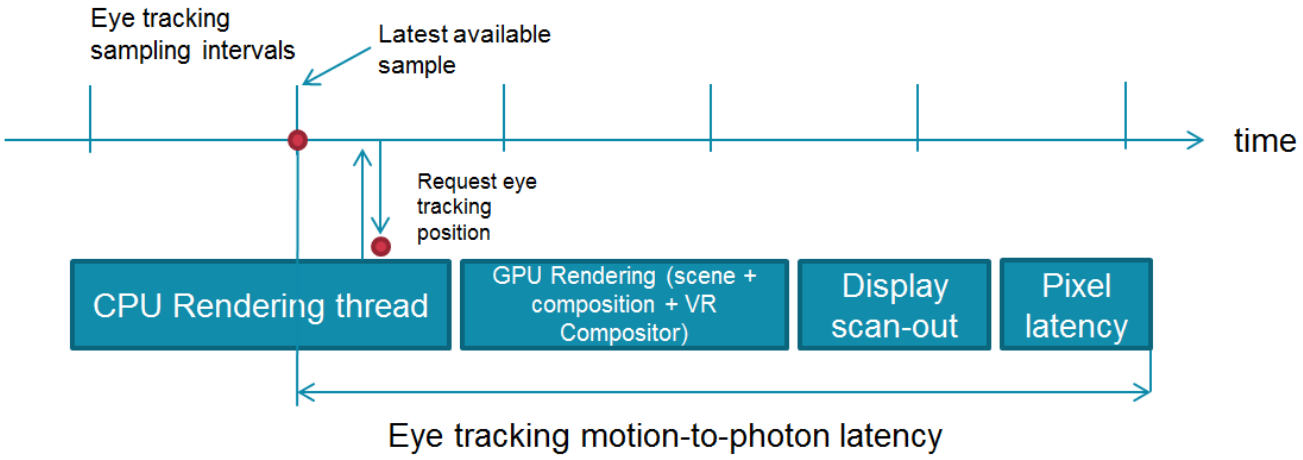


Figure 16: Latency factors when using the eye tracker data

3.5. Optimization with stencil mask

One possible optimization that could be applied is the usage of a stenciling mask for both the fovea and periphery views. This can reduce the number of pixels rendered each frame without any quality loss. When we render the periphery view we can avoid rendering the part of the scene that will be overwritten by the fovea view during compositing, and we can remove the area that is not going to be used due to the barrel distortion applied by the VR compositor. We use circular areas, so we will have the ability to also avoid rendering the corners of the fovea view, and vice versa, they will be overwritten by the periphery view.

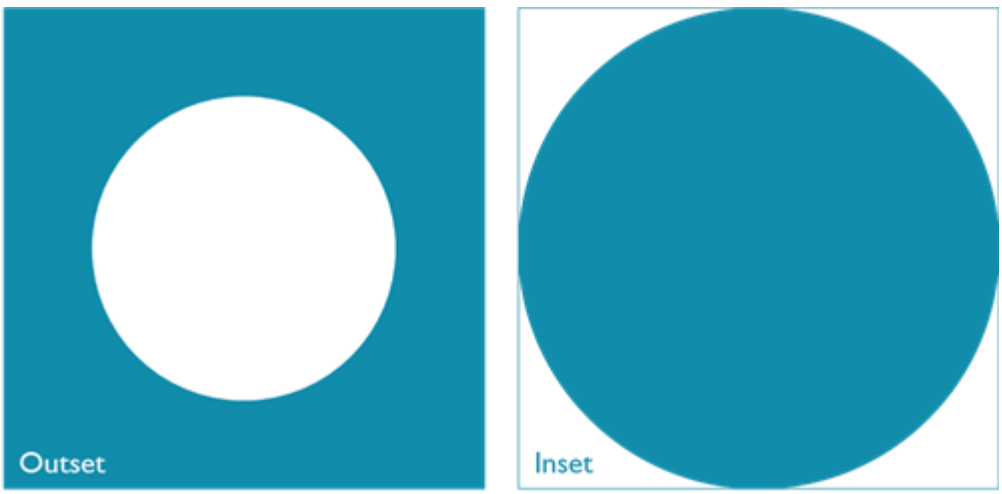


Figure 17: Inset (fovea) and Outset (periphery) masks

If you apply this optimization the number of pixels needed for the fovea becomes:

$$Pixels\ Fovea = \pi * (\frac{View_w}{2})^2$$

$$Pixels\ Periphery = View_w{}^2 - \pi * (\frac{View_w * ratio}{2})^2$$

Where $View_w$ is the width of the rendered view (366 pixels in our demo, we assume square render targets). As the users move their eyes the stencil mask needs to be regenerated for the periphery views while for the fovea can be the same. The overhead introduced by the stencil mask is a quad rendering properly placed to match the eye location.

The trade-off in quality vs performance can be set easily when using this algorithm. For example, we used a framebuffer size that is 35% the size of the one used in classic stereo rendering. A practical example from our demo; stereo rendering was using 1024x1024 pixels per-view while our algorithm uses 366x366 pixels per-view. The user has the ability to define the ratio of the sizes (35% for CircuitVR) compared to the classic rendering (lower values means a more stretched area in the periphery) and also what is the size of the fovea using stencil. If for example, the user would prefer to have better quality in the periphery then it can increase the ratio but in turn that will increase the number of pixels rendered. Instead, the user can decide to reduce the fovea area using the stencil mask and render a bigger portion of the periphery image to cover the area removed from the fovea view. This reduces the number of pixels rendered because one pixel in the fovea view covers a single pixel in the final screen framebuffer while a pixel in the periphery covers multiple pixels, so if we reduce the number of pixels rendered in the fovea we will need to render much less pixel in the periphery to cover up what is left.

The tables below show some examples of how many pixels are shaded for various values of ratio and techniques.

Table 2: Pixels rendered without Stencil Mask

	Resolution per-view	Number of views	Fovea pixels rendered (total for both eyes)	Periphery pixels rendered (total for both eyes)	Total pixels rendered	Reduction %
Stereo Rendering (with Multiview)	1024*1024	2	N/A	N/A	2,097,152	
Foveated 30% reduction, no stencil mask	716*716	4	1,025,312	1,025,312	2,050,624	2.2
Foveated 50% reduction, no stencil mask	512*512	4	524,288	524,288	1,048,576	50
Foveated 65% reduction, no stencil mask	366*366	4	267,912	267,912	535,824	74.4

Table 3: Pixels rendered with Stencil Mask optimization

	Resolution per-view	Number of views	Fovea pixels rendered (total for both eyes)	Periphery pixels rendered (total for both eyes)	Total pixels rendered	Reduction %
Stereo Rendering (with Multiview)	1024*1024	2			2,097,152	
Foveated 30% reduction, with stencil mask	716*716	4	804,870	630,926	1,435,796	31.5
Foveated 50% reduction, with stencil mask	512*512	4	411,566	421,396	832,963	60.2
Foveated 65% reduction, with stencil mask	366*366	4	210,311	242,149	452,460	78.4

Table 4: Pixels rendered with Stencil Mask optimization and fixed fovea view size of 256x256

	Resolution per-view	Number of views	Fovea pixels rendered (total for both eyes)	Periphery pixels rendered (total for both eyes)	Total pixels rendered	Reduction %
Stereo Rendering (with Multiview)	1024*1024	2			2,097,152	
Foveated 30% reduction, with stencil mask	716*716	4	102,892	975,008	1,077,899	48.6
Foveated 50% reduction, with stencil mask	512*512	4	102,892	498,565	601,457	71.3
Foveated 65% reduction, with stencil mask	366*366	4	102,892	254,768	357,659	82.9

The perceived quality depends on a good reduction ratio, which should be chosen to maintain both quality and performance depending on the algorithm used (Variable-resolution shading or Foveated rendering). One thing we have noticed, is that when using a high reduction ratio, the periphery view gets stretched to cover more pixels, and the change in resolution becomes more noticeable especially in the areas where there is high contrast, dynamic objects and specular reflection. Using a low resolution for the periphery will make all the objects that are rendered inside it to be scaled up when composited in the screen framebuffer. Our suggestion is to use a

higher resolution and give up a bit of performance for quality, especially when using only Variable-resolution shading. Using a resolution of 512x512 for the periphery views with a 256x256 fovea view will give good results for both the areas.

3.6. Who defines the quality vs performance trade-off?

We mentioned multiple times that the trade-off in quality vs performance is mainly driven by the user, so we should specify who the user is when talking about Foveated rendering. The way this technique is integrated in the platform guides who is responsible for deciding the quality vs performance trade-off. The first possibility is to let the person enjoying the VR experience be the one specifying this trade-off. This means the application will allow to setup a quality ratio in their options and leave the players decide what is the best for them. The drawback of this is that the performance of the application is in control of the player who doesn't know about the performance requirements of the application and can choose a non-optimal setting, so a good range of ratios needs to be provided. The other possibility is to leave the application/engine decide what is the best ratio to use. This is a smart choice since the application developer has more insight about the performance requirements of their application and can decide the best reduction value. On the other side, this method will require the application developer to test the application with multiple users and system specification to find the best values for all the combinations. The last option is to let the VR compositor subsystem decide which ratio do to use. The VR subsystem can also change dynamically this value based on metrics such as overall GPU load, CPU load, temperature, etc. Since the VR compositor is also used by all the VR application that run on the platform, its provider will have the best set of application use cases to define a good range of values for the ratio that work for all the possible use-cases.

3.6.1 Improving quality: Blending inset/outset

Human eyes are sensitive to sudden variation and a change in resolution between two areas of the field of view will be clearly noticeable. To reduce this variation, we applied a blending step during the composition. The fovea and periphery images will be blended together based on the distance from the center of vision. This will require the periphery to also contain some overlapping area with the fovea view. In CircuitVR we simply used a smoothstep function, but is only executed when we are approaching the edge of the fovea image. The result of the smoothstep function is used to linearly interpolate between the value of a pixel in the fovea image and the corresponding pixel in the periphery image. In our demo, we used ~100 pixels width for linearly interpolate the two images.

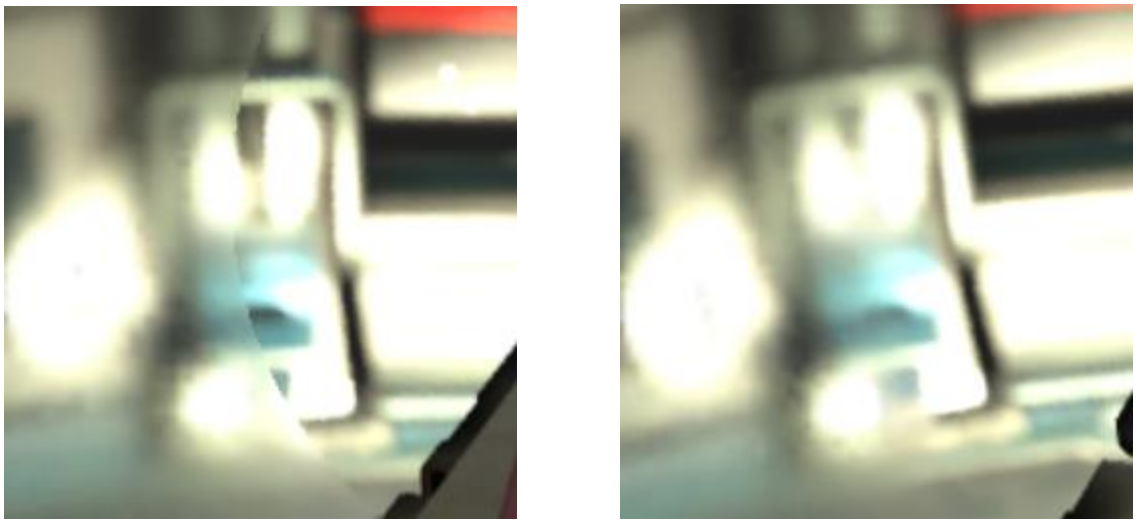


Figure 18: Comparison between composition with no blending (left) and with smoothstep blending (right).

3.6.2 Improving quality: Blurring outset/decrease contrast

Reducing the resolution of the scene framebuffer doesn't produce any quality degradation for the fovea image but reduces the amount of fine details in the periphery. This reduction can be quite noticeable since combining the image will have a blur effect in the periphery area when upscaling to the screen framebuffer resolution. In order to improve perception quality, we need to preserve the contrast while upscaling the periphery. This will require applying filters to the periphery image before composition to achieve a better picture.

3.7. The importance of MSAA

Multi-sampling Anti-Aliasing (MSAA) plays an important role in the quality of the final image. Mali GPUs have the ability to perform 4xMSAA, with very little cost in performance (<= 10%) but noticeable improvement in quality, especially for VR applications where the lenses have a magnifying effect on the pixels and edge aliasing is more visible. This is even more the case when using Foveated Rendering since reducing the framebuffer resolution will inevitably introduce aliasing problems. During our experiments developing CircuitVR we also tested using 8xMSAA. The cost for enabling it wasn't high since we were using 366x366 framebuffers and enabling 8xMSAA wouldn't have stressed the GPU so much to make it the bottleneck for the application but, the quality improvements were not worth the cost.

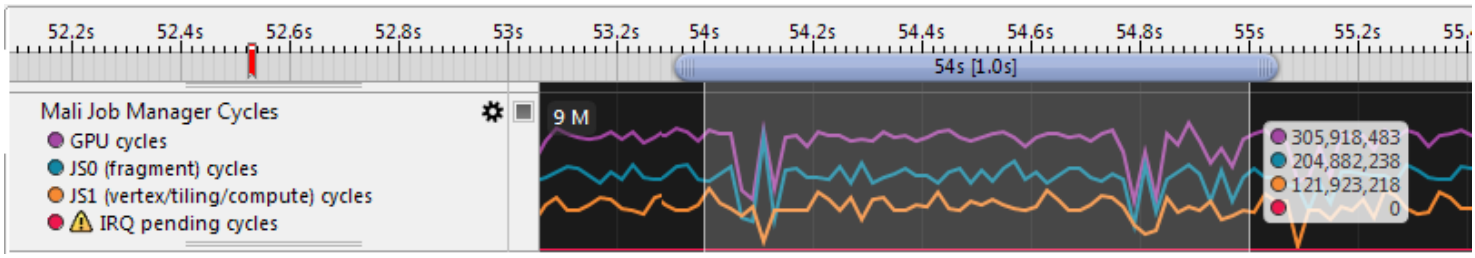


Figure 19: Arm Streamline counters for a scene rendered using Foveated Rendering and 4xMSAA

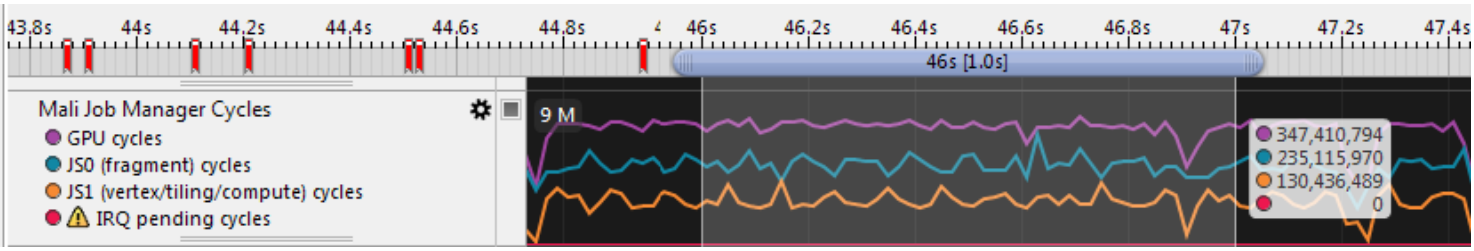


Figure 20: Arm Streamline counters for a scene rendered using Foveated Rendering and 8xMSAA

As can be seen in the pictures, the difference on the overall GPU cycles variation when moving from 4x and 8x MSAA were quite small (347Mcycles for 8xMSAA vs 305Mcycles for 4xMSAA. That is ~13% increase compared to 4xMSAA).

3.8. Power and performance statistics

This specific implementation of Variable-resolution shading and Foveated rendering require a correct balance of the various components of the system (CPU, vertex load and fragment load) but the advantages that we saw after striking a balance were positive. As mentioned at the beginning of the whitepaper using the Multiview extensions to implement this algorithm will move the load from fragment processing to vertex processing. High quality graphics does not mean extra vertices, however, applications that are vertex bound will not see any improvement from the algorithm but even a possible degradation in performance.

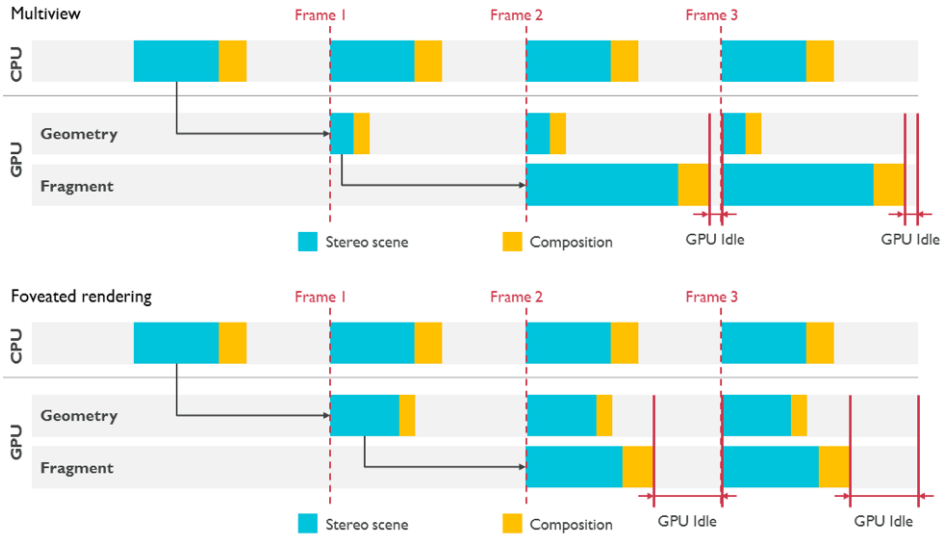


Figure 21: Vertex/Fragment load shift diagram

It is important to keep moderate number of total vertices per frame (CircuitVR demo: ~55K triangles). Below you can find the hardware counters information we acquired using the Arm Streamline profiler with a normal Stereo Rendering version of our demo (using only Multiview) and a Foveated Rendering implementation with framebuffers sizes reduced to 35% compared to the classic VR.

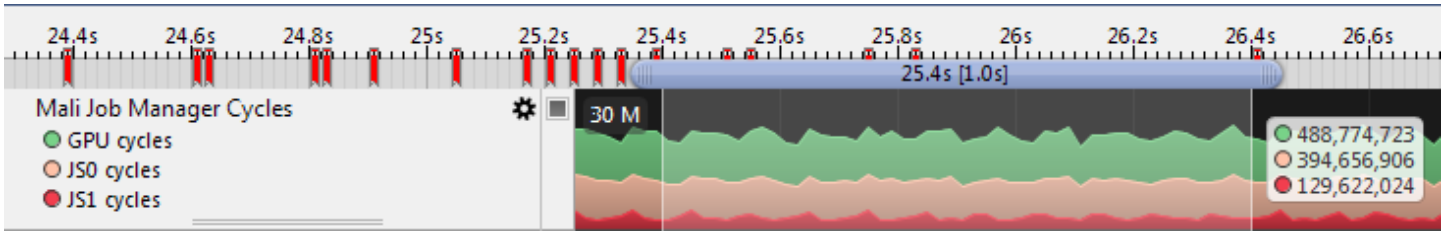


Figure 22: Stereo Rendering using Multiview

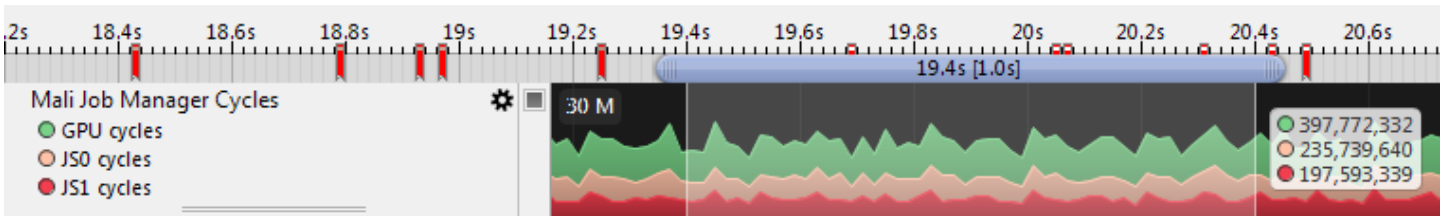


Figure 23: Foveated Rendering - 35% framebuffer sizes

The counters have the following meaning:

- GPU cycles: cycles spent in any work (Vertex/Tiler/Fragment)
- JS0 cycles: cycles spent for the Fragment processing
- JS1 cycles: cycles spent for the Vertex/Tiler processing

Beware that the values don't sum up since Mali GPUs are tile-based renderers and can parallelize vertex and fragment work.

The measurements refer to 1 second of runtime for the same scene which runs at 60FPS. We used 1s as interval so that it was easy to calculate what frequency is needed for the GPU to execute its work. For example in the Fig. 11 the GPU cycles are 488M and that means we would need a frequency of 488 MHz in this particular GPU configuration (Mali-T880 MP12) and scene to achieve 60FPS. If we analyze the numbers for the Foveated Rendering case (Fig. 12) we can see that we would need a frequency of 397 MHz to achieve 60FPS. As expected, the fragment load decreases when using Foveated Rendering from 394Mcycles for classic VR to 235Mcycles and as reported in the introduction of the section the vertex load increases from 129Mcycles to 197Mcycles.

The overall GPU load reduction was around 20% and the fragment load reduction was ~40%. The vertex load increased by ~50% (which is expected since we are rendering 2 more views) but the overall contribution is still less than the savings. Furthermore, improved techniques are under development to reduce if not eliminate completely the need for rendering the scene multiple times.

3.9. O.S. and app/middleware integration

Due to the peculiarity of the algorithm, various changes are required in the rendering pipeline of the utilized engine. An important part that needs to be considered is the complexity of the shaders used, especially the vertex shaders. Due to the fact that we need to run vertex shaders 4 times, the code needs to be optimized to allow the compiler to properly remove view-independent parts and execute them just ones. This is made more difficult if the engine uses some translation tool to convert the code from one language to the other. Our suggestion is to use the gl_ViewID_OVR as an index into an array of view-dependent data. We provide a complete tutorial with code and documentation inside our OpenGL ES SDK for Android repository available at **https://github.com/ARM-software/opengl-es-sdk-for-android**.

HMD middleware can improve the performance of the application by exposing a way to run Foveated Rendering without the need of the final composition pass. They can hide the composition stage inside the VR compositor process. Some of them already expose a way to handle the Stereo Rendering case with Multiview and some exposed functionality can already be used to implement Foveated Rendering.

There are no specific requirements on an OS to support Foveated Rendering. The platform only needs to support a version of Android which makes use of OpenGL ES 3.0 to expose the Multiview extensions to the users. For Mali, we also suggest to use the latest DDK version.

4 Conclusions

In this whitepaper, we covered all the steps needed to implement a fully functioning Foveated Rendering system. We showed how the Multiview extensions can enable this type of algorithms and how can it be used efficiently with the stencil mask optimization techniques. The algorithm is not a silver bullet for all type of application bottlenecks but it can give nice a performance boost for the applications that meet its requirements. Foveated rendering is also an important topic for future development. Various techniques are under investigation to support other algorithms that exploit the physiology of the eye. These algorithms require changes in the hardware and they are still under investigation for the quality, applicability and performance prospective and a common solution is still undefined. It is clear though that the requirements of VR applications are growing, and with that comes new techniques to efficiently support them.