OpenGL Foveated Rendering

Gustavo Silvera
gsilvera@andrew.cmu.edu
github.com/gustavosilvera/gl-fovrender
Carnegie Mellon University, Pittsburgh, PA, USA

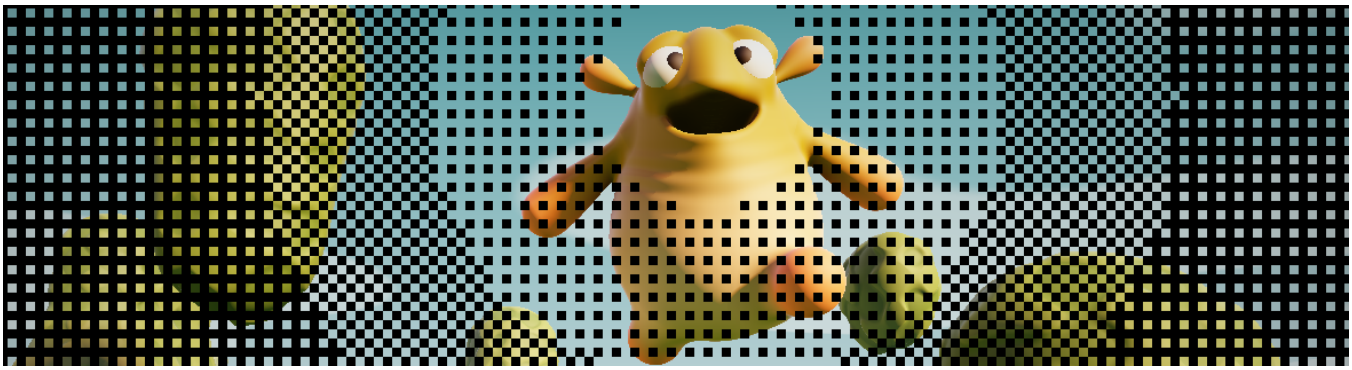

Figure 1: Foveated rendering on [15] with reconstruction shader effect disabled.

ABSTRACT

Human vision is naturally focused. Our vision system allows for very high fidelity detail in small regions of our sight based on our foveal focus region. This is especially noticeable as you might notice increased blurriness and loss of detail further in your peripheral vision. For the most part, our brains are highly performant at processing the raw signals from our eyes to create a complete picture and fill-in the missing/lower-fidelity detail, making it often unnoticeable to us. This project aims to leverage this natural phenomenon for rendering performance by only focusing detail in a small (focus) region and gradually decreasing the quality/resolution as the angle from the center increases. This technique is known as foveated rendering, where the eye gaze is used as a signal for where the central detail region is. This project will be implementing a simple renderer in C++, OpenGL [8] that includes mouse-cursor based foveated rendering and allows for performance comparison with native rendering. This paper will first describe the methodology and implementation, then demonstrate some results, then finally discuss performance implications and conclude.

KEYWORDS

Graphics, Rendering, Eye tracking, OpenGL

VCS Reference Format:
Gustavo Silvera. 2022. OpenGL Foveated Rendering. In *Special Topics: Visual Computing Systems, Spring semester 2022, Carnegie Mellon University, Pittsburgh, PA.* , 8 pages.

VCS, Spring 2022, Pittsburgh, PA, USA

1 INTRODUCTION

The crux of foveated rendering comes from the natural phenomena that human visual system follows. In our visual system (eyes), the vast majority of our photoreceptor cells (cones) are concentrated in the fovea centralis [12]. This quantity quickly drops as the angular separation from the fovea increases as demonstrated in Figure 2. When we humans look around our brain does a significant amount of processing to the image to fill in missing information [12]. By leveraging this automatic biological behaviour, foveated rendering could theoretically improve performance without a perceptible cost to visual quality because the shading rate would follow our natural vision mechanisms. There have been several initial investigations into foveated rendering, primarily in a VR (virtual reality) context. For this project I tried to implement work from [2] but was unsuccessful in modifying the Unreal Engine [7] to work for my purposes.

Mask-based foveated rendering (MBFR) [18] is a variable rate shading technique for rendering that gradually decreases shading rate of peripheral regions by dropping groups of pixels based on a checkerboard pattern and reconstructing them with a fast post-processing pass. This can improve performance when the fragment (pixel) shader is very expensive by greatly reducing the number of pixels/fragment necessary to rasterize. This leads to a overall experience improvement when the cheaper fragment shader pass is worth the cost of performing the reconstruction and visual artifacts.

This project entails a hardware accelerated OpenGL [8] renderer written from scratch that is designed to compare foveated rendering to native rendering on a flat 2D fragment shader. Due to a lack of a low-latency and reliable eye tracker for this project, I simplified the problem to use mouse cursor screen coordinates as my "eye gaze point on screen". However, this work can be naturally extended by using a consumer grade eye tracker device with a programmable API. Furthermore, while this project is mostly a proof-of-concept

to evaluate the performance of foveated rendering, the implementation can be extended for virtual reality applications with built-in eye trackers such as the HTC Vive Pro Eye [9], upcoming VR headsets with eye tracking [6], and VR eye tracking attachments [11].

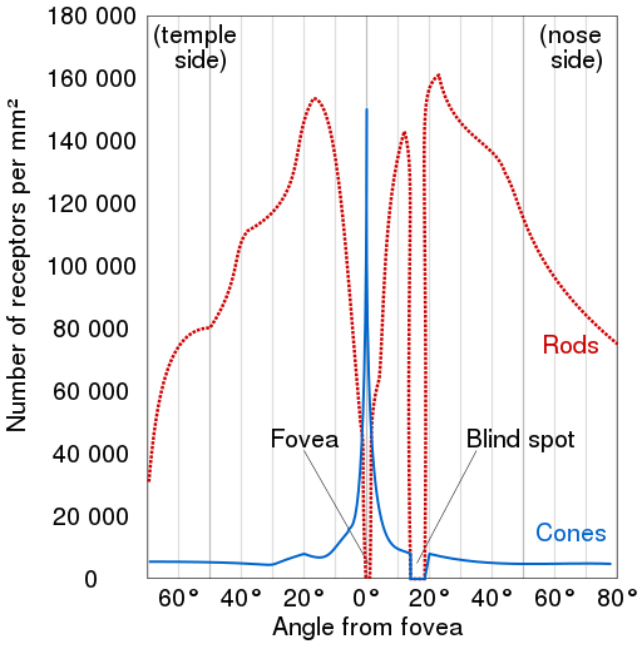

Figure 2: Distribution of photoreceptors in human eye relative to angle from fovea. [Public domain], via Wikimedia Commons. (https://en.wikipedia.org/wiki/File:Human_photoreceptor_distribution.svg).

2 METHODOLOGY

The process for this project was to first implement a simple OpenGL renderer from scratch, then add the foveated-rendering implementation which includes pixel dropping and pixel reconstruction, then finally perform quantifiable performance analysis.

First implementing the OpenGL renderer was primarily through following this tutorial [4] which describes how to get started with OpenGL and C++ on an M1 Mac machine. This works great for my case since my primary development machine is a 10 core M1 Max arm64 laptop.

With the baseline renderer done and ability to update shaders in runtime present, I gathered several fragment shaders from the popular web-based shader host ShaderToy [16] for the underlying "expensive" fragment shader computations.

In getting started with the actual foveated rendering implementation, there were several interesting parameters I needed to consider when designing the system. The most obvious parameters for foveated rendering is how many and how large should the regions defining variable shading rate be? For the "how many" case, this is simply 4 because the MBFR [18] technique allows up to four discrete variable rate shading settings.

2.1 Foveal Regions

For the "how large" question of the foveal regions, this primarily comes down to user preference and content. There is a relationship between visual quality and performance with these settings since shrinking the higher quality regions leaves more room for low quality regions, increasing visual artifacts and decreasing quality but increasing performance. The standard (non-foveated rendering) technique is a case where the parameter for the native-quality encompasses the entire display as the entire framebuffer is rendered at native quality. For my case, I found that static thresholds of 10%, 25%, and 40% of the diagonal length of the framebuffer were pleasant region sizes that struck an acceptable balance between performance and quality.

2.2 Block Size

Another very important parameter for the renderer is the number of pixels that are being dropped/reconstructed as a "block" at a time. In the original implementation [18] the block sizes were fixed at 2×2 quads which made up a total of $4 \times 4 = 16$ pixels. I found more success with increasing this number on my high-dpi displays, and this required some additional engineering to extend the algorithms for > 2 pixel correspondences (especially for interpolation). I experimented with dropping $n \times n$ pixels for $n \in \{2, 4, 8, 16, 32, 64, 128, 256\}$ and found that a block size (n) of 16 was optimal for my 2560×1440 display and viewing experience. My implementation enables simple and fast runtime block size updates, such as increasing and decreasing the block size n during runtime to see and compare immediate results. This is useful for debugging and tuning a specific parameter setting for one's visual enjoyment and display configuration. It is also important to note that this can likely be extended to be content aware, so increased shading (smaller block size n) can occur where high frequency content is present, and low frequency data can use a larger block size.

2.3 Pixel Dropping

The main performance gains from foveated rendering come from decreasing the amount of work for the fragment (pixel) shader by dropping fragments that would otherwise need to be rendered. In cases where the fragment shader is especially expensive, such as most VR applications, this can provide considerable performance improvements. However, if the fragment shader is not a bottleneck then foveated rendering likely has a zero or negative performance effect.

As described earlier, the MBFR [18] technique defined four distinct quality settings for their checkerboard pattern. Specifically, the central (full-quality) region retains full resolution by not dropping any quads; the high-quality region drops one in four quads: the top-right quad; the medium-quality region drops two in four quads: the top-right and the bottom left; and the low-quality region drops three in four quads: the top right, bottom left, and bottom right. This is visually depicted in Fig. 3.

Implementation wise, this is handled by a new shader I call the "dropping" shader that is designed to drop quads if the pixel coordinate input `gl_FragCoord.xy - 0.5` lies within a quad that should get dropped depending on the region. This is implemented as follows:

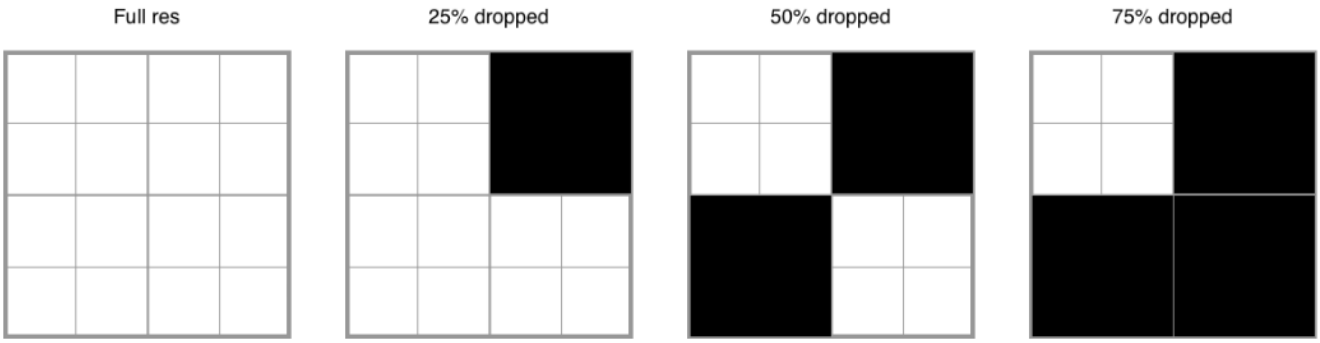

Figure 3: Distribution of pixel dropping (block size $n = 2$) according to the quality levels. Image from [18].

```
int n = 16;     // size of quad
int s = 2 * n;  // side length of 2x2 quad block
vec4 clear = vec4(0, 0, 0, 1); // black

vec2 coord = gl_FragCoord.xy - 0.5; // top left pixel corner
// determine which quad this fragment is
float xmod = mod(coord.x, stride);
float ymod = mod(coord.y, stride);

// compute (boxy) distance to foveal region (low freq)
vec2 boxy_coord = floor(coord / s) * s;
vec2 center = floor(vec2(focus) / s) * s;
// internally computes dist squared for performance
float d = dist(boxy_coord - center);

if (xmod < quad && ymod < quad) { // top left
    fragColor = expensive_main(); // always render
}
else if (xmod < quad && ymod >= quad) { // top right
   fragColor = (d > thresh1) ? clear : expensive_main();
}
else if (xmod >= quad && ymod < quad) { // bottom left
   fragColor = (d > thresh2) ? clear : expensive_main();
}
else { // bottom right
   fragColor = (d > thresh3) ? clear : expensive_main();
}
```

This block of code describes the branching behaviour used to model the checkerboard pattern from Fig. 3 and effectively filters calls to `expensive_main()` which is the original fragment shader that is used for the visual content. I will continue to discuss the effect of branching on performance in the performance section below.

Notice that there is a special value `boxy_coord` that is used to ensure the distance computations is done in a low-frequency domain. This ensures that all pixels are assigned a block that is aligned to the standard coordinate system and transitions between regions is seamless.

2.4 Reconstruction

Once the various pixels are dropped from the image according to Fig. 3, the resulting image is left with many gaps in content as seen in Fig. 1. The second part of the foveated rendering algorithm is to undergo a postprocessing pass that efficiently cleans up the gaps in the image. This is a postprocessing pass because this fragment shader runs after the foveated rendering fragment shader is completed, and uses the results from that shader directly. I implement this by rendering the first fragment shader to a `GL_TEXTURE_2D` which is then passed as an input to the reconstruction shader which either directly copies or interpolates pixels to the output framebuffer object.

As per the MBFR [18] implementation, there are two main approaches to reconstruction on the various quality regions. Clearly for the full resolution quads (none dropped) no reconstruction is necessary, so this region can be discarded in the shader. For the high-quality (drop 25%) and medium-quality (drop 50%) regions, since there are pixel quads above, below, left, and right of all the dropped quads, trilinear interpolation is possible and efficient. However, for the low-quality (drop 75%) region there are more dropped quads than not, which makes bilinear interpolation the only feasible option.

2.5 High/Medium Quality Reconstruction

For the high/medium quality regions, there are at least as many dropped quads as there are rendered quads. This allows reconstruction to occur with the the four cardinal directional colours weighted by their distance to a fragment.

As demonstrated in Fig. 4 there are always available pixels above, below, left, and right of any dropped quad (not counting edge cases where bilinear interpolation is used) and since my blocks are of size n, the following trilinear interpolation scheme is used:

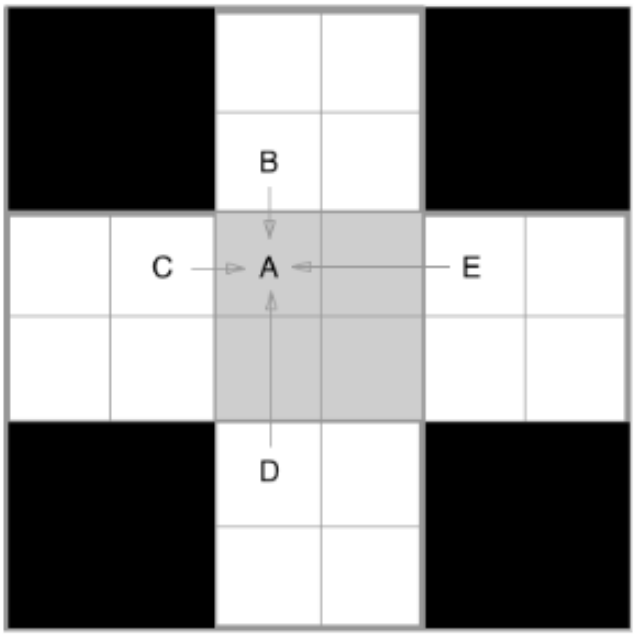

Figure 4: Trilinear reconstruction of pixels in high/medium quality regions. Image from [18].

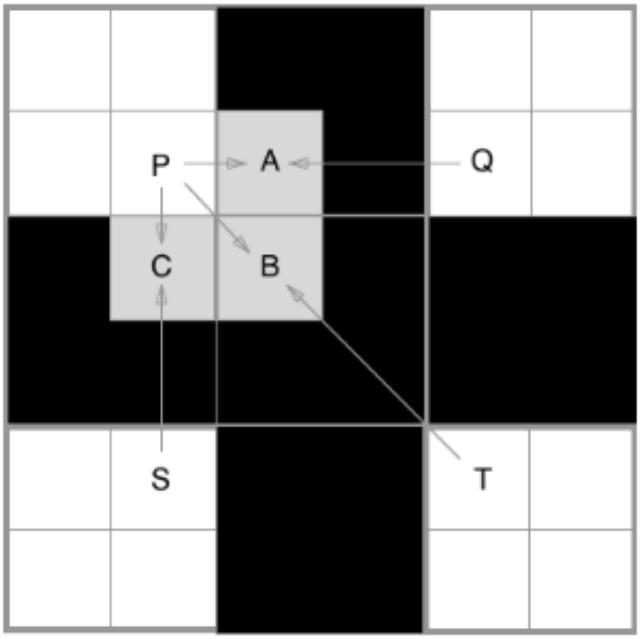

Figure 5: Bilinear reconstruction of pixels in low quality region. Image from [18].

Suppose a scheme similar to Fig. 4 but with the block side lengths of size n instead of 2

let $n = 16$ (though this can really be any power of 2)
Goal is to fill in pixel A
let B be the closest rendered pixel in the same column above A
let C be the closest rendered pixel in the same row left of A
let D be the closest rendered pixel in the same column below A
let E be the closest rendered pixel in the same row right of A
let $w_x \in (0, 1) = ||AC||/n$ (horizontal spatial weight for C)
let $w_x \in (0, 1) = ||AB||/n$ (vertical spatial weight for B)
let $h = w_x \cdot C + (1 - w_x) \cdot E$ (horizontal interpolation)
let $v = w_y \cdot B + (1 - w_y) \cdot D$ (vertical interpolation)
$A = \frac{1}{2}(h + v)$

Notice that this requires four `texelFetch` operations (to obtain the `vec4`'s for $B, C, D, \&E$) which can be costly if memory bandwidth/latency is a bottleneck. For my case since the M1 is a SoC with unified CPU/GPU memory, so memory accesses are quite fast.

For edge cases, such as on the boundary between the medium and low quality regions, or with the framebuffer edges, the algorithm is changed to bilinear interpolation for pixels that are coloured in to not reach out of bounds.

2.6 Low Quality Reconstruction

For the low quality region, there are more dropped quads than there are rendered quads, this makes it difficult to perform high quality interpolation since there is considerably less overall data and more guesswork needed. As demonstrated in Fig. 5 the typical case is to perform simply bilinear interpolation along either the vertical or horizontal axis corresponding to the closest pixels in a quad. Additionally, this method includes bilinear interpolation along the diagonal axis (specifically the NW-SE axis such as between $P, B, \&T$) but I implemented trilinear interpolation by additionally using the NE-SW axis to further improve the interpolation results for the bottom right quad.

The interpolation logic is very similar to the trilinear case in *Section 2.5* but with added casework for selecting bilinear (top right quad or bottom left quad) interpolation along a particular axis or diagonal (bottom right quad) interpolation along both.

3 RESULTS

The results of this foveated render implementation show promise in striking a balance between improved performance and decreased perceptible visual quality. I found that the block size (n) and the type of content was especially important in determining this balance, and it depends on several factors.

Some of the best results from this new rendering technique were demonstrated with shaders such as [15] (Fig. 9) and [3] (Fig. 10) since the primary focus of the scene is in a small region in the center and the rest of the background generally has a lower frequency and can be approximated well with the linear interpolation from the reconstruction shader.

This is especially true when the shaders are in motion, as I found that finding perceptible differences was very difficult for these cases.

However, this approach struggles when there is high-frequency detail throughout the entire image, especially as seen in [1] (Fig. 11), [10] (Fig. 12) and [14] (Fig. 15) where there are strong lines that are being very clearly blurred and significant detail is lost in the periphery.

Additionally, when some shaders are in motion, the peripheral region seems to show some flickering when the reconstruction is continuously updating patches based on high frequency data. This is especially noticeable in [17] (Fig. 13) and [5] (Fig. 14) since their peripiheral regions have lots of high detail that often drastically changes the linear interpolation results per pixel.

4 PERFORMANCE

For all the following performance graphs, the color key is as follows: (leftmost) Blue is [3] (Fig. 10), Red is [15] (Fig. 9), Yellow is [17] (Fig. 13), Green is [5] (Fig. 14), Orange is [1] (Fig. 11), Teal is [10] (Fig. 12), Light blue is [14] (Fig. 15), and finally (rightmost) Salmon is [13]. The measurements were taken on an M1 MAX 24 core GPU with a 1280×720 framebuffer resolution.

4.1 Fragment Shader Runtime

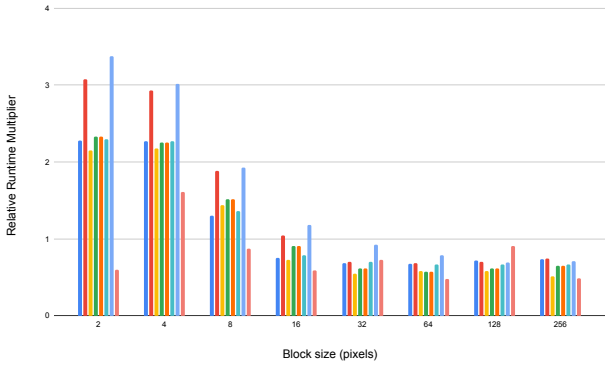

Figure 6: Relative runtime multiplier of fragment shader (lower is better).

As seen from Fig. 6, comparing the relative runtimes for the fragment shader show that a smaller block size ($n < 16$) often leads to significantly worse performance ($2 - 3x$ longer runtime) until $n = 16$ where the relative runtime is < 1 which means the performance actually shows an improvement. However, there is also clearly diminishing returns as the block size increases past > 16 but with significantly worse visual quality and increased artifacts, so the optimal point is likely at $n = 16$.

4.2 Reconstruction Shader Runtime

As seen from Fig. 7, comparing the runtimes of the postprocessing reconstruction shader show that the overall runtime for all levels is very low ($< 0.0006s$) and does not scale much along block size. This is good because it shows that the reconstruction shader runtime is consistently significantly lower than the fragment shader on all levels, indicating that the postprocessing effect should have very little effect on overall performance.

4.3 Overall Speedup

Finally, by looking at Fig. 8 we can see that the overall performance (relative framerate) of the foveated rendering approach follows

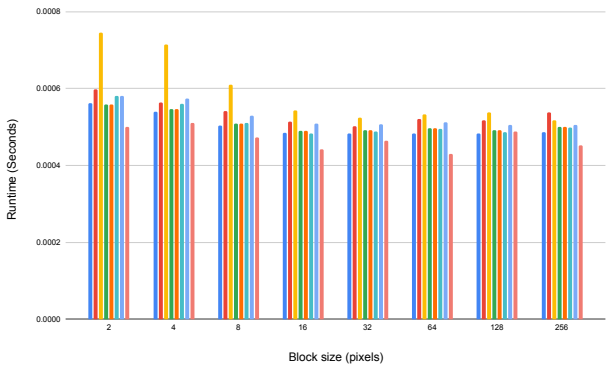

Figure 7: Runtime of reconstruction shader (lower is better).

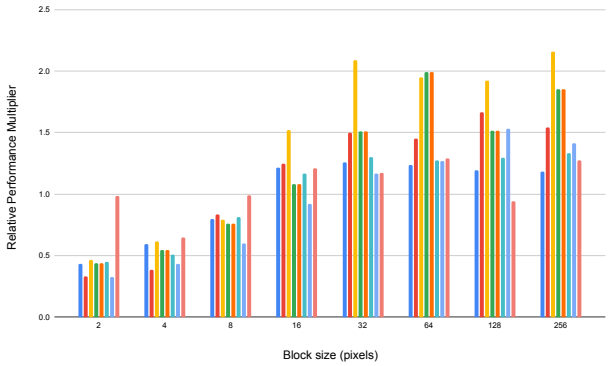

Figure 8: Relative performance multiplier with various block sizes (higher is better).

these findings as well. Since the x axis demonstrates block size, which directly affects visual quality (left/smaller blocks lead to fewer artifacts, right/larger blocks lead to more artifacts), it is best to find the leftmost block size that still improves performance. This is found at $n = 16$ since the performance for nearly all the shaders is improved by a 20% on average. Importantly, the performance improvement after this point ($n > 16$) is not as significant and varies more per shaders. This is likely because the different shaders have different costs in terms of where in the framebuffer is the expensive region, how much gains are to be made in the periphery, the tiled nature of the GPU processing, and the bottleneck computations of the shader itself.

Since the OpenGL spec uses workgroups to compute various fragments in batches on the GPU, this explains why having a low block size $n < 16$ often leads to overall worse performance. Since there is a lot of branching in the dropping/reconstruction shader, with a low-enough block size there is probably increased branching within a workgroup itself, so it is still performing most of the work it would have if it weren't branching. Since the branching adds a cost by reducing pipelining efficiency, this likely leads to performance degradation. However, once the n gets large enough, it is likely

the case that the branching is leading to entire workgroups being dropped and overall significantly reduced computation.

5 CONCLUSION

In conclusion, the technology shows promise! While it introduces more parameters for visual fidelity vs performance tradeoff, there are considerable gains to be had when used properly. However this rendering technique relies on several features such as real-time eye tracking in order to be imperceptible to the user. Overall, the project was a success in demonstrating the performance implications of foveated rendering in an OpenGL, flat-screen desktop context.

It is important to note that while this technology has promise, it also makes several assumptions that normal rendering wouldn't. For instance, foveated rendering assumes there is only a single person viewing some content at any time, which might not be true in a group session such as for couch gaming. Additionally, since this technique lightens the load for the fragment shader at the cost of a postprocessing effect, applications with a very light fragment shader and/or a heavy postprocessing pass would likely worsen both quality and performance, hence worsening the overall viewing experience.

Virtual reality is likely the best case scenario for this kind of technology, since VR applications often have expensive fragment shaders, little to no postprocessing, potential for cheap on-device eye tracking, and only a single viewer of the content at a time.

6 FUTURE WORK

Ideally this work can be extended to work with proper eye gaze signals. It is likely possible to incorporate a high-fidelity low-latency robust eye tracker such as Tobii desktop trackers for this kind of task. Or theoretically, if a robust and fast enough webcam-based approach is possible, then I'd be very interested in experimenting with that.

It would also be great to see this effect extended to VR, where the gains are likely best appreciated.

It would also be interesting to test this application on various GPUs, since the M1 Max 24 core GPU is the only GPU this has been tested on so far. I would like to see how GPUs with non-uniform dedicated video memory might fare with the postprocessing and specifically `texelFetch` overhead.

ACKNOWLEDGMENTS

To Oscar, for a very fun and insightful course on rendering and imaging!

REFERENCES

[1] Bers. 2016. Geomechanical. https://www.shadertoy.com/view/MdcXzn
[2] Marios Bikos. 2020. Getting started with VRS Foveated Rendering using HTC Vive Pro Eye Unreal Engine. https://mariosbikos.com/vive-unreal-foveated-rendering/
[3] bradjamesgrant. 2020. Fractal Pyramid. https://www.shadertoy.com/view/tsXBzS
[4] Antonin Carette. 2022. The case of OpenGL, in C++, on m1 mac. https://carette.xyz/posts/opengl_and_cpp_on_m1_mac/
[5] EvilRyu. 2014. Mandelbulb. https://www.shadertoy.com/view/MdXSWn
[6] Jamie Feltham. 2022. Project Cambria: Everything We Know About Meta's Next Headset. https://uploadvr.com/project-cambria-everything-we-know/
[7] Epic Games. 2022. Unreal Engine. https://www.unrealengine.com/en-US
[8] Khronos Group. 1992. OpenGL - The Industry's Foundation for High Performance Graphics. https://www.opengl.org/
[9] HTC. 2019. HTC Vive Pro Eye. https://www.vive.com/us/product/vive-pro-eye/overview/
[10] Kali. 2013. Fractal Land. https://www.shadertoy.com/view/XsBXWt
[11] Pupil Labs. 2014. Binocular Add-on/Mount. https://pupil-labs.com/products/vr-ar/
[12] Chris L. E. Paffen Anil K. Seth Ryota Kanai Marte Otten, Yair Pinto. 2017. The Uniformity Illusion: Central Stimuli Can Determine Peripheral Perception. https://doi.org/10.1177/0956797616672270
[13] Naxius. 2019. simple 2d animation. https://www.shadertoy.com/view/WdsGRl
[14] Inigo Quilez. 2013. Raymarching - Primitives. https://www.shadertoy.com/view/Xds3zN
[15] Inigo Quilez. 2019. Happy Jumping. https://www.shadertoy.com/view/3lsSzf
[16] Inigo Quilez and Pol Jeremias. 2013. Shadertoy. https://www.shadertoy.com/
[17] TDM. 2014. Seascape. https://www.shadertoy.com/view/Ms2SD1
[18] Xiang Wei. 2018. Tech Note: Mask-based Foveated Rendering with Unreal Engine 4. https://developer.oculus.com/blog/tech-note-mask-based-foveated-rendering-with-unreal-engine-4-/

Figure 9: Same scene as Figure 1 (Shader [15]) with reconstruction shader enabled. Focus on the creature's face.

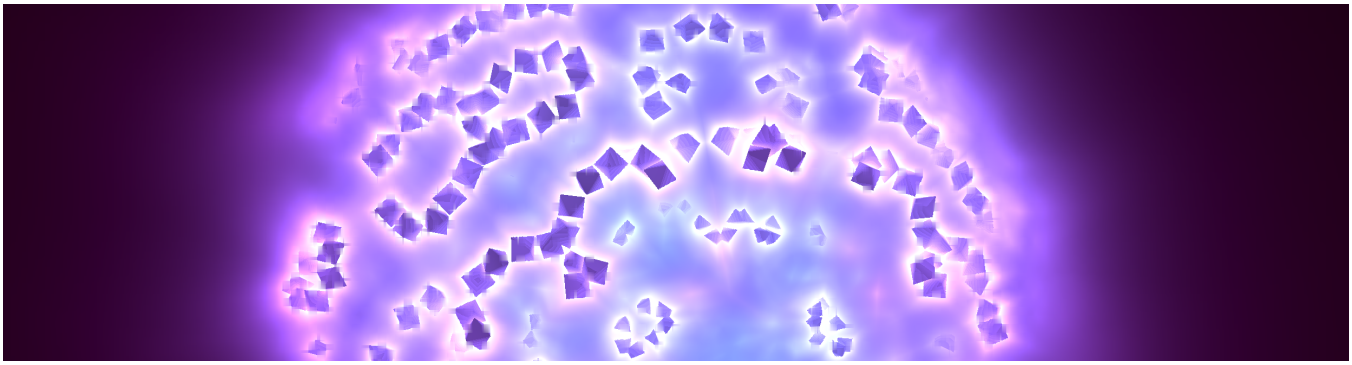

Figure 10: Foveated rendering result of shader [3] rendered with focus near the center.

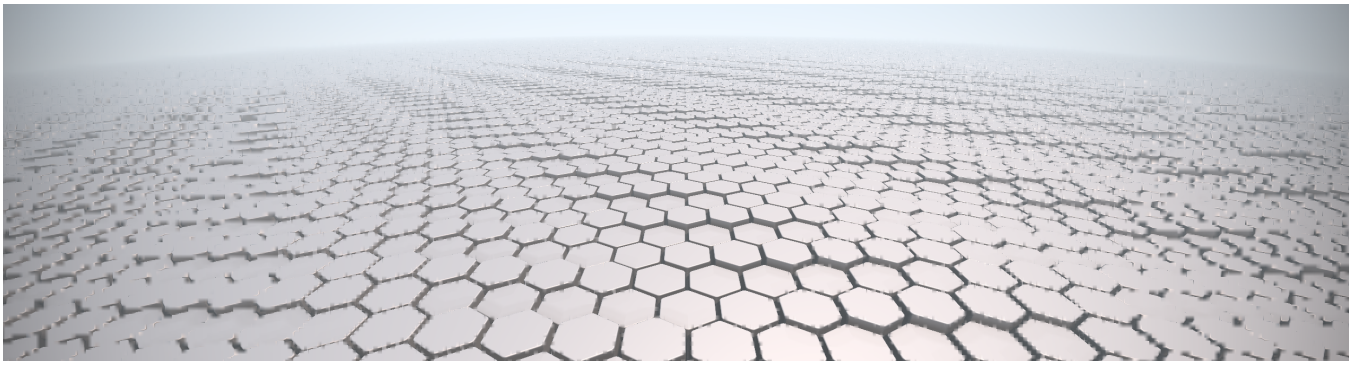

Figure 11: Foveated rendering result of shader [1] rendered with focus near the center.

Figure 12: Foveated rendering result of shader [10] rendered with focus near the center.

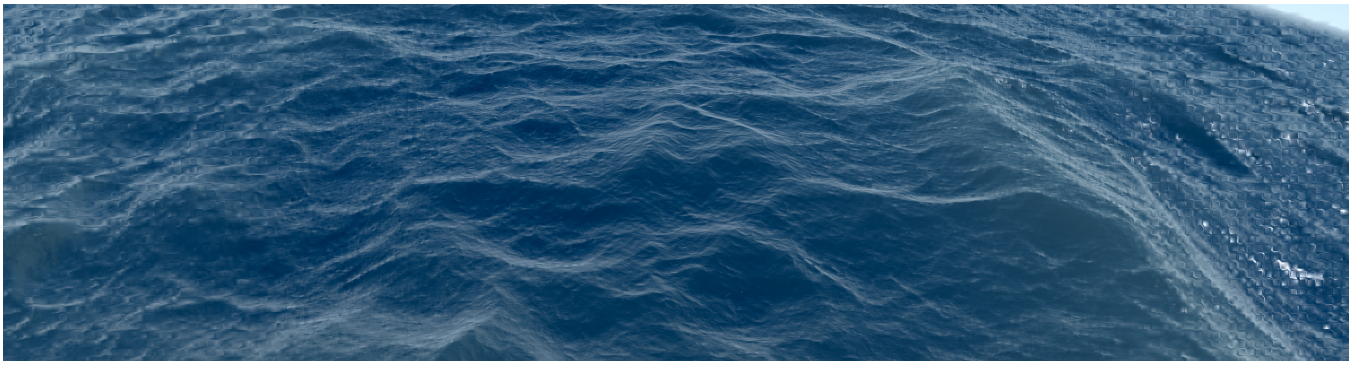

Figure 13: Foveated rendering result of shader [17] rendered with focus near the center.

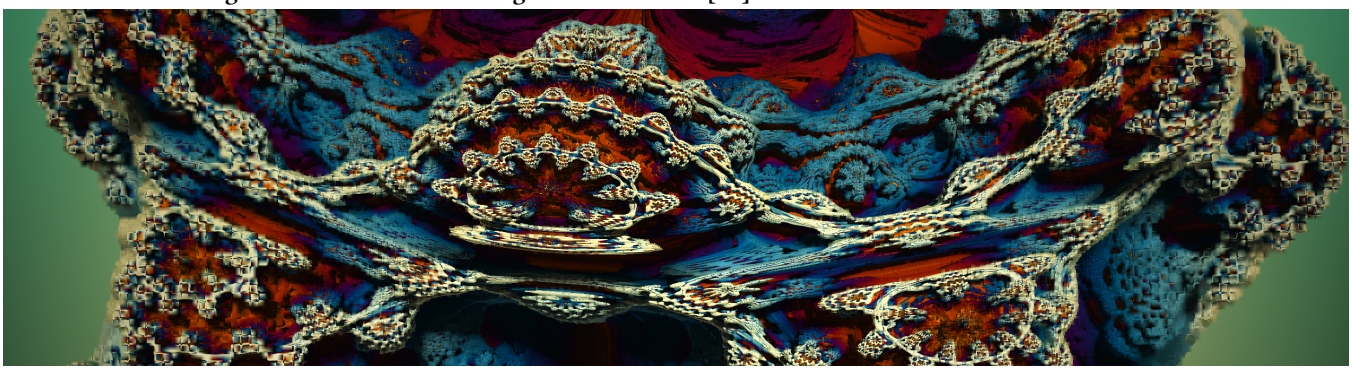

Figure 14: Foveated rendering result of shader [5] rendered with focus near the center.

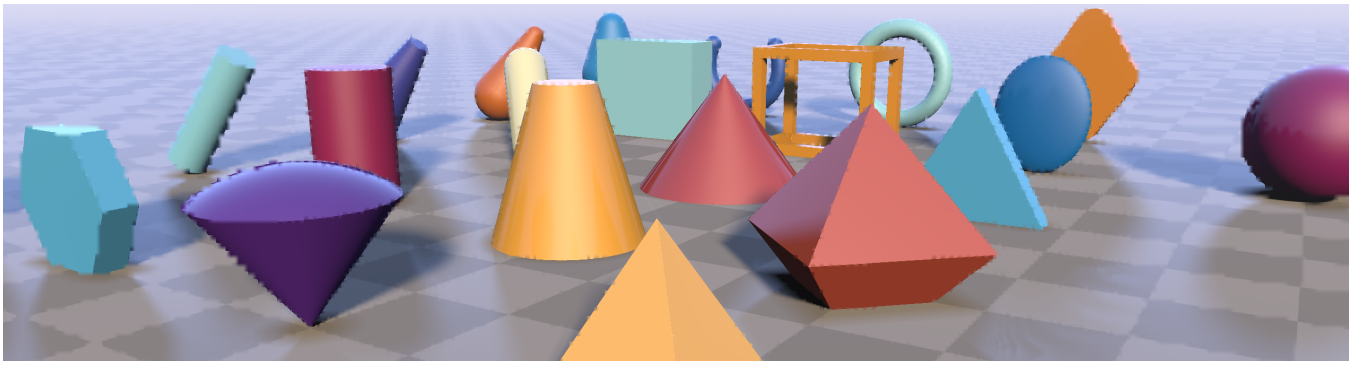

Figure 15: Foveated rendering result of shader [14] rendered with focus near the center.