METALSTIGMATISM: VR Metal Compute Path Tracer Lens Simulation for Aberrated Vision

Brayton Lordianto
UC Berkeley
brayton.lordianto@berkeley.edu

Han Li
UC Berkeley
han.li@berkeley.edu

William Wu
UC Berkeley
wuyongxuan@berkeley.edu

Ricardo Blanco
UC Berkeley
rickyblanco333@berkeley.edu

Abstract

We present a GPU-based path tracing solution on the Apple Vision Pro using Metal. Our system extends a thin-lens ray generation model with per-ray dioptric variation derived from clinical eye parameters (SPH, CYL, AXIS). Aberrations are modeled by perturbing each ray's focal point based on angular lens phase, producing realistic defocus patterns.The system was evaluated against complex scenes derived from Homework 3-2 and extended to the Arnold render engine to demonstrate scalability and performance. This work offers insights into the implementation of path tracing solutions on VR hardware and demonstrates the potential for real-time analysis of vision aberrations.

ACM Reference Format:
Brayton Lordianto, Han Li, William Wu, and Ricardo Blanco. 2025. METALSTIGMATISM: VR Metal Compute Path Tracer Lens Simulation for Aberrated Vision. In . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Simulating vision aberrations such as myopia and astigmatism in immersive environments offers potential benefits for accessibility research, user empathy tools, and virtual optometric testing. Traditional vision simulation techniques often focus on screen-space post-processing, leading to a not fully immersive experience in understanding these ailments. We create simulators for these aberrations that draws from clinical models of refractive error and lens anatomy to produce convincing results, even across stereoscopic views.

Our main contributions to computer graphics are the following:

- An extension on top of CS 184/284A's path tracing project 3[1] that allows for visualizations of depth of field and aberration simulation of scenes from the project.
- An Arnold plugin which create static, high quality renders with aberrations which runs path tracing on the CPU.
- A real-time physically-based GPU path tracer in Metal which runs direct on the Apple Vision Pro which supports rendering USDZ 3D models, powered by compute shaders/kernels.
- An interactive menu interface on the Vision Pro app that enables real-time adjustment of aberration/eye parameters, allowing immediate visualization of how different optical conditions affect object perception.
- The first public real-time GPU path tracer with real-time aberration simulation running on a commercial AR/VR hardware.

[1] https://cs184.eecs.berkeley.edu/sp23/docs/proj3-2

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Environment Setup

2.1 Capability for Depth of Field

Our camera simulation supports thin-lens DOF by sampling an aperture disc and shifting ray origins appropriately. Aberration-specific focal offsets are introduced by modeling dioptric deviations in the lens.

2.2 CPU and GPU Path Tracing

2.2.1 HW3 and Arnold CPU Rendering. We first start with existing code from Homework 3's Path Tracer and Homework 3-2 of Spring 2023, which includes CPU ray generation, intersection, BSDFs, and extension support for camera Depth of Field (DoF). We built DoF and aberration simulation on top of the homework for proof of concept.

We then built on top of Autodesk's Arnold render engine by creating a plugin that creates a new custom camera on top of the engine, adding support for visually abberated simulations. Specifically, we are seeking high–quality, non-real-time renders.

2.2.2 Vision Pro (Metal) GPU Rendering. We then extended our results to real-time path tracing on Vision Pro's Metal-based GPU pipeline, setting up compute, vertex, and fragment render passes that support real-time physically-based rendering. Compute shaders were authored in Metal Shading Language. Our goal was to simulate abberated vision with high quality passthrough. Additionally, there are virtually no path tracers on the Vision Pro, and certainly none with implemented Depth of Field, and was interested in creating a Metal Path Tracer.

We started with a basic Swift application with a barebones Metal Renderer, which supports a rendered scene that shows a simple box mesh running a vertex and fragment shader. On top of this, we built support on the Renderer to support three compute shaders (also known as compute kernels in Metal), implemented real-time GPU

Figure 1: Normal Vision v.s. Astigmatism Abberation

path tracing in Metal on it, along with depth of field and aberration simulation.

3 Research and Implementation[2]

3.1 Literature Review

Our work is informed by Barsky et al.'s vision-realistic rendering, Nießner's real-time GPU-based simulation of human vision, and recent work on aberrated human vision rendering (Csoba, I., Kunkli, R. 2023). István Csoba & Roland Kunkli's work and Zilong Wang and Shuangjiu Xiao's methodology for thin lens approximation optical simulation were especially helpful in lens calculations.

Myopia and hyperopia are caused by the lens of an eye focusing light at a point not perfectly on the retina, which is characteristic of a normal depth of field simulation.

Astigmatism (see Figure 1[3]) is caused by having multiple focal points in an eye, determined by the primary and secondary meridians on the lens. Here, we assume a 90° angle between the two meridians for regular astigmatism. Each meridian has a certain focal power, the primary meridian having a strength `SPH + CYL` while the secondary meridian has a strength of `SPH`. The variables `SPH`, `CYL`, and `AXIS` are found on a typical glasses prescription and represent the eye power in diopters needed to correct standard vision. The result from this is a streaking light image and double blurring effect.

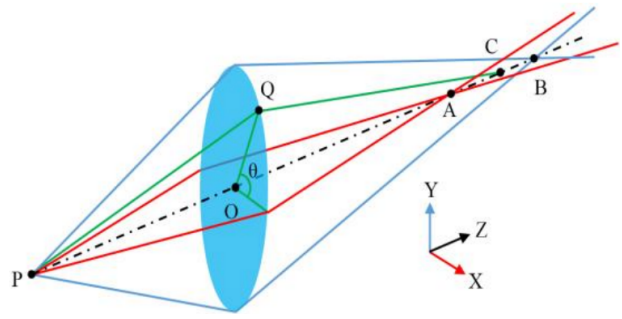


Figure 2: Optical path of astigmatism

[2]https://github.com/Cookei/184FinalProject_Submission
[3]https://eyecandys.com/blogs/news/how-astigmatism-affects-light

The starting angle offset from the horizontal axis is determined by the variable AXIS. For an angle principal to a meridian, we focus perfectly for that meridian's power. In a raytracing approach, this constitutes to having two focal points.

With a `CYL` value of 0, this represents myopic/hyperopic vision due to both meridians having the same eye power, leading to a singular focal point.

Standard depth of field sampling creates samples on the lens. A simple approach would be to uniformly randomly sample $r \in [0, 1]$ and $\phi \in [0, 2\pi]$ in polar coordinates. With the angle ϕ we can interpolate between the two eye powers associated with each meridian to determine the `eyePower`. It's important to note here that we use a sin squared weighted interpolation instead of a linear one. This gives us more realistic results that match a toric lens.

$$\text{eyePower} = \text{SPH} + \text{CYL} \cdot \sin^2(\phi + \text{AXIS})$$

$$f = \text{focalDistance} + \frac{1}{\text{eyePower}}$$

Here, given a target `focalDistance`, we can use our calculated `eyePower` as offsets, since they represent the power needed to correct normal vision. We convert our eyePower from diopters to meters and add that to the focalDistance.

With the new focal distance for the ray and can easily raytrace the scene with $\text{pLens\_cam} - \text{pFocus\_cam} \cdot f$

3.1.1 Weighted Sampling. In Zilong Wang and Shuangjiu Xiao's paper for simulating human eye optical systems, they note a more efficient sampling method that introduces angular and central biasing for sampling of the lens, leading to better sampling for astigmatism due to its inherent asymmetry. A summary of the sampling technique is noted here. We implemented both uniform and biased sampling to compare results.

$$\text{DistantIndex} \in [0, 1]$$
$$\text{AngleFactor} = \text{some constant}$$
$$\text{AngleFactor} \in [0, \text{AngleFactor}]$$
$$r = \text{DistantIndex}$$
$$\phi = \frac{2\pi}{\text{AngleFactor} \cdot (\text{AngleIndex} \cdot r)}$$

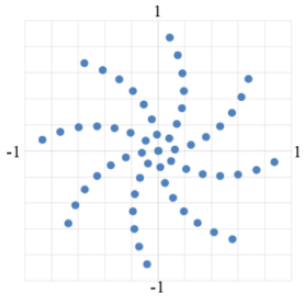


Figure 3: Biased sampling

3.2 CPU Pathtracing

3.2.1 Implementing on HW 3-2. To add astigmatic support for the HW 3-2 depth of field approach, we created a new sampling function based on the previous approach. This returns a sample in polar coordinates are that passed into `rndR` and `rndTheta` in the `generate_ray_for_thin_lens` function.

We calculate the new focal distance for the ray and then use that to calculate the new ray direction.

3.2.2 Implementing in Arnold. In order to see higher fidelity renders with support for Arnold shaders, custom meshes, etc, as well as speed up rendering times, we implement this rendering approach through a custom Arnold plugin.

Here, we create a custom camera node. Modifying the `create_camera_ray` function allows us to manipulate the ray from the camera. Porting over the code from Hw 3-2, we can calculate `x_sensor_cam = input.sx * tan(hFov * AI_DTOR * 0.5)` where `input.sx` is the sample point in normalized screen space. Here we note that although `hFov` and `vFov` is noted here, they actually represent the same Fov value. Having a non-uniform Fov skews the final render, unlike in Hw-3-2 where there were separate Fovs for each axis.

Since the Arnold camera outputs the ray, in camera space, we can get rid of the camera to world space matrix transforms and instead only operate in camera space.

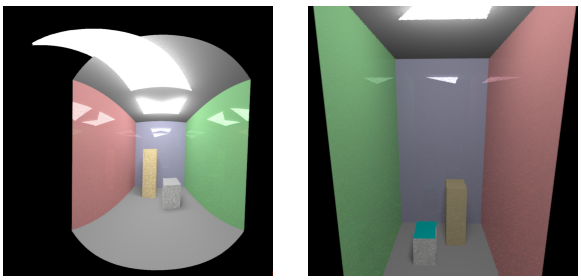
To give us more flexibility, modifying `node_parameters` allows us to add custom sliders to change various variables relevant to the final render. We can then compile to a `.dll` file and use them with Arnold. Here we did not implement support with a `.mtd` file for integration with Maya, but doing so would be simple. Instead we render directly with Arnold .ass files using kick. So we *kick ass!*

3.3 GPU Path Tracer - Architecture Overview

3.3.1 The Need for a Display Mesh - Sphere v.s. Plane. On the Vision Pro, third party developers are not given access to the frame buffer or fragment shader of the actual screen. Therefore, it is impossible to run as if the user's eyes are the screen of the display (as you would on a computer screen running OpenGL or shadertoy, for instance). Instead, we are forced to run using a screen mesh in front of the user and path tracing through it using a shader that runs on the mesh.

Additionally, to add perspective all around the user, we chose to use a sphere mesh that encloses the user, much similar to how a virtual skybox is implemented. This means it is a sphere mesh that is centered at the user / camera position and is scaled inversely or inside-out. The sphere then runs a fragment shader that samples from a path traced texture. Table 1 shows what an example texture looks like, which is sampled by the sphere for correct perspective. To apply this projection, we first take the `uv` coordinates that we consider, and then transform them to polar and azimuthal angles[4]. We then have the following formula to transform them into world or Cartesian coordinates[5]:

$$\begin{aligned} x \text{ (horizontal)} &= \cos(\phi)\sin(\theta), \\ z \text{ (inwards)} &= \sin(\phi)\sin(\theta), \\ y \text{ (upwards)} &= \cos(\theta) \end{aligned}$$



(a) Spherical Texture (b) Viewed on Sphere Skybox

Table 1: Spherical Texture (Left) and Sphere-Sampled (Right) Cornell Box

3.3.2 Compute Shader v.s. Fragment Shader Path Tracing. A problem with running path tracing in the GPU is that we are unable to do recursion, as for a limitation in Metal and many other shading languages. We also cannot create random numbers built-in, which is used for Monte Carlo termination. We instead do so using Halton sequence numbers based on which frame (modded by a number to cap it) the compute shader is currently running. We then implemented the basic logic of the path tracing from homework 3 onto the shader in the Vision Pro.

The initial Vision Pro Path Tracer we built is ran directly on the fragment shader of the Sphere. When doing this, we get to employ the method of knowing where every pixel position in world space, allowing us to get a ray from the camera to the pixel, and allowing us to shoot a ray very intuitively through the sphere. However, the problem with this approach was that it was way too slow. Upon further research online, it seems that path tracers built in fragment shaders suffer from this approach, leading most path tracers to be done in compute shaders. Compute shaders are also much better than fragment shaders at handling conditional IF statements and suffer less from conditional branching issues, as it has full control over memory access patterns and work distribution. While initially we thought that compute kernels were not accessible to third party developers, we found very scarce but existent resources that allowed us to enable this capability [6].

The methodology to do path tracing on a compute shader is to perform path tracing using texture coordinates, and then sampling in the fragment shader after the texture is written to. The compute shader performs path tracing in the GPU and writes it into a CPU texture, which then gets sampled by the fragment shader in the GPU. This takes a bit of overhead, but in exchange, we get some performance benefits. Our path tracing algorithm takes advantage of the compute shader's explicit thread organization: `uint2 gid [[thread_position_in_grid]]`. This allows direct control over

[4]https://mathworld.wolfram.com/SphericalCoordinates.html
[5]https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/spherical-coordinates-and-trigonometric-functions.html
[6]https://developer.apple.com/documentation/realitykit/generating-interactive-geometry-with-realitykit

thread allocation and work distribution. Unlike fragment shaders which are tied to the rasterization pipeline and process one fragment at a time, compute shaders can skip rendering for out-of-bounds pixels with simple early-exit conditions.

3.3.3 Path Tracing Algorithm and BSDF Considerations. For light transport, we implemented direct and indirect illumination differently. The Metal version uses explicit light sampling in its pathTrace function with a single light sampling loop per bounce, whereas the C++ version on Homework 3 separates these concerns into multiple functions. Our GPU implementation also incorporates Russian Roulette termination after several bounces to improve efficiency, similar to the CPU version but implemented directly in the main path tracing loop. A significant difference is our scene representation - the Metal version uses a flat array of triangles passed from Swift, whereas the CPU implementation relies on a BVH acceleration structure. This simplified approach works well for our smaller scenes but sacrifices the performance benefits of spatial acceleration structures.

As with the Hw 3 code, the material of our object affects two things: 1) the contributions of light given an input and output ray direction and normal for a material (`evaluateBRDF` in code) and 2) the bias that we sample for each next ray after a bounce on that material (`sampleDirection` in code). Unlike the CPU implementation which uses separate BSDF classes, our Metal version employs a more unified and simplified material system with three primary material types (diffuse, metal, and dielectric) directly encoded in the shader. We then implemented the two main functionalities for each of these three BSDFs, which approximate physically based material behavior. We then created a USDZ[7] parser, and we classify triangles as one of the material types based on its extracted properties.

3.4 BSDF Evaluation and Sampling Equations

Given the input and output ray directions $\vec{\omega}_i, \vec{\omega}_o$ we evaluate the BSDFs. When calculating for the next ray after a bounce, we sample based on the BSDF given the input ray direction and the normal of the point where the ray intersects an object and use that as our new $\vec{\omega}_o$. We also importance sample based on lights to calculate direct lighting at every bounce.

3.4.1 Diffuse. For diffuse materials, we implemented a Lambertian reflection model with cosine-weighted hemisphere sampling[8]:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{\rho}{\pi}$$

Where ρ is the albedo.

3.4.2 Metallic. For metal surfaces, we implemented a simplified model based on perturbed perfect reflections for sampling:

$$\vec{\omega}_o = \text{reflect}(\vec{\omega}_i, \vec{n}) + \alpha\vec{r}$$

Where α is the roughness parameter, $\vec{r}$ is a random unit vector, and the reflected direction is computed with the standard reflection equations:

$$\text{reflect}(\vec{\omega}_i, \vec{n}) = \vec{\omega}_i - 2(\vec{\omega}_i \cdot \vec{n})\vec{n}$$

This is a simplified but computationally efficient approach to microfacet-based reflection [9]. What we are doing is basically to get a ray perturbed a little bit from the specular reflection direction [10].

The BRDF component combines base reflectance with a specular term and we just interpolate by a reasonable amount. This makes sense because the resulting color is made up of the base color along with a contribution from the specular term, where the specular term is some power of alignment of the reflected ray to $\vec{\omega}_o$:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \rho \cdot (0.2 + 0.8 \cdot \text{specular})$$

3.4.3 Dielectric. The results of this implementation can be found in part (e) of Figure 4, where the walls at the back of the Cornell Box are glass like (they are dielectric). A full implementation tutorial of Dielectric materials for Ray Tracing was used to implement ours [11]. The idea is that we first calculate the Fresnel term using Schlick's approximation:

$$F_r(\theta) = F_0 + (1 - F_0)(1 - \cos\theta)^5$$

Where $F_0 = 0.2$ in our implementation. We can easily calculate cos_θ as $|\vec{n} \cdot \vec{\omega}_o|$. The sampling probability for whether the scattered ray is reflected or refracted is directly based on this Fresnel term:

$$P(\text{reflection}) = F_r(\theta)$$

Finally, our dielectric BRDF accounts for the Fresnel effect:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \text{albedo} * \text{fresnel} = \rho \cdot (0.2 + 0.8 \cdot (1 - |\vec{n} \cdot \vec{\omega}_o|)^5)$$

4 Compute Pipeline for Path Tracing on Metal

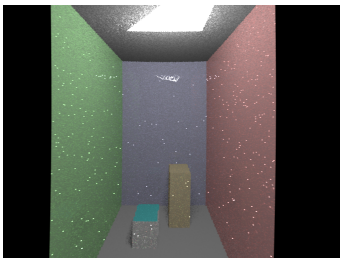
Our rendering system uses a three-stage compute pipeline to maximize image quality while maintaining interactive frame rates on Vision Pro hardware. Using compute kernels give us this flexibility. We have three compute passes that outputs a final texture that is then read by the fragment shader ran on the sphere previously discussed. Our approach is inspired by NVIDIA's paper [12] which uses temporal accumulation of samples per frame and applying a denoising filter on top of it. A large amount of unforgiving and time-consuming setup (`dispatchComputeCommands` etc) on the CPU Renderer side was required to get implement these passes correctly.

4.0.1 Path Tracing Compute Pass. The primary compute kernel (`pathTracerCompute`) generates raw path-traced samples at a configurable resolution. This kernel processes one ray per thread in a 32×8 threadgroup configuration for optimal cache utilization and finally writes to a dedicated `RGBA32F` texture on CPU that captures single-sample radiance values. However, these single samples are noisy. We don't have the flexibility to easily do multiple samples like we do on CPU, so instead we bypass this limitation by accumulating the results.

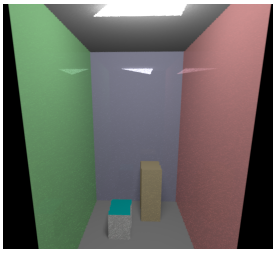
[7] A 3D scene file exchange format created by Pixar
[8] https://computergraphics.stackexchange.com/questions/4394/path-tracing-the-cook-torrance-brdf
[9] https://graphicscompendium.com/gamedev/15-pbr
[10] https://stackoverflow.com/questions/32077952/ray-tracing-glossy-reflection-sampling-ray-direction
[11] https://viclw17.github.io/2018/08/05/raytracing-dielectric-materials
[12] https://research.nvidia.com/labs/rtr/publication/schied2017spatiotemporal/

4.0.2 Accumulation Compute Pass. The second compute kernel (`accumulationKernel`) performs temporal accumulation to progressively reduce noise. We hope to essentially average the values over multiple samples. The kernel combines the current sample with previously accumulated samples using a running average. To do this, we use a numerically stable accumulation formula [13]: `new_avg = old_avg + (new_sample - old_avg) / sample_count`, and automatically reset accumulation when camera movement is detected so there are no incorrect overlays of views. However, these results are still quite noisy across different pixels.

4.0.3 Denoising Compute Pass. We implemented a denoising filter which was implemented before in CUDA [14] which is based the a research paper [15] "Edge-Avoiding À-Trous Wavelet Transform for fast Global Illumination Filtering". This filter implements a 5×5 kernel based on B3-spline interpolation applied in an A-trous style with iteratively increasing step widths. Our result is that we get edge-preserving filters, so our path traced results still look sharp at their edges while it's high frequency noise is removed. We translated this to fit our needs for our Metal renderer. Our results pre-denoising and post-denoising can be seen in Table 2.



(a) Noisy (1 spp)



(b) Denoised with À-Trous filter

Table 2: Comparison before and after compute pass

4.1 Metal GPU Data Structures

4.1.1 USDZ Parsing. To extend our path tracing functionality, we implemented a USDZ parser with the help of online sources in Swift, which we then pass down as triangles to the GPU with properties including emission, color, vertex positions, roughness, and material type (dielectric/diffuse/metal). We first used an exising Swift/Metal API that converts USDZ models to Metal formed Mesh objects. These objects contain submeshes, vertex buffers, vertex strides, and vertex data. Each submesh is mapped to a single material, which we determine it's type via testing based on `diffuse`, `metal`, and `index of refraction` properties encoded in the USDZ (if it is provided). We add pretty low-level accesses into raw pointers using these structures to extract the positions of the every submesh's triangles. They are extracted from a flat array, we apply any transforms, then convert them into an array of our `Triangle` data structure. These `Triangles` are then converted to `GPUTriangles` that we pass to the GPU.

4.1.2 Metal-Specific Memory Considerations. One of the challenges when implementing the Triangle data structure to pass down the GPU was managing the tradeoff between memory bandwidth and alignment patterns. When designing GPU data structures in Metal, the choice between packed float representations (such as `packed_float3`) and aligned float types (such as `float3`) presents an important performance consideration. This decision impacts both memory usage and access patterns, particularly when dealing with large triangle meshes required for path tracing.

Packed float types in Metal provide memory savings by eliminating padding bytes. For instance, a packed_float3 occupies exactly 12 bytes (3×4 bytes), while a standard float3 typically consumes 16 bytes due to alignment requirements. When storing geometry data for complex scenes with thousands or millions of triangles, this 25% reduction in memory footprint can significantly improve memory bandwidth utilization. This would be extremely useful for allowing us to render complex scenes with many, many triangles.

We implemented both. However, we decided to focus on smaller scenes without acceleration structures makes aligned floats the more practical choice. We avoid the complexity of handling unaligned memory access patterns and potential byte-addressing issues (which we attempted, with the cost of convoluting our code logic), and we keep memory access benefits for our compute-bound workload, where each triangle may be accessed multiple times during ray traversal and intersection testing

4.2 GPU Depth of Field and aberration Simulation on Metal

Our `pathTracerCompute` Metal kernel samples ray directions starting from the spherical coordinate UVs from the compute pass, which we then transform to world coordinates. This gives us an origin at the camera and a direction to the sample point on our sphere. Normalizing this is how we raytrace normally, however, for depth of field we need points along a given sensor.

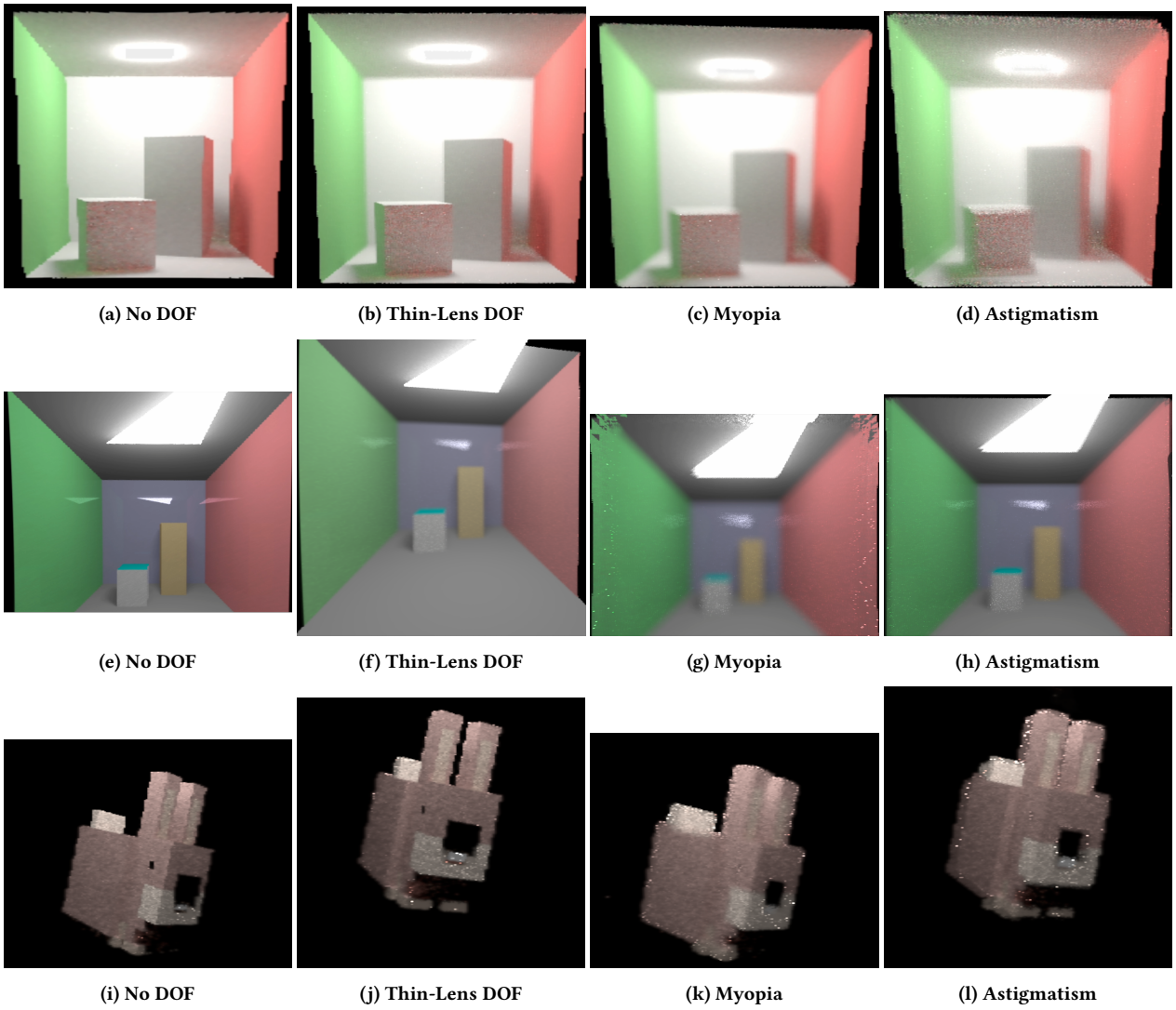
We can define a sensor by stepping forward one unit in the direction of the initial ray. Since this comes from the sampling of the sphere from the vision pro on compute, this already takes into account the FOV of the camera. With this sensor position, we can proceed as before to calculate our lens weighted ray direction based on our lens prescription parameters `SPH`, `CYL`, and `AXIS`. It is important to make sure to transform the coordinates to camera space before doing the sensor to focal point calculation. The resulting rays are transformed into world space and used in scene traversal and shading.

5 Results

5.1 Simulating aberrations on Vision Pro (Real-Time Results)

We can compare 4 different configurations between scenes: no DOF, thin-lens DOF, DOF with aberration (myopia and astigmatism). See Figure 4 [16] [17]. It is to note here that an astigmatic render produced a doubling affect in one direction defined by the `AXIS`. This mimics

[13] https://jvminside.blogspot.com/2010/01/incremental-average-calculation.html
[14] https://toytag.net/posts/pt-denoiser/
[15] https://jo.dreggn.org/home/2010_atrous.pdf
[16] https://sketchfab.com/3d-models/cornell-box-original-0d18de8d108c4c9cab1a4405698cc6b6
[17] https://sketchfab.com/3d-models/brown-bunny-crossy-road-c7b92c46ae2e4ba08b46384000280be2



(a) No DOF (b) Thin-Lens DOF (c) Myopia (d) Astigmatism

(e) No DOF (f) Thin-Lens DOF (g) Myopia (h) Astigmatism

(i) No DOF (j) Thin-Lens DOF (k) Myopia (l) Astigmatism

Figure 4: Comparison of depth-of-field and aberration configurations across three scenes.

both the the lens equations and real life experiences. It is most prominent when examining the light, just like in real life. When rendering with a point light and exaggerate the CYL effects, we can see the streaking of light in real-time, such as in this result **??**.

5.2 High Quality Renders (Arnold)

We start with a very simple scene, which shows the exagerrated light streaking effects of severe astigmatism 7. We then render a bedroom scene[18]. We begin by showing both a standard pinhole camera render (`lensRadius = 0`). Areas of interest has been enlarged. Additionally, the camera is set to focus on the lamp (as demonstrating in the standard depth of field render). In the comparison between the uniform vs biased sampling, we did not notice any noticeable improvements. Additionally, it's noted that there is a substantial amount of noise for the light streaks.

[18] "ROOMV1352233754567" (https://skfb.ly/puT6t) by tokoissick is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/)

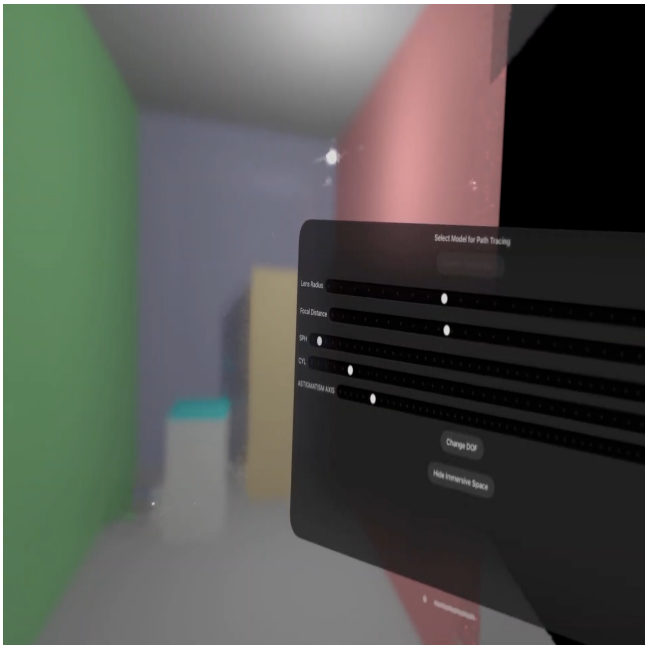


Figure 5: Lights Streaking on a Vision Pro Rendering of the Cornell Box

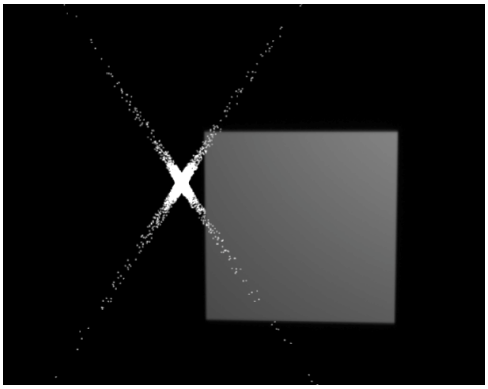


Figure 7: Simple Scene Simulating Astigmatism

It is noted here in this image that the light is placed behind the cube in space. This demonstrates the observed behavior of lights streaking across objects in front of it, as this is a function of the camera samples directly sampling the light source.

5.3 Problems Encountered

5.3.1 Vision Pro Problems. High Quality Renders on the Vision Pro were too computationally expensive, so we settled on processing more complicated scenes using Arnold. Metal supports accelerated Ray Tracing structures, but they are not available on the Vision Pro as they have not been made compatible. A potential extension is to implement our own acceleration structures on CPU and traverse them in the GPU shaders. Additionally, converting clinical parameters (diopters) into usable focal lengths required approximation.

One of the difficulties with our current approach is that there are many reads and writes from the CPU, as each texture result between each compute pass is read and written to between GPU and CPU, which is expensive. This can actually change if we take advantage of Apple GPU's Tile Based Architecture by implementing Tile Shading. Tile Shading stores the textures as colors in tile memory lying in GPU units instead of CPU, and having all render and compute passes access that memory[19]. This will complicate our implementation significantly; we failed to implement this correctly in time.

Another challenge was the limitations of the headset hardware. There are jaggies on the object because the number of pixels on the Vision Pro is so high compared to what we can render in real time. These jaggies disappear on the simulator on a computer screen. Additionally, since the human head is moving slightly all the time, our accumulation shader gets reset too often, which results in fireflies constantly while wearing the device, which disappear on the simulator where the user can be completely static.

5.3.2 CPU Rendering Problems. It's noted that the light streaks are caused by camera samples. As such, this leads to a really noisy image unless the sample counts are cranked up a lot, which increases render times across the entire scene. A better sampling approach could possibly be taking to reduce noise.

Additionally, since we cannot model eye accommodation, we approximated it using a predefined `focalDistance` and added the `eyePower` to it to simulate it. In real life, the `eyePower` of each meridian is preset and the `focalDistance` is a function of the accommodation of the eye. It was difficult to find a formula that worked in a path tracer that was as accurate as possible.

6 Conclusion and Future Work

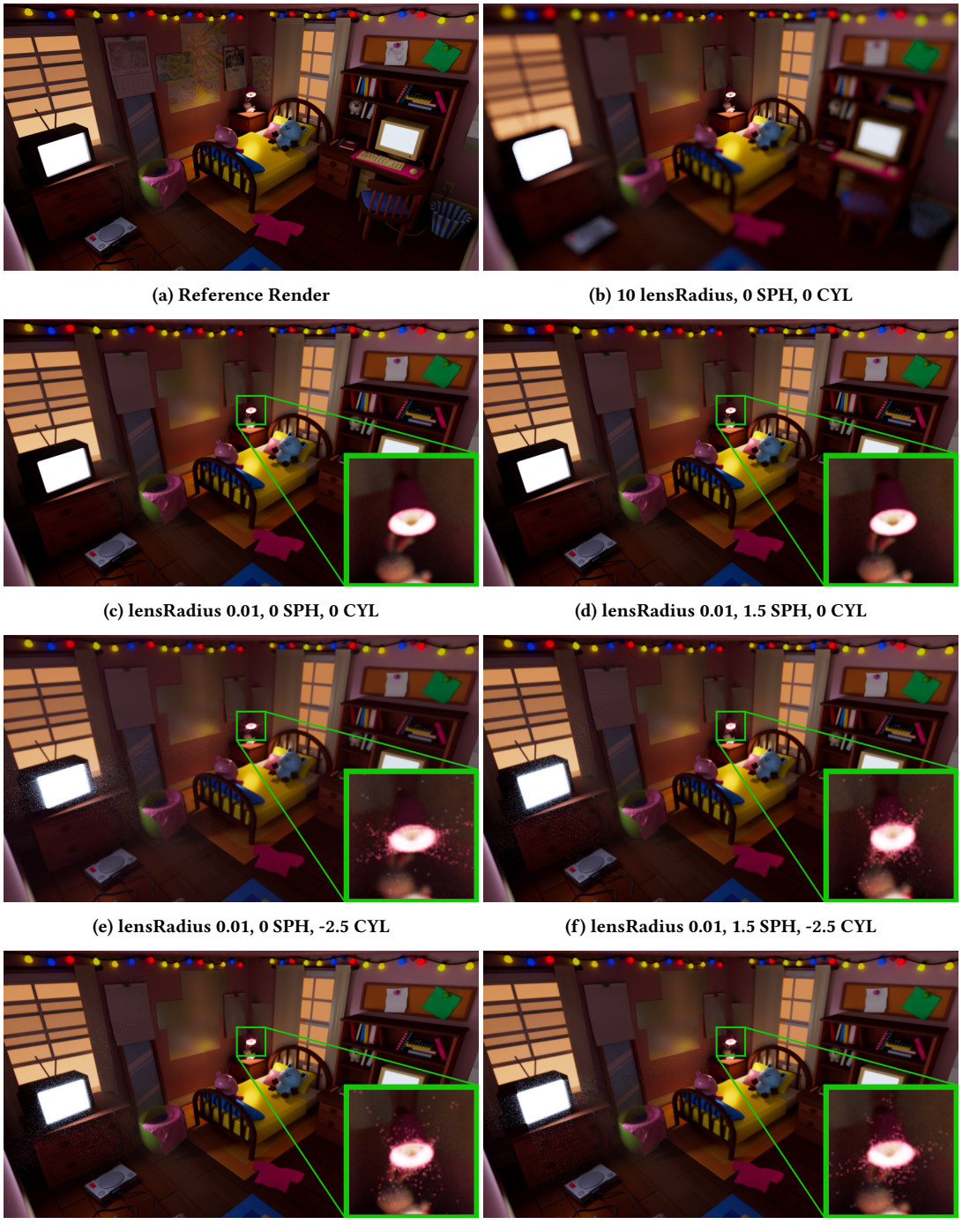
There are many optimizations that can be done to the Vision Pro's path tracer. Some of these include implementing acceleration data structures for path tracing, implementing our render and compute passes in one encoder that takes advantage of Tile Shading. Future work might include applying these aberration simulations to real-world objects seen and scanned in the Vision Pro.

Furthermore, this current approach only supports regular astigmatism in which the meridians are 90 deg perpendicular to each other. An additional modification of the lerping function could accommodate irregular astigmatism as well. With our current approach of projecting the focalPlane forward based on the sensor location and the `focalDistance`, and then calculating the ray from the lens to the focal point, it lacks the ability to trace scenes accurately for negative focal distances. This creates an inability for our current implementation to render hyperopia. However, a modification to the final ray direction algorithm can fix it.

7 Contributions

- Brayton Lordianto: I was heavily involved and worked on all the implementation, research, and testing of the Metal Path tracer (including materials BSDFs, Metal coding, Renderer changes, Compute shaders and more) and contributed to implementing depth of field and aberration simulation in Metal.

[19] https://developer.apple.com/videos/play/tech-talks/604/



(a) Reference Render

(b) 10 lensRadius, 0 SPH, 0 CYL

(c) lensRadius 0.01, 0 SPH, 0 CYL

(d) lensRadius 0.01, 1.5 SPH, 0 CYL

(e) lensRadius 0.01, 0 SPH, -2.5 CYL

(f) lensRadius 0.01, 1.5 SPH, -2.5 CYL

(g) lensRadius 0.01, 1.5 SPH, -2.5 CYL, biased sampling

(h) lensRadius 0.01, 1.5 SPH, -2.5 CYL, 30 AXIS

Figure 6: Comparison of different prescription and sampling methods

- Han Li: I was primarily involved in the vision research for eyes and lenses, as well as implementing the CPU algorithm in Hw 3 and Arnold renderer. I also integrated the astigmatism and dynamic focal distance algorithm into the Metal compute shader.

- William Wu: I also worked on the vision research and spearheaded the understanding and implementation of the mathematical formulations for the depth of field and visual aberration simulation. I was involed with implementing the depth of field and anisotropic focal distances on top of Hw3 as well as debugged the simulation in the Arnold renderer, and in the Metal version.
- Ricardo Blanco: I assisted with the porting of DOF logic with support for aberrations from Homework 3-2 into the metal shader that Brayton built up and worked on all deliverables.

References

[1] Brian A. Barsky, Scott A. Klein, and Daniel H. Shevell. Vision-realistic rendering: Simulation of the scanned foveal image from wavefront data of human subjects. *Proceedings of the 29th annual conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2002. https://people.eecs.berkeley.edu/~barsky/VisRendPapers/vrr1.pdf

[2] Matthias Nießner and Marc Stamminger. Real-time simulation of human vision using temporal compositing with CUDA on the GPU. *Vision, Modeling and Visualization (VMV)*, 2013. https://niessnerlab.org/papers/2013/0temporal/niessner2013temporal.pdf

[3] Matthias Nießner and Marc Stamminger. Real-time simulation and visualization of human vision through eyeglasses on the GPU. *Eurographics (Short Papers)*, 2012. https://graphics.stanford.edu/~niessner/papers/2012/0eyeglass/niessner2012eyeglass.pdf

[4] Paul Jones et al. Degraded reality: Using VR/AR to simulate visual impairments. *IEEE Transactions on Visualization and Computer Graphics*, 2018. https://discovery.ucl.ac.uk/id/eprint/10065529/1/PJ_vr_ieee_v6_nonblind.pdf

[5] P. Johnson et al. Simulating vision impairment in virtual reality: a comparison of visual task performance with real and simulated tunnel vision. *Virtual Reality*, 2024. https://link.springer.com/article/10.1007/s10055-024-00987-0

[6] Mengjie Xu et al. XREye: Simulating visual impairments in eye-tracked XR. *IEEE Transactions on Visualization and Computer Graphics*, 2020. https://ieeexplore.ieee.org/document/9090438

[7] Timo Koehler et al. Realistic simulation of progressive vision diseases in virtual reality. *International Conference on Image Analysis and Processing (ICIAP)*, 2018. https://www.researchgate.net/publication/329259460_Realistic_simulation_of_progressive_vision_diseases_in_virtual_reality

[8] David McGookin et al. Simulating visual impairments using the Unreal Engine 3 game engine. *IEEE International Conference on Serious Games and Applications for Health (SeGAH)*, 2012. https://ieeexplore.ieee.org/document/6165430

[9] Philip W. Quinn and David A. Ross. Real-time simulation of arbitrary visual fields. *ETRA*, 2002. https://svi.cps.utexas.edu/ETRA2002.pdf

[10] Csoba, I., Kunkli, R. Rendering algorithms for aberrated human vision simulation. *Visual Computing for Industry, Biomedicine, and Art*, 2023. https://doi.org/10.1186/s42492-023-00132-9

[11] Zilong Wang et al. Simulation of Human Eye Optical System Properties and Depth of Field Variation. *International Journal of Machine Learning and Computing, Vol 3, No.5*, 2013. https://www.ijml.org/papers/351-C011.pdf

[12] Zachary J. Rawlins et al. VisionaryVR: An optical simulation tool for evaluating and optimizing vision correction solutions in virtual reality. *arXiv preprint*, 2023. https://arxiv.org/html/2312.00692v1

[13] Samuli Laine and Tero Karras. Megakernels considered harmful: Wavefront path tracing on GPUs. *High Performance Graphics*, 2013. https://research.nvidia.com/sites/default/files/pubs/2013-07_Megakernels-Considered-Harmful/laine2013hpg_paper.pdf

[14] Jacco Bikker. Wavefront path tracing. *Ompf Blog*, 2019. https://jacco.ompf2.com/2019/07/18/wavefront-path-tracing/

[15] MLX Contributors. Custom Metal Kernels. *MLX Developer Docs*, 2023. https://ml-explore.github.io/mlx/build/html/dev/custom_metal_kernels.html

[16] Stack Overflow. Metal Compute Kernel vs Fragment Shader. *StackOverflow Thread*, 2016. https://stackoverflow.com/questions/41014713/metal-compute-kernel-vs-fragment-shader

[17] Apple Inc. Explore USD tools and rendering. *Apple WWDC 2022*, Session 10141. https://developer.apple.com/videos/play/wwdc2022/10141/

[18] Pixar Animation Studios. OpenUSD C++ API Documentation. *OpenUSD Developer Documentation.* https://openusd.org/docs/api/index.html

[19] Pixar Animation Studios. OpenUSD Documentation and Source Code. *GitHub Repository.* https://github.com/PixarAnimationStudios/OpenUSD

[20] Pixar Animation Studios. OpenUSD API Reference: UsdShade. *OpenUSD Release Documentation.* https://openusd.org/release/api/usd_shade_page_front.html

[21] Pixar Animation Studios. Hydra Delegate Getting Started Guide. *OpenUSD Release Documentation.* https://openusd.org/release/api/_page__hydra__getting_started__guide.html

[22] Apple Inc. Creating a 3D Application with Hydra Rendering. *Apple Developer Documentation.* https://developer.apple.com/documentation/metal/creating-a-3d-application-with-hydra-rendering

[23] M. Pierscionek et al. Refractive index distribution and optical properties of the isolated human lens measured using magnetic resonance imaging (MRI). *Vision Research*, Vol. 45, Issue 18, 2005. https://www.sciencedirect.com/science/article/pii/S0042698905001690