METALWAVE: VR Metal Wavefront 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

This milestone report presents the current progress and theoretical design of our project to develop a physically accurate path tracer capable of simulating visual impairments. Thus far, we have implemented a basic ray tracer to validate our rendering framework and have outlined a detailed pipeline for full system integration.

ACM Reference Format:
Brayton Lordianto, Han Li, William Wu, and Ricardo Blanco. 2025. METALWAVE: VR Metal Wavefront Path Tracer Lens Simulation for Aberrated Vision. In . ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Summary of Accomplishments

We have successfully implemented a basic ray tracer as the foundation of our rendering system. This has helped us validate our setup and get started with GPU-based path-tracing rendering.

The majority of our current work has been focused on pipeline architecture and system design. We have thoroughly researched and mapped out the following components:

- Integration of USDZ scene file specification.
- Scene parsing using Hydra and the HdSceneIndexObserver interface.
- Creating a custom METALWAVE render delegate to interface with Hydra.
- Planning for mesh, material, and camera data transfer to the renderer.
- Research on visual lens impairment and their specific causes.
- Lens modeling for simulating refractive vision impairments.

1.1 Limitations We Found

The Vision Pro is unable to allow compute shaders and also does not allow us to write directly to the frame buffer on the screen. The first issue means that we are unable to run recursive path tracing, wavefront path tracing, or utilize GPU acceleration structures such as Bounding Volume Hierarchies (BVH). To bypass these issues, we:

- bypass the second issue by using a screen mesh in front of the user and path tracing through it using a shader that runs on the mesh. We are still looking into how to make this get affected by the perspective of the camera. At the moment, it is quite static (see video in slide submission).
- Since we cannot run recursion on the fragment shader, we run path tracer iteratively, and then we accumulate the results over multiple frames by reading and writing to a texture to simulate the result of recursive path tracing over multiple samples of rays.
- we plan to create BVH in the CPU and pass them to the fragment shader and perform path tracing as we were in the GPU. If this is not successful, we will stick with simple scenes to render.

2 Preliminary Results

At this stage, our ray tracer is functional and renders basic primitives. We also have a path tracer, albeit it is noisy. While we don't yet have visual output from our full pipeline, the initial tracer sets the stage for GPU acceleration and lens simulation.

All other components—USD support, custom rendering delegate, and optical simulation—remain in the design phase but are well documented and scoped. Notes on how to pass USD mesh data (e.g., topology, points, normals) and set up the camera through RealityKit and OpenUSD are in place.

3 Reflection and Updated Work Plan

We originally planned to have USD and Hydra integration by this point, but more time was needed for foundational setup. The USD specification was quite complex and render pipelining was new to us. We're confident in our pipeline design, however, and feel well prepared to begin implementation.

Next Steps:

- Begin parsing USDZ files using Hydra.
- Start building the render delegate interface.

4 Updated Timeline

- **Week 3:** Debug Path-Tracing code.
- **Week 4:** Start Hydra integration and delegate planning.

Conference'17, Washington, DC, USA

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

5 Appendix: Pipeline Theory and Setup Notes

The full rendering pipeline we have planned is:

(1) Load USDZ file with a file picker in VisionOS.
(2) Parse mesh and material data via Hydra and the HdSceneIndex interface.
(3) Implement a custom render delegate that transfers Hydra data to METALWAVE.
(4) Use METALWAVE to handle physically-based rendering and apply optical distortions.
(5) Add lens modeling for visual impairments (myopia, hyperopia, astigmatism).

Key technical insights gathered:

- USD mesh data requires 'GetMeshTopology()', 'GetPoints()', and 'GetNormals()' from the 'HdMesh' prim schema.
- Primitives are detected via 'PrimsAdded' callback in the observer and streamed into GPU buffers.
- Apple's USD example code can be adapted for camera setup, stage loading, and file access.

Camera, material, and lighting data handling are still TBD but scoped in our notes.

6 Links

- https://drive.google.com/file/d/1M_fZZQyKjAV9gV-t26Vh_uzpQSM6ZH5F/view?usp=sharing
- https://docs.google.com/presentation/d/1Jv0HV-rqpvEsjhGwI6ovAgRwjD4v edit?usp=sharing
- https://brayton-lordianto.github.io/Computer-Graphics-Sites/gpu-path-tracing/