EE267 Final Project

Divyahans Gupta (dgupta2@stanford.edu)

June 2019

1 Summary

I built a foveated ray tracer for virtual reality using NVIDIA's Falcor real-time rendering framework. The Falcor framework runs on hardware that supports Microsoft DXR raytracing, such as the NVIDIA RTX series.

2 Raycasting

I started from the tutorial code provided by Chris Wyman in his 2018 SIGGRAPH course "Introduction to DirectX Raytracing" [1] . I specifically used helper libraries and abstract classes that removed much of the verbosity of using DirectX. The `RayTraceGBufferPass` class and its accompanying shader traces a ray from each pixel out to the screen from the Falcor camera and saves the intersection point in a G-buffer, which holds normal, diffuse, albedo, and other maps that will be used for deferred shading. I modified the `RayTraceGBufferPass` class and raytracing shaders to support stereoscopic rendering. I passed two cameras to the raytracing shader that are separated by an adjustable IPD parameter on each side of the default Falcor camera. The correct camera is chosen based on which stereo image the pixel is in. Thus, **the stereo image pair is rendered simultaneously in one pass**. Doing so improves the memory locality of the BVH intersection calculations in DXR since there is coherence amongst the rays coming from both cameras. I used the `LambertianPlusShadowPass` code from the tutorial as is to shade the pixels. This pass performs Lambertian shading and shoots a single shadow ray per pixel.

3 Foveation

Once that was complete, I added support for foveated rendering. The intuition behind foveated raycasting is that the rendering can sample rays in the eye gaze more densely than rays in the periphery, thus saving rendering costs overall. This performance benefit is not realizable in rasterization and actually introduces overhead as we implemented it in class. For raycasting to realize this performance benefit, we must render the image in log-polar space. As illustrated in 1, areas closer to the center of the Cartesian coordinate system cover more area in log-polar space than areas farther from the center. After validating the transformations in Python, I modified my raycaster to sample ray directions from from log-polar space, causing the center of the Cartesian coordinate system to be more densely sampled than the periphery. The rendering savings are achieved by ensuring that the log-polar buffer is smaller than the full resolution buffer. We use the ratio of the log-polar buffer size to the full resolution buffer size to measure the computational savings realized. The deferred shading maps are saved to this smaller log-polar buffer and the shading pass is performed in the log-polar buffer as well. **The smaller log-polar buffer reduces rendering cost because it allows the raycaster to 1) sample fewer rays and 2) shade fewer pixels compared to a full resolution rendering pipeline.**

In order to finally present images on the full resolution display, we perform the inverse transformation in the `inversePolar` shader. That is, for each pixel in the full resolution display, the shader samples from corresponding point in the shaded log-polar buffer. Because fewer samples exist in the peripheral space, this pass will introduce some aliasing in the periphery. As mentioned in [2], this effect can be mitigated by performing a Gaussian blur on the right side of the log-polar buffer. I will add the blurring feature later. You can see the foveation effect in Figure 2 and Figure 3. The aliasing should not be so pronounced that it catches the attention of the user in their periphery.

My implementation allows the user to easily adjust the size of the stereo log-polar buffer size in Falcor to analyze the tradeoff between performance improvement and visual degradation.

The `inversePolar` also optionally performs a barrel distortion pass for VR viewing. The computational waste induced by barrel distortion (which effectively deletes some pixels that have already been traced and shaded) is reduced since we render to a smaller log-space buffer.

In my evaluations, I found that the log-polar buffer size can be reduced to about half of the full resolution size without major visual aberrations, which effectively halves the amount of tracing/shading work and doubles the framerate. For simplicity, I assume the Cartesian coordinate center of each stereo image is its optical center, so the eye gaze is fixed.

I partially implemented the focus of [2], which introduces a parameter in the log-polar transformation to adjust the relative areas of center and peripheral regions in log-polar space. Future work would tune this parameter to model the MAR function we learned in class.

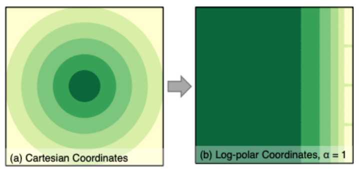

Figure 1: Cartesian to log-polar space. Taken from [2]

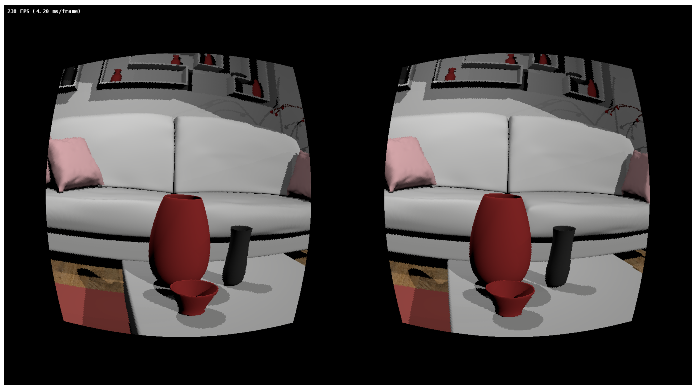

Figure 2: Foveated view centered at top of red vase sampled from log-polar buffer that is half the resolution of display. It is indistinguishable from full resolution at the center but has aliasing in the periphery.

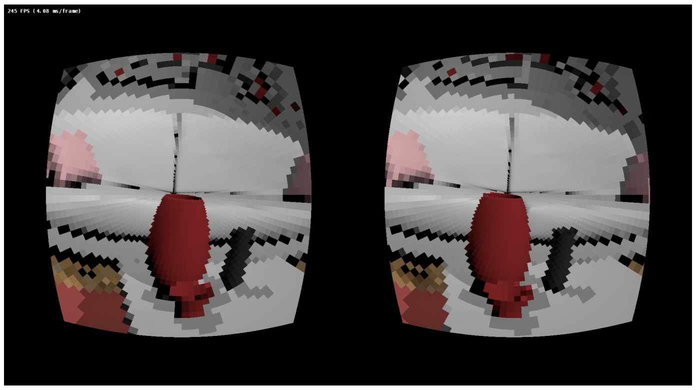

Figure 3: Foveated view centered at top of red vase sampled from very small log-polar buffer for illustration purposes.

4 Files

The primary source files I modified/wrote are:

- `inversePolar.ps.hlsl`
- `RayTraceGBuffer.rt.hlsl`
- `InversePolar.(cpp, h)`
- `RayTraceGBufferPass.(cpp, h)`

References

[1] C. Wyman, S. Hargreaves, P. Shirley, and C. Barré-Brisebois, "Introduction to directx raytracing," in *ACM SIGGRAPH 2018 Courses*, August 2018.

[2] X. Meng, R. Du, M. Zwicker, and A. Varshney, "Kernel foveated rendering," *Proceedings of the ACM on Computer Graphics and Interactive Techniques (I3D)*, vol. 1, pp. 1–20, May 2018.