Virtual and Augmented Reality

This report discusses our findings from implementing pixel-based and mesh-based solutions to distortion correction for VR lenses in Unity. Both solutions use shaders to convert the cartesian coordinates of a pixel or vertex to polar coordinates and then apply Brown's model of radial distortion to calculate the distorted radius before converting back to cartesian coordinates to obtain the new location of the pixel or vertex. Our mesh-based methods for both pre-distortion and correction were implemented in the vertex shader in Clip space (with all vertex coordinates between -1 and 1). This allows us to convert the vertices into polar coordinate form without needing to calculate the centre of the distortion which would be required if the calculations were performed in world space. For pixel-based solutions, the shader scales the coordinates to between 0 and 1 and so we multiply by 2 and then subtract 1 from these values to centre the radius about the origin as assumed by Brown's model.

To view a problem in the accompanying Unity submission, run the scene in Game Mode. Select the "ViewSceneCamera" in the Unity Hierarchy and enter the corresponding value from Table 1 in the problem field in the problem script section of the inspector for this camera. To change the shader coefficients for a problem, select the corresponding mesh in the Unity Hierarchy from Table 1 and change the c1 and c2 values in the Render Texture section of the inspector for this mesh.

Problem	Value for Problem Field in ViewSceneCamera	Mesh For Shader Coefficients
1	1	Problem1
2	2	Problem2
3.1	3.1	Problem3.1
3.2.1	3.2.1	uvmesh/Problem3.2.1
3.2.2	3.2.2	uvmesh2/Problem3.2.2
3.2.3	3.2.3	uvmesh3/Problem3.2.3
3.3	3.3	Problem3.3

Table 1.

Effect of Geometric Complexity of Pre-Distortion Quality

We used Blender to create three different planes of varying geometric complexity. The first plane (Figure 1) consisted of 5.86k vertices and 2.1k triangles. Our second plane (Figure 2) consisted of 10.0k vertices and 10.24k triangles and our final plane (Figure 3) consisted of 1.06M vertices and 2.10M triangles. For the first mesh (which had the fewest number of vertices and triangles), the cubes nearest the edges of the mesh experienced more warping of their edges which curve slightly inwards. This was also experienced by the second mesh, although this was more apparent during animation. The third mesh experienced the least warping of cube edges which appear straight both in Figure 3 and when run in Unity.

The CPU profiler showed that our first mesh had a CPU Usage of approximately 1.50ms per frame whilst the GPU Usage showed a rendering time per frame of 0.60ms (of which 0.30ms was due to shadows). For the second mesh, the GPU profiler revealed a rendering time of

0.64ms of which 0.28ms was due to shadows. In addition, the CPU profiler displayed a rendering time of 1.53ms (including 0.49ms for rendering). The third mesh had a GPU rendering time of 0.79ms mainly due to shadows and opaque rendering and a CPU rendering time of 1.57ms of which 0.49ms was due to rendering. It is clear that for large triangle and vertex counts, the opaque rendering time per frame increases significantly (from 0.10ms for mesh 1 to 0.31ms for mesh 3). The shadow rendering time also makes up a major portion of the GPU usage per frame however it does not increase much with vertex count. We can also see that CPU usage remains fairly similar for all vertex counts that were tried and roughly the same amount of time was spent on CPU rendering for all meshes.

A Comparison of Pixel-based and Mesh-based Distortions

For pixel-based distortion methods, we need to apply Brown's model (or a similar model) to every pixel. The size of our render texture is 256×256 pixels. Therefore, a pixel-based method requires 65,536 calculations. On the other hand, a mesh-based method only requires us to apply Brown's model to each vertex in the mesh (about 5.8k in total for our initial mesh-based implementation in 3.1). Many VR applications require framerates of 90+ FPS to run smoothly and avoid causing nausea. The number of vertices needed for mesh-based methods is typically much smaller than the number of pixels needed to achieve the same quality output for pixel-based methods so we can reach higher framerates. Mesh-based methods also tend to lose accuracy in the distortion generated by a mesh-based method as any distortion between vertices is simply calculated using linear interpolation and we often use far fewer vertices than pixels. Both methods require the same number of texture lookups as texture lookups are performed in the fragment shader and so we still process on texture lookup per pixel.

The mesh-based approach is also less suited for chromatic aberration pre-distortion than a pixel-based method as a vertex cannot be warped to more than one location on a single mesh. Therefore, to correct chromatic aberration, we would require warping a separate mesh for each of the R, G, B channels and we would then need render each of these meshes in turn to render the overall image. This is harder to implement and requires triple the number of calculations compared to mesh-based pincushion correction and therefore is three times less efficient.

We examined the performance of our pixel-based (Figure 4) and mesh-based (Figure 6) solutions for barrel distortion using the Unity Profiler. The mesh-based method had a GPU rendering time of 0.60ms per frame whereas the pixel-based method had a GPU rendering time of 0.62ms per frame, showing that the mesh-based method is slightly more efficient as expected. Both methods had similar CPU rendering times of around 1.50ms per frame.

Use of Eye-Tracking for Distortion Correction

The lens distortion correction shaders we have implemented assume that the centre of the lens is in line with the centre of the eye. We therefore apply radial distortion symmetrically from the centre of the screen. An assumption made by this approach is that the distortion experienced is circularly symmetric and so the amount of distortion experienced only depends on distance from the lens centre. This would imply that as a headset user looks

around, the distortion would not change since the distortion does not depend on the direction from which it is perceived. However, in virtual reality the human eye is not fixed. The position of the pupil is able to change by a few millimetres as when the eye rotates, the pupil's trajectory forms an arc. This means that as the eye rotates, the lateral distance from the eye to the lens changes as when a user of a VR headset looks to the side (or up or down), the pupil will be further from the lens than if a user was looking directly at the lens due to the elliptical shape of the human eye. The distortion received through the lens to the pupil is therefore misaligned with the software pre-correction.

Using eye-tracking, we can determine at which part of the lens a user's pupil is focused. From this, we will be able to calculate the distance from the lens and thus determine the strength of the distortion experienced by the user and apply the appropriate strength of software pre-correction to counteract this. Eye-tracking could therefore be used to improve corrections for both pincushion distortion and lateral chromatic aberration as these are both forms of radial distortion. For chromatic aberration, we should recalculate the distortion correction needed for each of the R, G, B channels as the eye's z-distance from the lens changes.

Implications of Barrel Distortion on Resolution

In barrel distortion, parts of the image at the edges of the image are compressed. This causes the resolution of the image to be lower nearer the edge compared to the centre and, as a result, some of the shading calculations performed for pixels corresponding to the edges of the image are wasted as the detail is not preserved. In addition, the centre of the image will appear more detailed to a user than the edges of the image.

To solve this problem, we can use multiresolution shading; the centre of the image (where compression does not occur) is rendered at a high resolution and the edges (which are compressed by distortion) are then rendered at a lower resolution to save calculations. This allows us to improve the performance of our VR headset whilst maintaining the same output quality. Typically, an image is divided into a 3×3 grid where the centre square is rendered at full resolution and the other squares are rendered at a lower resolution. For example, sections corresponding to the edge of the image might be rendered at half resolution and corner sections may be rendered at quarter resolution.

Rendering in VR is computationally expensive. In addition to resolution, we also need to optimise latency. Typically, a framerate constantly above 90 FPS is required to avoid experiences of nausea as objects will appear delayed in adjusting their position on the screen compared to a user's head movements. We can therefore only increase our rendering resolution so long as our framerate remains above 90 FPS. A higher rendering resolution will enable a user to see more detail in an image, particularly as the image is very close to the user's eyes and so they may see a pixelated screen for lower resolutions. However, no additional detail will be obtained by rendering at a higher resolution than the hardware resolution as this is the maximum number of pixels that we can obtain from the screen prior to any distortion and so we cannot gain any more information from the image for a higher quality rendering. Therefore, we should aim to render at the hardware resolution so long as our framerate remains above 90 FPS to reduce latency, lowering our rendering resolution until the framerate passes this threshold if necessary.

The apparent resolution of a scene is the level of detail at which a headset user views the scene. This is lower than the rendering resolution because the scene is first per-corrected using barrel distortion, causing loss of detail at the edges of the image, and then experiences pincushion distortion as it passes through the lens which causes loss of detail due to compression at the centre of the image. Therefore, the overall resolution of the image as seen by a VR headset user is lower than the resolution that the scene was rendered at. Figure 7 shows the output after both the pre-distortion and unwarping stages have been completed. It was difficult to select appropriate coefficient values to exactly calculate the inverse for our generated barrel distortion and so some of the pincushion distortion is still visible in this image. We can also see that the detail for the cubes is significantly lower as they often appear curved and show some aliasing compared to the original scene. A possible fix for this would be to render our image at a very high resolution so that after image distortion and pre-correction the scene is displayed to the user at a suitable apparent resolution. As the screen is very close to the user's eye, this would require a very high-resolution screen which is very expensive and would increase the latency of our headset due to the hardware constraints of modern designs.

As super-sampling is typically too expensive for modern VR headsets, we need to consider more efficient variations of anti-aliasing such as multisample anti-aliasing (MSAA) or morphological anti-aliasing (MLAA). For MSAA, instead of performing super-sampling for all pixels, we only consider border conditions at a higher resolution whereas MLAA examines edges that form a discontinuity in the image and calculates the likely orientation of that edge.

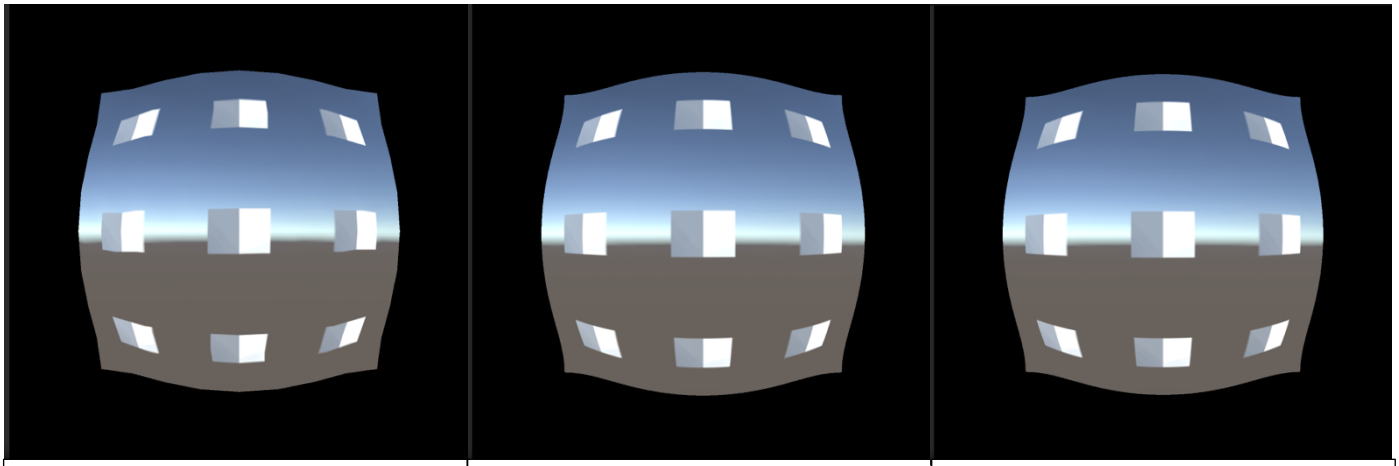

Figure 1: Barrel Distortion using a mesh with the 5.85k vertices and 2.1k triangles

Figure 2: Barrel Distortion using a mesh with the 10.0k vertices and 10.24k triangles

Figure 3: Barrel Distortion using a mesh with the 1.06M vertices and 12.10M triangles

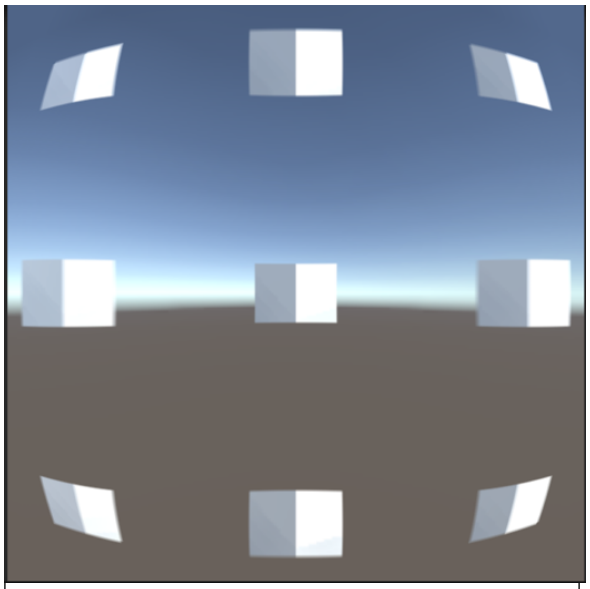

Figure 4: Pincushion Pre-distortion correction for Problem 1.1 using c1=-0.4, c2=0.3

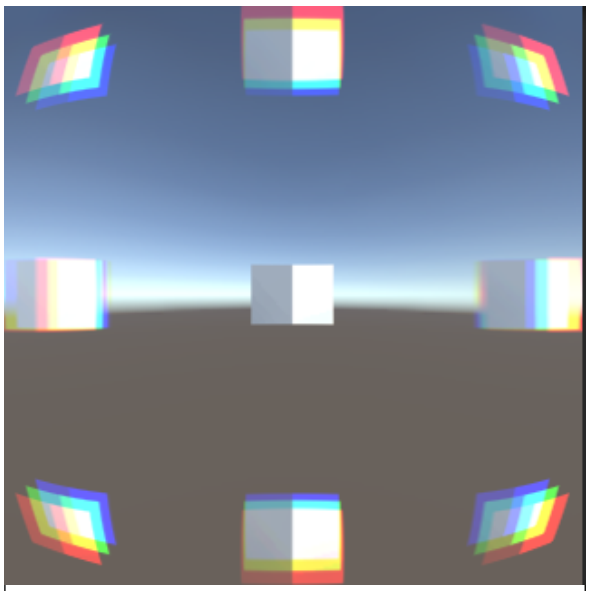

Figure 6: Lateral Chromatic aberration pre-correction for Problem 2 using c1=-0.5, c2=0.3 for red, c1=-0.45, c2=0.3 for green and c1=-0.4,c2=0.3 for blue

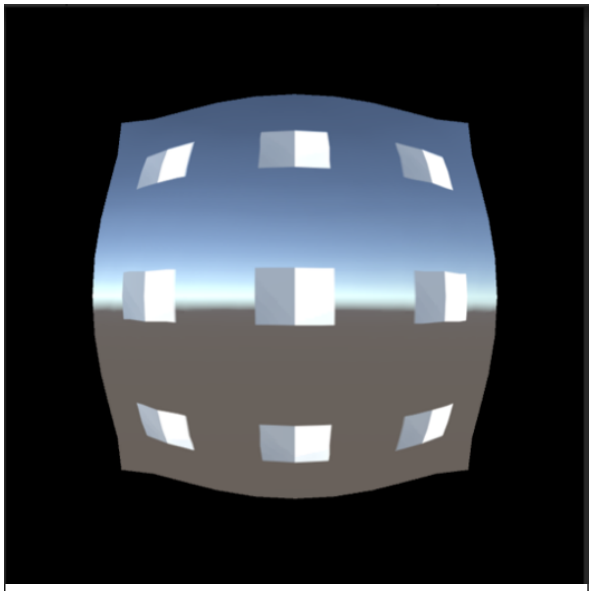

Figure 5: Mesh-based Barrel Distortion for Problem 3.1 using c1=-0.3, c2=0.1

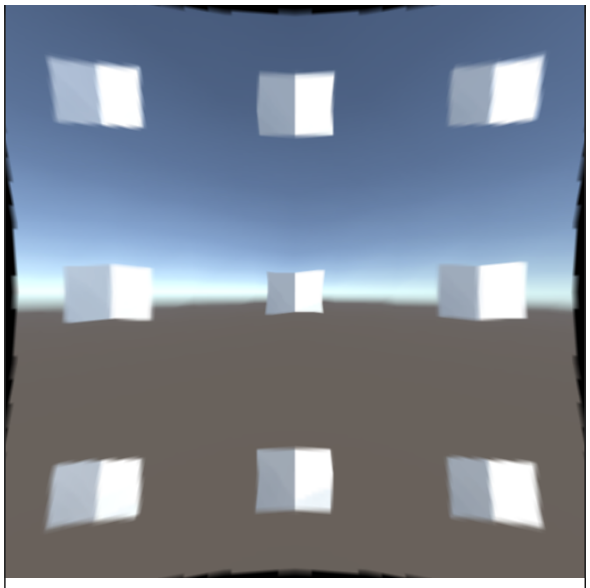

Figure 7: The resulting image from Problem 3.3 simulating the corrected image passing through a lens