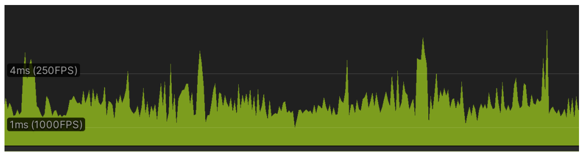
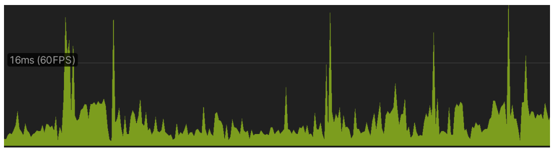
**Research Report: Virtual and Augmented Reality**
CIS ID: cssg28

*3.2) Performance impact of different mesh complexities in Unity Profiler*

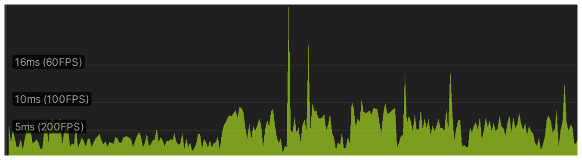
When distortion is performed in the vertex shader, I observed that pre-distortion quality improves significantly with more complex meshes; renders of barrel and pincushion distortion using meshes of various geometric complexity are displayed in section 4.1c. I progressively increased the complexity of the meshes to observe the difference in pre-distortion quality; meshes with 2, 50, 162, 452, 800, 1682, 5000, 9800, 20,000, 45,000, 96,800, and 500,000 triangles we used. After the complexity of the mesh reached ~1000 triangles, I noted that the visual quality of the distortion began to plateau. However, the impact of mesh complexity on performance (i.e. Unity Profiler analysis) was minimal for meshes containing less than ~5000 triangles; significant performance impact was observed with meshes of size 9800+ triangles. The Unity Profiler was used to quantify performance impact of meshes of differing complexity. The rendering speed in FPS was observed for each mesh. For meshes of less than 5000 triangles, I noted that the average rendering speed hovered around 1000 to 250 FPS, spiking occasionally. For larger meshes, the average rendering speed decreased, and more notable spikes were prevalent; for the 500,000 triangle mesh, for example, rendering speed reached a lower bound of ~ 60 FPS. As the average rendering speed likely depends on computer hardware/processing power (e.g. number of cores, access to GPU), I anticipate that the Unity performance would likely differ amongst devices.



*i. Unity Profiler output for 50 triangle mesh*



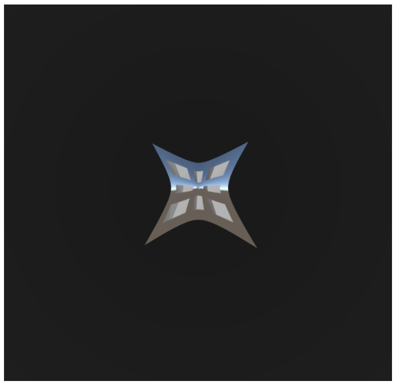
*ii. Unity Profiler out for 500,000 triangle mesh*

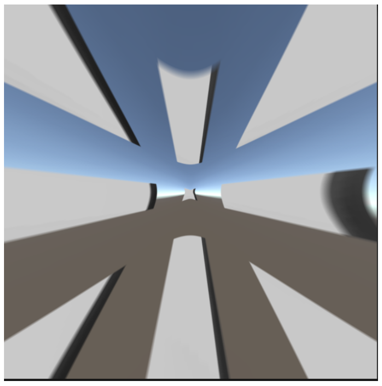


*iii. Notable decrease in rendering speed when the mesh is updated from 50 to 500,000 triangles*

*3.3) Inverting output using mesh-based method; comparison to original image*

The quality of the inverse radial distortion was quite poor; the resulting output did not closely correspond to the initial input image, particularly with small values of C1 and C2. Renders of a sample inverse distortion using the mesh-based method with a high-complexity mesh (500,000 triangles) is shown below; left: C1 = 0.2, C2 = 0.1, right: C1 = 1.1, C2 = 1:

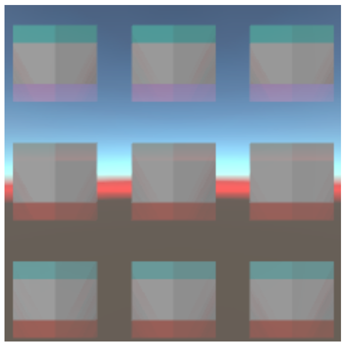
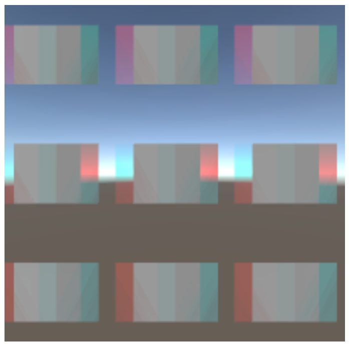




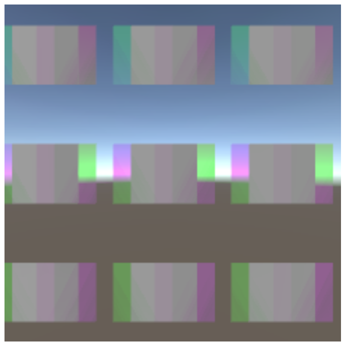
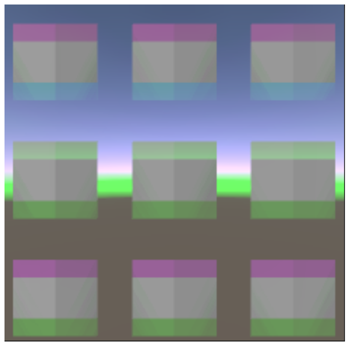
*4.1) Static renders*

a. Inverse chromatic aberration pre-correction

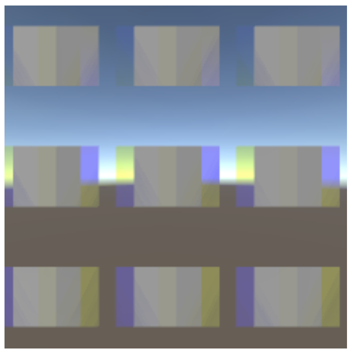
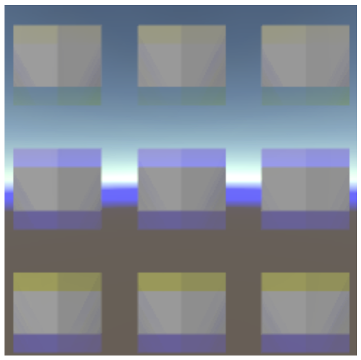
*i. LCA in the red channel; left: X offset = 0.05 , right: Y offset =  0.05*

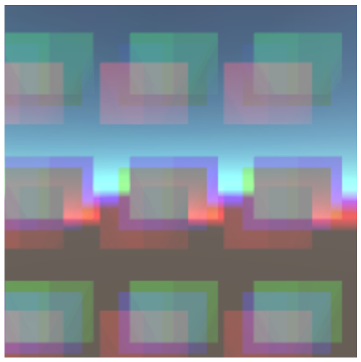
*ii. LCA in the green channel, left: X offset = 0.05 , right: Y offset = 0.05*

*iii. LCA in the blue channel, left: X offset = 0.05, right: Y offset = 0.05*

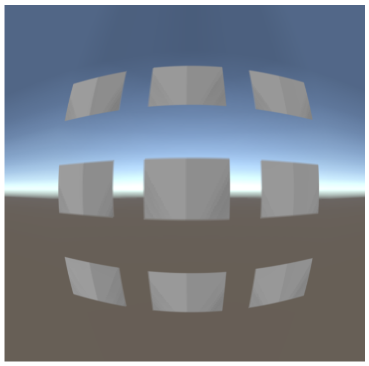
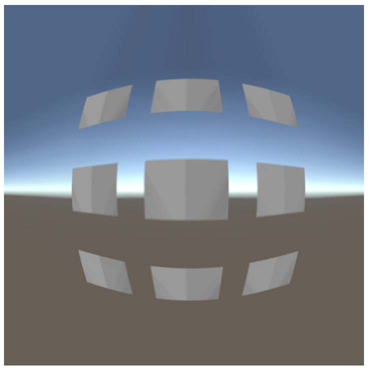
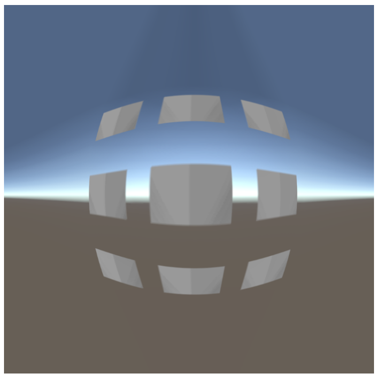



*iv. LCA in R,G, and B channels; R X, Y offset = 0.1, B X, Y offset = 0.02, G X, Y offset = 0.05*

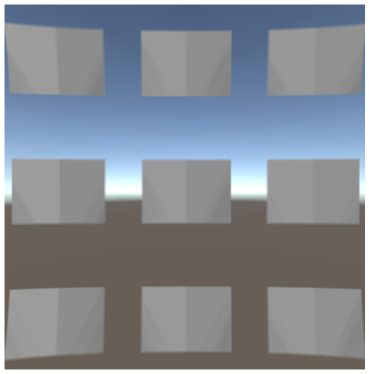
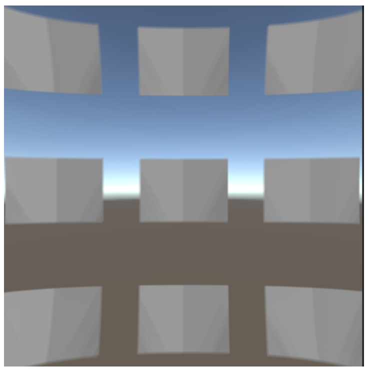
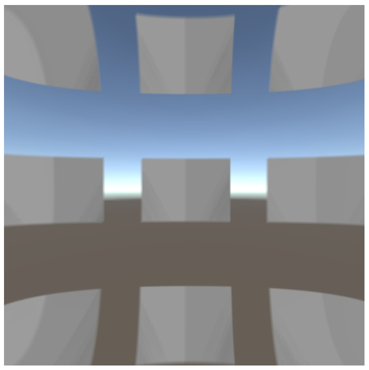


b. Barrel/pincushion distortion in the fragment shader

*i. Barrel distortion; left: C1 = -0.5, C2 = -0.25, center: C1 = -1, C2 = -0.5, right: C1 = -2, C2 = -1*

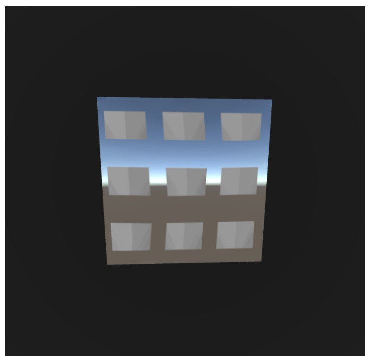
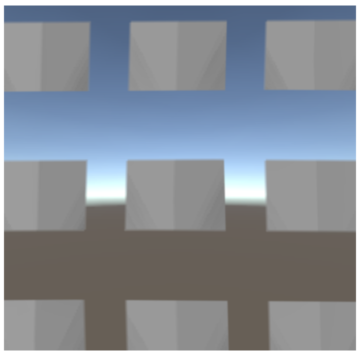




*ii. Pincushion distortion; left: C1 = 0.02, C2 = 0.01, center: C1 = 0.04, C2 = 0.02, right: C1 = 0.08, C2 = 0.04*

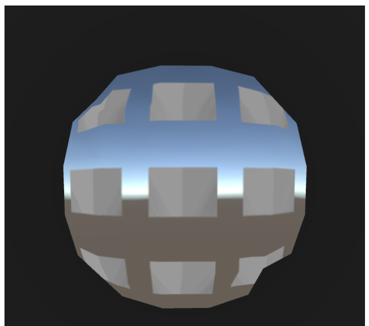
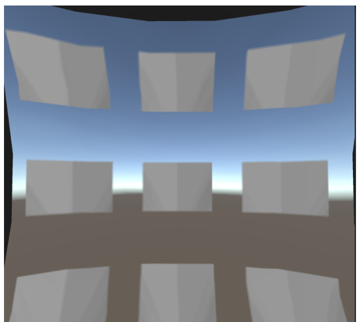




c. Barrel/pincushion distortion in the vertex shader using pre-calculated mesh

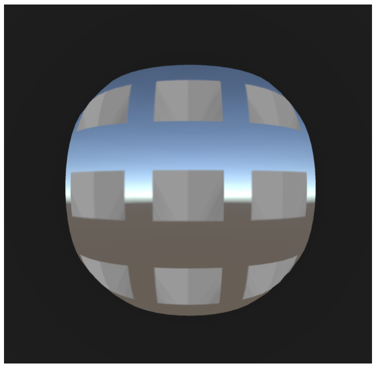
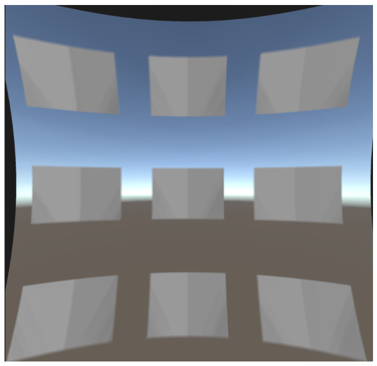
*i. 2 triangle mesh; left: barrel distortion, C1 = -0.2, C2 = -0.1, right: pincushion distortion, C1 = 0.2, C2 = 0.1*

*ii. 50 triangles; left: barrel distortion, C1 = -0.2, C2 = -0.1, right: pincushion distortion, C1 = 0.2, C2 = 0.1*

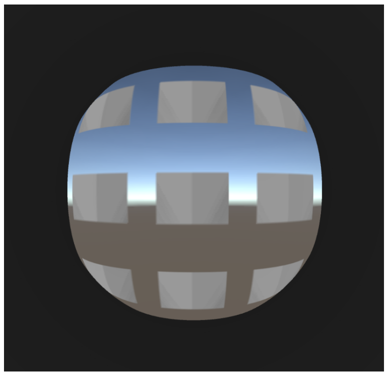
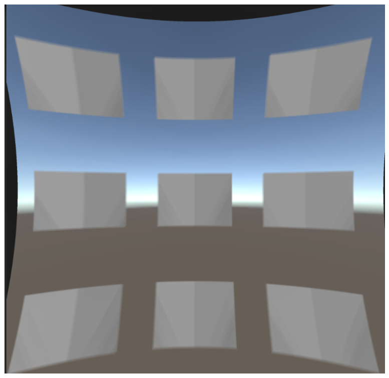



*iii. 800 triangles (~1000); left: barrel distortion, C1 = -0.2, C2 = -0.1, right: pincushion distortion, C1 = 0.2, C2 = 0.1*

I observed that, after ~1000 triangles in the mesh, the quality of the distortion began to plateau; see the below renders with a ~10000 triangle mesh, noting the minimal apparent difference in visual quality.

*iv. 9800 triangles (~10000); left: barrel distortion, C1 = -0.2, C2 = -0.1, right: pincushion distortion, C1 = 0.2, C2 = 0.1*

*4.2)*

The pixel-based (i.e. fragment shader) approach results in visually higher quality pre-distortion at the expense of a computationally expensive implementation; each pixel in the render texture is processed separately. Thus, for a square frame buffer with resolution 1024 x 1024 pixels, 1024*1024 = 1,048,576 position-update computations (one per pixel in the image) are required. A texture lookup is subsequently required for each pixel, resulting in 2 * 1,048,576 = 2,097,152 computations.
Conversely, the mesh-based (i.e. vertex shader) approach only distorts vertices of the mesh, rather than altering the position of every pixel comprising the render texture. The quality of the distortion scales with the complexity of the mesh; a heavily triangulated mesh will produce visually better results than a simple (e.g. 2-triangle) mesh, as only the mesh points are displaced using the distortion calculation and remaining pixels between the vertices are interpolated. Pre-distortion correction was implemented with meshes of varying complexity, including 2, 50, 162, 452, 800, 1600, 5000, 9800, 20000, 45000, 96800, and 500,000 triangles; visual quality improved significantly with increasingly complex (i.e. more triangulated) meshes, although the improvement in quality plateaued to the human eye at ~1000 triangles. One must compromise between output quality and rendering speed for the optimal mesh size. For a mesh containing n vertices, n distortion computations (one per vertex) are required; in the case of a triangulated mesh, where each triangle contains 3 vertices, 3*# of triangles computations are performed. Further, as the lens parameters remain constant while wearing the headset, the mesh can be pre-calculated (i.e. as the vertex positions remain constant, the mesh can be pre-transformed) at initialization time

and re-used at every frame. Thus, distortion implementations utilizing complex meshes with 10s or 100s of thousands of vertices are still much more efficient than a naive fragment-based implementation (e.g. 1,048,576 per-pixel calculations for 1024x1024 frame buffer, more as resolution increases). The computational savings increase with resolution size, as the fragment shader-based implementation must still perform one calculation per pixel, whereas the vertex shader-based implementation utilizes the same mesh pre-calculation, regardless of screen resolution. Texture look ups, however, will still be performed in the vertex shader-based implementation (once per-pixel per frame).

*4.3)*

Eye tracking refers to a “camera-based system” through which “eye rotations and blinking rates can be determined” (Lecture Notes ,Week 10). To implement eye tracking, distances between the pupil center and reflections from the cornea are computed; these distances change relative to the angle and gaze direction of the eye []. An invisible infrared light is used to generate reflections off of the cornea, while data recorded by a camera is used to compute tracking information. This generalized eye-tracking setup is modified slightly to work in the VR case, as the eyes do not exhibit vergence when a VR display is positioned immediately in front of the viewer. The notion of vergence suggests that normally, the viewer’s gaze converges at the object being looked at; each eye is directed, at an angle, towards some point at which the gaze meets. As the headset screen is located so close to the eyes, the vergence effect is cut short in VR and the captured 3D information does not convey the eyes’ gaze meeting at a single point. VR eye-tracking techniques must thus recover this missing data by using depth information available from other objects in the VR scene; using the direction of the eyes’ gaze, a virtual line can be traced into the scene to determine what particular object/point is being looked at.

Eye tracking can be of benefit to lens distortion pre-correction by accommodating real time lens correction based on pupil orientation. The current implementation of pincushion/barrel distortion in both the fragment and vertex shaders assumes a static pupil position in which the eye is looking through the lens head-on; the camera position remains static. However, in practice, as the user wearing the headset looks around the VR scene, rotations of the eye socket alter the pupil’s (i.e eye’s center) position, causing it to move both horizontally and vertically from its initial/“at rest” position. Thus, in the current setup, the distortion calculation is only computed at the pupil’s initial location; the implementation does not account for possible changes in pupil position and their resulting effect on distortion correction calculations. If the change in pupil position/orientation were to be drastic, it is possible that the computed distortion would appear unnatural to the user; the computed distortion correction is non-optimal when the eye orientation deviates from the initial, centrally fixed position. By incorporating eye tracking, one can keep track of updates to the pupil position at incremented timesteps, and repeatedly recompute or retrieve the distortion calculation using the most recent pupil position data; thus, an accurate lens distortion correction can be computed for various views.

Further, eye tracking can be used to implement foveated rendering in VR, “in which only those elements of the environment that are looked at are rendered”. As the human eye does not retain the same level of detail across the entire field of view (i.e. sharpness of vision decreases towards the periphery), the frequency of sampling can decrease towards the periphery of the lens. Doing so reduces the computational load required to render a scene, as rendering the complete VR environment is computationally expensive. By reducing the portion of the rendered scene, the required processing power is minimized and can otherwise be reallocated to other processes. In the case of pincushion and barrel distortion, implementing foveated rendering can facilitate high frequency sampling in the center of the FoV, while the periphery of the FoV can be sampled at a lower frequency to save shader resources. As lens distortion correction need not require the same level of detail at each area of the lens, performing distortion on foveated content can result in significant computation savings.

4.4)

Visually, barrel distortion introduces curvature in the image, causing image magnification towards the center of the FoV; straight lines appear to curve inwards, reminiscent of a barrel shape. Pincushion distortion also introduces curvature, however image magnification increases towards the edge of the FoV. Both barrel and pincushion distortion misplace image information geometrically. In terms of resolution and image quality, the image is likely to suffer a reduction in quality in areas where the distortion causes the image to expand and pixels are dispersed; in regions where the area of the distorted region is greater than that of the original image, one would need to interpolate pixel values, sampling from a smaller portion of the captured image to fill the expanded region. In areas where pixels are compressed to a smaller real estate, image information would be lost as the smaller region must accommodate pixels from the previously larger region. Interpolation could similarly be used (e.g. by averaging nearby pixel values) to compress pixels into the smaller region in a visually appealing/more natural way, as opposed to simply deleting arbitrary pixels such that the remaining number of pixels fits into the region. The loss of detail in the compressed and expanded regions of the distortion results in overall reduction of image quality.

If the barrel/pincushion distortion is inherent to the lens, post-processing techniques could be used to restore the original image. Otherwise, if the distortion is introduced for pre-correction purposes, the distortion can be performed at significantly higher resolution (i.e. in hardware) and rendered at a lower resolution such that the pixel data lost during distortion at a higher resolution is not noticeable/prevalent when rendered at a lower resolution. The image is first rendered at a higher resolution before distortion is introduced, essentially "oversampling" to capture image information at a higher level of detail; then, once distortion is performed and some information is lost, the distorted image can be rendered at a lower apparent resolution such that the loss of detail is not visible. By setting the rendering resolution lower than the hardware resolution, the visual impact of information loss caused by image magnification or compression can be minimized. Another possible solution, which was introduced earlier in Question 4.3, is foveated rendering; as the human eye naturally observes less detail towards the edge of the FoV and tends to focus on the center of the lens, loss of edge detail caused by distortion may not be a significant issue. Further, blurring of lens edges, which is commonly implemented in modern headsets, may further compromise edge details; thus, expending significant computational resources on high quality rendering of the edge pixels may be inefficient. Foveated rendering helps save computational resources by rendering certain parts of the output image at lower resolution; parts of the image that make up the edge of the eye's FoV are rendered at lower resolution, and central parts of the image (where the eye tends to focus) are rendered at higher resolution to account for high frequency details (i.e. multi-resolution shading). Those parts of the image which will be compressed by distortion can be rendered at a lower resolution, whereas those where detail is retained can be rendered at a higher quality. Implementing this strategy results in significant improvements in GPU performance without compromising the apparent visual quality of the distortion.