Assignment 4 – due 5/4/2017 at 11:59 PM
Topic: Build your HMD, Implement Stereo Rendering and Lens Distortion Correction

Students should use javascript and GLSL for this assignment, building on top of the provided starter code found on the course webpage. We recommend using the Chrome browser for debugging purposes (using the console). Other browsers might work too, but will not be supported by the teaching staff in labs, piazza, and office hours. Students may use the computers in Packard 001, or their own laptops, and desktops.

Homeworks can be submitted individually or in teams of two. Make sure you acknowledge your team members when submitting on Gradescope. You can change your teams throughout the term. Teams will share a computer and hardware (later on in the course) in the lab.

Please document all your answers, plots, and insights in a single pdf file containing all requested results. When you are ready to submit drag the entire homework folder into the browser with the render.html file loaded and running. This will zip up the relevant files for submission and place a zip file in the homework directory. Submit the zip file to Gradescope. Do not modify the submit.js file or you risk not submitting all required files.

Introduction

It's time to finally assemble the head mounted display! You'll be given some hardware during the lab and you'll put the (simple) system together. You'll extend the idea of anaglyph rendering from last week to stereo VR rendering, resulting in an even better 3D experience of your favorite teapot. The main take away you should have from this assignment is that for the best VR experience, the rendering/software and hardware have to be intertwined. You'll be using physical parameters to setup the view frusta, pre-warping to correct for optical distortions, etc.

HMD Details

The ViewMaster VR Kit housing that we use in this homework has the same specifications as Google Cardboard 1.0. You may need these numbers for your program.

- Focal length of lenses: 45 mm
- Lens diameter: 25 mm
- Interpupillary distance: 60 mm
- Distance between lenses and screen: 42 mm
- LCD screen width: 132.5 mm, height: 74.5 mm
- Eye relief: 10mm

When you put the screen into the HMD, make sure you align the line rendered on the center of the screen with the line on the bottom holder in the ViewMaster. This will be important when you implement lens distortion so that you now where the lenses are above the screen.

Task 1 (30 points)
The anaglyph rendering you implemented in HW3 assumed that each eye saw the same screen, and therefore to present different views to the eyes we filtered out certain color channels with the anaglyph glasses. Now, with the HMD, each eye actually sees a different portion of the screen, so we don’t have do do this color multiplexing to see in stereo. This way we can retain the full color information of the scene as well as providing a wider field of view with optics!

Implement stereo rendering on your HMD.

A. Given the focal length of the lenses in your HMD, the distance between lenses and LCD, and the eye relief, implement `computeDistanceScreenViewer()` and `computeLensMagnification()` in `displayParameters.js` to compute the distance of the virtual image from the viewer as well as the magnification factor. Report the computed values in your write-up. (5 points)

B. Using these values and the screen parameters reported above, we can implement stereo rendering. Fill in the `update()` function in `transform.js` to set the appropriate view matrix (using `computeViewTransform()`) and projection matrix (using `computePerspectiveTransform()`) for each eye. The off-axis frustum for the HMD is different from that in HW3 (see lecture slides), so make sure to update your calculations! For stereo rendering in this method, you only need two rendering passes, one for each eye, as opposed to the three for anaglyph rendering. That's because we don't need to color multiplex the two views, and can directly display them to the screen. (10 points)

C. Report the top, bottom, right, left, and near clipping plane parameters that are input into `computePerspectiveTransform()` for both eyes. These values are used in Task 1.B to compute the perspective matrices. (5 points)

D. Perform some simple perceptual experiments once the stereo rendering works and report your experience in a paragraph for each:
   a. Vary the interpupillary distance of the same 3D scene. Describe your experience and observations. (3 points)
   b. Experiment with the vergence accommodation conflict – how far can you move objects out of and into the screen while still being able to fuse the stereo images? Report min and max depth for you and your partner. (4 points)
   c. Try to describe the level of immersion in your own words. How does it compare to that of the stereo anaglyph experiment from HW3? Does the field of view affect your level of perceived immersion? (3 point)

Task 2 (35 points)

If you've implemented the stereo rendering correctly, you'll hopefully see a nice 3D teapot floating in front of you. If you look carefully however, you might notice that some things look a little distorted (lines you know are straight might not be straight anymore). However, if you look at the display itself, nothing looks distorted! The distortion comes from the lenses you see the screen through. Luckily, the distortion can be nicely modeled by the Brown-Conrady model you learned in class. Using this model, we can pre-distort the image we display on the screen so that when the light passes through the lenses and gets distorted by the lenses, straight lines are straight again. You'll implement a fragment shader based version of lens distortion correction in this homework in `fShaderUnwarp.js`. Because we need to post-process the scene that we initially render, we need to add an extra rendering pass for each eye. The first two rendering passes render the left and right viewports, and the second two perform the lens undistortion. You might find it useful to quickly swap between the standard rendering mode and the unwarped mode by pressing the “1” button on the keyboard or click the button on the top.
A. Calculate the center of the lens distortion as discussed in class. Implement `updateUniforms()` in `stereoUnwarpRenderer.js` to compute the values of the lens distortion centers in normalized texture coordinates (between 0 and 1) for each eye. Report the values for the given display in your HW submission. The values are passed as uniforms into `fShaderUnwarp.js`. (5 points)

B. In `fShaderUnwarp.js` implement the undistortion warp. The lens distortion parameters are stored in the uniform K, where the first element corresponds to the \( K_1 \) and the second corresponds to \( K_2 \) in the lens distortion equations in lecture 7. Apply the lens distortion formula we discussed in class to compute the distorted texture lookup coordinates. Use these distorted coordinates to do the texture lookup. Unfortunately, WebGL 1.0 doesn't support the boundary condition where the values outside of \([0, 1)\) is a user-defined constant value. To achieve this effect with your fragment shader, assign black color to the fragments if the lookup coordinate is outside of \([0, 1)\).

Keep in mind that the distortion is symmetric around the center of distortion. This means that you will need to center your texture coordinates around the center of the lens, already defined in \( u,v \) coordinates in the `centerCoordinate` variable. HINT: Think whether \( u,v \) space is appropriate when computing the radial distance (15 points)

C. Make the distortion parameters user-defined parameters. You should be able to decrease/increase \( K_1 \) with the "2"/"3" keys and decrease/increase \( K_2 \) with the "4"/"5" keys. This functionality is already implemented for you. You can see the updated values displayed at the top of your screen. By pressing the space bar you can switch to a scene where a grid pattern is displayed to make it easier to see the distortion effects.

Manually adjust your distortion correction parameters \( K_1 \) and \( K_2 \) until the lines are as straight as they get throughout your entire field of view. Report the values \((K_1, K_2)\) that worked best. Submit 1 screenshot as well as 1 photo (i.e. captured with your cellphone) through the lens that show the grid lines before and after distortion correction each. That’s 4 images in total you should submit. Even though multiple set of the distortion parameters may show equally good results, you can report one set of the best parameters. We will grade based on the photos and your codes. (10 points)

D. Implementing the lens distortion correction in the fragment shader is common, and warps every single pixel correctly. However, because the lens distortion doesn’t change we can pre-compute the warp to save some computation time. One way to do this is to warp the mesh onto which we “paste” the texture. Instead of pasting it on a rectangular mesh that fills the viewport, what if we pasted it on a pre-distorted mesh? The distortion correction would be inherently performed in the texture lookup. Is every single pixel warped correctly? What are some benefits / drawbacks of implementing a warp this way? How would increasing the number of vertices in this mesh affect the benefits/drawbacks? (5 points)

**Task 3: Keyboard Navigation** (20 points)
As we make baby steps in creating a true VR system, let’s add another component: navigation. With the simple keyboard-based navigation technique you’ll be able to fly through whatever virtual environment you
end up building.

A. First, as we did in HW1, we need to record keyboard inputs. In stateController.js fill the in the keydown() function to record key presses into the member variables viewerPosition and viewerRotation. The viewer position is modified by the WASD keys (W – forward motion, A – leftward motion, S – backward motion, D – rightward motion) and for every time unit that the key is pressed, move along an axis by 5 units. Your viewer rotation (defined in Euler angles) is modified by the SHIFT + WASD keys (“SHIFT + W”: +x rotation, “SHIFT + A”: +y rotation, “SHIFT + S”: -x rotation, “SHIFT + D”: -y rotation) and for every time unit that the key is pressed, update the rotation by 0.5°. Specifically, when the SHIFT+W key is pressed, we expect the view to pan upwards. When SHIFT-A is pressed we expect to pan to the left. (10 points)

B. Now, for the actual flying part. We can easily do this by modifying the view matrix. Implement computeViewTransform() in transform.js to compute the view matrix from the viewer's position and rotation that you updated in stateController.js. (10 points)

Questions?

First, Google it! It’s a good habit to use the internet to answer your question. For 99% of all question, the answer is easier found online than asking us. If you can’t figure it out this way, post on piazza and definitely make sure you attend the lab on Fridays (in Packard 001).