Homework 3: Foveated Rendering, Depth of Field, Anaglyph Stereo Rendering

EE267 Virtual Reality 2022

Due: 04/21/2022, 11:59pm

Instruction

Students should use JavaScript 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 and built-in debugger). Make sure hardware acceleration is turned on in the advanced settings in Chrome. Other browsers might work too, but will not be supported by the teaching staff in labs, piazza, and office hours.

The theoretical part of this homework is to be done individually, while the programming part can be worked on in groups of up to two. If you work in a group, make sure to acknowledge your team member when submitting on Gradescope. You can change your teams from assignment to assignment. Teams will share handed-out hardware (later on in the course).

Homeworks are to be submitted on Gradescope (sign-up code: **5V8PPE**). You will be asked to submit both a PDF containing all of your answers, plots, and insights in a **single PDF** as well as a zip of your code (more on this later). The code can be submitted as a group on Gradescope, but each student must submit their own PDF. Submit the PDF to the Gradescope submission titled *Homework 3: Foveated Rendering, Depth of Field, Anaglyph Stereo Rendering* and the zip of the code to *Homework 3: Code Submission*. For grading purposes, we include placeholders for the coding questions in the PDF submission; select any page of the submission for these.

When zipping your code, make sure to zip up the directory for the current homework. For example, if zipping up your code for Homework 1, find `render.html`, go up one directory level, and then zip up the entire `homework1` directory. Your submission should only have the files that were provided to you. Do not modify the names or locations of the files in the directory in any way.

Please complete this week's lab and watch the video before you start to work on the programming part of this homework.

1 Theoretical Part

(i) **(Vergence)** Vergence refers to an oculomotor process in the human visual system where the eyeballs rotate in their sockets as we fixate on objects at different depths. The purpose of vergence is to keep the fixated on object on the foveas of both eyes. To achieve this, the eyes will be almost parallel when we fixate on an object that is far away, and they rotate inwards when we fixate on an object close by.

In Figure 1, the eyes fixate on $\mathbf{p_1}$ but there is also another point $\mathbf{p_2}$ in the scene that the eyes could alternatively fixate.

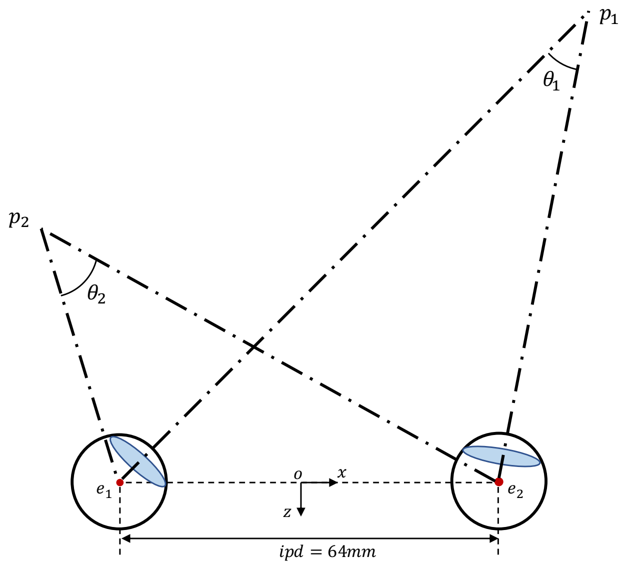

Figure 1: *An illustration of the eyes fixating on point $p_{1,2}$, resulting in vergence angle $\theta_{1,2}$. The origin, o, is placed at the midpoint of the two eyes.*

(a) Given $\mathbf{p_1} = (200, -500)$ mm, $\mathbf{p_2} = (-200, -250)$ mm, and an interpupillary distance of $ipd = 64$ mm, compute the vergence angles θ_1 and θ_2, which correspond to the eyes fixating on $\mathbf{p_1}$ and $\mathbf{p_2}$, respectively. For this question, we neglect the finite diameter of the eyeballs and only consider their centers. (*5pts*)

(b) Derive a parametric representation (i.e., a formula) that models the set of points $\mathbf{p_i} = (x_i, z_i)$ with the same vergence angle as p_1. Express the relation as a function of x_i (i.e., $z_i = f(x_i)$). Plot this curve and label the coordinates of: the location of the eyes, the points on the curve that are farthest to the left and right, and the point farthest from the eyes in the z direction. (*5pts*)

Hint: This is called the horopter, look it up online for more information. You might find the following trigonometric identity useful: $\tan^{-1}(x) + \tan^{-1}(y) = \tan^{-1}(\frac{x+y}{1-xy})$. You may need to re-derive part (a) to get your expression in this form.

(ii) **(Retinal Blur)** Accommodation is another oculomotor process. Here, the eye changes its focal power by deforming the crystalline lens via the ciliary muscles. This mechanism ensures that the fixated object is focused on the retina. Inevitably, this means that objects at other distances appear blurred. The accommodation state of both eyes is usually linked. Given an eye accommodated at point $\mathbf{p_3}$ and given the following parameters:

- diameter of the eye D_e = 24 mm
- pupil size S_e = 5mm
- distance from $\mathbf{p_3}$ to lens d_3 = 1000 mm
- distance from $\mathbf{p_4}$ to lens d_4 = 500 mm

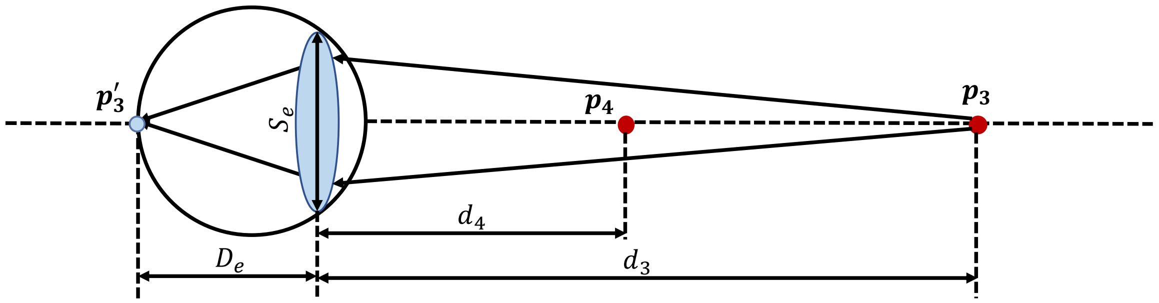

Figure 2: *The eye is accommodated at a distance d_3 focusing on point p_3. Points that are closer or farther away will be blurred on the retina. The diameter of the blur circle on the retina is called circle of confusion.*

(a) Compute the focal length of the lens using the Gaussian thin lens equation. (*5pts*)

(b) Assuming the eye is accommodating at point $\mathbf{p_3}$, compute the diameter of the blurred point $\mathbf{p_4}'$ on the retina (i.e., the circle of confusion) in mm. (*5pts*)

(iii) **(Visual Acuity)** In vision science, the size of the retinal projection of an object is usually defined as the visual angle it subtends (see Fig. 3). An object's visual angle, measured in degrees, can be calculated via the equation listed in Figure 3.

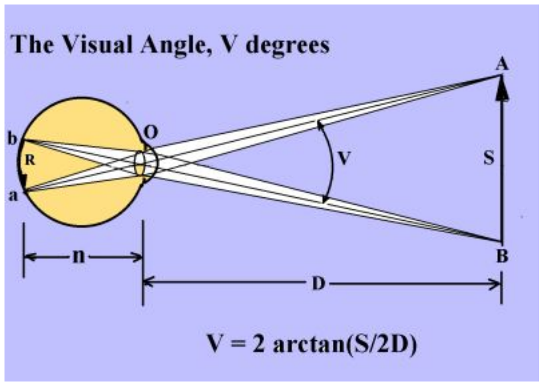

Figure 3: *To model the size of the retinal image of an object, we usually use degrees of visual angle as a metric.*

Visual acuity, or sharpness of vision, can be defined as the number of "cycles" from bright to dark that are perceivable within one degree of visual angle; this is called cycles per degree or cpd. You can think of this as the number of line pairs (pairs of black and white lines) one can distinguish clearly in one degree of visual angle. A related concept to visual acuity is the minimum angle of resolution, or MAR, which is the smallest angle between two points that can be resolved. The MAR, given in degrees (per cycle), is the reciprocal of acuity. On average, humans resolve around 30 cpd (i.e., 20/20 vision), although some can distinguish as many as 40–50 cpd. For reference, most VR displays today only support an acuity of around 5 cpd.

Apple claims that its retina displays have a pixel density higher than what a human can perceive. Is that true? Let's take a look at the screen in a 3rd generation 13.3" MacBook Pro. It has a resolution of 2560 × 1600 pixels. Assuming square pixels and a viewing distance of 50 cm, what is the visual angle of one pixel? What is the maximum acuity (in cpd) that the display can support at this distance? Is it higher or lower than the 30 cpd that humans can perceive? (*5pts*)

(iv) **(Eccentricity and Visual Acuity)** As discussed in class, the distribution of photoreceptors on the retina is not uniform. The density of the cones is much higher in the fovea than in the periphery of the visual field, which results in visual acuity decreasing rapidly away from the fovea. It turns out that a simple linear model is quite accurate in modeling this falloff. The linear model matches both anatomical data (e.g., photoreceptor density) as well as performance on low-level vision tasks. This model is defined as

$$\omega = m\theta_e + \omega_0$$

where ω is the MAR in degrees (per cycle), θ_e is the eccentricity angle in degrees, ω_0 is the smallest resolvable angle in degrees (per cycle), and m is the MAR slope. For this question, you may assume $m = 0.0275$, $\omega_0 = \frac{1}{48}^\circ$.

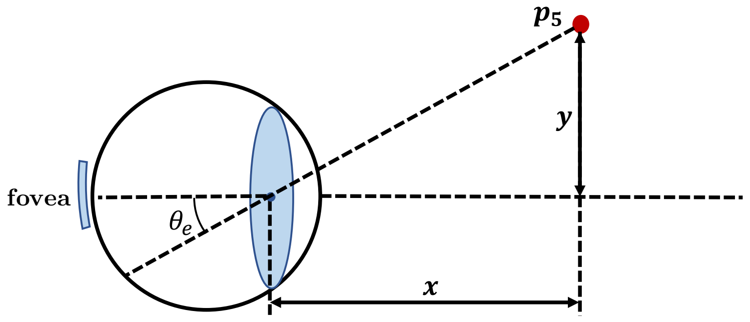

Figure 4: *The distance of a point's retinal projection to the fovea is called eccentricity, here denoted as θ_e, and measured in degrees.*

(a) Given a point $\mathbf{p}_5$ at the location $x = 400$ mm and $y = 300$ mm, what is the eccentricity of this point on the retina when the eye looks down the x axis? (*5pts*)

(b) What is the highest frequency (in cpd) that one would be able to resolve at that eccentricity? (*5pts*)

Programming Part PDF Deliverables

The following questions in the programming part ask you to provide written responses:

- 2.4.4 Anaglyph Perceptual Question (*5pts*)

Make sure to append your responses to the end of the PDF submission.

2 Programming Part

In this week's homework we will render our first 3D teapot via anaglyph stereo! Before that we will implement and experiment with a few other important concepts of VR: foveated rendering and depth-of-field rendering.

The purpose of foveated rendering is to save rendering time by taking advantage of the non-uniform distribution of photoreceptors on the retina. Gaze tracking is required to estimate where the user fixates on the screen. With the known gaze position on the screen, we render the region around this position at the highest possible display resolution, but then gradually decrease the resolution at farther distances without the user noticing.

Depth-of-field rendering will try to make a rendered image look more realistic by simulating the limited depth of field of the eye. For this technique, we need to know at what distance the eye is accommodated; we simply use the distance of the fixated object as the accommodation distance. Further, we need to blur every object in the image in a depth-dependent manner, such that the size of the blur kernel depends on the distance of that object to the accommodation distance of the user. Unfortunately, this technique does not save computation, it actually adds to it, but it may create a more realistic and immersive experience.

Finally, stereo rendering allows you to see a 3D image with the use of anaglyph glasses.

Note that in the starter code, you can flip between these rendering modes by pressing the button at the top left corner, or by pressing the 1–4 buttons on your keyboard. It will make it easier to see the difference between the modes.

Tip: You'll be implementing several shaders in GLSL, which can be difficult to debug. You can't use `print` statements, but you can instead use `gl_FragColor` to serve a similar purpose. For example, you can set `gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0)` (red) to check if an `if` statement evaluated to true. You can also set `gl_FragColor.r` to the output of `distToFrag()` to see if the distances make sense. Remember that any values will be clamped to the [0,1] range, so rescale the distance first; e.g., divide by 2000 to make "red" mean ≥2 m away.

2.1 Setting up your display parameters

Each of the rendering methods we will implement depend on your specific monitor. Therefore, you need to update `screenDiagonal` in `render.js` to the value of the monitor that you are using for this homework. You need to use a ruler or a measuring tape to determine the exact dimensions of your monitor (or better: look up the specs listed by the manufacturer)!

2.2 Foveated Rendering

As mentioned above, the primary goal of foveated rendering is to save computation time by taking advantage of the non-uniform distribution of photoreceptors on our retina. The falloff of visual acuity with eccentricity is shown in Figure 5. At a given eccentricity, the minimum angle of resolution is given by the green line (see equation above). Therefore, one would not be able to resolve objects that subtend a smaller visual angle than this MAR at a particular eccentricity. Similarly, for a given visual angle or MAR, we can calculate the eccentricity angle at and after which this angle is smaller than or equal to the smallest resolvable feature size. You can see that the farther from the fovea we are, the lower the image resolution can be without the user noticing.

To minimize computational cost, we would like to follow the green line as best as possible with our rendering strategy. Rendering features with visual angles far below this line means that we are rendering extra resolution that users will not be able to perceive. In current graphics accelerators (GPUs), it is not straightforward how to implement this

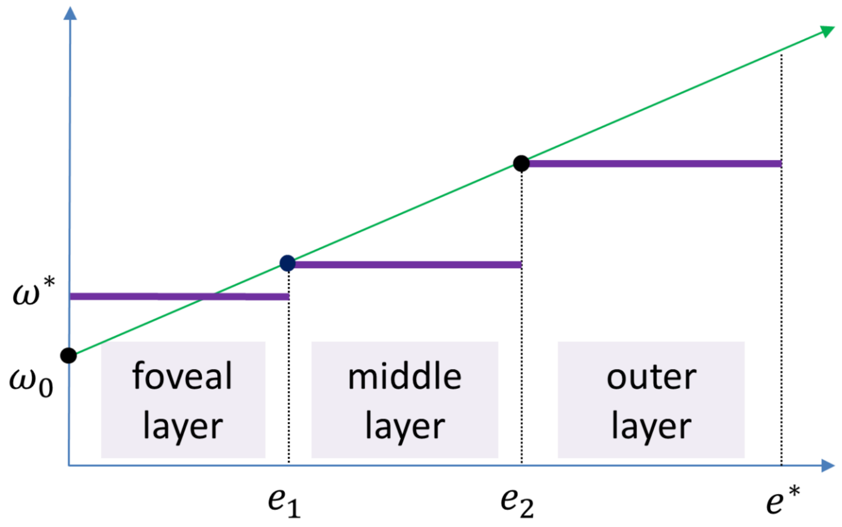

Figure 5: *The minimum angle of resolution (MAR) ω increases linearly with increasing eccentricity, i.e., distance from the fovea.*

continuously varying resolution efficiently. Thus, we follow the foveated rendering scheme proposed by Guenter et al. and divide the visual field into three discrete layers: the *foveal layer* that has the highest resolution supported by the screen, the *middle layer* which is rendered at a slightly lower resolution, and the *outer layer* which is rendered at the lowest resolution. At each layer, we support a minimum feature size that is equal to or smaller than what we can perceive. If implemented correctly, we should not see a difference between the foveated image and a full-resolution image.

We will implement a somewhat inefficient variant of foveated rendering in the GLSL fragment shader `fShaderFoveated.js`. Our variant is mostly educational in that it teaches you the basic idea of foveated rendering without actually reducing computational cost. In fact, our implementation actually increases it but that is okay for now.

For our foveated rendering implementation, we will use a computer graphics technique called multi-pass rendering. In the first rendering pass, we render an image not on the screen but directly into a chunk of memory on the GPU that can later be accessed as a texture. This chunk of memory is called *framebuffer object* (FBO) and in Three.js the FBO is wrapped in `WebGLRenderTarget`. In this first rendering pass, we use the Phong shaders from HW2 defined in `fShaderMultiPhong.js` and `vShaderMultiPhong.js`. The foveation is implemented in the second rendering pass, where we blur the sharp image produced in the first rendering pass according to our MAR calculations. We will use separate vertex and fragment shaders for this rendering pass, which implements a spatially varying blur of the image.

The shader for the second rendering pass is defined in `fShaderFoveated.js` and `vShaderFoveated.js`. Your starter code and this shader already implement the double-pass rendering; the FBO generated in the first rendering pass is passed into the second pass as a texture by rendering a rectangle that fills the entire screen (vertex coordinates [−1,1,0], [1,1,0], [−1,−1,0], [1,-1,0]) and texture coordinates [0,0], [1,0], [1,1], [0,1]). The shader then copies the texture from the first rendering pass to the output with:

```
gl_FragColor = texture2D( textureMap,  textureCoords );
```

The image from the first rendering pass is stored in `textureMap`, which can be indexed using the `texture2D()` function and the interpolated texture coordinates from the rectangle stored in `textureCoords`. Remember that texture the coordinates u, v are normalized to the range [0, 1], with the origin in the bottom left corner. To index into pixel (500, 600) of a texture whose initial size was 800×1000 pixels, you would need to use the texture coordinate (500/800, 600/1000) = (0.625, 0.6). It is important for you to understand this seemingly simple pass-through shader before moving onto foveation. Try and play around with it a bit to get some more intuition before moving on.

Remember that the second rendering pass implements a spatially varying image blur based on the user's fixation point on the screen and the distance of a pixel to that point. Ideally, the fixation point would be determined by an eye tracker, but for the purpose of this homework we will use the mouse pointer to control the gaze position represented by a black dot in a browser. You can move the gaze position by dragging the mouse while pressing the shift key on the keyboard.

2.2.1 Visual Angle of a Pixel (*5pts*)

You will first have to determine the visual angle that one pixel on your monitor covers. Implement the `computePixelVA()` function in `foveatedRender.js` (should be 1 line of code). The output, i.e. visual angle per pixel in degrees, is stored in the `pixelVA` member variable. We will test that this was implemented correctly with our unit checker code.

2.2.2 Eccentricity for some MAR (*5pts*)

We can now determine the regions of the foveated blur, defined by eccentricity angles e_1 and e_2 (see Fig. 5). Implement the `computeEcc()` function in `foveatedRenderer.js`, which takes in the MAR you would like support and returns the eccentricity (in degrees) at which that resolution becomes imperceptible. The equation to compute the MAR is provided in the lecture slides (use $m = 0.0275$ and $w_0 = \frac{1}{48}$). This function will be checked for correct implementation in our unit checker code.

2.2.3 Foveation Zones (*5pts*)

We can choose the MAR values to be whatever we wish. In general, we would choose them such that the pixel savings would be greatest, but for this task we will choose them such that the middle layer will have 1/4 the resolution of the foveal layer, and the outer layer will have 1/8 the resolution of the foveal layer. Fill in the arguments to `computeEcc()` in the `foveatedRenderer.js`, which assigns `e1` and `e2` based on the above description. **Remember that the MAR at 0 degrees eccentricity is two times the visual angle of one pixel, which is the inverse of the maximum cycles per degree of the display.**

2.2.4 Fragment Shader Foveation Blur (*10pts*)

Now that all of the high-level parameters are set, you will need to implement the blurring itself. Apply a 2D Gaussian blur to the middle and outer regions with the 1D blur kernels provided in the shader (`middleBlurKernel` and `outerBlurKernel`). Remember, that a 2D Gaussian filter is separable and you can construct a 2D blur kernel by taking the outer product of a 1D Gaussian blur kernel with itself. These Gaussian kernels correspond to roughly reducing the maximum frequency by a factor of 1/4 and 1/8 respectively. When figuring out the foveation regions in the shader you can use the small angle approximation to determine how far a given fragment is from the gaze position (i.e., assume that the total visual angle is linearly proportional to the number of pixels). The distance to the screen is not needed with this approach.

You can implement the blur at each fragment with nested `for` loops. When doing so, you want to index into pixels of the FBO/texture of the first pass around the current fragment position by varying the texture coordinate at each step of the `for` loops. Doing a lookup into the next fragment over to the left, for example, would correspond to subtracting by 1/(window width or height in pixels).

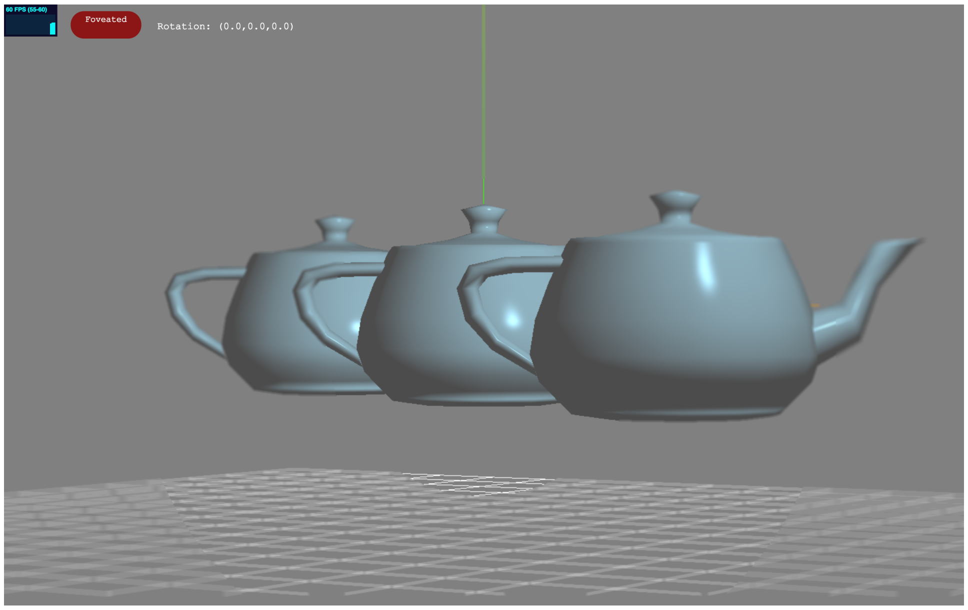

Figure 6: *Foveated rendering. Here is something you might see when you implement the foveated rendering. Please note that this is exaggerated, and the actual blurs you will implement will be much more subtle. This is just so you can see the effect of the of the different regions in this document.*

2.3 Depth-of-Field Rendering

The next task is depth-of-field rendering, where instead of trying to save computation we attempt to make things more realistic. We will again use a multi-pass rendering approach where the first pass saves the image and depth maps into an FBO and the second pass blurs that image. This time, the blur depends on the depth of the fragment and the accommodation state of the user's eye. You can find this new shader in `fShaderDof.js` and `vShaderDof.js`.

2.3.1 Distance to Fragment (*5pts*)

Your first task will be to determine the depth of each fragment (i.e., the Euclidean distance to camera position in view space). Remember that the depth map is normalized (between 0 and 1) and does not correspond to distances. Thus, we have to invert the projection transform (similar to Q1 of HW2). For this, you will need the projection matrix, the inverse projection matrix, and the fragment's (normalized) depth buffer value. You can access the depth buffer value of a fragment by indexing into the `depthMap` in the fragment shader the same way we can access the color information from `textureMap`; use the first index of the texture (i.e., `[0]` or `.x`) because not all hardware returns anything in the other indices. Refer to Q1 of HW2 for how to perform this computation. Implement this conversion in the `distToFrag()` function in `fShaderDof.js`. The argument, `p`, to the function is the texture coordinate of the fragment in question.

2.3.2 Computing the Circle of Confusion (*5pts*)

Knowing the accommodation distance given by the depth value of the fragment below the gaze position and the depth of the other fragments, we can now estimate the amount of blur each fragment would exhibit. Update the function `computeCoC()` to calculate the circle of confusion (diameter of blur circle) exhibited *at the accommodation distance*, not the retina (i.e., pretend the screen is at the accommodation distance and compute the circle of confusion for each fragment relative to that distance)! You need the fragment depth, accommodation distance, and pupil diameter. All of these values are in mm. **Hint:** use similar triangles to compute the circle of confusion.

2.3.3 Retinal Blur (*10pts*)

Now you have everything you need to implement the blur itself. Implement it with a double `for` loop in the `computeBlur()` function. The loop will average all the neighboring pixels that fall into the circle of confusion of that particular fragment. Loop over a maximum search radius of 11 pixels. In that search, average only over the pixels that fall within the circle of confusion in pixels. Remember that the u and v texture coordinates are normalized to [0, 1]. To move one pixel to the left from the current texture coordinate, you will have to move by 1/(window width or height in pixels) in texture coordinate units. Return the color value of the blurred fragment from `computeBlur()` and assign it to `gl_FragColor`. This will be the value displayed on the screen.

2.4 Anaglyph Rendering (*25pts*)

Rendering a 2D teapot can get a little boring, especially if that's all you have been seeing for these last 2 weeks. So let's make it pop out of the screen with anaglyph stereo rendering! You will see your first 3D teapot after completing this task.

This time, you will need three rendering passes to get the right image displayed on the screen. You can find these passes in the `animate()` function in `render.js`. The first pass will render the scene from the viewpoint of the left eye into an FBO (using the Phong shader). The second pass will render the scene from the viewpoint of the right eye into a different FBO (using the Phong shader). The third rendering pass will access the textures from these two FBOs, convert the respective color values to grayscale, and assign the images to the color channels of the anaglyph glasses. We have already implemented all of the nitty gritty stuff to make the multiple rendering passes work but will require you to implement key elements – the view and projection matrices for the left and right eyes, and the fragment shader that will combine the two views in the third rendering pass.

When you first run the starter code in the anaglyph mode you won't see anything interesting happening. That's because we render the same image to each FBO and just did a simple pass-through of one of the viewpoints to the screen. Your first tasks will be to generate the correct view and projection matrices in the `update()` function in `transform.js`. These matrices are the ones that will be used to compute the renderings from the left and right viewpoints of the scene that will be saved in the FBOs.

2.4.1 View Matrix Computation (*5pts*)

Modify the `computeViewTransform()` function in `transform.js`, which you implemented in HW1, to correctly compute the view matrices for the left and right eyes for anaglyph rendering. In addition to `StateController.state`, the input of this function has `halfIpdShift`, whose absolute value is the half of the `interpupillary` distance (default value: 64 mm). `halfIpdShift` is positive for a left eye and negative for a right eye.

2.4.2 Projection Matrix Computation (*5pts*)

Compute the projection matrices for the left and right eyes in `update()` and assign them to the `anaglyphProjectionMat` member variable. In the starter code, `computePerspectiveTransform()` is implemented for you by using `THREE.Matrix4().makePerspective()`, which is equivalent to the one you implemented in HW1. Feel free to use this function to compute the projection matrices. Essentially, you need to compute `left, right, top, bottom` for each eye based on the physical screen parameters (pixel pitch, window resolution, distance from the viewer to the screen). These parameters are stored in `dispParams`, which is an instance of the class `DisplayParameters`.

2.4.3 Color-Channel Multiplexing (*10pts*)

Once the stereo images are rendered, you will need to combine them in such a way that each eye sees a different image when viewing the screen through anaglyph glasses. You will do this in the fragment shader defined in the `fShaderAnaglyph.js` file. There are different ways of combining the color channels, as discussed in class, but for this homework you will use the method of converting each RGB image to grayscale first, and then assigning the grayscale left image to the red color channel of the output and the grayscale right image to the green and blue color channels of the output. For the conversion to grayscale, you can use the formula: gray = 0.2989 × red + 0.5870 × green + 0.1140 × blue. Welcome to your first 3D teapot experience!

2.4.4 Anaglyph Perceptual Question (*5pts*)

A different way of doing anaglyph rendering is by assigning the red color channel of the left image directly to the red color channel of the output, and the green and blue color channels of the right image directly to the green and blue color channels of the output. One advantage of this method is potentially gaining back some color lost by converting the images to grayscale. However, there is a drawback to such a method. What is it? Think of some scenes, specifically how the colors might impact the output. Provide your responses in the PDF with your solutions to the theoretical part.

Questions?

First, Google it! It is a good habit to use the Internet to answer your question. For 99% of all your questions, the answer is easier found online than asking us. If you cannot figure it out this way, post on piazza or come to office hours.