Homework 4: Perception of Motion and HMD Software Simulation

CS 5360/6360: Virtual Reality

Due: 11/15/2021, 11:59pm

Instruction

Students should use the Arduino environment and 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.

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 can share hardware (needed for this homework).

Homeworks are to be submitted on Gradescope. 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 4: Perception of Motion and HMD Software Simulation* and the zip of the code to *Homework 4: 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.

Getting Started

Task 1: Tutorial (Refresher) on Partial Derivatives and Taylor Series Approximation

For those of you not familiar with partial derivatives and/or Taylor series approximation—how to calculate them, what they conceptually mean—you may find it helpful to go over these short tutorials:

- Partial Derivatives by Khan Academy: https://www.youtube.com/watch?v=AXqhWeUEtQU.
- Taylor series by 3Blue1Brown: https://www.youtube.com/watch?v=3d6DsjIBzJ4

This will be helpful for the theoretical part of the homework.

Task 2: Optic Flow Tutorial

As a complement to the class discussion on Optic Flow, you may find the following tutorial on estimating optic flow helpful: http://www.cs.toronto.edu/~fleet/research/Papers/flowChapter05.pdf. It is from the perspective of researchers in Computer Vision.

Task 3: Obtain your HMD Parts

By this time, you should have obtained the casing you will use to house your HMD (with corresponding lenses) as well as the LCD screen you will eventually render to. If not, be sure to obtain them as soon as you can: you will need them for both the theoretical and coding parts of the homework.

Task 4: Getting started with Arduino and Teensy

We briefly discussed Arduino in class, but you may want to read the brief "official" introduction. The Arduino website should be your go-to place for all questions related to Arduino. Also read the information on Getting Started with Arduino.

The VRduino uses a Teensy microcontroller. This Teensy is an Arduino-compatible board, but it is a bit more beefy than the official Arduino boards. The Teensy 3.2 uses an ARM Cortex-M4 processor that runs at up to 72 MHz and it is also smaller than the Arduino UNO, for example. The official Teensy website is https://www.pjrc.com/teensy/.

If this is the first time you use a Teensy, install Teensyduino by following the installation instruction page. This is a stand-alone app or a plug-in for the Arduino IDE app, which lets you compile your code for the Teensy. Depending on your operating system, you may need to reinstall the Arduino IDE app if you have already installed one on your computer. The Teensy First Use article will also help you get started.

Due to the fact that the Teensy is Arduino compatible, you can program it in pretty much the same as any other Arduino and benefit from the Arduino IDE and strong community support on the internet. For example, to upload the starter code to the IDE, open vrduino.ino with the Teensyduino/Arduino app. Connect your VRduino to your computer via the USB port. Under Tools, change the port in the IDE to the new serial Teensy port and make sure that the Board is set to Teensy 3.1/3.2. Click the right arrow on the IDE to compile and upload the program to Teensy. Here is a brief tutorial on How to Setup Teensyduino.

You are free to edit your code within the Arduino IDE, but if you find the editor lacking (e.g. lack of syntax highlighting), you can use your favorite editor to edit the code, and use the IDE to compile/upload the program. You can do this by checking off "Use External editor" in the settings/preferences.

Task 5: Practice your Arduino skills

If you feel confident about your C programming skills and the Teensy, go ahead and get started on the homework. If you'd like some more info on how to program Arduinos, you can find a lot of Arduino tutorials online. For example, these Arduino Video Tutorials for Beginners are very helpful.

1 Theoretical Part

1.1 Designing (our HMD) for the Human Eye (*5pts*)

In class, we discussed the calculations needed in order to correctly render stereoscopic images on a Head-mounted Display. We needed to calculate the following parameters: the magnification factor M, the odd (w_1) and even (w_2) widths of the virtual image formed for each eye, and the viewing frustrum coordinates `top`, `bottom`, `left`, `right` for each eye. In order to perform these calculations, you must identify the following specifications for your particular HMD casing (all units in mm):

- Focal length of lenses:
- Lens diameter:
- Interpupillary distance (IPD):
- Distance between lenses and screen:
- LCD screen width:
- LCD screen height:
- Eye relief:

Please identify all the above specifications and calculate the parameters mentioned above. (*5pts*)

1.2 Optic Flow Estimation and Sensor Fusion (*35pts*)

In class, we discussed *optic flow*—the change in light from objects in motion, as detected by an external sensor. For people, the sensor is the retina. Figure 1 has been presented in class, and illustrates this process.

When we discussed optic flow, we discussed individual photoreceptors as having a "time-dependent position" p, represented as a function $p(t)$. In reality, photoreceptors have a *fixed* position on the retina. What *does* change in a single photoreceptor is the amount of electrochemical signals it sends to the brain based on the light it detects. Each photoreceptor releases electrochemical signals that report its *perceived luminous intensity*.

Our visual system *estimates* optic flow based on how perceived luminous intensity changes across all photoreceptors over time. Let the function $I(\mathbf{p}, t)$ represent the detected luminous intensity of the photoreceptor at position $\mathbf{p} = (x, y)$ and time t. Given this function, we can model optic flow as the *differential change* in perceived luminous intensity with respect to time, or:

$$\frac{d\ I(\mathbf{p}, t)}{dt}, \text{ the derivative of } I(\mathbf{p}, t) \text{ with respect to time.} \tag{1}$$

This model is used to tackle the *optic flow estimation* problem in Computer Vision. There, the task is: given two images I_1 and I_2 (say for example frames 1 and 2 in Figure 1-b), estimate the apparent motion of the objects in the images. To do this, we rely on several assumptions:

1. **Coherence**: The images I_1 and I_2 are related: they are images of the same scene, captured in sequence approximately dt seconds apart.
2. **Color Constancy**: The color at position $\mathbf{p}$ and time t (within I_1) *looks the same* at time $t + dt$ (within I_2), despite having translated by some amount $d\mathbf{p}$.[1] When dealing with grayscale images, this is sometimes referred to as *Brightness Constancy*. Practically, this assumption means that: $I(\mathbf{p}, t) = I(\mathbf{p} + d\mathbf{p}, t + dt)$

[1] We have studied why this is not true in reality: in our discussion of light in the graphics pipeline, we stated that *object light*—the light that is emitted from sources and reflected off the object in question—*depends* on the position of the object relative to the camera (specifically, how that position interacts with ambient, diffuse, and specular light effects).

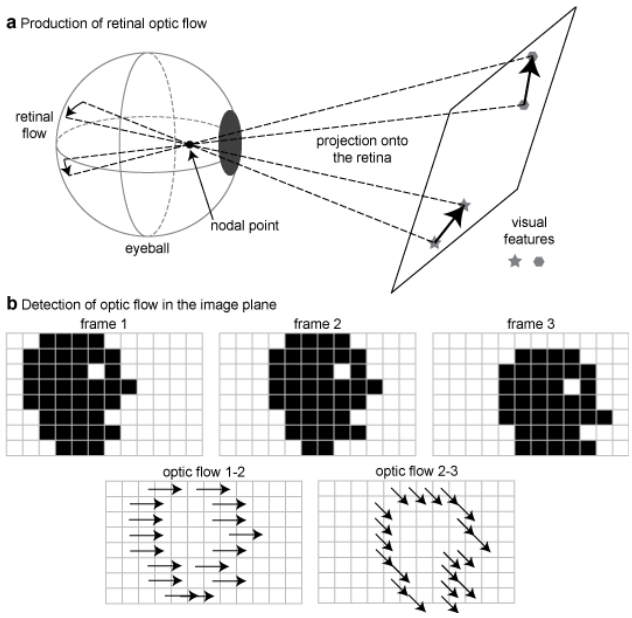

Figure 1: *An illustration of optic flow. In (a), optic flow is generated on the retina by changes in the patterns of light. The illustration shows the shift of two visual features (star and hexagon) on a plane and their angular displacements on the surface of the eyeball. In (b), if the structured light is sampled spatially and temporally this results in an image sequence. The example shows three frames, which show the movement of the silhouette of a head. The optic flow is depicted as the correspondence of contour pixels between frame 1 and 2 as well as frame 2 and 3.*

3. **Small Motion**: During dt, the displacement $d\mathbf{p}$ is less than 1 pixel. This displacement is calculated based on the velocity of apparent motion; *i.e.* $d\mathbf{p} = \mathbf{v}\ dt$, where $\mathbf{v}$ is the *optic flow vector* at position $\mathbf{p}$ and time t.

By combining Equation 1 with all the assumptions above, we arrive at:

$$\mathbf{v}_x \frac{\partial\, I(\mathbf{p}, t)}{\partial x} + \mathbf{v}_y \frac{\partial\, I(\mathbf{p}, t)}{\partial y} = -\frac{\partial\, I(\mathbf{p}, t)}{\partial t}, \text{which is equivalent to: } \nabla I(\mathbf{p}, t) \cdot \mathbf{v} = -\frac{\partial\, I(\mathbf{p}, t)}{\partial t} \tag{2}$$

Unfortunately, this is one equation with two unknowns: $\mathbf{v}_x$ and $\mathbf{v}_y$ (respectively, the horizontal and vertical components of the optic flow vector). The fact that this equation is underdetermined is referred to as the *aperture problem* as illustrated in Figure 2.

To find optical flow, additional constraints are needed. These constraints are usually problem-specific simplifications that one can deploy to solve the Computer Vision task of arriving at a good approximation of $\mathbf{v}_x$ and $\mathbf{v}_y$.

(i) Derive Equation 2 step-by-step and conceptually explain the derivation in your own words. Note, this derivation is explained in the tutorial linked in Task 2 of **Getting Started**. Thus, we are not looking for a verbatim copy of that text. Make sure you answer with sufficient detail such that we are *convinced* that you know what you are talking about. Assume that your reader has good knowledge of differential and integral calculus, Newtonian physics, and the human eye. (*10pts*)

(ii) We can think of the *optic flow estimation* problem within Computer Vision as a *computational model* of how the human visual system performs the equivalent biological process. In that light, there should be a real-world interpretation of every term in the model. For example: we have already mentioned that $\mathbf{p} = (x, y)$ is an individual photoreceptor's position and that $I(\mathbf{p}, t)$ is that photoreceptor's perceived luminous intensity at

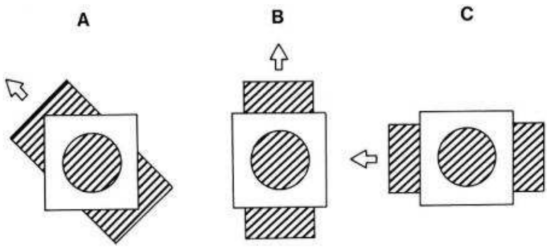

Figure 2: *The aperture problem results when trying to estimate the motion of an object that is larger than the window through which the object can be seen. Above, despite the motion of the striped plane being different in* **A**, **B**, *and* **C**, *the motion is ambiguous: that is, the motion will look the same when viewed through the aperture.*

time t. Given this:

- How should we interpret dt? From calculus, we know that dt is an infinitesimally small amount. Mathematically, it's a span of time. (*10pts*)
- Similarly, how should we interpret $d\mathbf{p}$? As with dt, we know that $d\mathbf{p}$ is an infinitesimally small amount. Mathematically, it's a translation. (*10pts*)

What do dt and $d\mathbf{p}$ represent in the real world? For each one: is it *actually* infinitesimally small or do we just approximate it that way? Does it have an upper or lower bound? If there is an upper/lower bound: what is it? How might you calculate it? Is it a property of a person? Is it a property of a perceived stimulus in motion?

(iii) The aperture problem from Equation 2 manifests for our human visual system as the *barber pole illusion*.[2] However, unlike the Computer Vision case, our brain can rely on other cues to help it disambiguate motion and arrive at a better estimate of optic flow.

Here, we will look at the *auditory* sense, focusing on the *Doppler effect* in specific. In classical physics, this effect is modeled as a *frequency shift*: a sound source of frequency f_0 that is traveling at $\mathbf{v}_s$ will be perceived by a receiving sensor traveling at $\mathbf{v}_r$ as a sound of frequency f, such that:

$$f = f_0 \times \left(\frac{c \pm \mathbf{v}_r}{c \pm \mathbf{v}_s} \right), \text{where } c \text{ is the propagation speed of waves in the medium.} \tag{3}$$

In Equation 3, $\mathbf{v}_r$ is added to c if the receiving sensor is moving toward the sound source (subtracted otherwise). In contrast, $\mathbf{v}_s$ is added to c if the sound source is moving away from the receiving sensor (subtracted otherwise).

Briefly describe how the Doppler effect might help the human brain disambiguate the optic flow estimate that arises from the barber pole illusion. How would the stimulus in Figure 2 need to change in order to take advantage of the Doppler effect? What information would you need to get from said stimulus? How would you use it to disambiguate optic flow? Do you need to guarantee additional constraints for the person? Describe exactly what information would be needed and how your solution would work. (*5pts*)

1.3 Rotation Quaternions (*5pts*)

Given rotation angle θ and normalized rotation axis $\mathbf{v} = \begin{pmatrix} v_x, & v_y, & v_z \end{pmatrix}^T$, i.e., $\|\mathbf{v}\| = 1$, show that the corresponding rotation quaternion always has unit length.

[2] https://en.wikipedia.org/wiki/Barberpole_illusion

2 Programming Part

In this programming part, we are going to implement code that should make jumping into hardware easier for us down the line (in the next homework).

In the browser, you will extend the idea of anaglyph stereo rendering from the last homework to stereo rendering for your HMD, that will eventually result in an even better 3D experience of your favorite teapot. For the HMD stereo rendering to work, we need a few parameters to set up our view frusta, pre-warp images to correct for optical distortions, etc.

We will also try out using the VRduino/Arduino for the first time. Detailed documentation for each function can be found in the header (`.h`) files. Before implementing a function, read the documentation in addition to the question writeup.

A brief overview of the VRduino code:

1. `vrduino.ino`: This file initializes all the variables and calls the tracking functions during each loop iteration. It also handles all serial input and output.
2. `OrientationTracker.cpp`: This class manages the orientation tracking. It contains functions to implement the IMU sampling, and the variables needed to perform orientation tracking.
3. `Quaternion.h, OrientationMath.h`: These files implement most of the math related to quaternion and orientation tracking. (You will primarily focus on this code for homework 4, and we will return to the other source code above for homework 5.)

2.1 HMD Stereo Rendering

The anaglyph stereo rendering you implemented in HW3 assumed that each eye saw the same screen and we used color filters to separate the different views. With the HMD, each eye actually sees a different portion of the screen, so the stereo rendering will be slightly different. The HMD retains full color information of the scene and also provides a wider field of view. Of course, because you are not expected to have your monitor hardware set up, you will be limited in how you can test the full rendering.

2.1.1 Calculating Parameters of the Magnified Virtual Screen Image (*5pts*)

Given the focal length of the lenses in your HMD, the distance between lenses and microdisplay, and the eye relief, implement `computeDistanceScreenViewer()` and `computeLensMagnification()` in `displayParameters.js` to compute the distance of the virtual image from the viewer's eyes as well as the magnification factor.

2.1.2 Stereo Rendering (*15pts*)

Using these values and the screen parameters reported above, we can implement stereo rendering. Fill in the `computeTopBottomLeftRight()` function in `transform.js` to define the appropriate view frustum for each eye. We use these values in `computePerspectiveTransform()` to create the projection matrices. Every parameter you need for the computations are found in the function arguments: `clipNear`, `clipFar`, and `dispParams`. The off-axis frustum for the HMD is slightly different from that in HW3 (see lecture slides), so make sure to update your calculations! For stereo rendering with this method, you only need two rendering passes, one for each eye, as opposed to the three passes used for anaglyph rendering.

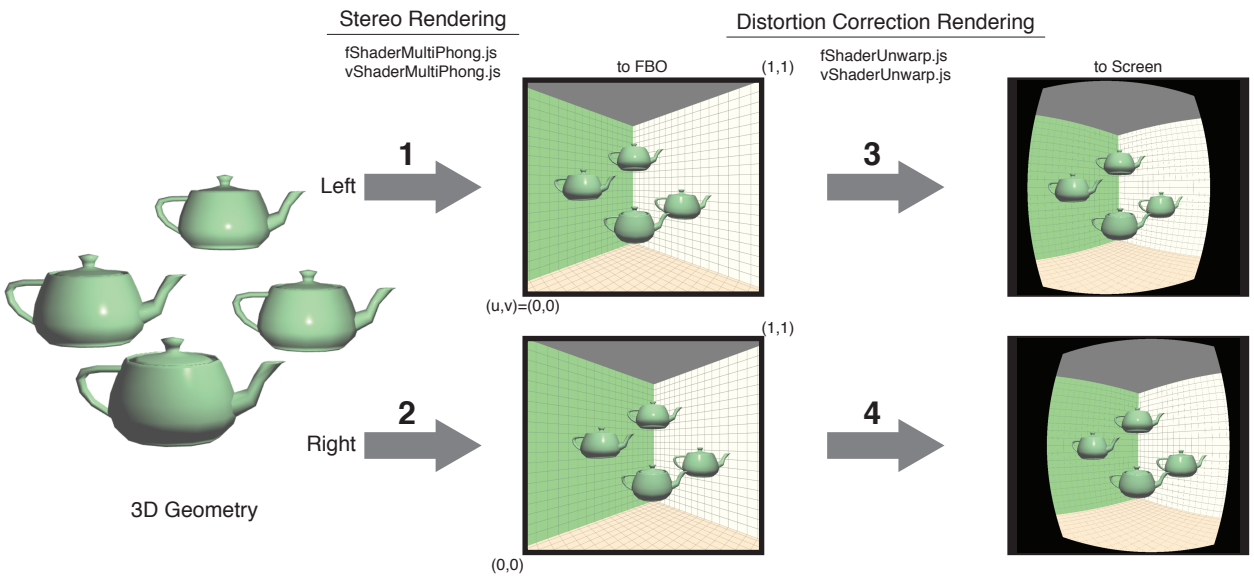

Figure 3: *The four rendering stages (as shown by the numbered arrows) for the multi-pass rendering pipeline including stereo and lens distortion correction rendering.*

2.2 Lens Distortion Correction

If you have implemented the stereo rendering correctly, you should see a 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). Eventually though, if you look at the physical display, nothing looks distorted!

These distortion will emerge from the lenses you see the screen through. Luckily, the distortion can be nicely modeled by the Brown–Conrady model we talked about 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, lines are actually straight.

The distortion correction is applied to whatever image we want to ultimately display, and is applied as the final rendering passes. Because the correction may be different per eye, we have four rendering stages (Figure 3): the first two render the left and right views of the scene into an FBO with Phong lighting using the `fShaderMultiPhong.js` and `vShaderMultiPhong.js` shaders; and the last two apply the distortion correction you will implement in `fShaderUnwarp.js` to each eye's image. Keep in mind during your computations that your distortion correction shader will only be applied to one eye's image at a time, and that the image will only populate half of the screen.

Remember: we're going to be simulating this effect now, prior to our being able to see it fully in the hardware.

2.2.1 Lens Distortion Center (*10pts*)

The first thing you'll have to do is to calculate the distortion center for each eye. As described in the lecture, the center of the distortion is the point on the screen which intersects the optical axis of the lens. Fill in `computeCenterCoord()` function in `stereoUnwarpRenderer.js` to compute the distortion center in texture (u,v) coordinates for each eye. The computed center coordinates are passed as uniforms (`centerCoordinate`) into `fShaderUnwarp.js`.

Hint: You'll end up using these values in the shader when computing your lens un-distortion and indexing into

texture maps. Remember that texture are indexed with normalized texture coordinates (i.e. between 0 and 1).

2.2.2 Fragment Shader Implementation (*15pts*)

In `fShaderUnwarp.js` implement the distortion correction using the center of lens distortion parameters you've defined above. The lens distortion parameters are stored in the uniform K, where the first element corresponds to K_1 and the second corresponds to K_2 in the lens distortion equations from class. 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, which corresponds to the center of the lens defined as the `centerCoordinate` uniform variable. Follow the lecture slide to normalize the distance between the lens center and the fragment.

You might find it useful to quickly toggle between the standard rendering mode and the unwarped mode by pressing the `1` button on the keyboard or click the button on the top.

2.3 Quaternion Algebra (*10pts*)

While Euler angles are intuitive, they cannot be used for orientation tracking in 3D. We need quaternions for that. To use quaternions on the VRduino, we will need to implement some of the basic quaternion operations discussed in class like multiplication, inversion, rotation, etc. We already implemented some of these functions for you in `Quaternion.h` and we also included a few simple unit tests in `TestOrientation.cpp` to help you verify your implementation. To run the tests, set `test = true` in `vrduino.ino`. You will need these functions to work properly for the next parts of this homework, so make sure to test them thoroughly. You can add your own tests to `TestOrientation.cpp`.

(i) Implement `setFromAngleAxis()`. This function should create a quaternion from a rotation axis and an angle given in degrees. (*2pts*)

(ii) Implement `length()`. This function should return the length of the quaternion. (*1pts*)

(iii) Implement `normalize()`. This function should return the normalized quaternion. (*1pts*)

(iv) Implement `inverse()`. This function should return the inverted quaternion. (*2pts*)

(v) Implement `multiply()`. This function should return the product of two given quaternions. (*2pts*)

(vi) Implement `rotate()`. This function should rotate the quaternion. (*2pts*)

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.