

Build Your Own VR Headset

ICCP 2025 Summer School

Brian Chao Gordon Wetzstein

Introduction

Students will use JavaScript and Arduino C++ for the lab component of this course, building on top of the provided starter code in the [GitHub repository](#). 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.

You can find starter code for the homework on our [GitHub repository](#). Please download and use this starter before starting to work on the programming part of this assignment.

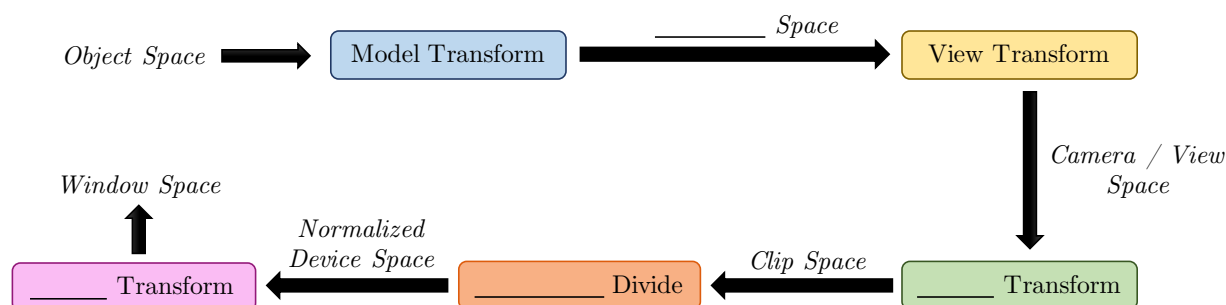
This lab consists of three components, each complementing one lecture:

1. **The Graphics Pipeline:** implement the *graphics pipeline*. In this lab, students will familiarize themselves with matrix algebra and the sequence of transformations that convert a 3D model from world space to the screen space. Students will also experiment with a pre-implemented three.js demo of various shading algorithms.
2. **The Human Visual System (HVS) and Head-mounted Display (HMD) Optics:** implement a simple stereo rendering algorithm. In this lab, students will understand the fundamentals of VR head-mounted optics, and demonstrate the stereo rendering algorithm on a VR headset.
3. **Orientation Tracking:** implement 1D orientation tracking. In this lab, students will a simple orientation tracking algorithm by combining gyroscope and accelerometer measurements.

1 The Graphics Pipeline

1.1 Graphics Pipeline Refresher

Before we dive into the code implementation, let's review the graphics pipeline as shown in the figure below:



Please fill in all the blanks. Make sure you review the graphics pipeline of transforming from object space to window space before starting to solve the following questions.

1.2 Graphics Pipeline Lab Introduction

When you first open the `render.html` file in your web browser you should see three teapots, a grid, and a set of axes. The axes are defined as follows: the red line corresponds to the $+x$ axis, the green line corresponds to the $+y$ axis, and the blue line corresponds to the $+z$ axis. Note that when viewing a scene, you look down the $-z$ axis, as described in class. The starter code provides fixed model, view, and perspective matrices. In this homework you will learn about each transform individually by interacting with them via mouse movements. You might find THREE's `Matrix4()` functions useful for this assignment when creating rotation and translation matrices. You can find the documentation [here](#). Before you begin, please update the `screenDiagonal` global variable in `render.js` with your physical screen's diagonal in inches.







1.3 Model Transforms

When building scenes in a virtual environment, the first thing you need to do is to place different 3D objects in the scene. As mentioned in the lecture, the vertices of a 3D object are typically defined in the local space where the origin (0, 0, 0) is defined with respect to the object. So if you were to load multiple 3D objects, each centered around (0, 0, 0), all the 3D objects would overlap which wouldn't be very interesting. We use transformations (translation, rotation, scaling) to move the models' vertices to different positions. Typically, these transformations, defining a 3D object's position, orientation, and size in some common world coordinate system, are called the model transformation.

We have already implemented **model translation** and **model rotation** for you. This allows you to rotate the move the teapot around and rotate the teapot, given the same camera viewpoint. Refer to the mouse movements below and try it out!

1.3.1 Model Translation Control

The direction of the translations are defined as follows:

-  with +x mouse movement (horizontal to the right): translate teapot along +x axis
-  with +y mouse movement (vertical to the bottom): translate teapot along -y axis
-  (or  on Macs) with +y mouse movement: translate teapot along -z axis

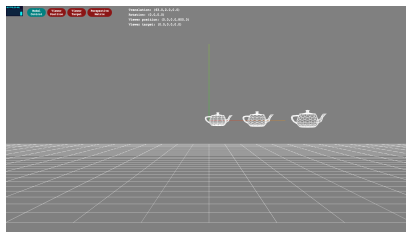


Figure 1: translation along +x

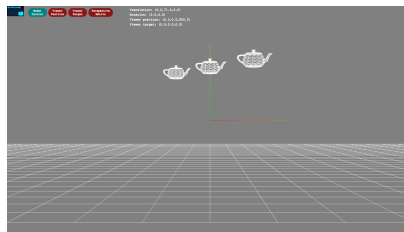


Figure 2: translation along +y

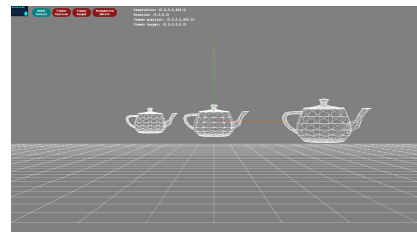


Figure 3: translation along +z

1.3.2 Model Rotation Control

The direction of the rotations are defined as follows:

- +x mouse movement (horizontal to the right): rotate teapot around the +y axis
- +y mouse movement (vertical to the bottom): rotate teapot around the +x axis
- you will not need to implement rotations around the z axis

1.4 View Transform

Once the objects are positioned in world space, we apply the view transform to shift *each* model's world vertices into view space. This space will essentially dictate the position and orientation of the camera through which we see the virtual content.

1.4.1 Move Viewer Position

You will now interact with the view matrix dynamically via mouse movements. **We have implemented this part for you.** You can see the implementation in `StateController.updateViewPosition()`.

The direction of the position changes are defined as follows:

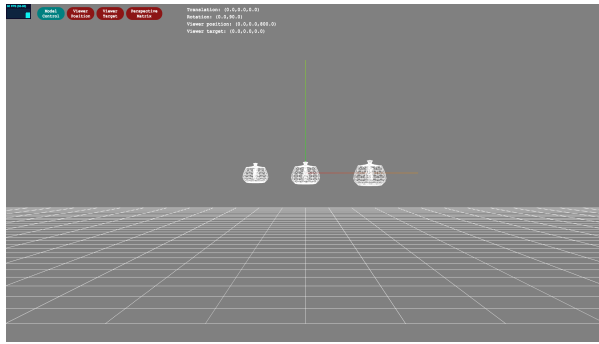


Figure 4: *model rotation around y axis, i.e. after horizontal mouse movement*

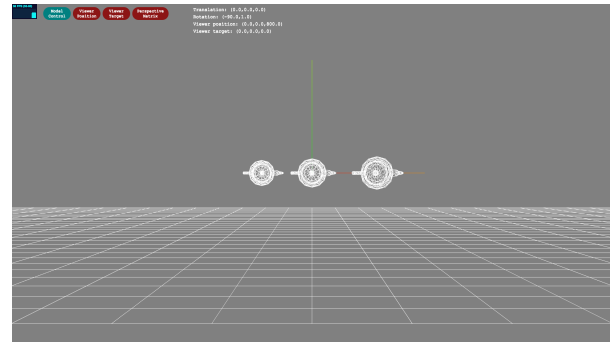


Figure 5: *model rotation around x axis, i.e. after vertical mouse movement*

- $+x$ mouse movement (horizontal to the right): translate viewer position along $+x$ axis
- $+y$ mouse movement (vertical to the bottom): translate viewer position along $-y$ axis
- `ctrl` with $+y$ mouse movement (vertical to the bottom): translate viewer position along $-z$ axis

1.4.2 Move Viewer Target

We'll now update the point to which we are looking to, i.e. the viewer target (x, y, z) . **We have also implemented this part for you.** The implementation can be found in `StateController.updateViewTarget()`.

1.4.3 Implement View Transform

Now, after defining how mouse movements update the various viewing parameters, we will construct our view matrix.

A simple, fixed view matrix dictating the view transform is already provided. The fixed matrix places the camera at 800 units along the world z -axis, and points to the world origin. In this task, you will implement the view transform function defined in class in the function `MVPmat.computeViewTransform()` in `transform.js`. From the state, you may use `viewerTarget` specifying the coordinate in world space at which the camera will look and `viewerPosition` specifying the coordinate in world space at which the camera will be positioned. Using these two inputs, generate and output the view matrix by filling in the `MVPmat.computeViewTransform()` function. We will check the functionality of your implementation when grading.

Attention: Three.js has a built-in function called `lookAt`. Do not use this function, because it only implements the rotation part of the view transform (rather than translation and rotation). Just implement it yourself, otherwise you may get unexpected results!

Hint: In this task, you assume that the viewer/camera does not rotate around z -axis. With this assumption, there exists a unique “up” vector that works for all rotations about the other axes due to a convenient mathematical property of how the up vector is calculated for the view transform. These two related but not necessarily identical vectors, one that determines the orientation of the 2D rendering and one that defines the camera coordinate frame’s up direction, are often both referred to as the up vector depending on the context.

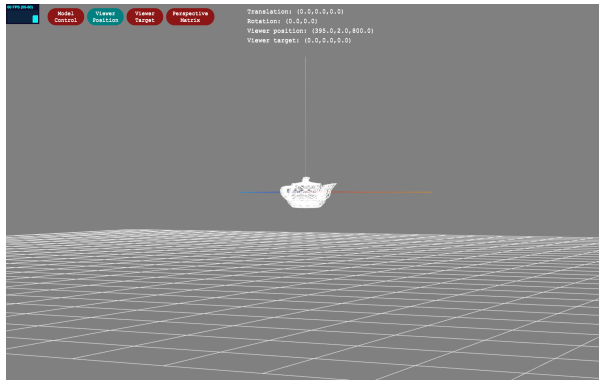


Figure 6: camera rotation around y axis, i.e. after horizontal mouse movement

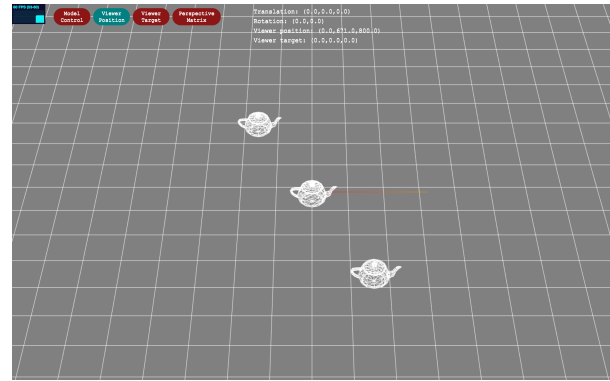


Figure 7: camera rotation around x axis, i.e. after vertical mouse movement

1.5 Projection Transform

Once the objects are placed in world space and then transformed to be placed in front of the camera via the view transform, the final step is to project the 3D content onto a 2D plane so that it can be displayed as an image on a display. This projection is called the Projection Transform. If we think of the view transform as setting the position and orientation of the camera, we can think of the projection matrix as describing the camera's lens parameters: focal length, field of view, and aspect ratio. Two commonly used types of projection transforms were described in class: perspective and orthographic. You will implement each here.

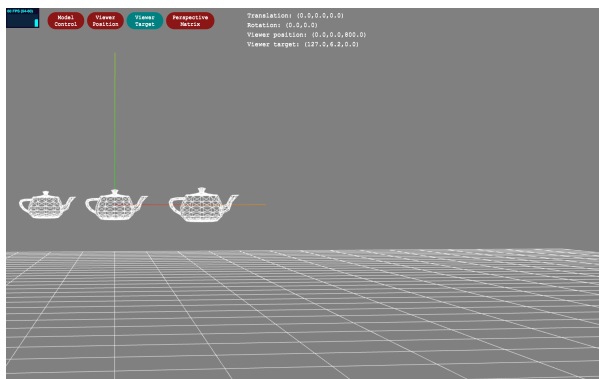


Figure 8: camera translation along x axis, i.e. after horizontal mouse movement

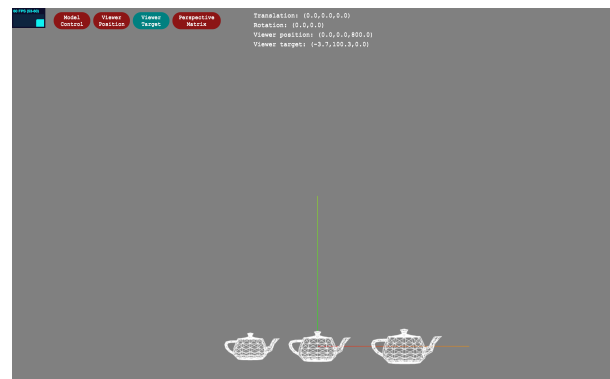


Figure 9: camera translation along y axis, i.e. after vertical mouse movement

1.5.1 Implement Perspective Transform

An initial fixed perspective projection transformation was given to you so that you could see anything at all in the previous parts. In this task, you will fill in the function `MVPmat.computePerspectiveTransform()` that will automatically generate a perspective projection matrix based on the top/bottom, left/right, and near/far clipping planes. This follows the general definition of the perspective matrix as defined in lecture.

We have again implemented updating the near/far clipping planes (`clipNear`, `clipFar`) using mouse movements for you. Check `StateController.updateProjectionParams()` for details. For the map-

ping, $+y$ mouse movement (vertical to the bottom) will pull the near clipping plane closer to the camera, while $-y$ mouse movements will push it farther away. In addition, to enforce the nonnegativity of `clipNear`, `clipNear` should be clamped to at least 1.

Based on the `clipNear` parameter input, implement `MVPmat.computePerspectiveMatrix()` to compute the perspective projection matrix.

1.5.2 Implement Orthographic Projection

Now, you will implement the orthographic projection transformation, which preserves parallel lines in the perceived image. Fill in the function `computeOrthographicTransform()` based on the top/bottom, left/right, and near/far clipping planes.

1.5.3 Perspective vs Orthographic

Describe some of the differences that you perceive between the perspective and orthographic projections. When would you want to use the perspective projection? Can you think of some applications where you would want to use the orthographic projection?

2 The Human Visual System and Head-mounted Display (HMD) Optics

2.1 Consistent Occlusions in Optical See-through AR

Optical see-through (OST) augmented reality (AR) displays overlay digital content onto the physical world using an optical combiner. Thus, the physical and digital objects follow an additive superposition principle with each component being mostly independent of the other. This is unlike the physical world where the lighting and shading of different objects influence each other, for example by casting shadows or by occluding each other. Occlusions in particular are one of the most important depth cues of human vision but, unfortunately, it is not straightforward to implement consistent occlusions between physical and digital objects in OST AR systems.

In general, we can distinguish between two cases of occlusions in AR: (a) a digitally displayed object should appear (at least partly) behind a physical object, i.e. the physical object occludes the digital object and (b) a physical object should appear (partly) behind a digital object, i.e. the digital object occludes the physical object.

- (i) Briefly describe an engineering solution that solves problem (a). What information do you need of the physical object and the digital object so that your solution works well? Do you need any additional sensors, such as special cameras? Describe exactly what sensors or information you need and how your solution works.
- (ii) Briefly describe an engineering solution that solves problem (b). What information do you need of the physical object and the digital object so that your solution works well? If you cannot come up with a good technological solution for this case, describe in detail what would have to happen for this to work.

2.2 Stereo Rendering Lab Introduction

It's time to assemble your head mounted display! You will be implementing a stereo rendering algorithm and demonstrate it on your own VR headset. 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.

View-Master Deluxe VR Viewer satisfies the Google Cardboard specs and the following specifications:

- Focal length of lenses: 40 mm
- Lens diameter: 34 mm
- Interpupillary distance (IPD): 64 mm
- Distance between lenses and screen: 39 mm
- LCD screen width: 132.5 mm, height: 74.5 mm
- Eye relief: 18 mm.

The distance between lenses and screen is measured when the lens is pulled to the end by the adjustment wheel. Even if you need to adjust the focus to see the display clearly, please stick to these values for your implementation.

After you have received your HMD, make sure to align the line rendered in the center of the screen with the line on the bottom holder in the View-Master. This mechanical centering is critical for achieving the correct stereo effect. Also, you need to move your browser window to the provided external display and set your browser to full-screen mode. Once you enter the full-screen mode, refresh the browser while opening `render.html` to initialize internal parameters. The task bar should not be visible on your screen!

2.3 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.

2.3.1 Calculating Parameters of the Magnified Virtual Screen Image

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.

2.3.2 Stereo Rendering

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`.

2.3.3 Stereo Rendering Perceptual Experiments

Once the stereo rendering works, perform a few simple perceptual experiments on yourself. Describe your experience in a short paragraph for each of the following questions:

- (i) Vary the interpupillary distance (IPD) in the code while looking at the same, static 3D scene. Describe the perceptual effect of the varying IPD.
- (ii) Experiment with the vergence–accommodation conflict (VAC). How far can you move objects out of and into the screen, i.e. away from the virtual image, while still being able to fuse the stereo images? Your visual system is quite resilient during continuous viewing, so errors might not be immediately apparent. Try closing your eyes for a bit, and then viewing the scene fresh. You should start seeing the effects then. Report min and max depth in which you can comfortably fuse the stereo image pair. You can move the teapots back and forth by pressing `w` and `s` on the keyboard and know the teapot's location from the z-component of model translation displayed on top of the browser window.

3 Orientation Tracking

All of the following tasks, should be compiled for and tested with your VRduino using Arduino programming. 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 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.

Before starting the programming part, please check that the IMU is functioning correctly. After uploading the starter program to the Teensy, open the serial monitor and input `3` and `4` to check the sampled gyroscope and accelerometer values. You should expect to see a stream of varying numbers populating the serial monitor. If you see the same number (most likely 0.0) being printed over and over, check that the Teensy is properly mounted onto the VRduino. If you cannot resolve this, contact the course staff as you may have a faulty VRduino.

3.1 Noise and Bias Estimation

How great it would be if the measurements we get from the IMU were perfect! Unfortunately, this is not the case in practice and, like most real-world sensors, all IMU sensors are subject to noise and potentially also bias. In this task, we will calibrate both measurement noise and bias for all three axes of the gyro and accelerometer.

Note that the bias terms may change in different environments (possibly even if the IMU is just turned on/off) due to changes in temperature, mechanical stress on the system, or other factors. Here, you will only calibrate these values once and ignore possibly changing values.

The answers you obtain from solving this question are critical for your orientation tracking algorithms and also simple, so this question will help you get started programming the VRduino.

3.1.1 Bias Estimation

Let's start by estimating the bias terms of the gyroscope and accelerometer. To calculate bias, we simply average a large number of samples from each of the sensors **while the VRduino is perfectly still** (e.g., put it on a table and don't touch it while taking the measurements). Specifically, in `measureImuBiasVariance()` of `OrientationTracker.cpp`, compute the mean of 1000 consecutive x, y, z gyroscope and accelerometer measurements. Store the bias values in the `gyrBias` and `accBias` arrays defined in the `OrientationTracker` class. Remember that in C++, you have to be careful with types (`float` or `int`) when performing any arithmetic operations.

To print these variables, set `streamMode = INFO` in `vrduino.ino`. Then, open the Serial Monitor (Tools > Serial Monitor) in the Arduino IDE and in the bottom right corner of the Serial Monitor, set the baud rate to 115200 and line ending to No line ending. Make sure to update the Serial Port to the Teensy's port (Tools > Serial Port). You can recalculate the bias at any time by sending the 'b' character to the Teensy.

What are the 6 calibrated bias values? Briefly comment on what values one would expect to see for the gyro and accelerometer if these sensors were perfect.

Note: To avoid having to calibrate this every time, you can set the `measureImuBias` variable to `false`. Set the variable `gyrBiasSet` to the values you measured above. The bias value can change due to temperature, so you may need to recalibrate every so often for best performance.

3.1.2 Noise Variance Estimation

Now that the bias term is estimated, also calibrate the noise variance for the IMU. As before, we simply want to observe a large number of samples from the sensors **without moving it** and get an estimate of the variance in the data, which corresponds to the noise inherent to these sensors. Modify the `measureImuBiasVariance()` function to compute the noise variance for the gyro and accelerometer. You should only need to take 1000 measurements to compute the bias and variance. Store these values in `gyrVariance` and `accVariance`. Again, the values can be printed to the serial port and observed with the serial monitor of the Arduino IDE.

3.1.3 IMU Sampling and Bias Subtraction

Implement `updateImuVariables()` in `OrientationTracker.cpp`, which is called at every loop iteration. In the starter code, we store the raw accelerometer and gyroscope values in the variables `acc` and `gyr`, which are going to be used for orientation tracking in the following questions.

First, to compensate for the bias computed in the previous question, you need to modify the variable `gyr` such that it equals the gyroscope value **minus the estimated bias**. Also update the timing variables, `deltaT` and `previousTimeImu`. Call `micros()` to get the current time; pay special attention to the units and data types that these variables and functions use. You should only call `micros()` once in your function.

3.2 Orientation Tracking in Flatland

For starters, we will track the orientation of the VRduino in 1D, i.e., in *flatland*. For this purpose, we will track the rotation angle of the VRduino around its z axis, so that we are estimating the angle between the global y axis and the local y axis of the VRduino, i.e., the *roll* angle of the device. In this particular case, we can represent the orientation of the VRduino by a single angle θ . Please review section 3 of the course notes on “3-DOF Orientation Tracking with IMUs” for further details.

Throughout this task, you need to hold the VRduino as shown in the video and rotate only around the z -axis. You do not need to use JavaScript for this task. The values calculated by the following subtasks can be visualized using the serial plotter of the Arduino IDE as described at the end of this section.

3.2.1 Gyro-only Orientation Tracking

Your first tracking task is simple: implement the simple forward Euler integration scheme we discussed in class when only a single gyro measurement is available at each time step. Implement `computeFlatlandRollGyr()` in `OrientationMath.cpp`. You should only need to use the z element in the gyro measurement.

Then in `updateOrientation()` of `OrientationTracker.cpp`, call this function and update the `flatlandRollGyr` variable. The inputs to the function should only come from the class variables you updated in the previous step.

3.2.2 Accelerometer-only Orientation Tracking

Now let's use only the accelerometer values for tracking orientation. Implement `computeFlatlandRollAcc()` in `OrientationMath.cpp`. You only need to use the x and y components of the measurement. For the ac-

celerometer, we do not need to integrate anything. We will compute the orientation angle θ (in degrees) directly from the two accelerometer measurements as discussed in class and in the course notes. Call this function in `updateOrientation()` and update the `flatlandRollAcc` variable.

3.2.3 Flatland Orientation Tracking with a Complementary Filter

Now that you have implemented both the gyro-only and the accelerometer-only version of flatland orientation tracking, let's go ahead and fuse them using a complementary filter. Applying this filter should remove the noise of the accelerometer and prevent the drift of the gyro.

Note that the complementary filter combines the integrated gyro angle with the angle estimated by the accelerometer. The proper way to integrate the angle in this case is to add the current gyro measurements, weighted by the time step, to the complementary-filtered angle of the last time step and not to the gyro-only angle from the last time step. Therefore, $\theta^{(t-1)}$ in Equation 9 of the course notes should be the angle estimated by the complementary filter at the previous time step.

Implement `computeFlatlandRollComp()` in `OrientationMath.cpp`. Call it in `updateOrientation()` in `OrientationTracker.cpp` and update the `flatlandRollComp` variable.

To plot the data, set `streamMode = FLAT` in `vrduino.ino`. You can also change the stream mode by sending '1' to the Teensy with the Serial Monitor. Then, open `Tools > Serial Plotter` to plot all the angles at each time step. This overlays the time-varying plots of the angle estimated from only the gyro in red, the angle estimated from only the accelerometer in green, and the angle estimated by the complementary filter in yellow. The color might shift depending on Arduino versions, but the order of the legend should be the gyro, the accelerometer and the complementary filter.

3.2.4 Flatland Orientation Tracking Comparison

Describe the differences you see between the three methods. What happens when you vary the blending parameter `alphaImuFilter` in `vrduino.ino`?