CS 5360/6360: Virtual Reality

Homework 5/6: Inertial Measurement Units and Sensor Fusion/Pose Tracking

Due: 12/09/2021, 11:59pm

Description

In this assignment, you will gain experience with sensors in VR using inertial measurement units (IMUs) and sensor fusion, and you will gain experience using lighting and shading in relation to the OpenGL Shading Language (GLSL).

Please review the course materials (lecture slides and readings) as part of working on this homework.

Overview

This homework is the combined fifth and sixth assignments that together result in you developing a head-mounted display (HMD) for virtual reality. This assignment is one way to assess Learning Outcome 1.

This homework only contains a programming portion which 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 Canvas/Gradescope. You can change your teams from assignment to assignment. Teams can share the HMD hardware (later on in the course).

Students should use JavaScript for this assignment, building on top of the provided starter code found on the course Canvas. 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 the class nor in office hours.

Submission Format

Homeworks are to be submitted on Canvas/Gradescope. You will be asked to submit a zip of your code (more on this later). The code can be submitted as a group on Canvas/Gradescope. Submit the of the code to the Code Submission assignment.

When zipping your code, make sure to zip the directory for the current homework. For example, if zipping up your code, find `render.html`, go up one directory level, and then zip up the entire homework 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.

Questions? Problems?

You will gain the most from this class by first trying to solve your own question first: review class materials (lecture slides, notes, textbooks, and other readings) and if that fails you, *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. Once you've done all you can do, post your question to the class (all of us are smarter than one of us). Finally, come to office hours. Be prepared to share what you've already attempted to resolve your own problem so we can best support you.

1 Programming Part – HW 5

All of the following tasks, except for Section 1.4, 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.

1.1 Noise and Bias Estimation (*6pts*)

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.

1.1.1 Bias Estimation (*2pts*)

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.

You may wish to write all 6 calibrated bias values in your comments in your code and 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.

1.1.2 Noise Variance Estimation (*2pts*)

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.

You may wish to report the noise variance values in your code in the comments.

1.1.3 IMU Sampling and Bias Subtraction (*2pts*)

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.

1.2 Orientation Tracking in Flatland (*6pts*)

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.

1.2.1 Gyro-only Orientation Tracking (*2pts*)

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.

1.2.2 Accelerometer-only Orientation Tracking (*2pts*)

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 accelerometer, 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.

1.2.3 Flatland Orientation Tracking with a Complementary Filter (*2pts*)

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.

1.3 3D Orientation Tracking with Quaternions (*28pts*)

Now that our quaternion library is set up, we can implement quaternion-based orientation tracking in 3D. Visualizing 3D orientations in the serial monitor is challenging so we built a 3D orientation visualizer for you which renders a digital twin of your VRduino. Your implementations in the next sections will cause the digital VRduinos to rotate as the physical one does. To run it, open `visualizer/visualize.html` from the homework directory, and select "3D Orientation Mode."

To stream data from the IMU to the browser you will need to setup Node and run `node server.js`. For instructions on how to install Node, please refer to the lab writeup. You should see values streaming out depending on the stream mode. Send '1' to change it to "FLAT" mode. You should see the boards "rolling" in the visualizer!

As you debug your code, you may want to reset the tracking. Sending 'r' resets all orientation estimates to 0.

1.3.1 Gyro-only Orientation Tracking (*4pts*)

Implement `updateQuaternionGyr()` in `OrientationMath.cpp`. This implements quaternion-based orientation tracking using only the gyro integration model, as discussed in class and in the course notes. Be careful with the units – make sure you always know what units you are working with (e.g., degrees or radians) for each task. Also watch out for division by zero. For this purpose, check the magnitude of the 3 gyro measurements and if any are below some threshold, e.g. `1e-8`, ignore that measurement. In addition, always normalize your quaternions to ensure that they represent valid rotations.

Call this function in `updateOrientation()` in `OrientationTracker.cpp` to update the `quaternionGyr` variable. Again, the input arguments should only come from the class variables.

Use the visualizer to see if your implementation behaves as expected. Set `streamMode = THREED` in `vrduino.ino`, or press '2' in the shell running `server.js`.

1.3.2 Tracking Tilt with the Accelerometer (*4pts*)

Now, let's use only the accelerometer to track the tilt angles, i.e., pitch and roll. Implement `computeAccPitch()` and `computeAccRoll()` in `OrientationMath.cpp`. We cannot track yaw this way, but that may be okay for certain applications.

We will track this in Euler angles and compare it with the quaternion-based tracking.

Call the functions in `updateOrientation()` and update the `eulerAcc` variable.

1.3.3 Quaternion-based Orientation Tracking with the Complementary Filter (*20pts*)

Implement `updateQuaternionComp()` in `OrientationMath.cpp`. This implements a complementary filter with quaternions. Make sure you review the lecture material and Section 5 of the course notes. Here's is a list of steps you should follow in each time step:

(i) Query all 3 gyro sensors and calculate the rotation update or instantaneous rotation q_Δ from these measurements. Prevent division by zero! For this purpose, check the magnitude of the 3 gyro measurements and if it is below some threshold, e.g. `1e-8`, then do not try to normalize gyro measurements by that magnitude. (*2pts*)

(ii) Update the complementary filter quaternion estimated in the previous time step by q_Δ using quaternion multiplication. (*2pts*)

(iii) Query the accelerometer and create a vector quaternion from these measurements. (*2pts*)

(iv) This vector quaternion is given in sensor coordinates, so rotate it from the sensor frame into the inertial frame using the current estimate of the rotation quaternion of the complementary filter (output of (ii)). Normalize the resulting rotated vector quaternion. (*3pts*)

(v) Compute the angle ϕ between the rotated, normalized accelerometer vector quaternion (output of (iv)) and the y axis (0, 1, 0) using the dot product. (*4pts*)

(vi) Compute the rotation axis $\mathbf{n}$ between the rotated, normalized accelerometer vector quaternion (output of (iv)) and the y axis (0, 1, 0) using the cross product. Normalize this axis. (*3pts*)

(vii) Implement the quaternion-based complementary filter by multiplying the tilt correction quaternion (using ϕ, $\mathbf{n}$, and the blending weight α) and the current estimate of the rotation quaternion of the complementary filter (output of (ii)). Normalize the resulting vector quaternion. (*4pts*)

Call this function in `updateOrientation()` in `OrientationTracker.cpp` to update the `quaternionComp` variable.

1.4 Head orientation and the Head & Neck Model (*10pts*)

Now that we can accurately estimate the IMU's orientation, we can track the user's head orientation in VR and use that as a natural way to control the virtual camera such that we can look around in a 3D scene. As shown in the video, mount the VRduino to the front of the HMD using the double-sided tape provided for you in the lab space. *Do not use duct tape or any other tape that leaves permanent marks on the View-Master!*[1] By the end of this task you will have implemented a rudimentary version of the Google Cardboard or Samsung GearVR all by yourself!

With the IMU orientation accessible in JavaScript, implement the view matrix updates in `transform.js` which will be run upon opening `render.html`. Similar to the way we used the visualizer discussed above, you need to run the WebSocket server to access IMU orientation data from the serial port. For this purpose, you need to stream 3D quaternion data from complementary filtering only, so either set the `streamMode` variable to `QC` in `vrduino.ino`

[1] If we find permanent tape marks on the ViewMaster when you return it, you will have to replace it with a new unit.

or enter 5 into the terminal running the WebSocket server with Node.js. Additionally, you can stop printing/displaying the data in the terminal by entering d into the terminal, which improves the latency between the orientation tracking and the rendering. The most recent IMU orientation data is stored in `imuQuaternion` of the `state` object within `stateController.js`.

1.4.1 Head Orientation (*5pts*)

In `computeViewTransformFromQuatertion()` in `transform.js`, make updates to the view matrix based on the IMU's orientation to make the viewing content update properly with head rotation. Compute the rotation matrix for the view matrix directly from the streamed quaternion. You might find the THREE function `Matrix4().makeRotationFromQuaternion()` useful. We have already implemented the IPD translation and world-to-view-space translation for you. You just need to compute the rotation matrix and insert it at the correct location. If the teapots don't appear in front of you at first, try reseting the 3D orientation by entering the 'r' character into the terminal running Node.

1.4.2 Head & Neck Model (*5pts*)

In `computeViewTransformFromQuatertionWithHeadNeckmodel()`, incorporate the kinematic constraints on head motion with the head and neck model discussed in class. Use the neck and head length parameters found in `displayParameters.js`. You can toggle the head and neck model on and off by pressing h on your keyboard with the window focus to the browser (not in the terminal).

2 Programming Part – HW 6

In this part of the homework, you will implement homography-based pose estimation with the VRduino and the Lighthouse.

The structure of the starter code is similar to the code in Homework 5. Here is a brief overview:

1. `vrduino.ino`: Initializes all tracking variables and calls tracking functions during each loop iteration.
2. `PoseTracker.cpp`: Manages the pose tracking. Inherits the orientation tracking functionalities from `OrientationTracker` in Homework 5.
3. `PoseMath.cpp`: Implements the math for the pose tracking algorithms.
4. `Lighthouse*.cpp` Handles the photodiode interrupts to record pulse timings from the Lighthouse. These are implemented for you.

Implementation hints:

- You do not need a Lighthouse base station to implement the following tasks. We provide pre-recorded timing data from the Lighthouse so you can implement and test your code off-line. Make sure that `simulateLighthouse = true` in `vrduino.ino` (this should be the default setting), so you are able to work with the pre-recorded data instead of live-captured photodiode timings.
- Before implementing a function, read the documentation in the header file for additional details.
- We provide a framework for unit testing in `TestPose.cpp`. Feel free to add your own tests to this file. Set `test = true` in `vrduino.ino` to run the tests. Each part in this assignment depends on the previous part, so make sure each part works before moving on.
- Use the provided visualizer with the "6D pose" mode selected. See Lab 6 for instructions on how to run it.

The goal of the remaining programming questions is to implement the function `updatePose()` in `PoseTracker.cpp`. Here, you will have to call several other functions, all included in the file `PoseMath.cpp`, with the correct input and output variables. In the following, we will go through each of these functions step by step. You will implement each of the functions in `PoseMath.cpp` and then call them with the correct arguments in `updatePose()`. Before starting the homework, please make sure that the CPU speed of Teensy is set to 48 MHz in `Tools` of Arduino IDE.

After implementing all functions, you can run `render.html` to dive into the VR world with 6-DOF capabilities. As we unfortunately cannot provide the access to the Lighthouse this year, you may not be able to try it with the actual physical measurements (if you have it, you can!). However, you can still feel the functionality with the pre-recorded measurements. To match the coordinate system between the lighthouse and the virtual world, reset the orientation tracking by pressing `r` in Terminal while holding your headset. If your pose estimation is jittery, you can perform denoising by temporal filtering, which is already provided in the starter code. To increase and decrease the denoising amount, press `2` and `3` in the browser window.

2.1 Clock Ticks to Normalized 2D Coordinates (*10pts*)

In every loop iteration, we query the photodiodes to see if new timing data is available for all 4 diodes. If new data is available, the `clockTicks` variable in `PoseTracker` is populated with these timings. We have implemented this functionality for you.

Your first task is simple: implement the function `convertTicksTo2DPositions()` in `PoseMath.cpp`. In this function, you should first convert the raw clock ticks measured for horizontal and vertical sweep directions to

azimuth and elevation angles, respectively. Then, you convert this angle to normalized coordinates x^n and y^n on an imaginary plane at unit distance away from the base station. Follow the equations of lecture 11. Remember that division of two integers in C returns another integer.

Once you have implemented `convertTicksTo2DPositions()`, call it with the correct arguments in `updatePose()` in `PoseTracker.cpp`. Remember that the clock ticks are available in the `clockTicks` array (sorted in the following order: `photodiode0.x`, `photodiode0.y`, `photodiode1.x`, ...) and you want to write the normalized coordinates into the variable `position2D` (same order as `clockTicks`), so these should be the arguments for `convertTicksTo2DPositions()` in `updatePose()`.

You can run the JavaScript visualizer provided to you in the starter code to visualize your normalized coordinates (upper left browser sub-window with black background, should look like Figure 1 but without labels). Make sure you see these coordinates update correctly in the browser window before continuing.

Note that the main file `vrduino.ino` will automatically stream values of the `position2D` array with the prefix 'PD' to the serial port. This is how the JavaScript visualizer reads and renders them, but you don't have to do anything for this.

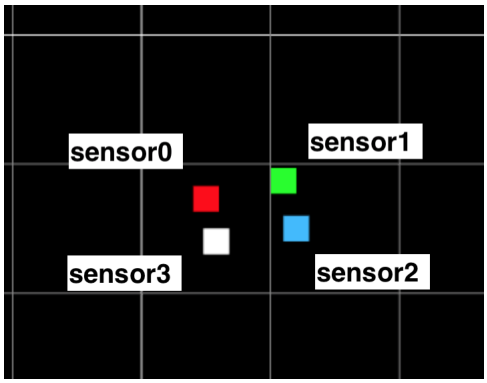

Figure 1: *Visualization of the normalized 2D coordinates x^n and y^n for all 4 photodiodes.*

2.2 Populate Matrix A (*10pts*)

Now, set up the matrix A as discussed in the lecture. For this purpose, implement the function `formA()` in `PoseMath.cpp`. Once you have implemented this function, call it with the appropriate arguments in `updatePose()`.

Remember that A describes the linear mapping between the 8 homography values h that we need to estimate and the normalized coordinates of the photodiode timings, which you computed in the last subsection. The physical reference positions of the photodiodes on the VRduino board are listed in the lecture slides and provided in the starter code in the array `positionRef` (defined in `PoseTracker.h`), so this should be one of the arguments for `formA()`. Also, create an empty 8×8 array in `updatePose()` to pass in as the output variable A. Keep in mind that you can access 2D arrays in C/C++ as `A[i][j]`, where i is the row index and j is the column index.

2.3 Solve for h (*10pts*)

With the matrix A and the vector of measurements b in place, we have all the information we need to solve for the homography values h. The measurements, called b in the lecture slides, are just your array `position2D` that you have already computed above. Implement the function `solveForH()` in `PoseMath.cpp` to compute h. Make use

of the `MatrixMath` library perform matrix inversions and multiplications. Check if `Matrix.Invert()` returns 0, which indicates that the matrix is not invertible or is ill-conditioned. This is usually only the case if you made a mistake somewhere else before calling `Matrix.Invert()`. Make sure to return `false` in `solveForH()` if `Matrix.Invert()` fails. If the inversion is successful, multiply the matrix inverse with `b`, storing the result in the output variable `hOut`.

Now call `solveForH()` in `updatePose()` with the appropriate arguments; `updatePose()` should return 0 if `solveForH()` returns false.

2.4 Get Rotation Matrix and Translation from h (*10pts*)

Next, you will implement the function `getRtFromH()` in `PoseMath.cpp` to estimate the rotation and translation from the homography matrix. Once implemented, call `getRtFromH()` in `updatePose()`. The input to `getRtFromH()` is the array of 8 homography values h and the output should be a 3×3 rotation matrix and a 3-element translation vector. In `updatePose()`, make sure to initialize the latter two variables and pass them into `getRtFromH()` where they should be populated with the correct values. The output translation vector passed into `getRtFromH()` by `updatePose()` should be called `position` – please use this variable, because we already implemented the code that streams this variable to JavaScript.

The 8 elements of h have so far only been estimated up to a scale factor s (i.e, we assumed that $h_9 = 1$). So in `getRtFromH()`, you want to estimate the scale factor s first, then use that to calculate the translation vector from $h_{3,6,9}$ and then estimate the 3×3 rotation matrix from s and $h_{1,2,4,5,7,8}$. Make sure that the columns of the rotation matrix are orthogonal and unit length, as discussed in lecture.

2.5 Convert Rotation Matrix to Quaternion (*10pts*)

Now that we have estimated the 3×3 rotation matrix representing the orientation of the VRduino with respect to the base station, we want to convert it to a rotation quaternion before we stream it via the serial port to the host computer for rendering. For this purpose, implement the function `getQuaternionFromRotationMatrix()` in `PoseMath.cpp`. Refer to the appendices of this week's course notes for all relevant math. Call the function `getQuaternionFromRotationMatrix()` in `updatePose()` with the rotation matrix you computed in the previous subsection and write the resulting quaternion into the variable `quaternionHm`.