Absolute Orientation for Head-Tracking Using Gyroscope, Accelerometer, and Camera

Kyle Fisher

Abstract—Using an ordinary 6-degree-of-freedom inertial measurement unit (IMU) for headset orientation tracking in virtual reality is susceptible to drift error in yaw. However, under the condition that the user is stationary in a reasonably lit room, the environment around the user offers visual clues that can be used to correct the drift. This project integrates a gyroscope, accelerometer, and camera to produce a fused orientation estimate which is robust against drift and produces absolute estimates of yaw which are accurate to 2°.

1 Introduction

For virtual reality headsets, head orientation tracking is fundamental to the end-user experience. In some applications, the accuracy of the head-tracking is critical for the actions performed virtually. As a result, the traditional 6-degree-of-freedom inertial measurement unit (6-DOF IMU) is often augmented by additional sensory (e.g. infrared pose tracking) to produce a more reliable reading that is more robust against measurement error. However, such solutions are often less portable or require delicate or frequent recalibration. In this project, I specifically address accumulated error (drift) by fusing gyroscope and accelerometer measurements with images taken using an inexpensive camera which is mounted to the headset. The project assumes a headset user who is stationary (e.g., seated in a living room) and an environment with several illuminated objects which can be used as visual anchors into the world.

2 Related Work

Most commercial-grade HMDs are designed to correct for the accumulated drift inherent to inertial sensors using hardware peripherals. For example, the HTC Vive uses the Vive Lighthouse Base Station for pose tracking using infrared lasers and photosensors [2]. The Oculus Rift HMD uses a 9-axis IMU and depends on the magnetometer to correct yaw-drift, but is highly susceptible to measurement error and is sensitive magnetic disturbances near the headset. [3].

3 Methods and Implementation

3.1 Capturing a Reference Frame

At initialization, the system is assumed to be at yaw-0 in world coordinates (though pitch and roll may be any angle). The appearance of the environment around the camera at initialization will be the reference against which future yaw measurements will be compared to detect yaw-drift. Thus, the first photograph taken by the camera is used to record the location of keypoints found in the environment and this image is labeled as having yaw-0. A fundamental assumption made herein is that the application user will occasionally look in the direction corresponding to the reference orientation so that the yaw-drift estimate can be updated according to the difference between the initial and most recent camera images.

3.2 System Architecture

3.2.1 Yaw-Drift Correction Loop

The high-level algorithm used to fuse the IMU and camera data and correct for yaw-drift is illustrated in Figure 2, where each functional block is executed in at a regular period in separate software threads. At intervals of approximately 1 sec., the camera is polled and the keypoints are matched with the keypoints from the initialization of the orientation tracker (i.e. the keypoints which are labeled with yaw-0). A second thread estimates the yaw-drift from the most recent image captured by comparing the locations of keypoints which match between the current image and the first image taken. For minimal latency, the IMU is polled at approximately 100Hz, and a low-pass filtered version of the yaw-drift is applied to the raw IMU quaternion such that the fused rotation can be outputted to the user and fed back to the yaw-drift estimator on the next cycle.

3.2.2 High-Level Architecture

The IMU and camera are connected directly to the Raspberry Pi, as shown in Figure 3. The IMU is attached using I2C, and the camera is connected via the ribbon bus on the Raspberry Pi board. The Ethernet port of the Raspberry Pi is used to output the fused sensor readings, including metadata from the measurement and the quaternion describing the final orientation estimate.

3.3 Camera and IMU Mounting

In my design of the physical system, the camera and IMU are mounted in such a way that the z-axis of the IMU is aimed approximately along the principal ray of the camera. The world coordinate system is configured as shown in Figure 4. Yaw, pitch, and roll are in sensor coordinates defined using right-handed coordinates along the y, x, and z axes, respectively. The camera is mounted in a portrait-style orientation such that the vertical field of view is increased. The intention of this is design decision is to increase the likelihood of capturing recognizable features for a given

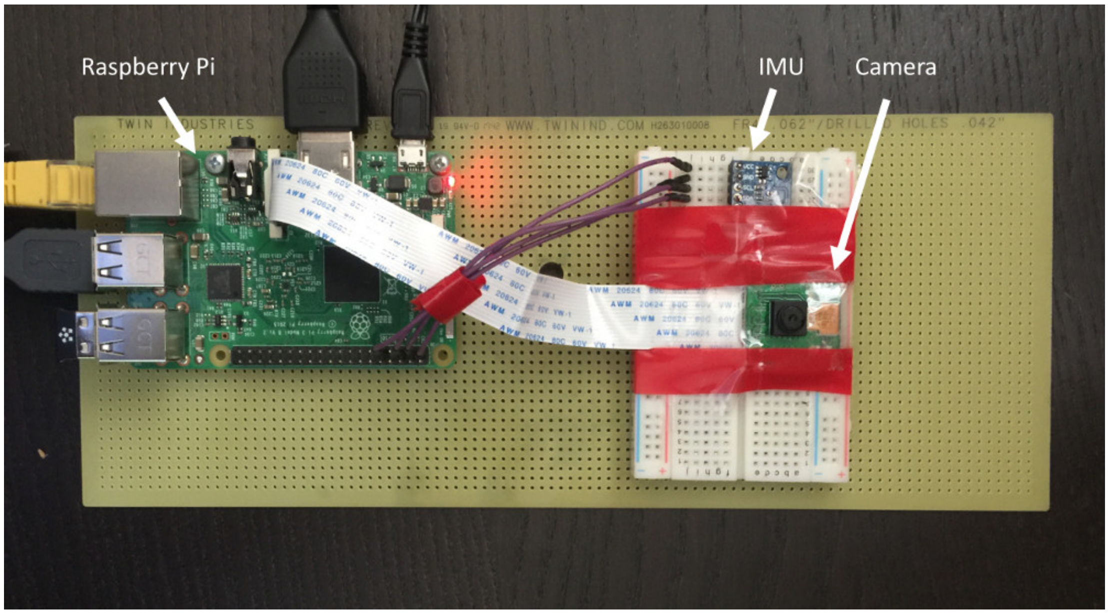

Fig. 1. Feedback loop used for fusing camera and IMU data

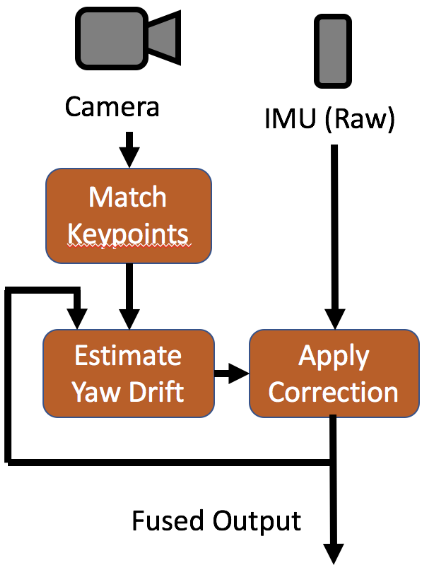

Fig. 2. Physical Prototype of the IMU-Camera Head-Tracking Setup

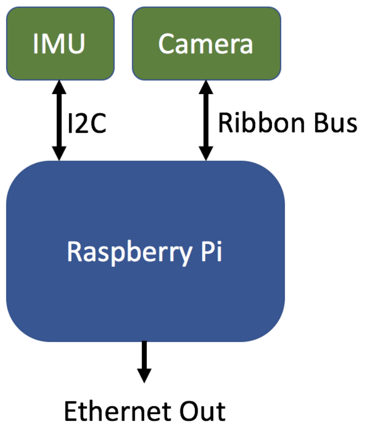

Fig. 3. System connections. The Raspberry Pi is used to process the IMU and camera data, and outputs a fused reading using the Ethernet port.

yaw angle. The zero-rotation orientation of the headset in world coordinates is defined to be y-up with respect to gravity, looking directly at the horizon and with an arbitrary yaw angle (defined to be 0 degrees at application startup).

3.4 Camera Calibration

3.4.1 Intrinsic Parameters

An important part of the system is accurate mapping from image feature locations (in pixel coordinates) to world coordinates. To perform this mapping, I adopt a pinhole camera model and assume that each keypoint detected is infinitely distant (i.e. lies on an infinitely distant hemisphere). This approximation introduces noticeable error, given moderate parallax effects (see Results section), but the system nonetheless is robust against yaw-drift even with some small parallax. To calibrate the camera, I use the OpenCV Camera Calibration tool [1] which estimates the camera's intrinsic parameters given a series of images of planar checkerboards. For this project, I use a script to capture 30 images in succession and pass these into the OpenCV calibration tool. The outputted projection matrix and distortion coefficients are given below in (1). Once calibrated, the camera parameters are assumed to be fixed for the lifetime of the camera, so no recalibration is necessary.

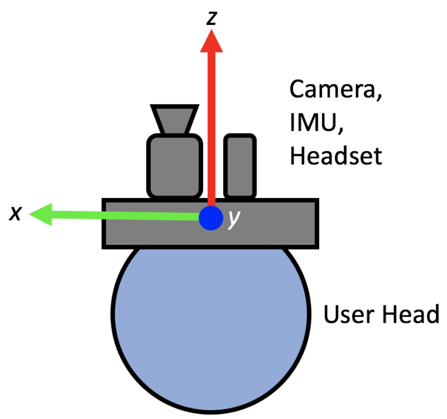

Fig. 4. Top-down view of the world coordinate axes used for head orientation tracking.

$$C = \begin{bmatrix} 638.37 & 0.0 & 173.93 \\ 0.0 & 637.99 & 328.11 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (1)$$

$$k_1, k_2, k_3 = 0.27, -1.76, 3.21$$
$$p_1, p_2 = 7.51e-04, 2.68e-03$$

3.4.2 *Still Camera Settings*

Since the camera captures images with substantial angular velocity from the motion of the user's head, the camera is programmed to capture images with a shutter speed of $\frac{1}{100}^{th}$ sec, which in my experiments proved quick enough to obtain clear images even during moderate head movement. The other exposure settings of the camera (specifically ISO and digital gain) are set automatically. With the faster shutter speed, the room must be reasonably lit in order for meaningful features to be discerned. Fortunately, even in low-lit indoor areas (e.g. a room with only one lamp illuminating the furniture), the camera is sensitive enough for SIFT to resolve enough keypoints to continue tracking as long is there is some contrast on multiple objects.

3.5 Local Image Feature Detection and Description

For feature detection, I use the Scale-Invariant Feature Transform (SIFT) [4]. SIFT is a patented algorithm for detecting local features (i.e. keypoints) in images using a series of image processing steps. It outputs a 128-vector of integers which describe the local keypoint. Given that SIFT is empirically proven to be robust against orientation changes of local features, it is an appropriate tool for matching local features from one oriented image to another (e.g. two images taken at different times and orientations from the headset camera) by comparing feature descriptor similary using a simple L_2 distance metric. For this project, I use the OpenCV Contrib [1] implementation of SIFT for feature detection. An example image of the keypoints captured using OpenCV's SIFT are shown in Figure 5.

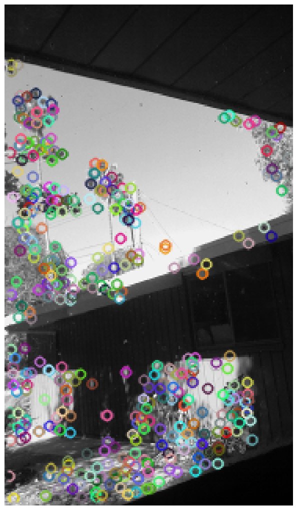

Fig. 5. Example of SIFT keypoints detected in an outdoor scene using OpenCV Contrib library

3.6 Reprojecting Keypoints

Once the camera is calibrated and keypoints have been collected, the pixel coordinates of keypoints are translated to the sensor frame using the projection and distortion coefficients from calibration. After the pinhole-model projection, the normalized coordinates correspond to the $z = 1$ plane (in sensor coordinates) in front of the headset, resulting in x and z coordinates assigned to each keypoint. In general, the original set of keypoints from $t = 0$ initialization will have changed in orientation in sensor coordinates, so it is not possible to perform effective matching without reorienting the original keypoints' locations. To correct for this misalignment, the current estimated rotation of the headset is applied inversely to the original set of keypoints such that corresponding features are approximately aligned with the new keypoints in sensor coordinate space. The remaining error is assumed to be yaw-drift and noise.

More specifically, let Q^t_{IMU} be the quaternion describing the current raw IMU reading (without fusion with the camera) at time t , and let Q^t_{Yaw} be the quaternion which prescribes a yaw correction to the raw IMU data according the current estimate of total accumulated yaw-drift. If K^t is the set of normalized positions of the keypoints detected at time t, then we can align the keypoints by applying the following rotations quaternion:

$$Q^{t=0}_{IMU} * Q^t_{Yaw} * Q^t_{IMU} \quad (2)$$

This rotation accounts for the orientation of the headset at initialization, the current yaw-drift estimate, and the current IMU reading. Each of these is necessary for producing an accurate reprojection of the initialization keypoints into the current sensor frame.

3.7 Keypoint Matching

3.7.1 *Descriptor Matching*

Keypoint matching between the initial and current keypoint sets is done using the OpenCV Brute-Force Matcher, which uses the L_2 distance between feature descriptors to determine similarity. To correct for yaw, only one point correspondence is necessary, but since there is redundancy available, I use the five strongest matches are as candidates correspondences for matching and average their contributions the the yaw-drift estimation.

3.7.2 *Keypoint Match Filtering*

The candidate correspondences are sometimes erroneous, especially when there are duplicates of objects in an environment (e.g. a set of drawers with matching knobs). Therefore, among the five strongest, matches, only the matches which suggest small drifts in yaw are accepted. Specifically, candidates are removed if the L_2 distance between their reprojected pinhole-model coordinates is greater than 0.1. This threshold worked well for removing erroneous candidates. Occasionally, no correspondences remain after the filtering, but subsequent frames typically yield true correspondences soon enough to correct yaw-drift reliably.

3.7.3 *Estimating Yaw-Drift*

Once the correspondence have been obtained, their points are projected onto the x-z plane and the yaw angle of each point with respect to the principal ray (z-axis) is calculated. Figure 9 illustrates this calculation. The angle θ_{drift} is treated as the estimated yaw-drift angle

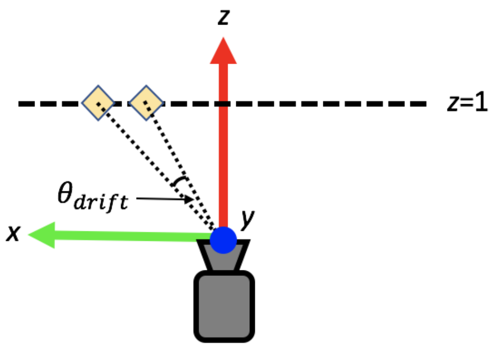

Fig. 6. Estimating yaw-drift by comparing keypoint locations in pinhole camera space. The different in yaw between two keypoints is calculated by projecting their normalized ($z = 1$) coordinates onto the x-z plane and computing the relative angle.

3.8 Visualization

3.8.1 *JAVA Demonstration*

For visualization purposes, I developed a JAVA application which shows the initial and current keypoint sets in the frame of the sensor. Figure 7 shows a screen capture from the visualization. The initial set of keypoints is plotted in white. After a panning motion, the initial set of keypoints is reprojected to the leftward size of the frame, since that is its new apparent position in sensor coordinates. The red keypoints show new keypoints in sensor coordinates. Notice that there is some overlap between the new keypoints and the initial keypoints.

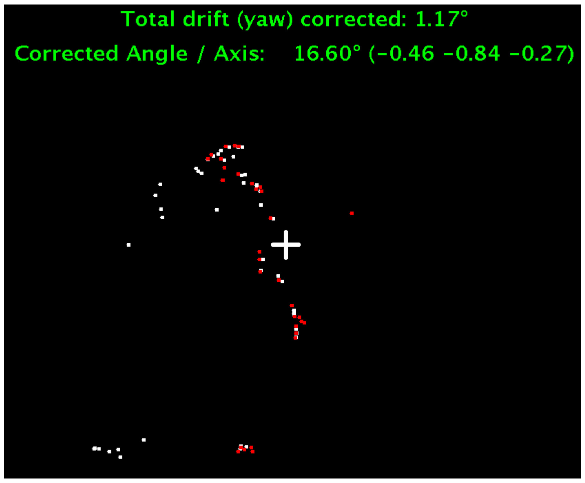
Total drift (yaw) corrected: 1.17°
Corrected Angle / Axis: 16.60° (-0.46 -0.84 -0.27)

Fig. 7. JAVA visualization demonstrating the reprojection of keypoints in sensor coordinates after a panning motion. The initial set of keypoints is plotted in white, and the new set of keypoints is plotted in red. Note the correspondence between overlapping keypoints.

3.8.2 *Communication*

For plotting the keypoints at a high framerate, the graphics are offloaded from the Raspberry Pi. The JAVA client runs on a MacBook, while the orientation estimation runs in Python on the Raspberry Pi and services HTTP requests for updated orientation estimates.

4 Experiments and Analysis

4.0.1 *Experimental Hardware*

The gyroscope, accelerometer, and camera used in this project are each small and lightweight enough to mounted comfortably on a virtual reality headset. For prototyping, I used a Raspberry Pi 3 Model B to process the image and IMU data. The camera used is the Arducam 5 Megapixels Mini Camera for Raspberry Pi, which in my tests was capable of capturing high resolution images in burst approximately 3 frames per second. The gyroscope and accelerometer readings are captured using an InvenSense MPU-9250.

4.1 Yaw-Drift Estimation

4.1.1 *Stationary Test*

As a first experiment, I did a stationary test where the IMU and camera are fixed to a tripod and drift error is computed starting from $t = 0$. In this experiment, it was easy to see the precision of the drift estimator given the ground-truth yaw angle of 0 degrees. Over 15 minutes, the system correctly accounted for $-5°$ of yaw drift error. During operation, there are several fluctuations (e.g. at $t = 12$). These fluctuations are caused by misidentified SIFT keypoints as lighting and noise conditions change slightly, but a low-pass is used to remove this effect (the smoothed curve is not plotted).

4.1.2 *Rotational Motion Test*

To simulate a more realistic scenario, I created a test in which that camera and IMU are in nearly-constant motion. By mounting the prototype rig on a tripod, I limited the movement of the IMU and camera to be almost purely

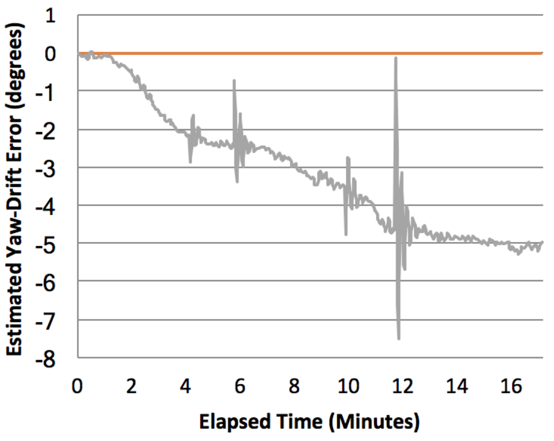

Fig. 8. Testing the yaw-drift estimation for a stationary HMD (fused yaw-drift reading in orange, raw yaw-drift in gray

rotational. Then, the rig was swiveled manually about all axes for 6.5 minutes, with occasional pauses (as in natural headset movement) to allow the camera to capture clearer images with more accurate raw IMU data. At the end of the experiment, the rig was centered back at the position from initialization. The corrected yaw measurement reads 1.22°, while the uncorrected yaw is in error by over 20°.

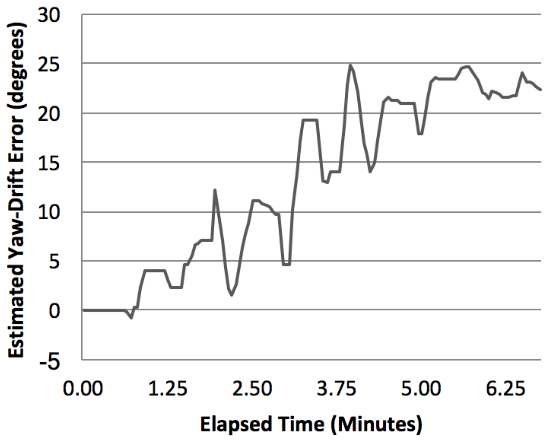

Fig. 9. Testing the yaw-drift estimation for a moving HMD

5 Future Work

5.1 Accounting for Translational Effects

An unfortunate consequence of assuming that the headset never translates relative to the room (or equivalently, that physical objects seen by the camera are infinitely distant) is that there is some motion among keypoints which is not accounted for by the yaw-correction described in this paper. One possible solution is the use of multi-view stereo geometry; by assigning a depth to the keypoints, the approximate shape of the room could be mapped and the translational pose of the headset roughly inferred [5]. In this model, the yaw-draft would be calculated using the relative position of the headset and the keypoints. Further filtering, such as an EKF, would likely be needed.

6 Conclusion

In conclusion, fusing images from an inexpensive camera with IMU readings is a viable solution to the problem of yaw-drift in HMDs for virtual reality. In this project, the prototype was observed as predicting a yaw angle which was within 1.22° of the true angle, while the uncorrected strayed by over 20°.

References

[1] OpenCV Contrib Library, 2018.
[2] O. Kreylos. ”lighthouse tracking examined”, 2016.
[3] S. M. LaValle, A. Yershova, M. Katsev, and M. Antonov. Head tracking for the oculus rift. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 187–194, May 2014.
[4] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004.
[5] S. Thrun, W. Burgard, D. Fox, and R.C. Arkin. *Probabilistic Robotics*. Intelligent robotics and autonomous agents. MIT Press, 2005.