EE 267 Virtual Reality
Course Notes: 6-DOF Pose Tracking with the VRduino

Gordon Wetzstein
gordon.wetzstein@stanford.edu

This document serves as a supplement to the material discussed in lectures 11 and 12. The document is not meant to be a comprehensive review of positional tracking for virtual reality applications but rather an intuitive introduction to the basic mathematical concepts of pose estimation with the VRduino.

1 Overview of Pose Tracking

The goal of 6-DOF pose estimation is to recover the relative position (three degrees of freedom) and rotation (another three degrees of freedom) of some rigid object, for example a headset, a controller, or the VRduino, with respect to some reference coordinate system, such as that of a camera.

Positional tracking can be implemented with a variety of technologies. Commercially-available systems include mechanical trackers, magnetic trackers, ultrasonic trackers, and GPS or WIFI-based tracking. The most widely used technology, however, is optical tracking. In optical tracking, one or more cameras observe a set of reference points, for example infrared LEDs or actively illuminated retroreflective markers mounted on a VR controller or a headset (e.g., Oculus Rift and Sony's Playstation VR headset). The pose of HMD or controller is then estimated from the measured locations of the LEDs or markers in the camera image. The arrangement of the markers on the tracked device is usually known from its design or calibrated by the manufacturer. This problem is known as the *perspective-n-point problem* and of importance in camera calibration, 3D computer vision, and other fields.

The HTC Vive also uses an optical tracking system, but rather than using a camera to observe LEDs on the headset, the Vive uses a slightly different approach. The camera is replaced by a projector and instead of LEDs, photodiodes are mounted on the device. The projector emits structured illumination to help the photodiodes determine their own 2D location in the reference frame of the projector. An early paper on this technology was published by Raskar et al. [2007], who used spatially-structured illumination. HTC calls their technology *Lighthouse* and it uses temporally-structured illumination. Specifically, the Lighthouse projector or *base station* sweeps horizontal and vertical laser stripes across the room, hence the name lighthouse. It does that very fast – 60 times per second for a full horizontal and vertical sweep with sync pulses in between. The photodiodes are fast enough to time-stamp when the laser sweeps hit them relative to the last sync pulse. Using these measurements, one of several optimization techniques can be employed to estimate the 6-DOF pose of the tracked device with respect to the base station. For more information on the Lighthouse and its communication protocols, refer to the unofficial documentation at `https://github.com/nairol/LighthouseRedox`.

The VRduino has 4 photodiodes similar to those used by HTC's VR controllers and headsets. It also has a microcontroller attached to them, so we can implement all computations for pose estimation on the VRduino directly. In this class, we use the VRduino together with HTC's official Lighthouse base station as the light source. You can buy a base station online[1], but we also provide them in the teaching lab for the class.

In the remainder of this document, we will learn the fundamental mathematics of pose tracking using the VRduino. It will probably not work quite as well as any commercial solution, partly because the VRduino only uses 4 photodiodes in a planar configuration rather than a large number of photodiodes in a 3D arrangement. Nevertheless, the VRduino is an educational tool that helps you understand and implement pose tracking from scratch.

[1] `https://www.vive.com/us/accessory/base-station/`

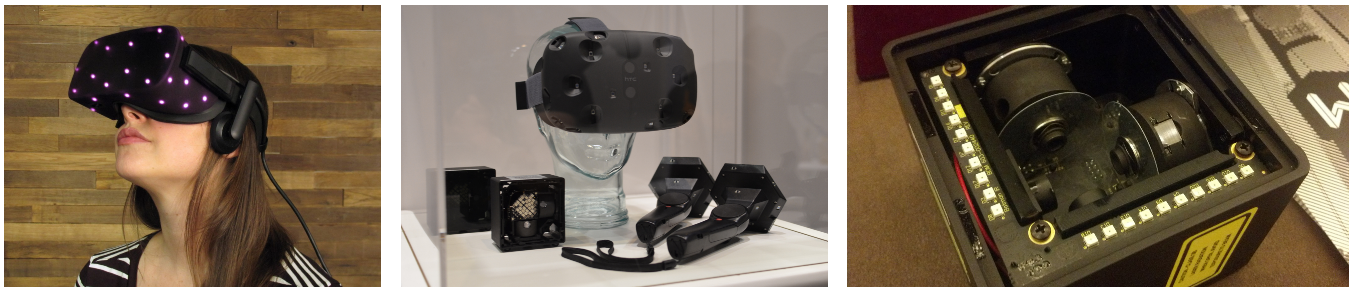


Figure 1: *Examples of optical tracking in VR. Left: near-infrared (NIR) LEDs of the Oculus Rift recorded with a camera that is sensitive to NIR (image reproduced from ifixit.com). Center: HTC Vive headset and controllers with exposed photodiodes (image reproduced from roadtovr.com). Right: disassembled HTC Lighthouse base station showing two rotating drums that create horizontal and vertical laser sweeps as well as several LEDs that emit the sync pulse (image reproduced from roadtovr.com).*

2 Image Formation in Optical Tracking Systems

The image formation for optical tracking systems is almost identical to the graphics pipeline. A 3D point $(x,\ y,\ z,\ 1)$ in the local device or object coordinate frame is represented as a four-element vector via homogeneous coordinates. A matrix is multiplied to this point and transforms it into view space where the camera is in the origin. This matrix is closely related to the modelview matrix in the graphic pipeline. Another matrix is applied that is very similar to the projection matrix: it may scale the x and y coordinates and it flips the z coordinate. Together, this is written as

$$\begin{pmatrix} x^c \\ y^c \\ w^c \end{pmatrix} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{1}$$

The coordinates $(x^c,\ y^c,\ w^c)$ are the transformed 3D points in view space such that the camera or Lighthouse base station is in the origin looking down the negative z axis. What is different from the graphics pipeline is that the "modelview" matrix has a strict order for the transformations: a 3×3 rotation matrix is first applied to the x, y, z coordinates of the point, followed by a translation by t_x, t_y, t_z. Using this particular sequence of transformations, we can get away with a 3×4 matrix for the combined rotation and translation. With this definition, we only need a 3×3 projection matrix that flips the sign of the z component to transform it into view space. As opposed to the graphics pipeline, the projection matrix here does not use a near or far clipping plane and it represents an on-axis perspective projection. Moreover, in the image formation model above we do not have access to z^c because we only measure the 2D projection of the 3D points. The graphics pipeline keeps track of the third dimension in the form of the depth buffer. Here, we only have the projected 2D coordinates $x^c,\ y^c$ and also the homogeneous coordinate w^c.

In the particular case of tracking with the Lighthouse base station, we have an aspect ratio of 1, i.e. $aspect = 1$, and we can also ignore the focal length, i.e. $f = 1$. For the remainder of this document, we will make these assumptions on f and $aspect$. Note that this image formation only models a single camera. In general, optical tracking systems with multiple cameras are quite common and could use a similar image formation model for each camera.

For optical tracking, we usually mount M known reference points on the device with local positions $(x_i,\ y_i,\ z_i,\ 1)$, $i = 1 \dots M$. In camera-based tracking applications, this could be a planar checkerboard [Bouguet 2015] or a set of retroreflective markers, as for example used by many motion capture system in the visual effects industry. The VRduino has 4 reference points: the 4 photodiodes on the printed circuit board. In this case, all reference points are actually on the same plane and we will just define that plane to be $z = 0$ in the local device frame. Thus, $z_i = 0,\ \forall i$

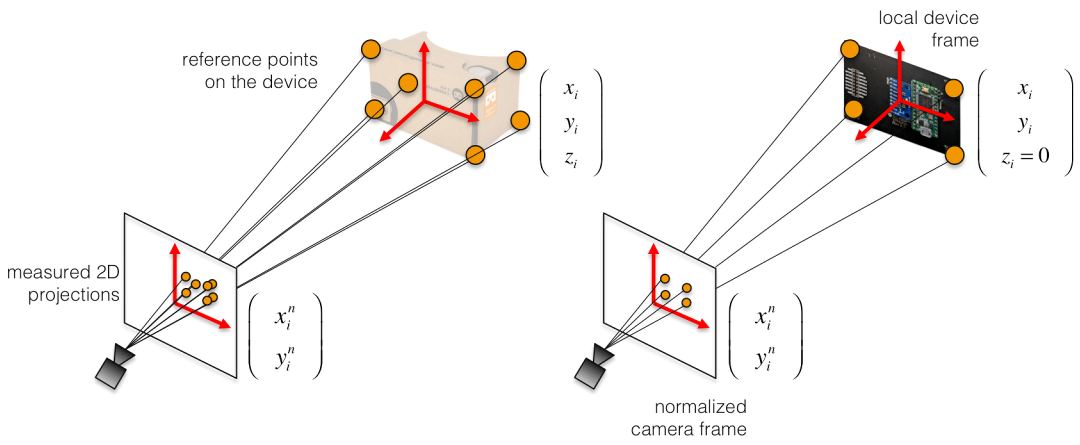


Figure 2: *Illustration of projection from 3D points (x_i, y_i, z_i) to normalized 2D locations (x_i^n, y_i^n) for a set of reference points in a general 3D arrangement (left) and for a set of planar reference points (right).*

and Equation 1 reduces to

$$\begin{pmatrix} x^c \\ y^c \\ w^c \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix}}_{\mathbf{H}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2}$$

We see that the combined transform (think modelview-projection matrix) **H** is a 3×3 matrix. We call it the *homography matrix*.

Similar to the graphics pipeline, we now perform the perspective divide by dividing x^c and y^c by the homogeneous coordinate w^c. Technically, the perspective divide is done by the homogeneous coordinate but in this particular application it is the same as the distance of the point to the camera along the z axis

$$x^n = \frac{x^c}{w^c} = \frac{h_1 x + h_2 y + h_3}{h_7 x + h_8 y + h_9}, \qquad y^n = \frac{y^c}{w^c} = \frac{h_4 x + h_5 y + h_6}{h_7 x + h_8 y + h_9} \tag{3}$$

Here, x^n and y^n are the normalized lateral coordinates on a plane at unit distance from the camera (see Fig. 2). In the graphics pipeline, these are called normalized device coordinates.

Now that we have a model for the image formation in optical tracking systems, we can start thinking about the inverse problem. Usually, we know the set of reference points in local coordinates $(x_i,\ y_i,\ z_i)$ and we have some way of measuring the corresponding normalized coordinates in camera space $(x_i^n,\ y_i^n)$. How do we get these measurements? For camera-based tracking, we use image processing to locate the reference points in the camera image. We will discuss how to get them for the VRduino in the next section of this document. Given the mapping between several 3D points in the local coordinate frame and their measured 2D projections, the problem for all optical tracking systems is to estimate the pose of the object that we want to track. Pose usually includes position and rotation. Note that the pose can only be estimated relative to the camera or Lighthouse base station. This inverse problem is also known as the perspective-n-point problem[2] [Lepetit et al. 2008].

In this document, we use the same right-handed coordinate system that we have been using throughout the course. If you read papers or other sources on optical tracking or camera calibration, signs and variables may be slightly different from our notation due to the use of other coordinate systems.

[2] `https://en.wikipedia.org/wiki/Perspective-n-Point`

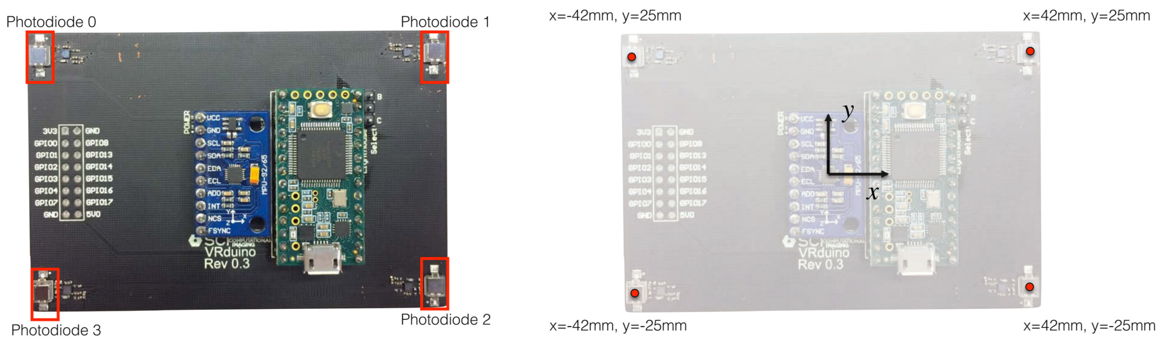


Figure 3: *The VRduino. Left: photograph of VRduino showing photodiodes, the inertial measurement unit, and the microcontroller. Right: the local coordinate system is centered in the VRduino, which is also directly in the center of the IMU. The x and y locations of the photodiodes in the local frame are indicated. The local frame is defined such that the z locations of all photodiodes in that frame is 0 and the z-axis comes out of the VRduino (also see last week's lecture notes).*

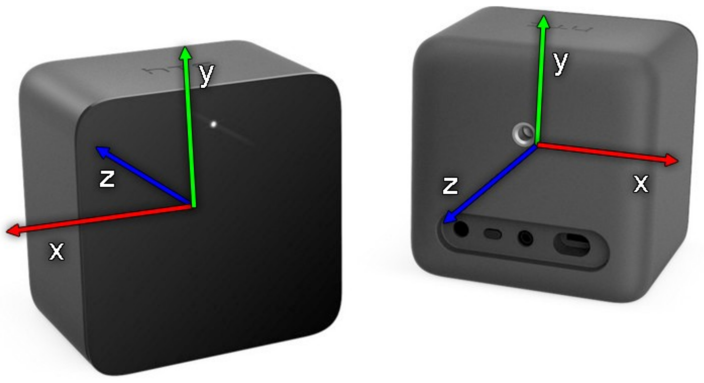


Figure 4: *HTV Vive base station (front and back) with local coordinate system illustrated.*

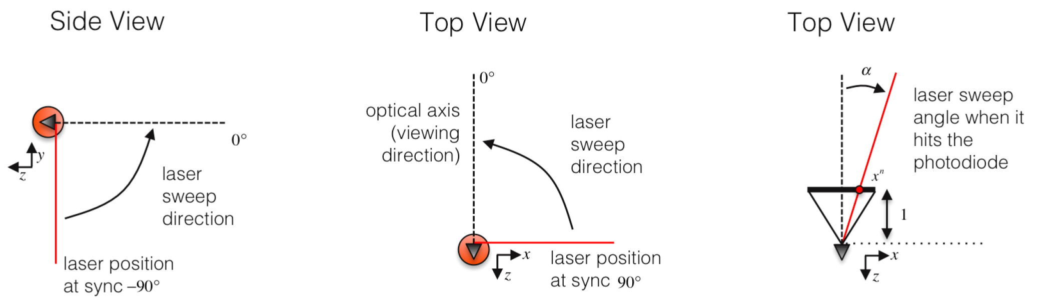
The image formation outlined above is consistent with that of the graphics pipeline and it is also commonly used for camera-based tracking and pose estimation as well as camera calibration. For more details on camera-based optical tracking, please refer to Section 9.

3 VRduino and Base Station

As discussed in the context of orientation tracking last week, the VRduino is basically an Arduino shield, i.e. a small PCB attachment, that has the IMU mounted on it as well as 4 photodiodes. The Arduino is a Teensy 3.2, which uses a 32 bit ARM processor running at 48 MHz. As indicated in Figure 3, the local coordinate origin is in the center of the VRduino (directly centered in the IMU). The specific locations of the photodiodes are also illustrated.

A photograph of the front and back of the base station is shown in Figure 4. As usual, we use a right-handed coordinate system and also adopt the convention from OpenGL that the camera, or here the base station, looks down the negative z-axis.

The signal of the photodiodes on the VRduino is digitized and amplified before it reaches the Teensy, so we can use interrupts to time-stamp rising and falling edges. The VRduino will see two different types of signals emitted by the base station: a sync pulse and a sweep. We distinguish these two different events by their duration, i.e. the time difference between a detected rising and falling edge. There are a few pre-set durations that will indicate whether a detected signal is a sync pulse and the duration also encodes other information. For example, if the duration of the



Side View
0°
y
z
laser sweep direction
laser position at sync −90°
Top View
0°
optical axis (viewing direction)
laser sweep direction
x
z
laser position at sync 90°
Top View
α
laser sweep angle when it hits the photodiode
x^n
1
x
z

Figure 5: *Laser sweep directions of the Lighthouse base station. Left: the vertical sweep moves bottom to top. At the time of the sync pulse, the sweeping laser is pointing down. Center: the horizontal sweep is right to left. At the time of the sync pulse, the sweeping laser is pointing right. Right: the detected sweep angle α is projected onto a plane at unit distance away from the base station for further processing.*

pulse is 62.5 μs, we know that the following signal will be a horizontal sweep whereas a pulse length of 72.9 μs indicates that the next signal will be vertical sweep. For a list with all the information encoded in a sync pulse, please refer to the unofficial documentation of the Lighthouse[3].

When a sync pulse is detected for one of the photodiodes, a timer is reset and waits for the rising edge of the next signal. This relative time difference is what we are looking for. It will be reported in "clock ticks" of the microcontroller. If the microcontroller runs at 48 MHz, 48 million clock ticks would correspond to 1 second. Therefore, we convert the number of measured clock ticks to metric time as $\Delta t = \#ticks/48,000,000$. This conversion can be adjusted if the microcontroller runs at a different frequency.

From the perspective of the base station, the sweep direction for the horizontal sweep is right to left and for the vertical sweep bottom to top (see Fig. 5). At the time of the sync pulse, the laser stripes are 90° away from the optical axis sweeping towards it. Each laser completes a full rotation in $1/60$ of a second and the horizontal and vertical lasers are offset such that they do not interfere with one another. We can therefore convert the relative timing between sync pulse and sweep Δt into an angle relative to the optical axis

$$\alpha_h = -\Delta t_h \cdot 60 \cdot 360 + 90, \qquad \alpha_v = \Delta t_v \cdot 60 \cdot 360 - 90 \tag{4}$$

where α_h and α_v are the horizontal and vertical angle in degrees. Knowing the angles allows us to convert them into normalized lateral coordinates. As illustrated in Figure 5 (right), this is done by computing the relative x^n and y^n coordinate of the detected horizontal and vertical sweep on a plane at unit distance in front of the base station. Remember that we have no idea how far away the base station is, we just know from which relative direction the sweep came. Thus,

$$x^n = \tan\left(2\pi\frac{\alpha_h}{360}\right), \qquad y^n = \tan\left(2\pi\frac{\alpha_v}{360}\right) \tag{5}$$

After we estimate x^n and y^n for each of the photodiodes, we can continue and estimate the position and orientation of the VRduino.

4 Estimating Pose with the Linear Homography Method

Now that we know how to measure the 2D coordinates (x_i^n, y_i^n) for all photodiodes of the VRduino, we can estimate the pose of the VRduino with respect to the Lighthouse base station. However, we only have 8 measurements (the x^n and y^n coordinates of all 4 photodiodes) but the homography matrix has 9 unknowns. This is an ill-posed inverse problem, because the number of unknowns is larger than the number of measurements.

[3]https://github.com/nairol/LighthouseRedox/blob/master/docs/Light%20Emissions.md

Luckily, the homography matrix actually only has 8 degrees of freedom, i.e. any scale s that is applied element-wise to the matrix as $s\mathbf{H}$ does not change the image formation. We can see that more clearly by writing

$$x^n = \frac{x^c}{w^c} = \frac{sh_1x + sh_2y + sh_3}{sh_7x + sh_8y + sh_9} = \frac{s\left(h_1x + h_2y + h_3\right)}{s\left(h_7x + h_8y + h_9\right)} = \frac{h_1x + h_2y + h_3}{h_7x + h_8y + h_9} \tag{6}$$

Thus, scaled versions of a homography matrix result in the same transformation from 3D to 2D coordinates. A common way of dealing with this situation is to set $h_9 = 1$ and only attempt to estimate the remaining 8 parameters $h_{1...8}$. Although this does not allow us to estimate the scaling factor s, which we need to get the rotation and translation from the homography, we will see later in this section that we can compute s in a different way after we estimated $h_{1...8}$.

The reduced image formation is thus

$$x^n = \frac{x^c}{w^c} = \frac{h_1x + h_2y + h_3}{h_7x + h_8y + 1}, \qquad y^n = \frac{y^c}{w^c} = \frac{h_4x + h_5y + h_6}{h_7x + h_8y + 1}, \tag{7}$$

which we can multiply by the denominator

$$\left(h_7x + h_8y + 1\right)x^n = h_1x + h_2y + h_3, \qquad \left(h_7x + h_8y + 1\right)y^n = h_4x + h_5y + h_6, \tag{8}$$

and rearrange as

$$\begin{pmatrix} x^n \\ y^n \end{pmatrix} = \begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -x\,x^n & -y\,x^n \\ 0 & 0 & 0 & x & y & 1 & -x\,y^n & -y\,y^n \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix} \tag{9}$$

We see that the mapping from one 3D point to its measured lateral coordinate results in 2 measurements with 8 unknowns. This is still an ill-posed problem. To get 8 measurements for this linear equation system to become square and (hopefully) invertible, we need at least 8 different measurements. The minimum number of 3D references points for solving this problem is therefore 4, which is exactly the number of photodiodes on the VRduino. With the normalized 2D coordinates of all 4 photodiodes in hand, we solve the following linear problem

$$\underbrace{\begin{pmatrix} x_1^n \\ y_1^n \\ \vdots \\ x_M^n \\ y_M^n \end{pmatrix}}_{\mathbf{b}} = \underbrace{\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1\,x_1^n & -y_1\,x_1^n \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1\,y_1^n & -y_1\,y_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & 1 & 0 & 0 & 0 & -x_M\,x_M^n & -y_M\,x_M^n \\ 0 & 0 & 0 & x_M & y_M & 1 & -x_M\,y_M^n & -y_M\,y_M^n \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix}}_{\mathbf{h}} \tag{10}$$

Here, M is the number of reference points and we generally require $M \geq 4$. Then, we just solve the resulting linear equation system $\mathbf{Ah} = \mathbf{b}$ for the unknown homography $\mathbf{h}$. In Matlab this can be done using the backslash operator $\mathbf{h} \approx \mathbf{A}\backslash\mathbf{b}$. On the Arduino we can use a matrix math library to invert the matrix and multiply to the measurements as $\mathbf{h} \approx \mathbf{A}^{-1}\mathbf{b}$. Note that $\mathbf{A}$ may be ill-conditioned in some cases, but usually that only happens when the measurements of the photodiodes are incorrect for some reason.

With the homography matrix in hand, our next goal is to estimate the actual translation vector $t_x,\ t_y,\ t_z$. Let's start by repeating how the rotation and translation, i.e. the pose, are related to the scaled homography (see Eq. 2)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{pmatrix} = s \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{pmatrix} \tag{11}$$

To estimate the scale factor s we use the insight that any valid rotation matrix has normalized rows and columns, i.e. their length or ℓ_2-norm equals 1. During our homography estimation, we did not actually enforce any normalization on the matrix columns, so let's just impose this constraint now by setting s to the inverse of the average length of the two rotation matrix columns:

$$s = \frac{2}{\sqrt{h_1^2 + h_4^2 + h_7^2} + \sqrt{h_2^2 + h_5^2 + h_8^2}} \tag{12}$$

Multiplying this scale factor with the estimated homography results in the first two columns to be approximately normalized.

Estimating translation from the homography matrix Using the scale factor s and the estimated homography matrix, we can compute the translational component of the pose as

$$t_x = sh_3, \qquad t_y = sh_6, \qquad t_z = -s \tag{13}$$

Estimating rotation from the homography matrix We can also compute the full 3×3 rotation matrix from the first two columns of the homography matrix. This is done by orthogonalizing the first two columns of the rotation matrix that we can now easily compute and then by computing the third row using the cross-product of the others.

Specifically, we compute the first column $\mathbf{r}_1$ as

$$\mathbf{r}_1 = \begin{pmatrix} r_{11} \\ r_{21} \\ r_{31} \end{pmatrix} = \begin{pmatrix} \frac{h_1}{\sqrt{h_1^2+h_4^2+h_7^2}} \\ \frac{h_4}{\sqrt{h_1^2+h_4^2+h_7^2}} \\ -\frac{h_7}{\sqrt{h_1^2+h_4^2+h_7^2}} \end{pmatrix} \tag{14}$$

Similarly, we extract the second column of the rotation matrix $\mathbf{r}_2$ from the homography, but we have to make sure that it is orthogonal to the first column. We can enforce that as follows

$$\widetilde{\mathbf{r}}_2 = \begin{pmatrix} r_{12} \\ r_{22} \\ r_{32} \end{pmatrix} = \begin{pmatrix} h_2 \\ h_5 \\ -h_8 \end{pmatrix} - \begin{pmatrix} r_{11}\left(r_{11}h_2 + r_{21}h_5 - r_{31}h_8\right) \\ r_{21}\left(r_{11}h_2 + r_{21}h_5 - r_{31}h_8\right) \\ r_{31}\left(r_{11}h_2 + r_{21}h_5 - r_{31}h_8\right) \end{pmatrix}, \qquad \mathbf{r}_2 = \frac{\widetilde{\mathbf{r}}_2}{\|\widetilde{\mathbf{r}}_2\|_2} \tag{15}$$

Now, $\mathbf{r}_2$ should be normalized and orthogonal to $\mathbf{r}_1$, i.e. $\mathbf{r}_1 \cdot \mathbf{r}_2 = 0$.

Finally, we can recover the missing third column of the rotation matrix using the cross product of the other two

$$\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2 = \begin{pmatrix} r_{21}r_{32} - r_{31}r_{22} \\ r_{31}r_{12} - r_{11}r_{32} \\ r_{11}r_{22} - r_{21}r_{12} \end{pmatrix} \tag{16}$$

This gives us the full 3×3 rotation matrix $\mathbf{R} = [\mathbf{r}_1\, \mathbf{r}_2\, \mathbf{r}_3]$.

Usually, we would convert the rotation matrix to another, less redundant rotation representation such as a quaternion (see Eq. 39) or Euler angles (see Eq. 37). Please refer to Appendices C and D for more details on these conversions.

In summary, the linear homography method is very fast and it gives us a reasonably good estimate of both position and orientation of the VRduino in the reference frame of the base station. However, this is a 2-step process where we estimate the homography matrix first and then extract an approximation of the pose from it. During the homography matrix estimation, we did not enforce that it should be exclusively contain a rotation and a translation. Other transforms, such as shear, may be part of it too, which is the reason for us having to orthogonalize the rotation matrix columns afterward. This introduces small errors and also makes the homography method somewhat susceptible to noise in the measurements. These errors may be improved by using an iterative nonlinear method, which we derive in Appendix A.

5 Integrating the Estimated Pose into the Rendering Pipeline

Similar to 3-DOF orientation tracking (see lecture notes from last week), 6-DOF pose tracking would typically be implemented on the microcontroller, i.e. the VRduino. The output is then streamed via serial USB to the host computer.

In the following subsections, we outline how to use the six values, i.e. yaw, pitch, and roll angle as well as the estimated 3D position, in the rendering pipeline. To this end, we first discuss how to render a test object to validate that the pose tracking pipeline works correctly. We then outline how we can use the estimated pose to control the camera.

5.1 Visualizing the Estimated 6-DOF Pose with a Test Object

Rendering a test object relative to the base station is important when you mount the VRduino on a physical object that you'd like to track, like a controller or some kind of an input device, and also as a debug mode for verifying that the pose tracking works when it it mounted on the HMD.

Let us consider the easiest case, where a user holds the VRduino and moves it around facing the base station. For now, we assume that the base station is the world's origin and looks down the negative z-axis (see Fig. 4).

With the estimated pose, we write a transformation from 3D points x, y, z in the local device coordinates of the VRduino to the reference frame of the base station as

$$\begin{pmatrix} x^c \\ y^c \\ z^c \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{T}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{R}} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{17}$$

Again, this formulation assumes that the base station is in the world origin. We know t_x, t_y, t_z from the pose estimation, so we can use these values directly to compute a translation matrix $\mathbf{T}$, whereas the three estimated rotation angles are converted to $\mathbf{R}$ by multiplying rotations for yaw, pitch, and roll, respectively.

Usually, we want to use Earth's reference frame instead of that of the base station. Especially when we want IMU values, which are measured with respect to Earth coordinates, and those from pose tracking to be the same coordinate system. Unfortunately, this is difficult when the base station is not level and not pointing up or down or leaning sideways, which is often the case. In that case, there is a mismatch between Earth's frame and that of the base station. One way of correcting for this mismatch would be to mount an accelerometer on the base station so that we can track the orientation of the base station in Earth coordinates. Fortunately, HTC already did that for us and the base station optically transmits the values of its internal accelerometer via the data bits encoded in the sync pulse to the VRduino. We implemented the necessary decoding routines for you, so the pitch and roll angles of the

Algorithm 1 Render test object to verify pose estimation

```
double pose_translation[3], yaw, pitch, roll;                            // position and orientation from pose tracking
double pitch_bs, roll_bs;                                                // tilt of the base station
updatePoseFromSerialStream(pose_translation,yaw,pitch,roll,pitch_bs,roll_bs);  // get pose from serial stream
glRotatef(roll_bs, 0, 0, 1);                                             // roll of base station
glRotatef(pitch_bs, 1, 0, 0);                                            // pitch of base station
glTranslatef(pose_translation[0], pose_translation[1], pose_translation[2]);   // translation
glRotatef(roll, 0, 0, 1);                                                // roll
glRotatef(pitch, 1, 0, 0);                                               // pitch
glRotatef(yaw, 0, 1, 0);                                                 // yaw
drawTestObject();                                                        // draw the target object
```

base station are available on the VRduino[4].

Using the pitch and roll of the base station, we can transform points from the local coordinate system of the VRduino into world space as

$$\begin{pmatrix} x^w \\ y^w \\ z^w \\ 1 \end{pmatrix} = \widetilde{\mathbf{R}} \begin{pmatrix} x^c \\ y^c \\ z^c \\ 1 \end{pmatrix} = \underbrace{\widetilde{\mathbf{R}}\,\mathbf{T}\,\mathbf{R}}_{\mathbf{M}} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{18}$$

Here, $\widetilde{\mathbf{R}}$ models the rotation from the base station frame to the world frame, as estimated with the base station's internal accelerometer. We denote the combined transformation from local device coordinates to world frame as $\mathbf{M}$.

Pseudo-code for implementing all of these transformations efficiently in OpenGL to render a test object are outline in Algorithm 1. Figures 6 (A) and (B) show screenshots of a simple OpenGL program that renders the base station and VRduino from different views. This is done by streaming the 6-DOF pose estimated by the homography-based algorithm from the VRduino to the host computer. In addition, the pitch and roll angles of the base station, which are optically transmitted to the VRduino, are streamed to the host computer via USB as well. We then render a wireframe model of the base station and VRduino with the estimated pose to verify correct operation.

5.2 Using the Estimated Pose to Control the Viewer in VR

The estimated 6-DOF pose can also be integrated into the rendering pipeline so as to control the viewer in VR. That is, the VRduino is mounted on the HMD with the Teensy and the photodiodes looking toward the base station. Changes in the 3D position and orientation of the VRduino, as computed using the pose estimation outlined above, can now be used to to map physical head motion of a user to the viewpoint of the stereo camera in VR. The basic idea of controlling the virtual camera with the estimated pose is to apply the inverse transformation of Equations 17 or 18 as the view matrix. This translates and rotates the world around the VRduino such that the virtual camera is aligned with the VRduino, looking down its negative z-axis. If we mount the VRduino on an HMD, it is usually more convenient to arrange it so that its z-axis along with the Teensy and IMU face away from the user (see Section 4.5.2 of the last course notes on IMUs). In this case, we need to add another coordinate transform, i.e. we need to flip the x and z axes, to account for the rotated VRduino.

We usually also want to do stereo rendering and for that we need to include the interpupillary distance ipd of the user, e.g. 64 mm, and we can use a head and neck model by introducing the offsets l_h and l_n in the z and x dimension, respectively, to model the distance between the center of rotation of the user's head and the actual position of the VRduino. For more details on these parameters, please see Section 4.5.2 of the course notes on 3-DOF orientation

[4]Due to the fact that the base station only transmits 1 bit of data per sync pulse, optically transmitting the accelerometer values along with other meta data usually takes a few seconds after the VRduino starts.

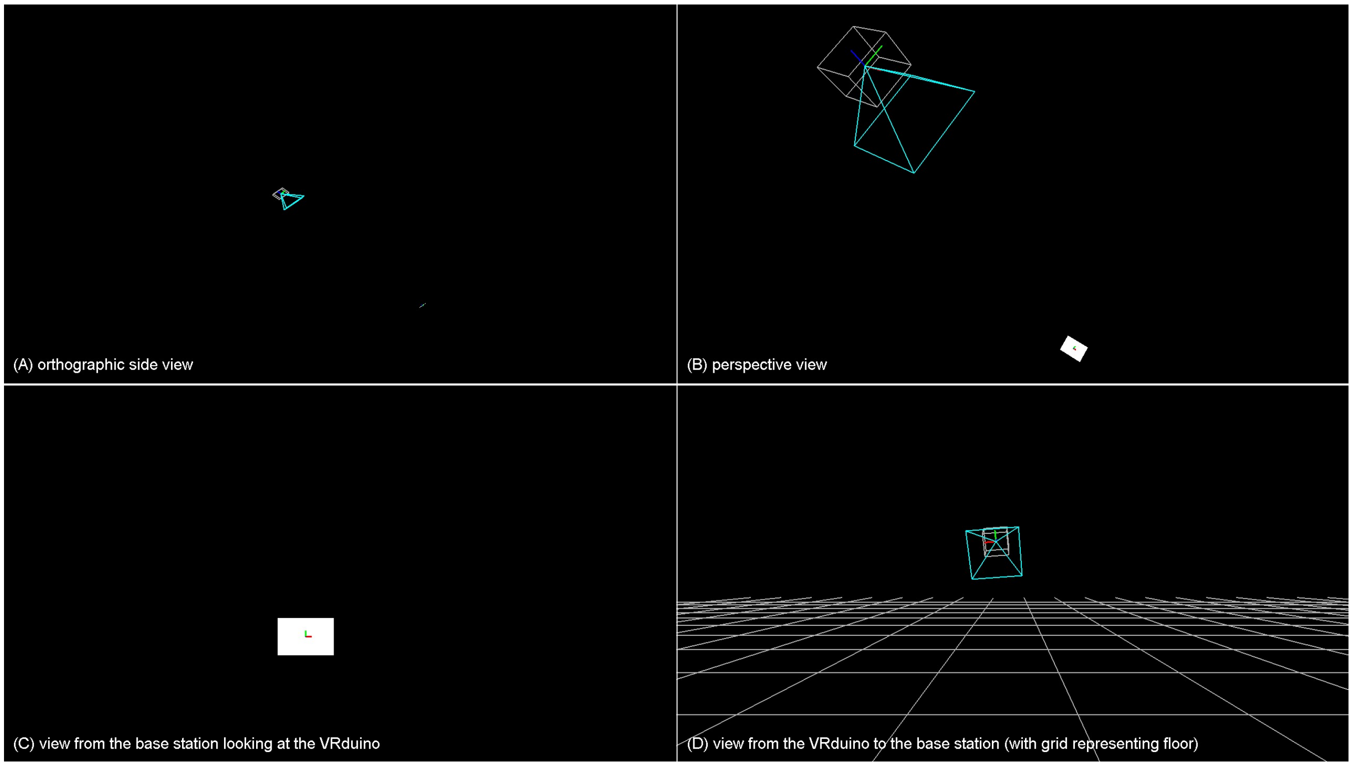

(A) orthographic side view
(B) perspective view
(C) view from the base station looking at the VRduino
(D) view from the VRduino to the base station (with grid representing floor)

Figure 6: *Pose tracking data integrated into rendering pipeline. (A) and (B) show the base station and the VRduino rendered in world coordinates. Their local coordinate systems are drawn as colored lines. Note that the base station is rotated with respect to the world according to the base station's accelerometer values that are optically transmitted to the VRduino and then sent to the host computer via serial USB from there. (C) shows the scene rendered from the perspective of the base station – you see only VRduino with its local coordinate system. (D) is representative of an application where the IMU is mounted on an HMD to control the camera motion in VR. Here, we see the base station from the VRduino's perspective along with a grid that represents the floor.*

tracking with IMUs. Pseudo code for stereo rendering with the estimated 6-DOF pose is outlined in Algorithm 2 and screenshots from a sample program are shown in Figures 6 (C) and (D).

5.3 Combining Position from Pose Tracking and Orientation from the IMU

The rotation estimated by the 6-DOF pose tracking algorithm is oftentimes very noisy and unreliable. A more robust way of integrating the estimated pose into the rendering pipeline is to use the position estimated from the 6-DOF pose tracking but the orientation from the IMU. Not only will this be more robust, usually the sampling rate of the IMU is much higher than the sweeping rate of the base station. Therefore, it makes sense to use the best of both worlds. More advanced sensor fusion algorithms would likely further improve the quality of the 6-DOF tracking. For now, let's assume we get the orientation quaternion from the IMU streamed at several hundred or a few thousand Hz and the estimated pose separately at 60-120 Hz. We will combine the two on the host computer.

To render a test object as outlined in Section 5.1, we can follow the description in that section but we will replace lines 7–9 of Algorithm 1 by the rotation estimated by the IMU. Remember that this approach is only possible if we also have the tilt angle of the base station, because the estimated 6-DOF pose is relative to the base station whereas the IMU measurements are relative to Earth. With the tile angle in hand, we apply the rotation estimated by the IMU and rotate out the tilt of the base station. Pseudo-code for this is outlined in Algorithm 3.

To use the IMU orientation and the position from pose tracking for controlling the camera or viewer in the virtual

Algorithm 2 Render stereo view with estimated pose

```
double pose_translation[3], yaw, pitch, roll;                    // position and orientation from pose tracking
updatePoseFromSerialStream(pose_translation,yaw,pitch,roll);     // get pose from serial stream
setProjectionMatrix();
glTranslatef(0, -$l_n$, -$l_h$);                                 // undo head & neck model after rotation
glScalef(-1,1,-1);                     // flip x and z axes to account for rotation of VRduino on HMD
glRotatef(-yaw, 0, 1, 0);                                        // head rotation
glRotatef(-pitch, 1, 0, 0);
glRotatef(-roll, 0, 0, 1);
glTranslatef(0, $l_n$, $l_h$);         // apply head & neck model by translating to the center of rotation
double ipd = 64.0;                                               // define interpupillary distance (in mm)
setLookAtMatrix( ± ipd/2, 0, 0, ± ipd/2, 0, -1, 0, 1, 0 );       // set view matrix for right/left eye
glTranslatef(-pose_translation[0], -pose_translation[1], -pose_translation[2]);   // head translation
glRotatef(-pitch_bs, 1, 0, 0);                                   // pitch of base station
glRotatef(-roll_bs, 0, 0, 1);                                    // roll of base station
drawScene();
```

Algorithm 3 Adjustment to Alg. 1: replace lines 7–9 with the following when working with a rotation quaternion

```
glRotatef(-pitch_bs, 1, 0, 0);                                   // pitch of base station
glRotatef(-roll_bs, 0, 0, 1);                                    // roll of base station
double q[4];
updateQuaternionFromSerialStream(q);                             // get quaternion from serial stream
float angle, axisX, axisY, axisZ;
quatToAxisAngle(q[0], q[1], q[2], q[3], angle, axisX, axisY, axisZ);   // convert quaternion to axis and angle
glRotatef(angle, axisX, axisY, axisZ);                           // rotate via axis-angle representation
```

Algorithm 4 Adjustment to Alg. 2: replace lines 6–8 with the following when working with a rotation quaternion

```
double q[4];
updateQuaternionFromSerialStream(q);                             // get quaternion from serial stream
float angle, axisX, axisY, axisZ;
quatToAxisAngle(q[0], q[1], q[2], q[3], angle, axisX, axisY, axisZ);   // convert quaternion to axis and angle
glScalef(-1,1,-1);                     // flip x and z axes to account for rotation of VRduino on HMD
glRotatef(-angle, axisX, axisY, axisZ);                          // rotate via axis-angle representation
glRotatef(roll_bs, 0, 0, 1);                                     // roll of base station
glRotatef(pitch_bs, 1, 0, 0);                                    // pitch of base station
```

environment, we follow a similar strategy as outlined in Section 5.2. Again, we have to combine the IMU measurements with the tilt of the base station as show in Algorithm 4. Without knowing the tilt angle of the base station, it would be much more challenging to combine IMU measurements and estimated pose.

6 Appendix A: Estimating Pose with the Nonlinear Levenberg-Marquardt Method

In this section, we derive a nonlinear optimization approach to pose tracking using the Levenberg-Marquardt (LM) algorithm. The derivation of the LM algorithm can be found in the lecture slides and you can find more details in standard optimization textbooks or on wikipedia[5]. For this approach, we need to specify which representation for the rotations we use. In the following derivations, we use Euler angles in the yaw-pitch-roll order, but you can derive a similar algorithm using quaternions. Using Equations 2 and 36, we can relate the set of pose parameters $\boldsymbol{p} = (\theta_x, \theta_y, \theta_z, t_x, t_y, t_z)$ via the function $g : \mathbb{R}^6 \to \mathbb{R}^9$ to the homography as

$$g(\boldsymbol{p}) = \begin{pmatrix} g_1(\boldsymbol{p}) \\ g_2(\boldsymbol{p}) \\ g_3(\boldsymbol{p}) \\ g_4(\boldsymbol{p}) \\ g_5(\boldsymbol{p}) \\ g_6(\boldsymbol{p}) \\ g_7(\boldsymbol{p}) \\ g_8(\boldsymbol{p}) \\ g_9(\boldsymbol{p}) \end{pmatrix} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{pmatrix} = \begin{pmatrix} \cos(\theta_y)\cos(\theta_z) - \sin(\theta_x)\sin(\theta_y)\sin(\theta_z) \\ -\cos(\theta_x)\sin(\theta_z) \\ t_x \\ \cos(\theta_y)\sin(\theta_z) + \sin(\theta_x)\sin(\theta_y)\cos(\theta_z) \\ \cos(\theta_x)\cos(\theta_z) \\ t_y \\ \cos(\theta_x)\sin(\theta_y) \\ -\sin(\theta_x) \\ -t_z \end{pmatrix} \tag{19}$$

Note that we are using all 9 elements of the homography matrix for this nonlinear approach. The elements of the homography matrix are only used as an intermediate variables. The function $f : \mathbb{R}^9 \to \mathbb{R}^8$ maps them to the projected 2D point coordinates of our 4 reference points as

$$f(\mathbf{h}) = \begin{pmatrix} f_1(\mathbf{h}) \\ f_2(\mathbf{h}) \\ \vdots \\ f_7(\mathbf{h}) \\ f_8(\mathbf{h}) \end{pmatrix} = \begin{pmatrix} x_1^n \\ y_1^n \\ x_2^n \\ y_2^n \\ x_3^n \\ y_3^n \\ x_4^n \\ y_4^n \end{pmatrix} = \begin{pmatrix} \frac{h_1 x_1 + h_2 y_1 + h_3}{h_7 x_1 + h_8 y_1 + h_9} \\ \frac{h_4 x_1 + h_5 y_1 + h_6}{h_7 x_1 + h_8 y_1 + h_9} \\ \vdots \\ \frac{h_1 x_4 + h_2 y_4 + h_3}{h_7 x_4 + h_8 y_4 + h_9} \\ \frac{h_4 x_4 + h_5 y_4 + h_6}{h_7 x_4 + h_8 y_4 + h_9} \end{pmatrix} \tag{20}$$

Equations 19 and 20 model the same image formation that we have been using throughout the document. The objective function that we are trying to minimize is

$$\begin{aligned} &\underset{\{\boldsymbol{p}\}}{\text{minimize}} \; \|\mathbf{b} - f(g(\boldsymbol{p}))\|_2^2 \qquad (21) \\ &= (x_1^n - f_1(g(\boldsymbol{p})))^2 + (y_1^n - f_2(g(\boldsymbol{p})))^2 + \ldots + (x_4^n - f_7(g(\boldsymbol{p})))^2 + (y_4^n - f_8(g(\boldsymbol{p})))^2 \end{aligned}$$

The Levenberg-Marquardt algorithm is an iterative method that starts from some initial guess $\boldsymbol{p}^{(0)}$ and then updates it as

$$\boldsymbol{p}^{(k)} = \boldsymbol{p}^{(k-1)} + \left(\mathbf{J}^T\mathbf{J} + \lambda \text{diag}\left(\mathbf{J}^T\mathbf{J}\right)\right)^{-1} (\mathbf{b} - f(g(\boldsymbol{p}))) \tag{22}$$

For these updates, we just need $\boldsymbol{p}^{(0)}$ as well as a user-defined parameter λ. The Jacobian matrix $\mathbf{J} \in \mathbb{R}^{8\times 6}$ includes the partial derivatives of the image formation model.

6.1 Partial Derivatives

Deriving partial derivatives can be a bit tedious, so we outline them in detail here as a reference. It is most convenient to derive the partial derivatives of f and g separately, as two separate Jacobian matrices $\mathbf{J}_f$ and $\mathbf{J}_g$. We can use the

[5] https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm

chain rule to assemble $\mathbf{J}$ from the individual Jacobian matrices as

$$\frac{\partial}{\partial \boldsymbol{p}} f\left(g\left(\boldsymbol{p}\right)\right) = \mathbf{J} = \mathbf{J}_f \cdot \mathbf{J}_g = \begin{pmatrix} \frac{\partial f_1}{\partial h_1} & \cdots & \frac{\partial f_1}{\partial h_9} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_8}{\partial h_1} & \cdots & \frac{\partial f_8}{\partial h_9} \end{pmatrix} \begin{pmatrix} \frac{\partial g_1}{\partial p_1} & \cdots & \frac{\partial g_1}{\partial p_6} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_9}{\partial p_1} & \cdots & \frac{\partial g_9}{\partial p_6} \end{pmatrix} \tag{23}$$

where $\mathbf{J}_f \in \mathbb{R}^{8\times 9}$ is the Jacobian matrix of the function $f\left(\mathbf{h}\right)$ and $\mathbf{J}_g \in \mathbb{R}^{9\times 6}$ is the Jacobian matrix of the function $g\left(\boldsymbol{p}\right)$.

You should derive the partial derivatives yourself, but for completeness we list them in the following. If you want to derive these yourself or you just want to verify the equations below, take a really close look at Equation 19 when deriving the entries of $\mathbf{J}_g$ and Equation 20 when deriving the entries of $\mathbf{J}_f$.

The formulas to compute the entries of the first row of the Jacobian matrix $\mathbf{J}_f$ are

$$\begin{aligned} &\tfrac{\partial f_1}{\partial h_1} = \tfrac{x_1}{h_7 x_1 + h_8 y_1 + h_9}, && \tfrac{\partial f_1}{\partial h_2} = \tfrac{y_1}{h_7 x_1 + h_8 y_1 + h_9}, && \tfrac{\partial f_1}{\partial h_3} = \tfrac{1}{h_7 x_1 + h_8 y_1 + h_9}, \\ &\tfrac{\partial f_1}{\partial h_4} = 0, && \tfrac{\partial f_1}{\partial h_5} = 0, && \tfrac{\partial f_1}{\partial h_6} = 0, \\ &\tfrac{\partial f_1}{\partial h_7} = -\left(\tfrac{h_1 x_1 + h_2 y_1 + h_3}{(h_7 x_1 + h_8 y_1 + h_9)^2}\right) x_1, && \tfrac{\partial f_1}{\partial h_8} = -\left(\tfrac{h_1 x_1 + h_2 y_1 + h_3}{(h_7 x_1 + h_8 y_1 + h_9)^2}\right) y_1, && \tfrac{\partial f_1}{\partial h_9} = -\left(\tfrac{h_1 x_1 + h_2 y_1 + h_3}{(h_7 x_1 + h_8 y_1 + h_9)^2}\right) \end{aligned} \tag{24}$$

The entries of the second row are

$$\begin{aligned} &\tfrac{\partial f_2}{\partial h_1} = 0, && \tfrac{\partial f_2}{\partial h_2} = 0, && \tfrac{\partial f_2}{\partial h_3} = 0, \\ &\tfrac{\partial f_2}{\partial h_4} = \tfrac{x_1}{h_7 x_1 + h_8 y_1 + h_9}, && \tfrac{\partial f_2}{\partial h_5} = \tfrac{y_1}{h_7 x_1 + h_8 y_1 + h_9}, && \tfrac{\partial f_2}{\partial h_6} = \tfrac{1}{h_7 x_1 + h_8 y_1 + h_9}, \\ &\tfrac{\partial f_2}{\partial h_7} = -\left(\tfrac{h_4 x_1 + h_5 y_1 + h_6}{(h_7 x_1 + h_8 y_1 + h_9)^2}\right) x_1, && \tfrac{\partial f_2}{\partial h_8} = -\left(\tfrac{h_4 x_1 + h_5 y_1 + h_6}{(h_7 x_1 + h_8 y_1 + h_9)^2}\right) y_1, && \tfrac{\partial f_2}{\partial h_9} = -\left(\tfrac{h_4 x_1 + h_5 y_1 + h_6}{(h_7 x_1 + h_8 y_1 + h_9)^2}\right) \end{aligned} \tag{25}$$

The remaining 6 rows of $\mathbf{J}_f$ can be computed using the same pattern, only the coordinates x_i, y_i have to be adjusted.

Similarly, here are formulas to compute the entries of the first row of the Jacobian matrix $\mathbf{J}_g$

$$\begin{aligned} \tfrac{\partial g_1}{\partial p_1} &= -\cos\left(\theta_x\right)\sin\left(\theta_y\right)\sin\left(\theta_z\right), \\ \tfrac{\partial g_1}{\partial p_2} &= -\sin\left(\theta_y\right)\cos\left(\theta_z\right) - \sin\left(\theta_x\right)\cos\left(\theta_y\right)\sin\left(\theta_z\right), \\ \tfrac{\partial g_1}{\partial p_3} &= -\cos\left(\theta_y\right)\sin\left(\theta_z\right) - \sin\left(\theta_x\right)\sin\left(\theta_y\right)\cos\left(\theta_z\right), \\ \tfrac{\partial g_1}{\partial p_4} &= 0, \quad \tfrac{\partial g_1}{\partial p_5} = 0, \quad \tfrac{\partial g_1}{\partial p_6} = 0 \end{aligned} \tag{26}$$

the entries of the second row

$$\frac{\partial g_2}{\partial p_1} = \sin\left(\theta_x\right)\sin\left(\theta_z\right), \quad \frac{\partial g_2}{\partial p_2} = 0, \quad \frac{\partial g_2}{\partial p_3} = -\cos\left(\theta_x\right)\cos\left(\theta_z\right), \quad \frac{\partial g_2}{\partial p_4} = 0, \quad \frac{\partial g_2}{\partial p_5} = 0, \quad \frac{\partial g_2}{\partial p_6} = 0 \tag{27}$$

the third row

$$\frac{\partial g_3}{\partial p_1} = 0, \quad \frac{\partial g_3}{\partial p_2} = 0, \quad \frac{\partial g_3}{\partial p_3} = 0, \quad \frac{\partial g_3}{\partial p_4} = 1, \quad \frac{\partial g_3}{\partial p_5} = 0, \quad \frac{\partial g_3}{\partial p_6} = 0 \tag{28}$$

the fourth row

$$\begin{aligned} \tfrac{\partial g_4}{\partial p_1} &= \cos(\theta_x)\sin(\theta_y)\cos(\theta_z), \\ \tfrac{\partial g_4}{\partial p_2} &= -\sin(\theta_y)\sin(\theta_z) + \sin(\theta_x)\cos(\theta_y)\cos(\theta_z), \\ \tfrac{\partial g_4}{\partial p_3} &= \cos(\theta_y)\cos(\theta_z) - \sin(\theta_x)\sin(\theta_y)\sin(\theta_z), \\ \tfrac{\partial g_4}{\partial p_4} &= 0, \quad \tfrac{\partial g_4}{\partial p_5} = 0, \quad \tfrac{\partial g_4}{\partial p_6} = 0 \end{aligned} \tag{29}$$

the fifth row

$$\frac{\partial g_5}{\partial p_1} = -\sin(\theta_x)\cos(\theta_z), \quad \frac{\partial g_5}{\partial p_2} = 0, \quad \frac{\partial g_5}{\partial p_3} = -\cos(\theta_x)\sin(\theta_z), \quad \frac{\partial g_5}{\partial p_4} = 0, \quad \frac{\partial g_5}{\partial p_5} = 0, \quad \frac{\partial g_5}{\partial p_6} = 0 \tag{30}$$

the sixth row

$$\frac{\partial g_6}{\partial p_1} = 0, \quad \frac{\partial g_6}{\partial p_2} = 0, \quad \frac{\partial g_6}{\partial p_3} = 0, \quad \frac{\partial g_6}{\partial p_4} = 0, \quad \frac{\partial g_6}{\partial p_5} = 1, \quad \frac{\partial g_6}{\partial p_6} = 0 \tag{31}$$

the seventh row

$$\frac{\partial g_7}{\partial p_1} = -\sin(\theta_x)\sin(\theta_y), \quad \frac{\partial g_7}{\partial p_2} = \cos(\theta_x)\cos(\theta_y), \quad \frac{\partial g_7}{\partial p_3} = 0, \quad \frac{\partial g_7}{\partial p_4} = 0, \quad \frac{\partial g_7}{\partial p_5} = 0, \quad \frac{\partial g_7}{\partial p_6} = 0 \tag{32}$$

the eigth row

$$\frac{\partial g_8}{\partial p_1} = -\cos(\theta_x), \quad \frac{\partial g_8}{\partial p_2} = 0, \quad \frac{\partial g_8}{\partial p_3} = 0, \quad \frac{\partial g_8}{\partial p_4} = 0, \quad \frac{\partial g_8}{\partial p_5} = 0, \quad \frac{\partial g_8}{\partial p_6} = 0 \tag{33}$$

and the ninth row

$$\frac{\partial g_9}{\partial p_1} = 0, \quad \frac{\partial g_9}{\partial p_2} = 0, \quad \frac{\partial g_9}{\partial p_3} = 0, \quad \frac{\partial g_9}{\partial p_4} = 0, \quad \frac{\partial g_9}{\partial p_5} = 0, \quad \frac{\partial g_9}{\partial p_6} = -1 \tag{34}$$

With these partial derivatives, we are now ready to implement the entire nonlinear algorithm for pose estimation.

6.2 Pseudo Code

To help you with the implementation, Algorithm 1 outlines pseudo code for pose tracking with Levenberg-Marquardt.

Algorithm 5 Levenberg-Marquardt for Pose Tracking

```
1: initialize $\boldsymbol{p}$ and $\lambda$
2: for $k = 1$ to $maxIters$
3:     $\mathbf{f} = \text{eval\_objective}(\boldsymbol{p})$            // see Eq. 21
4:     $\mathbf{J}_g = \text{compute\_jacobian\_g}(\boldsymbol{p})$      // see Eqs. 26–34
5:     $\mathbf{J}_f = \text{compute\_jacobian\_f}(\boldsymbol{p})$      // see Eqs. 24–25
6:     $\mathbf{J} = \mathbf{J}_f \cdot \mathbf{J}_g$
7:     $\boldsymbol{p} = \boldsymbol{p} + \text{inv}\left(\mathbf{J}^T\mathbf{J} + \lambda\,\text{diag}\left(\mathbf{J}^T\mathbf{J}\right)\right) \cdot (\mathbf{b} - \mathbf{f})$
8: end for
```

7 Appendix B: Connection to Camera-based Tracking and Pose Estimation

As a side note that is not relevant for the VRduino, camera-based tracking systems oftentimes also model lens distortions and a mapping from metric space to pixel coordinates via camera matrix $\mathbf{K}$

$$\begin{pmatrix} x^d \\ y^d \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \begin{pmatrix} x^n \left(1 + k_1 r^2 + k_2 r^4 + \ldots\right) \\ y^n \left(1 + k_1 r^2 + k_2 r^4 + \ldots\right) \\ 1 \end{pmatrix} \tag{35}$$

where f_x, f_y are scale factors, known as focal length, that transform the normalized 2D image coordinates into pixel coordinates. The principle point c_x, c_y models the center of the lens in the image. Together, focal length and principle point define the *intrinsic parameters* of a camera, which are encoded in the camera matrix $\mathbf{K}$. This is similar to the viewport transform in the graphics pipeline and it is applied *after* the perspective divide. The distortion parameters of the Brown-Conrady model k_1, k_2 are similar to those we used for the lens distortion of the head mounted display in week 4 and $r = \sqrt{x^{n2} + y^{n2}}$.

In camera-based optical tracking systems, we often encounter two slightly different but related problems: *pose tracking* and *camera calibration*.

Pose tracking is similar to what we do with the VRduino: given all the intrinsic camera parameters f_x, f_y, c_x, c_y and the distortion coefficients $k_1, k_2, \ldots$, we wish to estimate the pose of the camera from several observed reference points in a single camera image. This is done by finding the distorted coordinates x_i^d, y_i^d in a recorded image, undistort them using the known intrinsic parameters to get x_i^n, y_i^n, and then follow the approaches outlined in Sections 4 and 6. The homography method is often used as an initialization step, which is useful as the initial guess for the refinement step, i.e. the Levenberg-Marquardt method.

For the camera calibration problem, we do not know what the intrinsic camera parameters and the distortion coefficients are, so we have to estimate them along with the *extrinsic parameters* (i.e., the pose). This is not possible with the homography method, but can be done with the Levenberg-Marquardt (LM) method. In this case, we can use an initial guess of the intrinsic parameters, run the homography method to get an estimate for the pose, and then run LM. As opposed to the LM approach outlined in Section 6, we will have to include the intrinsic parameters in our objective function and in the Jacobian matrices. You can use the derivations in Section 6 as a starting point if you'd like to derive the LM formulation for camera calibration. Also, camera calibration is also usually done with a set of camera images, each showing the same calibration target with a different pose. So the problem is to estimate the pose for each of these images simultaneously and intrinsic parameters (which are shared between all images). Once the intrinsic parameters are calibrated, we can revert back to solving the pose tracking problem in real time. Camera calibration is usually done only once (or whenever the camera parameters change, like zoom and focus) as a preprocessing step.

For pose tracking with the VRduino, we can omit distortions and intrinsic camera parameters, which makes the problem much easier than pose estimation with cameras. For details on general camera-based tracking and calibration consult [Heikkila and Silven 1997; Zhang 2000; CV 2014; Bouguet 2015] or standard computer vision textbooks, such as [Hartley and Zisserman 2004; Szeliski 2010]. Personally, I recommend to look into the Matlab Camera Calibration Toolbox [Bouguet 2015] to get started on pose tracking with cameras, because it not only has excellent online documentation but also includes reference code that works robustly and is easy to understand.

8 Appendix C: Rotations with Euler Angles

In this appendix, we outline a few useful formulas that may come in handy if you want to work with Euler angles (which is usually not recommended due to the gimbal lock problem and other issues).

As discussed in class, there are several ways we can represent rotations, for example using rotation matrices, Euler angles, or quaternions. A rotation only has three degrees of freedom, so rotation matrices with 9 elements are redundant. Euler angles explicitly represent rotations around the coordinate axes and remove that ambiguity, because we only need three of them.. However, it is important to define in which order these rotations are applied, because they are not commutative. For three rotation angles, there are many different possible choices for the order in which they are applied and this has to be defined somewhere. Let's work with the order yaw-pitch-roll for now, so that a rotation around the y axis is applied first, then around the x axis, and finally around the z axis.

Given rotation angles $\theta_x, \theta_y, \theta_z$, which represent rotations around the x, y, z axes, respectively, we can then compute the rotation matrix by multiplying rotation matrices for each of these as

$$\underbrace{\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}}_{\mathbf{R}} = \underbrace{\begin{pmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{R}_z(\theta_z)} \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{pmatrix}}_{\mathbf{R}_x(\theta_x)} \underbrace{\begin{pmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{pmatrix}}_{\mathbf{R}_y(\theta_y)}$$

$$= \begin{pmatrix} \cos(\theta_y)\cos(\theta_z) - \sin(\theta_x)\sin(\theta_y)\sin(\theta_z) & -\cos(\theta_x)\sin(\theta_z) & \sin(\theta_y)\cos(\theta_z) + \sin(\theta_x)\cos(\theta_y)\sin(\theta_z) \\ \cos(\theta_y)\sin(\theta_z) + \sin(\theta_x)\sin(\theta_y)\cos(\theta_z) & \cos(\theta_x)\cos(\theta_z) & \sin(\theta_y)\sin(\theta_z) - \sin(\theta_x)\cos(\theta_y)\cos(\theta_z) \\ -\cos(\theta_x)\sin(\theta_y) & \sin(\theta_x) & \cos(\theta_x)\cos(\theta_y) \end{pmatrix} \tag{36}$$

For some applications, we may wish to extract the Euler angles from a 3×3 rotation matrix. We can do that using these formulas:

$$\begin{aligned} r_{32} &= \sin(\theta_x) && \Rightarrow \theta_x = \sin^{-1}(r_{32}) = \operatorname{asin}(r_{32}) \\ \frac{r_{31}}{r_{33}} &= -\frac{\cos(\theta_x)\sin(\theta_y)}{\cos(\theta_x)\cos(\theta_y)} = -\tan(\theta_y) && \Rightarrow \theta_y = \tan^{-1}\left(-\frac{r_{31}}{r_{33}}\right) = \operatorname{atan2}(-r_{31}, r_{33}) \\ \frac{r_{12}}{r_{22}} &= -\frac{\cos(\theta_x)\sin(\theta_z)}{\cos(\theta_x)\cos(\theta_z)} = -\tan(\theta_z) && \Rightarrow \theta_z = \tan^{-1}\left(-\frac{r_{12}}{r_{22}}\right) = \operatorname{atan2}(-r_{12}, r_{22}) \end{aligned} \tag{37}$$

Note, however, that this way of extracting of the Euler angles is ambiguous. Even though whatever angles you extract this way will result in the correct rotation matrix, if the latter was generated from a set of Euler angles in the first place, you are not guaranteed to get exactly those back.

9 Appendix D: Rotations with Quaternions

Here, we outline two useful equations that are important for working with quaternions. Remember that quaternions represent a rotation using an axis-angle representation and only unit quaterions are valid rotations.

Given a unit quaternion $q = q_w + iq_x + jq_y + kq_z$, $\|q\|_2 = 1$, we can compute the corresponding rotation matrix as

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} q_w^2 + q_x^2 - q_y^2 - q_z^2 & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y \\ 2q_xq_y + 2q_wq_z & q_w^2 - q_x^2 + q_y^2 - q_z^2 & 2q_yq_z - 2q_wq_x \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{pmatrix} \tag{38}$$

We can also convert a 3×3 rotation matrix to a quaterion as

$$q_w = \frac{\sqrt{1 + r_{11} + r_{22} + r_{33}}}{2}, \qquad q_x = \frac{r_{32} - r_{23}}{4q_w}, \qquad q_y = \frac{r_{13} - r_{31}}{4q_w}, \qquad q_z = \frac{r_{21} - r_{12}}{4q_w} \tag{39}$$

Due to numerical precision of these operations, you may want to re-normalize the quaterion to make sure it has unit length. Also note that the two quaterions q and $-q$ result in the same rotation. If you test converting a quaternion to a matrix and back, you may not get the exact same numbers, but the rotation matrix corresponding to each of the quaternions should be the same.

References

BOUGUET, J.-Y., 2015. Camera calibration toolbox for matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/`.

CV, O., 2014. Opencv: Camera calibration and 3d reconstruction. `http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html`.

HARTLEY, R., AND ZISSERMAN, A. 2004. *Multiple View Geometry in Computer Vision*. Cambridge University Press.

HEIKKILA, J., AND SILVEN, O. 1997. A four-step camera calibration procedure with implicit image correction. In *Proc. CVPR*.

LEPETIT, V., MORENO-NOGUER, F., AND FUA, P. 2008. Epnp: An accurate o(n) solution to the pnp problem. *International Journal of Computer Vision 81*, 2.

RASKAR, R., NII, H., DEDECKER, B., HASHIMOTO, Y., SUMMET, J., MOORE, D., ZHAO, Y., WESTHUES, J., DIETZ, P., BARNWELL, J., NAYAR, S., INAMI, M., BEKAERT, P., NOLAND, M., BRANZOI, V., AND BRUNS, E. 2007. Prakash: Lighting aware motion capture using photosensing markers and multiplexed illuminators. *ACM Trans. Graph. (SIGGRAPH) 26*, 3.

SZELISKI, R. 2010. *Computer Vision: Algorithms and Applications*. Springer.

ZHANG, Z. 2000. A flexible new technique for camera calibration. *IEEE Trans. PAMI 22*, 11, 1330–1334.