Course Notes: 6-DOF Pose Tracking

CSE 490V: Virtual Reality Systems

Gordon Wetzstein and Douglas Lanman

This document serves as a supplement to the material discussed in Lectures 11 and 12. Note that this is not intended to be a comprehensive review of positional tracking for virtual reality applications. Rather, this document provides an intuitive introduction to the basic mathematical concepts of pose estimation, particularly for the Stanford “VRduino” and the related ARToolKit markers used in CSE 490V.

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, e.g., a headset, a controller, a marker tag, 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. Currently, the most widely used technology 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 headsets). The pose 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 (PNP) problem* and is crucial for camera calibration, 3D computer vision, and within other fields.

The Valve Index and HTC Vive also use an optical tracking system, but rather than applying a camera to observe LEDs on a headset, these systems use 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. Valve calls their implementation of this technology *Lighthouse*, which adopts a different form of temporally structured illumination. Specifically, the Lighthouse projector or *base station* sweeps horizontal and vertical laser stripes across the room, hence the name. It does this very rapidly – 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. Note that the Stanford VRduino has four photodiodes, each of which are similar to those used by Valve’s and HTC’s VR controllers and headsets. It also has a microcontroller in order to implement the necessary computations for pose estimation.

In the remainder of this document, we will review the fundamental mathematics of pose tracking using the VRduino. Note that many of these derivations can be similarly applied to related pose tracking methods (e.g., using ARToolKit markers observed by a calibrated camera for Homework 6 in CSE 490V). The methods reviewed in this document will likely not work quite as well as any commercial solutions. (For example, the VRduino only uses 4 photodiodes in a planar configuration, rather than a large number of photodiodes in a 3D arrangement.) Nevertheless, these methods are educational and intended to help you understand and implement pose tracking from scratch.

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

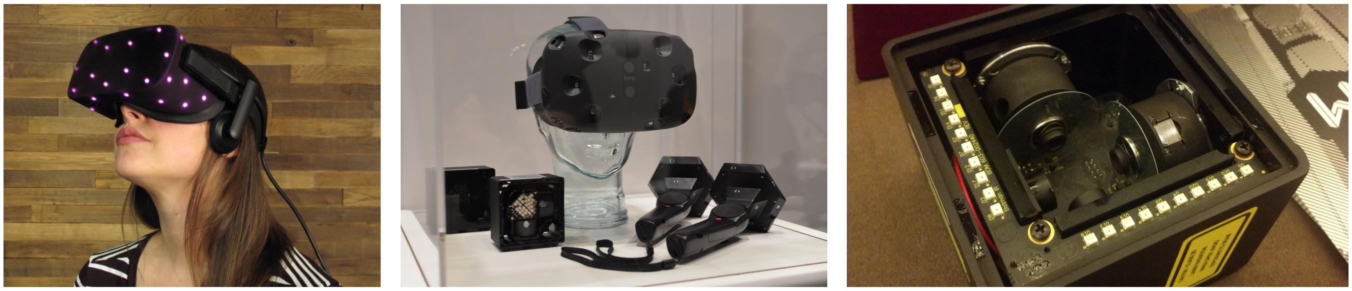

Figure 1: *Examples of optical tracking in VR. Left: near-infrared (NIR) LEDs of the Oculus Rift recorded with a camera that is sensitive to NIR (image reproduced from ifixit.com). Center: HTC Vive headset and controllers with exposed photodiodes (image reproduced from roadtovr.com). Right: disassembled 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).*

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

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

The coordinates $(x^c,\ y^c,\ z^c)$ 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.

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 $(x_i,\ y_i,\ z_i,\ 1)$, $i = 1 \ldots M$. 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, $z_i = 0,\ \forall i$ and Equation 1 reduces to

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

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

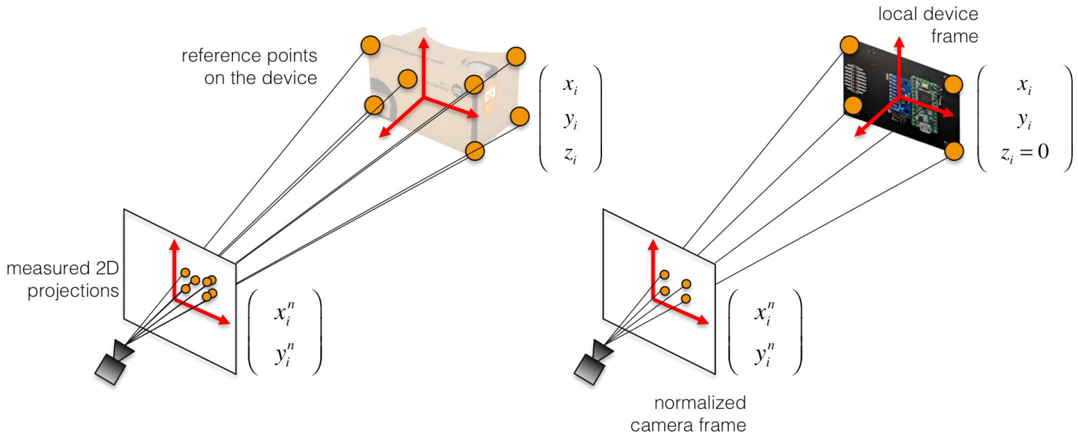
reference points
on the device
$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$
measured 2D
projections
$\begin{pmatrix} x_i^n \\ y_i^n \end{pmatrix}$
local device
frame
$\begin{pmatrix} x_i \\ y_i \\ z_i = 0 \end{pmatrix}$
$\begin{pmatrix} x_i^n \\ y_i^n \end{pmatrix}$
normalized
camera frame

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).*

Similar to the graphics pipeline, we now perform the perspective divide by dividing x^c and y^c by the distance to the camera z^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

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

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[1] [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.

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 7.

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.

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

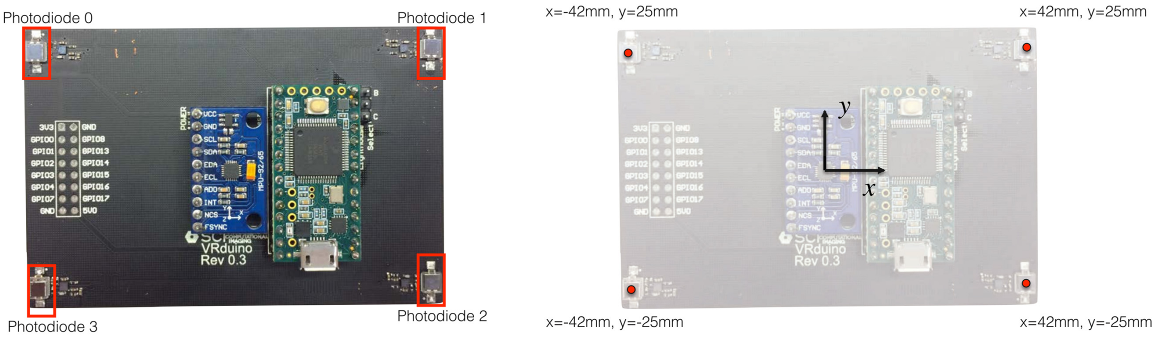

Figure 3: *The VRduino. Left: photograph of VRduino showing photodiodes, the inertial measurement unit, and the microcontroller. Right: the local coordinate system is centered in the VRduino, which is also directly in the center of the IMU. The x and y locations of the photodiodes in the local frame 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).*

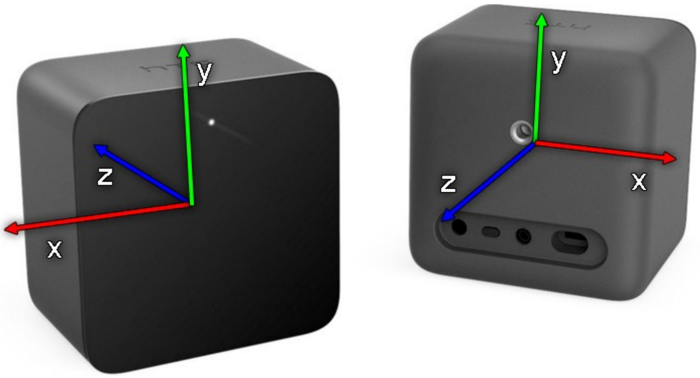

Figure 4: *HTV Vive base station (front and back) with local coordinate system illustrated.*

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 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[2].

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.

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

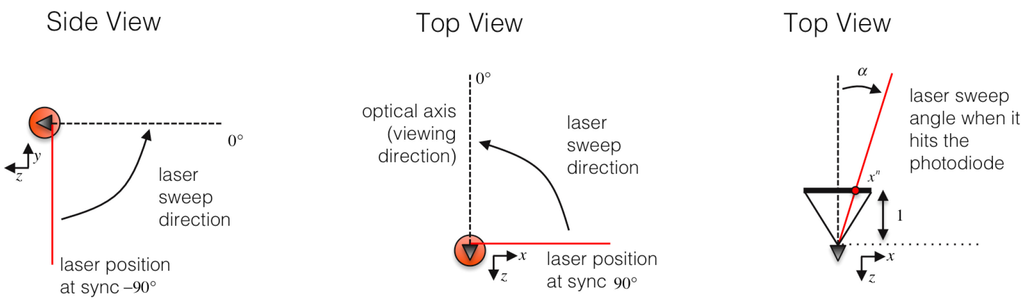

Figure 5: *Laser sweep directions of the Lighthouse base station. Left: the vertical sweep moves bottom to top. At the time of the sync pulse, the sweeping laser is pointing down. Center: the horizontal sweep is right to left. At the time of the sync pulse, the sweeping laser is pointing right. Right: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

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.

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

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

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

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

which we can multiply by the denominator

$$\left(h_7 x + h_8 y + 1\right) x^n = h_1 x + h_2 y + h_3, \qquad \left(h_7 x + h_8 y + 1\right) y^n = h_4 x + h_5 y + h_6, \tag{8}$$

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. 21) or Euler angles (see Eq. 19).

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 was described in class.

5 Appendix A: Connection to Camera-based Tracking and Pose Estimation

Camera-based tracking systems, such as the ARToolKit framework applied in Homework 6 of CSE 490V, often also model lens distortions and a mapping from metric space to pixel coordinates via the camera matrix $\mathbf{K}$

$$\begin{pmatrix} x^d \\ y^d \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \begin{pmatrix} x^n \left(1 + k_1 r^2 + k_2 r^4 + \ldots\right) \\ y^n \left(1 + k_1 r^2 + k_2 r^4 + \ldots\right) \\ 1 \end{pmatrix} \tag{17}$$

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 model k_1, k_2 are similar to those we used for the lens distortion of the head mounted display in Homework 4, where $r = \sqrt{x^{n2} + y^{n2}}$.

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 was outlined in this document for 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 (or equivalently an ARToolKit marker tag) 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 the recorded image, undistorting them (using the known intrinsic parameters) to get x_i^n, y_i^n, and then follow the approaches outlined in Section 4.

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, as reviewed in class. 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. Note that we will have to include the intrinsic parameters in our objective function and in the Jacobian matrices. 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 along with the 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].

6 Appendix B: 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

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

$$= \begin{pmatrix} \cos(\theta_y)\cos(\theta_z) - \sin(\theta_x)\sin(\theta_y)\sin(\theta_z) & -\cos(\theta_x)\sin(\theta_z) & \sin(\theta_y)\cos(\theta_z) + \sin(\theta_x)\cos(\theta_y)\sin(\theta_z) \\ \cos(\theta_y)\sin(\theta_z) + \sin(\theta_x)\sin(\theta_y)\cos(\theta_z) & \cos(\theta_x)\cos(\theta_z) & \sin(\theta_y)\sin(\theta_z) - \sin(\theta_x)\cos(\theta_y)\cos(\theta_z) \\ -\cos(\theta_x)\sin(\theta_y) & \sin(\theta_x) & \cos(\theta_x)\cos(\theta_y) \end{pmatrix} \tag{18}$$

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

$$\begin{aligned} r_{32} &= \sin(\theta_x) & &\Rightarrow \theta_x = \sin^{-1}(r_{32}) = \operatorname{asin}(r_{32}) \\ \frac{r_{31}}{r_{33}} &= -\frac{\cos(\theta_x)\sin(\theta_y)}{\cos(\theta_x)\cos(\theta_y)} = -\tan(\theta_y) & &\Rightarrow \theta_y = \tan^{-1}\left(-\frac{r_{31}}{r_{33}}\right) = \operatorname{atan2}(-r_{31}, r_{33}) \\ \frac{r_{12}}{r_{22}} &= -\frac{\cos(\theta_x)\sin(\theta_z)}{\cos(\theta_x)\cos(\theta_z)} = -\tan(\theta_z) & &\Rightarrow \theta_z = \tan^{-1}\left(-\frac{r_{12}}{r_{22}}\right) = \operatorname{atan2}(-r_{12}, r_{22}) \end{aligned} \tag{19}$$

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.

7 Appendix C: 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

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} q_w^2 + q_x^2 - q_y^2 - q_z^2 & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y \\ 2q_xq_y + 2q_wq_z & q_w^2 - q_x^2 + q_y^2 - q_z^2 & 2q_yq_z - 2q_wq_x \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{pmatrix} \tag{20}$$

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

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

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.