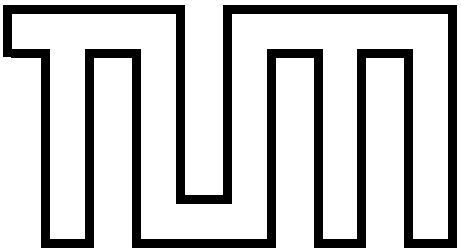


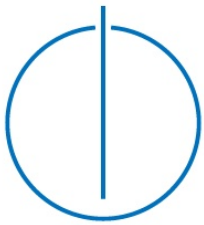
Technische Universität München

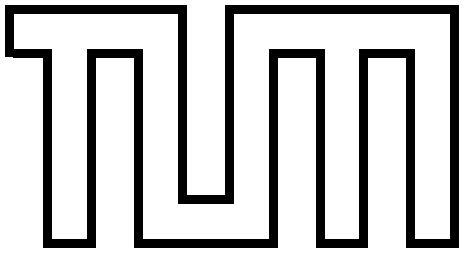
Fakultät für Informatik

Bachelor's Thesis in Informatics: Game Engineering

Pose-aware rendering of live ultrasound data for mixed medical AR

Felix Scheidhammer





Technische Universität München

Fakultät für Informatik

Bachelor's Thesis in Informatics: Game Engineering

Pose-aware rendering of live ultrasound data for mixed medical AR

Live Rendering von Ultraschall-Bildern in einer medizinischen mixed Reality Umgebung

Author: Felix Scheidhammer

Supervisor: Prof. Dr. Nassir Navab

Advisor: Benjamin Busam, Dr. Christoph Hennersperger

Submission: 15.01.2018

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 15.01.2018

(Felix Scheidhammer)

Abstract

Using a ultrasound probe is a hard task, it may take half a year or even a year until one can use ultrasound probes efficient. Right now the standard way to use ultrasound is with a mounted screen showing only the input of the probe. It is reasonable to use an augmented reality setup instead to show the ultrasound images, since this shows directly where the image lies in the patient. This also works as a learning base for not proficient users to learn the efficient usage of ultrasound. It is also quite hard to find old features again in the ultrasound image once the probe has moved.
This thesis tries to provide an AR application based on android devices in real time with live ultrasound data to guide the user to desired feature poses with an outside optical tracking system tracking both probe and android device. In this thesis a ad hoc hand-eye calibration is being provided, with [Tsai 89].
A new hand-eye calibration scheme is also provided if the pose of the calibration jig is known relative to the tracking system. This can be achieved via computing the hand-eye calibration at several stations, and then averaging the transformation. The averaging is being done via first finding the L1-mean of the rotations via Weiszfeld and then averaging the translation.
During the hand-eye calibration also the distortion of the camera is being corrected to correctly blend the ultrasound into the field of view of the camera. For this a look-up table is being created to correct the distortion for each pixel in the camera captures.
The live guidance to desired poses is being rendered via showing the coordinate axes of the goal pose at its origin in the AR application and at the same time the ultrasound with its coordinate system. In case of invisibility also the relative position of the goal to the probe is also provided, together with a description of the necessary rotation to re-orient the ultrasound probe.

Contents

Chapter 1

Introduction

As has been mentioned in the Abstract, this thesis tries to provide an augmented reality application for pose-aware rendering of live ultrasound data with live guidance. This means, that the android device renders the field of view of the camera. The user uses a ultrasound probe inside the field of view of the camera. The live images from the ultrasound probe will be rendered in real time into the field of view of the camera where they where captured.
And now comes the clue in the application: once the user sees something interesting in the ultrasound images, he may store the current pose of the ultrasound probe and later on the application will guide the user back to the stored pose.
For this we need an ultrasound probe calibrated to a tracking system. A Tracking system which tracks both the android device and the probe, and an android device with a camera. First, the camera on the android device will be calibrated with a calibration pattern to find the pose of the camera relative to the marker on the android device. Next distortion inside the camera will be corrected, which is necessary for a correct fusion of camera and ultrasound images. Then the ultrasound images will be rendered into the field of view of the camera according to the poses tracked in the tracking system.
The calibration of the camera relative to the marker of the tracking system is the so-called hand-eye calibration. [Tsai 89] is being used in the standard case. This means, that the user first has to capture multiple images of the calibration pattern, and with this information the hand-eye calibration is being computed.
In the other case - when the calibration pattern is known relative to the tracking system - a new method to compute the hand-eye calibration is being used. In this case, the hand-eye calibration could be computed with only one single image, as will be explained later on. But since there will most likely occur error, still multiple images of the calibration pattern

have to be taken. For each of these images the hand-eye calibration can be computed. Every image should result in the same calibration. This way we can compute the average of those hand-eye calibrations to minimize the error.
The hand-eye calibration is a transformation matrix and consists of a rotation and translation. To average such a transformation, first the rotation will be averaged via [I Ha 11] to find the L1-mean. Afterwards with this mean rotation the arithmetic mean of the translation will be computed.
With the multiple images also the distortion and projection into the field of view of of the camera can be computed using the library OpenCV [Open]. The distortions will be corrected using a look-up table to undistort the camera images in every frame.
The complete setup of the calibration is desired to be performable by any user, in a reasonable time, like 5 minutes. To render the ultrasound images on the correct pose inside the field of view of the camera, the probe is also being tracked, and the pose of the ultrasound relative to the camera is then being computed with the transformation computed during the hand-eye calibration.
Up till now it would still be possible for every part to move independently. The optical tracking system could be re-stationed, the ultrasound can be moved. The only important thing after all calibrations is, that the marker mounted onto the android device and the marker mounted onto the ultrasound probe should be fixed rigidly. Otherwise the calibration of the ultrasound to the marker or the hand-eye calibration would be wrong.
For the live guidance the pose of the ultrasound probe at the desired pose will be stored relative to the tracking system. After this point the tracking system should not be moved, otherwise this pose would become wrong. Then if the user desires to be guided to the stored pose - further on called the goal pose - he can make the guidance visible. The goal pose is being rendered at its position in the field of view of the camera just as the ultrasound pose was being transformed into the field of view of the camera. It might be, that the goal pose is currently outside of the field of view of the camera. In this case the user can decide to see the relative translation vector from the current pose to the relative pose. And at the same time the necessary rotation to align the orientations of both poses will be shown as X-Y-Z fixed angles.

in the following chapters first the related work will be discussed. In the preliminaries all necessary prior knowledge to understand the methods used in this thesis will be provided. Then all the methods implemented in this application will be provided. At last the results will show how well the final application performed, and the conclusion will hint to possible future work.

Chapter 2

Related Work

Augmented reality approaches ultrasound imaging are not new. Already in 1992 [Baju 92] provided a system to show ultrasound augmented into a head mounted display (HMD). They used for the hand-eye calibration a calibration jig known relative to the tracking system - in their case a polhemus tracker as described in [Baju 92]. With capturing the calibration jig from only one pose they computed the hand-eye calibration, and further refined the result with "manual adjustments".
And needle guidance has already been augmented with ultrasound, creating volumetric information via ultrasound, or only the current ultrasound image. First in [Stat 96] with a mechanical tracker of ultrasound probe and the video-seethrough HMD has been tracked via magnetic tracking and additional vision based landmark tracking. then this has been improved in [Rose 01]. In the newer approach the mechanical tracking of the ultrasound has been exchanged with opto-electronic tracking. In both cases the HMD has been calibrated to the tracking system similar to [Baju 95]. This is also a calibration via a known calibration jig, just as [Baju 92] did. But instead of "manual adjustments" now "manual feedback to converge on a solution". So, both [Baju 92] and [Rose 01] [Stat 96] rely on a long calibration using only one capture and manual corrections. This way the android device would not be interchangeable.
Regarding optical tracking, also inside-out tracking should be mentioned. For example [Stol 14] mounted a stereo camera setup onto the ultrasound probe and implemented needle guidance.
Even already for Android and iOs augmented reality applications have been implemented in [Kiss 14] and [Palm 15]. [Kiss 14] implemented a tablet based application which finds feature points in the ultrasound image of the heart and then fuses the ultrasound image with a 3D object of the heart to provide the user with information of the anatomy.

[Palm 15] Did the same, but additionally rendered the ultrasound image in the view of the camera. [Kiss 14] did not use the camera. [Palm 15] used marker based tracking to render both the 3D-object of the heart and the ultrasound. But they did not calibrate the ultrasound probe towards the marker, the ultrasound image is simply being rendered at the position of the marker, not at the correct pose.

Chapter 3

Preliminaries

Pinhole camera

A Pinhole Camera is a standard way to define cameras in perspective rendering. In figure 3.1 you can see one example of how a pinhole camera works. Ths pinhole camera is a box with a very small hole, the pinhole. The light reflected from the tree is being sent through this hole. Since light can be imagined as a ray, the light then projects the image of the tree through this hole onto the plane in the backside of the camera. This plane then shows the resulting image.

Since this results in a flipped image of the tree, the plane is thought as to be infront of this hole at the same distance to the hole. See Figure 3.2.
the pinhole camera model is now being defined according to translated [Simo 07]:
"The pinhole camera consists of a point, the optical centre, and a plane in the 3-dimensional room on which the image is being projected. This plane is the image plane. The Pose of the optical centre is also the centre of the camera." The pose of the camera is being described through the position of the centre of the camera F_c.
"The orthogonal projection on the optical centre onto the image plane is being called the principal point" (c_x, c_y).
The image plane is thought of as being parallel to the xy-plane at z=1, and thus can be referred to as the $\{z = 1\}$-Plane. In reality this is not necessarily the case. Then the distance of the image plane to the centre is the focal length f.

The pinhole camera model can be described through a set of parameters. There are two groups of parameters; intrinsic and extrinsic parameters. The intrinsic parameters describe

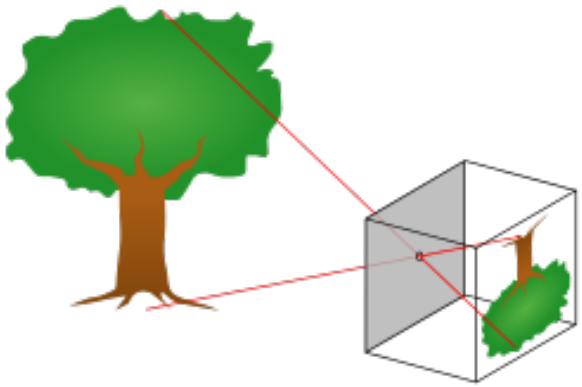


Figure 3.1: Example of a pinhole camera. Source: https://upload.wikimedia.org/wikipedia/commons/thumb/3/3b/Pinhole-camera.svg/256px-Pinhole-camera.svg.png

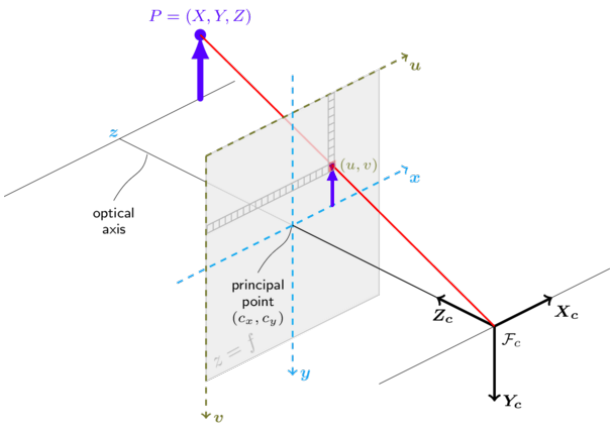


Figure 3.2: The pinhole camera model

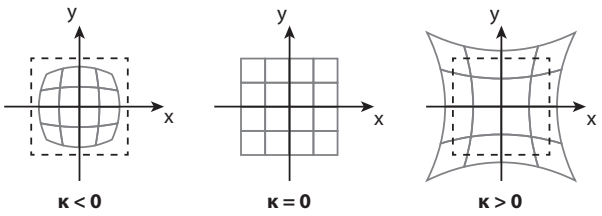


Figure 3.3: Radial distortion. In the middle the perfect case, the left and right side are two possibly occurring radial distortions. B.Busam. Projective Geometry and 3D point cloud matching. MSc Thesis. Technische Universität München. 2014/4.

the projection of the world onto the image plane, furthermore into the pixel-space of the resulting image. The extrinsic parameters describe the position of the camera centre relative to the world coordinate system. The extrinsic parameters consist of a orientation and a position of the camera centre, together the pose or transformation from world to camera centre T_{wc}. Transformations are being described in a different section. For now it is only important to understand, that a object described via world coordinates first has to be transformed to be described relative to the camera centre in order to be projected onto the image plane.
The intrinsic parameters can be described with the camera matrix K from [Open]:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

With f_x, f_y as the focal lengths expressed in pixel units.
With both extrinsic and extrinsic parameters, points $X, Y, Z, 1$ in the real world coordinate system can be projected onto the pixel of the image via, if $Z \neq 0$:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K * T_{wc} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.2}$$

Calibration of the Camera

The pinhole camera model may be a nice way to project objects into the view of the camera, but it is only a model. In reality there are some aspect which will result in a change in the view of the camera. For example the lens may be placed with a very small

focal distance, for example for zooming or wide-angle cameras as fish-eye cameras do. Or the lens may be slightly misplaced due to fabrication errors. The first case will lead to radial distortion, as depicted in Figure 3.3. This distortion depends solely on the radius to the principal point, with its magnitude depending on the length of the radius. This distortion either pulls pixel closer to the center, as in barrel distortion - the left example - or away from the center, called pincussion distortion - on the right in the figure. The most used model for describing radial distortion is the Brown-Conrady Model [BROW 66] with $r_d^2 = x_d^2 + y_d^2$ as the radius of the distorted pixel p_d.

$$r_u = r_d + k_1 * r_d^2 + k_2 * r_d^4 + k_3 * r_d^6 + ... \tag{3.3}$$

The tangential distortion occurs as described by fabrication errors, and is results in a shift of the image, as described in [Simo 07]. Both the radial and tangential distortion can be combined into one distortion model. OpenCV itself uses both distortions, but the tangential parameters can be set to zero to only use radial distortion. It also does not use the Brown-Conrady model for radial distortion, but a similar polynomial, the rational model. The model in OpenCV as of 2.4.11 is as follows:

$$x_u = x * \frac{1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6}{1 + k_4 * r^2 + k_5 * r^4 + k_6 * r^6} + 2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2) \tag{3.4}$$

$$y_u = y * \frac{1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6}{1 + k_4 * r^2 + k_5 * r^4 + k_6 * r^6} + p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x * y \tag{3.5}$$

Rotation

For pose-aware rendering knowledge about poses is necessary. A Pose is a description of the 6 degrees of freedom of a object. This is the combination of position and orientation relative to a 3D vector space. Such a pose relative to a frame can be described as a transformation from the object to the frame, or its inverse transformation from the frame to the object. But how does this transformation look like? Consider 3.4.

There are two different frames depicted, frame $\{A\}$ and frame $\{B\}$. The origin of frame $\{B\}$ is relative to $\{A\}$ as the vector ${}^AP_{BORG}$. point P is being described relative to frame $\{B\}$, ${}^BP$. In this case the transformation from $\{B\}$ to $\{A\}$, T_{BA} will map point ${}^BP$ onto ${}^AP$. This is a rotation of ${}^BP$ around the origin ${}^AP_{BORG}$ to coincide the coordinate axis of

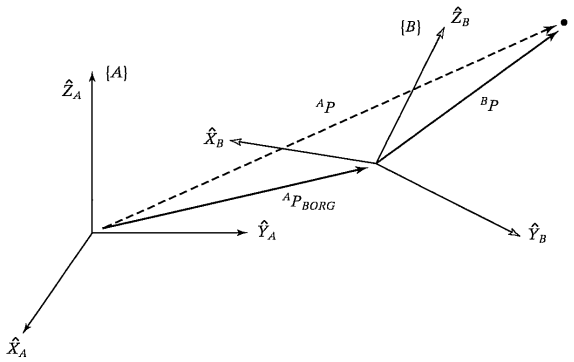


Figure 3.4: [Crai 05, Figure 2.7: General transform of a vector]

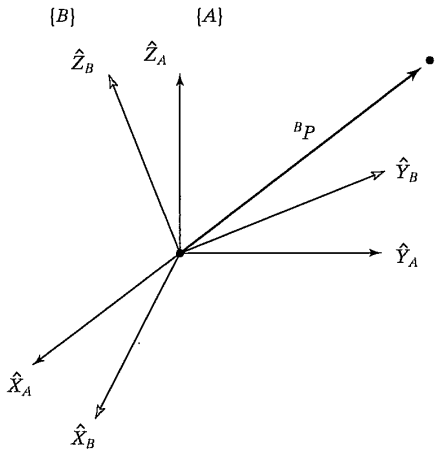


Figure 3.5: [Crai 05, Figure 2.5: Rotating the description of a vector]

$\{A\}$ and $\{B\}$ and then translating by ${}^AP_{BORG}$. [Crai 05, eq. 2.18]

$$^AP = T_{BA} *^B P$$

The translation of a vector is a simple adding onto the vector to apply the translation. But the rotation is a more difficult task, and will be introduced now.

A Rotation is a basis change of one frame to another of the same handedness. This means, that for two coordinate frames $\{A\}$ and $\{B\}$ the rotation from $\{B\}$ to $\{A\}$ changes the description of a point ${}^BP$ described relative to $\{B\}$ to the description ${}^AP$ of the exact same point relative to $\{A\}$. Refer to 3.5. A rotation matrix R_{BA} from frame B to frame

A in 3D is a change of basis of the following form: [Crai 05, eq 2.11]

$$R_{BA} = \begin{pmatrix} {}^{B}X_A & {}^{B}Y_A & {}^{B}Z_A \end{pmatrix} = \begin{pmatrix} & R_{BA} & & {}^{A}P_{BORG_x} \\ & & & {}^{A}P_{BORG_y} \\ & & & {}^{A}P_{BORG_z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

This means, that the cordinate-axes of A relative to B are the columns of the rotation matrix. rotation matrices are orthonormal, thus the transpose of the rotation is its inverse, $R^T = R^{-1}$. All 3D rotations define the set of the special orthogonal group $SO(3)$, since the rotation matrices have determinant equal to 1.
And then $SO(3)$ is a Lie group. The rotations form a group with a smooth function, the matrix matrix multiplication. This means, that it forms a smooth differentiable manifold. In the special case of $SO(3)$ you can imagine the manifold as the surface of a sphere. This will be shown in section 3.3.4. Another special thing about Lie groups is, that there is also a associated Lie algebra. For $SO(3)$ the Lie algebra is the set of all skew-symmetric 3x3-matrices, $so(3)$. There is a surjective mapping from $so(3)$ onto $SO(3)$, called the exponential map. Since any skew-symmetric matrix can be described by a 3-vector v, we will not use the skew-symmetric matrices, but the 3-vector. Thus the skew-symmetric matrices form a vector space isomorphic to $\mathbb{R}^3$. A Lie group has the same dimensions as the associated Lie algebra, thus $SO(3)$ is a 3-dimensional manifold.

The rotation matrix has 9 entries, but a rotation can be described with only 3 parameters. For one there are the euler angles, and on the other side the angle-axis representation, often stored in quaternions.

Fixed Angles and Euler-angles

A common way to define rotations is to use 3 different consecutive rotations about known coordinate axes. This can be interpreted as follows: See 3.6. We can restrict ourselves to only rotate our head about three axes: one othogonal to the tip of our head, the other to pitch our head and the last to rooll our head about the nose. If we ignore corporal restriction, we can with those three rotations after one another - first about the tip, then pitch in the corrent orientation and lastly roll - reach any orientation of our head. This works not only in this order, but there are 12 different ways to define the so-called Euler-angles. Contrary to the Euler-Angles, which are 3 consecutive rotations about changing axes, one can construct 3 rotations about the axes of a fixed frame. There are also 12 different ways to define the 3 rotations. Here only the X-Y-Z fixed angles will

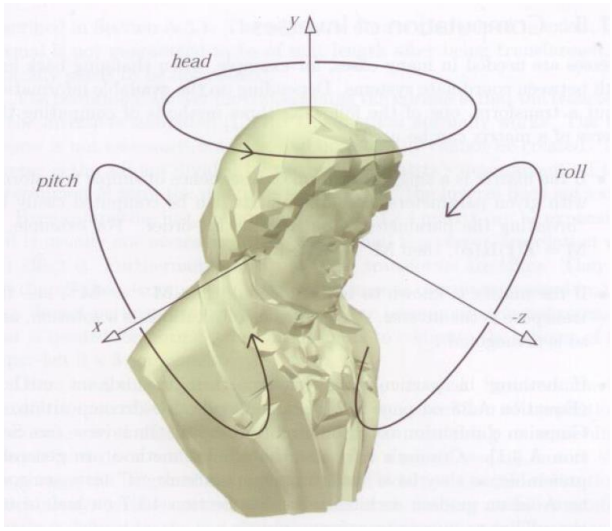


Figure 3.6: [Aken 08, p. 66]

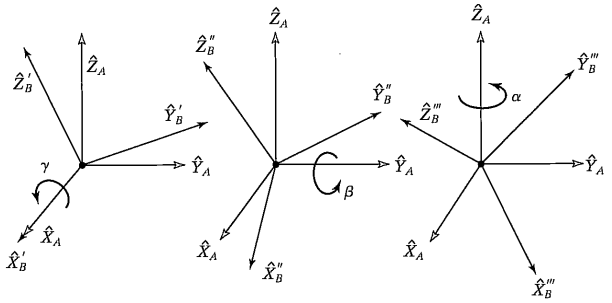


Figure 3.7: [Crai 05, Figure 2.17: X-Y-Z fixed angles]

be described, as in 3.7 [Crai 05, p.41]: " Start with the frame coincident with a known reference frame $\{A\}$. Rotate $\{B\}$ first about $\hat{X}_A$ by an angle γ, then about $\hat{Y}_A$ by an angle β, and, finally, about $\hat{Z}_A$ by an angle α " Such a rotation matrix from frame $\{B\}$ to $\{A\}$ can be described as [Crai 05, eq. 2.63, 2.64]:

$$R_{BA} = {}^{A}_{B}R_{XYZ}(\gamma,\beta,\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\alpha)\cos(\beta) & \cos(\alpha)\sin(\beta)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\sin(\beta)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ -\sin(\beta) & \cos(\beta)\sin(\gamma) & \cos(\beta)\cos(\gamma) \end{pmatrix} \tag{3.7}$$

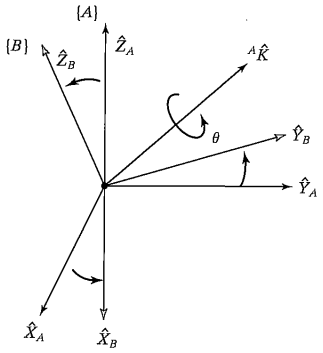


Figure 3.8: [Crai 05, Figure 2.19] angle-axis representation

Angle-axis Representation

Another way to mathematically represent rotations is the angle-axis representation, depicted in 3.8. This describes the rotation as a rotation about a unit vector $\hat{k} = (k_x, k_y, k_z)^T$ by an angle θ. Then the angle-axis representation is $k = \theta\hat{k}$. This means that the angle-axis representation is the axis along the rotation with the angle of rotation as length of the vector. Note that the rotation about the axis by an angle is the same as the rotation about the negative axis by the negative angle. This is also true for the negative angle $2 * \pi - \theta$ since this is the same as the rotating by the negative angle. The relation between the rotation matrix and the angle-axis representation is the Rodrigues formula [Crai 05, eq 2.80]:

$$R_K(\theta) = \begin{pmatrix} k_x k_x v(\theta) + \cos(\theta) & k_x k_y v(\theta) - k_z \sin(\theta) & k_x k_z v(\theta) + k_y \sin(\theta) \\ k_x k_y v(\theta) + k_z \sin(\theta) & k_y k_y v(\theta) + \cos(\theta) & k_y k_z v(\theta) - k_x \sin(\theta) \\ k_x k_z v(\theta) - k_y \sin(\theta) & k_y k_z v(\theta) + k_x \sin(\theta) & k_z k_z v(\theta) + \cos(\theta) \end{pmatrix} \tag{3.8}$$

with $v(\theta) = 1 - \cos(\theta)$, also called the exponential map, $exp(\theta\hat{k})$. The use of the exponential and logarithmic map will be explained in 3.3.4 The inverse, computing the angle-axis vector k out of a rotation matrix can both be computed from [Crai 05, eq. 2.81 - 2.82] and [Hart 13, log(R)]. This is also often referred to as the logarithmic map. In this approach the formula of [Hart 13, 3.2] is being used:

$$log(R) = \begin{cases} \arcsin(\|y\|_2)\frac{y}{\|y\|_2} & y \neq \hat{0} \\ 0 & y = \hat{0} \end{cases} \tag{3.9}$$

$$\frac{1}{2}(R - R^T) = \begin{pmatrix} 0 & -y_3 & y_2 \\ y_3 & 0 & -y_1 \\ -y_2 & y_1 & 0 \end{pmatrix}$$

with $y = (y_1, y_2, y_3)^T$ and $\|\|_2$ as the second euclidean length.

Quaternions

Then there are quaternions. In this section the importance of quaternions for rotations will be explained, not what quaternions are. Quaternions are 4 dimensional vectors. Or simply put 4 different numbers. With 4 numbers it is possible to describe the euler parameters.In the previous section angle-axis representation has been introduced, $\theta\hat{k}$. The euler parameters are a combination of the angle and the vector of the following form: [Crai 05, eq 2.89+ eq 2.90]

$$\epsilon_1 = k_x \sin(\frac{\theta}{2})$$

$$\epsilon_2 = k_y \sin(\frac{\theta}{2})$$

$$\epsilon_3 = k_z \sin(\frac{\theta}{2})$$

$$\epsilon_4 = \cos(\frac{\theta}{2})$$

and one important result: $\epsilon_1^2+\epsilon_2^2+\epsilon_3^2+\epsilon_4^2 = 1$, which is always true. The 4 euler parameters can be used as entries in the quaternion. A quaternion built from the euler parameters looks like this [Hart 13, eq. 4]:

$$q = (\epsilon_4, \epsilon_1, \epsilon_2, \epsilon_3) \tag{3.10}$$

This means, that the unit quaternions, quaternions with length equal to 1, describe all possible rotations. But since the angle-axis representation itself is not unique, this also holds for quaternions. For rotation R represented as quaternion r, both r and $-r$ represent the same rotation.

Another important part of quaternions is its multiplication. The quaternion quaternion multiplication is defined as follows [Hart 13, 3.3], if the last three entries of the quaternion are put together into a vector $\vec{v} = (k_x \sin(\frac{\theta}{2}), k_y \sin(\frac{\theta}{2}), k_z \sin(\frac{\theta}{2}))^T$, so that quaternion q is $(\cos(\frac{\theta}{2}), \vec{v})$.

With the quaternions $q_1 = (c_1, \vec{v_1})$ and $q_2 = (c_2, \vec{v_2})$ the multiplication is:

$$q_1 * q_2 = (c_1 * c_2 - \langle\vec{v_1}, \vec{v_2}\rangle, c_1 * \vec{v_1} + c_2 * \vec{v_2} + \vec{v_1} \times \vec{v_2}) \tag{3.11}$$

$\langle \vec{a}, \vec{b} \rangle$ defines the vector product, and $\vec{a} \times \vec{b}$ is the cross product.
For unit quaternions there is a important property: the multiplication of two unit quaternions results in a unit quaternion. If we consider two quaternions r s and their corresponding rotation matrices $r \Rightarrow R$ and $s \Rightarrow S$. Then the multiplication of the quaternion result in a matrix matrix multiplication of the rotations, or performing both rotations after another. $r * s \Rightarrow R * S$

Gnomonic projection and the Riemannian Manifold SO(3)

Considering $SO(3)$ as the unit quaternions, the unit quaternions form a sphere in $\mathbb{R}^4$ with radius one. To find a average it is important to find the shortest path from one rotation to another, and since they are points on the sphere, it is the shortest path along the surface of the sphere. Since the surface of a sphere is not flat, the term geodesic is necessary to describe the shortest path on the Riemannian manifold the sphere. A geodesic is a locally shorrtest path consisting of locally shortest paths, where any intermediate step is on the manifold. This can be imagined as a rubberband, where the end and start are fixed. This rubberband will try to find the locally shortest way from start to end. For spheres there always consist two geodesics. Both of them lie on the same great circle of the sphere. Great circles are circles on the sphere, where the midpoint is the center of the sphere. So a geodesic between two points is along a great circle, where both point are on the circle, either ockwiese or anti-clockwise.
Finding the geodesics on a sphere is no new task, they are being used for example as routes for airplanes. There the gnomonic projection is useful. Gnomonic projection as in 3.9 is a projection, where a sphere is being projected on a hyper-plane, where the plane tangents the sphere at one point, for quaternions at $(-1, 0, 0, 0)$. Then for every point on the sphere a straight line at the point through the center point $(0, 0, 0, 0)$ will be formed. The intersection of this line and the tangent space is then the point on the tangent space where the point will be projected to. It is clear, that points lying opposite have the same projection. This is good, since a quaternion and its opposite both represent the same rotation. Another important property of the gnomonic projection is, that great circles on the sphere are straight lines at the tangent space. The tangent space for rotations is the angle-axis representation with the logarithmic map as already described. To recenter the logarithmic map $log(R)$ to a rotation S as $log_S(R)$, is the same as $log(S^T R)$. For more information see [Hart 13] Hartley-Trumpf-Gnomon-Projection

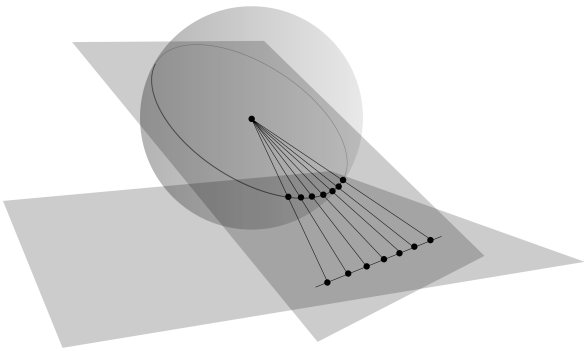


Figure 3.9: [Hart 13, Figure 1: Gnomonic projection of a sphere]

LP-Mean of Rotations

Often it is necessary to find the mean of a rotation, in our case to minimize the error. But unfortunately the set of rotation are no euclidean space, which would have had a easy way to compute the mean. Therefore we have to consider different ways to define the distance of two rotations. For any distance-metric $d(R_i, R)$ minimizing the cost function $C(R)$ would directly lead to the mean rotationof the given rotations R_i, where $C(R)$ is:

$$C(R) = \sum_{i=1}^{n} d(R_i, R)^p \tag{3.12}$$

One condition for the mean to be unique and to converge to the mean is for the cost function to be convex. As [Hart 13] mentions, the convexity is "tightly coupled with the notion of convex sets". And [Hart 13, Definition 1]:
"A non-empty region $U \subset SO(3)$ is called weakly convex if for any two points R_0 and R_1 in U exactly one geodesic segment from R_0 to R_1 lies entirely inside U. A weakly convex region $U \subset SO(3)$ is called convex if the geodesic segment from R_0 to R_1 in U is always the short geodesic segment between these points, having length strictly smaller than π. The empty set is not considered to be convex or weakly convex." Furthermore consider [Hart 13, Corrollary 1], which states that the cost function is strictly convex on the set $B(S, \frac{\pi}{2}) = \{S \in SO(3) | d_{\angle}(S, R) \leq \frac{\pi}{2}\}$ and has a "single isolated minimum" on that set. For the distance different metrics can be used to be minimized, but only chordal metric and geodesic metric will be described in the following sections, because they are being used.

Chordal Metric

One metric using the euclidean metric of distances is the chordal metric. The chordal metric is the Frobenius-norm of the difference of the rotation matrices. $d_{chord}(S,R) = \|S-R\|_F$ where $\|\|_F$ is the Frobenius norm. Refer to [Hart 13, Chordal distance] for more information. For the L2-mean exists a closed-form solution. If $\hat{S} = \sum_i^n R_i$ is the sum over all n rotations R_i, then the closest rotation to $\hat{S}$ can be computed via the singular value decomposition (SVD), as proposed in [Moak 02]. The SVD solves $\hat{S} = UDV^T$, where D is a diagonal matrix with descending diagonal elements. Then the closest rotation to $\hat{S}$ is UV^T if $det(UV^T) \geq 0$, otherwise $Udiag(1,1,-1)V^T$.

Geodesic Metric

The maybe most intuitive way to describe the distance of two rotations is to use the angle between both rotations. For two rotations S and R the distance would be the angle in the angle-axis representation of the rotation SR^T, where the angle is to be chosen to lie in $[0;\pi]$ to be locally unique. Thus the geodesic metric is also called the angular metric and is defined as the length of the angle-axis vector:

$$d_{\angle}(S,R) = \|log(SR^T)\|_2 \tag{3.13}$$

OpenGLES2.0

Frustrum

One question in rendering is, where on the screen will the object be rendered? And when is it no more visible? OpenGL (https://www.khronos.org/opengles/) uses a unit-cube to define what is outisde and what is inside the view. This means that all vertices with the all three entries in between -1 and 1 will be rendered to the screen. But since there homogeneous coordinates are possible to describe vertices, those vertices will be scaled by their 4th entry, and only then tested if they lie inside the unit cube. In this thesis this is not important since we did not use this entry. And even if all vertices lie outside of the unit cube, some parts of the object might still be rendered. For example if there is a triangle with vertices $\{(2,2,0,1)^T,;(-2,-2,0,1)^T;(2,0,0,1)^T\}$, then all three vertices are clearly outside the cube, but this triangle still has parts inside the cube, and those parts will be rendered.

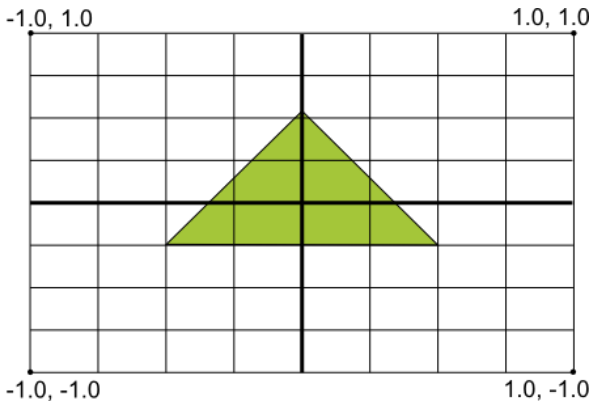


Figure 3.10: Transformations known in the projection setup

The visible vertices are still 3 dimensional vertices, but the third entry will not be used to determine the resulting pixel on the screen. The screen itself is also normalized, so that any visible vector $(-1, -1, z, 1)^T$ will be shown at the bottom left corner of the screen, and a vector $(1, 1, z, 1)^T$ on the top right corner of the screen. And anything in between will be rendered accordingly in between. See 3.10. Be aware that "top left" on a screen in Android is not always the same (https://developer.android.com/). Android uses different orientations for activities, and in some cases will change orientation during use if the user rotates the device. For our implementation only the orientation called "landscape" will be used.

Textures

there are certain conditions on textures that have to be considered. For one, for rendering the Camera2 input a OpenGLOES_ext_texture_2D is required, this is a ratified extension of the GLES specification and is available for all Android devices since API-level 21 (https://developer.android.com/). And secondly, the colour values of pixels in the texture have to be integers in [0;255] for the red, green, blue and alpha values. This is due to the reason, that in the embedded version of OpenGL floating point colours in textures are not supported in the specification [Aaft 09]. Those integer values then are being normalized in the shader programs.

Chapter 4

Methods and Implementation

In this chapter first the details of the setup are explained, then both methods to compute the hand-eye calibration are presented. Afterwards the implementation of the live guidance is being discussed. In the Rendering section the correction of the distortions in the camera is tackled, and finally the implementation of the pose-aware projection of the ultrasound images explained.

Setup

Before the different methods and their implementation is discussed, an overview and the setup of the application is given. For the application one Android smartphone with at least API Level 21 is necessary, because the application uses the android.hardware. camera2 package for the better performance of the camera capture. For the camera calibration a circles grid pattern is used, in this application a 4 by 11 pattern with spacing 20mm, downloadable at https://nerian.com/support/resources/patterns/. Next is a ultrasound probe which is calibrated to a given marker for the optical tracking system (OTS), for example via [Lass 14a] using N-wire calibration. A second marker is fixed onto the android smartphone to track both the android device and the ultrasound device by the OTS. Furthermore we need one computer, which will receive the input of OTS and probe and send the poses and images through TCP sockets using the OpenIGTLink [Toku 09] protocol.

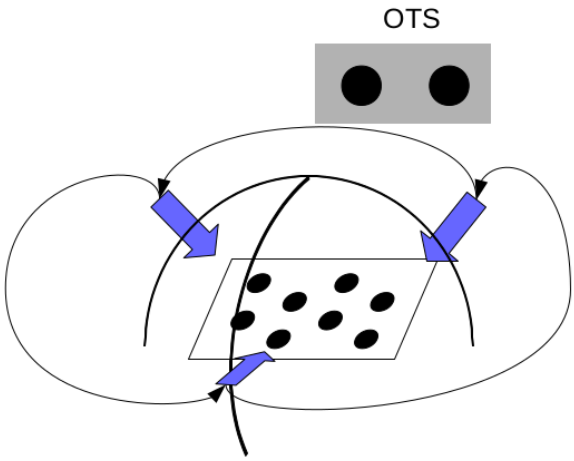


Figure 4.1: Pairs of stations of the Android device suggested for the capturing, with almost no rotation about the z-axis of the camera. This can be repeated until enough stations are taken

Workflow

Now the workflow of the application is explained. At first a modified OpenCV example application is processed to calibrate the camera to the tracker. This is a standard Hand-Eye calibration problem, which will be solved either with [Tsai 89] or with a simple averaging of the transformations, depending whether the pose of the pattern relative to the optical tracking System is known or not. This means that first a number of pictures are captured, where the circles grid pattern can be found by the OpenCV pattern. In the following such a capture with the tracked pose of the marker mounted on the android device will be denoted as "station". It is suggested to use pairs of stations for the capturing of the images similar to [Tsai 89, Fig.7], but since the android smartphone has to be trackable at any station, you may have to take the three stations depicted in figure 4.1 that allows the tracker to find the marker, and repeating them until enough stations are captured. Be aware, that figure 4.1 is not a good option if you want to use tangential distortion, since it will most likely lead to very large tangential distortions as most captures will see the pattern with a tilt to the same side. Using radial distortions alone will already lead to good results. For this reason only radial distortion has been corrected, but more distortion parameters can be used easily.
Once this is done, the application will compute the camera calibration and the extrinsic parameters of the camera at each capture. With those poses and the simultaneously received poses of the tracker mounted on the camera, the transformation between camera view and the coordinate system of the OTS will be computed. Afterwards the distortion of the camera will be corrected via computing the inverse distortion and computing the resulting mapping from each distorted pixel to its undistorted pixel in a lookup-table

to undistort the camera view. Only if the images captured from the camera are being undistorted the perspective blending of the ultrasound will be mapped correctly into the same view.
Then the Application is ready for its designed function, and will blend the received Ultrasound-data in the view of the camera. The live-guidance can be started via a double tap, or two fast clicks of the GoogleVR trigger, to set or reposition the goal pose of the guidance towards the current pose of the ultrasound device. A single tap will toggle the visibility of the guidance. The following sections will explain how the various steps and problems will be solved, starting with the two methods for computing hand-eye calibration. Afterwards we discuss the live guidance, which will guide the user to a given goal pose. Then the rendering is explained, starting with inverse distortion up to blending the image into the AR application.

One important first step in rendering the ultrasound correctly into the view of the camera is to know where the image lies relative to the camera. But since the optical tracking system only tracks the marker on the android device, it tracks effectively the transformation from marker to tracker and not the transformation from camera to tracker. That means that the relation between both frames is unknown. Finding the transformation from the camera - the "eye"- to the mounted marker - the "hand" -, is a standard hand-eye-calibration problem. The reason will be explained hereafter. In a hand-eye-calibration the transformation betweeen the hand, or often called the end-effector of a robot, is holding a camera in it's end-effector. The position of the end-effector relative to its base can be computed in a robot via inverse kinematics, thus the transformation from hand to base is known. The camera can find the transformation from a observable pattern to the cameras center. If both transformations are known at the same time in at least two different stations, one can compute the fixed transformation from hand to eye. But how does this relate to the problem of finding the transformation between the marker and the camera? The marker can be seen as a end-effector of a robot and thus the marker represents the hand. That way the hand-eye-calibration can be applied here. In this chapter two different methods for hand-eye calibration will be explained, even though the problem solved by both approaches are completely different.

Hand-Eye Calibration Problem

Consider the following transformations, depicted in [Tsai 89, Fig. 5], shown in figure 4.2 occuring at each station: There are two different base coordinate system, which are fixed during the calibration. One is the coordinate frame of the optical tracking system,

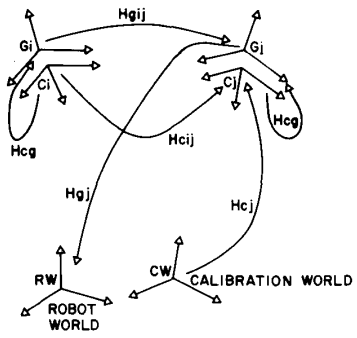


Figure 4.2: [Tsai 89, Fig. 5]Relationship between the homogeneous matrices and the coordinate frames

the other is the fixed frame of the calibration pattern. At every station i the following transformation are known: The transformation from the pattern to the camera H_{c_i} as extrinsic parameter after the camera calibration, and the transformation from the marker to the tracking system, H_{g_i}. The transformation from the "eye" -camera - to the "hand" - the marker or the gripper - then is H_{cg}. [Tsai 89] mentioned, that the result can be easily computed if the fixed frames of pattern and tracking system coincide. The result would then be $H_{cg} = H_{g_i}^{-1} \cdot H_{c_i}^{-1}$. The relation between pattern and tracking system to find and fix is no trivial matter, but if the transformation from pattern to tracker H_{pt} is known, then it can be computed by

$$H_{cg} = H_{g_i}^{-1} \cdot H_{pt} \cdot H_{c_i}^{-1} \tag{4.1}$$

This might already give a sufficient result, but since H_{c_i} depends on fitting and finding the circles grid pattern in the image is, there might be an error in the transformation. Since already several stations were taken during the calibration step, we can compute for each of them equation 4.1. All of those transformations should ultimately lead to the same transformation, since the marker is mounted on the camera and thus the transformation is fixed. Hower, all of them may have a slight error, but if we where to compute the mean transformation, we would minimize this error. How this average transformation can be achieved will be proposed in section "Hand-Eye Calibration via Transformation Averaging", but let us first consider the computation of H_{cg} if the relation between pattern and tracking system is not known.

Hand-Eye Calibration via TsaiLenz

If the pose of the pattern is not known relative to the tracking frame, it is rather hard to compute the hand-eye transformation. [Tsai 89] proposed an algorithm to find the hand-eye transformation H_{cg} given at least three stations, where stations means taking a tracked pose of the marker and computing the extrinsic parameters of the camera at the same time. This section will explain the algorithm for our implementation. The algorith mitself has not been modified, the only difference is, that we have a marker mounted on a camera instead of a grabber. For more information see [Tsai 89]. The rotation between those three needs to be about different rotation axes, otherwise H_{cg} can not be computed. It will lead to more accurate results, if more stations are available, as explained in [Tsai 89, Section III-A]
To understand how [Tsai 89] works, following aspects have to be considered:
1. Given a rotation matrix R_x [Tsai 89] does not use the angle-axis representation, but a different representation. They use the rotation axis, but instead of using the rotation angle θ as the length, they use $2\sin(\theta/2)$. Thus the vector P_x is the modified angle-representation of the rotation matrix R_x.
2. Be aware of the direction of the transformations. OpenCV will result in transformations from pattern to camera, but the optical tracking system will not necessarily give the transformation from marker to tracking system center, but rather the inverse transformation. This was the case in our testing base, and is directly handled by our implementation.
This approach uses the movement of both rigidly connected frames, the gripper and the camera to compute the hand-eye calibration H_{cg}. This means that it will compute the transformations between stations as follows:

$$H_{gij} = H_{gj}^{-1} H_{gi}$$

$$H_{cij} = H_{cj} H_{ci}^{-1}$$

Where H_{gij} is the transformation between the two stations i and j of the gripper pose. Similarly H_{cij} for the camera poses. For both transformations we get a rotation with their modified angle axis representations P_{gij}, P_{cij} Then [Tsai 89] computes H_{cg} with this procedure: 1. Solve the linear least squares problem of the given equations to find P'_{cg}:

$$Skew(P_{gij} + P_{cij}) P'_{cg} = P_{cij} - P_{gij}$$

Where $Skew(v)$ constructs a skew-symmetric matrix with the given 3-vector as entries. The proof for this linear least squares problem follows from [Tsai 89, Lemma VI] There may occur a special exception [Tsai 89]: " If $P_{gi_1j_1} + P_{ci_1j_1}$ is colinear with $P_{gi_2j_2} + P_{ci_2j_2}$ while $P_{gi_1j_1}$ is not colinear with $P_{gi_2j_2}$, then the rotation angle of R_{cg} must be 180° and the rotation axis the same as $P_{gi_1j_1}+P_{ci_1j_1}$" 2. P'_{cg} is not the modified angle axis representation, but defines as [Tsai 89, equation 11.4], so to compute the modified angle-axis P_{cg} from ${P'}_{cg}$ do:

$$P_{cg} = \frac{2{P'}_{cg}}{\sqrt{1 + \|{P'}_{cg}\|_2^{\,2}}}$$

3. Find the translation vector T_{cg} of H_{cg} via linear least squares of the set of equations

$$(R_{gij} - I)T_{cg} = R_{cg}T_{cij} - T_{gij}$$

where I denotes the identity matrix. For the proof of this equation refer to [Tsai 89, Lemma VII]. With this, H_{cg} can be formed from the rotation R_{cg} and the translation T_{cg} For more information on why this works, see [Tsai 89]. They also provided a error analysis, and a with it some suggestions to improve accuracy, summarised here briefly:

Suggestions

1. Use large rotations between stations.
2. The axes of those rotations should have a large angle in between. That is the main reason why you should follow figure 4.1 to capture the stations, since then the rotation angle and the axes between two stations are large. If you were to use a tripod, try to mount the tracker in such a way so that [Tsai 89, Fig.6] can be used to further improve the capturing.
3. Minimize distance between the camera and the pattern, for example by using a small pattern.
4. Keep the movement of the gripper small. This might sound hard to achieve, since the marker is mounted rigidly on the android device, and thus moves along. But if you were to mount the android device on a tripod, you have a lot of places to choose a mount for the marker. Then this becomes more important. Try to put the marker in front of the camera, but as far away as possible. That way the marker will be closer to the pattern during capturing the stations, and thus will not move as much.
5. Use a very accurate tracking system. This property will have the strongest influence on the accuracy.

This algorithm has obviously been implemented earlier and is provided in libraries, for

example the C++ library ViSP (see https://visp.inria.fr/). Unfortunately ViSP is not supported on Android yet. As one might read from [Fabi 12], ViSP uses right now the normal equation to solve the linear least squares problems. We use a different Java library, [Joe 12], which uses QR-decomposition as solver, which is better for the condition of the left matrix in the equation.

Hand-Eye Calibration via Transformation Averaging

As has been shown, if the pose of the calibration pattern relative to the optical tracking system is known - for example if the black circles in the circles grid pattern can be found by the tracker - then the hand eye calibration problem does not exist, and the transformation from hand to eye can be computed from only one capture. This result may already be good enough, but there may be some error in the transformation of a single station. For one point, the camera is not always facing the pattern up front, but may see the pattern from one angle where the pattern is strongly deformed in the capture. However there might be some error due to time difference. The tracker itself sends the messages at a different frequency as the images are taken by the camera.
Now imagine the camera moving. The tracker tracks the device and sends the message. The camera acquires an image at a different time and a different pose due to the movement. The movement will result in a error in the hand eye transformation. An error occurence in the tracking would be beyond the scope of this thesis, thus we consider the tracking to be exact. If the tracking is lost, the tracker does not send messages, and then the application will not take captures, up to a small delay, which is negligible and can be thought of as the error due to asynchronous capturing. So, there are enough reasons to consider the presence of noise, and one common way to minimize noise is to average several measurements.
We have several stations, and all of them should theoretically lead to the exact same result, the correct hand eye calibration. For every station i, we have its own hand-eye calibration H_{hei}. Since the marker is fixed rigidly on the Android device, all of them should be the same, except for the noise. Hereafter, a way to average transformation matrices will be proposed.
We only consider transformations without the presence of scaling factors, so a transformation H_{hei} consists of two parts: A translation vector ${}^{e}P_{hORGi}$ and a rotation matrix R_i. First the average rotation will be found, and with this average rotation the translation will be averaged. For rotations, the average is not easy, since it is not an Euclidean space, but a manifold. As discussed, computing the L2-mean can be done as closed-form solution, but it is not as robust as the L1-mean, also called the median. That

is due to the fact, that the L1-mean is more robust to outliers. This is important since outliers in this setup are most likely outliers because there were greater errors in the capturing. Those should not be weighted stronger than other captures with less error.
The L1-mean on the geodesic metric has been proposed in [I Ha 11]. There they proofed in [I Ha 11, Lemma 5.2] a connection between the geodesic median S of the given rotations R_i and the geometric median of points $log_s(R_i)$. The geometric median in this case has to be the $(0, 0, 0)^T$. And this works also the other way round, if the angles between the rotation and the geometric median is less than $\pi/2$. With this property they proposed an algorithm based on the Weiszfeld algorithm on the tangent space to find the geodesic median in $SO(3)$. This approach only converges if all R_i lie in a convex set. They stated in [I Ha 11] that this is the case when the angle between those rotation is smaller then $\pi/2$. A proof of this can be found at [Hart 13, Appendix - Convexity]. In our case, the angular distance is only due to the error. Since we consider the tracking to be perfect, a large error can only happen, if the extrinsic parameters are bad. In this case, they would have to be wrong by at least 45 degree in one direction, and another in -45 degree in the inverse direction. But such a large error would mean, that some points in the circles grid pattern where not found. Fast rotating the android device is not able to lead to such big errors. The camera is not able to capture anything of value if the camera rotates too fast, and 45 degree in this small time to create such a big error is way too fast to capture anything of value with the camera. So, it is reasonable to consider this approach to work for our setup.

L1 Rotation Averaging using Weiszfeld

The Weiszfeld algorithm is a gradient descent algorithm. One common problem in gradient descent algorithms is to find the step size. The Weiszfeld algorithm follows the gradient direction by a step size computed via a closed form solution. This algorithm has the downside, that it will get stuck if the estimation S^t at some time step t coincides with one of the estimates R_i. To tackle this problem, the algorithm is initiated with the L2-mean. If the L2-mean coincides with one estimate, it gets slightly displaced, since the Weiszfeld algorithm will "escape" from estimate points, as stated in [I Ha 11].This way it will not get stuck at some point. With this initialisation the following steps will be repeated until convergence. In our implementation we repeat it 1000 times, which still would not use a lot of time compared to the time nexessary to acquire the stations.
[I Ha 11, 5.]:
compute the angle-axis representation centered at the intermediate result S^t at time step

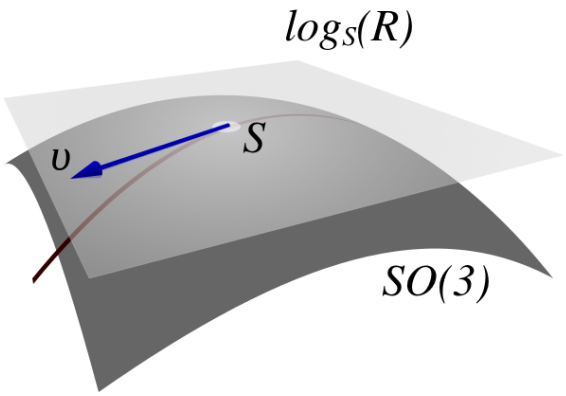


Figure 4.3: [mcSu 08]Tangential space $log_S(R)$ centered at intermediate step S, with the current step δasv.

t

$$v_i = log_{S^t}(R_i)$$

Compute the Weiszfeld gradient with the step-size λ:

$$\lambda = \sum_{i=1}^{n} 1/\|v_i\|_2$$

$$\delta = \frac{\sum_{i=1}^{n} v_i/\|v_i\|_2}{\lambda}$$

Apply the step on the intermediate result:

$$S^{t+1} = exp(\delta)S^t$$

In summary, we compute a gradient in the tangent space, and go along this direction with the step size and recenter the tangent space on the new rotation, as depicted in figure 3.4.

Those steps can and will also be done in the implementation from the quaternion presentation instead of in $SO(3)$, since the multiplication in quaternion space is the same as the matrix multiplication. So, if we exchange the steps to not use the logarithmic and exponential map, but map the angle-axis representation as shown earlier to quaternions, we can perform this algorithm in the space of quaternions. This is more efficient since the multiplication of quaternions is faster, this is also true for the mappings between quaternions and angle-axis. With this, we have the average rotation S computed for the transformations.
For the translational part we have to consider one problem: Consider a transformation from frame h to frame e. Then the translation is ${}^eP_{hORG}$: the vector from the origin of

Figure 4.4: Example picture of the live guidance. The goal pose is shown via its coordinate axes in red, green and blue. The current pose of the ultrasound with its coordinate axes in magenta, cyan and yellow. The textual guidance is shown on the top left.

frame e to the origin of frame h, described relative to coordinate frame e, as explained earlier. Since in our case the transformation and with it the frame has been estimated, this also means, that the translation vector is being described relative to the estimated frame.
So, if we want to compare the translation vectors, we first have to describe them in the same coordinate frame. The translation relative to the average rotation S is ${}^{S}P_{hORGi}$.

$$ {}^{S}P_{hORGi} = S \cdot R_i^{-1} \cdot {}^{e}P_{hORGi} \tag{4.2} $$

This way we undo the erroneous rotation and exchange it with the average rotation, since the translation vector has to be described relative to the coordinate axes of the corrected frame.
Averaging the resulting translations is simple, since the translation vector is in the Euclidean space, the average of translation vertices ${}^{S}P_{hORGi}$ is the arithmetic mean:

$$ t = \sum_i \frac{{}^{S}P_{hORGi}}{\|{}^{S}P_{hORGi}\|_2} \tag{4.3} $$

This way both rotation and then translation can be averaged, and the resulting rotation and translation can be combined into a new transformation as described in equation 3.6

Live Guidance

The next task of this application is to provide some way to guide the user to a predefined goal. This goal is a stored transformation from the ultrasound to the tracker H_g. This way the user can store a goal position and still move the android and ultrasound independently

without corrupting the goal position. But if the user wants to move the optical tracking system, he should then take the goal pose again, since the old goal pose will be misplaced. If there is a goal transformation from ultrasound to tracker stored as H_g and a current transformation from ultrasound to tracker H_u is received, then the transformation from ultrasound to goal H_{ug} can be computed:

$$H_{ug} = H_g^{-1} \cdot H_u \tag{4.4}$$

This is then the misalignment of the current ultrasound pose to the goal pose. As described in [Crai 05, Eq. 2.19] the transformation H_{ug} consists of a rotation R_{ug} and a vector ${}^{g}P_u$ describing the center of the coordinate frame of the current ultrasound relative to the center of the coordinate frame of the goal. To guide the user from the current pose to the desired goal pose, the user has to perform the transformation H_{ug}.

Rendering Guidance

There are two different use-cases to be considered for the guidance:
In one case the ultrasound and the goal pose are visible in the view of the camera, and the user wants to move the ultrasound to coincide with the goal pose. In the other case the user has a goal pose stored, which is not visible in the view of the camera. Then it should still be possible to guide the user to the goal pose. There is a large difference between both cases, since in the first case, we can simply render the goal at its pose, and in the other case we have to give a textual guidance to the goal pose. If we consider the goal pose visible in the field of view of the camera, we can do it as shown in 4.4. It is a rather simple but effective approach, where the coordinate axes of the goal pose are being rendered at its pose. And the coordinate axes of the current pose are being rendered at the current pose. The different axes are all coloured differently. The coordinate axes are coloured in red for x-axis, green for y-axis and blue for z-axis. The axes of the current pose are being rendered in magenta, cyan and yellow for x-,y- and z-axis. This way the user can see both poses, and move the current pose to the goal pose, if the user lets the coordinate axes coincide.

Textual Guidance

The other use-case is showing a textual guidance since the goal pose would not be rendered into the view of the camera. It consists a translation between the two poses, and a rotation. To correct the translational component of H_{ug} the user has to move the current center

towards the center of the goal pose. ${}^{g}P_u$ is the vector from the center of the goal to the current position, thus the user has to move the current position by $-1 \cdot {}^{g}P_u$ to coincide both positions.
The same vector movement can be expressed from different points of view respectively coordinate frames. In this setup there are several coordinate frames: The optical tracking system, the camera of the Android device with its center in the camera, both marker on the devices and the coordinate frame of the image of the ultrasound.
We will use the frame of the image to describe the translation vector, since frame is known to the user. The axes of the ultrasound image are being rendered. The user sees them next to the ultrasound image in the AR environment. This means that the positional correction will be shown by a vector relative to the images coordinate frame in millimeter. This is exactly $-1 * {}^{g}P_u$.
Along with the positional correction comes another desired movement: bringing the current pose to the right orientation. We will compute the x-y-z fixed angles from the rotation matrix as described in equation 3.7, in degrees and show them to the user. This way the user can perform the correction of the orientation by rotating first around the x- then the y- and at last around the z-axis. To perform the rotations in the correct order might be the biggest problem, since rotating about the wrong axis first will not necessarily make the angle between the current and the goal orientation any smaller. And this textual guidance will fail, or flicker between values, if the rotation about the x-axis goes near to ± 90 degrees, since in this case the matrix degenerates and the rotation about z will be considered to be about 0 degree in this case. But in this case, the user will not need to know the correct rotation about y- and z-axis, he should first rotate about the x-axis. This way, the degenerated case is not an issue for the guidance.

Rendering

In this chapter first the correction of the distortion in the camera is being tackled and thereafter the pose-aware rendering of the ultrasound images explained.

Distortion Correction

As has been explained above, the camera calibration model of OpenCV consists of equations 3.4 and 3.5. But this equation computes the undistorted pixel position (x_u, y_u) of a distorted pixel (x, y). But during the rendering, (x_u, y_u) of each pixel is known and the distorted pixel has to be found, to render it at the correct undistorted pixel. Thus

the inverse of equations 3.4 and 3.5 have to be found. Due to the complexity of the function there is no analytic inverse [Benl 15]. OpenCV provides a function to directly undistort one image, and we can use this. But first, what does the function provided by openCV? This can be excerpted from the OpenCV documentation [Open] It first calls one function called initUndistortRectifyMap() to get a mapping from distorted to undistorted pixel, and then applies those mappings with bilinear interpolation onto the received image vie remap(). The initUndistortRectifyMap compute the following according to OpenCV documentation [Open] of version 2.4:

$$x = (u - c_x)/f_x$$

$$y = (v - c_x)/f_y$$

$$[X, Y, W]^T = R^{-1} * [x, y, 1]^T$$

$$x' = X/W$$

$$y' = Y/W$$

$$x'' = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$

$$y'' = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$

$$map_x(u, v) = x'' f_x + c_x$$

$$map_y(u, v) = y'' f_y + c_y$$

For every pixel (u, v), where $[k_1, k_2, p_1, p_2, k_3]$ are the distortion parameters, (c_x, c_y) is the principle point with focal lengths f_x, f_y and camera matrix R. Up till now homogeneous coordinates where not introduced or used, so W is always 1.
In other words, they first compute for each undistorted pixel the corresponding point at $\{z = 1\}$ plane (x', y'). Then they compute onto which pixel this point would be projected with this given distortion, and store the resulting u- and v-coordinate position in two separate maps. This is what initUndistortRectifyMap does in our case. If there are more distortion parameters, the same process would be done, but obviously with a different distortion model to compute (x'', y''). Both maps will then be used to fill the final image with the pixels in the distorted image with bilinear interpolation and zeroes at the boundaries.
This way one pixel can be undistorted. But we can not simply create a full image at every frame. This would be too slow for real-time. But we can use a lookuptable and fill it into the rendering as a texture. OpenGLES (https://www.khronos.org/opengles/) works with UV-maps to get the pixel out of a texture. UV-maps are not very hard to

understand. Instead of using directly the pixel position in a texture with a given width and height, the pixel coordinates are normalized, which means that $(0, 0)$ is the top left pixel, $(1, 0)$ is the top right pixel, and $(1, 1)$ is the pixel at the bottom right of the texture. This way the resolution of the texture itself is not important durign rendering, only the relative mapping is important. A normal UV-map has a smooth transition between the corner values, but we can define a simple texture with the same resolution as the pictures received from the camera, and store at every pixel not a colour, but a uv-position. Then in the rendering to instead render the distorted pixel, we may look in the other texture, and use the UV-position there to get the colour of the pixel at the different position.
The question is now how to use the undistortion of OpenCV to create a lookup-table. If we create first an image containing the simple UV-map coordinates, a texture where every pixel at UV-coordinate u, v would contain u, v as entry. Then we can use the provided undistortion to warp this texture. After applying the undistortion, every pixel (px, py) in the texture would not contain the UV-coordinate to itself, but the UV-coordinate of the distorted pixel. With this texture we can directly find for every undistorted pixel its corresponding distorted pixel, or the nearest pixel if there is no direct correspondence.
There is only one disadvantage of this procedure: Pixels for which the distorted pixel position would be outside of the image will be set to zero. This itself is not an issue, and can be used to not use those pixels. But at the bottom right boundary this will lead to artifacts. The reason for those artifacts is the bilinear interpolation. At one pixel we get UV-coordinates where at least one entry is close to 1. So if this pixel gets interpolation with a neighbouring zero pixel, the pixel in between would get for the entry close to 1 something close to 0.5. This way we receive a pixel-wide artifact band at the bottom and right side of the screen, where the colour is incorrect.
This texture then has to be brought into OpenGLES, which can not be done directly, since the UV-coordinates are vectors consisting of floating points. It has already been shown, that a floating point texture is not necessarily supported on every android device. For this reason we have to come up with a way to fill the texture into OpenGLES while keeping as much accuracy as possible. But we can store a 4-vector containing integer values in $[0; 255]$ to fill the red, green, blue and alpha part of a pixel. This way, we can use for one UV-coordinate (u, v) the following scheme:

$$rgba(u{*}256{*}255, v{*}256{*}255) = \begin{cases} (255, 255, 255, 255) & \text{iff u or v not in (0;1)} \\ (u/256, u \mod 256, v/256, v \mod 256) & \text{otherwise} \end{cases} \tag{4.5}$$

Where $(255, 255, 255, 255)$ means that the pixel should not be rendered, and every other texture with up to 255*256 pixel in width or height this approach is at least pixel-perfect

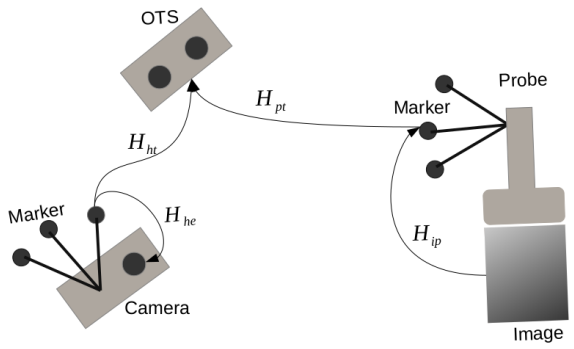


Figure 4.5: Transformations known in the projection setup

and even more accurate then necessary since we use nearest neighbour to find the pixel in the final camera image.

Projecting Ultrasound Images

After the correct rendering of the camera input the next question is: Where will the ultrasound be rendered in the screen? First of all, the image of the ultrasound is a simple rectangle, thus we can render it using two triangles representing the rectangle. This means, that we can draw both triangles with 4 vertices, and the rendering pipeline will fill everything accordingly in between the triangle with the given image data. Because of this, we only have to find the position of those 4 vertices in the frustrum. Since the perspective projection only works if the vertices are being described relative to the frame of the camera, first the transformation into the view of the camera has to be discussed, as depicted in figure 4.5. In the rectangle, the four corner points are only known as pixel position in the ultrasound image. If w is width and h height of the image we have the pixel positions $\{(0,0);(0,h-1);(w-1,0);(w-1,h-1)\}$. Since we do not necessarily know the resolution of the image beforehand, we might think of not using those pixel vertices, but rather normalized with $(0,0);(1,0);(0,1);(1,1)$ and computes a scaling matrix S_{px} to scale the vertices accordingly to fit the previous pixel positions once we received the first image. the next step is to find the 3D position of those pixels relative to the marker mounted on the probe. Thankfully this is the fixed precalibrated transformation combined with a scaling to scale from pixel size to millimeter matrix from image to probe H_{ip}. This way we now know where the corners of the pixel are relative to the marker of the probe. To describe those positions relative to the camera, we have to simply apply the received tracking transformations: First the tracking of the ultrasound H_{ut}, and then the inverse of the transformation from android marker to tracking system H_{ht}^{-1}. Be aware that in this

case we are independent of the movement of the optical tracking system, since moving the OTS will warp both tracked poses accordingly. The only problem may be that we might lose the tracking during the movement, or if one of the markers leaves the field of view of the OTS. This means that this approach will transform the corners $(p_x, p_y, 0, 1)^T$ of the ultrasound-image relative to the camera, if we also apply the hand-eye transformation H_{he}:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = H_{he} H_{ht}^{-1} H_{pt} H_{ip} S_{px} \begin{pmatrix} p_x \\ p_y \\ 0 \\ 1 \end{pmatrix} \tag{4.6}$$

The next step is to find the position on the screen for those corners. Therefore we apply perspective projection - division by z - to find the position at the $\{z = 1\}$ plane. Then we apply the camera-matrix from the intrinsic calibration to find for every corner the pixel in the camera view. The width and height w_c, h_c of the camera are known, so we know that only pixels in $[0; w_c - 1] \times [0; h_c - 1]$ are visible. This pixel frame has to be mapped onto the frustrum of the screen, see figure3.10. This can be done with dividing the pixel positions by width and height and multiplying them by two, so that all visible pixels lie in $[0; 2] \times [0; 2]$. If we subtract the result by one, the visible pixels lie in the correct set of visible points of the frustrum. But there is one last step to do: the y-axis of the screen pixel space is in the opposit direction of the y-axis of the camera pixel space. This can be fixed by mirroring the pixel space around the X-axis. The final matrix from perspective projection points $(X/Z, Y/Z, 0, 1)$ to the normalized screen coordinates then is:

$$\begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2/w_c & 0 & 0 & 0 \\ 0 & 2/h_c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} K \begin{pmatrix} X/Z \\ Y/Z \\ 0 \\ 1 \end{pmatrix} \tag{4.7}$$

Where K is the camera matrix of the pinhole model. Those matrices can be combined and stored into a single matrix, the model view projection matrix.
In summary we apply the transformation to bring the image to the view of the camera via the image to probae matrix, the tracked poses and the hand-eye transformation. Afterwards the vertices will be projected perspectively, and mapped onto the pixel in the $z = 1$ plane. Those pixel will be mapped onto the normalized screen coordinates via normalizing with the resolution of the camera and multiplying by two, shifting by one in negative - and y-direction to coincide both origins, and finally mirroring by y. This way the ultrasound images get rendered into the view of the camera. You may notice, that we lose the information of the z-coodrinate in the above matrices, but this is wanted. We do

no culling, and simply draw everything in the correct order with its own shader programs.

Chapter 5

Results

The application has been implemented on a Samsung J5 2017 Android smartphone. OpenIGTLink was used via a modified version of the provided Java library available at "https://code.google.com/archive/p/igtlink4j/". For optical Tracking the POLARIS Optical Tracking system has been used. Refer to "http://www.ndigital.com/medical/wp-content/uploads/sites/4/2013/12/Polaris.pdf" for more information. To compute the hand-eye calibration each time 20 captures were taken, and they were calibrated via [Tsai 89] since the used Optical Tracking System was not able to find the calibration pattern. As server for streaming the ultrasound and tracking messages [Lass 14b] has been used. The ultrasound probe is already calibrated to fit [Lass 14b].
For the tests a phantom has been built. This is a box, where nylon threads have been in the box, so that the nylon threads form a cross with one tangenting point in the middle of the cross. The position of this tangent point relative to the calibration pattern has been measured. The Android smartphone has been taped on a small plank with the marker taped approximately 40cm along the plank facing the same direction as the android main camera. This has been done so that the android smartphone is trackable even behind the ultrasound probe.

Quantitative Evaluation

In the quantitative evaluation the hand-eye calibration will be computed multiple times. After each time the android smartphone will take one extra capture. This will lead to one pose of the calibration pattern relative to the smartphone, and one tracked pose of the marker for the android device. This will be taken from then on as the current pose

of the smartphone for the misalignment computation.
This way we can compute the position of the tangent point in the cross two different ways: For one, the position of the point is known relative to the pattern. Since we know the pose of the pattern relative to the camera, we can simply apply this transformation on the tangent point and this way receive the position relative to the camera. On the other side, we can find the cross in the ultrasound image, if we position the probe to do so. With this, we know the pixel position of the tangent point in the ultrasound image and 4.6 will give us th position of the point relative to the camera.
Both positions compared will lead to the misalignment of the hand-eye calibration. 6.3 shows the results from several different calibrations. The ultrasound has been moved in between captures to see the tangent point from different orientations, and for one calibration also different positions of the camera to the pattern have been captured and tested.
If the position computed via pattern stays the same in 6.3 this means, that the same pose of the camera has been used, but the ultrasound probe has been moved. But if the pose of the pattern changes, the ultrasound probe was still rigidly fixed on the same position. This way only the pose of the camera changed, not the pose of the probe. This way we can see for example that with the fifth calibration, the pose of the ultrasound probe did not change, only the pose of the camera.

Empirical Evaluation

Since this is an apllication, we can test its usage with real participants. The application has been empirically tested with the given test-setup. But this time, the participants were given three different Tasks to fulfill. First, given a ultrasound probe, find the crossing point in the phantom. They had to give a signal at the start and at the end - when they thought they had found the tangent point. This had to be repeated three times, and how long each attempt took was measured. We provided the participants three different use-cases of the ultrasound image. In the first case, they had to perform the task thrice with the standard usage of the ultrasound probe - the probe and a fixed screen where only the current ultrasound image was shown. In the second case, they were given the application pre calibrated, so that they could immediately start without taking into account the calibration or errors due to bad calibration. This time they had to find the tangent point with the ultrasound image with the application without guidance. The Smartphone has been fixed onto the table to match the field of view of the camera best to the task environment. Int the third case, the participant was again given the application, but this

time we stored a goal position for them to easily find the cross. The results can be seen in 6.1.

Delay Evaluation

In the implementation there was a perceiveable delay in the application. This delay can be measured. During the empirical evaluation we recorded the complete setup with another smartphone recording with 30 frames per second. In some videos both the real world - probe and phantom - and the screen of the android device running the application were visible at the same time. In those cases it was possible to measure the delay in frames until the android device either showed the same movement due to to tracking in the application as happened in reality, or until the same ultrasound image shown at the computer with the server was visible in the android application. The phantom was not completely under water, so at the start and end of the tasks a very large movement out of water and back again into water was necessary. When the probe left the water, it immediately changed its view drastically. So, even if in some videos the screen displaying the ultrasound image is not visible, we can still see when the probe leaves the water, and when the application shows the start of the same large movement, or renders the first time the ultrasound image captured outside of the water. This has been stored into 6.2

Interpretation

In the quantitative evaluation the mean distance towards the wire was 23.39mm with a standard deviation of 14.62mm and the angle between the vertices was 2.33° with a standard deviation of 1.62°. In Comparison [Rose 01] had a mean deviation of only 2.48mm. [Palm 15] had an overall distance error of 6.5mm in the marker tracking with a standard deviation of 23.1mm and an overall mean angle error of 0.32∘ with a standard deviation of 4.6°. The angle error can not directly be compared since in our case the translational error is also present in the rotational error. For the deviation it can be said, that [Tsai 89] alone is not sufficient to provide accurate hand-eye calibration in this setup. This is most likely due to the delay of sending the tracking data via network combined with the low frame rate of 1-0.5Hz with which OpenCV captures images during the calibration process.
There was a large delay during the augmented application with a mean delay of 1.15 seconds. This delay may result from different aspects. For one, the network itself may result in a small delay, but the images where still small enough to be sent and received

with 21.5Hz. And since the delay did not really increase over time, it is not likely to be due to reading the OpenIGTLink messages in time The second reason might be the rendering via OpenGLES. The rendering of the ultrasound data was kept rather simple, so there may be room for optimizing exchanging the live ultrasound images.
The empirical evaluation received quite good feedback from the participants. Regarding the time the participants needed to fidn the cross node, they needed on average 8.58s with the standard setup, 9.56 seconds with AR application and 9.4s with additional guidance. This might look like the approach did not help but even worsen the usage of the ultrasound probe, but if we consider the time delay of 1.15s then both cases with the provided application where faster on average. Furthermore, there was not considered, that in one case the cross node has not been found with the standard setup. It might be interesting to see the results if a tablet would have been used instead of the small screen of the smartphone, since two participants had problems with the small screen. As had been mentioned, the task to find the cross node was rather simple, it would maybe way harder to find a complete 2D-plane again as is necessary for example in echocardiography of in viewing arteries.

Chapter 6

Conclusion

With this a pose aware rendering of live ultrasound data into augmented reality application has been provided. For future work it may be interesting to include an additional camera to the android device to use it as a head mounted display. Right now the goal pose is fixed. But if the ultrasound is used on living beings, it is most unnlikely that they hold still since every creature has to breath. For this it may be interesting to store the goal pose in a way so that nonrigid movement of the goal can be handled.
As a last word, it may be nice, if this would be expended to store multiple goal poses. With this it would be possible to build a gamification to learn ultrasound usage, for example in exhibitions as a parcour through the anatomy.

Table 6.1: Qualitative Evaluation of the application. Task "Standard" is to find the cross only with the standard ultrasound screen. Task "AR" is to find the cross with the given AR application, but no guidance, and task "Guided" is to find the cross via AR application with guidance to the cross.

Person	Task	1st	2nd	3rd Time	Behaviour
1	Standard	-	4.18s	5.11s	1st attempt cross not found, wrong area. 3rd not exact
	AR	21.22s	5.28s	5.14s	got confused in 1st attempt by phantom borders. "so cool by the way"
	Guided	4.21s	5.02s	4.14s	"super easy"
2	Standard	6.26s	20.13s	8.28s	wrong area at start in 2nd attempt
	AR	missing	10.5s	6.26s	"wow"; 1st attempt not recorded
	Guided	14s	10.1s	20.1s	
3	Standard	10.25s	5s	5.19s	
	AR	17.25s	13.19s	4.19s	"too small for me."
	Guided	14.03s	5.19s	8.2s	
4	Standard	11.23s	6.05s	4.05s	
	AR	4.25s	7.09s	9.07s	
	Guided	10s	6.28s	9.23s	
5	Standard	5.25s	4.2s	3.17s	experienced user, found the cross in the 1sr attempt in 2.26s and then verified the pose.
	AR	17.07s	5.25s	8.08s	"too small, maybe show ultrasound on the left third of the screen."
	Guided	14.25s	12.26s	4.03s	1st attempt only participant looking away to not see the pose before start. "I think the pose was wrong"
6	Standard	27.10s	22.27s	18.27s	held the probe rotated and stood at a weird angle to the phantom.
					"the task is too easy. It is way harder to find a plane correctly."

Table 6.2: Delay evaluation of the application. The Smartphone, the screen showing the ultrasound and the scene have been recorded simultasneously with one camera recording with 30 frames per seconds. The frames of delay have been counted in the videos. The delay of renderin the camera was between 6 and 11 frames. Delays in the same row were taken at the same time. "Ultrasound" means the delay until the ultrasound image in the application showed the right image. "Guidance" sais wether the guidance was on, and if the textual guidance was in use. "Tracking" means the delay the tracking had until the start of a movement in the tracking is the first time visible in the application.

Time	Ultrasound	Tracking	Guidance
1.07s	-	32	text
0.53s	-	16	text
1.57s	-	47	text
2.00s	-	60	text
1.00s	-	30	text
1.33s	-	40	text
1.00s	30	-	off
1.60s	48	-	off
1.10s	33	-	off
1.67s	50	-	off
1.00s	30	-	off
0.77s	23	-	off
0.73s	22	-	off
0.70s	-	21	off
1.37s	-	41	off
0.93s	28	28	on
1.57s	47	47	on
1.60s	48	48	on
1.07s	32	32	on
0.93s	28	28	on
1.23s	37	37	on
0.73s	22	22	on
1.07s	-	32	on
1.20s	-	36	on
0.97s	29	-	on
1.17s	35	-	on
1.15s			on average

Table 6.3: Quantitative Evaluation of the hand-eye calibration. A nylon cross has been calibrated to a pattern. This cross will be found in the ultrasound image, transformed to be described relative to the camera. This vector will be compared to the vector found via finding the pattern relative to the camera, and so the cross relative to the pattern transformed to be described relative to the camera. Calibration 1 to 3 were done wiht a marker mounted on a stick 30cm away from the camera, 4 and 5 were done with the marker mounted on the backside of the android smartphone. All entries are in millimeter, the angle in degree. the length of the tracked wirepose to the android camera is also shown as "length"

calib	wirePose via Pattern	via hand-eye	length	distance	angle
1	(56.89, 228.85, 189.62)	(51.92, 238.17, 192.96)	310	11.08	1.37
	(56.89, 228.85, 189.62)	(52.19, 232.76, 195.58)	308	8.54	1.16
	(56.89, 228.85, 189.62)	(52.24, 232.44, 194.93)	307	7.93	1.12
	(56.89, 228.85, 189.62)	(54.18, 232.85, 195.12)	308	7.32	0.79
	(56.89, 228.85, 189.62)	(53.22, 232.88, 194.97)	308	7.64	0.94
	(56.89, 228.85, 189.62)	(53.29, 232.21, 196.01)	308	8.07	1.03
	(56.89, 228.85, 189.62)	(53.29, 232.11, 195.96)	308	7.99	1.03
2	(34.74, 247.25, 296.39)	(34.97, 247.67, 296.92)	388	0.71	0.03
	(34.74, 247.25, 296.39)	(35.22, 247.61, 296.97)	388	0.83	0.06
2	(34.74, 247.25, 296.39)	(33.95, 247.45, 294.37)	386	2.18	0.24
3	(211.5, −255.2, 266.34)	(194.88, −239.20, 277.01)	415	25.43	3.16
	(96.70, 272.23, 315.15)	(91.02, 276.86, 317.94)	431	7.84	0.92
	(96.70, 272.23, 315.15)	(91.27, 277.07, 316.79)	430	7.46	0.90
4	(36.33, 289.11, 377.78)	(41.05, 289.34, 346.77)	453	31.37	2.54
	(−35.46, 288.74, 318.63)	(−30.98, 291.59, 314.32)	430	6.84	0.88
	(89.15, 243.55, 300.74)	(95.34, 242.05, 284.35)	385	17.58	1.92
	(19.98, 293.94, 285.56)	(24.46, 295.62, 296.41)	419	11.86	1.06
	(79.53, 325.83, 381.04)	(86.28, 327.81, 362.78)	496	19.57	1.84
	(−118.00, 259.18, 322.14)	(−112.86, 264.12, 295.01)	412	28.05	2.90
	(−26.86, 284.94, 235.98)	(−22.19, 283.48, 225.87)	363	11.23	1.26
	(249.85, 51.83, 321.57)	(250.98, 50.15, 352.51)	435	31.00	2.46
5	(64.36, 250.21, 293.33)	(73.29, 246.96, 339.16)	426	46.80	4.36
	(202.66, −36.075, 491.00)	(200.20, −40.86, 524.88)	563	34.31	1.57
	(125.85, 306.23, 385.87)	(130.86, 310.14, 368.70)	499	18.30	1.80
	(155.97, 80.26, 547.97)	(153.81, 74.31, 556.95)	583	10.99	0.82
	(295.73, −40.15, 189.32)	(290.77, −44.85, 229.98)	373	41.23	5.69
	(19.151, 290.98, 195.44)	(27.50, 288.15, 243.59)	378	48.95	6.39
	(66.074, 222.21, 350.35)	(76.17, 220.28, 336.56)	409	17.20	1.86
	(−36.98, 247.37, 330.83)	(−39.05, 245.72, 380.63)	454	49.87	3.93

List of Figures

Bibliography

[Aaft 09] M. Aaftab, G. Dan, and S. Dave. *OpenGL ES 2.0 programming guide*. Addison-Wesley, 2009.

[Aken 08] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. Peters, Wellesley, Mass., 3rd Ed., 2008.

[Baju 92] M. Bajura, H. Fuchs, and R. Ohbuchi. "Merging virtual objects with the real world: Seeing ultrasound imagery within the patient". Vol. 26, pp. 203–210, 07 1992.

[Baju 95] M. Bajura and U. Neumann. "Dynamic Registration Correction in Video-Based Augmented Reality Systems". Vol. 15, pp. 52–60, 10 1995.

[Benl 15] B. Benligiray and C. Topal. *Lens Distortion Rectification Using Triangulation Based Interpolation*, pp. 35–44. Springer International Publishing, Cham, 2015.

[BROW 66] D. C. BROWN. "Decentering Distortion of Lenses". *Photogrammetric Engineering and Remote Sensing*, Vol. , No. , p. , 1966.

[Crai 05] J. J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson-Prentice Hall, Upper Saddle River, NJ, 3rd Ed., 2005.

[Fabi 12] S. Fabien. "Tangentialvektor.svg". http://visp-doc.inria.fr/doxygen/visp-2.6.2/calibrateTsai_8cpp-example.html, 3 2012. taken at 2nd of december 2017.

[Hart 13] R. Hartley, J. Trumpf, Y. Dai, and H. Li. "Rotation Averaging". *International Journal of Computer Vision*, Vol. 103, No. 3, pp. 267–305, Jul 2013.

[Heik 00] J. Heikkila. "Geometric camera calibration using circular control points". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 22, No. 10, pp. 1066–1077, Oct 2000.

[I Ha 11] R. I. Hartley, K. Aftab, and J. Trumpf. "L1 rotation averaging using the Weiszfeld algorithm". pp. 3041–3048, 06 2011.

[Joe 12] H. Joe, M. Cleve, W. Peter, B. F. Ronald, M. Bruce, P. Roldan, and R. Karin. "JAMA: A java Matrix Package". http://math.nist.gov/javanumerics/jama, 11 2012. taken at 2nd of december 2017.

[Kiss 14] G. Kiss, S. Storve, B. O. Haugen, and H. Torp. "Augmented reality based tools for echocardiographic acquisitions". In: *2014 IEEE International Ultrasonics Symposium*, pp. 695–698, Sept 2014.

[Lass 14a] A. Lasso, T. Heffter, A. Rankin, C. Pinter, T. Ungi, and G. Fichtinger. "PLUS: Open-Source Toolkit for Ultrasound-Guided Intervention Systems". *IEEE Transactions on Biomedical Engineering*, Vol. 61, No. 10, pp. 2527–2537, Oct 2014.

[Lass 14b] A. Lasso, T. Heffter, A. Rankin, C. Pinter, T. Ungi, and G. Fichtinger. "PLUS: Open-source toolkit for ultrasound-guided intervention systems". *IEEE Transactions on Biomedical Engineering*, No. 10, pp. 2527–2537, Oct 2014.

[Mall 04] J. Mallon and P. F. Whelan. "Precise radial un-distortion of images". In: *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, pp. 18–21 Vol.1, Aug 2004.

[mcSu 08] mcSush. "Tangentialvektor.svg". https://commons.wikimedia.org/wiki/File:Tangentialvektor.svg, 9 2008. taken at 2nd of december 2017. Text has been modified.

[Moak 02] M. Moakher. "Means and Averaging in the Group of Rotations". Vol. 24, p. , 04 2002.

[Open] dev team OpenCV. "OpenCV". https://www.docs.opencv.org/3.1.0. taken at 2nd of december 2017.

[Palm 15] C. Palmer, B. Haugen, E. Tegnander, S. H Eik-Nes, H. Torp, and G. Kiss. "Mobile 3D augmented-reality system for ultrasound applications". p. , 10 2015.

[Rose 01] M. Rosenthal, A. State, J. Lee, G. Hirota, J. Ackerman, K. Keller, E. D. Pisano, M. Jiroutek, K. Muller, and H. Fuchs. *Augmented Reality Guidance for Needle Biopsies: A Randomized, Controlled Trial in Phantoms*, pp. 240–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[Simo 07] G. Simone. "Kamerakalibrierung mit radialer Verzeichnung - die radiale essentielle Matrix". Sept 2007.

[Stat 96] A. State, M. A. Livingston, W. F. Garrett, G. Hirota, M. C. Whitton, E. D. Pisano, and H. Fuchs. "Technologies for Augmented Reality Systems:

Realizing Ultrasound-guided Needle Biopsies". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 439–446, ACM, New York, NY, USA, 1996.

[Stol 14] P. J. Stolka, P. Foroughi, M. Rendina, C. R. Weiss, G. D. Hager, and E. M. Boctor. *Needle Guidance Using Handheld Stereo Vision and Projection for Ultrasound-Based Interventions*, pp. 684–691. Springer International Publishing, Cham, 2014.

[Toku 09] J. Tokuda, G. Fischer, X. Papademetris, Z. Yaniv, L. Ibanez, P. Cheng, H. Liu, J. Blevins, J. Arata, A. Golby, T. Kapur, S. Pieper, E. Burdette, G. Fichtinger, C. Tempany, and N. Hata. "OpenIGTLink: An Open Network Protocol for Image-guided Therapy Environment". *Int J Med Robot*, Vol. 5, No. 4, pp. 423–434, 12 2009.

[Tsai 89] R. Y. Tsai and R. K. Lenz. "A new technique for fully autonomous and efficient 3D robotics hand/eye calibration". *IEEE Transactions on Robotics and Automation*, Vol. 5, No. 3, pp. 345–358, Jun 1989.