Virtual Hand Interactions with ARKit

Bidipta Sarkar
Stanford University
bidiptas@stanford.edu

Abstract

An ideal user input method for immersive interactions in augmented reality (AR) would treat the user's body as a part of the AR system. In this work, we use Apple's ARKit to predict 3D locations of hand joints for a headset-based application. We set up a collaborative AR experience with an iPhone and an iPad using Multipeer Connectivity to triangulate the hand joints. We verify that this pose reconstruction satisfies the constraints on joint connections and adequately handles occlusions. Finally, we show examples of our application working in real time. The source code is located at `https://github.com/bsarkar321/ARHandInteraction`.

1. Introduction

A big question when designing augmented reality experiences is choosing an immersive user input method. Many mobile AR apps rely on touchscreen input, but this would not be practical for AR experiences where headsets are used. Another common option is the use of handheld controllers with buttons, but this limits the expressiveness of user input. An ideal solution would instead track body movements, especially hands, and insert a valid model of the user's pose into the virtual space. In this project, we use Apple's ARKit [1] with my phone mounted on my head along with a secondary iPad to track my hands in an AR experience. With this, we want to determine if hands can be expressed in AR and perform complex gestures.

We approach this problem by first finding the locations of finger joints projected into both cameras. With this information, we can construct rays from each camera representing the possible locations of the joints. We then find the points that best satisfy the ray constraints from both views as estimates for the 3D joint locations.

2. Related Work

The existing literature on virtual hand interactions investigates a variety of approaches to estimating 3D hand poses.

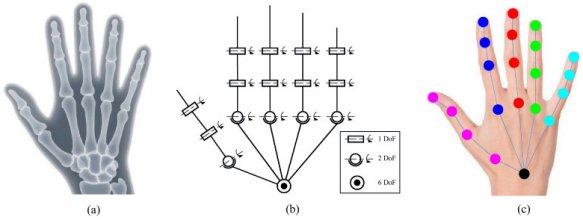


Figure 1. The typical hand model.

Many approaches use depth cameras along with RGB video streams to collect shape information [7]. However, none of our devices have depth sensors, so we would not be able to implement these techniques in ARKit. Many other techniques involve reconstructing the entire hand mesh [6], but this would be too expensive to perform in real-time on a mobile processor.

Apple's Vision framework [3] provides a small set of optimized 2D vision tasks, including 2D hand joint detection. Vision also gives predictions for occluded joints, which makes this a very helpful framework for our task. To find the best fitting point, we can use the least-squares intersection of lines technique [8].

2.1. Hand Model

Within the literature, there are 21 relevant joints on the hand [7], which are highlighted in Figure 1.

Each "bone" connecting these joints is fixed in length. At the base, we have the wrist, which can have any position and orientation. For each finger (excluding the thumb), we have the metacarpophalangeal joint (MCP), the proximal interphalangeal joint (PIP), the distal interphalangeal joint (DIP), and finally the tip of the finger [5]. For the thumb, we have the carpometacarpal joint (CMC), the metacarpophalangeal joint (MP), the interphalangeal joint (IP), and the tip.

The MCP and CMC are ball joints with 2 degrees of freedom. The other finger joints have 1 degree of freedom since they must bend in the direction of the finger.

3. Approach

The main problem is to reconstruct the 3D locations of hand joints in real time given images of two hands from different perspectives. We also have access to the camera matrices associated with each device. In the following subsections, we will describe how this can be accomplished within iOS.

3.1. iOS Frameworks

Within this project, we use three high-level frameworks that Apple provides to developers: ARKit [1], Vision [3], and Multipeer Connectivity [2]. In its current state, ARKit is able to track points in the real world and add a virtual overlay that appears to be anchored to the real world. The Vision framework is able to detect and track hand and bodies in real time [4], and it can determine the 2D locations of joints in screen space. Finally, the Multipeer Connectivity framework allows multiple devices to communicate with one another over a local network, allowing for shared AR interactions.

3.2. Datasets

The only data needed for this task are the real-time images from both cameras. The Vision framework already has pre-trained networks for (2D) hand pose detection, so additional offline datasets for hand segmentation are not necessary.

3.3. Physical Setup

The environment will consist of a head-mounted phone, a secondary ARKit-compatible device (like an iPad) with a good view of the hands, and a room with clear space so the user does not bump into other objects.

The head-mounted phone will have a stereo view with one AR scene for each eye. The user will need to look at their hands in order to get joint locations from both perspectives. Using Vision, the application will predict 2D locations of the joints.

The secondary device with a good view of the hands will also use the Vision framework to find the locations of the user's hand joints. Using Multipeer Connectivity, the secondary device will be in the same scene as the user, so it can communicate the state of the hands to the head-mounted phone.

4. Theory for Reconstruction

In this section, we describe ways to generate the 3D joint locations given 2D projections.

4.1. Single View Reconstruction

Using Vision, we can determine the 2D locations of the joints projected onto the camera. Using just this information, each joint is constrained onto a ray originating from the camera location, instead of a specific 3D point. We investigated if additional constraints could lead to a unique solution. Suppose we had the lengths of all the bones in Figure 1. Then, if we knew the wrist location, we can find the two possible locations for MCP joints for each finger by performing ray-sphere intersections. Unfortunately, we cannot verify which choice is correct because both choices project onto the same set of points.

In general, even if we know the full model of the metacarpophalangeal joints and the wrist and approximate that entire system as a rigid polygon, we will not know the precise 3D position. That problem is equivalent to calibrating a camera using just points on a single plane, which is a degenerate case for calibration. Therefore, single view reconstruction is not possible given just the 2D joint positions in the camera frame. Note that this is different from the single-view reconstruction from the literature, because they attempt to recreate the entire hand from the image, which has more information like shadows and finger thickness.

4.2. Two View Reconstruction

Suppose we want to find the 3D location of a single joint given its projections in two separate views. Suppose the camera origins are $\vec{o}_1$ and $\vec{o}_2$ in some common reference system. If $\vec{n}_1$ is the (unit-length) direction of the first ray and $\vec{n}_2$ is the direction of the second ray, we know that the point $\vec{p}$ satisfies

$$\vec{p} = \vec{o}_i + \vec{n}_i t_i \tag{1}$$

where $i = 1$ or 2, and t_i is the distance from camera i. To find the intersection point, we can use the least-squares intersection of lines technique [8] to get

$$\left[\sum_{i=1}^{2}[I - n_i n_i^T]\right]\vec{p} = \sum_{i=1}^{2}[I - n_i n_i^T]\vec{o}_i \tag{2}$$

where I is the $R^{3\times 3}$ identity matrix. To derive this formula, we start with the distance from $\vec{p}$ to each line. If d_i is the distance from $\vec{p}$ to line i, we know that it is the distance from $\vec{p}$ to the projection of $\vec{p}$ onto the line. Using the Pythagorean theorem, this is equivalent to

$$d_i = \sqrt{(\vec{p} - \vec{o}_i)\cdot(\vec{p} - \vec{o}_i) - ((\vec{p} - \vec{o}_i)\cdot n_i)^2}$$

We want to minimize the sum of the squared distances, which is $\mathcal{L} = \sum_{i=1}^{2} d_i^2$. To find the value of $\vec{p}$ that minimizes $\mathcal{L}$, we can take the derivative of $\mathcal{L}$ with respect to $\vec{p}$,

which we can evaluate as follows:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \vec{p}} &= \sum_{i=1}^{2} \frac{\partial [d_i^2]}{\partial \vec{p}} \\ &= \sum_{i=1}^{2} 2(\vec{p} - \vec{o}_i) - 2((\vec{p} - \vec{o}_i) \cdot n_i) n_i \\ &= 2 \sum_{i=1}^{2} (\vec{p} - \vec{o}_i) - n_i n_i^T (\vec{p} - \vec{o}_i) \\ &= 2 \sum_{i=1}^{2} (I - n_i n_i^T)(\vec{p} - \vec{o}_i)\end{aligned}$$

Setting this equal to zero, we can simplify as follows:

$$\begin{aligned}2 \sum_{i=1}^{2} (I - n_i n_i^T)(\vec{p} - \vec{o}_i) &= 0 \\ \sum_{i=1}^{2} (I - n_i n_i^T)\vec{p} &= \sum_{i=1}^{2} (I - n_i n_i^T)(\vec{o}_i)\end{aligned}$$

We can solve the normal equations to get the value of $\vec{p}$ as

$$\vec{p} = (M^T M)^{-1} M^T \sum_{i=1}^{2} [I - n_i n_i^T] \vec{o}_i \tag{3}$$

where $M = \sum_{i=1}^{2} [I - n_i n_i^T]$.

5. Implementation Details

For testing the system, I use an iPhone XS as the primary device and an iPad Pro (2nd generation) as the secondary device. As a headset, I'm using a standard Google Cardboard-like headset provided by Stanford. This headset has a removable lid so I can create a hole for the camera to see the environment.

5.1. Headset View

I modified an existing implementation of a headset view [9] so it is compatible with my current phone. However, I modified the left and right eye viewports to show the same AR scene, because otherwise the world felt like it was on a flat plane with hand joints in front of it.

Here is an example of the phone in the headset:

I also added a text UI element which displays general information about the application status, like if a device wants to connect.

5.2. Real-time Vision Processing

Using the Vision framework and asynchronous processing, I was able to get real-time 2d joint locations of the hand. The code runs at 60 fps on both devices, but the actual vision sampling rate is much lower.

To convert each (u, v) coordinate into a ray, I calculated the direction that each point represents using the formula

$$\vec{d} = \begin{bmatrix} \alpha(v - 0.5) \\ -(u - 0.5) \\ -1 \end{bmatrix} \tag{4}$$

where α is the ratio of the image width to the image height. The u and v are switched from their typical correspondences to x and y because Apple treats the "portrait" phone orientation as the default, but we had to swap these since the application is running in landscape mode. We also had to subtract 0.5 from each coordinate to shift the range from [0, 1] to [-0.5, 0.5]. Using homogeneous coordinates, we can express this vector as a direction by normalizing $\vec{d}$ and adding 0 as the fourth coordinate. However, this is still in the perspective of the camera. To get this into world frame, we just left-multiply by the $R^{4\times4}$ camera transform matrix, which encodes the camera's translation and rotation from the world origin. Note that this transformation is rigid and does not require a homogeneous divide.

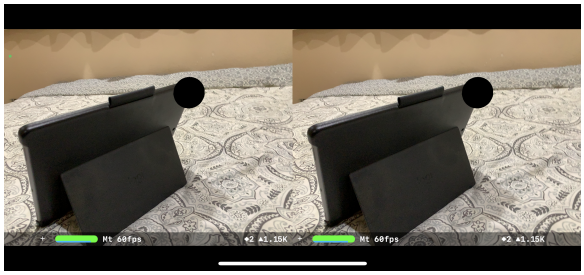
5.3. Multipeer Connectivity

Apple's Documentation for Multipeer Connectivity [2] demonstrates some techniques for enabling device-to-device communication. For this project, either device can be a browser or advertiser, since both sides will receive the same information from one another. Using ARKit, we automatically receive an ARAnchor representing the location and orientation of the other device when AR collaboration is enabled. Using these anchors, we can reproject the rays from the one device into the space of the other device.

One issue with the base Multipeer Connectivity anchor is that it finds the location of the center of the device. Unfortunately, this means that we have to hard-code the position of the camera relative to the device center. This creates a source of error in the system, and it may need to be recalibrated to fit different types of devices.

Each time the 2D joint locations are processed on one device, we need to send the new set of rays to the peer device. Each device has a separate "world-space" coordinate system but we can use the calibrated anchors to regenerate the rays. The only information we need to send between the devices is the 3D ray directions corresponding to each joint. Since there are 21 joints in total, this results in 63 floats, or 252 bytes per hand each time joint information can be communicated.

Each device associates the other device's calibrated anchor as a colored sphere. When properly calibrated, the sphere is centered on the camera of the other device, as shown below as a black sphere:



6. Experiment

Since this entire application is based on real-time data, we do not have ground-truth joint positions to compare our estimates to. Instead, as a quantitative evaluation of the quality of the reconstruction, we can measure the distances between various joints at different points in time. A successful algorithm will have a low standard deviation for the measured distances between connected joints. During the entire experiment, I moved my fingers and wrist to ensure that the system is robust to different hand poses.

I also tested various poses and motions hands as a form of a qualitative stress test. Visually, we can determine if the joint tracking points are following the expected locations of the joints despite occlusions.

7. Results

7.1. Quantitative Results

In Table 1, we measure the distances between joints over 100 samples from the environment. When joints should be rigid, the standard deviation is below a centimeter, which shows that the two view reconstruction case mostly preserves the joint distance constraints in real hands. For this test, the thumb was the closest to the phone, and the little finger was mostly occluded. We can see that the standard deviation for the thumb measurements is much lower than that of the little finger, which shows that having a clear view of the joints helps with reconstruction.

Joint 1	Joint 2	Mean	SD
Wrist	Index MCP	0.06831	0.00818
Ring MCP	Ring PIP	0.0348	0.00627
Little PIP	Little DIP	0.0223	0.00933
Thumb IP	Thumb Tip	0.0278	0.00371
Thumb Tip	Ring Tip	0.0938	0.03309

Table 1. Various Distances between joints in meters. The first 4 are rigid connections while the last one is not.

The last test in the table just evaluates the distance between the tip of the thumb and ring fingers, which do not have a rigid constraint. In this case, we see a much higher standard deviation compared to the rigid cases, as expected.

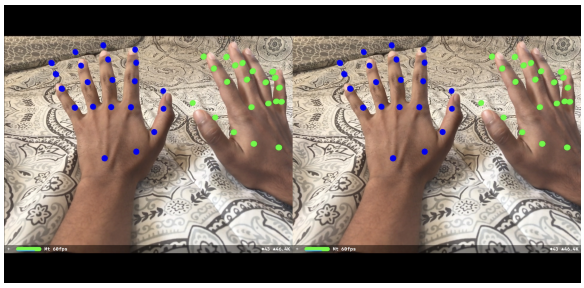
7.2. Common Qualitative Scenarios:

In all of the following scenarios, we are examining the screen of the primary camera, and the secondary camera is kept stationary around 30 cm away from the hands to the right of the scene. The two views are therefore perpendicular to one another. Furthermore, the left hand is typically occluded by the right hand in most of these views in the perspective of the second camera.

To help with visualization, we put spheres with a diameter of 1 cm at the predicted location of the 21 joints in each hand. The left hand joints are colored blue while the right hand joints are green.

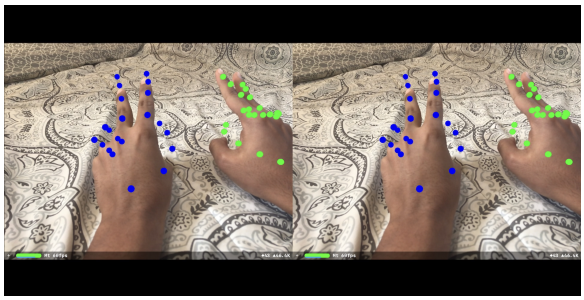
In general, I was able to get 60 fps for the display refresh rate, but the true sampling rate of the joint detection was around 30 Hz.

Here is a typical scenario for the hand tracking:



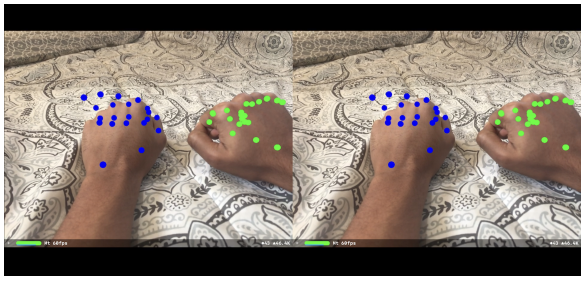
As you can see above, the projected tracking locations are slightly off from the true locations of the joints. The left little finger is almost entirely occluded from the perspective of the second camera, so its location is quite far off. However, the right little finger is quite accurate because it is in the full view of both cameras. The other fingers are reasonably located.

Here is a scenario where two fingers are placed down:



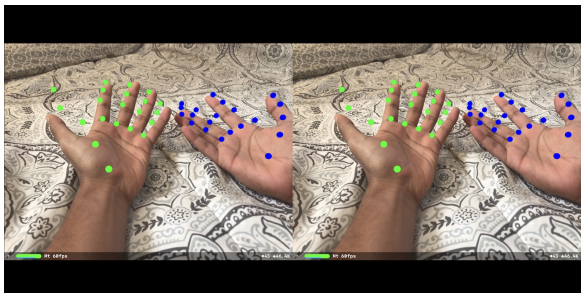
The index and middle fingers of both hands have reasonable predicted locations. The locations of the occluded fingers in the left hand seem to be reasonable since these fingers wrap into the palm. However, the occluded fingers on the right hand are predicted to be along the same line, which is unlikely.

Here is a scenario where the hands are fists:



All of the fingers are fully occluded, but it seems like most of the predictions are somewhat accurate. The wrist locations line up with previous measurements, and the knuckles (MCP) are on the correct spots for the right hand. The left hand has significant drift for the little and ring fingers because all of those knuckles are occluded from the second camera view.

Here is a scenario where the hands are facing upwards

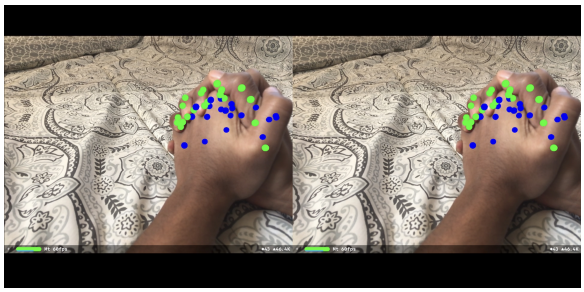


The current program somewhat works when the hands are facing upwards, as you can see good predictions for joint locations, especially on the right hand. However, Vision has a hard time processing the chirality (left or right handedness), of my hand when my palms are facing upwards. I worked around this in my code to ensure that both observations do not have the same chirality, but this means each hand could be misidentified.

7.3. Complex Hand Interactions:

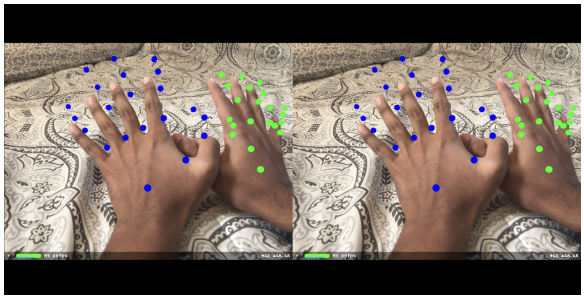
In this section, we test complex hand-hand interactions. Occlusions are already tricky for hand segmentation, but multiple hands open up the possibility for misidentifications of joints across hands.

Here is a scenario where two hands are clasped together



The joint locations look mostly fine, but there are a couple of issues with this reconstruction. The wrist locations for both hands are far off. Furthermore, the reconstruction seems to show that the left hand is inside of the right hand, but the fingers are actually interlacing. This demonstrates that the interactions of two hands harms 2D joint prediction in Vision.

Here is a scenario where hands are creating a shadow "butterfly"



The reconstruction of the right hand seems mostly fine, but the left hand seems to be suffering from drift due to occlusions from the right hand. Furthermore, the right thumb does not seem to have its points properly tracked because it is occluded by the left thumb in the primary camera's view.

8. Conclusions

The Vision framework appears to be quite robust and able to handle many scenarios where the hands are occluded. In addition, ARKit seems to handle object placement quite well when lighting conditions are ideal and there is not too much motion. However, there are still a few cases where the outputs are not completely reliable, like the chirality output when hands are facing upwards and in cases of hand-hand interactions.

Multipeer Connectivity was able to reliably send information between devices, though it often took a long time

to establish a solid connection. Overall, I think the idea of collecting 2D joint information across multiple devices has merit for hand pose reconstruction tasks.

In the future, I would like to continue making this system more robust. Currently, the "rigid" bone constraints are not a factor in the code, but adding them could make the system more stable overall. I would also like to see if full hand model reconstruction is possible with mobile chips, because this would allow for more sophisticated techniques for hand reconstruction with a single RGB data stream. I would also like to see it being used for grasping AR objects once it is more robust, because this could improve the expressiveness of AR applications.

The full source code for this project is located at `https://github.com/bsarkar321/ARHandInteraction`.

A video for single-view projections (with incorrect depth projections) is located at `https://drive.google.com/file/d/1fWSlkDgma4XyI2Jrr3vJpDZEXEFga5WP/view?usp=sharing`.

A video for two view reconstruction is located at `https://drive.google.com/file/d/1Rh1sHnOpdZxG6oVDIftx-JYUXNY7n2GF/view?usp=sharing`.

References

[1] Apple. Arkit. `https://developer.apple.com/documentation/arkit/`.

[2] Apple. Multipeer connectivity. `https://developer.apple.com/documentation/multipeerconnectivity`.

[3] Apple. Vision. `https://developer.apple.com/documentation/vision`.

[4] Apple. Vndetecthumanhandposerequest. `https://developer.apple.com/documentation/vision/vndetecthumanhandposerequest`.

[5] Apple. Vnhumanhandposeobservation.jointname. `https://developer.apple.com/documentation/vision/vnhumanhandposeobservation/jointname`.

[6] M. de La Gorce, D. J. Fleet, and N. Paragios. Model-based 3d hand pose estimation from monocular video. *IEEE transactions on pattern analysis and machine intelligence*, 33(9):1793–1805, 2011.

[7] R. Li, Z. Liu, and J. Tan. A survey on 3d hand pose estimation: Cameras, methods, and datasets. *Pattern Recognition*, 93:251–272, 2019.

[8] J. Traa. Least-squares intersection of lines. `https://silo.tips/download/least-squares-intersection-of-lines`, 2013.

[9] H. Weng. Arkit headset view. `https://github.com/hanleyweng/iOS-ARKit-Headset-View`, 2018.