Rendering Physical Objects in VR

JASON LANGLEY, University of Washington

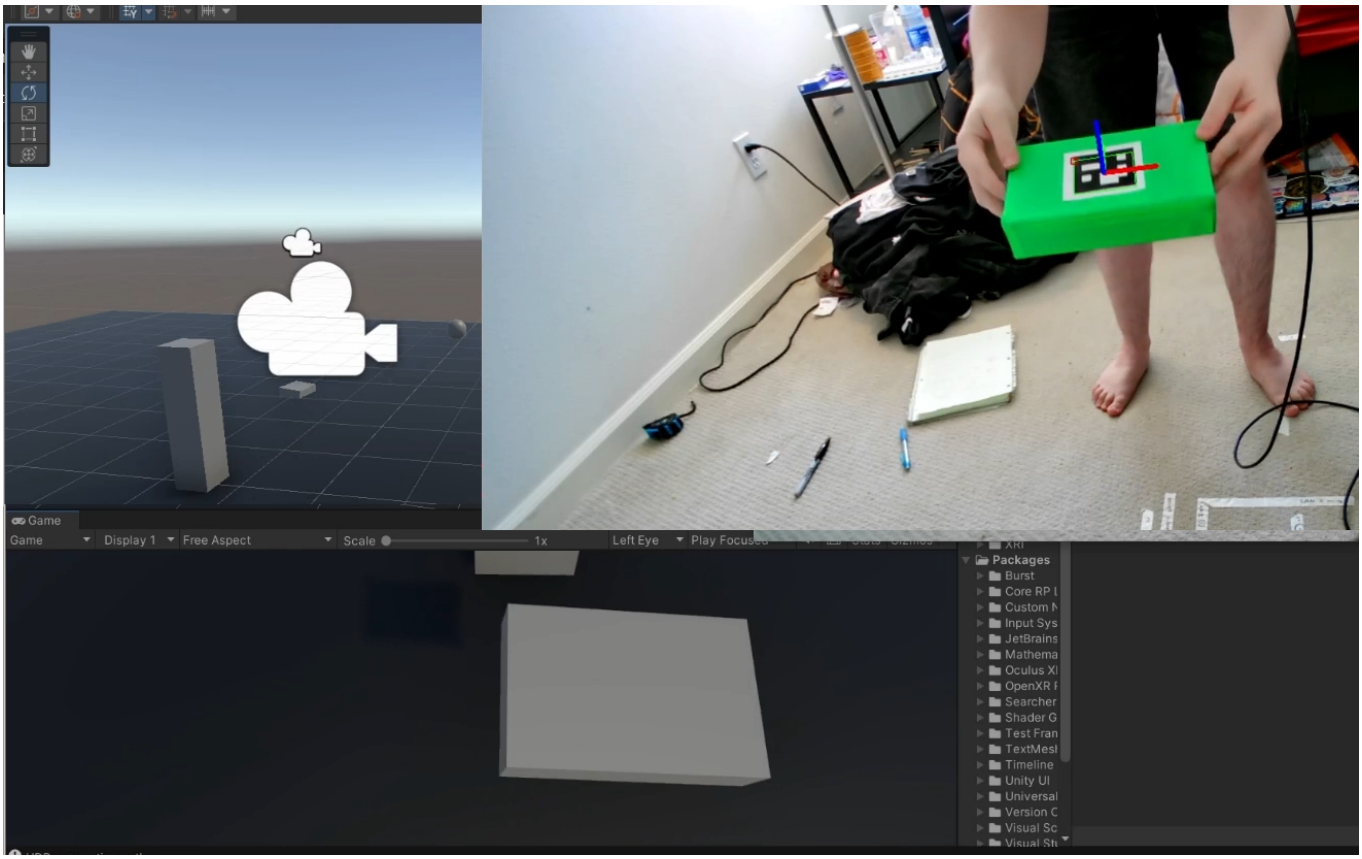

Fig. 1. A user wearing an HMD moves a specially marked box in real life. An equivalent box is simultaneously translated and rotated in VR. The HMD's view can be seen in the bottom panel.

VR systems are limited to audiovisual output in creating an immersive experience. In non-home settings, however, it's theoretically possible to use an instrumented environment to also engage a user's sense of touch. Users could manipulate a rendered VR environment by physically interacting with a specially-built real environment in commercial settings like amusement parks or escape rooms. This project investigated using computer vision techniques for tracking physical objects and recovering their world pose to be rendered in Unity VR.

1 INTRODUCTION

Virtual reality displays provide a convincing, immersive image of an environment. Even when the environment is rendered at lower fidelity, motion tracking provides enough additional information to the brain that the illusion can be realistic. However, this illusion dissipates easily- a user can't *touch* a simulated object, and their movements are constrained to the square footage of their room, regardless of the environment they're in. For home use, these are limitations that VR applications must work within. However, commercial 'VR experiences' as could be used in amusement parks or escape rooms have the ability to build a physical environment to provide the missing information. One envisions the old haunted house trick of a bowl full of peeled grapes, but the VR headset could provide the imagery of eyeballs.

In building such a system, several goals emerge that motivate possible solutions. For the infrastructure to be scaleable, adding 'new interactables' to the system should be relatively painless and put as few size/shape constraints on a tracked object as possible. Printed marker tags, for example, do not scale well to small objects or anything that comes in unpredictable quantities. (A silly goal I had was to track and render donut holes dyed with food coloring, as any good amusement park must sell snacks. A small box can be tracked with tags; *anything* edible presents considerably more challenges.) To me, this suggested taking a cue from chromakeying and painting objects a flat neon color that can be somewhat easily

Author's address: Jason Langley, jlangley@cs.washington.edu, University of Washington.

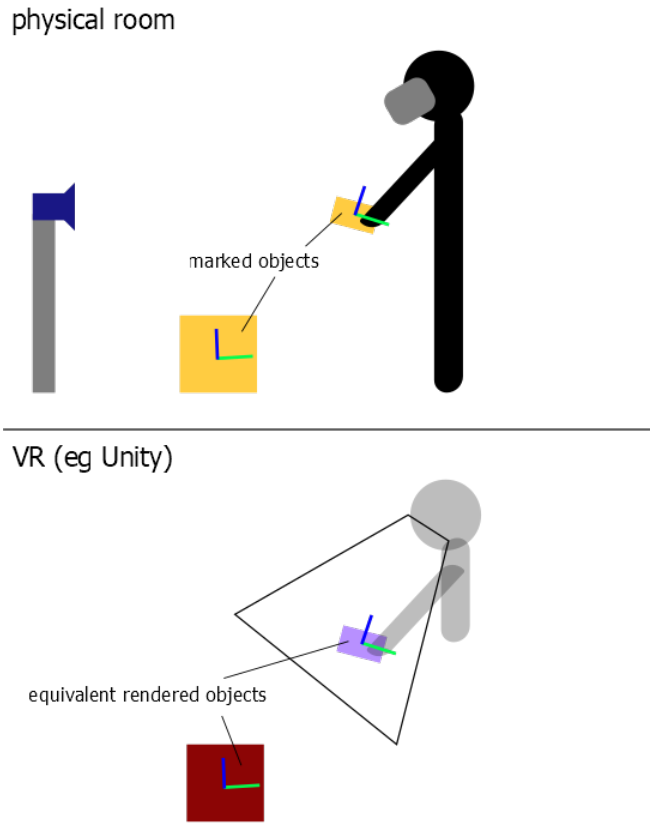

Fig. 2. The user moves a physical box while wearing an HMD. In VR, a rendered box moves equivalently.

isolated. Black or reflective dots could be added at key points, or different faces could be different colors, to aid PnP problem-solving.

Additionally, we need a system that can detect an object consistently, even in motion or in the presence of other occluding users. Inside-out tracking of some kind might be better suited to this than outside-in, and I did consider duct taping a webcam to a headset. After all, if a user can't clearly see an object from where they're standing, their headset might not need a high-quality pose estimate in order to render it for them. A motion-capture company did a demonstration of a 'physical VR' system where players wore backpacks that wirelessly sent and received pose data to the coordinating server and other players [Chagué and Charbonnier 2016]. *Some* kind of networking is likely necessary to do pose computation on a system monitoring the physical scene and then communicate this data to whatever is actually rendering the VR scene.

It's also important that the user's physical location within the scene be synchronized with their viewpoint's location in the virtual scene. (I should have anticipated this being an issue; standard VR doesn't need to be precious about your physical location beyond warning you that you're close to the edge of your play area, which requires a lower level of detail than we need to match two environments.) An MIT Media Lab project scanned the entire environment with a Google Tango to potentially make this problem easier [Sra and Schmandt 2015].

For the most part, I did *not* do any of these very sensible ideas. I hoped to experiment with non-marker-based tracking once I had the simpler version working, but didn't have the time; my proof of concept was entirely based on ArUco marker tags. My implementation used a single outside-in camera and OpenCV image processing to estimate object pose from these ArUco marker tags on a cardboard box. This was chosen largely because it seemed like a simple, accessible way to experiment with the concept without a ton of equipment or infrastructure. By the end of my time, I was able to get the simplest version of this process to mostly work, but it's very limited and wouldn't scale well. That said, with some extra time I think it could be expanded to a result that is at least *slightly* novel, considering how low-budget it is.

1.1 Contributions

To be frank, I didn't really contribute anything. I stapled a bunch of OpenCV tutorials together according to how I thought this system might need to work and what I could find, and dug through YouTube comments/StackOverflow to debug the tutorials. All the math I did was derived elsewhere.

2 RELATED WORK

The MIT Media Lab MetaSpace II project [Sra and Schmandt 2015] presents an interesting application of Google Tango to digitize the physical space, which simplifies (but does not fully solve) the problem of getting the virtual and the real geometry to match. This also streamlines the process of creating an analogous virtual space, which is an important goal for a robust, scaleable system. The project used the Microsoft Kinect as a cheap RGB-D camera for tracking both human and object poses. In retrospect, Kinects could be a great option to investigate for cheap outside-in tracking- however, given the time constraints and probable lack of user-friendly API, I didn't try to use one. The project's objects use marker tags for tracking due to ease of use and detection speed, but I hoped to explore more robust solutions to support objects that can't easily be tagged (small, strangely shaped, etc).

Researchers at Artanim, a foundation for motion capture research, presented a similar system for physical VR interactions, which seems to have seen commercial entertainment use with Dreamscape Immersive [Chagué and Charbonnier 2016]. Their project uses a motion capture system with multiple infrared cameras surveying each scene for human/object pose, noting that markers (in this case, reflective motion capture balls) need to be carefully placed on interactables in places users are unlikely to grab. The paper I found isn't as detailed, but its commercial use implies the system is reasonably robust, even though motion capture balls still need to be placed on each tracked object. This again wouldn't work with my goal of tracking edible objects.

3 METHOD

Objects first need to be chosen and marked in some way so that we can track them. In the simplest proof of concept I did, this just means a marker tag. Theoretical expansions of the concept would require

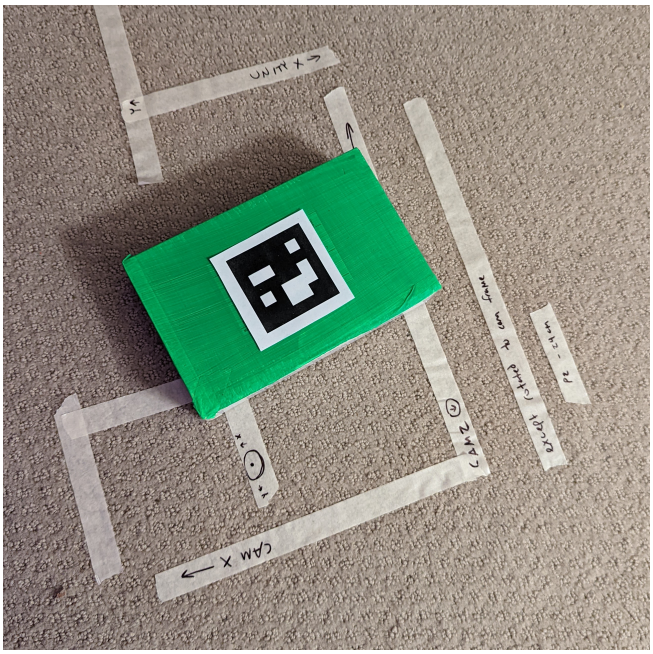

Fig. 3. The box used for 95% of this project, along with some tape markings used while trying to visualize different axes.

Fig. 4. The webcam can be seen at the right, looking down at the box, which is placed at the origin in this image.

painting objects bright colors and/or adding dots. The objects should presumably be chosen such that they can be painted or otherwise colored easily. Any object then needs to be carefully measured and replicated as a 3d model. Our chosen engine, Unity, uses meters as its units, so we make sure to measure all of our real-world quantities in meters as well. A camera must be placed surveying the play area. In theory, we'd require multiple cameras to ensure coverage and triangulate the location of certain objects.

We then 'calibrate' the system by choosing a world origin in the play area. This can be anywhere, but it's reasonable and easy to put it somewhere on the floor roughly in the center of the space. If multiple cameras were present, it would make sense to place the origin roughly equidistant from all of them. We do this by placing a marker tag on the floor and estimating a pose for it. This gives us a rotation necessary to transform between camera space- where image processing software will estimate distances- and our 'world' space, which we represent in Unity. We also get camera space coordinates of this origin. We then measure and translate objects relative to this origin, and we can use these translations natively with the Unity origin and get reasonable movement more or less for free.

At runtime, we process the camera feed to identify and track our marked objects. As often as we can, we compute world space coordinates for these objects as well as vectors to recover their orientation, then send these to be used to update the analogous object's translation and rotation in Unity.

Even if we measure carefully, the user's perspective is likely to be slightly off; in other words, the VR box will appear slightly translated to the user from where its real twin actually is. There are potential changes we could make to the system to alleviate this- using a marker tag to track the headset pose and moving the origin to the headset, reevaluating the hardware to do inside-out tracking rather than outside-in, etc. For the purposes of the proof of concept, this is adjusted by simply resetting the player's viewpoint in VR to a measured location while they stand at that location in real life. It's not perfect, but this somewhat remedies the problem.

4 IMPLEMENTATION DETAILS

4.1 Setup/calibration

I used a cardboard box I had laying around (pictured above), about 19x12x4cm. The green paint was cheap 'reflex green' Amsterdam acrylic, though the paint wasn't relevant to the work that I managed to complete. The ArUco marker was printed to be 5.5cm on a side, though note that this would be too large to fit additional markers on the narrower sides. I received an iMiSES 2k webcam to use- likely the CC1006- which is apparently capable of 2560x1440 images. However, when plugged into a PC, the image defaulted to a very small 4:3 image, and manually setting the resolution higher introduced a *severe* delay. I was unable to figure out what caused this and ended up having to live with about a 1 second delay on a 1280x720 feed. The webcam was mounted in a very high-tech manner on an air conditioning unit, aimed a bit towards the floor.

I then modeled the box in Blender, using some plugin to manually set specific edge lengths to match the real box. I left it untextured, as this was considered low-priority. Doing a Blender export to .fbx for Unity can try to move the axes to Unity's left-handed system for you. Some people online suggested this may not always work well, but for this very simple model it was fine. Additionally, I measured the distance between my chosen origin and the camera, and put a roughly analogous post there in Unity to confirm that directions and facing were correct.

OpenCV was used for all image processing. I spent several hours trying to install the C++ implementation before giving up and downloading the Python version, because that installation process is literally 'type a line in the terminal'. It's possible the Python implementation may cause performance to suffer in some cases, and there are a handful of OpenCV modules that don't seem to have been ported (a quaternion library, for one), but for proof-of-concept purposes it seems fine.

Our world origin is chosen by detecting an ArUco marker on the floor with OpenCV's ArUco library and estimating its pose. This pose estimation gives us **rvec**, or a rotational vector that rotates from the camera frame into the marker's orientation, and **tvec**, or a translation from the camera viewpoint *in camera space*. We can then use this to compute other markers' translations relative to this origin.

4.2 Computing object pose in world

The pose of a marked object is estimated by OpenCV analyzing webcam frames. This pose includes a position in camera space; call this P_{cam}. We also get the aforementioned rvec, with the rotation that transforms the camera space axes into the marker pose. We can use the Rodrigues formula (OpenCV provides an implementation) for converting this axis-angle vector to a rotation matrix; call this matrix R.

In the pinhole camera model, we have the following equation relating these quantities for some marker pose [Hoang 2017] (in retrospect I had all this information from course material but am dumb and needed it spelled out):

$$P_{cam} = R \cdot P_w + tvec$$

Writing this out I am now severely doubting that this is at all sound, but we continue on because I need to put something down. When we save our origin, we set P_w in this equation to $(0, 0, 0)$. P_{cam}, or the origin's location in camera space, is now equal to this original tvec. We save this P_{cam} and the rotation that moves it to $(0, 0, 0)$. We can rearrange the equation to yield:

$$R^{-1} \cdot (P_{cam,origin} - tvec) = P_w$$

... and then plug in the tvec of a new detected marker pose to compute its world coordinates. I now worry this doesn't make any sense, but when I moved the box to measured test locations and computed world coordinates, they matched my measurements, so I went with it. (Additionally, perhaps hinting that this whole thing is garbage, this gives coordinates with 3 flipped signs. I reversed the direction of the subtraction to yield the axes I expected.)

We then need to compute the rotation needed to rotate the Unity box object to match its detected pose. I had a grand theory that this could be done as follows: we have the rotation needed to rotate from the camera pose to 'world', and the Unity object is aligned to 'world' by default. We have a detected rvec that rotates the camera pose into the box. If we can compute the 'difference between' these two vectors, or the rotation needed to rotate 'world' into the box pose, this would be the rotation we should apply to the Unity box. This did not work, either because it's not logically sound or because the 'rotate one vector into another vector' computation I looked up was inappropriate for this situation. Rotating the real-life box about one axis would *almost* work in some cases, but mostly produced a strange off-axis rotation.

So I gave up and did more Googling and came across a reply [RCYR 2016] recommending that someone extract 'front' and 'up' vectors from the pose rotation matrix and use Unity's Quaternion.LookRotation function for this, which is effectively the Unity equivalent of the lookAt functions we've discussed. I tried this and it was correct,

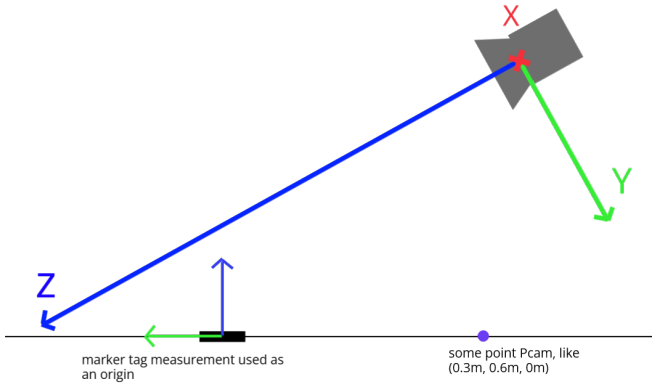

Fig. 5. An illustration of camera space. The camera looks down the +z axis, which is rotated into the viewing angle of the camera. A sample point in camera coordinates and a sample world origin are shown.

but appeared to be rotated into the camera frame, so I attempted to 'move it to world' by applying the R matrix from our origin computation to the pose rotation matrix first. This feels like a copout but it seems to work fine.

We bundle up the world pose we computed and these world space front/up vectors into a string and send them via UDP to Unity, using the Python sockets library.

4.3 Unity data reception

We receive the data in Unity in an infinite while loop. Communicating data between Python OpenCV and Unity was lifted from a tutorial [Singh 2018]; it was the only thing I could find that did not involve spending $100 on a Unity plugin. If we receive data that differs from the last data we processed, we compute a Quaternion that describes the object's rotation and compute a new position, then update the box's pose.

Many axes need to be switched around, flipped, or both. To be perfectly honest, I tried what people online recommended for moving to Unity and then trial-and-error waved the box around and flipped some signs to match my intended axes.

$$rot_{unity} = LookRotation((-f_w.x, f_w.y, f_w.z), (-u_w.x, u_w.y, u_w.z))$$

$$pos_{unity} = (-pos_w.x, pos_w.z, -pos_w.y)$$

This moves and rotates the box more or less correctly. However, there's no guarantee that the user's viewpoint in Unity will be perfectly aligned with their actual HMD. Experimentation often

put the Unity box a little more than a foot above and to the right of the real one. This motivates needing a better solution for object localization; inside-out tracking and calculating position relative to the HMD might provide a more accurate result. For now, however, I looked up tutorials on how to reset the Unity XR Origin to a specific location/facing and figured out how to bind it to a controller button. I then measured a specific spot on the floor, placed an invisible cube with that position and the facing I wanted, and reset the facing while standing on that spot. This does not fix the problem entirely, but it does at least improve things.

4.4 Additional

With very little time remaining before I had to hand this in, I spent some time playing with tracking green spheres, wondering if I could get a last-minute 'eat a green donut hole in VR' demo working. I couldn't, but I feel it's worth mentioning here. More specifically, I was able to tune a HSV-based mask and detectContours to track green balls fairly accurately, but recovering their world coordinates seems non-trivial. While I can calculate the difference between a ball's measured size and its detected 2d diameter in the frame to give some indication of distance from the camera, this does not give me an easy tvec that I can just plug into my existing position code.

There's likely some more math I could do to estimate a tvec for a detected ball. Additionally, if I set up a second camera, I could theoretically use this size-based camera distance and the two frames to triangulate a ball's position relatively easily. However, I did not have time for this (as I didn't have time for so many other things), so VR-simulated edibles will have to wait for another day.

5 EVALUATION OF RESULTS

This will largely discuss failures and limitations, because I managed to accomplish very, very little (although I will admit I feel like I understand how I could extend what I have to at least improve on this). I have exactly one success to talk about, which is that I managed to make the simplest, most-constrained, most elemental version of the idea work: one marker tag on one box, which rotates and moves the VR box roughly analogously. Effectively, I spent this many hours to replicate what may as well be the AR equivalent of 'hello world' (render a cube on a marker tag) in VR, and I relied on a huge pile of tutorials and StackOverflow answers to do it. At least I have a video of it kinda-working.

Limitations:

- With only one marker tag and one camera, we can't simulate any rotations of the box that would remove the marker tag from view.
  There are various ways to address this. We can use more cameras, and if an object is visible in multiple frames, we compute quantities with the data from multiple frames and then use the average. We can also use more tags, which is something I wanted to experiment with. (In theory, if we detect a marker on the *bottom* of the box, we rotate it 180 degrees about the y axis in camera space before computing the rotation we should apply to the box? I think?) We can also try ditching the markers entirely and moving to bright colors, dots, and OpenCV's solvePnP functions, which was my intended end result.
- Box motion and rotation are very jittery, low-framerate, and subject to incorrect readings.
  Having more cameras and/or tags gives us more data to smooth this out with. A very simple low-pass filter might improve the jittering. We should also probably be throwing out unrealistic measurements; marker tag readings often 'get confused' and the box will appear to jump to a strange angle.
  Additionally, the code as-written sets a new position and rotation value on the object every time new data is received. This was done for simplicity. However, Unity can theoretically do some motion interpolation for us if I learn how those transformation functions work, and that would probably present a more immersive effect.
  This problem is worsened by the significant delay on the camera feed, so fixing that issue somehow would also help. (The delay was not present when plugged into my significantly weaker MacBook, even through two USB-USBC adapters. I tried both USB2 and USB3 ports. I have no idea.)
- Marker tags, as discussed earlier, do not scale or support a range of objects that I want.

6 FUTURE WORK

I didn't do anything unprecedented or particularly interesting, so this isn't really applicable to anyone else's work. I do think additional tags on one object could be added by pre-applying a rotation to the box before calculating its in-world lookAt orientation. The code I have could be reasonably easily extended to support multiple cameras, multiple tags, and multiple objects (even if the results would still have some problems and significant limitations). Doing any kind of processing on the computed pose data (filtering, averaging multiple data points, motion interpolation, outlier removal...) could improve the effect.

Beyond improvements to the marker tag method, if I were to continue working on this (and I kind of want to), I'd experiment with more robust PnP-based methods and using color to make object identification and tracking easier.

7 CONCLUSION

I completely flamed out in this class but I want to at least turn in *something*.

ACKNOWLEDGMENTS

None of this would exist without wholesale copying a lot of tutorials and StackOverflow comments. My bibliography is thus pretty silly, but I want to credit all of the flimsy sources I used anyway.

For what it's worth, I really enjoyed the class even if I couldn't take advantage of the opportunity it provided. I wanted to come to the poster session but managed to get sick at the very end of the quarter.

REFERENCES

Justin P. Barnett. 2022. *Re-center your VR Player in Unity.* Retrieved June 7, 2023 from https://www.youtube.com/watch?v=EmjBonbATS0

Sylvain Chagué and Caecilia Charbonnier. 2016. Real Virtuality: A Multi-User Immersive Platform Connecting Real and Virtual Worlds. In *Proceedings of the 2016 Virtual Reality International Conference* (Laval, France) *(VRIC '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/2927929.2927945

Quang Hoang. 2017. *answer; Aruco markers with openCv, get the 3d corner coordinates?* Retrieved June 7, 2023 from https://stackoverflow.com/a/46370215

Nicolai Nielsen. 2022a. Building an Augmented Reality Application with ArUco Marker Pose Estimation in OpenCV. Retrieved June 7, 2023 from https://www.youtube.com/watch?v=GEWoGDdjlSc

Nicolai Nielsen. 2022b. generateAruco.py (code for marker tag generation). Retrieved June 7, 2023 from https://github.com/niconielsen32/ComputerVision/blob/master/ArUco/generateAruco.py

RCYR. 2016. *answer; OpenCV rotation (Rodrigues) and translation vectors for positioning 3D object in Unity3D.* Retrieved June 7, 2023 from https://stackoverflow.com/a/36580522

Gur Raunaq Singh. 2018. *Introduction to Using OpenCV With Unity.* Retrieved June 7, 2023 from https://www.kodeco.com/5475-introduction-to-using-opencv-with-unity

Misha Sra and Chris Schmandt. 2015. MetaSpace II: Object and full-body tracking for interaction and navigation in social VR. *CoRR* abs/1512.02922 (2015). arXiv:1512.02922 http://arxiv.org/abs/1512.02922