POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Colocation for Oculus VR Headsets through Hand Tracking Functionality

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Giancarlo Danese**

Student ID: 945265
Advisor: Prof. Pierluca Lanzi
Academic Year: 2022-2023

Abstract

This work aims at developing a Virtual Reality software which allows users to easily create a shared space environment where physical and virtual space coincide.

Users will use a hand gesture to trigger the colocation process and easily position other users who shares the same physical room with them, in the same virtual room, within the same reference system.

We will first describe other colocation methods with their pros and cons, then our personal hand-tracking method and under which circumstances our method is better or worst than the other already existing ones.

Then we will describe experiments and demos which aim at proving the preciseness of our colocation method together with an idea on how one could make use of colocation in a VR application.

Finally, we bring our conclusions on how the software can be enhanced and what can be expected from the future of VR.

Keywords: Colocation, Oculus, Virtual Reality, Unity, Hand Tracking, Shared Space

Abstract in lingua italiana

Questo lavoro consiste nello sviluppo di un software di Realtà Virtuale che permette agli utenti di creare facilmente un ambiente di spazio condiviso dove lo spazio fisico e virtuale coincidono.

Gli utenti useranno un gesto con la mano per innescare il processo di colocazione e posizionare facilmente gli altri utenti che condividono la stessa stanza fisica con loro, nella stesso spazio virtuale, all'interno di un sistema di riferimento comune.

Prima descriveremo gli altri metodi di colocazione esistenti con i loro pro e contro, poi il nostro metodo personale basato sul tracking delle mani e sotto quali condizioni il nostro metodo è migliore o peggiore degli altri metodi già esistenti.

Poi descriveremo le sperimentazioni e le dimostrazioni fatte che mirano a provare la precisione del nostro metodo di colocazione insieme a degli esempi su che tipo di uso si possa fare della colocazione in una applicazione di realtà virtuale.

Infine, concludiamo sul come il software potrebbe essere migliorato e su cosa aspettarci dal futuro della realtà virtuale.

Parole chiave: Colocazione, Oculus, Realtà Virtuale, Unity, Tracking delle Mani, Spazi Condivisi

Contents

1 | Introduction

Many Virtual Reality (VR) applications exploit multi-user experiences, such as multi-player games in VR (FPS style shooters [51] or Survival Horror games [16][43]), virtual meeting environments emphasizing social interactions or casual VR chat platforms. Even in the area of facility management, a multi-user shared virtual environment can be used to improve the efficiency of communication. Multi-user VR experiences can take place in a number of physical setups. Setups in which users share the physical and virtual space simultaneously are defined as colocated shared VR. Such colocated VR scenarios can be used in walkable arena-scaled environments (e.g., games or exploratory VR scenarios [24][1]), often exploiting the arena in a shooter style game, such as Space Pirate Trainer DX Arena [44].

They can also be used in seated VR scenarios, where users typically do not navigate extensively but still need to be aware of the positions of other users in the same virtual (and physical) environment. In a functioning colocated VR application, poses of all users within the same coordinate frame need to be known. Often, this coordinate frame is associated with the shared physical environment itself. Other softwares offer camera systems for large physical environments to track each user inside the shared space. These solutions always need external cameras and a complicated setup to implement a colocated shared VR experience, as we can see here at Oculus Connect [5].

Currently, head-mounted displays (HMDs) that use in-built visual SLAM (Simultaneous Localization and Mapping) techniques for head tracking are gaining popularity in the consumer VR market [12]. SLAM-tracked HMDs demonstrate a substantial advantage of not requiring an external camera setup for fast and precise 6-DOF (Degrees Of Freedom) tracking, often in environments that are larger than those that similarly-priced external tracking camera installations can cover. Tracking algorithms based on visual SLAM map the environment while simultaneously calculating the pose of the HMD within the map that is being created.

In Figure 1.1, a robot observes the environment relative to its own unknown pose. Also, the relative motion of the robot is measured. From this input, a SLAM algorithm computes estimates of the robot's pose and of the geometry of the environment. A camera on a robot measures the relative position of artificial features on the floor (light lines), while the sensor's motion is provided by the robot's odometry (light arrows). The output is the robot's pose (dark, circled arrow below the robot) and the global position of each feature (dark crosses). In Figure 1.2, a SLAM device is attached on a robot and the pose is calibrated via its head. After calibration, the SLAM device is located in front of the robot head, and the robot is localized in the map coordinate system.

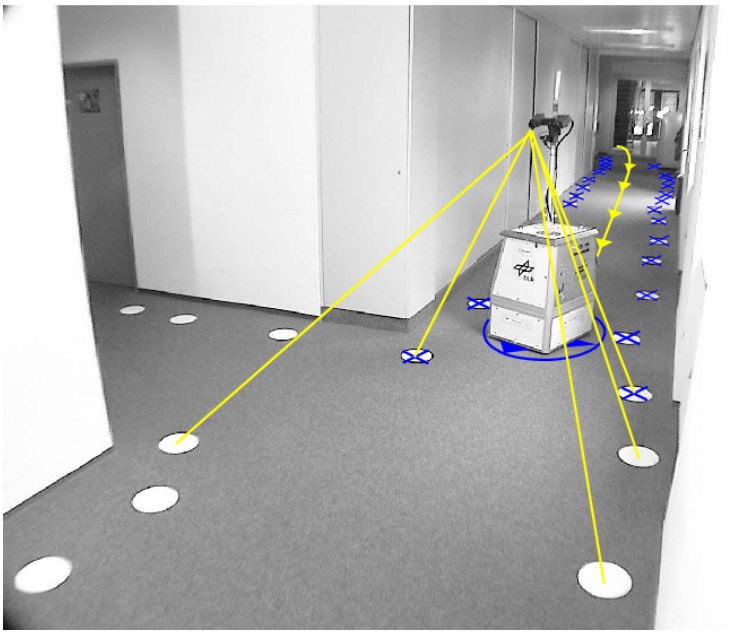

Figure 1.1: The camera equipped robot [12].

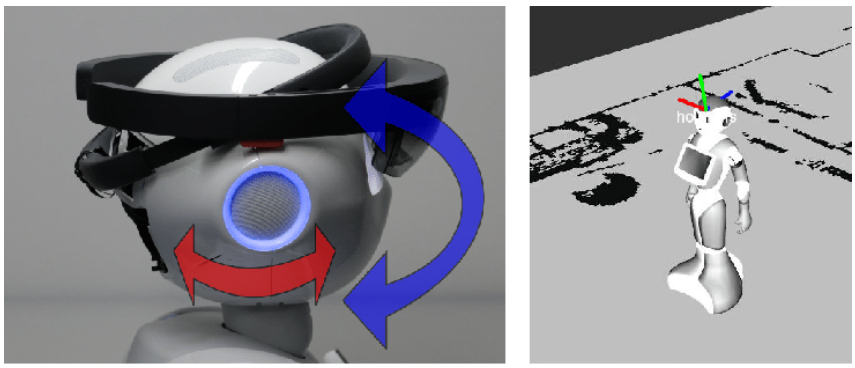

Figure 1.2: (a) Attaching the SLAM device on a robot and calibrating the relative pose via moving the robot head (b) [12].

Oculus Quest is an example of a SLAM-enabled HMD available to consumers. Compared to other popular VR headsets, the Oculus Quest is independent of external hardware such as tracking cameras. Being a standalone Android-running computer, it provides an entirely untethered experience. As a SLAM-tracked device, each Oculus Quest creates an individual tracking map that cannot be read out and copied to other devices. This fact makes creating a shared colocated VR experience a challenging task.

So we developed a software that enables users in the same room to easily colocate in the same virtual space, using hand tracking [23][26]. Hand tracking is a new functionality of Quests which let users ignore controller usage and use their hands to point, pinch, grab and much more. The reason hands can be used for colocation, is that Oculus Quests can recognize only a pair of hands at a time [48], and therefore the hands of a secondary user may be perceived as its own, making them an excellent object usable as a point of information sharing.

In this thesis, we will first speak about the current state of the art, describing the other 2 main methods for colocation, tested out by [34], with their requirements, their setup, and their calculations, also discussing for each method their pros and cons.

Then we will speak about our personal colocation method, used for this software, which exploits Oculus Quests' hand tracking functionality; a big section will help readers understand which math calculations we used for the colocation process, then an accuracy report will compare this method with the other availables, finally concluding with a discussion about pros and cons.

We will then describe 3 experiments present in the software, which will help users understand how colocation works; furthermore, following the various descriptions, each demo will have its personal accuracy evaluation that will demonstrate the preciseness of our colocation system. Each demo is developed directly inside the software and can be easily triggered by users with a series of controller buttons pressed simultaneously.

Finally, we will conclude by first describing how users can actually use the software and which changes they could bring to their own taste and/or usability, and then by describing possible future enhancements and some thoughts on the future of Oculus HMDs and hand tracking functionalities.

2 | State of the Art

In this chapter, we will spek about the other 2 possible different ways to colocate two or more users in the same room. We will first describe their basic mechanism and their requirements, both hardware and software, and we will then debate their pros and cons based on the above-said information.

2.1. Fixed-point Calibration

In the fixed-point calibration method, all colocated HMDs are placed at predefined positions within the physical environment. This calibration method is the most simple and straightforward of the two presented methods, and we will first describe an analytical procedure brought on by [34].

To prepare the physical environment for calibration, a specific point is fixed and marked (e.g., the room's center): we will call this point U_R. Then, a point in the virtual world U_V is manually set up that should correspond to U_R. After the calibration, a user at the position U_R in the physical space should have the position U_V in the virtual space. A set of distinct U_R and U_V positions is determined for each colocated user.

As a calibration example, we could position the HMD of each user on the floor at their corresponding U_R and rotate in the direction that is set with U_V. Reference points can be set at arbitrary distances, as long as their relative poses in the physical world correspond to those in the virtual environment.

In Figure 2.1, on the left, the users' headsets are positioned on predefined locations in the real world. On the right, virtual users who are repositioned to U_V, which is the virtual representation of U_R. The distance Δd_U is the same in the real and virtual world. Red arrows represent the view direction of the user.

As you might have inferred by the description above, you can either use the same reference points for all users, calibrating user positions one after another, or you can choose to setup a unique reference point pair for each user, enabling simultaneous calibration. Using the same reference points for all users means that every one of them will have to place their HMD on the predefined spot (usually on the floor) with the correct rotation. This can be incredibly time consuming, not to mention the recalibration effort that could be needed in case of a big positional or rotational drift that can happen during usage.

In the former case, a button could be pressed to associate each player with the predefined virtual position; every user must go back to the center of the room, facing a coherent direction with the virtual transform predefined in the software, while developers will ensure that the software's initial player spawn position respects the given parameters. Simultaneous calibration instead can be really fast and reliable, but it may need extra implementation steps, bigger effort by each player and a correspondence between the virtual scene and the physical room in which players are positioned;

Indeed, another possible implementation could be that every user gets marked with a number and a corresponding position in the physical room. The same mark is associated to a virtual position that is correspondant to the physical position associated to it; if the distance between these positions and the rotation of each one is coherent with the virtual ones set up by the developers, then pressing a button while each player is standing on top their mark is all that is needed to colocate them correctly.

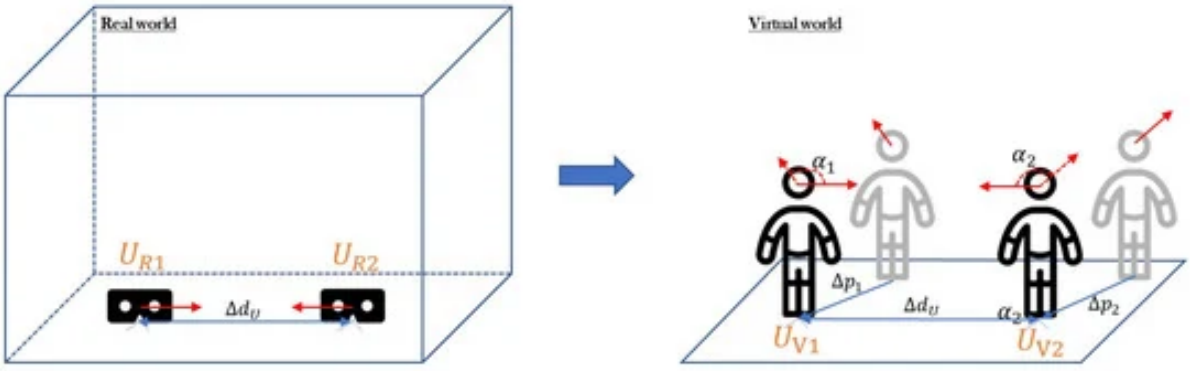

Figure 2.1: A visual representation of the fixed-point calibration mechanism [34].

2.1.1. Pros

First of all, fixed-point calibration doesn't need any additional hardware (in opposition to the marker-based calibration described in the subsequent section): a generic HMD is all that is needed to perform this kind of colocation. It is not mandatory to have an Oculus Quest or another HMD endowed of hand-tracking either, or any other type of specific tracking functionalities, as any SLAM-device is enough.

As said above, this is the easiest between the methods as no additional software and/or plugins are required either: in the case of the hand tracking colocation, plugin and functionalities must be manually activated from within the Quest software, while here, both software and hardware requirements are zeroed out. The absence of any type of gesturing, and the fact that no moving object is involved in this method, makes it incredibly easy to perform in any situation.

Both in the standard case of colocating users one by one, or in the more complex but better simultaneous colocation scenario, a simple button press is what is actually required by the software; we will see later in Section 3.3.2 how using hands can cause usability problems with users bound to remove their HMD at a certain point of the process: here, pressing a simple button gives user the chance to keep their HMD at all times, and this can be really scalable for any kind of game or social experience a developer might add as a corollary of the colocation software.

We must notice how not needing any of these requirements is helpful for performance reasons too: the lesser plugins are in the software, the lighter it is; we can imagine that in a big future evolution of VR where colocated experiences are more common, a non-resource consuming colocation software might be really useful in helping developers choose which calibration method to pick for their games.

Finally, we can say that precision can still be quite accurate, as we will see in Figure 3.8 in Chapter 3, with respect to the one-handed hand tracking-based calibration method: the median error between real and virtual positions, as you can see in the figure, is placed around 10-12 mm.

2.1.2. Cons

The main problem with this colocation method is that execution of the fixed-point calibration requires certain involvement on the part of users as they need to take place (or place their HMDs) at predefined locations as accurately as possible. The virtual initial positions and rotations must correspond with the HMDs position and rotation so preciseness depend on these factors. That means that both communication and coherency is needed between users and developers during the whole process: while positional miscalculations can be actually negligible, if users place their HMD with a slightly different rotational angle with respect to their virtual rotation, the small drift that is initally created, might become bigger and bigger in case a user decides to move in the room.

Moreover, this method may ask for a little extra effort by the side of developers as a correspondence between the scene structure and the physical room could be required. In case of a fully immersive experience, such as a shooter game where the map level is structurally identical to the room [51], developers must take in account where to make players stand while spawning their virtual bodies.

Another big downside is recalibration: when the positional and rotational drift becomes too big during the usage of the software, recalibrating is necessary. And if recalibration is necessary during the application run-time, users would have to remove their HMDs and reposition them into their original/initial place; consequently, a software restart may be necessary too if the software doesn't keep the association between players and their original spot throughout the level: this slows down the whole pace of recalibration.

Finally, we must mention another possible inconveniency: since the colocation process happens while HMDs are placed on the floor and not worn by users, in case of bugs it might not be so easy to understand what is going on inside the software; this could be avoided with Unity's play mode or Meta Quest's mirroring function, but it can still be annoying to perform this kind of extra step as a mandatory procedure for debugging. We will speak further on in Chapter 3, Subsection 3.2.2, why the hand-tracking based calibration will instead, on the contrary, help developers in this.

2.2. Marker-Based Calibration

In the marker-based calibration method, a marker is placed in the physical environment and needs to be tracked simultaneously by all client applications running on users' HMDs. A possible implementation of the method, brought on by [34], uses an ArUco (Augmented Reality University of Cordoba) marker placed on the floor in the tracking space as a spatial anchor for the colocated HMDs.

A Zed mini AR (Augmented Reality) camera [35] was attached to Oculus Quest to detect the ArUco marker. However, HMDs front-facing cameras could be used as well, as long as their video feed can be made accessible to the developer. Using the OpenCV (Open Source Computer Vision Library) framework [28], the position and rotation of the detected marker in camera space was used to recalculate the position and rotation of the camera in the coordinate frame associated with the marker. This marker also has a reference representation in the virtual world. This representation was the virtual anchor from which users are relocated. Opposed to the fixed-point calibration, where U_V was known as the location to be aligned with, in marker-based calibration, this pose was calculated after a marker was detected. Since the markers world space pose (e.g, M_V) is known in the virtual world, the world space location (U_V) to which the user is wanted to be aligned was determined.

The marker recognition software provides the marker location in the user's camera space. This can be inversed to determine the user's location in marker space (we define this with U_M). It is now easy to determine U_V with $U_V = M_V + U_M$. Since U_V is known, it can be possible to relocate the virtual user to this location. This calibration can be used for each user independently or simultaneously. The process is either triggered by pressing a button on a controller or from an admin computer. This way, it could be controlled and easily redone during the experiment, as long as the marker is in sight of the AR camera.

Therefore, an extra implementation step may be needed: both the camera and the marker's API's must be integrated into the code so to enable the possibility to access their functionalities; the marker's position must be easily accessable to everybody, and its information must be rapidly sent via network to the other users; more in detail, what we would send to other users is not quite the marker's position, rather than our own position with respect to it.

The best case scenario would be simultaneous calibration: at each frame, every user sends the marker's position to the other users, and everybody relocates consequently. This can be highly resource-consuming and performance could be heavily affected, but it would be undoubtly one of the most precise calibration methods at all time, since the position of the players is updated so frequently.

In Figure 2.2, on the left, we can see the users standing in the real world detecting a marker; on the right, virtual users who are relocated depending on the detected marker. The virtual user is moved by Δp and rotated by α to get to U_V, which is the user's position $\vec{p_m}$ and rotation $\vec{r_m}$ in marker space.

While computer vision frameworks are getting more precise, we can expect this method to produce better results in the future; nonetheless, until these camera functionalities will still be only available on dedicated hardware, and not integrated into the cameras of modern HMDs, users will always need to resort to extra hardware to perform this type of calibration, spending extra money and more time to implement the APIs.

Finally, we could conclude saying that this method is overall better than the fixed-point one, even with the economical and physical constraints of extra equipment needed. We might say that our priority when researching the best calibration method for colocating users is definitely the accuracy, and, as said above, we can easily affirm that having the user-marker positions sent at every frame is probably a zero-error precision method. If on the contrary, users prefer an easy setup/calibration method, and avoiding to buy extra equipment, we can instead say that the hand-tracking calibration method, which we will describe in the following Chapter 3, is definitely the best option to choose.

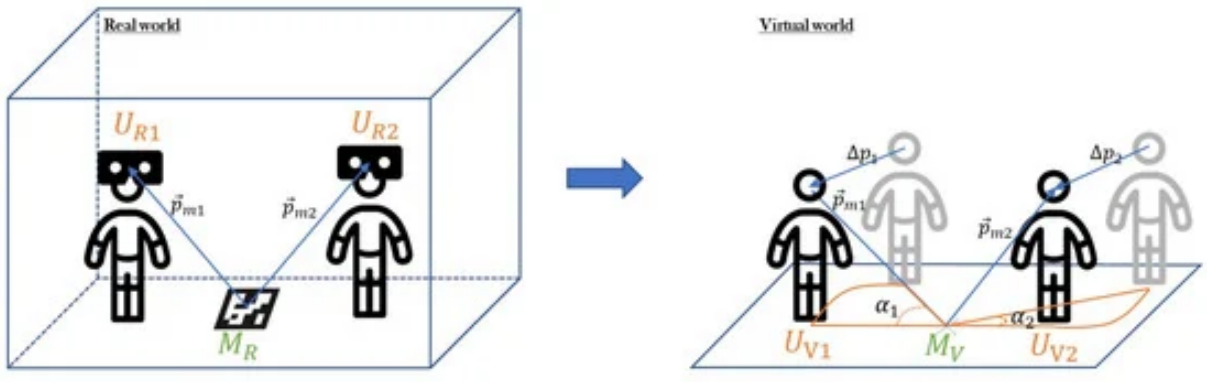

Figure 2.2: A visual representation of the marker-based calibration mechanism [34].

2.2.1. Pros

The execution does not present any difficulty for users since they would only need to position themselves in a way that allows the calibration marker to be seen in the camera image. So the HMD frontal side must be always looking at the marker without any other objects obstructing the view.

The calibration process can be made even easier if continuous tracking is used and markers are attached directly to each user. So instead of tracking a single marker in the middle of the room, users can directly track each other, sending each others positions at every frame, via network layer.

For the reasons just mentioned, using continuous tracking could potentially be the most precise method of all. Unfortunately, we do not have an accuracy test as we never explored this mechanism, but it is pretty straightforward that if at every frame, we could calculate the position of each user by tracking a marker directly attached to their HMD, we could possibly reach a 100% accuracy software, as the drift problem which usually comes out after a long usage, would never occur.

Simultaneous tracking doesn't only have the quality of accuracy in his assets: it literally cancels any type of effort needed by the side of the users; continuously tracking each other's position is a developer-side feature, that must be programmed at compile-time and therefore overrides users from taking any kind of actions such as button presses, hand pose gestures and other possible inputs.

Consequently, we could say that while this method might seem complicated for developers to set up for a ready-to-use scenario, it can potentially be really easy for users: we can't imagine a better user experience than literally wearing an HMD, loading up a software and realizing that the virtual users you see in the world are already aligned with their real-world positions as the tracking algorithms that relate cameras to markers do all the hard work by themselves.

Finally, we can conclude saying that as in the fixed-point calibration method, not relying on any type of internal procedure to colocate users, is an excellent merit as this gives users the chance to keep their HMD on their head at all times; this is comfortable and contributes to a positive usability of the software during the play session.

2.2.2. Cons

The first big con of this method is pretty straightforward: it might require additional hardware and software, depending on whether the HMD has integrated cameras that can be accessed to enable marker tracking or whether an external camera needs to be used.

As of today, access to Oculus Quest external cameras is subject to several restrictions for privacy issues: the novel Passthrough API (Application Programming Interface) [20] only allows users to modify what they see through their external cameras, and do not give you the chance to access other people's cameras. Not only this, but the use of marker tracking itself requires additional implementation.

Therefore, there is indeed a contraindication in using this type of calibration if developers aren't already aware of how to put in relationship the tracking algorithms of cameras and markers chosen as pivotal points. Another consequence of this is that putting together markers, cameras, and computer vision algorithms, can become seriously performance heavy if not optimized in the correct way; especially in the simultaneous tracking scenario.

Another consequence of the need for extra equipment, is definitely the economical side of cameras: while most computer vision frameworks are open source and available to everybody, modern cameras for advanced AI driven image recognition, such as the latest version of the zed camera mentioned in the section above, are becoming more and more expensive; it therefore would be an economic disadvantage in case users are looking for the most precise software, hence the best type of equipment.

Finally, we can conclude by saying that in case of recalibration, users may need to return to the marker, making recalibration dependant on the location of the real-world marker. In a big open spaced environment where users may get very far away from their original spawning point, it may result uncomfortable to move back to the initial position. This obviously won't be a problem in case of simultaneous tracking, as attaching markers directly to players makes recalibration a process that will actually never be needed.

3 | Hand Tracking

In this chapter, we will first describe the mechanism and the setup needed to perform colocation with the hand-tracking method, with the help of images taken both internally (screenshots from inside the software) and externally (photos of the physical room during the process). Then, we will list the formulas and discuss the calculations that allowed us to align virtual and reality world of both users. Finally, we will compare pros and cons, speaking about some of the reasons for which we believe that the hand-tracking methodology is the best between the other mentioned ones, and concluding with a couple of downsides that we noticed while developing this system.

3.1. Hand-Tracking Calibration

In the hand-tracking calibration method, the hands of one of the colocated users are used as spatial anchors: these hands will be used as a point of information sharing between the virtual worlds of the players. In our implementation, we worked with an Oculus Quest; technically one could use just one tracked hand (usually the right one) to relocate the user according to the tracked pose, but in our case, we relocate the user by tracking both hands and calculating a mean point between the two to which the user is reoriented.

In Figure 3.1, we can see a visual representation of how this method works; on the left, the users standing in the real world detecting the same hand; On the right, the user gets relocated by difference Δp. α is the difference in rotation of the tracked own hand and received reference hand. What we actually do here is rotate around the mean point between the two hands, of an amount α which we calculate with the formulas described in subsection 3.1.1. The positional change only happens after the rotation, as the mean point may or may not be wrongly rotated of 180°: this can happen because the mean rotation of a point between the hands can often be negative. After the first rotation, we calculate the conditions for an eventual extra rotation of 180° so to ensure that players are one in front of the other: only after this last step, the position of the player is moved so to match the real position as if he was behind those hands virtual position in the master

client's origin system. Keep in mind that using the mean point between hands not only solves the problem of an up-side down rotation, but greatly enhances the precision and therefore diminishes the median error, as shown in the accuracy table 3.8.

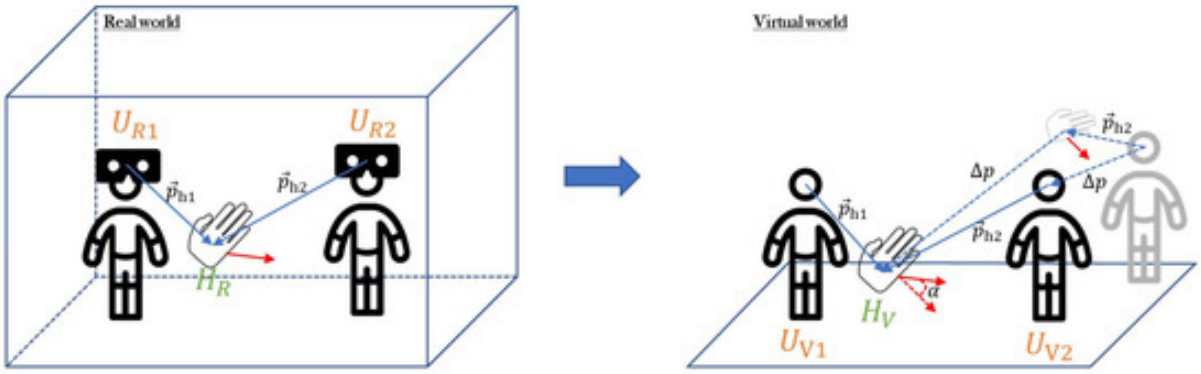

Figure 3.1: A visual representation of the hand-tracking calibration mechanism [34].

First of all, the user who wants to be colocated, must press simultaneously the A button, the Grip button and the Trigger button on their right controller. A blue capsule will appear above their head, indicating that he is the player who wants to be colocated, as shown in Figure 3.2; his ID will be sent to the master client so the admin knows who, between the several possible players in the scene, must be positioned in front of him. When the colocation is over, or if the user presses those buttons again, the blue capsule will disappear. Users not ready for colocation won't have a capsule above their head at all.

The master client (admin) cannot candidate for colocation, as his reference system will be the origin system of all the rest of the users. He will have a red capsule above his head, as shown in Figure 3.3: this will always indicate who is in charge of colocating other users. In case of disconnection, the red capsule and therefore the master client role, will pass to the oldest player (i.e, the player who was in the game before everyone else remaining) [30]. In this case, if the colocation was already done for every player, nothing will change. But if some users were not colocated yet, then the process must be restarted using the new master client as admin, as his origin system will be the new reference for the rest of the players from that moment on.

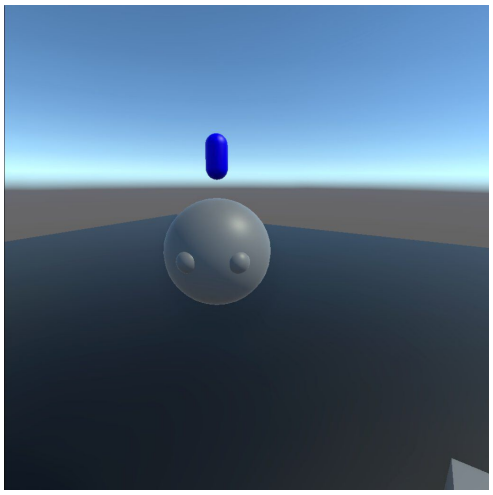

Figure 3.2: Blue capsule above the head of the player ready for colocation.

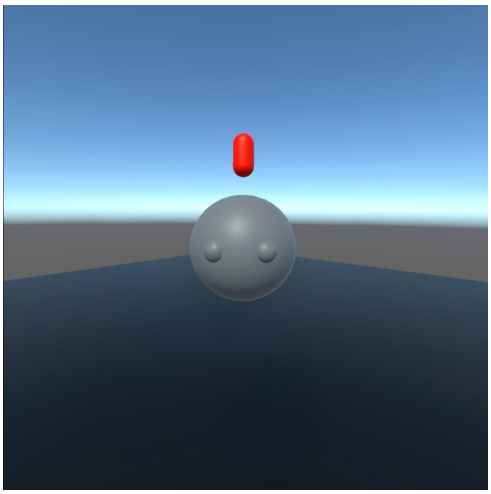

Figure 3.3: Red capsule above the Master Clients' head.

The blue-capsuled user must walk towards the master client's real position in the room, standing right in front of him, and watching directly towards their HMDs; in case of more players, the master client must communicate its real position in the room to the player to colocate. In case the master client is not aware of his role, as it may happen if the first admin disconnects, he can read his personal UI (User Interface) Canvas that follows him everywhere which not only contains information on his role, but will help him throughout the whole process with clear instructions. As a last resort, he may always ask to the other players to check the presence or not of a red capsule above his head.

During the calibration process, the blue capsule users' hands are held behind their back to prevent the headsets from accidentally tracking these hands, as shown in Figure 3.5: the admin user is then holding their hands in front of both headsets, so both track it. These hands must be still and steady: users will have a green cylinder in front of them (which represents the position and rotation of the mean point between the hands), and must try to maintain the green cylinder vertical and still during the process, as shown in Figure 3.4. The colocation must not be triggered in case one of the hands isn't fully distinguishable to one of the users, and cannot be triggered if both hands aren't entirely visible to the admin. In 3.4 you can see the green cylinder mentioned above, while in 3.5, as already said, on the right the admin shows his hands to the client on the left, which keeps his hands behind his back.

The colocation process starts by posing a number 2 with the fingers: the master client sends the position and rotation of the mean point of both tracked hands via Unity Event [10]. We can see this process happening in Figure 3.6 and in Figure 3.7. The receiving blue capsule user calculates the position and rotation of his mean point (the hands are the same, but the position and rotation are different due to the two distinct reference systems). Finally, the blue capsule user rotates around the mean point to match the first users origin system. From now on, the users are colocated. In 3.6, if the admin sees two cylinders at the same height, then that means his hands are both visible to other player and colocation can be triggered; in 3.7, we can see the client on the left keeping his hands behind his back while the admin (on the right) makes a number two with his right hand.

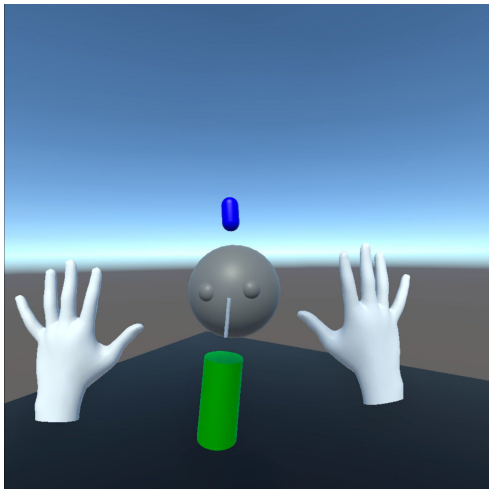

Figure 3.4: Internal view of the first phase of the colocation process.

Figure 3.5: External view of the first phase of the colocation process.

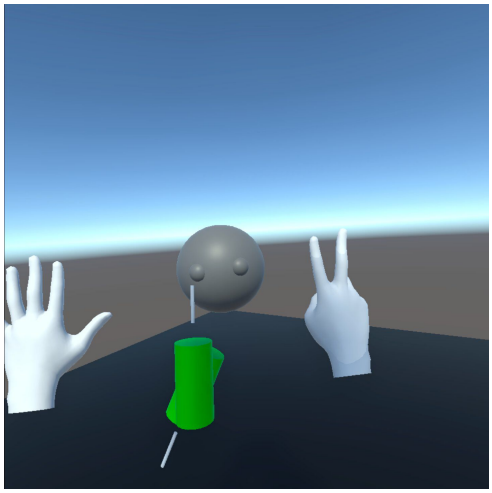

Figure 3.6: Internal view of the second phase of the colocation process.

Figure 3.7: External view of the second phase of the colocation process.

3.1.1. Calculations

In this subsection, i will describe the formulas used to calculate and apply the correct position and rotation to the user to colocate. Pedices 1 and 2 will refer respectively to player 1 (master client) and player 2 (player who wants to be colocated), while pedices l and r being respectively left and right (hands).

First we calculate the mean hand position of the master client (origin system) with the equation below (which is the extended version of the lerp function):

$$P_{M_1} = P_{r_1} + \frac{P_{l_1} - P_{r_1}}{2} \tag{3.1}$$

Unity's lerp function is then used to approximate the mean rotation between the two hands of the master client:

$$R_{M_1} = Lerp(R_{l_1}, R_{r_1}, 0.5) \tag{3.2}$$

The two mean positions and rotations are sent from the master client to the second player (the user who wants to be colocated). The user then calculates his own mean hand position with the equation below (as above, the extended version of the lerp function):

$$P_{M_2} = P_{r_2} + \frac{P_{l_2} - P_{r_2}}{2} \tag{3.3}$$

Unity's lerp function is used to approximate the mean rotation between the two hands of the user who wants to be colocated:

$$R_{M_2} = Lerp(R_{l_2}, R_{r_2}, 0.5) \tag{3.4}$$

Then we can use the calculated mean positions and rotations of both players to compute the (delta) position and rotation that we need to apply to player 2 so that his virtual transform, matches his real world transform in player's 1 origin system. For the rotation, we only take in consideration the Y component of the transform, while we set to 0 the Y component of the position as the height is automatically detected by the HMD:

$$\Delta R_{M_2} = R_{M_{1_y}} - R_{M_{2_y}} \tag{3.5}$$

$$\Delta P_{M_2} = P_{M_1} - P_{M_2} \tag{3.6}$$

$$\Delta P_{M_{2_y}} = 0 \tag{3.7}$$

Finally, we can apply the above found delta position and rotation to the position and rotation of player 2:

$$R_{2_{Final}} = R_2 + \Delta R_{M_2} \tag{3.8}$$

$$*(CONDITIONAL) : R_{2_{Final}} = R_{2_{Final}} + 180 \tag{3.9}$$

$$P_{2_{Final}} = P_2 + \Delta P_{M_2} \tag{3.10}$$

*As briefly described in Section 3.1, there are some cases in which the rotation of the client is wrongly drifted of 180° around the hands: it is as if the client gets colocated in the same exact place of the admin. This is due to the fact that the mean point between the hands, can wrongly perceive an upside down rotation as it probably takes as input a negative rotation of one the two hands around the Y axis.

Therefore, after rotating, but before positioning, an extra and final rotation of 180° around the hands is applied in case the dot product between the forward vector of the two players is positive (i.e, they are watching the same direction): we can see this in Equation 3.9.

The reason it happens before positioning, is that the rotation always goes around the mean point of the two hands; if the player's position get changed, his initial hands position will be in the wrong place; rotating around that position, sets the player far away from the admin's hands. With this extra step, we can ensure that the rotation of the cylinder around the X axis is not going to influence the final result anymore.

As mentioned above, the only requisite of the green cylinder is that it is vertical to the ground, so parallel to the Y axis: therefore his Z axis rotation remains the most important parameter to respect in order to avoid misplacements; we will speak about this better in Subsection 3.3.1.

3.1.2. Accuracy

If colocation is triggered correctly, accuracy can potentially be 100% precise: the virtual head's position and rotation often corresponded millimetrically equal to the real one.

In Figure 3.8, we can see the comparison between the different colocation methods: for each method, the calibration was performed 25 times; the error between the ground-truth distance between the HMDs and the distance derived from the calibrated positions was recorded during 1000 frames after each calibration. They then calculated the median error for each dataset of 1000 error values, obtaining four datasets of median errors with 25 entries in each.

The fixed-point calibration method showed the greatest variability among the four evaluated methods demonstrated by the largest span of median error values and their interquartile range. The hand tracking-based calibration method demonstrated the strongest consistency when two tracked hands were used in the calibration process. The increased consistency compared to the other tested methods is reflected in the much more compact span of the median error values and their interquartile range.

According to our evaluation, the greater accuracy of this method combined with the clearly better consistency and the ease of execution on the part of the users makes it the best method for calibrating two-user colocated scenarios.

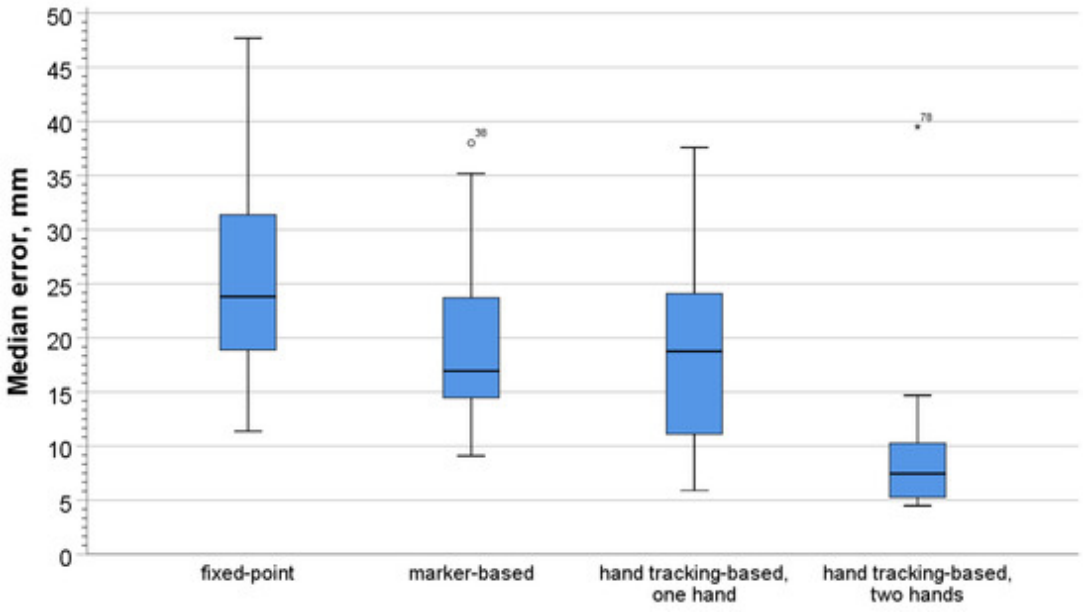

Figure 3.8: Accuracy comparison between the different methods [34].

3.2. Pros

In this section, we will describe the two main pros about this method: how easily one can setup the hardware and software to prepare for the process, and how little effort is needed in order to calibrate and recalibrate the orientation of players.

3.2.1. Ease of Setup

While it is indeed true that this method has still some hardware requirements (an HMD with integrated hand tracking is needed [26]), for preparation this method shows to be the least demanding among the above described methods: neither a physical marker nor a fixed location has to be set in the real world.

Avoiding to have the obligation to place users in predefined spots is extremely helpful as it is a procedure which can take time and space to perform correctly; in this method instead, users can start their software anywhere in the same room, without any kind of positional or rotational requirement. They may be far away from each other, facing in any direction, while they select the software from the Oculus menu and wait for Unity to load it up, and it will still never affect the colocation system as it will be a process which will trigger inside the software at run-time.

In the end, since both type of users (non-master clients and admins) will have personal instructions written out on their personal UI Canvas which updates dynamically the instructions based on the phase of the process, any type of player can decide to be a master client, booting up the software before anyone else; there are no type of restrictions on who must be the admin and who must not.

The absence of any type of marker or external equipment, is a big pro in terms of both economical comfortability and technical feasibility; at the status quo, we could say that any type of HMD still available on the market (therefore not declared obsolete), has all the tracking functionalities needed to perform this kind of operation.

3.2.2. Calibration/Recalibration Effort

Calibration is really easy due to the fact that the only real requirement is that both hands of a user need to be simultaneously visible to both users. While other type of objects used as markers may or may not be easily found and bought on the market, relying on parts of your own human body to trigger the colocation makes it virtually impossible for anyone to have any kind of difficulties.

As already decribed in the previous sections, pressing predefined buttons to candidate for colocation and standing in front of the admin with their hands behind their back [23] on a side, and simulating a pre-defined hand pose that doesn't need any type of external vision on the hands on the other side, makes it an incredibly easy and comfortable calibration method.

Depending on the HMDs tracking accuracy, the reorientation can be redone every time the headsets' drift gets too big. Since it does not require any preparation in the real world, this can be done anytime and anywhere during application; the users infact do not need to return to a specific spot in the real world: all they need to do is press the buttons again, walk towards the master client, and remake the procedure.

While we saw in previous methods that recalibration forces users to take off their HMD and place them in predefined spots, in the hand-tracking calibration method, recalibration can be performed exactly in the same way as calibration works.

Actually, we could say recalibration is even much easier: while for a first time calibration, virtual and real positions don't match and therefore users must rely on communication to find each other in the same room, for recalibration we can expect virtual users to be just slightly drifted away from their real position, so finding each other in the same room, is a much easier process to endure rather than before.

Finally, since this method entirely happens at run-time, while users are actually wearing their HMDs, users can have the possibility to actually watch and experience how each one of them gets moved and rotated during the colocation process: this can be really helpful in terms of debuggability as observing the process come together piece by piece can help developers in case of any kind of problems that may occur during the software's possible and various enhancements.

3.3. Cons

In this section, we will describe the two main cons about this method: first of all, the dependancy that the software's precision has on the green cylinder's transform, and then, the incapability of the HMD to simultaneously track both hands and controllers, which can cause problems in usability, especially since it slows down the whole process.

3.3.1. The green cylinders' transform

A big con of this calibration method is that accuracy is highly dependant on how "correctly" the hands are rotated and positioned in front of both users. The green cylinder will help out users in understanding how to position their hands in the best way to achieve the most precise colocation possible.

Tilting incorrectly the hands may lead to a faulty colocation. This will induce the software into thinking that the player's head will have a different rotation with respect to the real rotation of the head of the player looking directly towards the admin; therefore, this will lead to a slightly wrong colocation; this type of error is still acceptable as it will be a difference of only some centimetres.

The worst case scenario is if one or both hands are not entirely visible or distinguishable to one or both users: in this case, the colocation will miss completely; this is a pretty obvious and expected outcome, as if one of the hands is not visible, the Meta Quest software will perceive it as if it was on the floor, therefore calculating the mean point between the visible hand and the floor. This is the most important requisite to respect and not fulfilling it can be avoided with good communication between the two users.

We conclude nonetheless by saying that this situation just described can be easily recognized by other users, even if the hands aren't theirs, as you would see the green cylinder at a lower height with respect to the hands. Patience at the beginning of the process is required as old HMDs (especially Oculus Quest 1, who only supports the old Hand Tracking functionality [21]) always take a little while in rendering, positioning and rotating the hands correctly, as well as distinguishing them from one another.

3.3.2. Absence of simultaneous hands-controller tracking

Another big problem is the impossibility for the Oculus Quest to track simultaneously both the hands and the controllers; this is due to the fact that the Quest uses IR (Infrared) to track the controllers, while using visible light to track the hands: only one of these two tracking systems may be activated at a time.

When the controller buttons for colocation are pressed, the hands disappear, as controllers are effectively being used; the same happens if the controllers are moving: hands will still disappear as the controllers will still be perceived as being used. On the contrary, when hands are tracked, the controllers will disappear from sight. So after the client presses those buttons, he won't be able to track any kind of hands until he lays the controllers on a table, or on the floor; after the colocation is ended, while hands are still being tracked, on the contrary, he won't be able to see any controller until he picks one of them back up.

This causes a major usability problem: both the former laying and the latter picking-up of the controllers, necessitates vision of the users' surroundings; the blindness of this operation may cause the user to inadvertently step on the controllers if laid on the floor, or bump into obstacles while trying to pick up controllers which were previously laid on a table.

Users therefore are obliged to either uncomfortably remove and re-wear their HMDs during both phases, or to activate their external cameras so to see where to lay the controllers and consequently, where to pick them back up. Unfortunately, the external cameras of the Quest cannot be activated by code (as one could access other users cameras, and in case those users aren't in your same room, you could possibly observe their personal surroundings, rising several privacy issues).

The best option for users is to activate the possibility to turn on and off the cameras with a double tap on the side of the Quest; this function is easily accessable in the settings menu of the Quest and is highly suggested as it is easy, fast and comfortable to use, and it generally avoids any problems caused by the above mentioned impossibility to track the controllers at certain times.

4 | Experiments

Users can experiment the accuracy of the colocation system through the use of 3 different demos. In this chapter, we will describe each demo one by one, accompanied by pictures, both taken internally and externally to the software, and for each demo, we will discuss several accuracy tests we performed.

4.1. First Demo

Our first demo uses circles to demonstrate the software's positional accuracy of players. Every time a user presses the X and Y buttons simultaneously on their left controller, a circle gets drawn under their location at that moment. If those buttons are pressed again, the circle disappears.

In Figure 4.1, we can see the two pink circles spawning beneath the players: to draw them, we used a Line Renderer component [40] attached to a spawned object [29].

The aim of the demo is to make two normal users (or a user and the master client) swap places, by walking to the other users' circle and standing inside it. If a player is in the same real world position as the other user was before (i.e, when he generated the circle), then that means that real world and virtual world positions correspond and therefore the colocation process was successful.

In Figure 4.2 and 4.3 respectively, we can see the first and second phase of the demo: before swapping places, each player standing on top of their circles, and the aftermath, when they are standing in each other's circles.

The best results are achieved if players are looking directly in front of them: since it is not possible to track a users' feet (unless you make use of additional hardware and sensors), circles are spawned directly beneath the head of the user; therefore, if the user's head is tilted forward, then the circle might spawn slightly in front of them; circles must

be spawned while the user is keeping his head vertical and aligned to his body, and is watching straight in front of them, as we see in the mentioned 4.2/4.3.

The best way to check the preciseness of this demo is to place some real-life objects just beneath every users' feet as soon as the circles are spawned, and use them as markers. Then, users may temporarily take off their HMDs (or activate their passthroughs [20]) so to walk towards the real life objects of the other users: when they put back their HMDs, if they are exactly on the virtual circles spawned by the other users, then the colocation is correct.

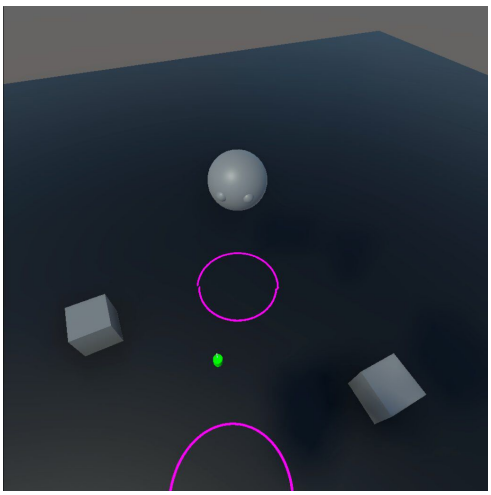

Figure 4.1: The two drawn circles beneath the players' heads.

Figure 4.2: First phase of the first demo.

Figure 4.3: Second phase of the first demo.

4.1.1. Accuracy

The tests results were extremely good when circles spawned exactly under the users. As said above, since there exists no leg/body tracking in VR without external hardware, the circles spawn under the head of the player (which follows the HMD real world position). If a player keeps his head vertical to his body while spawning the circles, the circles will effectively spawn right beneath them.

In Table 4.1 below, we can observe the measurements errors in six different tests for demo one. The numbers in millimetres represent the distance between the two players: P_1 -> P_2 means how "far" was Player 1 from Player 2 original position, and viceversa P_2 -> P_1, means how "far" was Player 2 from Player 1 original position. The measurements are taken from the player's (feet) position to the center of the other user's circle; the two factors that make this test precise is how users' heads are positioned while spawning the circle and how precisely the real-world object/marker is actually placed beneath their feet/at the center of their personal circle.

	Test One	**Test Two**	**Test Three**	**Test Four**	**Test Five**	**Test Six**
P_1 -> P_2	70mm	10mm	50mm	13mm	80mm	20mm
P_2 -> P_1	65mm	10mm	40mm	11mm	90mm	19mm

Table 4.1: Measurements errors in six different tests for the first demo.

Swapping places with more than two users demonstrated that two different (non-master) clients could find themselves in slightly different positions, but nethertheless, still keeping a really small measurement error, depending on how precisely the circles were spawned under them. Users must keep track of each other's markers in the real world so to not confuse them when players are more than two.

4.2. Second Demo

Our second demo uses crosses to demonstrate the software's positional accuracy of the mean point between two players. Every time a user (non-master client) presses the Grip and Trigger buttons simultaneously on their left controller, a cross gets drawn with a Line Renderer component attached [40]; the cross is drawn at the median point between that player and the master client. If those buttons are pressed again, the cross disappears.

You can see this pink cross in Figure 4.5; in our case, the cross is actually composed of four different segments [4] that go on and forth between the four angles and the center.

The aim of the demo is to make the user and admin meet at halfway, where the cross is drawn. If the two players find themselves in the same position, nearly bumping into each other in real life, then that means that real world and virtual world positions correspond and therefore the colocation process was successful: you can see this happening in Figure 4.5, where the two players are meeting at halfway in the room.

Similiarly to before, since we evaluate the position of the heads, the best mid point for the X is found if both users are looking straight in front of them. Ideally a third user should place a real-life object onto the real-life midpoint between the two players: then the two players can walk to each other and find themselves onto the marker to demonstrate that the virtual and real world mid-point correspond.

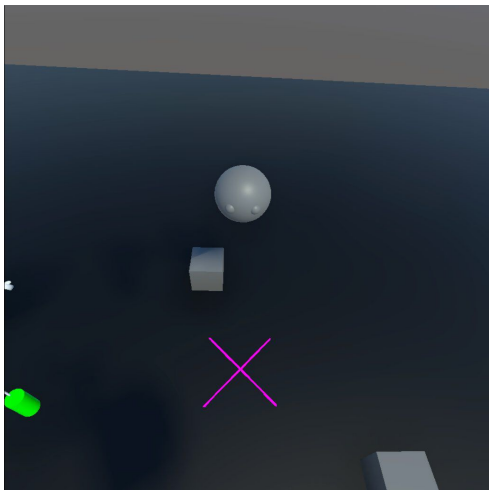

Figure 4.4: Internal view of the pink spawned cross.

Figure 4.5: External view of the two players meeting halfway.

4.2.1. Accuracy

As said before, the X spawns in the median point between the two heads. So for maximum precision, the users heads must be positioned coherently to one another: if one user keeps his head vertical to its body, the other user should do the same; if one user tilts his head forward while spawning the X, the other user should do the same.

In Table 4.2 below, we can see the comparison between distances to the X in six different tests for demo two. The number in millimetres represent the distance between a player and the X: P_1 -> X means the distance between Player 1 and the X, and, coherently, P_2 -> X means the distance between Player 2 and the X. More precisely, the distance between a player and the X means exactly the distance between its feet, and the center of the X. Since it is impossible for two people to stand in the exact same spot, measurements were taken while they were standing at the borders of the X. The more these two values are close to each other, the more it means that the X was effectively the median point between the user and the master client, and therefore confirms the preciseness of the colocation software.

	Test One	**Test Two**	**Test Three**	**Test Four**	**Test Five**	**Test Six**
P_1 -> X	20mm	19mm	8mm	15mm	17mm	22mm
P_2 -> X	15mm	18mm	8mm	14mm	17mm	21mm

Table 4.2: Comparison between distances in six different tests for the second demo.

This test can still be performed with more than two users, but only non-master client can make the X spawn, and the tests would still be measured between each client and the master client, and not between non-master client users. So this test cannot be performed simultaneously by more users, but must be done one at a time between the admin and each client; a third user can still be helpful by positioning a real-life object at the midpoint between the client and the admin and/or ensuring that the two players won't bump onto each other by communicating their position in real-life.

4.3. Third Demo

Our third and last demo, uses boxes that can be grabbed and passed [7] to each other. This demo is not triggerable by any button, instead, it consists of a series of boxes already present in the scene and positioned on the floor. Once colocated, users can experiment both the precision of the colocation software (by experimenting the precision of their hands in this case) and the hand tracking functionality by grabbing and passing the boxes to one another.

In Figure 4.6 and in Figure 4.7 respectively, we can see the internal and external view of the two players passing boxes to each others. As you can see, to grab the box it is sufficient to place the right controller to the side or into the box and press the Grip button. While you keep the button pressed, you keep the box gripped; as soon as you release the button, you release the grab of the box. You can also see how the real-life right hands of the two players are very near to one another: the distances between the two controllers will be the object of the accuracy tests displayed in Table 4.3.

If users manage to correctly and coherently grab, pass or throw/catch the boxes, then that means that users see the boxes in the same virtual spot. Therefore, the two reference systems coincide and the colocation process was successful. Keep in mind that automatic interpolation is not very good in Photon [11] and/or Unity [37], so you may see the box stutter while under the control of other players. Instead, while the box is under your control and ownership [31], you can see it move perfectly throughout the frames.

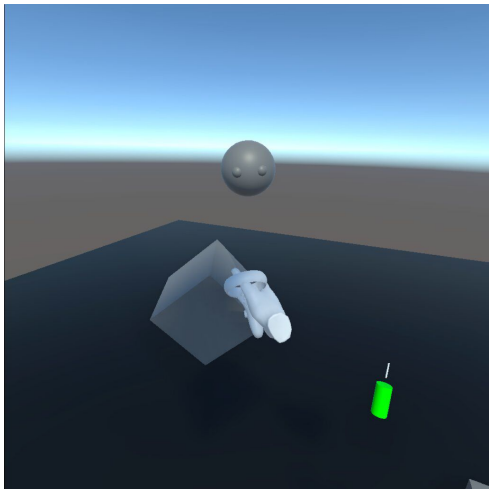

Figure 4.6: Internal view of the third demo.

Figure 4.7: External view of the third demo.

4.3.1. Accuracy

This demo was the most difficult demo in which to test errors/measurements, as it is not a positional demo. As long as the colocation is averagely correct, players would still see the boxes in the same place and therefore easily grab and pass them to the other player; precision in this case actually depends on how big or small the cube collider is, and therefore how much close you have to get to actually manage to grab it.

In Table 4.3, we can see the distances between the two controllers (Oculus Touch) while passing the boxes. The distance is measured in millimetres and is taken between the upper edge of the controllers: as grabbing the edge of the box was enough to take control of it, we decided that taking the center of the controllers as a reference would not have been correct and coherent to the measurement system; it may seem obvious that for better results and error measurement, the boxes should be as small as possible; nethertheless, the boxes were of a sufficient dimension (i.e, as big as normal human head) so to easily favour the grabbing.

	Test One	**Test Two**	**Test Three**	**Test Four**	**Test Five**	**Test Six**
Distance	10mm	9mm	3mm	7mm	1mm	5mm

Table 4.3: Distances between the two controllers (Oculus Touch) in the third demo.

Since what we actually managed to measure here was the distance between the two right controllers engaged in the box passing, it helped us understand how much offset was actually present between the position of the box in the reference system of one player and the reference system of the other player: the smaller the number, the better the colocation.

As in demo one, every user can engage with each other in this demo, by passing these boxes from hand to hand with any other user (both non-master and master clients); while the precision between two different authorative clients was usually extremely high, when two different non-master clients tried engaging in the box passing, the offset between the boxes was slightly bigger than before.

5 | Conclusions

In conclusion we will first speak about which changes an average user can bring to his own taste/usability, and then we will finalize the thesis by speaking about future developments.

Regarding gestures, users have three choices: first of all, they could obviously simply use our predefined gesture; another option available to them, is creating their own, by pressing the spacebar button while hand posing with the right hand in play mode [46]; this will copy on the Unity editor, during play mode, the transform positions of all of the finger bones with the respect to the base of the hand. Since these coordinates will disappear as soon as play mode is interrupted, users must copy them and finally paste them into the inspector, outside of play mode, in an ready-to-use array which will be under the Gesture Detector script component. Our personal script reads the transforms of the tracked hand finger bones and compares them with the above mentioned pre-saved arrays: if the distance between the finger bones and the base of the hand are below a customizable threshold, then the gesture is recognized as such and a customizable function is triggered.

Finally, the third option available for users is that they could directly create their own Gesture Detector (the most recent Oculus SDK [36] brings a whole new kind of gesture detection, easier to implement (as no code is required) and more precise) and simply program the detector to trigger the CoLocate() function inside the Co-Location Synchronizer script when the gesture is made.

The above mentioned Oculus SDK gives users the chance to customize for each hand, in the Unity inspector, which finger bones to keep closed, semi-open or open, with different kind of orientations with the respect to the ground, and how to easily trigger a function when such criterias are met. This greatly enhances the gesture detector precision and not only allows the software to avoid reading a wrong pose, but gives a great performance edge as the software must not compare potentially big transform arrays with each finger bone anymore to understand which hand pose has just been performed.

Regarding colocation, users have two choices, and the first path is probably the less intuitive: they can add the first level of their game to the main scene of the software. Therefore, they can teleport/load a new scene, ensuring to keep the relative positions. This may not always be the case, so players must autonomously implement the players spawning in the same spots, in between scenes. Group teleportation [45] is becoming a frequently used functionality and it could help in this scenario.

The second and best path is to simply create your own player and/or Network Manager, being sure to create a public head variable of the player that obviously refers to its head (or a simple sphere). This head object will be needed to ensure colocation gets done correctly, and most importantly, the player's head/sphere must be programmed to follow the OVRCameraRig game object at all times. Users can then simply attach both the Co-Location Synchronizer script and the Gesture Detector script to their players.

In Figure 5.1, the Unity hierarchy of the OVRCameraRig [32]. The blue prefab icon means the prefabs were added afterwards. The OVRControllerPreab is simply a prefab with the shape of a controller, while the CustomHandLeft/CustomHandRight are 2 prefabs representing a fake hand holding onto the controller and pressing buttons. The real hand prefabs, fundamental for hand-tracking, are the OVRLeftHandPrefab and the OVRRightHandPrefab. Their simple presence as a child respectively of the LeftHandAnchor and the RightHandAnchor enables the Oculus to track and recognize hands bone movement.

In Figure 5.2, the Unity hierarchy of the Network Player. The Head, Right and Left eye are simply used to track the virtual position of the player. The Mean Hand game object is the green cylinder (with an Up vector indicator object as a child). Finally, we can see the PlayerStateCapsule (which can be red, blue or disabled) and the personal UI Canvas with the instructions and steps to take for each user based on its role.

After a couple of months of research and API studying, I started developing the code and it took me about a month to finish. First of all, Oculus SDK is in continuous improvement, therefore names, scripts, prefabs and objects are subject to change within a year of difference: this lead to several difficulties in understanding which were the best ready-to-use kits to use; furthermore, following tutorials prior to one or two years ago was useless due to their obsolete nature.

After the best prefabs were selected to use, and the software started taking place, some problems occurred in understanding how the VR Camera Rig worked (OVRCameraRig in my case [9]). This prefab contains the mandatory OVRManager script, which is a singleton and therefore can exist only on local players. It would have been much easier if every remote player could have their own Rig.

Finally, the last obstacle before the end of the software was the fact that the hand tracking functionality of the Oculus Quest, cannot distinguish between right and left hands if these hands are placed with their palms directed towards the external cameras (i.e, they are someone else's hands). I solved this problem by using a mean point between the hands as a reference point, which in the end, was the better choice in accuracy too.

Figure 5.1: The OVRCameraRig Unity prefab hierarchy.

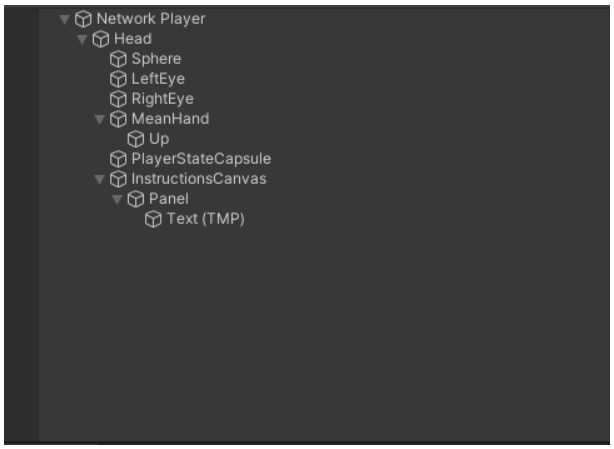

Figure 5.2: The Network Player Unity prefab hierarchy.

5.1. Future Developments

The software is ready to use; nonetheless, there could be some possible modifications/additions to it, both aesthetic and functional: a better and more realistic head and body may be added to players [47], together with a voice recognition to the player's head for lip sync [33]. Adding a skybox to the level, background music and name to players are still some aesthetic additions that could be made to make the software more comprehensible.

Creating a group teleportation system in order to give players the chance to move together in the level can be a really great functional improvement to the software so to help users mantain the same reference system as the post co-location system [45]. The software may be used to creating a unique game (e.g, a FPS game in VR), to be added or swapped with the already existing demos, which exploits the software and helps the user understand the huge potential of VR colocation [13][25]. Finally, while the Hand Tracking functionalities improve with new Oculus Quest, we can expect the improvement of the general precision of the system too, so to minimize the amount of attempts necessary to colocate users perfectly.

Using hands as a point of common information sharing between users in different reference system, has proved to be highly efficient and effective. Not only due to the reasons mentioned in the dedicated section above, but due to its easy preparation and prerequisites: all you need, other than your HMD, is your hands.

While it is true that we could consider a Meta Quest as an "external hardware with hand tracking capabilities", we can't ignore the fact that at the status quo, year 2022, most of the available HMD on the market, already have hand tracking capabilities included in the HMDs software. Not only this, but we must mention the fact that the software was developed on Oculus Quest 1, where hand tracking was still at its early stages.

Now, with Hand Tracking 2.0 [22], available only from Oculus Quest 2 onwards, small precision mistakes and bugs regarding hand tracking are nearly vanished and users' hands are virtually replicated in an extremely precise way: users can overlap their hands and the Quest 2 sensors are still able to track both hands and replicate fingers correctly. We can see this in figures 5.3 and 5.4; both are taken from showcase videos [18][19] made by Meta Quest.

In Figure 5.3, you can see how the main features allows the HMDs cameras, thanks to machine learning processes, to distinguish between two very close or even overlapping hands; now, several manual activities that employs both hands next to each other, can be simulated.

In Figure 5.4, there is an example of above said new manual activities: you can manipulate complex objects and use them to interact in the space; with the previous hand tracking software, you couldn't pass an object in between fingers of different hands; now, the software track hands so precisely you can literally play with small objects through your fingers.

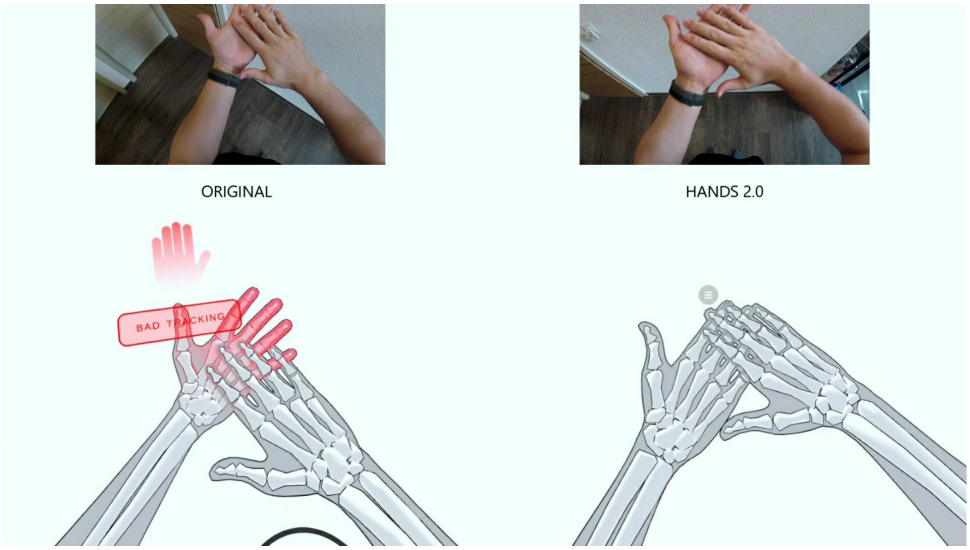

Figure 5.3: Comparison between overlapping hands recognition, Meta Quest, 2022

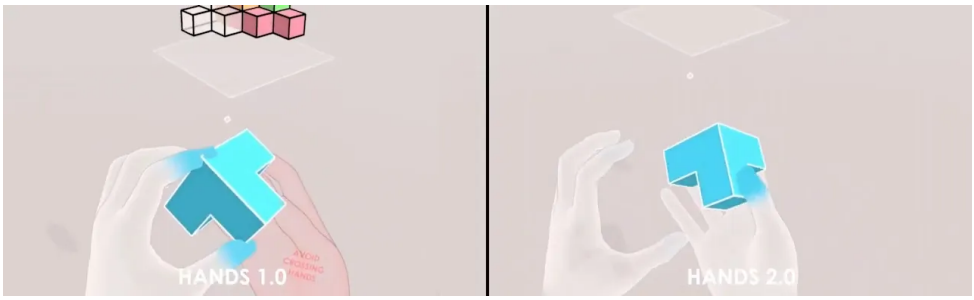

Figure 5.4: Comparison between close finger manipulation, Meta Quest, 2022

To finish off, we must say that finally, in October 2022, Meta Quest officially announced that their newest Meta Quest Pro headset, is getting support for colocation. Based on the description of the technology [8], it likely works similarly to the colocation features already present in Apple's ARKit and Google's ARCore.

Inside-out tracking generates a point cloud of static features in the room. Algorithms find shared unique patterns in the point clouds of multiple devices and use them to align the virtual spaces. The process requires no external sensors, base stations, or specific markers.

To be clear, this is a developer-side feature that will require apps to specifically support it. It's called Shared Spatial Anchors, an extension of the existing Spatial Anchors feature that lets you place virtual content in a specific position in your room so it stays there the next time you use the app.

This technology is still at its early stages of development, and therefore its not still a ready-to-use software, as it is in our case. But we can say that if the colocation algorithms will prove to be extremely precise in recognizing common patterns of the physical room shared by two or more players, we can easily predict that this might be the best option in terms of accuracy.

Since this will be an internal feature, which will probably use the Meta Quest enhanced Passthrough functionality, usability will be a great positive trait too; as a matter of fact, this kind of process looks definitely faster, comfortable and easier to use than the ones we mentioned in this thesis.

Further details are yet to be available, but knowing that this feature will only be available on the newest Meta Quest Pro, our colocation method might still be the best option for the average user in terms of accessibility: indeed, the newest Quest has a really prohibitive cost, and will not be available to everybody during the first years of its presence on the market.

On the contrary, our software can be deployed and used by any Meta Quest, and its algorithms are actually ready to be used on any HMD available on the market too; therefore, if this feature will ever be available on older and cheaper VR headsets model, until that moment, users should prefer relying on a much more scalable software such as the one we developed.

Bibliography

[1] M. Adventures. Shared space vr in the oculus quest. 2019. URL https://www.youtube.com/watch?v=ahA11GqymgE&t=517s.

[2] AnotheReality. God of boxes, 2022. URL https://www.anothereality.io/portfolio_page/god-of-boxes/?lang=it.

[3] AnotheReality. God of boxes, 2022. URL https://www.youtube.com/watch?v=JSOXWGeQm54.

[4] ardy3344. How to draw a cross using line renderer, 2011. URL https://answers.unity.com/questions/151518/can-you-help-me.html.

[5] A. B. Oculus quest shared space demo at oculus connect. 2019. URL https://www.youtube.com/watch?v=ALQypvxoJ8s.

[6] A. B. Shared space oculus quest with hand tracking and geolocated avatars. 2020. URL https://www.youtube.com/watch?v=rn2Nk0JSus0.

[7] CircuitStream. Unity programming guide to picking up and grabbing objects with oculus rift, 2019. URL https://circuitstream.com/blog/grab-oculus/.

[8] V. T. . Clips. Quest pro shared-space multiplayer mixed reality demo, 2022. URL https://www.youtube.com/watch?v=ZTWjql4_1Do.

[9] C. Doc. C# class ovrmanager, 2018. URL https://csharpdoc.hotexamples.com/it/class/-/OVRManager.

[10] P. Engine. Rpcs and raiseevent, 2022. URL https://doc.photonengine.com/en-us/pun/current/gameplay/rpcsandraiseevent.

[11] P. Engine. Introduction, 2022. URL https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro.

[12] U. Frese. A slam overview from a users perspective. 2019. URL https://www.researchgate.net/publication/220633576_A_SLAM_overview_from_a_users_perspective.

[13] B. Gaming. Space pirate trainer dx arena mode gameplay quest 2. 2022. URL https://www.youtube.com/watch?v=TLqBf3GlNp0&t=1244s.

[14] T. GRID. Free roaming shared space hyper reality with the oculus quest - part 1. 2019. URL https://www.youtube.com/watch?v=ni8vFZEhoBs.

[15] T. GRID. Vr arena oculus quest shared space - part 2. 2019. URL https://www.youtube.com/watch?v=RePUk-gqreA.

[16] IHMC-Research. Ihmc's oculus quest shared space vr environment. 2019. URL https://www.youtube.com/watch?v=9nEa21XNSik&t=36s.

[17] Jetbrains. Rider, 2021. URL https://www.jetbrains.com/rider/.

[18] Meta. Hand physics lab comparison, 2022. URL https://www.youtube.com/watch?v=8oMukwt7VDs&t=4s.

[19] Meta. Cubism comparison, 2022. URL https://www.youtube.com/watch?v=NzBOoSmn20Q&t=2s.

[20] MetaQuest. Passthrough api overview, 2019. URL https://developer.oculus.com/documentation/unity/unity-passthrough.

[21] MetaQuest. Quest one: Vr aio, 2019. URL https://www.oculus.com/quest/refurbished/?locale=it_IT.

[22] M. News. Meta's new quest 2 demo shows the magic of hand tracking, 2022. URL https://mixed-news.com/en/metas-new-quest-2-demo-shows-the-magic-of-hand-tracking/.

[23] Nic and Amelie. Co-local vr synchronizer - a unity plug-in for shared space vr. 2021. URL https://www.youtube.com/watch?v=e08_Elg0-H0.

[24] nolo design. Oculus quest multiplayer arena scale vr-experience. 2021. URL https://www.youtube.com/watch?v=R3aymJAzftk.

[25] V. R. Oasis. Space pirate arena is epic on oculus quest 2 - space pirate trainer dx. 2021. URL https://www.youtube.com/watch?v=OOkODclp00M&t=245s.

[26] Oculus. Oculus hand tracking, 2022. URL https://www.youtube.com/watch?v=2VkO-Kc3vks.

[27] Oculus. Oculus and unity integration, 2022. URL https://developer.oculus.com/documentation/unity/unity-gs-overview/.

[28] OpenCV. Detection of aruco markers, 2019. URL https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html.

[29] L. V. D. Ouweland. Use unity's linerenderer to draw a circle on a gameobject., 2022. URL https://www.loekvandenouweland.com/content/use-linerenderer-in-unity-to-draw-a-circle.html.

[30] Photon. Master client and host migration, 2019. URL https://doc.photonengine.com/en-us/pun/v2/gameplay/hostmigration.

[31] Photon. Ownership and control, 2022. URL https://doc.photonengine.com/en-us/pun/v2/gameplay/ownershipandcontrol.

[32] M. Quest. Add camera rig using ovrcamerarig, 2018. URL https://developer.oculus.com/documentation/unity/unity-add-camera-rig/?locale=it_IT.

[33] M. Quest. Meta lipsync for unity development, 2018. URL https://developer.oculus.com/documentation/unity/audio-ovrlipsync-unity.

[34] D. Reimer. Colocation for slam-tracked vr headsets with hand tracking. 2021. URL https://www.mdpi.com/2073-431X/10/5/58.

[35] StereoLabs. Zed camera, 2019. URL https://www.stereolabs.com/zed-2.

[36] V. Tutorials. Hand pose detection with oculus interaction sdk, 2019. URL https://www.youtube.com/watch?v=lxHI2l8dda4.

[37] UnityTechnologies. Unity lts releases, 2020. URL https://unity.com/releases/release-overview.

[38] UnityTechnologies. Input system, 2022. URL https://docs.unity3d.com/Packages/com.unity.inputsystem@1.4/manual/index.html.

[39] UnityTechnologies. Input action assets, 2022. URL https://docs.unity.cn/Packages/com.unity.inputsystem@1.1/manual/ActionAssets.html.

[40] UnityTechnologies. Line renderer, 2022. URL https://docs.unity3d.com/Manual/class-LineRenderer.html.

[41] UnityTechnologies. Quaternions, 2022. URL https://docs.unity3d.com/ScriptReference/Quaternion.html.

[42] UnityTechnologies. Vector3, 2022. URL https://docs.unity3d.com/ScriptReference/Vector3.html.

[43] UploadVR. Oculus quest arena 'colocation' with dead and buried. 2019. URL https://www.youtube.com/watch?v=QJXpHp_iQF4.

[44] UploadVR. Space pirate trainer dx arena gameplay. 2021. URL https://www.youtube.com/watch?v=2Z40j6LOsrc.

[45] Valem. How to teleport in vr - oculus unity tutorial, 2018. URL https://www.youtube.com/watch?v=r1kF0PhwQ8E.

[46] Valem. Hand tracking gesture detection - unity oculus quest tutorial, 2019. URL https://www.youtube.com/watch?v=lBzwUKQ3tbw.

[47] Valem. How to make a body in vr - part 1, 2019. URL https://www.youtube.com/watch?v=tBYl-aSxUe0.

[48] Valem. Hand tracking gesture detection - unity oculus quest tutorial. 2019. URL https://www.youtube.com/watch?v=lBzwUKQ3tbw.

[49] Valem. How to make a vr game in 2021 - new input system and openxr support. 2019. URL https://www.youtube.com/watch?v=u6Rlr2021vw.

[50] Valem. How to make a vr multiplayer game - part 1. 2020. URL https://www.youtube.com/watch?v=KHWuTBmT1oI&t=1615s.

[51] T. Vector. Oculus quest "shared space" demo. 2019. URL https://www.youtube.com/watch?v=msbTbfep_sY&t=154s.

[52] T. Vector. Oculus quest "shared space" demo (alternate rendering). 2019. URL https://www.youtube.com/watch?v=XlT_xd1andE&t=117s.

List of Figures

List of Tables

Appendix A: Glossary - Acronyms

Acronym	Description
VR	Virtual Reality
FPS	First Person Shooter
HMD	Head Mounted Display
SLAM	Simultaneous Localization and Mapping
DOF	Degrees Of Freedom
ArUco	Augmented Reality University of Cordoba
OpenCV	Open Source Computer Vision
API	Application Programming Interface
SDK	Software Development Kit
IR	Infrared
UI	User Interface
Lerp	Linear Interpolation
OVR	Oculus Virtual Reality

List of Symbols

Variable	Description
$\boldsymbol{U_V}$	Virtual world position of a player
$\boldsymbol{U_R}$	Real world position of a player
$\boldsymbol{\Delta d_U}$	Distance between two players in the virtual world
$\boldsymbol{\Delta P}$	Offset between the inital and final position of the player
$\boldsymbol{\alpha}$	Amount to rotate the user in marker-based calibration
$\vec{\boldsymbol{p}}_{\mathrm{m}}$	User's position in marker space
$\vec{\boldsymbol{r}}_{\mathrm{m}}$	User's rotation in marker space
$\boldsymbol{M_V}$	Markers position in world space
$\boldsymbol{U_M}$	Markers position in the user's camera space
$\boldsymbol{\alpha}$	Difference in rotation between tracked own hand and received reference hand
$\boldsymbol{P_{M_X}}$	Mean Hand Position of player X
$\boldsymbol{R_{M_X}}$	Mean Hand Rotation player X
$\boldsymbol{P_{l_X}}$	Position of the left hand of player X
$\boldsymbol{P_{r_X}}$	Position of the right hand of player X
$\boldsymbol{R_{l_X}}$	Rotation of the left hand of player X
$\boldsymbol{R_{r_X}}$	Rotation of the right hand of player X
$\boldsymbol{Lerp}$	Linear Interpolation Function
$\boldsymbol{\Delta R_{M_X}}$	Amount to rotate the user X
$\boldsymbol{\Delta P_{M_X}}$	Amount to move the user X
$\boldsymbol{\Delta R_{M_{X_y}}}$	Rotation of the Mean Hand of player X around y axis
$\boldsymbol{\Delta P_{M_{X_y}}}$	Amount to move the User X on the y axis
$\boldsymbol{R_{X_{Final}}}$	Final rotation of player X
$\boldsymbol{P_{X_{Final}}}$	Final position of player X
$\boldsymbol{P_X}$	Player X

Acknowledgments

I wish all the best to:

- Prof. Lanzi, for being an inspiration in my road to becoming a Game Developer thanks to his Videogame Design and Programming course, and, furthermore, for giving me the chance to develop this master thesis and the right suggestions for my future.
- AnotheReality, in particular Fabio Mosca, for lending me an Oculus Quest and for giving me the suggestion to develop a colocation system in VR. Special thanks to Riccardo Verza, Elia Russo and Arrigo Berton, for helping me throughout the thesis with technical support and for letting me test the software in their office.
- Dennis Reimer and his team, and then Nicolai Krapohl and Amelie Hetterich, not only for publishing papers and videos that gave a start to my thesis, but mainly for their patience in helping me throughout some technical problems I encountered during the development of the software.
- All of my family, my parents and my brothers, especially my mother Fausta, for the support I received throughout the years and the ambition that they helped me set into my mind in order to achieve all the tasks necessary to get where I am today.
- All of the friends I met in Milan in these years, especially my former housemate Gabriele and my latter Giovanni, for bearing with my problems and my university-related tantrums, and for giving me always inspiration to do better.
- All of the colleagues I met at Politecnico, with whom I experienced both failed and successful exams. Special thanks and mentions are definitely for Davide Cocco (aka Unzo), who played the role of the hard carry in my university life in several ways, and who always strived for the best of my computer engineering career.
- My videogame programming partners and (hopefully) future working colleagues Roberto, Diego, Federico and especially Marco, for his incredibly brilliant, positive and altruistic mindset, that helped me out in several moments, including this thesis.