Tracking and Communication of Devices in Shared Augmented Reality Experiences

Michael Staudinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2018

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 24, 2018

Michael Staudinger

Contents

Preface

This thesis was written for the successful graduation with the master's degree in Science in Engineering within the course Interactive Media at the University of Applied Sciences Upper Austria in Hagenberg im Mühlkreis. The research, development for a use-case, as well as the writing started in October 2017 and ended with the submission in September 2018.

I would like to thank my supervisor, Jeremiah Diephuis BA MA, for guidance and support during the process and in general the professors of the University of Applied Sciences Upper Austria at Campus Hagenberg im Mühlkreis, which led me through my studies and shared important knowledge. Further thanks go to my friends and family, which kept me motivated for going this far in my education.

I hope you enjoy your reading.

Hagenberg, September 24, 2018

Michael Staudinger, BSc

Abstract

Augmented Reality which allows extending the real surroundings with virtual objects was improved a lot in the last few years. Everyday devices are already able to interpret real objects using sensors, visual data and Computer Vision algorithms. Other embedded sensors such as gyroscopes and accelerometers are used to track the devices in 3D space, allowing to merge the data of the surroundings with virtual data creating the illusion of legitimate holograms.

Shared Experiences which offer multiple users at the same time and same space or in different parts of the world to experience Augmented Reality applications together, are supported by most developers of Augmented Reality systems but limited to users of the same platform. This makes, for instance, the use and supervision of the *Microsoft HoloLens* (wearable smart glasses with a superior experience) very expensive since multiple *HoloLenses* are required. The creation of a system which synchronizes the status of a holographic application in space and time platform-independently would allow a more cost-effective way to supervise users of the *HoloLens* and furthermore support cross-platform Shared Experiences.

This thesis should describe the creation process of a cross-platform Shared AR Experience using the *Microsoft HoloLens* and *ARCore*, describe and discuss the methods used and evaluate a test-case for the system, giving more insight in the possibilities of closing the gaps between different Augmented Reality solutions.

Kurzfassung

Augmented Reality erweitert die reale Umgebung mit virtuellen Objekten. Durch die Forschung der letzten Jahre ist es bereits möglich auf tagtäglichen Geräten Objekte durch Sensoren, visuelle Daten und Computer Vision Algorithmen zu erkennen. Weitere Sensoren wie Gyroskope und Beschleunigungssensoren werden verwendet um das Gerät im drei-dimensionalen Raum zu tracken, das es in Verbindung mit den virtuellen Daten der Umgebung möglich macht Hologramme realistisch darzustellen.

Shared Experiences erlauben es mehreren Personen mit mehreren Geräten zur selben Zeit und am selben Ort oder an verschiedenen Orten der Welt Augmented Reality Anwendungen gemeinsam zu erleben. Shared Experiences sind mit den meisten Systemen bereits möglich, aber auf die Plattform limitiert. Dies macht eine Verwendung der *Microsoft HoloLens* (tragbare Smart-Brille) sehr kostspielig, da mehrere *HoloLenses* benötigt sind. Die Erstellung eines Systems, das den Status einer AR Anwendung in Raum und Zeit plattform-unabhängig synchronisieren könnte, würde eine kosten-effizientere Lösung für die Unterstützung eines *HoloLens*-Benutzers und weiters Cross-Platform Shared Experiences ermöglichen.

Diese Arbeit soll sich mit der Erstellung eines Systems für Cross-Platform Shared Experiences in Augmented Reality für die *Microsoft HoloLens* und *ARCore* befassen und eine Test-Anwendung für das System evaluieren.

Chapter 1

Introduction

Augmented Reality (AR) and its new way of presenting information is getting more accessible and useable every day. Especially smartphone AR systems like *ARCore* by *Google* and *ARKit* by *Apple* are already available on all new *Android* and *Apple* devices. Not only are the costs of such devices decreasing, but furthermore the companies invest a lot of research into their systems to make them more stable and more powerful. Not only is the access to a supporting device probably granted, but the community has already created hundreds of *ARCore* and *ARKit* applications for a very promising future of Augmented Reality in our pockets.

These smartphone systems offer device tracking and therefore a three-dimensional AR experience in which the users can run around freely, only limited by real boundaries in their environment, but the systems lack real interactions with objects of the real world. Object recognition is barely possible, even with more advanced mapping techniques as for instance the *Microsoft HoloLens* uses. Still, *ARCore* and *ARKit* can identify flat planes such as the floor and walls, sometimes even the shape of a staircase, which can be used for surface related placement.

Furthermore, many AR frameworks including the smartphone's can detect for instance printed images in the three-dimensional space if they have enough information about them given as a database. So not only flat surfaces can be referenced, but using the *image recognition* the position can be defined more precisely and – most importantly – controlled by the user by simply moving cards with tracking markers around.

Google's ARCore and *Apple's ARKit* offer many features that can be already used by developers to already build applications for augmenting the real surroundings, even though cross-references between *reality-virtuality* are barely possible now. The systems offer as well platform-dependent networking features which allow synchronized content across the globe.

In addition to these systems, the *Microsoft HoloLens* – head-mounted AR smart glasses – are becoming more popular in exhibitions because of its far superior reality mapping which is not just limited by flat surfaces, but even more complicated shapes are at least roughly detectable and understandable. Companies like *CurvSurf* are already quite successful in detecting desired objects with their software *Find Surface* which opens up more reality-specific tasks the *HoloLens* could assist at (see figure 1.1).

Creating prototypes or full applications specialized for the *Microsoft HoloLens* are

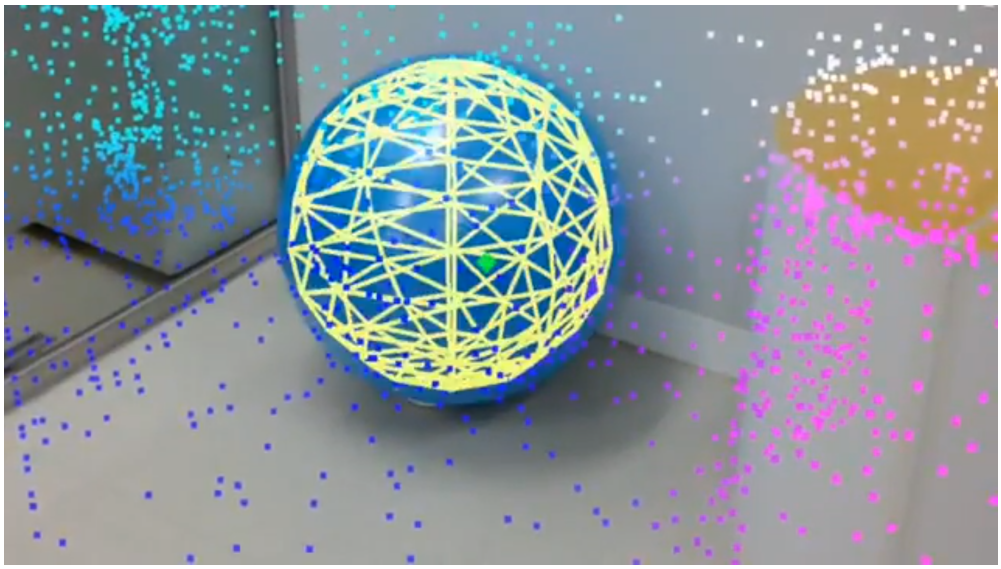


Figure 1.1: *CurvSurf's Find Surface* algorithm uses the sensors of *Microsoft's HoloLens* to detect and track 3D objects. AR applications which are reacting accordingly to the real environment's geometries are not currently possible on smartphone frameworks such as *ARCore* [7].

hardly observable though. If the user is not familiar with the new interaction methods, such as selecting with a certain finger gesture and controlling with the gaze, the user might be lost in the application and there is no instructor able to help in real-time since technical limitations.

Using network functions for observing and sharing an AR experience is possible, but limited to the same platform which is – as already mentioned – a high price-point for buying another *HoloLens* for this reason and needs additional work to add this shared aspect.

The goal of this Master's thesis is to implement a system into a use-case which allows joining a *HoloLens* session with an *ARCore* supported device. The applications should run synchronized in time – there are multiple states of the application and the hologram should be on every device in the same state – and space – the hologram should be placed on a table and all connected devices should display the hologram at the same spot.

For this approach, an optimized network procedure and both a geometric detection as well as a visual detection in the form of using image recognition will be evaluated.

Chapter 2

Mixed Reality Technologies

2.1 Mixed, Virtual and Augmented Reality or Virtuality?

With technologies that can either let virtual objects become reality or use data of the reality in virtual surroundings, many companies came up with different names to define the class of their device which processes and visualizes both real and virtual information.

Paul Milgram and Fumio Kishino introduced the *reality-virtuality continuum* [2] which tries to define which characteristics are essential for different types of reality and virtuality and how the real surrounding and the virtual objects can relate to each other.

Mixed Reality (MR) is used to describe any technology which is both using the real environment and the virtual world as a combination. No matter if the real world gets augmented with virtual objects using *Augmented Reality* (AR) or interpreting the real world and use this information in virtual worlds called *Augmented Virtuality* (AV) or anything in between. Since this year, the new term *Extended Reality* (XR) is used as a collection for any VR or MR technology [5].

Augmented Reality (AR) is the extension of the real environment using virtual objects which can act as if they were real. Unlike *Virtual Reality* (VR) in which the user dives into a visualization of a virtual world without any connection to the real world, using Augmented Reality systems makes it possible to expand the surroundings in a useful manner.

While the virtual environment in Virtual Reality is basically just a collection of data-sets, to offer an understandable behavior of *holograms* (virtual objects augmenting the real world) is highly dependent on the real environment which is not just accessible data in the memory. The environment has to be interpreted using mapping technologies and the device's position and orientation has to be tracked as well. This problem is related to achievements in Computer Vision.

2.2 Evolution of Augmented Reality

Since the first possibilities of Augmented Reality were shown, more and more companies have been working on making Augmented Reality accessible and worth using for the general public. Smaller companies started making *augmented objects* interesting for possible audiences.

2.2.1 3D Augmented Reality in Own Hands

A big change of the possibilities what Augmented Reality could be like one day was drastically changed by *Google* with *Project Tango* [10] in 2014.

Project Tango was unlike the earlier implementation which used visual markers to, for example, augment pages of magazines, but was able to perceive the world as it is; three-dimensional. This was achieved by embedding specialized hardware in *Tango*-enabled smart devices (see figure 2.1). In addition to empowered CPU solutions, a wide variety of sensors were used: The necessary kit included a wide-angle fish-eye camera, a high-resolution RGB-camera and depth sensors in combination with improved gyroscope and a fast-responding accelerometer.

Although *Project Tango* was highly supported by experts in Computer Vision and even facilities such as the *National Aeronautics and Space Administration* (NASA) which helped in testing and some of them as well in further development of the system, *Tango* was barely adopted by smartphone manufacturers due to the higher costs of embedding the additional sensors. Furthermore, due to the lack of handy applications, barely any potential user was convinced to pay more for a functionality of a device which was perceived more like a gimmick than an actual life-changer. Beside their internal testing devices, two *Tango* devices were released by *Google* and two further devices by the partners *Lenovo* and *Asus* with – as already mentioned – a smaller success than anticipated.

2.2.2 Standalone Device for our Heads

Officially since early 2016, *Microsoft* joined the development of Augmented Reality devices with the *Microsoft HoloLens*, formerly known as *Project Baraboo* [11]. Unlike *Project Tango* by *Google*, *Microsoft* did not focus on turning everyday devices into Augmented Reality devices but made *wearable smart glasses* with the only purpose of augmenting objects into the real world. The *HoloLens* projects the holograms onto the glasses, while the real world is just seen through the transparent glasses as the wearer would without them. Just as *Tango*, the *HoloLens* is able to perceive the real world in all three dimensions by using multiple cameras, depth sensors and other modules used for estimation the own position and the data of the surroundings [15]. While *Tango* works like a framework within Android applications, the *HoloLens* is a standalone device, so everything offered works in Augmented Reality only with a holographic version of Windows 10 and hardware optimized for tracking and mapping.

From the moment the *HoloLens* is started and is tracking the real world, it knows its position, orientation and the relation to the real surroundings and uses this information for every application installed. The *HoloLens* supports two types of applications: 2D applications running in a frame which can be freely positioned in the environment (e.g. stick it like a wallpaper onto the wall) and 3D applications which's frame acts as a launcher to dive into the three-dimensional application.

Although the use of gestures to control the *HoloLens* is unusual in the beginning compared to interactions such as using a mouse or a touchscreen, after a little use it feels much more natural to reach into the augmented environment and interact with the virtual objects with bare hands. Further, it offers more precision than a touchscreen in case of three-dimensional transformation manipulation. The *HoloLens* seems quite

Figure 2.1: Devices which support *Project Tango*, such as the shown *Asus ZenFone AR*, require a wide variety of visual and emitting sensors to gather enough data of the three-dimensional surrounding [6].

promising for daily use since it is not occupying the hands like holding a smartphone and offers a superior precision and stability of tracking, mapping and interactions, but for now, the technology is not accessible since the costs for it, and its knowledge included, are very high, but understandable.

2.2.3 Computer Vision Knowledge for Better Accessibility

The most accessible systems existing right now are still growing in functionality and stability. *ARCore* by *Google* [9] and *ARKit* by *Apple* are going back to the idea of the first successful Augmented Reality solution and use devices which are already used on a daily basis in our lives: smartphones.

Computing power as well as knowledge in *Computer Vision* improved drastically over the last few years which allows for a new way to handle the interpretation of the real surrounding without any additional hardware. Accelerometers and gyroscopes are now the standard sensors embedded in the latest smartphone generation and have now a higher precision in understanding position and movement of the devices. Furthermore, cameras built into smartphones improved a lot in terms of resolution, lighting situation and latency.

These improvements are what *ARCore* and *ARKit* rely on. Systematically, both systems work with the interpretation of movement and positioning using the accelerometer and gyroscope and try to interpret the surroundings using the camera data in Computer Vision techniques [8]. By doing this, the systems are able to detect rather simple references in the world which are additionally used to again improve tracking the device's position and orientation in the semi-virtual space.

By having those references and further scanning and interpreting the surrounding,

the systems are able to process the 2D images of the camera and the data of the sensors in a way the system can keep track of 3D information in both the real world and the created virtual objects.

For now, both *ARCore* and *ARKit* are able to detect flat big surfaces like the floor and – since the latest updates – walls and offers image recognition (see figure 2.2). These references can be used to place augmented objects along those surfaces or anywhere in 3D space in relation to them. A problem using this method is the *camera blur* when moving fast: Without enough detail, the systems are not able to identify reference points anymore which could lead to a temporary loss of mapping information needed to achieve a stable 3D environment, but in reasonable walking or viewing motions the tracking is steady and surprisingly precise.

This technique cannot be used for static spectating since just one and the same 2D image input is not enough to understand a 3D surrounding without any movement. Another problem for *ARCore* and *ARKit* are untextured and/or reflecting surfaces since there is no possibility to receive reliant reference points. Even within a fully tracked environment, going up close to a surface without any reference features can lead to a complete loss of mapping and tracking until the system recognizes an already mapped reference again.

2.3 Shared and Isolated Experiences

The term *Shared Experiences* will be used for describing how multiple users can see and interact with the same augmented objects *in a synchronized manner* with one or multiple devices, either in the same physical space or via networking in different parts of the world.

2.3.1 System-internal Synchronization

Both the *Microsoft HoloLens* (see section 2.2.2) as well as the smartphone solutions *ARCore* and *ARKit* (see section 2.2.3) allow multiple users with each one device to share holograms, their position and orientation as well as the status of it (for instance animations) in either the same room or via network connections.

The *smartphone solutions* allow multiple users on one device as well since the phone screen is visible from multiple angles and is not bound to the main user. On the other hand, the HoloLens visualizes the virtual content on its lenses which makes it only visible for the wearer. Additionally, the gesture controls of the HoloLens barely works from any other position as well.

2.3.2 Isolated Experience

As mentioned in the section before and in section 2.2.2, the *Microsoft HoloLens* offers the wearer a hands-free experience with a very precise tracking method, but it limits the visualization and interaction to the wearer him-/herself. At first, this might not sound like something bad but it is actually for a user who experiences everything the first time into this new world.

Microsoft offers a spectating possibility logging into the dashboard of the HoloLens

Figure 2.2: *ARCore* supports *image recognition* since version 1.2. Desired images are defined in a database. On recognition, a linked action is performed and the coordinates related to the origin can be used for further references [12].

with a secondary device such as a laptop. The dashboard offers tracking controls as well as a resulted visualization of the what the wearer sees with both the real and the virtual world embedded. The downsides are that encoding and sending the data via a wireless network connection results in a delay of multiple seconds in the preview and additionally reduces render-quality (resolution and in worst case even frame-rate) on the lenses of the HoloLens itself due to a higher need of processing power. This makes it impossible without the use of a second HoloLens to instruct the wearer in real-time and additionally destroys the illusion of a working system due to the quality changes.

2.4 Multi-Platform AR Experiences as a Solution for Isolated Experiences

To find a possible solution for the problem mentioned in the section above – it is not possible to manage or instruct wearers of Augmented Reality glasses – a combination of the systems would be necessary. The possibility to join the same augmented space with less expense than buying a second HoloLens is not existing for general use.

The idea and result of this thesis should be the thoughts and implementation of making a system reality that makes it possible to guide a HoloLens-wearer within the same 3D Augmented Reality space and administrate the running application in real-time using a smartphone running *ARCore*.

For a successful project, the system should be able to handle every needed communication between multiple devices (e.g. for synchronized behavior and animations) and further, the holograms should be placed in relation to the real environment using a visual calibration method.

Chapter 3

Red Bull Air Race Holo-Info

The creation of a framework which offers an instructor or spectator using *ARCore* to join a *HoloLens* session needs a lot of attention for managing a huge variety of use-cases and exception handling. Due to the limited time which can be used for the Master's Thesis, the essential work is to prove that such a system would work in a clearly defined use-case. After evaluation of the results of this experiment and probably even more specific applications (so far after writing this thesis), the implementation and refactoring into a framework to make this system usable for general *Shared Space* applications would start.

For this specified use-case, a collaboration with *Stefan Auer* was made, a *Human Computer Interaction* student who studied as well at the *University of Applied Sciences* in Hagenberg im Mühlkreis, Austria. For his Master's Thesis "Augmented - Reality: Development and Evaluation of a user-centered prototype for the Red Bull Air Race" [1] (translated title), he focused on possible interactions and general design thoughts for a holographic information application. The application was realized in Stefan's favour and was structured in a way it can be used as a use-case for this thesis as well.

Stefan Auer is a commentator and analyst for *Red Bull Air Races* (RBAR) and had an idea for this application which was an introduction to the rules and general information about the RBAR which could be used in the VIP-area of *Red Bull*. This use-case fits perfectly for this thesis, since instructors would be able to assist the VIP-members in interacting with the holographic application and allows them to have a talk with the viewers within the augmented space about the seen.

3.1 About Red Bull Air Races

The *Red Bull Air Race World Championship* is an extreme-sport tournament in which extremely skilled pilots compete every year since 2003 [13]. The championship 2018 consists of eight races which are flown all over the world. Every location has an own unique track with own challenges for each pilot. All tracks consist of multiple *pylons* which are air-filled obstacles used to create double gates the pilot has to fly through or around for a turn. Depending on the length of the course, the pilots fly point-to-point or lapped races.

The race directors have very precise rules about start speed, entrance angle, flight

height, g-forces and plane modification limits. The pilots are mostly flying at the rules' borderlines, physical limits and their own abilities to save every millisecond. Due to the high-speed and precision balance which every pilot tries to push as far as humanly possible, the Red Bull Air Races are very famous among flight enthusiasts and motorsport fans.

3.2 Presentation and Course of Action

The base for the real environment is a simple table which is the origin for the holographic application. A map of the track will be virtually placed on this table and shows the track and further information (see figure 3.1).

The user will be able to progress through instructions of a – in the application embedded – narrator presenting a total of five rule explanations. Every rule is displayed as a pin (see figure 3.2) which is triggered by the gaze of the viewer when looked at for a few seconds. Stefan's research regarding user attention resulted in the way that every pin is existing from the beginning on, but are activatable in a given order and color-coded so the narrator (as well as – for instance – the instructor) can refer to certain pins.

The guide shows a virtual plane which additionally enforces the vocal rule explanations with visual content via animations. After first evaluations and presentations in the public, tests clearly showed that especially users which are new to Augmented Reality and the HoloLens are likely to miss hearing more detailed information in the rules since they are overwhelmed by what is happening in front of their eyes. To balance this behavior, important details of the rules are visualized as well: For example, the values for start speed are shown in speed-o-meters, rules about flight height are visualized at the gates, etc. After the users finished the guide, they are able to replay any role explanations or get additional information about pilots.

3.3 Instruction and Possible Manipulations

The voice-files of the narrator and the structure of the program are organized in a way that the viewer can progress through the hologram without any help if he is able to. The role of the instructor is used for discussion inside the augmented world and – in the case the viewer is not familiar with the device – to assist the viewer.

All of the actions in the application are visual objects (such as pins and pilot cards) which are needed to be gazed at. To offer enough precision, a gaze point is shown as a red little ball which sticks to the virtual objects and is always in the center of the view.

Persons which are not familiar with Augmented or Virtual Reality and thus are not familiar with triggering actions by using the device comparable to the way a laser pointer is used, might still have problems activating the pins and pilot cards since the actual gaze of the viewer is not necessarily the same as the device's orientation.

In such cases, the instructor should be able to start the actions without interrupting the viewer's experience. This is realized by offering instructor buttons positioned on the edge of the screen which can be used to start related pins, so a seamless transition to the next rule explanation is possible.

Figure 3.1: The basic concept how the augmented objects are related to an existing table.

3.4 Augmented Reality Interfaces

The *Structural Design* of the use-case application is essential for the following design of network behavior described in the following chapter.

The pins are the triggers for any narrator-playbacks, animations and other visual components presented to the user. Every pin works with the same base system which handles interactions, cross-referenced pin management, network sharing and time-coded actions.

For checking which object is gazed at, a script called `GazeManager` was created. The `GazeManager` needs a game object with the `CameraAccessor` attached to it. The `CameraAccessor` is basically only a collection of the references to the Camera-object, the attached Audio-object and the cursor which shows the target of the ray-cast.

Each frame the camera position is used to fire a *ray-cast* along the gaze direction. The first object hit is checked if it has a component which implements the interface `IGazeAffected`, which offers the methods `NewGaze`, `ContinuedGaze` and `LeftGaze`. Depending on the state of the last ray-cast – which is saved and handled in the `GazeManager` itself – these, in the hit game-object implemented, functions are called accordingly.

To result in a behavior that ignores game-objects that should not block the ray-cast but just let the ray through the by *Unity* offered `IgnoreRayCast`-layer is used for objects like the plane or back-plates of the pilot profiles.

The handling how long a pin needs to be gazed at is handled in the pins itself. On activation of a pin, the pin starts multiple procedures to present the information as defined and the information that the pin got triggered is sent to the corresponding pin manager which handles the behavior of the other pins in terms of visibility and ability to be triggered.

Figure 3.2: Pin-system for triggering rule explanations.

3.5 `GuidePin` and `PilotPin`

The `GuidePin`s, which are used to guide the user through the rules, as well as the `PilotPin`s are implementing the `IGazeAffected` interface which is used by the `GazeManager` (see section 3.4) to make the pins interactable by the gaze of the user. Both of the pin-types use the interface-methods `NewGaze`, `ContinuedGaze` and `LeftGaze` to handle the activation of the pin and its animation. On the activation the pin-types differ.

`PilotPin`: On its activation the `PilotPin` shows the referenced pilot card which is shown above the map. The card shows important information about the pilot which is supported by a sound snippet of the narrator.

`GuidePin`: On the activation of the `GuidePin` additionally to the sound snippet of the narrator, an `AnimatedFlightPath` is started, which contains necessary information for the plane animations. To make it possible to add custom animations during or after the flight, the `GuidePin`-component offers the following extendable methods:

`additionalInitializing`: This method is used for additionally needed initialization such as loading reference textures, initial calculations and handling visibility of additional visual sources for a specific pin.

`additionalAnimationWhilePlaneAnimation`: Used for time-based visibility handling such as visualizing rule details, penalties, etc. Invoking those inner actions is based on the start of the plane animation (same time as pin activation).

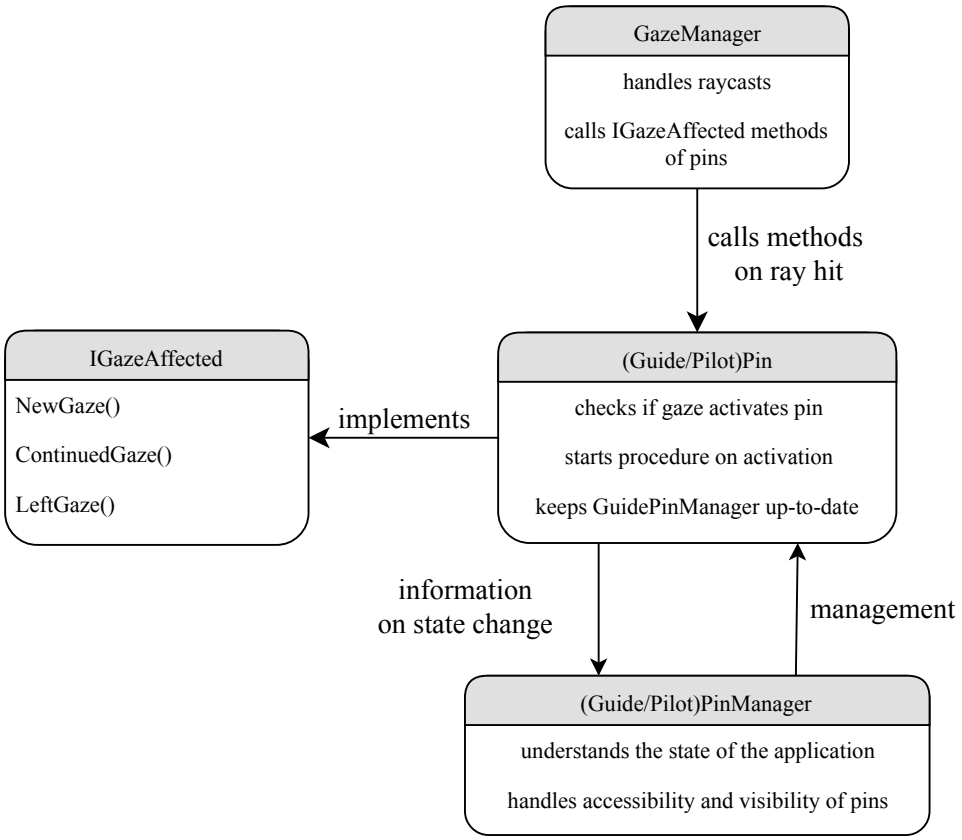


Figure 3.3: Simplified connections and actions in the pin activation process.

`additionalAnimationAfterPlaneAnimation`: Same as the method above, but uses the end of the plane animation as invocation time-stamp.

`additionalCleaning`: This method is needed to unload/hide/show the additional resources which are initialized in `additionalInitializing`. After the execution of this method, every source should be managed in a way the application shows as much information as before the pin activation.

3.6 Pin-Managers

For both types of pins, there are pin-managers. These are used for hiding all pins during the pin-actions or controlling which pins are shown. Pin-managers offer two actions `OnPinStart` and `OnPinFinished` which are called by the active pin and handle the visibility of other pins.

Another task which is handled by the pin-managers is the handling of which pins are activatable. Although all guide pins are shown, only the next one in the guide, as well as all pins that were already played, are activatable. The pilot pins are only activatable once the last guide pin with the last rule explanation was activated.

Chapter 4

Communication

The goal of the extension of the use-case application is creating a virtual behavior as if the content is existing in the real surrounding. To realize this illusion there are two essential parts to it: Synchronizing the visualization and other application output on all connected devices and merging the virtual spaces to one which will handle the position and orientation of the augmented content.

This chapter is about the communication between the devices to synchronize the applications running on the devices as well as managing authorizations which device can interact with the content or just watch to keep the run-time of the application clear and organized.

4.1 Workload distribution

Especially in the case of the *Microsoft HoloLens*, the performance overhead is very limited when running the application. Letting the devices manage permissions and information sharing between each other can get very complex and is hardly manageable in the case of debugging. Additionally, it will result in a very poor performance which can even lead to sickness due to lower frame-rate on the lenses.

To distribute this workload off the devices which are already busy in interpreting space, processing input and displaying content, a simplified *Server-Master-Slave* structure was chosen [4] as seen in figure 4.1.

The *Master* device is the device which is able to control the application, while the *Slave* devices are just receiving information. This is a very basic construct and will be extended in the following section.

The devices which are used for exploring the application are connected to the same network as a desktop personal computer which runs the server application. All relevant information which needs to be forwarded to devices is sent to the server which handles the information sharing as well as the role management.

4.2 Roles

To allow an organized interaction structure the introduction of roles and associated permissions is necessary.

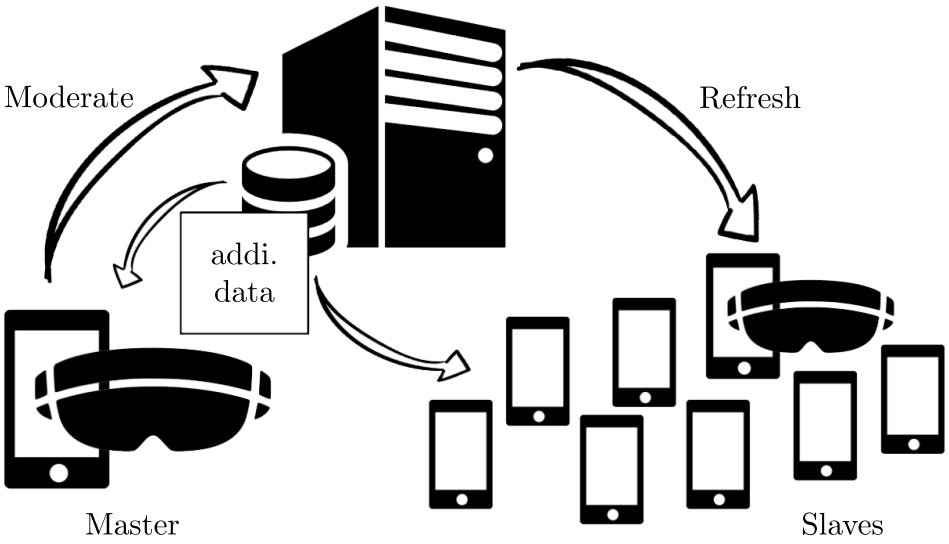


Figure 4.1: Basic network structure.

Initially, the goal of this solution is the combination of an actor who is in control of the application and an instructor who is capable of following the progress and manipulating the application in real-time – for instance, starting pins if the wearer is not able to on his own. But once this system is working, it is nearly effortless to add another role such as the spectator who can only watch, but not interact or interfere with the software.

The following role distributions are part of the initial idea, but theoretically, the instructor could as well use a *HoloLens*. It would be possible that everyone uses just an *ARCore*-supporting device without the need for a single *HoloLens* as well. It is important that the roles are distributed in the way described below, but these are not device specific.

4.2.1 Server

The server application runs on an external device such as a desktop PC and is the central control point for all information shared over the network. It is necessary to have exactly one server device in the same network as the client devices. The tasks of the server are

- confirming or changing desired roles of devices,
- forwarding data which is too confidential to save in the public application,
- keeping track of device information such as role and position,
- receiving state changes by the controller and instructor and
- forwarding these changes to all the other connected devices.

4.2.2 Controller

The controller is the main actor in the application. For the general idea of the *controller-instructor structure*, the controller is using a *Microsoft HoloLens*, but as mentioned in the role section (see section 4.2) basically any AR device could be used for this role.

To keep the run-time clean, there can be only one controller who is controlling the application. If there is an instructor, a controller is theoretically not necessary for the application to run properly, but at least either a controller or an instructor is needed since the spectators are not able to interact with the application. The tasks and permissions of the controller are

- navigating through the application and
- forwarding state changes to the server.

4.2.3 Instructor

The instructor can navigate through the application just as the controller but the instructor is meant to be able to assist the controller, if the controller is not able to activate triggers on his own or any other problem in the application shows up. The instructor application is optimized for smartphones and offers dedicated buttons to trigger pins. Only one instructor can join a session. The tasks and permissions of the instructors are the same as the controller, but navigation is not done with gazing but buttons.

4.2.4 Spectator

The role with the least amount of permissions is the spectator. The spectator is just as the name suggests only able to spectate the application from any angle, but not able to interact with it in any way.

4.3 Networking in Unity

The standalone version of the use-case application is made in *Unity 2018* which is further used to extend the application with the networking features and the space synchronization which is described in the next chapter.

Unity offers multiple versions of networking using a mix of *Transmission Control Protocol (TCP)* and *User Datagram Protocol (UDP)*. The two roles of devices – client and server – both offer to create handlers which get triggered if a certain type of message is received. *Unity Network Messages* using the `MessageBase` can hold multiple simple or advanced data types which get serialized and de-serialized by the network system.

4.3.1 Multiplayer Optimization in Unity 5

Unity introduced a serious change of networking features in *Unity 5*. With the new versions of Unity building a multiplayer game working in the *Local Area Network (LAN)*, as well as the connection to server structures handling online multiplayer, was made much easier to implement in Unity.

Scripts as the `NetworkServer` allow multiple `NetworkClient`-connections via a predefined `NetworkManager`. The `NetworkServer` offers to take care of all needed updates sent to clients, such as automatic transformations of the player avatars.

4.3.2 Using the Legacy Networking

For the planned interactions the system mentioned in the last section is far too superior since a few tasks such as the verification and transformation updates are done automatically and can barely be changed without deeper manipulation of the scripts themselves. The use of dummy objects which would be controlled by this system would be possible but the processing power needed for incoming and outgoing data does not compare with a system that is optimized for the needed tasks.

Conveniently, *Unity* still supports the networking structure which was the default way of handling networking before the multiplayer update in Unity 5.

This way the used `NetworkServerSimple` has no automated tasks but simply allows registering handlers which get triggered on incoming `NetworkMessage`s of a given type which is less data to process than in the updated network system. The message then gets processed in the handler methods which are defined by the programmer.

4.4 Network Procedure

This section shall describe how and when needed information is spread through the network and how the connections and roles are established.

4.4.1 Brief Overview

A visualization of the network procedure described in the following paragraphs is shown in figure 4.2 and 4.3. The following sections will describe the content of messages and feedback of the application in more detail.

Initializing Procedure: At the start of the application, all needed networking functions get initialized and a broadcast is sent into the network to find a possible server. To determine if a server is suitable and not just any device, a `DirectConnectRequestMessage` is sent. When the broadcast reaches the actual server, the server sends a `DirectConnectResponseMessage` back which triggers the client to change to a direct connection rather than further communication via broadcasts. After the direct connection is established, the client device sends a `RoleRequestMessage` to the server which gets answered with a `RoleResponseMessage`. With the information of the response, the client finishes the application initialization by managing content and functions which are restricted to certain roles.

Looping Procedure: As soon as the roles are set and confirmed by the server, the application is successfully running in network mode. Debugging information such as position, orientation and application status are sent ten times each second via `DeviceTransformationMessages` to the server which displays the information for a technician to be interpreted in an error case. The moment the controller or instructor interacts

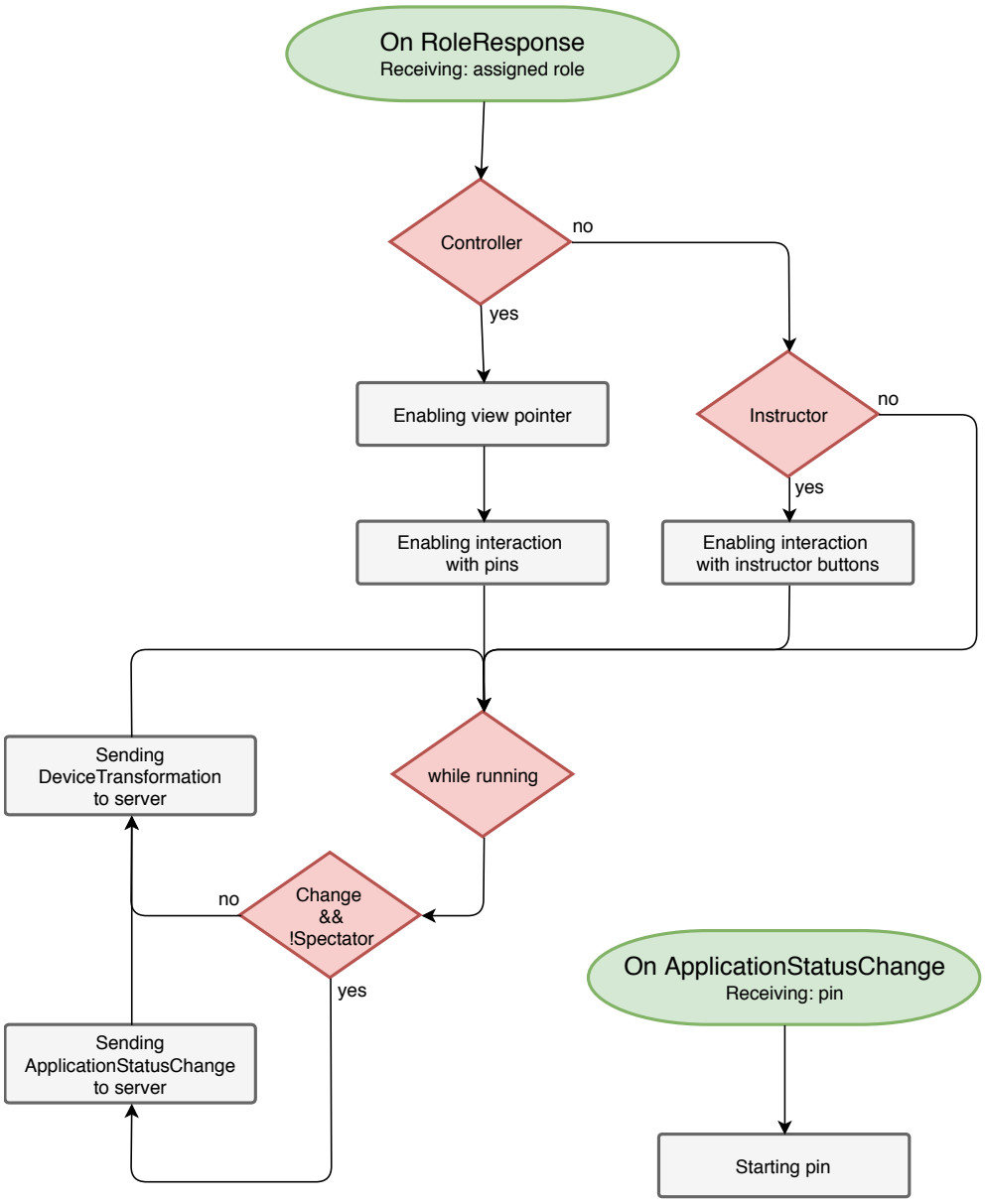


Figure 4.2: On receipt of the `RoleResponseMessage`, the application's initialization is completed and re-occurring methods are executed.

with the application which would cause a change in the application, the device sends a `ApplicationStatusChangeMessage` to the server which is checked by the server. If the authorization allows this process, the server will then send further `ApplicationStatusChangeMessages` to all other connected devices which then start the procedure locally.

Ending Procedure: Once the application is closed on the device or the connection would fail, the server will no longer receive `DeviceTransformationMessages` which lets the

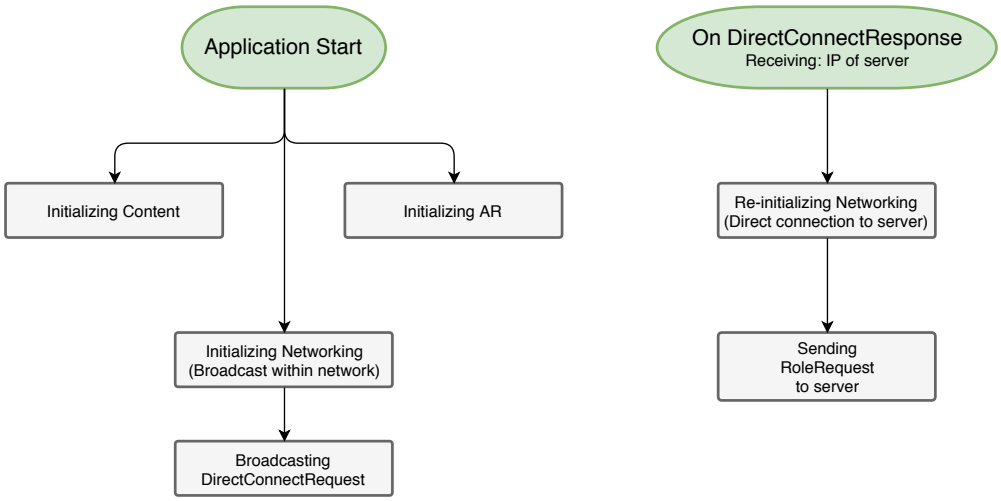


Figure 4.3: Initial network procedure.

server remove the device from its connection list. On a reconnect, the full initial procedure is done again.

4.4.2 Direct Connection Messages

When starting the client application, the state of the server, as well as a possible communication method, are unknown. Instead of making the connection setup complicated for the user by selecting the correct device or even prompt an IP address, the first message sent by the application is a broadcast into the network for an automated server search (see figure 4.4).

To send a broadcast, the message is sent to the *broadcast address* of the network. The broadcast address is the last IP address in a *subnet*. To make things easier and faster, the script assumes automatically that the subnet is a C-network – a network with a 24 net-bit subnet mask `255.255.255.0`. For instance, if the IP address of the client device is `192.168.0.20`, the target of the `DirectConnectRequestMessage` is the broadcast address `192.168.0.255`. The only content of the message is – for debugging reasons – the name of the client device.

On receiving the `DirectConnectRequestMessage` sent by a client, the server responds with a `DirectConnectResponseMessage`. This message holds the IP address of the server which is used by the client device to build up a direct connection instead of the continued use of broadcasting.

4.4.3 Role Messages

After receiving the `DirectConnectResponseMessage`, the client closes the connection to the broadcast address and establishes a direct connection using the server IP address sent via the `DirectConnectResponseMessage`.

The final initialization step is requesting the desired role – controller, instructor or spectator – using a `RoleRequestMessage`.

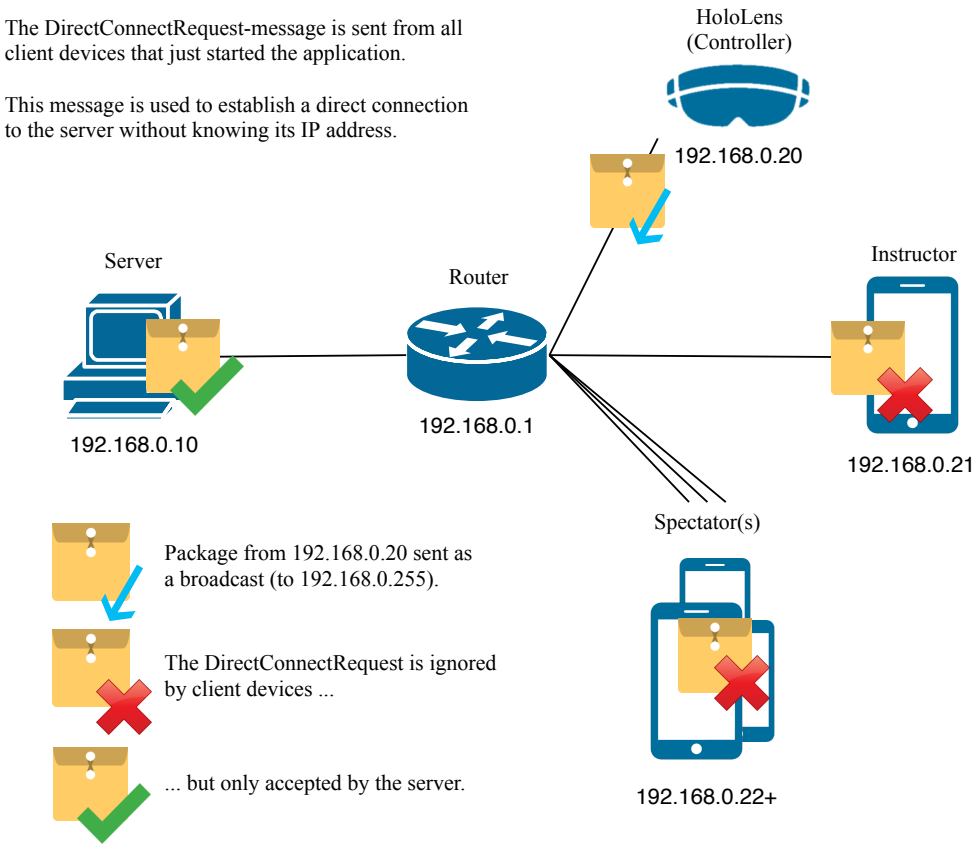


Figure 4.4: The `DirectConnectRequestMessage` is the first step for an automated server connection.

The server checks for the desired role if any other devices already claimed that role. If the desired role is a unique role such as the controller and instructor and the role is already in use, the server responds with a repellent `RoleResponseMessage`.

If the requested role can be accepted by the server, the server saves connection data sorted by roles including the time-stamp of the last received message by a certain device which is used for connectivity control (see the following section 4.4.4). Further, the server initializes visualizations and status information about the authenticated device for debugging in the server application.

Once the client received his accepted `RoleResponseMessage`, the application completes initialization of data and features which are role-dependant. On rejection, the client will send another `RoleRequestMessage` requesting the spectator role.

4.4.4 Device Transformation and Timeout

A re-occurring task for the client is started once its role got accepted in the `RoleResponseMessage`: The client sends now multiple times a second `DeviceTransformationMessages` to the server carrying the local position of the device which is displayed in the server application. This data could indicate possible tracking issues.

Furthermore, `DeviceTransformationMessages` are refreshing the server internal time-stamp which is used to show which devices are still connected. If a device would freeze, loose connection or disconnect for any other reason which would lead to a time-out due to the lack of received `DeviceTransformationMessages`, the server log would show it.

In general, there is much data sent within the network just for observing reasons. The server does not only manage network message and respond to them, but log nearly every action in the background. Additional to the raw log, much information is visualized as well to quickly identify possible network or tracking problems.

4.4.5 Changes of the Application

Both the controller as well as the instructor are able to interact with the application. To synchronize all devices to a new application state, the interacting roles send a `ApplicationStatusChangeMessage` to the server as soon as a pin got activated locally.

The `ApplicationStatusChangeMessage` saves information about the device which activated a pin, the pin number, if it is a guide pin or pilot pin – which triggers either the `GuidePinManager` or the `PilotPinManager` – and if the pin got activated or the animation of it ended.

If the server receives such a message, it checks if the sending device is authorized to request an application change and saves the new state of the application. If a pin is already running or a device requests a change without permission, the interaction gets reported in the server log which indicates a synchronization malfunction.

When a request is successful, the server sends every other connected device another `ApplicationStatusChangeMessage` which activates the regarded pin locally. This behavior prohibits multiple controlling users as well as it detects synchronization problems or other malfunctions of the software.

These tasks usually create a delay between the activating device and other clients of less than 100 milliseconds, since the activating device starts the pin already locally without waiting for a message. The delay could be reduced by ignoring the initial activation and submitting an own message for the actor, but the delay is short enough to not interrupt or worsen any experience.

4.5 Implementation

In this section, there will be brief descriptions and overviews of how the roles of section 4.2 and the procedure of section 4.4 were implemented in the Unity project.

The full source code for these scripts is collected on the DVD in the folder `RedBull-AirRace/Assets/Scripts/Networking`.

4.5.1 Messages

For every message a class extending `UnityEngine.Networking.MessageBase` was created in the sub-folder `Messages`.

Within the class the structure of the message is defined using public members:

```
public class DeviceTransformation : MessageBase {
    public string deviceName;
    public Vector3 position;
    public Vector3 rotationEuler;
}
```

This step alone is not enough for Unity to differ messages from each other. To allow a correct (de-)serialization every message type needs a unique message type identifier using the data type `short`. To allow easy access to those identifiers, they were defined in an own class `MsgTypeExt`:

```
public class MsgTypeExt {
    public static short DirectConnectRequest = MsgType.Highest + 1;
    public static short DirectConnectResponse = MsgType.Highest + 2;
    public static short RoleRequest = MsgType.Highest + 3;
    public static short RoleResponse = MsgType.Highest + 4;
    public static short DeviceTransformation = MsgType.Highest + 5;
    public static short ApplicationStatusChange = MsgType.Highest + 6;
}
```

The member `MsgType.Highest` is the highest already used unique identifier for message types defined by *Unity* itself. With these definitions it is already possible to send – for instance – a `DeviceTransformationMessage` to the server:

```
public class ClientDevice : MonoBehaviour {
    // ...
    protected NetworkClient client;
    protected Transform deviceTransformation;
    // ...
    protected void sendDeviceTransformationToServer() {
        DeviceTransformation message = new DeviceTransformation();
        message.deviceName = SystemInfo.deviceName;
        message.position = this.deviceTransformation.position;
        message.rotationEuler = this.deviceTransformation.eulerAngles;
        this.client.Send(MsgTypeExt.DeviceTransformation, message);
    }
    // ...
}
```

4.5.2 Roles

The roles are used to check if certain permissions are granted. The roles are defined as an `enum` and offer methods for casting the role from the `enum` to an `int` and vice versa.

```
public class DeviceRole {
    public enum Role {
        Controller = 0,
        Instructor = 1,
        Spectator = 2,
        Server = 3,
        Invalid = 4
    }

    public static int getRoleAsInteger(Role role) {
        return (int)role;
    }
```

```
        public static Role getIntegerAsRole(int integer) {
            return (Role)integer;
        }
    }
```

This offers versatility to cast them into a format that can be used in network messages:

```
public class ClientDevice : MonoBehaviour {
    // ...
    protected DeviceRole.Role role;
    protected NetworkClient client;
    // ...
    public void OnBasicConnect(NetworkMessage netMsg) {
        // ...
        RoleRequest request = new RoleRequest();
        request.deviceName = SystemInfo.deviceName;
        request.deviceRole = DeviceRole.getRoleAsInteger(this.role);
        this.client.Send(MsgTypeExt.RoleRequest, request);
    }
    // ...
}
```

and further allows an easy-to-read source code for permissions:

```
public class ClientDevice : MonoBehaviour {
    // ...
    protected DeviceRole.Role role;
    protected bool roleVerified;
    // ...
    public void sendApplicationStatusChangeToServer(/* ... */) {
        if (this.roleVerified &&
            ( this.role == DeviceRole.Role.Controller ||
            this.role == DeviceRole.Role.Instructor )) {
                // granted
                // ...
        }
    }
    // ...
}
```

4.5.3 Outsourcing Permissions

To keep the content of the `ClientDevice` script clean and remove any specific network- or role-depending features, the interface `INetworkAffected` was created. The interface consists of two methods which need to be defined in any network dependent script:

- `void OnNetworkVerification(DeviceRole.Role role)` and
- `void OnNetworkDisconnect()`

For instance, if the connection to the server is established and the role got confirmed, the `ClientDevice` script would call the `OnNetworkVerification` method of all scripts which implement `INetworkAffected`:

```
public class ClientDevice : MonoBehaviour {
    // ...
    public void OnRoleResponse(NetworkMessage netMsg) {
        RoleResponse response = netMsg.ReadMessage<RoleResponse>();

        if (response.accepted) {
            // ...
            GameObject[] affectedGOs =
                GameObject.FindGameObjectsWithTag("NetworkAffected");
            foreach (GameObject affectedGO in affectedGOs) {
                INetworkAffected[] affectedComponents =
                    affectedGO.GetComponents<INetworkAffected>();
                foreach (INetworkAffected affectedComponent in affectedComponents) {
                    affectedComponent.OnNetworkVerification(this.role);
                }
            }
            // ...
        }
        // ...
    }
    // ...
}
```

This event could, as in the example provided, permit using the instruction buttons:

```
public class ButtonActiveForRole : MonoBehaviour, INetworkAffected {
    protected Button button;
    protected Image buttonImage;
    protected GameObject buttonTextGO;
    protected bool buttonEnabled = false;
    [SerializeField]
    protected DeviceRole.Role allowedRole;
    // ...
    protected void updateButtonState() {
        this.button.enabled = this.buttonEnabled;
        this.buttonImage.enabled = this.buttonEnabled;
        this.buttonTextGO.SetActive(this.buttonEnabled);
    }
    // ...
    public void OnNetworkVerification(DeviceRole.Role role) {
        this.buttonEnabled = ( role == this.allowedRole );
        this.updateButtonState();
    }
    // ...
}
```

4.5.4 Server Functionality

The `Server` script consists of multiple members used for saving connection data, the application status and debugging information rendering references which are all used in methods for fast and easy checks of connections, their roles and timeouts. The general base of the network functionality is implemented as *network handlers*.

Only network specific code is described in the next sections; the full source code is available at `RedBullAirRace/Assets/Scripts/Networking/Devices/Server.cs` on the DVD.

Members

```
public class Server : MonoBehaviour {
    protected NetworkServerSimple server;
    protected int port = 9292;
    protected float timeout = 5f;

    protected NetworkConnection controllerDevice;
    protected float controllerDeviceLastPackage;

    protected NetworkConnection instructorDevice;
    protected float instructorDeviceLastPackage;

    protected Dictionary<int, NetworkConnection> spectatorDevices;
    protected Dictionary<int, float> spectatorDevicesLastPackage;

    protected bool pinRunning = false;

    [Header("Outputs")]
    // references to UI elements for outputs such as log, connected devices, etc.
    // ...

    [Header("Rendering")]
    // references to position rendering of the devices
    // ...

    // ...
}
```

`NetworkServerSimple`: As already mentioned in sections 4.3.1 and 4.3.2, the new networking features released in *Unity 5* are offering too many automatic tasks such as automatic player game-object transformation synchronization and for this use-case a self-impairing message-system. This is why `NetworkServerSimple` is used instead of `NetworkServer` which are both part of the package `UnityEngine.Networking`. This instance offers all the needed functions to send custom messages across the network or directed at a certain device.

`NetworkConnection`: These instances represent connections to other devices and offer functions to send a message to this connection instead of using the server instance to find the connection-details. The `NetworkServerSimple` has an own array to save active connections but since this is not easy extendable with more specific parameters like roles or similar, the connections are saved in the script in own objects for controller and instructor. For spectators, a `Dictionary` contains all the data using an `Integer` as key due to the fact that every connection has an own unique integer identifier in *Unity*.

Time-stamp of last Package: As soon as the devices are connected and got their role verified, the method `void OnData(int connId)` gets called on every message receipt. The function saves the recent time-stamp referenced to the role and connection identifier which is used to check every two seconds if devices disconnected without the usual disconnect message (this is described later in this section).

Application Status: Last but not least, the server simply saves if the last `ApplicationStatusChangeMessage` (see section 4.4.5) requested a pin activation or informed the server about a finished pin. This is used to make a pin activation impossible once a pin is already running. This could happen if the controller activates a pin and before the information about the change reaches the instructor, the instructor is activating a pin as well. Another case in which this system is necessary is on failed connections and another device, which thinks it is an instructor client-sided, tries to activate a pin (which is additionally secured with the server-sided role verification system).

Initialization and Message Processing

```
public class Server : MonoBehaviour {
    // ...
    protected void Start() {
        // initializing GUI access
        // ...

        this.Initialize();
    }

    public void Initialize() {
        this.spectatorDevices = new Dictionary<int, NetworkConnection>();
        this.spectatorDevicesLastPackage = new Dictionary<int, float>();
        this.spectatorRenderings = new Dictionary<int, GameObject>();
        this.server = new NetworkServerSimple();
        this.server.RegisterHandler(MsgTypeExt.DirectConnectRequest,
            OnDirectConnectRequest);
        this.server.RegisterHandler(MsgTypeExt.RoleRequest,
            OnRoleRequest);
        this.server.RegisterHandler(MsgTypeExt.DeviceTransformation,
            OnDeviceTransformation);
        this.server.RegisterHandler(MsgTypeExt.ApplicationStatusChange,
            OnApplicationStatusChange);
        this.server.Listen(this.port);
        InvokeRepeating("checkSteadyConnection", 2f, 2f);
        this.writeLog("The server listens to " +
            NetworkingTools.getLocalIPAddress() + ":" + this.port);
    }

    protected void Update() {
        if (this.server != null) {
            this.server.Update();
        }
    }
    // ...
}
```

In *Unity*, the method `void Start()` is executed in the beginning of the application. A typical task, as done here, is the initialization of the `Directory` instances. Further, the `NetworkServerSimple` instance, used for the communication between the devices, is created and *handlers* (see following sections) for all necessary incoming message types (see section 4.5.1) are defined. Once the server got initialized, every frame (using `void Update()`), the server checks the incoming messages and forwards them to the referred

methods defined in the handlers.

Assisting Methods

```
public class Server : MonoBehaviour {
    // ...
    protected void writeLog(string logLine) {
        // used to write something into the raw log (...)
    }

    protected void checkSteadyConnection() {
        if (this.controllerDevice != null) {
            if (this.controllerDeviceLastPackage + this.timeout < Time.time) {
                OnControllerDeviceDisconnect(true);
            }
        }
        // check other devices with other roles (...)
    }

    protected DeviceRole.Role checkRoleOfConnection(int connectionID) {
        // checking controller
        if (this.controllerDevice != null) {
            if (this.controllerDevice.connectionId == connectionID) {
                return DeviceRole.Role.Controller;
            }
        }

        // check other roles (...)

        // no role indentified
        return DeviceRole.Role.Invalid;
    }

    protected void setTransformationOfRendering(...) {
        // used for updating the rendering of the position in space (...)
    }

    protected void OnData(int connId) {
        // sent by controller?
        if (this.controllerDevice != null) {
            if (connId == this.controllerDevice.connectionId) {
                this.controllerDeviceLastPackage = Time.time;
                return;
            }
        }
        // check other roles and connections (...)
    }

    protected void OnControllerDeviceDisconnect(bool timedout = false) {
        // resetting visualization and updating information (...)
    }

    protected void OnInstructorDeviceDisconnect(bool timedout = false) {
        // resetting visualization and updating information (...)
    }
```

```
        protected void OnOtherDeviceDisconnect(int connId, bool timedout = false) {
            // resetting visualization and updating information (...)
        }
        // ...
    }
```

Timeout: As already mentioned in the initialization process of the server, the method `void checkSteadyConnection()` iterates through all connected devices and checks if the timeout threshold was reached. Usually, every device sends a `DeviceTransformationMessage` multiple times a second (see section 4.4.4). Every time the server receives a message, it refreshes the time-stamp of the connection in the method `void OnData(int connId)`. If a connected device does not send messages for five seconds (according to the member `float timeout`), it is assumable that the device froze or lost connection in another way which leads to a server-sided disconnect. Client-sided, the device should recognize the connection-loss as well and will remove all of the client-sided permissions and tries to reconnect to the server using again a broadcast (in case the IP of the server changed).

Role Check: Not only is the role confirmed by the server using the `RoleResponseMessage` (see section 4.4.3), but confirmed roles are saved on the server in case a bug in the client application would falsely interpret messages or a short connection loss would not remove the client-sided permissions. For diverse (network-related) exceptions, the server is the confirming instance which decides if an action is granted or not.

Other Methods: The rest of the methods shown in the above source code are mainly for managing connection data and rendering of the devices. These are mostly used for debugging information and role changes which are not further described to keep the focus on the important methods but the full source code available on the DVD is reasonable commented for further information.

Handlers

This section gives insight about how the server responds to certain types of network messages sent by the client devices. Again, only essential parts of the source code are presented and discussed here, while the full source code is available on the DVD in `RedBullAirRace/Assets/Scripts/Networking/Devices/Server.cs`.

```
public class Server : MonoBehaviour {
    // ...
    public void OnDirectConnectRequest(NetworkMessage netMsg) {
        // send response
        DirectConnectResponse response = new DirectConnectResponse();
        response.deviceName = SystemInfo.deviceName;
        response.ip = NetworkingTools.getLocalIPAddress();
        netMsg.conn.Send(MsgTypeExt.DirectConnectResponse, response);
        this.writeLog(
            "The device " + netMsg.ReadMessage<DirectConnectRequest>().deviceName +
            " is switching to direct connect..."
        );
    }
```

```
    // ...
}
```

On receiving a `DirectConnectRequestMessage`, the content of that message is not interesting, except for displaying debugging information. Handler methods receive a `NetworkMessage` which not only can be read to result in the needed message type but further holds information about the sender. The member `conn`, which is an instance of `NetworkConnection`, can be directly used to send a `DirectConnectResponseMessage`, containing the IP address of the server, back.

```
public class Server : MonoBehaviour {
    // ...
    public void OnRoleRequest(NetworkMessage netMsg) {
        // read message
        RoleRequest request = netMsg.ReadMessage<RoleRequest>();
        // logging information (...)

        // prepare response
        RoleResponse response = new RoleResponse();
        response.deviceName = SystemInfo.deviceName;
        response.accepted = false;

        // check if desired role can be accepted
        if (DeviceRole.getIntegerAsRole(request.deviceRole) ==
            DeviceRole.Role.Controller && this.controllerDevice == null
        ) {
            response.accepted = true;
            this.controllerDevice = netMsg.conn;
            this.controllerDeviceLastPackage = Time.time;
            this.controllerDeviceOutputText.text = request.deviceName;
            // logging information (...)
        }

        // similar check for instructor; spectators need no check (...)

        // refresh timeout
        OnData(netMsg.conn.connectionId);

        // send response
        netMsg.conn.Send(MsgTypeExt.RoleResponse, response);
        if (!response.accepted) {
            // logging information (...)
        }
    }
    // ...
}
```

The handler `void OnRoleRequest(NetworkMessage netMsg)` is executed on receiving a `RoleRequestMessage`. Once the `NetworkMessage` was read for the related message type, the information is checked. If the requested role is either controller or instructor, the server checks if the role is already in use; if it is free, it saves the connection data of the device, initializes the timeout functionality, displays information about the connection and sends a `RoleResponseMessage` to the client. From now on, the device is confirmed on the server and the client will enable permission-dependent actions on the device.

```
public class Server : MonoBehaviour {
    // ...
    public void OnDeviceTransformation(NetworkMessage netMsg) {
        // refresh timeout
        OnData(netMsg.conn.connectionId);

        // read message
        DeviceTransformation message = netMsg.ReadMessage<DeviceTransformation>();

        // refresh rendering transformation
        DeviceRole.Role role = this.checkRoleOfConnection(netMsg.conn.connectionId);
        Vector3 position = message.position;
        Vector3 rotationEuler = message.rotationEuler;
        this.setTransformationOfRendering(...);
    }
    // ...
}
```

As already mentioned in the initialization process of the server, `DeviceTransformationMessages` are sent multiple times a second by the connected and verified clients. Beside the debugging view in the server application which shows the position of the devices related to the map in the center of the scene (which would show tracking issues), these messages are used to check if the connection is still successful; if a device would loose the connection or freeze, the messages would not be sent, resulting in a timeout.

```
public class Server : MonoBehaviour {
    // ...
    public void OnApplicationStatusChange(NetworkMessage netMsg) {
        // refresh timeout
        OnData(netMsg.conn.connectionId);

        // read message
        ApplicationStatusChange request =
            netMsg.ReadMessage<ApplicationStatusChange>();

        // check if the request came from an authorized device
        DeviceRole.Role role = this.checkRoleOfConnection(netMsg.conn.connectionId);
        if (role == DeviceRole.Role.Controller ||
            role == DeviceRole.Role.Instructor)
        {
            // authorized

            // checking if action is possible due to application status (...)

            // change application status text (...)

            // send application change to other devices (only if pin got activated)
            if (request.pinStarted) {
                // pin started -> send to all other devices
                request.deviceName = SystemInfo.deviceName;

                if (role != DeviceRole.Role.Controller &&
                    this.controllerDevice != null)
                {
                    this.controllerDevice.Send(MsgTypeExt.ApplicationStatusChange,
                        request);
                }
```

```
                // same for instructor and spectators (...)
            }
        } else {
            // log action by spectator, client-sided error (...)
        }
    }
    // ...
}
```

When receiving a `ApplicationStatusMessage`, the first task for the server is to check if the sender is permitted to call such an action. If a message comes from a device other than a controller or instructor, there is probably a client-sided error since this device should not be able to send this request. Further, the server will cross-check if the request is able by comparing the application status with it. If all checks are successful, the server will forward this message to all other devices and force them to activate that pin locally. The moment the pin finishes, the sender of the activation request will send another `ApplicationStatusMessage`, telling the server that the pin was completed so the server is ready for a new message on activation by a controller or instructor.

4.5.5 Client Functionality

The functionality of the script `ClientDevice` – full source code at `RedBullAirRace/-Assets/Scripts/Networking/Devices/ClientDevice.cs` on the DVD – is not that different from the `Server` script (see section 4.5.4). It uses as well network handlers for receiving and processing messages sent by the server. Additionally to the server's functionality, the client obviously has some deeper connections to the actual application and offers a few methods which are used by other scripts (for instance: activation of pins in the `Pin` script and then sending an `ApplicationStatusMessage` using the `Client-Device` script).

Members

```
public class ClientDevice : MonoBehaviour {
    protected CameraAccessor cameraAccessorComponent;
    protected GuidePinManager guidePinManager;
    protected PilotPinManager pilotPinManager;
    protected Transform deviceTransformation;
    protected float refreshRateTransform = .05f;
    protected NetworkClient client;

    [SerializeField]
    protected DeviceRole.Role role;

    [SerializeField]
    protected string broadcastIPAddressOverride;

    protected int serverPort = 9292;
    protected bool serverFoundViaBroadcast = false;
    protected bool roleVerified = false;
    // ...
}
```

For the needed access to the local application data, the `CameraAccessor` – which offers methods for getting the camera object, audio source and cursor – the transformation (local position) of the device and access to the pin managers is saved.

The equivalent of `NetworkServerSimple` for client-sided use (only connection to one server device) is `NetworkClient` which's instance allows us to receive and send custom network message needed for the synchronization and management of devices.

The member `bool serverFoundViaBroadcast` remembers which type of connection the client device need to establish. If this member is `false` a server was not found yet by using a broadcast or the connection to an old server got lost and the next connection should be established using a broadcast to find a server. If a server was found and answers with a `DirectConnectionResponse` (see section 4.4.2), the member will have the value `true` to indicate to use a given IP instead of a broadcast IP as target.

Another safety feature is blocking actions which need a verified role without having a confirmation by the server. The role of the device is set in the script in the member `role` which is only a desired role which does not mean that the role was confirmed by the server; which is the task of `roleVerified`.

Initialization

```
public class ClientDevice : MonoBehaviour {
    // ...
    protected void Start() {
        this.Initialize();
    }

    public void Initialize() {
        #if UNITY_ANDROID
            StartCoroutine(CheckCompatibility());
        #endif
        // getting reference to CameraAccessor instance (...)
        if (this.cameraAccessorComponent == null) {
            Debug.LogError("CameraAccessor not found!");
        } else {
            // getting references to other local scripts (...)
            this.client = new NetworkClient();
            this.client.RegisterHandler(MsgType.Connect, OnBasicConnect);
            this.client.RegisterHandler(MsgType.Disconnect, OnDisconnect);
            this.client.RegisterHandler(
                MsgTypeExt.DirectConnectResponse, OnDirectConnectResponse);
            this.client.RegisterHandler(MsgTypeExt.RoleResponse, OnRoleResponse);
            this.client.RegisterHandler(
                MsgTypeExt.ApplicationStatusChange, OnApplicationStatusChange);
            this.searchAndConnectServer();
        }
    }
    // ...
}
```

The initialization of the `ClientDevice` is straight forward: Saving the references to needed local scripts, registering network handlers (details in further sections) and connecting to the server.

Additionally in the *ARCore* build, the method `void CheckCompatibility()` is called which checks if the device supports *ARCore*, needs an update or is unsuitable for

the application.

The method `void searchAndConnectServer()` is used to search for a server in the network using broadcast if the member `bool serverFoundViaBroadcast` is still `false`.

Assisting Methods

```
public class ClientDevice : MonoBehaviour {
    // ...
    protected void sendDeviceTransformationToServer() {
        DeviceTransformation message = new DeviceTransformation();
        #if UNITY_ANDROID
            message.deviceName = SystemInfo.deviceModel;
        #else
            message.deviceName = SystemInfo.deviceName;
        #endif
        message.position = this.deviceTransformation.position;
        message.rotationEuler = this.deviceTransformation.eulerAngles;
        this.client.Send(MsgTypeExt.DeviceTransformation, message);
    }

    public void sendApplicationStatusChangeToServer(string pinName, int
    internalPinNumber,
        bool guidePin, bool started
    ) {
        if (this.roleVerified &&
            ( this.role == DeviceRole.Role.Controller ||
            this.role == DeviceRole.Role.Instructor )
        ) {
            ApplicationStatusChange message = new ApplicationStatusChange();

            #if UNITY_ANDROID
                message.deviceName = SystemInfo.deviceModel;
            #else
                message.deviceName = SystemInfo.deviceName;
            #endif
            message.affectedPinName = pinName;
            message.internalPinNumber = internalPinNumber;
            message.guidePin = guidePin;
            message.pinStarted = started;
            this.client.Send(MsgTypeExt.ApplicationStatusChange, message);
        }
    }
    // ...
}
```

The method `void sendDeviceTransformationToServer` is executed multiple times a second – depending on the value of the member `float refreshRateTransform` – once the role was verified with a received `RoleResponseMessage`. So by default, every 50 milliseconds the local transformation in relation to the origin of the scene is read and sent to the server which will display this information. This is mainly for interpreting tracking issues.

Depending on the pin type, the `GuidePinManager` or `PilotPinManager` are called by a pin when it gets activated. The pin managers check if the activation is conform with

the local role and send an `ApplicationStatusChangeMessage` to the server using the method `void sendApplicationStatusChangeToServer` provided in the `ClientDevice` script. The server cross-checks the permission and if granted, forwards the information to other connected devices (see more details in section 4.5.4).

Handlers

Last but definitely not least, the network handlers are the most important definitions since they handle receiving messages and the correct response or following actions needed. A brief overview of which handlers need to be registered was already shown in the initialization section of the `ClientDevice` script.

```
public class ClientDevice : MonoBehaviour {
    // ...
    public void OnBasicConnect(NetworkMessage netMsg) {
        if (!this.serverFoundViaBroadcast) {
            // server found by broadcast
            this.serverFoundViaBroadcast = true;
            // requesting direct connection
            DirectConnectRequest request = new DirectConnectRequest();
            #if UNITY_ANDROID
                request.deviceName = SystemInfo.deviceModel;
            #else
                request.deviceName = SystemInfo.deviceName;
            #endif
            this.client.Send(MsgTypeExt.DirectConnectRequest, request);
        } else {
            // server was already found by broadcast
            // client connected now via direct connection
            // send role-request
            RoleRequest request = new RoleRequest();
            #if UNITY_ANDROID
                request.deviceName = SystemInfo.deviceModel;
            #else
                request.deviceName = SystemInfo.deviceName;
            #endif
            request.deviceRole = DeviceRole.getRoleAsInteger(this.role);
            this.client.Send(MsgTypeExt.RoleRequest, request);
        }
    }

    // ...

    public void OnDirectConnectResponse(NetworkMessage netMsg) {
        // read message
        DirectConnectResponse response =
            netMsg.ReadMessage<DirectConnectResponse>();
        // connect directly
        this.client.Disconnect();
        this.client.Connect(response.ip, this.serverPort);
    }
    // ...
}
```

The handler `void OnBasicConnect` is called on the very first connection step between client and server; when the client and server never communicated before or had a successful disconnect.

If the client and server did not connect before, the client used a broadcast to reach it (according to the planned network procedure described in section 4.4). To minimize network load the system changes after the broadcast server search to a direct connection via IP which is received after sending a `DirectConnectRequest` to the server. As seen in the handler method `void OnDirectConnectResponse`, the next step after a basic connection is done in the `void OnBasicConnect` handler and not further defined in the response handler.

The client receives the `DirectConnectResponseMessage`, closes the broadcast connection and connects again using the IP address in the received message. On the second basic connection between client and server, the member `bool serverFoundViaBroadcast` indicates that not another `DirectConnectRequest` is needed but the direct connection is established and a role request should be sent.

```
public class ClientDevice : MonoBehaviour {
    // ...
    public void OnRoleResponse(NetworkMessage netMsg) {
        // read message
        RoleResponse response = netMsg.ReadMessage<RoleResponse>();

        if (response.accepted) {
            // role accepted
            this.roleVerified = true;

            // start transmitting device-transformation
            InvokeRepeating("sendDeviceTransformationToServer", 0f, .05f);

            // calling OnNetworkVerification of INetworkAffected-scripts (...)
        } else {
            // role denied
            // change to spectator
            this.role = DeviceRole.Role.Spectator;
            this.client.Disconnect();
            this.searchAndConnectServer();
        }
    }
    // ...
}
```

After the server processes the sent `RoleRequestMessage`, it responds with a message handled by the `void OnRoleResponse` handler. If the desired role was accepted, the member `bool roleVerified` is set to `true` which is needed for local permission checks, the client starts to send the transformation messages needed for debugging and timeout features and calls the `INetworkAffected` scripts of the application (see section 4.5.3). After this procedure, the device is officially connected with the server, completed the initialization and is therefore synchronized in the application.

If the desired role was not accepted, the desired role is set to the spectator role and the client will disconnect from the server and complete the full initialization procedure again.

```
public class ClientDevice : MonoBehaviour {
    // ...
    public void OnApplicationStatusChange(NetworkMessage netMsg) {
        // read message
        ApplicationStatusChange message =
    netMsg.ReadMessage<ApplicationStatusChange>();

        if (message.pinStarted) {
            // need to start pin
            if (message.guidePin) {
                // is guide pin
                this.guidePinManager.startPin(message.internalPinNumber, true);
            } else {
                // is pilot pin
                this.pilotPinManager.startPin(message.internalPinNumber, true);
            }
        }
    }
    // ...
}
```

Once verified in the network communication, the client is the target of sharing changes in the application status. The server will send a `ApplicationStatusChangeMessage` if the controller or instructor activated a pin. When this message is received, it will be processed in the corresponding handler which will read the message, interpret which pin got activated and start it locally as well. If the client device is a controller or instructor, the pin managers will further send another `ApplicationStatusChangeMessage` once the pin is completed.

```
public class ClientDevice : MonoBehaviour {
    // ...
    public void OnDisconnect(NetworkMessage netMsg) {
        // stop sending device-transformation
        CancelInvoke("sendDeviceTransformationToServer");

        // calling OnNetworkDisconnect of INetworkAffected-scripts (...)

        // resetting device-status
        this.roleVerified = false;
        this.serverFoundViaBroadcast = false;
        // try to find new server
        Invoke("searchAndConnectServer", 3f);
    }
    // ...
}
```

If the connection to the server gets lost – no matter if network issue, stopping the application or shutdown of the server – the continuing transformation sharing to the server is stopped and all network-affected scripts are called which refresh the local permissions (see section 4.5.3). After this the network status gets reset by undoing permission members and searching a new server.

Chapter 5

Spatial References

Designing and implementing network behaviors (see chapter 4) made it possible to let applications on multiple devices run in a synchronized manner: The state on every device is the same and once a user with the permission activates a pin, the `ClientDevice` script and its cooperating scripts activate the pin locally and send the data to the server which forwards the information to all other devices that will start the pin as well.

This was one of two essential tasks to create a holographic application running virtual objects in the real world just as if they were really there. But until the completion of this second task which is discussed, conceived and implemented in this chapter, the 3D positioning of the scene depends on the position of the device when the application is started.

5.1 Requirements and Options

Mapping the application for every user to the same place, for instance on a table, depends on multiple factors such as

- tracking method,
- mapping method and
- referencing.

These factors are not always separable but often have direct effects on other factors.

The *tracking method* describes how devices positions and orientations are tracked and measured in a surrounding *mapped* either automatically due to spatial requirements or accept variable surroundings which are interpreted in real-time. Once the transformation of the device and the required properties of the surroundings are defined, a *reference* is needed to define the position and orientation of the holographic content.

5.1.1 Absolute and Relative Tracking

To correctly display the virtual content, not only the position and orientation of the content, designed in a virtual space, but the relation between virtual positioning and reality is important. Interpreting the position of a device in space can be either done in a limited or unlimited space.

Absolute Tracking in a Limited Space

In an enclosed area, it is possible to track a device absolutely in space. A common method in installations is the use of *HTC Vive trackers*, an accessory of the *HTC Vive* ecosystem further known as *SteamVR*.

The fist-sized trackers can be attached to any object and are tracked inside the *play area* which is defined by the positioning of the *light houses* emitting a timed laser grid. The timing how often a sensor on the tracker is hit by the moving laser plane makes it possible to calculate the movement every update in both axis by multiple light houses (see figure 5.1), resulting in a very precise tracking [3]. The use of multiple sensors on a tracker adds as well information about the orientation of the tracker.

Although the limitation in space would not be a problem in this use-case, it is a limitation which can be removed by using a relative tracking method offering the use of bigger holograms or even collections of holograms in combination with mapping data. Further, the use of *HTC Vive trackers* is not suitable if the users want to use their own devices which would need additional preparation time for mounting the tracker and an own system processing the tracking data.

Absolute tracking is very precise and useful for static installations with prepared devices offered to use by the visitors but introduces hardware requirements connected with additional costs and work, and is limited in space.

Relative Tracking

Relative tracking usually uses the initialization position as origin. Technologies such as the *Microsoft HoloLens* and *Google*'s *ARCore* offer this tracking method with their embedded hardware.

Both of the AR systems use their accelerometers, motion sensors and in case of *ARCore* visual input to track their position and map the surrounding (further details in section 5.1.2) and cross-use this data for enhanced stabilization of the tracking (for general information see chapter 2).

Positioning a virtual object at the coordinate point `P(0|0|3)` in *Unity* would result into a positioning three meters in front of the user once the application is initialized.

Relative tracking is an automatic working method if the devices support it. The quality of the hardware and software solutions result in varying tracking quality. Today's Augmented Reality solution offer surprisingly exact tracking and stabilization in most cases. While *Microsoft's HoloLens* has no troubles in poor lighting conditions and is nearly independent of mapping information, *Google's ARCore* depends on cross-referenced mapping data which is as well used for tracking stabilization due to the mainly visual tracking and its downsides such as camera blur on fast movement.

5.1.2 Static and Dynamic Mapping

The term *mapping* in this context is used for describing how the data of the real surrounding can be converted in data-sets which can be further used for referencing (further details in section 5.1.3) or features based on virtuality (see section 2.1).

Figure 5.1: The tracking of *SteamVR* by *HTC* and *Valve* works with moving laser planes emitted by *lighthouses*. Depending on the timing of laser hits on the sensors, the position of a sensor in the room can be calculated. Using multiple sensors on a tracking device, such as the *HTC Vive trackers*, allow orientation tracking as well [14].

Static Mapping

Static mapping is the easiest method to link the virtual content and reality's surroundings together but it only works with static installations using absolute tracking (see section 5.1.1).

The idea behind this mapping technique is to set up the real scenery in the *play area* (the tracked area) precisely dependent on the origin of the play area. This real scenery is directly implemented in the application, so all of the information of the objects in the application area are saved in the software already which can be used for physics simulations, advanced virtual lighting and further virtuality features.

In the case of using Augmented Reality devices such as the *Microsoft HoloLens* and *Google*'s *ARCore*, this mapping method is not usable since it is using relative tracking which means the origin of the virtual content does not match with the origin of the device tracking. This problem could be solved with physical initialization positions – for instance offered devices for the presentation have position-specified holders and the application is started in them, so the origin of the virtuality matches with the origin of the tracking. Further, could such physical references be used for resetting origins using *Near Field Communication (NFC)*; the user starts the application and puts the used device in a physical holder with an NFC chip which sets a new origin for the tracking, so the origins match.

Although these workarounds exist and work for closed area installations, the quality depends on the precision used to embed the real world in the virtuality. Further, tracking problems would offset the origin matching again which would force the user to once again go to the physical reference and re-calibrate.

Dynamic Mapping

Dynamic mapping is using sensors and *Computer Vision* techniques to interpret the surroundings in real-time. This method is becoming more common in the last years since the processing power of devices is increasing exponentially. It is not possible to completely 3D scan the environment out of the view of the user but today's technology does already a good job interpreting obvious geometry in real-time.

The *HoloLens* uses multiple distance sensors and cameras to scan the reality. The sensor data results in a 3D grid of the objects in front and to the side of the wearer comparable to throwing a blanket over the objects (see figure 5.2). The precision of this interpretation is limit to the resolution of the sensors and the processing power of *Microsoft*'s wearable smart glasses. The system of the *HoloLens* is able to detect most surfaces, understand whole objects with additional algorithms and uses the information for features such as occlusion.

Augmented Reality solutions running on the smartphone such as *ARCore* by *Google* and *Apple*'s *ARKit* use the renderings of the camera for mapping objects in reality. This is a much more complicated approach but does not require additional hardware. The quality of the mapping is directly connected camera specifications such as resolution, luminous sensitivity and blur due to aperture and shutter timing. *Google*'s and *Apple*'s algorithms are comparable with minor differences in results: Both are able to detect flat surfaces (horizontal and vertical) as planes which are not limited in size by real edges but rather work like tangents.

In general, dynamic mapping is and will never be as precise as static mapping but is not limited to certain handled scenes and works dynamically in every space which meets the conditions to have a qualitative mapping/detection.

5.1.3 Referencing

This section describes the possibilities to position and orientate virtual objects in the three-dimensional AR space depending on tracking references, geometric mapping references or visual references.

Relative Coordinate References

With coordinate references relative to the position or initialization point of the tracked device, it is possible to position virtual objects static or semi-dynamic in a very easy but as well feature-weak manner.

Using this method, tasks such as "on application start, position the hologram two meters in front of the viewer, 50 centimeters lower than the user's eyes" or "when the viewer looks down, show debugging information around the user" are easily implementable but any relations to the reality are not possible (except with preset static mapping information or in general, when the relation between virtuality and reality is known by using for instance a physical origin as described in section 5.1.2).

Geometric References

Geometric references (no matter if static pre-coded mapping or dynamic interpretation of the reality) are used to position and orientate augmented objects in relation to surfaces

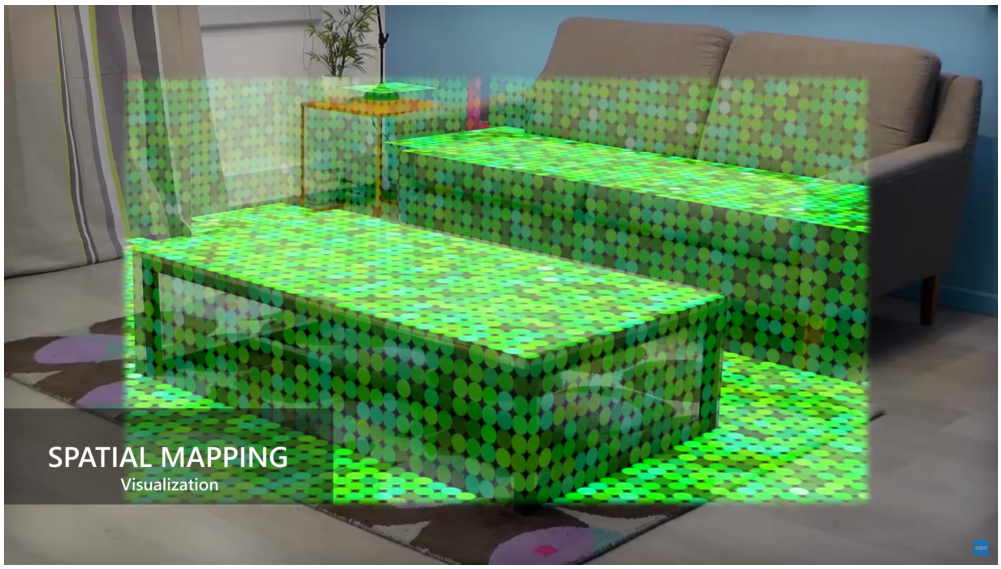


Figure 5.2: Devices such as the *Microsoft HoloLens* use distance sensors to map a 3D grid of the environment which' data can be used for physics simulations, path-finding, occlusion and further virtuality features [16].

of geometry data.

The *HoloLens* builds a full 3D grid of mapped objects which can be used to reference transformations anywhere in relation. Tasks such as "place the map on a table (flat horizontal surface) with a length between two and three meters and a width of one to one and a half meters" are natively possible by accessing the mapping information of the *HoloLens* in real-time.

For object-dependent placement, additional algorithms such as *CurvSurf*'s *Find Surface* (more in chapter 1) might be needed for tasks such as "place a virtual sphere (red, 10 centimeters in diameter) on top of all detected spheres with a diameter between 30 and 50 centimeter".

ARCore can detect flat surfaces such as tabletops and walls as well but does not detect edges which means the plane is not limited to the border of, for instance, a table. Further, the detection of bigger planes is much slower than on the *HoloLens* since the smartphone solutions need more movement for detection. Anyhow, positioning on such detected planes via touch is simple and intuitive.

Visual References

Visual references are detected by using visual sensors such as cameras. The detection and conversion of visual data representations such as QR codes are already possible for years but recent *Computer Vision* techniques make it possible to calculate the position and orientation of images in three-dimensional space if the size of the actual image is known.

ARCore supports multiple image recognition since version 1.2. On detection of an image stored in the (custom) database, an event is triggered and the origin of the image

is available in the virtuality which makes positioning related to the detected image possible.

Microsoft's *HoloLens* does not natively support image recognition but compatible frameworks such as *Vuforia* [17] support image recognition using a similar method as *ARCore* uses.

5.2 Concept

The goal of the spatial references is to synchronize the visual content of the application running on multiple devices in space. This task can be done in many ways as already indicated in section 5.1 but this should be done to be easy for the user without attracting too much attention to the references themselves. For the defined use-case, it was planned to place a map on a table, the users go towards the table and until they arrive at their desired viewing position, the synchronization – both the communication and spatial aspects – should be finished.

In the described concept using static tracking is only possible when using prepared devices which rest in a mapped region with the relation to the target space known. Since the users should be able to use their own devices as well if they wish to do so, static mapping in combination with the relative tracking the *HoloLens* and *ARCore* offer is not possible due to the fact that the origin (initialization point) of the devices is unknown.

Using geometric references of the dynamic mapping of the *HoloLens* would theoretically work – for instance, defining the size of the table in the software and the system is able to detect it – but purely depends on the positioning of the user. If the table is not fully detected, the software might not identify it and create exceptions. Further, the use of geometric references would not work on the *ARCore* platform since the mapping is rather simple and is not limited by the borders of the tabletop.

So probably the most suitable method would be the usage of the relative tracking and a visual indicator for image recognition. The marker should not be clearly identified as one to not attract the attention but still be valuable enough to be used for the referencing. For this use-case, the marker (a 30 by 16 centimeter big logo of the *Red Bull Air Race*) will be stuck to the tabletop in a horizontal alignment (see figure 5.3) to keep the positioning simple, although an evaluation in section 6.2.2 showed that vertical alignment is positively affecting the recognition distance. To detect the marker, *Vuforia* is required on the *HoloLens*; *ARCore* offers a natively included image recognition.

For possible adjustments in using certain frameworks, an evaluation of the two systems regarding image recognition was completed in section 6.2.2 showing that *Vuforia*'s image recognition works on greater distances and therefore do not need that much attention. The usage of *Vuforia* in combination with *ARCore* was tested without success due to manifest merging exceptions because of incompatibility at this point of time. A complete switch to *Vuforia* instead of *ARCore* might have been a possibility but it was not manageable in time to rebuild the whole application in the progress of this master's thesis.

Figure 5.3: The reality setup of the spatial referencing is sticking a 30 by 16 centimeter big *Red Bull Air Race* logo flat on the tabletop. Evaluations in section 6.2.2 showed that a vertical alignment would be increasing the detection range but a horizontal orientation is for test cases easier to recreate.

5.3 Implementation

As mentioned in section 5.2, two different systems are used for the image recognition used to get a spatial reference for positioning the scene in 3D space. For the *HoloLens*, the system *Vuforia* is used which is capable of image recognition and would even offer object detection. Sadly, *Vuforia* cannot be combined with *ARCore* since to incompatibilities but *Google* offers since version 1.2 an own image recognition algorithm natively implemented in *ARCore*.

Although these two systems work with similar approaches, the development for using them is very different. *ARCore*'s native image recognition offers very open development methods such as using own script for further processing, while *PTC* (the developers of *Vuforia*) keeps their system rather closed. In the case of *Vuforia*, deeper processing is only possible by changing deeper *Vuforia* aspects. Since some behavior of *Vuforia* is not perfect for this use-case and fully understanding the system and how to interrupt or manipulate it is barely manageable without huge time investments, some additional tasks could not be implemented in an optimized manner.

5.3.1 Vuforia on the HoloLens

About Vuforia and Licensing

Vuforia is an Augmented Reality system made by *PTC* offering advanced Computer Vision technologies. Additionally to the required image recognition in 3D space, *Vuforia* is capable of detecting multiple images at the same time – either flat images, on cuboids or cylinders – or even 3D objects in reality.

PTC offers one-time paid licenses which grant publication rights, cloud licenses extend the classic license with target databases synchronized in their cloud or for even more sophisticated projects, advanced licensing with even better visual knowledge embedded software. For personal or research use with basic functionality – as needed in this project – *Vuforia* is free to use with a limitation in actions per month which is more than high enough for this project.

Receiving a development license is done via *PTC*'s online platform in the developer section; only the registration of a free account is needed. After that, the developer has access to request a license for a specified app. The free license is limited to 1000 cloud recognitions per month and is only applicable for personal or research use. The about 200 character long license string is needed later in *Unity* to activate *Vuforia*'s features.

Databases and Targets

The creation and management of *databases* containing the collection of *targets* (references of either flat images, cuboids, cylinders or 3D models) is done on their online platform as well. Every target needs the visual reference and the width of the reference used in reality; this information is needed for the depth perception.

Once the database is completed for its use, it can be downloaded for their own development platform or for the use in *Unity* which outputs a custom `unitypackage` file. This file is later imported into the *Unity* project and automatically embedded in the *Vuforia* system.

Basic Usage of Vuforia in Unity

To include *Vuforia*'s features in the *Unity* package, the *Vuforia* package of the *Unity Asset Store* needs to be downloaded and imported. For general usage, the plugin *Vuforia Core SDK* is recommended but *PTC* created as well an asset called *Vuforia HoloLens Sample* which already offers a configuration prepared and optimized for the usage on the *Microsoft HoloLens*.

Once *Unity* is set up, the script `Vuforia Behaviour` is added to the `First Person Camera` game object to enable *Vuforia* features. The script itself offers to set the *World Center Mode* to `DEVICE` (using the device's initialization as the scene's origin), `FIRST_TARGET` (shifting the whole scene to the first detected anchor) or `SPECIFIED_TARGET` (shifting the whole scene to a defined anchor on detection). For the desired usage, `DEVICE` would be the recommended since we need the change of position for detecting certain events (see next section).

The linked `VuforiaConfiguration` represents all possible behaviors which are possible without changing the code of *Vuforia*. When using the *Vuforia HoloLens Sample*, all the settings are already optimized for the use with the *HoloLens* such as the definition of the Device Type as *Digital Eyewear* and a complete Device Configuration for the *HoloLens*. The imported *Vuforia Target Database* should be shown in the section *Databases* as well. Further configuration parameters allow setting up *extended tracking* as well as a simple debugger mode called *Vuforia Play Mode* which allows using a webcam for fast debugging tests without deploying the project to the *HoloLens* or other target devices.

The last needed task is to manage the targets in virtuality. In the example of the

use-case, every visual content (children of the game object `Las Vegas`) should be moved to the recognized position of the image. To implement this behavior, a new empty game object called `ImgRecogTarget` was created which is the new parent of the game object `Las Vegas`.

To set this new game object as the target for the image recognition, the script `Image Target Behaviour` needs to be added, offering settings for the detection. The type selection offers the values *Predefined* (using a specific target which cannot be changed at run-time), *User Defined* (using a specific target by name; offers changes at run-time) and *Cloud Reco* (allows using a database in the cloud). When selecting *Predefined* which is enough for the use, the database and image target need to be selected.

Further, the `Image Target Behaviour` script automatically adds a `Mesh Filter` and `Mesh Renderer` to the game object. Since the visualization of the recognition is not necessary, the `Mesh Renderer` was disabled. It is important to notice that the game object with the applied `Image Target Behaviour` script is automatically scaled to the size of the image which should be recognized, leading to a scaling of the children (the whole visual scene) as well. To prohibit this scaling, *Vuforia* offers an option in the *Advanced* section of the target behavior which is preserving the scaling of the children. Another way to solve this problem is to add another game object between the `ImgRecogTarget` and `Las Vegas` which has an inverted scaling – for instance: scaling of the target is 0.3, so the scaling of the compensator is 3.333333.

These short steps are already enough to position the scene to a recognized image reference but this does not lead to a good result since important features are not implemented in the behavior yet. There is no simple way to access an event which is triggered on detection to, for instance, hide certain objects before the detection and show them afterward as well as limitations in the target transformation are not possible.

Extending or Changing Vuforia's Behavior

Possibilities to extend or change the behavior of *Vuforia* are natively limited since there is no possibility to add in own code without breaking open the scripts which would be possibly lost progress on updates of the system. But it is known what *Vuforia* does on the image recognition in the `DEVICE` mode of the *World Center Mode*: It moves the image target game object to a new position.

A way to intrude into *Vuforia*'s actions would be scripts which monitor the movement of image target game objects. Such a script could be added to the game object which is the target of the transformation and extend the functionality or restrict *Vuforia*:

```
public class OnVuforia : MonoBehaviour {
    protected Vector3 startingPosition;
    // GUI-related members (...)
    protected GameObject welcomePinGO;
    // further location based objects which need actions (...)

    void Start() {
        this.startingPosition = this.gameObject.transform.position;
        this.BeforeReference();
    }
```

```
    void Update() {
        if (this.gameObject.transform.position != this.startingPosition) {
            if (this.isHorizontalX() && isHorizontalZ()) {
                this.OnReference();
            }
        }
    }

    protected bool isHorizontalX() {
        float angle = this.gameObject.transform.localEulerAngles.x % 360;
        return angle < 2f || angle > 358f;
    }

    // same method for z-axis (...)

    protected void BeforeReference() {
        foreach (
            Renderer renderer in
            this.welcomePinGO.GetComponentsInChildren<Renderer>()
        ) {
            renderer.enabled = false;
        }
    }

    protected void OnReference() {
        this.trackingStatusGO.SetActive(false);

        // show first pin
        foreach (
            Renderer renderer in
            this.welcomePinGO.GetComponentsInChildren<Renderer>()
        ) {
            renderer.enabled = true;
        }

        // disable recognition
        this.gameObject.GetComponent<ImageTargetBehaviour>().enabled = false;

        // stabilize horizontal positioning
        this.gameObject.transform.localEulerAngles =
            new Vector3(0f, this.gameObject.transform.localEulerAngles.y, 0f);

        // update absolute positioned content (...)
    }
}
```

In the initialization progress of the script, it stores the starting position of the target game object in the member `Vector3 startingPosition` and executes the method `BeforeReference()` which disables all renderers of the first pin (called `WelcomePinGO`) to have all content hidden before the detection.

The method `Update()` is the core functionality of this `OnVuforia` script. It checks if the game object which is the image target changed its position since the beginning. If it did, *Vuforia* detected the image and adjusted the position. Since *Vuforia* can even detect the angle of the image which is very intolerant for small changes but the content should be always parallel to the ground, the script checks if the angle is smaller than two

degrees in the `x`- and `z`-axis which indicates a correct track. Once the image is tracked and the tracked rotation is in the tolerance, the method `OnReference()` is called which activates the renderer of the first pin again, disables *Vuforia*'s image recognition, flat the content out and repositions absolute positioned content.

Using this method, the tracking is not running all the time which is optimal performance-wise but means once the image was tracked with enough precision, the scene will stay in the referenced status; moving the whole scene by moving the image tracker would not work which is not needed for this use-case but it should be kept in mind that an application with moving trackers could not be stabilized like this.

5.3.2 ARCore

Basics and Target Database

ARCore is as well as *Vuforia* included using a package from the *Unity Asset Store*. *Google* did a great job in already including some examples of how to use *ARCore* including its image recognition and offers a lot of prefabs for a fast and easy setup.

Unlike *Google*'s prefab *ARCore Device* suggests, the *First Person Camera* must not be parented by another game object since this leads to a wrong positioning of the detection. *ARCore* seems to move the parent object around and relates the local position with the scene instead of the global one. When using the prefab, the script `ARCoreSession` needs to be copied to the camera and the parent object removed. The script is the core of *ARCore* and has a session configuration linked to it. The configuration allows to set the *plane finding mode* to either horizontal, vertical or both – horizontal is enough for the use-case – enable features such as light estimation and offers to preload already an *Augmented Image Database*.

The creation of an Augmented Image Database for *ARCore* is greyed out for *Unity 2018.1.0f2*: Free Edition, meaning either this is only possible on *Pro* versions of *Unity* or it is simply not fully implemented in this version. But there is already an example database in *Google*'s examples which can be changed accordingly to the needs. Just as in using *Vuforia*, a target needs a visual reference and the width of the image in reality.

Further, the First Person Camera needs an `ARCoreBackgroundRenderer` for rendering the camera frames and a `TrackedPoseDriver` which makes further configurations such as device, tracking type or changes of *update-render-correlation* possible. All of the above steps, except the creation or edit of the database, are done automatically when using the prefab `ARCore Device`.

Image Recognition

Unlike *Vuforia*, *ARCore* is not automatically placing objects parented by a target virtuality object but need a custom controller script for actions. This may sound more complicated but allows much deeper behavior control than *Vuforia* natively supports. For this reason, a script called `OnImageRecognition` was created and added to an empty game object called `ImgRecogController`:

```
public class OnImageRecognition : MonoBehaviour {
    protected GameObject targetGO;
    protected GameObject welcomePinGO;
```

```
    protected bool correctlyPositioned = false;
    private List<AugmentedImage> m_TempAugmentedImages = new List<AugmentedImage>();

  void Start () {
        this.BeforeReference();
    }

  void Update () {
        if (this.correctlyPositioned) return;
        if (Session.Status != SessionStatus.Tracking) return;
        Session.GetTrackables<AugmentedImage>(
            this.m_TempAugmentedImages, TrackableQueryFilter.Updated);

        foreach (var image in this.m_TempAugmentedImages) {
            if (image.TrackingState == TrackingState.Tracking) {
                Anchor anchor = image.CreateAnchor(image.CenterPose);
                if (this.isHorizontalX(anchor) && this.isHorizontalZ(anchor)) {
                    this.OnReference(anchor);
                    this.correctlyPositioned = true;
                }
            }
        }
  }

    protected bool isHorizontalX(Anchor anchor) {
        float angle = anchor.transform.eulerAngles.x % 360;
        return angle < 2f || angle > 358f;
    }

    // same method for z-axis (...)

    protected void BeforeReference() {
        foreach (
            Renderer renderer in
            this.welcomePinGO.GetComponentsInChildren<Renderer>()
        ) {
            renderer.enabled = false;
        }
    }

    protected void OnReference(Anchor anchor) {
        this.targetGO.transform.position = anchor.transform.position;
        this.targetGO.transform.eulerAngles =
            new Vector3(0f, anchor.transform.eulerAngles.y, 0f);
        foreach (
            Renderer renderer in
            this.welcomePinGO.GetComponentsInChildren<Renderer>()
        ) {
            renderer.enabled = true;
        }
    }
}
```

Unlike in the *Vuforia* build, it is not necessary to save the initial position of the scene since the transformation of the content is done with the custom controller script instead of relying on the automatic process of *Vuforia*. As done in the *HoloLens* script,

the method `BeforeReference()` which is called on the initialization of the application is used for disabling all renderers of the first pin to have all content hidden before the detection.

Again, the `Update()` method represents the main functionality of the script. As long as the content was not correctly positioned and the tracking is enabled, the method checks for updated data of the trackables. If updated data is found and the tracking state is still the current state, the result of the query is used for generating an anchor which represents the origin of the tracked image. Since the content should always be horizontal, the algorithm for checking the angle around the `x`- and `z`-axis as used in the *Vuforia* build is used to validate the tracking interpretation. The method `OnReference()` is called once the image was tracked and the orientation of the anchor was validated.

The `OnReference()` method sets the transformation of the game object `ImgRecogTarget` which parents `Las Vegas` (the visual content of the application) – which is set as the `targetGO` (target game object) – to the transformation of the resulted anchor. To trace the rotation of the image, the angle of the anchor around the `y`-axis is as well set on the target. In the end, the rendering of the first pin is again enabled and the `Update()` method stops the tracking by setting the member `bool correctlyPositioned` to `true`.

Chapter 6

Evaluation

This chapter contains method description, measurable values and results of all done evaluations for network communication and spatial referencing (see chapters 4 and 5).

6.1 Network Communication

6.1.1 Closed Network

The goal of this evaluation is to find out the amount of delay between the activation of a pin on an instructor device, the receipt of the network message on the server and activation of the pin on another synchronized device. This delay should be hardly distinguishable when multiple users talk about a seen content; to have a practical value: A value below 1,000 milliseconds is needed, below 300 milliseconds is desirable. More important than the actual value is the correlation between the use of a closed network (a network only the users are connected to) and an open network (the system is running in a network that is already in use with other tasks; compare section 6.1.2).

Method

The system is running in a closed network, so only the users of the application are within the network and no further demanding network traffic is active. For this evaluation, a mobile hotspot is created with a *Samsung Galaxy S8* which is executing the instructor application (*ARCore* version). The *Microsoft HoloLens* is running the controller application (*HoloLens* version) and a laptop (*Windows 10*, *Intel i7* core) acts as the server application in the network. The instructor will activate a pin using the instructor button seven times for a valid average result.

Measurement

To be independent of possible differences in local timestamps, the delay is measured visually using an action-camera (*Apeman Trawo*) recording the screens of the devices in 120 fps (frames per second). The screens of all devices have a refresh rate of 60 fps resulting in a tolerance of $^{+0}_{-16}$ `ms`. The footage is interpreted frame by frame in *VLC player*, using the plugin *Jump to time v.2.1* for distinguishing the timing.

Figure 6.1: The activation on the instructor device (*Samsung Galaxy S8*), receipt on the server (laptop) and remote activation of the pin on the *HoloLens* are recorded using an action cam with 120 frames per second to be independent of the device's timestamps.

Footage

The recordings of the runs (`Run#.MP4`) as well as screenshots of the key frames (`Run#-Activation.PNG`, `Run#Server.PNG` and `Run#Client.PNG`) for calculating the delay are saved on the DVD in the folder `Evaluation/Network/Closed`.

Conclusion

The tests in the closed network, only used for this application without additional traffic, showed that the delay between activating a pin on a controller- or instructor-device and having the pin activated on another device using the network messaging is usually a little below 100 milliseconds (see figure 6.2). These results were expectable since, in earlier tests without measurement, the resulted output of the narrator's voice on multiple devices sounded like an echo.

The results, especially *Actor* → *Server*, may vary since the visualization of the activation is sometimes slower or harder to interpret. In run 4 the visualization was even slower than the sending of the message which led to an invalid measurement.

The evaluation is satisfactory but the more important evaluation is the comparison to an open network with other traffic if additional network usage would increase the delay which is presented in the following section.

6.1.2 Open Network

Goal

The goal of this evaluation is the same as the goal of the evaluation within a closed network (described in section 6.1.1). Further, this evaluation is additionally done to find out if additional network traffic is increasing the delay of the network synchronization.

Method

The system is running in an open network, so the network is not only for the system but is used by others for different (demanding) tasks as well. For this evaluation, my home-network is used which consists of multiple access points and terminals. Again, a *Samsung*

Run	Actor → Server	Server → Client	Actor → Client
1	16ms	25ms	41ms
2	42ms	50ms	92ms
3	57ms	49ms	106ms
4	invalid measurement		
5	42ms	59ms	101ms
6	25ms	42ms	67ms
7	91ms	59ms	150ms

Figure 6.2: The results of the tests regarding the delay in a closed network. The measured time is based on a 120fps recording (approx. 8 milliseconds per frame). The screens of the devices have a refresh rate of 60 frames per second, resulting in a tolerance of $^{+0}_{-16}$ milliseconds between measurement and action.

Galaxy S8 is used as an instructor but this time with simultaneous music streaming using *Spotify* and the laptop acting as server is watching *YouTube* videos. Additionally, another device within the network will copy data in the Local Area Network.

Measurement

The measurement method is the same as the one for the closed network evaluation (see section 6.1.1).

Footage

The recordings of the runs (`Run#.MP4`) as well as screenshots of the key frames (`Run#-Activation.PNG`, `Run#Server.PNG` and `Run#Client.PNG`) for calculating the delay are saved on the DVD in the folder `Evaluation/Network/Open`.

Conclusion and Comparison

In general, the tests in the heavily used open network show only small differences, with the average values around 100 to 120 milliseconds (see figure 6.3). It is hard to tell if this is due to the network traffic since even in the closed network the devices could use additional data because it is still connected to the internet.

In any case, it is a pleasant result that in bigger networks and heavy workloads within it there is barely any higher delay, probably due to small package sizes.

6.1.3 Synchronization Exceptions

Goal

On connection loss, the application stops in the state it is in and all permissions granted by accepting the role should be removed. This behavior is important to avoid failures in synchronization.

Run	Actor → Server	Server → Client	Actor → Client
1	58ms	67ms	125ms
2	16ms	151ms	167ms
3	42ms	17ms	59ms
4	82ms	33ms	115ms
5	50ms	33ms	83ms
6	25ms	41ms	66ms
7	58ms	67ms	127ms

Figure 6.3: The results of the tests regarding the delay in an open network. The measured time is based on a 120fps recording (approx. 8 milliseconds per frame). The screens of the devices have a refresh rate of 60 frames per second, resulting in a tolerance of $^{+0}_{-16}$ milliseconds between measurement and action.

Method

To cover the target of this thesis project, the application is run with three devices involved: A laptop which will execute the server application, the *HoloLens* is taking control over the application with the controller role and a *Samsung Galaxy S8* will run the instructor application. This test case is done multiple times, each time a device will be disconnected from the network and the resulting behavior is examined.

Results and Conclusion

Disconnect of Controller: On the disconnection of the controller, the controller is not able to activate any pins locally and a floating message "Reconnecting..." appears. Server-sided, the disconnect is logged and the controller role is free to use again. Once, the connection was built up again, the controller connected with the server, received the permissions and was able to control the application again.

Disconnect of Instructor: If the connection of the instructor fails, the access to the instructor buttons is denied (greyed out) and a message "Reconnecting..." appears. The server-sided behavior is the same as in the controller case, as well as the reconnection works fine.

Disconnect of Server: Once the server disconnects, all connections are logged to be closed after five seconds (due to the timeout). The controller and instructor lose all their permissions and wait for a reconnect. Once the server application is restarted, both client devices were able to reconnect automatically including granting their authorizations.

6.1.4 Further Interesting Evaluations

With the means available, not every test is possible to execute. An interesting evaluation would be how the amount of connected users affects the delay between the receipt of the network message on the server, activation of the pin on another synchronized device and the difference between the first and last connected device. For the sake of the thesis, this evaluation is not necessary because of the goal of a system that can be used to support

the wearer of the *HoloLens* using *ARCore*. The spectator-role was an additional idea for opening up a session to multiple users which' optimization would not be the core of this thesis.

6.2 Spatial Evaluation

6.2.1 Tracking Stability on Movement

Goal

This evaluation should give an impression about the stability of the tracking and mapping of the two used technologies, the *Microsoft HoloLens* and *ARCore*. Although the results of this evaluation are not validating any success of the Master's thesis surroundings, this technical information can be interesting for the choice of technology in future applications as well as it gives additional related information to the evaluation of the space synchronization (see section 6.2.3).

Method and Measurement

The method and measurement of this evaluation are the same for both technologies: The image marker used to position the scene is stuck to a tabletop as it would in the real execution of the application. The devices will recognize the image until a stable detection and transformation was done. A piece of paper with a printed 2D ruler is placed with its origin at the bottom left corner of the virtual map. Once the preparation is aligned, the user will walk around and look at the map from different angles, just as a viewer would. After approximately half a minute of movement, the same corner of the map is marked on the piece of paper, resulting in the displacement of the virtual coordinate system showing the stability of the tracking. This procedure will be done five times to see possible patterns and usual results.

Measurement Tolerance

Due to the visual tracking used in *ARCore*, it is not possible to come as close to the bottom left corner of the map as it is with the *HoloLens* since *ARCore* would lose its tracking because there are not enough valuable tracking references in the rendering. Especially on the *ARCore* evaluation, the measurement of the coordinate system shift will be measured from a further distance, leading to a less precise result. For both systems, the resulting points will be shown as circles which are bigger the less precision is possible on the measurement due to shifting of the holograms or other issues.

Conclusion

The results (see figure 6.4) show that the *HoloLens* has a very high level of precision in tracking and again detecting its references. The *HoloLens* stayed in all five runs below the one-centimeter mark of tracking errors on movement. *ARCore*'s results of usually up to three centimeters seems reasonable and are still satisfying for a visual system and its low (or non-existent) price point. On visual checks, changes in the angle of the map

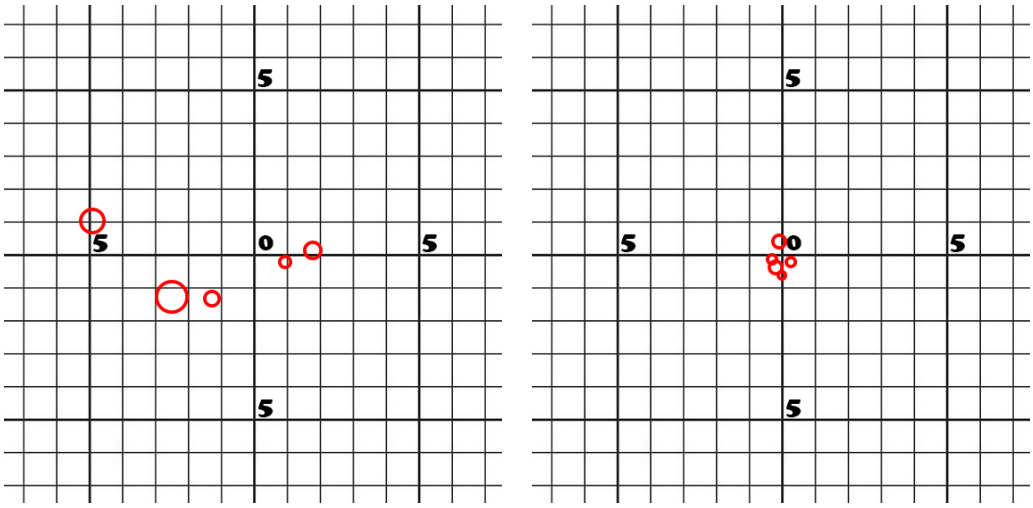


Figure 6.4: The results of calibrating the position of the map, half a minute of usual viewing movement and measuring the position difference in *ARCore* (left image) and on the *HoloLens* (right image).

were very slight or not recognizable on both systems. The runs showed a tendency to drift the content rather to the left and closer to the viewer, especially for *ARCore*, which will be confirmed in section 6.2.3.

6.2.2 Image Recognition

Goal

Depending on the size and orientation of the physical image marker which should be recognized, the user has to go in direction of it, have it in sight and in the best case it is not needed to directly look at it to get the reference of it. This evaluation is done with *ARCore*'s native image recognition on and on the *HoloLens* using *Vuforia*.

Method

The physical image is placed on a table with a light background. The user will come closer to the table until the image is recognized and a simple augmented object is placed above it. To learn more about the result differences depending on size and orientation, in one test case the image is rather small (15 by 8 centimeters) followed by another test with a bigger image (30 by 16 centimeters) and last but not least the bigger image is stuck to a standing card box, oriented for a clear flat rendering instead of a warped image when used horizontally.

Measurement: ARCore

The evaluation values are the direct distance to the origin of the image at the first recognition; the higher the values the better for a clean recognition. The values are directly measured in a test application using the following method:

ARCore uses a given controller script which defines what to do on the recognition, on the first recognition the distance between the image detection's resulting anchor and the camera object is measured and printed on the screen.

```
public class AugmentedImageExampleController : MonoBehaviour {
    // ...
    public void Update() {
        if (Session.Status != SessionStatus.Tracking) { return; }

        // Get updated augmented images for this frame.
        Session.GetTrackables<AugmentedImage>(
            m_TempAugmentedImages, TrackableQueryFilter.Updated);

        foreach (var image in m_TempAugmentedImages) {
            if (image.TrackingState == TrackingState.Tracking) {
                // get anchor and position object above it (...)

                if (!this.DistanceInformationGO.activeSelf) {
                    this.DistanceInformationGO.SetActive(true);
                    this.DistanceInformationGO.GetComponent<Text>().text =
                        "Distance: " +
                        Mathf.Floor(Vector3.Distance(
                            this.lastAnchorPosition,
                            this.CameraGO.transform.localPosition
                        ) * 100)
                        + "cm";
                }
            }
        }
    }
}
```

Measurement: Vuforia

Vuforia is a closed system which does not allow to interrupt or extend the detection process without deeper intrusion. To display the distance to the detected image, an additional script is continuously checking the position of the object which represents the image in virtuality. Once the position of the object changes, *Vuforia* detected the image and transforms the recognition object to the reference in reality. The moment this transformation is detected, the script displays the distance between the resulting position and the camera, just as done in the *ARCore* evaluation.

```
public class DistanceInformationDebugger : MonoBehaviour {

    private Vector3 startingPosition;
    public GameObject CameraGO;
    public GameObject DistanceInformationGO;
    private bool changed = false;

    void Start () {
        this.startingPosition = this.gameObject.transform.localPosition;
    }
```

```
    void Update () {
        if (!this.changed) {
            if (this.startingPosition != this.gameObject.transform.localPosition) {
                this.changed = true;
                this.DistanceInformationGO.SetActive(true);

                this.DistanceInformationGO.GetComponent<TextMesh>().text =
                    "Distance: " +
                    Mathf.Floor(Vector3.Distance(
                        this.gameObject.transform.localPosition,
                        this.CameraGO.transform.localPosition
                    ) * 100)
                    + "cm";
            }
        }
    }
}
```

Footage

The recordings of the runs (`Run#.MP4`) for the *ARCore* evaluation are stored on the DVD in the folder `Evaluation/ARCore Image Recognition/` containing the folders `15cm`, `30cm` and `30cm Vertical` for the corresponding test cases. The same folder structure exists for the *Vuforia* evaluation in the folder `Evaluation/Vuforia Image Recognition/`.

It is important to notice that the renderings of the *HoloLens* look like *Vuforia* did not perfectly recognize the position of the reference. Looking through the glasses, the application was always fitting perfectly; the difference is created by *Microsoft*'s Mixed Reality Capture which seems not to perfectly align reality and virtuality.

Conclusion

In general, figure 6.5 shows that *Vuforia* seems to be faster in detecting images (about double the distance), even at steeper angles and therefore more warped images. For both frameworks, variation in size results in slight changes in recognition distance but the image is still warped when approaching the table, making the detection harder. On the other hand, orienting the image vertical – by, for instance, sticking it to a wall – makes it possible to recognize the image from a further distance; *Vuforia* is capable of recognizing it from about three times further than *ARCore*. Further, *ARCore* is not as good in mapping vertical objects as in mapping horizontal objects which results in detecting the image sometimes faster than the wall it is stuck to. This could lead to a wrong positioning in the first place which should fix itself once the vertical mapping is completed – this usually took around one second at the maximum; *Vuforia* was able to identify and correctly position it on every try, no big re-adjustments needed.

Run	Horizontal 15x8	Horizontal 30x16	Vertical 30x16
1	44cm | 88cm	61cm | 116cm	67cm | 193cm
2	46cm | 92cm	50cm | 118cm	85cm | 198cm
3	40cm | 91cm	41cm | 117cm	71cm | 196cm
4	30cm | 90cm	68cm | 115cm	85cm | 210cm
5	39cm | 87cm	79cm | 117cm	86cm | 203cm

Figure 6.5: The distances the systems detected the images from. The first value shown is the distance for *ARCore*, the second is the result of *Vuforia*.

6.2.3 Space Synchronization

Goal

The evaluation of the image recognition (see section 6.2.2) was only done regarding the detection range but not about the precision of the positioning since the reference objects were rather small. Using the full application which places a map with a width of about two meters referenced on the 30 by 16 centimeter image reference, the goal is to find out about the differences of *Vuforia*'s and *ARCore*'s image detection precision. The resulting outcome includes tolerances of both systems in referencing and tracking stability, therefore the results of the tracking stability in section 6.2.1 should be kept in mind since they are affecting the results of this evaluation as well.

Method and Measurement

Just as in the evaluation of the tracking stability (see section 6.2.1), the image target is used to position the scene on the table. The image recognition will be done first on the *Microsoft HoloLens* and a piece of paper with a printed 2D ruler is placed with its origin at the bottom left corner of the transformed map. Afterward, the *ARCore* application is started and used to track the image as well and position the map related to it. The difference of positioning can then be seen and marked on the paper at the bottom left corner of the map. This is the most valuable evaluation in the area of spatial functionality since it is the difference which will be experienced by the users of the application and therefore will be tested seven times to interpret possible behaviors.

Conclusion

The difference between the map position on the *HoloLens* and *ARCore* (displayed in 6.6) are satisfying with a usual displacement of five centimeters at maximum. Very interesting is the pattern of the spatial synchronization error which is quite dense but moved three to five centimeters to the left which might be related to the stability results of *ARCore* in section 6.2.1. In the evaluation, *ARCore* already tended to displace the coordinate system rather to the left front which is confirmed in this evaluation as well. Further interesting evaluations would be the displacement when using a differently sized marker to see if the displacement error is bound to the size of the image target or just refers to a general error of about five centimeters which was the maximum error in the stability evaluation as well.

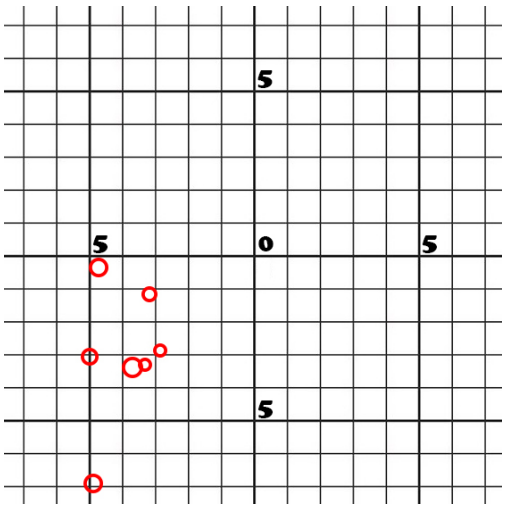


Figure 6.6: The results of calibrating the position of the map on the *HoloLens* and marking the position of the map detected on *ARCore*.

Chapter 7

Closing Remarks

The results of the evaluations and watching the progress from a standalone application for the *Microsoft HoloLens* and *ARCore*-supporting devices to a working system which synchronizes the application's content in time and space are very satisfying. Tests with friends and family showed the pleasure and astonishment Shared Augmented Reality Experience offer, especially when persons are able to discuss the seen and just point at a point of interest and act as if the holograms were real. Showing the application even technology-foreigners with full control over their actions and immediately seeing possible problems in controls and managing them in real-time in the same space is an advantage for developers of programs and supervisors.

ARCore's tracking, mapping and reference recognition is not at an impressive point in development compared to superior systems as the *HoloLens* offers but is still very valuable due to the simplicity in terms of hardware requirements and accessibility which clearly shows that the visual systems are on a good way to be used in daily life. The precision of the *HoloLens*'s tracking and mapping as well as the strength of *Vuforia* in combination with the smart glasses are still not comparable with any other solutions which definitely makes the *HoloLens* the exemplary model for Augmented Reality and its possibilities. Exactly this is the importance of the system created in the progress of this Master's Thesis, the *HoloLens* is an unbelievable powerful Augmented Reality device bringing Augmented Reality to a new level but due to new interactions not everyone is able to control or even set up the device in the beginning, possibly resulting in a tremendous first experience with it or Augmented Reality in general which such a cross-platform Shared Experience solution could prevent.

The future of this system is further testing and optimizing, using it as well in the additional development of applications and finding similarities between the applications creating a knowledge base for a possible implementation of a framework which could be used to offer more Shared Experiences in Augmented Reality. In addition to the standalone features of each device, sharing additional data which the devices are specialized for could improve Augmented Reality on all devices: The advanced mapping information of the *HoloLens* could be shared with the *ARCore* devices making dynamic occlusion possible on the smartphone solution or the data of the real light estimation of *ARCore* could be applied to the virtual lighting of the *HoloLens* application.

Appendix A

Technical Details

A.1 Project

The use-case itself – a holographic information application describing the rules and general information about the *Red Bull Air Race* – was implemented in *Unity 2017.3.0b9* starting October 2017 until February 2018. From this point on, the networking and spatial features were programmed in *Unity 2018.1.0f2* due to better implementation of *ARCore* functionality. From start to finish, *Visual Studio 2017* was used for writing the required C# scripts.

Built versions of the *ARCore* application are saved on the DVD as *apk* files which can be installed on any device with *Android 8.0 (Oreo)* or higher. Keep in mind, the permission about third-party applications needs to be enabled in the *Android* settings and the application *requires ARCore*; if the device supports *ARCore* but is not installed, the application will download and install it automatically. In other cases, *toast messages* (*Android*'s messages in bubbles) will inform the user about the status of their device in relation to *ARCore*.

The built *HoloLens* application is as well available on the DVD but as a *Visual Studio Project*. In order to build and deploy it to a *HoloLens*, the PC must be set up for *HoloLens* development which is described in *Microsoft*'s *Beginner Guide for HoloLens*.

A.2 Thesis

This thesis was written using ShareLaTeX – a free online LaTeX-editor – which merged later in 2018 into *Overleaf* using the *Hagenberg* LaTeX template.

Most figures showing structures or flow diagrams were created using *draw.io* which offers designing and creating many different types of graphs.

Appendix B

DVD Contents

Format: DVD, Single Layer, ISO9660

B.1 Evaluation

```
./
└── Evaluation
    ├── ARCore Image Recognition
    │   ├── 15cm
    │   ├── 30cm
    │   └── 30cm Vertical
    ├── Network
    │   ├── Closed
    │   └── Open
    └── Vuforia Image Recognition
        ├── 15cm
        ├── 30cm
        └── 30cm Vertical
```

The footage for all evaluations is within the folder "Evaluation". Every listed subfolder contains the runs as *.mp4 files; the network folders additionally contain screenshots of the essential moments of the evaluations including timestamps.

B.2 Project

```
./
└── Project
    ├── ARCore
    ├── Microsoft HoloLens
    └── Server
```

In every folder listed above, there is a file `RedBullAirRace.zip` which contains the *Unity* project for a specified use as the folders are labeled. Every project has the following folder structure:

```
RedBullAirRace
├── Assets
│   ├── (...)
│   ├── Scenes
│   ├── Scripts
│   └── (...)
├── Builds
└── (...)
```

The folder “Scripts” contains all scripts written for this project. The usage of the scripts in *Unity* and structure of game objects can be seen in the scene which is in the folder “Scenes”. The projects for the different platforms are already built in the folder “Builds” which just need to be executed on the devices; the *HoloLens* version is a built *Visual Studio* C# project and needs to be built by following the instructions in appendix A.

B.3 Thesis

```
./
└── Thesis
    ├── Source
    └── Thesis.pdf
```

A digital copy of this thesis as well as the source code of this thesis written in LaTeX is available in the folder “Thesis”.

References

Literature

[1] Stefan Auer. "Augmented-Reality: Entwicklung und Evaluierung eines benutzerzentrierten Prototypen für das Red Bull Air Race". Masterarbeit. Hagenberg, Austria: University of Applied Sciences Upper Austria, Human Computer Interaction, June 2018 (cit. on p. 8).

[2] Paul Milgram and Fumio Kishino. "A taxonomy of mixed reality visual displays". *IEICE TRANSACTIONS on Information and Systems* 77.12 (1994), pp. 1321–1329 (cit. on p. 3).

[3] Diederick C. Niehorster, Li Li, and Markus Lappe. "The Accuracy and Precision of Position and Orientation Tracking in the HTC Vive Virtual Reality System for Scientific Research". *i-Perception* 8.3 (2017). URL: https://doi.org/10.1177/2041669517708205 (cit. on p. 37).

[4] Dion Pike. *Master-slave communications system and method for a network element*. Sept. 27, 2001. URL: https://patents.google.com/patent/US7460482B2/en (cit. on p. 13).

Online sources

[5] North of 41. *What really is the difference between AR/MR/VR/XR?* 2018. URL: https://medium.com/@northof41/what-really-is-the-difference-between-ar-mr-vr-xr-35bed1da1a4e (cit. on p. 3).

[6] *Asus ZenFone AR*. URL: https://www.asus.com/Phone/ZenFone-AR-ZS571KL/ (cit. on p. 5).

[7] *CurvSurf's Find Surface*. URL: http://www.curvsurf.com/ (cit. on p. 2).

[8] *Environmental understanding of ARCore*. URL: https://developers.google.com/ar/discover/concepts#environmental_understanding (cit. on p. 5).

[9] *Google's ARCore*. URL: https://developers.google.com/ar/ (cit. on p. 5).

[10] *Google's Project Tango*. URL: https://get.google.com/intl/de/tango/ (cit. on p. 4).

[11] *Microsoft's HoloLens*. URL: https://www.microsoft.com/hololens/ (cit. on p. 4).

[12] *Presentation of ARCore's image recognition*. URL: https://youtu.be/xo54ldvOO34 (cit. on p. 7).

[13] *Red Bull Air Race*. URL: http://airrace.redbull.com/ (cit. on p. 8).

[14] *Simulation of the functionality of SteamVR's tracking*. URL: https://youtu.be/J54dotTt7k0 (cit. on p. 38).

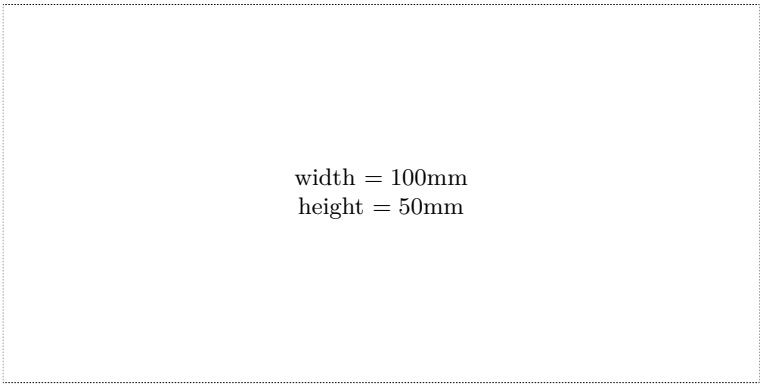
[15] *Spatial mapping of Microsoft's HoloLens*. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/spatial-mapping/ (cit. on p. 4).

[16] *Visualization of the HoloLens' spatial mapping*. URL: https://youtu.be/zff2aQ1RaVo (cit. on p. 40).

[17] *Vuforia*. URL: https://www.vuforia.com/ (cit. on p. 41).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —