Technical University of Crete
School of Electrician and Computer Engineering

HOLO-BOARD: AN AUGMENTED REALITY APPLICATION MANAGER SUPPORTING MACHINE VISION

Author: Daskalogrigorakis A. Grigorios
(gdaskalogrigorak@isc.tuc.gr)

Dissertation Thesis

October 2018

Acknowledgements

For this project I would like to especially thank Prof. Katerina Mania for giving me a chance to experiment with a new idea. I would also like to thank Professors V. Samoladas and M. Zervakis for accepting to be on my committee and approved the idea of Holo-Board.

I would also like to thank everyone in the MUSIC Laboratory for their valuable assistance, as well as all the colleagues that helped me throughout my studies and my friends and family for standing by me. Special thanks to Papadogiannis Sevastianos, Psihas Konstantinos, Loukas Harisis, Fragoulis Logothetis, Sason Nektarios and many more for their valuable assistance. Additional thanks to Konstantinidis Konstantinos for also showing me the Garamond font used in this thesis!

Abstract:

In this thesis we present an Augmented Reality Application Manager for Android smartphone applications using Google Cardboard. The main focus is to make an Application Manager that links smaller, more specific sub-applications and manages the flow of execution. It should also work as a Software Development Kit which provides tools that assist in developing new Cardboard-based AR applications. In addition, we provide alternative interaction methods between users and AR graphics, so users can interact with AR graphics without physical contact to the smartphone itself, as it will be in a Cardboard Mask. A custom Input Manager is also provided which can receive inputs from any external sources, such as a Machine Vision application, and then forward them to graphics applications in a distributed manner for future improvement.

Holo-Board was developed as a cheaper alternative to the newly developed Microsoft Holo-Lens, to run on Google Cardboard. This way developers not only have a cheaper alternative until AR masks leave their prototyping stages but also a much wider user audience, as almost everyone with an Android smartphone can run Holo-Board.

Holo-Board was developed in Unity 2017.3 for Android smartphones running with Android 3.X and above. We also use ARToolkit 5.3.2 for Unity plugin for Square based marker tracking. For the marker-based tracking we used a Hiro square marker (included in ARToolkit) of size 1,5x1,5cm mounted on a ring. Development was done on a Dell Inspiron 15 3000 series laptop and a Xiaomi Mi A1 smartphone.

Table of Contents:

List of Figures

1. Introduction

Augmented Reality is a new and rapidly developing new method of Human and Computer Interaction. In general, AR aims to take virtual graphics and blend them naturally into the real world. While there are multiple methods to achieve this, the general idea is we use various sensors to extract information about the world around the user, as well as his own position in it. Using this information, we can show objects around the user in a seemingly natural way, such as an object sitting correctly on a flat surface.

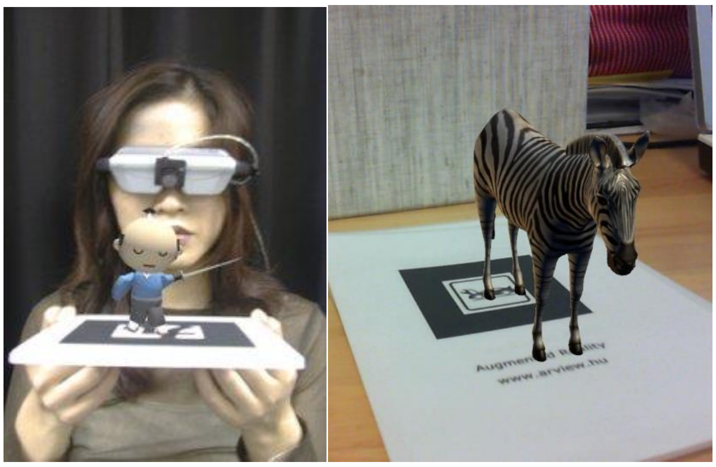


Figure 1.1. AR objects on flat surfaces.

The most common way to achieve this is through a screen with a camera on the back, usually a smartphone, which scans the environment, extracts necessary information and blends graphics appropriately. Others use projectors to project graphics on a surface and by utilizing projection mapping tools give simple 2D graphics a pseudo-realistic look. Finally, the most optimal method is using dedicated AR masks. These masks are head mounted with a transparent screen in front of the user's face and dedicated hardware to scan and project graphics as realistically as possible.

Although all 3 approaches to AR have the same ultimate goal, due to their different natures all have different perks and limitations. Smartphone applications are usually hardware-limited due to the high performance demands of AR, but a smartphone app can be used by almost anyone at any time and smartphones are much cheaper than the alternatives. AR masks are the most immersive of the 3 and they have dedicated hardware for all necessary features for AR, but the masks themselves are still early in development so they are both expensive and not in high demand in the market. Projectors have numerous tools to assist in projection mapping and projected applications can be enjoyed by anyone in a certain area, not only those that wear a special mask, but projection mapping has plenty of limitations to keep up the pseudo-realism.

Another major problem with all AR approaches is how users will interact with the graphics. Most applications have no or minimal interaction, and are mostly used to project visuals before the user. Smartphone applications usually use the smartphone's touch screen for interactions, which is not useful when developing a Cardboard-based application. Some Cardboard apps as well as some AR masks interact heuristically with objects relative to where the user is looking at. Finally, some applications use controllers, which works well even though is not as immersive.

1.1. Brief Description of our Approach

Holo-Board is a Software Development Kit, or SDK for short, which provides developers with tools to design their own AR application. The main focus is Cardboard-based AR applications which have specific limitations not covered by other existing SDKs. Holo-Board also aims to imitate the flow of execution of an AR mask, similar to an Operating System linking multiple smaller programs in a distributed manner and working as an Application Manager that handles the flow of execution and communication between them. Finally, Holo-Board provides support for alternative input methods to the smartphone's touch screen, and also includes support for alternative Machine Vision Apps to be added in the future.

As a Development tool, Holo-Board uses the ARToolkit library for Square Marker tracking and Natural Feature Marker tracking, through which we have developed a custom-made Machine Vision based cursor and buttons for interactions in the UI. ARToolkit also automates the process of making a stereo view on the phone's screen from the camera feed. We have also added support for DS4 Playstation 4 controller inputs via Bluetooth, a Main Menu as an overlay to the screen usable both by Machine Vision or DS4 controller buttons. Holo-Board also includes a camera Handler which allows programmers to design the UI over one eye and then it automatically duplicates it to the second, as well as providing us with a single camera perspective useful for debugging.

As an Application Manager, Holo-Board has a premade reprogrammable Main Menu made with our custom-made Machine-Vision based buttons and DS4 controller inputs in mind, a Heads-Up Display manager which automates enabling and disabling a graphics overlay on the screen as well as a FULL-app manager that switches from the basic Holo-Board's perspective to a new empty one to give full freedom to any fully functional application another developer may make. For the communication between objects we have made dictionaries through which any object can reference another, while if we want to inform the user about anything we have designed a notification text that shows a message on the user's screen's overlay for a few seconds. We have also made a skeleton demo App through which any user can test how all our tools are used.

Finally, we have included a custom-made Input manager through which any developer can link his own Machine Vision based inputs and any Holo-Board sub-application, as Unity does not support non-Hardware-based inputs.

1.2. Thesis Structure

In this chapter we gave a brief description of what Augmented Reality is as well as the limitations and problems developers face when developing AR applications. We also provided a brief description of what our application accomplices.

In Chapter 2, we provide an in-depth introduction to Augmented Reality. We start by defining what AR is, analyze what the two key issues of AR are (the registration problem and user interactions) and we showcase some key AR applications through the years. We then focus more on Smartphone applications, smartphone SDKs and AR Masks, which are directly tied to Holo-Board.

In Chapter 3 we make a requirement analysis about what we should do to consider our application complete through different perspectives. We also outline all the Hardware and Software used in the development, as well as recommend alternatives not present when our development started.

Chapter 4 is a presentation of the use-cases of our application. Even though our application is a development platform we do analyze it both from the perspective of a programmer wanting to develop his application, and also from the perspective of an End-user that executes the demo application we provide, as it is very likely a future End-user application made using Holo-Board will have the same use-cases.

Chapter 5 is the complete developer's manual for Holo-Board. In this chapter we provide a full analysis of how the demo app works, as well as how all the tools present in Holo-Board are used from a developer's perspective. We have a full analysis for Holo-Board's Architecture and an explanation of how and why everything is linked the way it is. Finally, for every tool in Holo-Board, we analyze how it is used, how a future developer can change it to fit his needs and why/when he should use it.

Chapter 6 is the full Implementation process from our point of view. There we analyze exactly what we did and the reason we made everything the way it is. We also explain all the issues we faced in the development process and how we solved them or why we didn't solve them, with suggestions for anyone planning to fix them in the future.

Finally, in Chapter 7 we have a summary of everything we mentioned above, focusing more on what we achieved or didn't achieve. We then summarize some results from tests with Holo-Board made by people outside of the developing team and their comments on our application. In addition, we list future improvements that can be made to Holo-Board to solve some aforementioned problems, mostly left out due to time constraints.

2. Augmented Reality

2.1. Introduction

Augmented Reality (AR) is the act of superimposing digital artifacts on real environments. In the reality-virtuality continuum (Milgram 1994) (figure 2.1-1), AR is a part of the broader Mixed Reality spectrum. In contrast to Virtual Reality where the user is immersed in a completely synthetic environment, AR aims to supplement reality. While early research limited the definition of AR in a way that required the use of head-mounted-displays (HMDs), a taxonomy introduced in (Azuma 1997) tried to differentiate it from the required technologies and defined that any system that;(1) combines virtual and real, (2) registers (aligns) real and virtual objects with each other and (3) runs interactively in three dimensions and in real time, is considered an AR System.

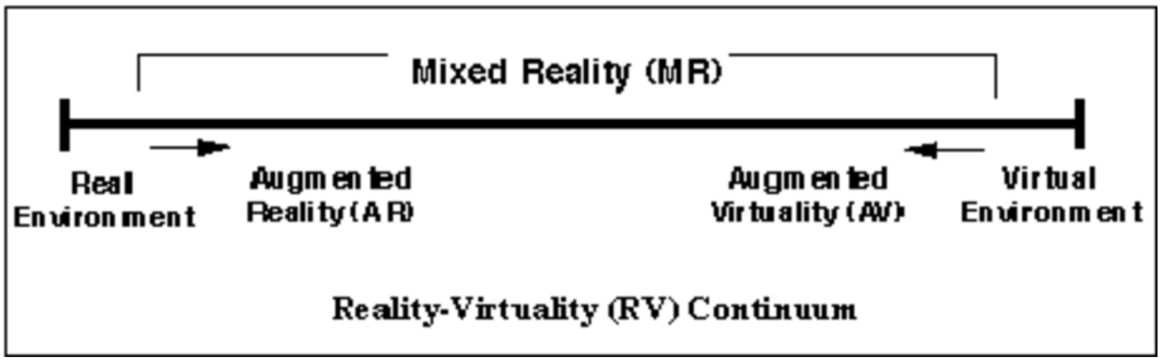


Figure 2.1. Milgram's Reality-Virtuality (RV) Continuum.

Keeping the above definition in mind, in our application we focus on two key issues of AR: 1) How do we align virtual objects with real ones and 2) how do we interact with said virtual objects in a natural way.

Before we explore these two issues, we should start from the beginning. First we will present a few interesting applications of early AR systems, which mostly attempted early solutions of the first issue, the registration problem. Then we will analyze the registration problem, outlining exactly what it is and how we can solve it. Next, we will talk about the present state of AR, especially focusing on AR masks and smartphone applications which are tied directly to our application. Finally, we will talk about the second issue, interactions in AR and tried solutions over the years, as this is also directly related to our work as well.

2.2. History of AR

While AR is most widely known for its modern applications, it has been around and experimented upon for approximately 20 years. Thus, various technologies have already been tested using a multitude of tools, especially when trying to align the virtual and real worlds. Older technologies consisted of Head-Mounted Displays (or HMDs), eyeglasses or contact lenses that showed virtual objects in front of the user's eyes. This posed a multitude of problems because the tolerance when tracking sudden movements of the user was low and the precision of the then available instruments could not match it, thus users experienced frequent nausea and disorientation.

Moving further into the future, AR applications moved from HMDs to handheld tablets and smartphones. As users were distanced from the virtual screens, the tolerance for accurate tracking rose making it simpler to test new ideas. In addition, with the rising popularity and demand for better smartphones and tablets, many complicated Head-Mounted sensors were integrated into smartphones and tablets, making them the ideal environment for developing new applications.

Below, we will showcase a few early AR applications using different approaches through the years.

2.2.1. Vlahakis et al (2001): Archeoguide: First results of an augmented reality, mobile computing system in cultural heritage sites

One of the first Mobile AR (MAR) Systems was built in 2002, as a predecessor to the modern AR masks, for the site of Ancient Olympia (Vlahakis 2001). The system provided on-site help and Augmented Reality reconstructions of ancient ruins. The system made use of a compass, a DGPS receiver and together with the comparison of live view images from a webcam it obtained the user's location and orientation. Visitors had to carry a backpack computer which performed the calculations and wear a See-through Head Mounted Display (HMD) to display the digital Content. The mentioned components were hooked on the backpack computer making it a cumbersome MAR unit not acceptable by today's standards. In addition, the optical tracking approach requires a large number of images to be compared in real time which leads to fixed viewpoints, thus disallowing movement while viewing the reconstructions, and adds additional system delays as the communication with a central database that holds the original images is required. Despite the ergonomic restrictions, the system was very well received by the visitors as it provided a unique site-seeing experience.

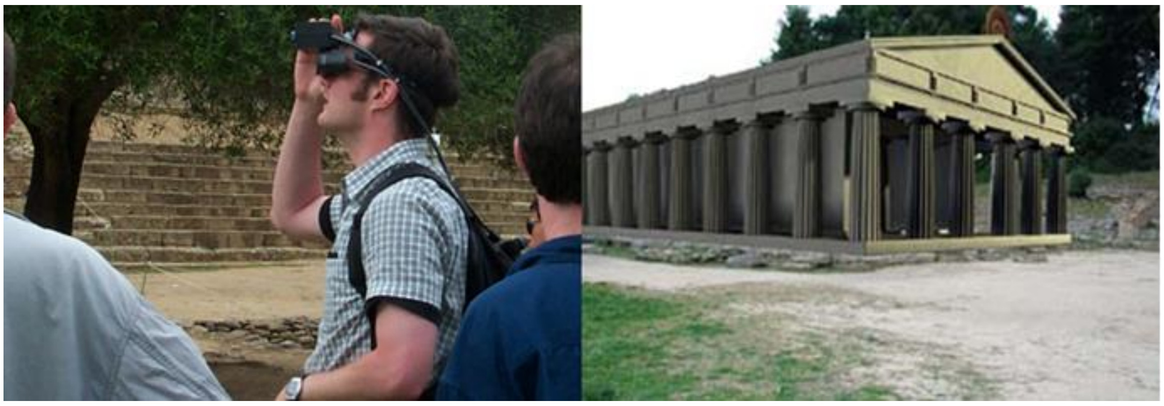


Figure 2.2. The HMD in action (Left). Digital reconstruction of the temple seen through the HMD (Right).

2.2.2. Choudary et al (2009): MARCH: Mobile Augmented Reality for Cultural Heritage

MARCH was a mobile Augmented Reality application developed for digitally enhancing the visits of prehistoric caves. It was developed in Symbian C++, running on a

Nokia N95. It was the first attempt of a real time MAR application without the use of grey-scale markers. Instead, it was using coloured patches added to the corners of images containing prehistoric cave engravings. The system made use of the phone's camera to detect these images and overlay them with complete drawings made from experts. The augmentations would either be available in museums or by acquiring the prepared images, uprooting the experience from its original context and presenting it in a context-less object. MARCH works very similarly to more modern Smartphone AR applications even though it was made for a standard mobile device.

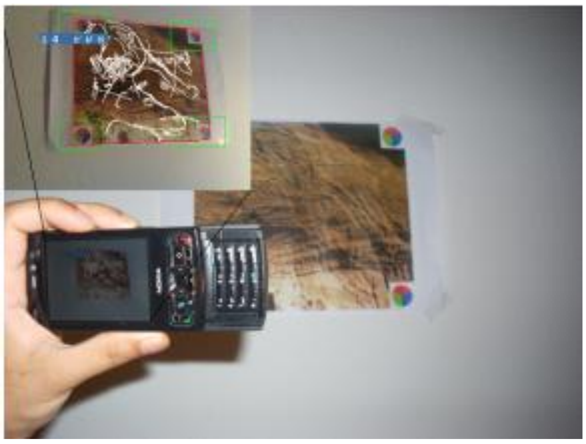
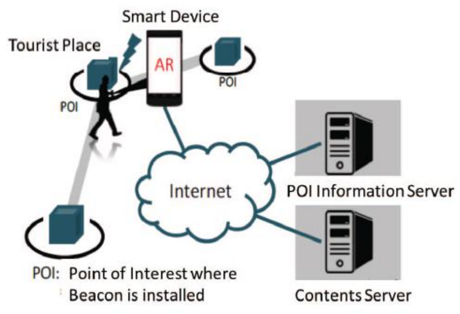


Figure 2.3. MARCH running in a museum-like environment.

2.2.3. Yoshitaka et al (2016): Tourist Information System based on Beacon and Augmented Reality Technologies

In this project, Yoshitaka et al developed a new sightseeing information system for tourists using Augmented Reality on a Smartphone. By installing beacons on Points Of Interest (or POIs), key locations were marked. These beacons were standard Bluetooth beacons that connected to the phones of visitors using the AR application. When a beacon was connected to the phone two things would happen. First, the phone would calculate the angle at which the content came from, and when that content was in view would connect to an online server and retrieve data relative to that beacon's ID. It would then show that information in AR through the phone's screen over the estimated beacons' positions. This application is one of the first smartphone applications that was developed in the modern standards for AR applications.



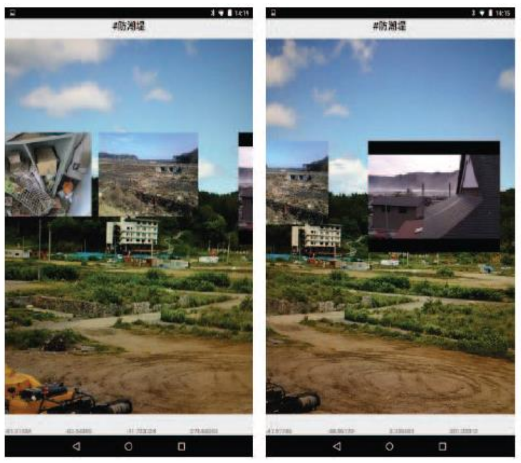


Figure 2.4. Yoshitaka et al system layout (Left). Yoshitaka et al AR view of POI information (Right).

2.3. The Registration Problem

Registration in an AR system is the degree in which the virtual information is accurately presented with the real environment. The objects in the real and virtual worlds need to be properly aligned with respect to each other, or the illusion that the two co-exist will be compromised (Azuma 1997). In Virtual Reality such issues would only cause visual-kinesthetic disorientation, while in Augmented Reality such issues cause visual-visual conflicts so they are much easier to detect.

Earlier AR systems had major issues regarding the registration problem. Most registration problems stemmed from end-to-end system delays, from sensors detecting movement to the system showing the updated visuals to the user. Another issue of older systems was the computational cost of the calculations needed for AR. Even if the sensors were instantaneous in sending data to the processing units, the calculations themselves were too timetaking for the hardware available at the time. Because of that, early AR systems focused on developing new methods to track the environment and/or the user's position and pose in it.

In order to achieve that, multiple methods were devised, the most popular of which will be explained below. These include Marker-based AR, Natural Feature Tracking, Sensor-based tracking and the newer Markerless AR for Edge Detection.

2.3.1. Marker-Based AR

Marker- based AR is the simplest form of tracking for AR. It consists of tracking a predefined shape, usually a black and white pattern printed on a piece of paper. Detecting these shapes is simpler than other more complicated objects, and because of that Marker-based AR was widely used even in early AR systems. Marker-based detection could be based on a single marker or even on a collection of markers, usually for more complicated objects or shapes, ex four markers on 4 edges of an object. Markers are usually either 1D barcodes or 2D square shapes, similar to QR codes.

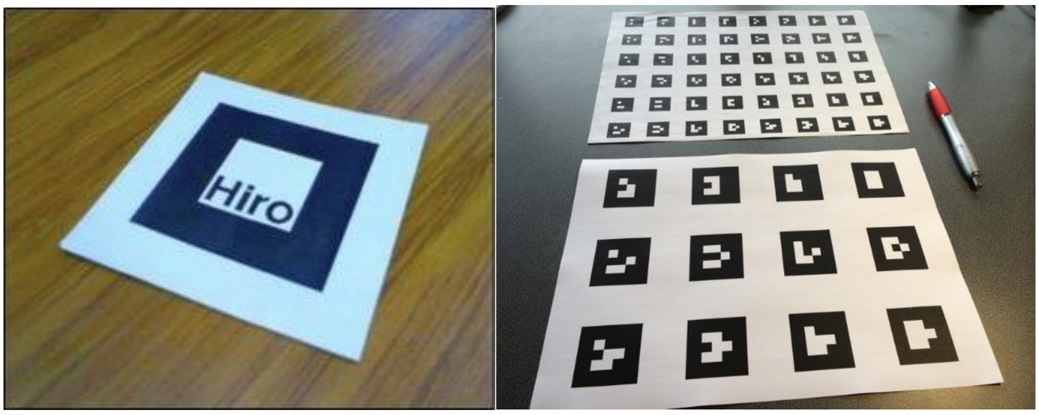


Figure 2.5. ARToolkit's square markers. Single Hiro Marker (Left). Multimarker surfaces (Right).

By defining these objects as size specific we can also calculate how far we are from a marker by comparing its size with the expected size at some range. Also, another benefit of using square markers is we know the expected shape of its edges would be two horizontal and two vertical lines. If we see a marker from an angle instead of a square border we will see a trapezoid. Using triangular calculations we can determine the marker's rotation relative to the camera. Calculating the rotation of markers was widely used only after AR was developed for smartphones, where simply tracking the markers had become a more trivial task.

Despite all the advantages of Marker-based AR, it still remains the simplest form of AR. Due to its nature, the markers are usually black-and-white blocks that feel out of place in most environments, sometimes enough to break the users' immersion. AR content is also tied to those markers, thus Marker-based AR is mostly used at specially designed places rather than on the fly AR.

While the above disadvantages certainly make Marker-based AR a more outdated alternative, in our application we will show that creative use of Marker-based tracking can be beneficial when it comes to interacting with AR, even more so than the more modern registration methods we will analyze below.

2.3.2. Natural Feature Tracking

Natural Feature Tracking, commonly referred to as NFT is a more immersive alternative to Marker-based AR. Similarly to Marker-based AR, NFT also tracks pattern shapes, the difference being these shapes can be infinitely more complex like photographs. By selecting any digital picture, NFT extracts a collection of key features about the shape and colors of what is depicted in that picture and use that collection as a complex marker.

Figure 2.6. ARToolit's demo NFT marker "gibraltar".

Compared to Marker-based AR, NFT is more computationally expensive due to its nature, but NFT markers can be hidden anywhere and blend into the environment naturally, making it better for immersive experiences.

Even though we could have used an NFT marker in our application we decided to stick to a Square marker as we did not need to track anything complex for the scope of our demo application.

2.3.3. Sensor-Based Tracking

Sensor-based tracking is an alternative perspective when it comes to AR tracking. Instead of tracking key points around the user, we try to track the position and pose of the user himself and build AR content around him. In general, there are two approaches to Sensor- based tracking: using external and internal sensors.

External sensor tracking (like Yoshitaka et al, 2016) uses sensors in pre-specified key points in the environment. Using the users' relative position to these points we can determine their actual position and determine what part of the virtual world is visible in front of them.

Figure 2.7. Yoshitaka et al (2016) Beacons on POIs.

Internal sensor tracking is the opposite approach. Sensors are integrated into the AR hardware, like HMDs or smartphones, and using their readings we determine the users' position. Common sensors used are GPS and AGPS locations, compass angle, Accelerometer's acceleration and Gyroscope's relative rotation.

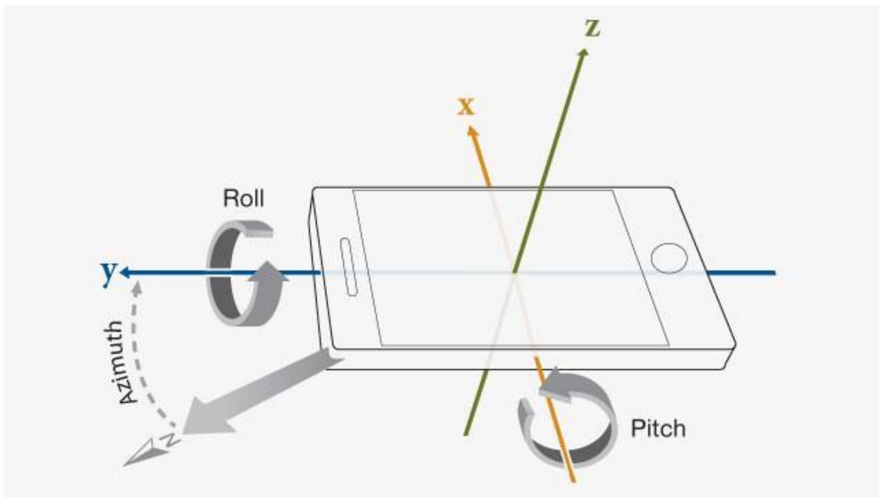


Figure 2.8. Pose estimation using a phone's internal sensors

Compared to NFT and Marker-based AR, sensor-based tracking works quite differently. Instead of AR objects being centered on key points in space, they are instead focused around the user himself. Because of that, AR does not need to be tied to a specific location and instead can be available anywhere on demand. Also, since smartphone technology advances in rapid succession, both more and better sensors are available every year making sensor-based apps more precise and more optimized with each passing year.

Even though most modern SDKs use internal sensor readings in order to track the user's movement in the world we did not use Sensor-based tracking in our application as we are not interested on how the user moves or what is outside our field of view, we are only interested in what is visible in front of the user.

2.3.4. Markerless AR

With the rapid development of new technologies when it comes to machine vision, Markerless scene tracking has become possible in real time. With high resolution, high framerate cameras becoming widely available and cheap, we can extract highly detailed information about the surrounding environment, analyze the structure of the world and update virtual objects to blend in, all in realtime.

Usually Markerless AR focus on detecting specific key features of the environment, not a full recreation of the real world. The most commonly tracked feature is edge detection between objects and identification of flat surfaces. Flat surface detection is popular because when AR objects stand on a flat surface or are aligned with the walls of a room they immediately look blended into the environment.

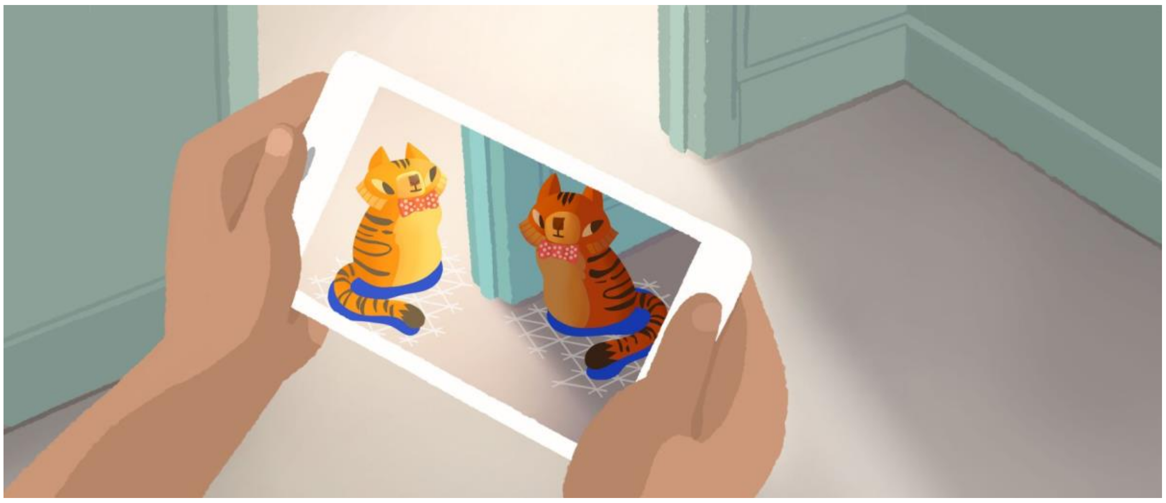


Figure 2.9. ARCore's Markerless AR flat surface detection with brightness calculation.

Furthermore, some newer smartphones as well as new HMDs further enhance markerless detection using a collection of cameras. By using multiple cameras or Infrared sensors we can also detect the depth of objects relative to the user allowing for better precision.

Compared to other registration methods, Markerless AR is both the most immersive and the most complex. Markerless AR has become widely popular only in the last few years because with older systems it was nearly impossible to detect scenes correctly, with precision and in real time all at once. Even as of writing this paper, Markerless AR is still in early development stages as the necessary hardware is still expensive and still in prototype stages.

2.3.5. Hybrid Implementations

As all the above registration methods are not mutually exclusive and detect different things, most developers tend to use multiple at once. Usually, Sensor-based tracking is combined with visual detection, as having information about the users' position and movement can be used to improve precision. Sensor-based tracking is also much faster relative to other registration methods, so using it is more beneficial than its computational cost.

In addition, visual registration methods are also combined. Since fully immersive methods are usually computationally expensive, they are also combined with NFT or Marker-based AR to reduce the computational cost, or to improve accuracy by detecting key points.

Due to Holo-Board's architecture a programmer may find it easier to link different libraries and design a Hybrid AR system even if it is not fully supported by one SDK, but for our implementation we did not need a Hybrid AR system.

2.4. 2018: A New Era for Augmented Reality

As of writing this paper, the last few years have seen immense improvements in AR technologies. Not only are smartphones becoming equipped with high end sensors mandatory for AR, but also major companies have started prototyping new HMDs, commonly known nowadays as AR masks. AR masks tend to make standalone dedicated hardware specifically for AR, but unlike old-school HMDs, these will not be application specific but reprogrammable AR hardware. On the other hand, there are multiple Software Development Kits, or SDKs for short, which can be used to develop smartphone applications. As our work reflects technologies from both sections, we will analyze them both below.

A broader look into the current state of AR is shown in Ling et al. (2017).

2.5. AR Masks: The Future of AR

AR masks are the epicenter of modern AR research. Using new technologies borrowed from the nowadays popular VR masks and combining them with spatial scanning technologies such as Microsoft's Kinect promises to create a new standard for AR research. These new masks promise to have fully 3D visuals for all necessities, from industrial and academic usage to integrating standard computer functionality into a mask. As many major companies, such as Microsoft, are investing into developing their own AR masks, this medium promises to be as big an evolution in technology as smartphones were 15 years ago.

Although the new era of AR masks started back in 2016, up until 2018 their production and shipping were very limited. Still, AR masks are still a prototype idea starting to slowly take form. Major companies are competing to design the optimal User Environment, usually with completely different approaches into both the hardware as well as the software of these devices. As such, it is a new technology that still requires years of optimization and improvement until it is widely known and accepted.

Below we will analyze a few such AR masks we believe will have major influence in the years to come.

2.5.1. Microsoft Holo-Lens

Developed by Microsoft, back in 2016. One of the first AR masks to be announced and sell their prototypes, although in limited regions. Backed up by Microsoft's name, Kinect's tracking technology and an ambition to fully integrate Windows in an AR environment, this mask has set very high expectations both for itself and competitors. Even our application Holo-Board was inspired by the Holo-lens' announcement.

Figure 2.10. Microsoft Hololens.

When it was first announced, Hololens not only promised to integrate traditional computer graphics in AR, but also full scale holograms, for example human holograms, that could move naturally or even recreate scenes like a full-scale soccer match from a recording.

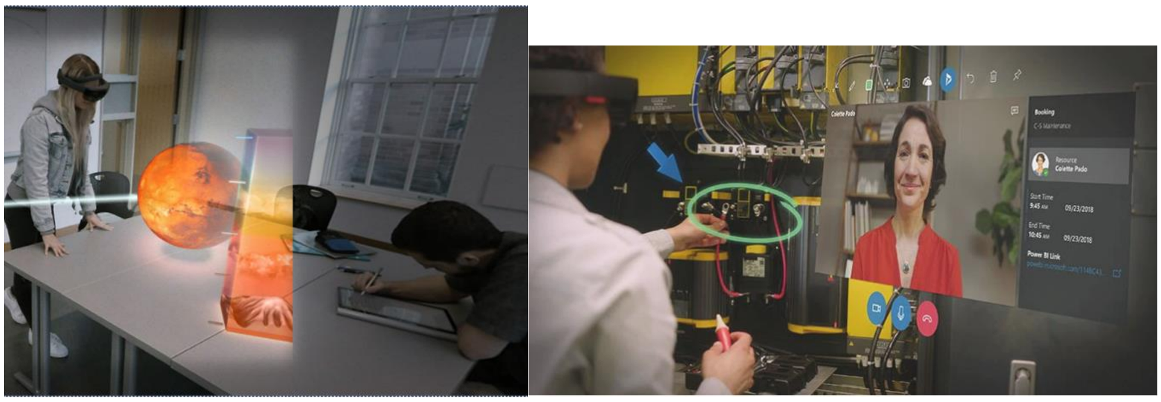


Figure 2.11. Full scale 3D Holograms in Hololens (Left). Traditional windows in AR (Right).

Packed with the processing power of an average laptop and a multitude of sensors Hololens aims for precision tracking and world scanning around the user. On the software side, it uses an optimized version of the already trained and tested Microsoft Kinect's Neural Network for tracking and environmental scanning.

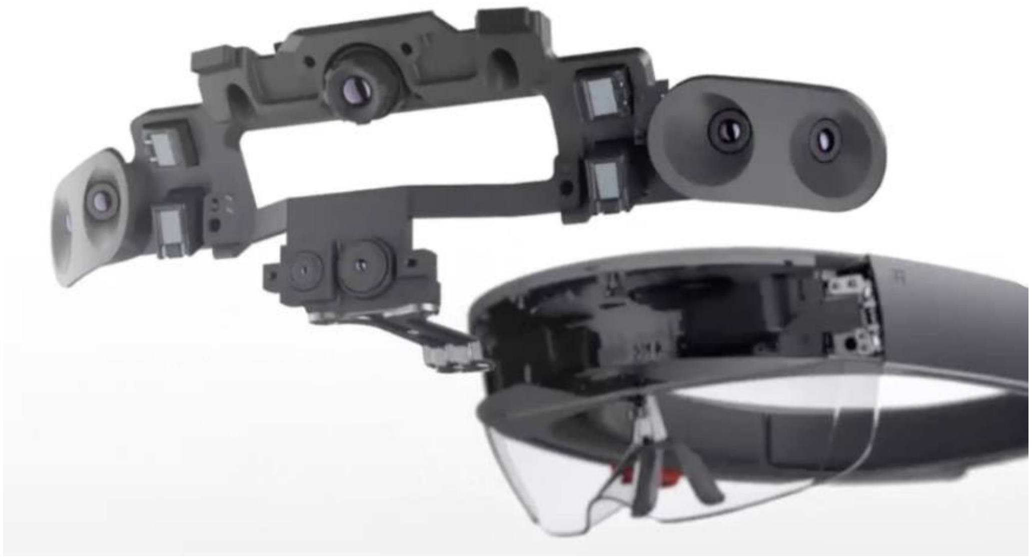


Figure 2.12. Internal sensors of the Hololens.

2.5.2. DAQRI AR Mask

Developed by DAQRI, mostly for professional use. This mask focuses on tools useful mostly for professionals instead of everyday use. Using DAQRI's long term expertise in AR the goal is to equip this AR mask with any tools a professional environment would need.

Figure 2.13. DAQRI AR mask.

Although still in development, DAQRI have defined the 5 apps included in their basic DAQRI Worksense environment: Show, Tag, Scan, Model and Guide

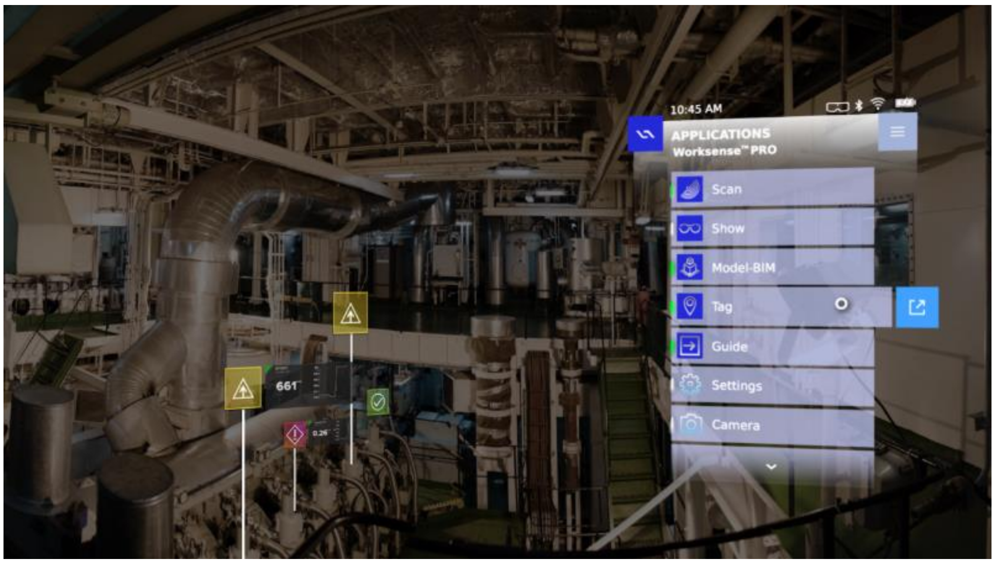


Figure 2.14. DAQRI Worksense.

Show consists of streaming video, audio and the 3D environment to a distant user. These users can then observe, give instructions or even annotate the real world using visuals through a digital tool. This is useful for remote assistance from experts, remote product support or even for remote presentations.

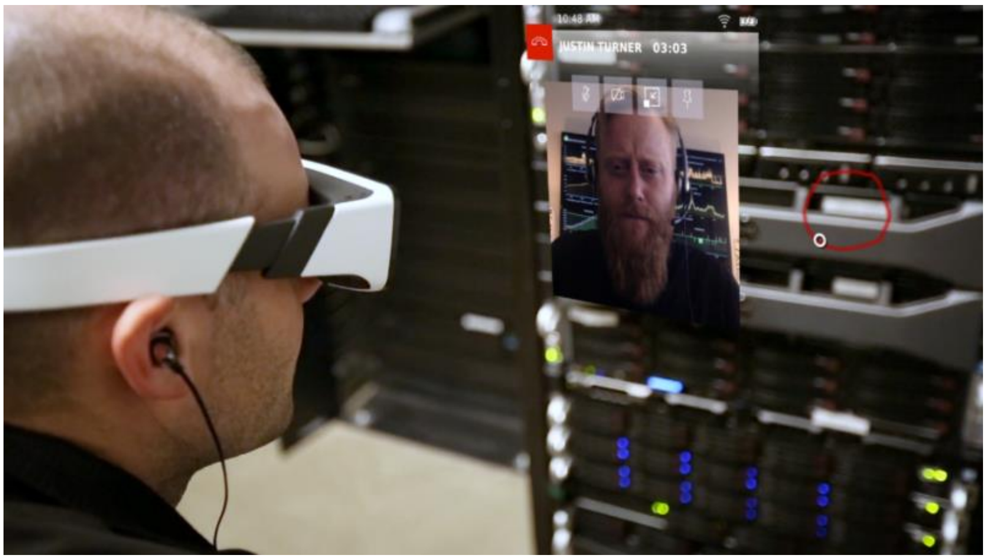


Figure 2.15. DAQRI Show.

Tag helps users mark key objects in the environment, and view that information at a glance. Tag attaches critical information on physical objects and shows that information in real time on the real world. Also, Tag can also connect to existing IoT systems and present a live feed of sensor data.

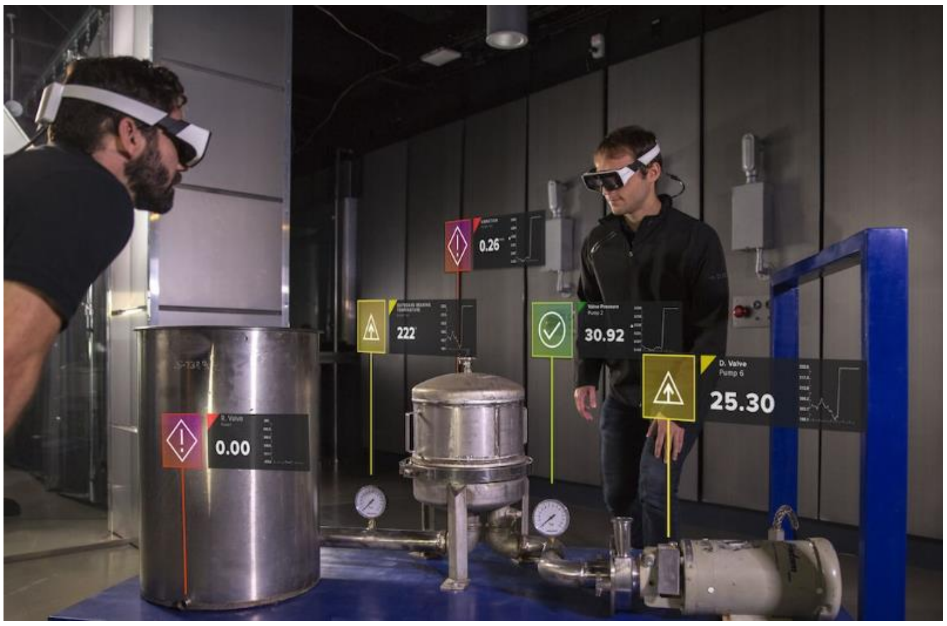


Figure 2.16. DAQRI Tag.

Scan is designed to capture the environment into photorealistic 3D models by scanning them with the mask. These models can then be enhanced remotely by tagging from a computer or be extracted and used in other programs such as Unity.

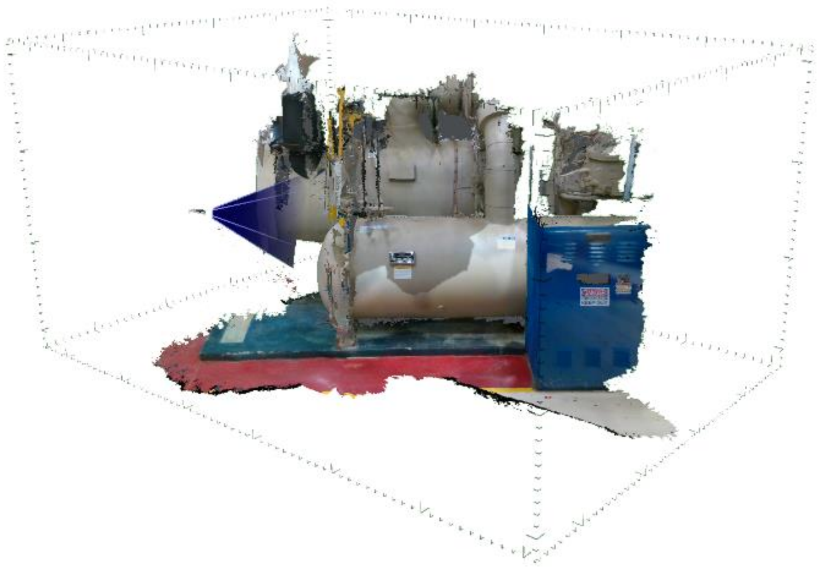


Figure 2.17. DAQRI Scan.

Model transforms 3D objects from Autodesk BIM 360 Docs into immersive walkthroughs. This can help compare complete virtual designs with real world in-progress constructions and also keep a full sync of the progress with a central office.

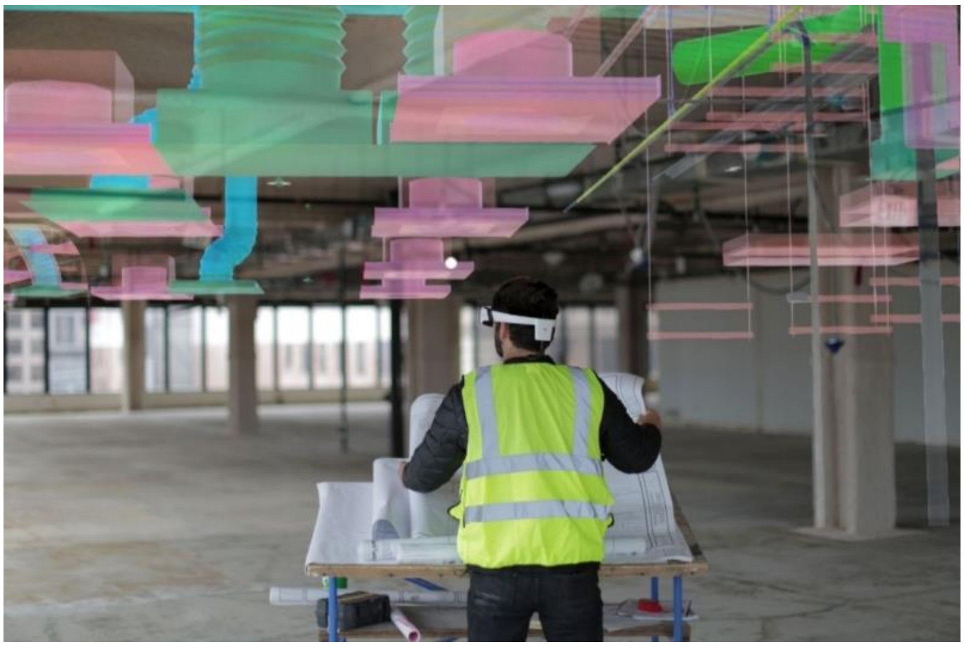


Figure 2.18. DAQRI Model.

Finally, Guide provides full scale digital assistance in an AR environment. This helps show full scale tutorials, guidance or manuals in AR view.

2.5.3. Magic Leap

Magic Leap is another popular modern AR mask. Contrary to the serious nature of the previously mentioned Hololens and DAQRI masks, Magic leap's focus is graphics and immersion.

Figure 2.19. Magic Leap One AR mask.

Magic Leap uses Machine vision to thoroughly scan the environment around the user and make virtual objects context sensitive to the world around them. In addition, virtual objects are not only visually immersive but also use spatial audio with increasing depending on the distance from virtual objects.

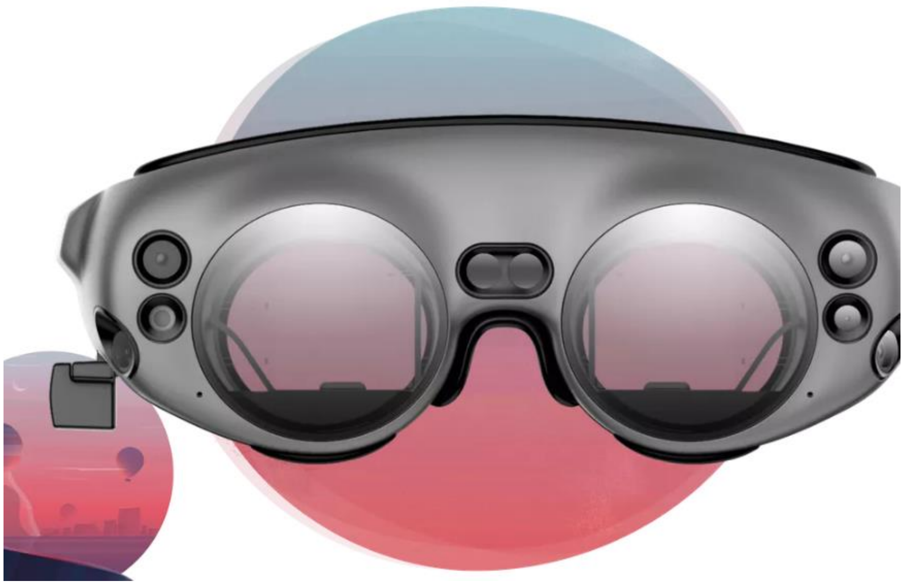


Figure 2.20. Magic Leap frontal view.

Magic Leap's hardware consists of more than just the mask. The mask consists of the glasses and stereo headphones the user wears, but all the processing power resides in a wearable pouch that clips on the user's pocket, named Magic Leap Lightpack. Since all processing tools are not on the mask itself, it is more comfortable than the previous masks. It also comes bundled with a controller with 6-DoF (Degrees of freedom) of movement called Magic Leap Control.

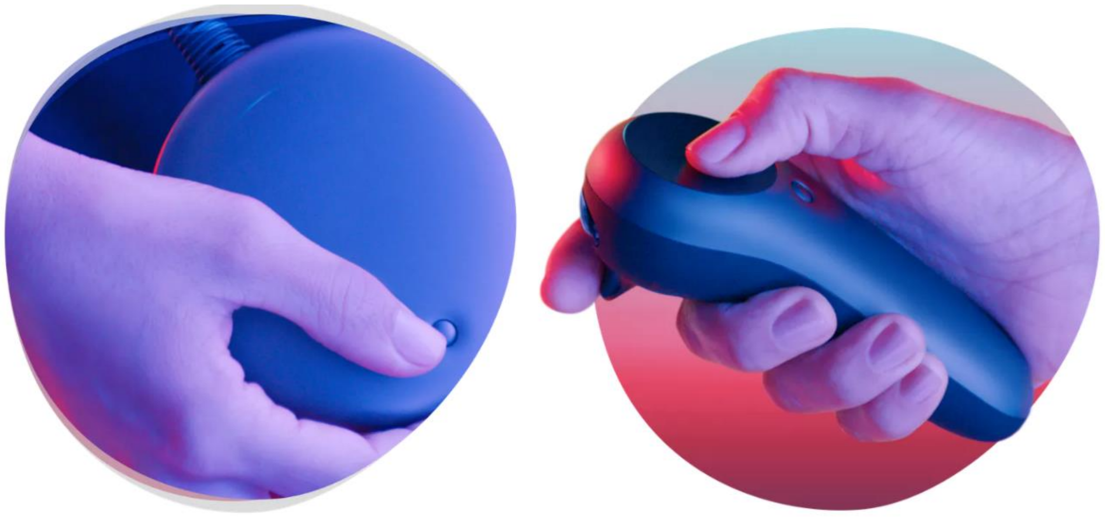


Figure 2.21. Magic Leap Lightpack (Left). Magic Leap Control (Right).

Finally, Magic leap uses its own Operating System called LuminOS, which aims not only to assist in developing immersive AR experiences, but also making them a social experience that can be shared with others.

2.6. AR for Smartphones

AR apps utilize the sensors and cameras already present in modern computers or, more commonly, smartphones in order to gather information from the real world and allow virtual graphics to blend into the natural environment seen through a camera. In order to develop an AR application, developers frequently use a pre-built Software Development Kit, or SDK for short, which provides them with premade tools useful for AR. These tools vary from automating simple jobs, like setting up a new AR scene, to complicated algorithms like using Machine Vision to scan the environment and extract data like marker detection.

While there exist plenty of AR SDKs they usually each have a specific focus, and it is quite frequent for companies to stop supporting and killing dated SDKs and newer companies publishing brand new SDKs. Luckily, there are still a few older SDKs still in existence, even with less support from their developers like Vuforia and Wikitude. In our application we used ARToolkit, which is still supported until today due to it being Open source. Finally, the newest SDKs available are ARCore and ARKit that unlike previous ones are supported by Google and Apple directly. For our application, we selected between using Wikitude, Vuforia and ARToolkit as a base and in the end we decided to use ARToolkit. Below we will analyze these aforementioned SDKs.

2.6.1. Vuforia

Developed by Qualcomm, Vuforia is a very popular low level library. It is widely known for its Computer Vision capabilities as it supports the natural feature tracking of planar images, detection of cylindrical surfaces, small 3D objects, text and small boxes with flat surfaces. Even though in recent years it has not been updated there is ample documentation in its site and online forum. Using the Vuforia library can either be done with the Android NDK in Cor in Unity using the Vuforia pugin.

2.6.2. Wikitude

Wikitude is a popular high level AR SDK that combines image and object recognition, extended tracking, even after recognized objects leave the user's view as well as geo-location services using the GPS signal. It also provides cloud-based recognition for big datasets and instant tracking, a combination of sensor readings and image processing for environmental tracking and placing objects in AR. Wikitude provides implementations for multiple platforms such as Java, JavaScript and Unity. Unlike Vuforia, Wikitude is still being actively updated and supported.

2.6.3. ARCore and ARKit

As of writing this paper these two SDKs are the top of the line. Both of these SDKs provide almost the same tools, ARCore being for Android smartphones and ARKit for Apple smartphones. These SDKs were also integrated into Unity Engine so they can be added in Unity Projects without externally importing them. ARCore and ARKit are equipped with top of the line Macihne Vision for detecting flat surfaces, like walls and floors, with detailed information about the lighting conditions and changes in birghtness on the whole visual field. In addition, they also calculate the exact position and orientation of the phone using the built-in accelerometer and compass and then try to recreate the real world continously instead of independently each frame. This way, it recreates the real world as a continouous scene, correcting parts of it when viewed from different angles, allowing for complex visual interactions with AR graphics.

On the other hand, ARCore and ARKit have some drawbacks. In order to execute such high precision analysis of both the world and the smartphone's position they require very demanding top of the line smartphones. As of writing this paper, ARCore and is only available on the top of the line smartphones with Android 8.0 (released in 2018) and ARKit only on IPhone 6S and 7 and above. Currently these smartphone have an average minimum price of around 700 Euros, which is around the same estimated commercial cost AR masks will have when mass produced. In addition, ARCore is not built with Google Cardboard in mind. If an application wants to have physical interaction with objects it must use the touchscreen.

2.6.4. ARToolkit

ARToolkit is an older SDK than ARCore and ARKit bought by DAQRI, which later even designed their own AR masks based on their experience with AR. Compared to the previous SDKs, ARToolkit uses a simpler marker-based approach, with additional support for Natural Feature Markers. ARToolkit tracks the markers while in view, calculating which markers are in view, their orientation relative to the camera as well as their depth from the camera.

By using markers as position trackers, AR graphics do not blend as naturally to the real world and immersion can be broken either by the graphics' pseudo-3D displaying over obstacles which should obstruct them or even by the marker itself. The most commonly used markers are 2D barcodes and 3D boxed pattern barcodes, which can often look out-of-place if they are not hidden correctly.

On the other hand, ARToolkit has a few advantages not present in the previous SDKs. The best perk of ARToolkit is it is an Open source library. This way, any programmer can alter its code and improve it as they see fit. As a result, even though DAQRI recently stopped supporting it Realmax Inc. created their own version of ARToolkitX and continue to support it.

Additionally, ARToolkit's marker tracking can be used as a substituted position tracking, for example to track a finger. Using this approach, we can create a pseudo-gesture recognition, which because it's tracking a marker is much faster than tracking physical objects like fingers. This way, ARToolkit can generate inputs using mid-air gestures or positions similar to how Kinect and Leap Motion work and thus be used in Google Cardboard with interactable graphics.

On another note, if it is necessary for markers to hide in the environment NFT can substitute traditional markers, at the cost of some performance. As marker tracking is a much easier operation than what is used in other SDKs even with the increased performance of NFT, ARToolkit can work on almost smartphones due to having very low performance cost overall.

Because of the above reasons, ARToolkit is used as a plugin to Holo-board providing us with a multitude of tools while not restricting it to high end smartphones.

2.7. Popular AR End-User Smartphone Applications

In the previous section we talked about SDKs for developers to use when develop AR applications. But since developing an app is insignificant if no one is interested in the medium below we will present a few AR applications which are widely popular and made people interested in AR.

2.7.1. Pokemon GO

The first widely popular AR game of 2016. Developed by Niantic back in 2016 and widely known because it was the first smartphone game that incentivized everyone to go outdoors to play. Using the GPS signal of players phones it tracks their location and various events happened depending on both relative position and distance travelled. It also used key locations from Google Maps worldwide and certain events would happen around those key locations, incentivizing users to visit multiple places around town.

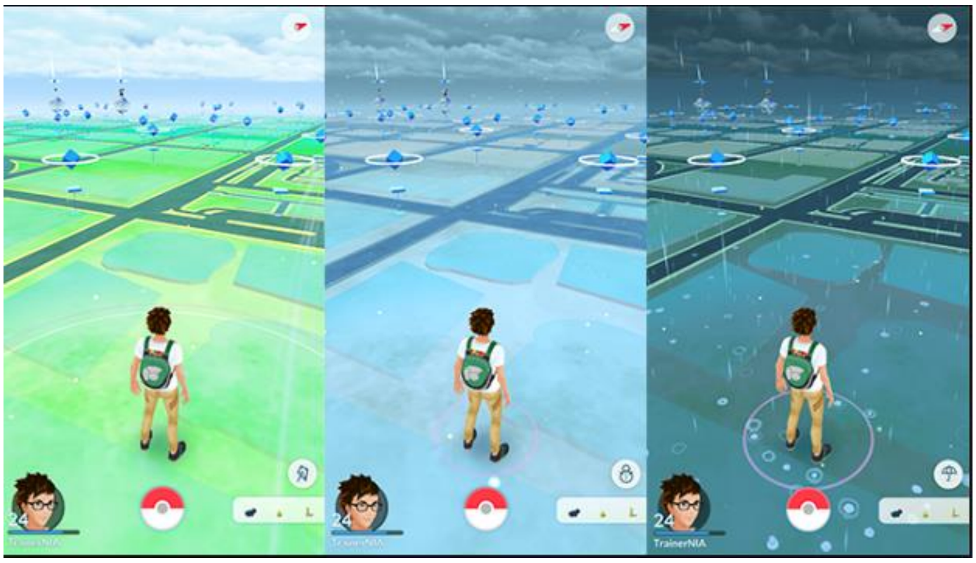


Figure 2.22. Pokemon Go's GPS-based map Interface.

In addition, the main gameplay mechanic is battling wild Pokemon. These battles took place in an AR environment, with the target Pokemon being rendered on a flat surface around the user through his phone's camera.

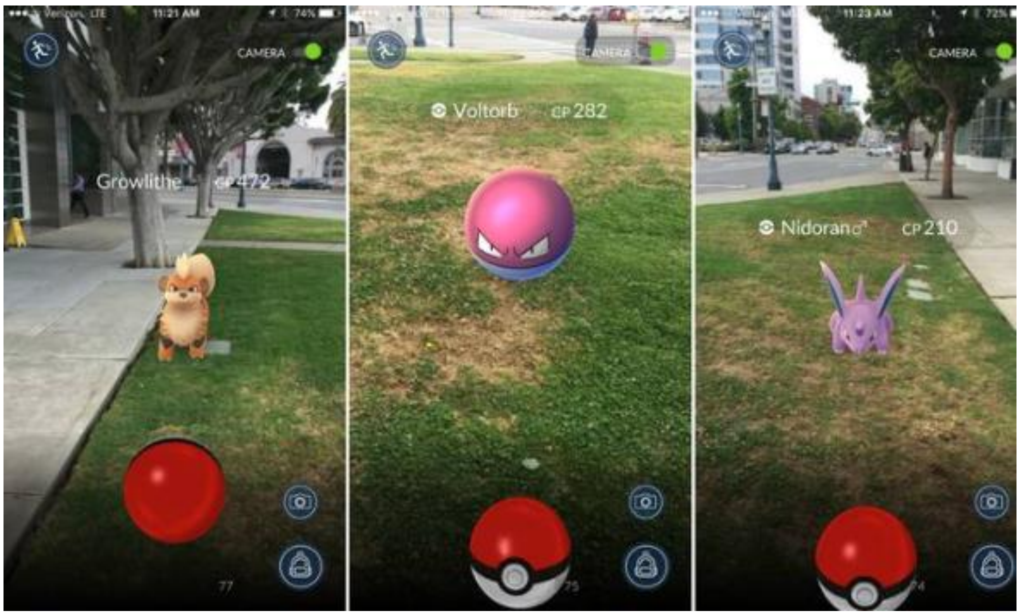


Figure 2.23. Pokemon Go's AR batlle screen.

In Pokemon GO, all necessary interactions with the user are done through the touch screen or automatically with the user's Geolocation.

2.7.2. The Ring brought to life in AR

This is one of many smaller demo applications made by an indie developer in ARKit to show off the available tools of the platform. Based on a popular scene from the movie "The Ring" it features a monster girl emerging from a TV and then walking around in AR. The monster also uses the Geolocation of the user to track him wherever he goes. All of his projects are frequently shared in Facebook and all of them are in his website.

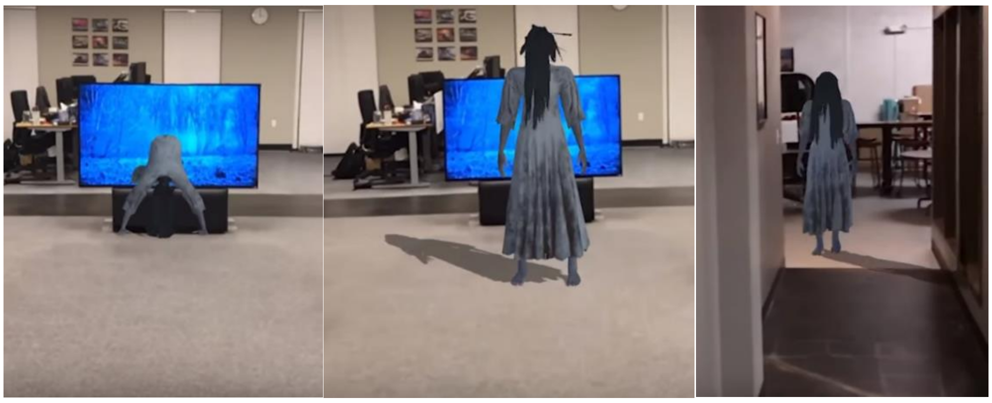


Figure 2.24. The Ring brought to life in AR: Monster girl emerging from the TV (Left), standing up (Middle) and chasing the user (Right).

Unfortunately, other than the Geolocation, there is no direct interaction between the user and any AR content.

2.7.3. Nerf Laser Tag AR Mode

Nerf foam guns have been very popular with people of all ages for the past few years. In 2018, Nerf also announced they were developing plastic guns for use in Laser tag. In addition to traditional Laser tag, Nerf also uses an Android application which includes an AR game for playing in single player. The user mounts their phone on the laser gun itself and uses the screen to aim as drones appear in AR around the user. When the user presses the trigger of the gun he shoots the flying drones for points.

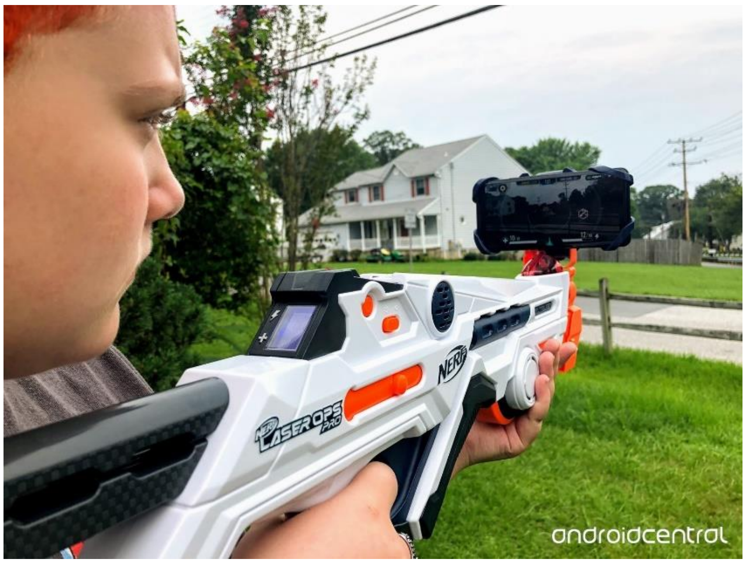


Figure 2.25. Nerf Laser Ops with mounted smartphone for AR gameplay.

2.8. The problem of AR Interaction

Up until now we have mentioned how AR has developed, what the registration problem is and how we attempt to solve it. Earlier AR systems often only cared about solving the registration problem without interacting with AR objects. Nowadays, we have more tools in our disposal, thus newer AR applications also attempt to solve the interaction problem.

Similar to how we want AR visuals to look immersive, the same can be said about interacting with them. Unfortunately, interacting with virtual objects in the real world immersively is just as hard, but not having immersive interactions in AR is slowing down AR's development (Di Capua et al 2011). Below, we will showcase what approaches are used in previously said modern AR, as well as some that could be used.

2.8.1. Touch Screen

Using the touch screen on smartphones is a tried and true method. Smartphones have been a key component in our daily lives for over a decade so most people know how to use a touchscreen instinctively by now. Programmers also use an automated method to receive touch inputs in a simple and tested way.

Due to its simplicity, smartphone apps frequently only use the touch screen for inputs. The drawback to this is it is not immersive in an AR environment. Using the touch screen is by default looking and interacting with virtual objects on a screen. In many simple applications interacting through the touch screen is enough but in general we want AR to be more immersive so we want alternatives to a touch screen. In addition, when in Cardboard mode, the touchscreen becomes inaccessible thus an alternative is mandatory if we want any interaction.

2.8.2. Controllers

Wireless controllers are another tried and true input receiver. Controllers can vary from something simple like a keyboard to something truly immersive.

Multiple new controllers are equipped not only with buttons but also position trackers like accelerometers and gyroscopes. That way we have more freedom to select a controller which can be as immersive as we need it.

Examples of immersive controllers are Hololen's clicker, Magic Leap's control and Nerf's laser gun.

2.8.3. Head Movement Interaction

As we mentioned before Sensor-based tracking allows us to track where the user is looking at. The difference this time is we use that information as inputs. DAQRI's AR mask uses this approach by drawing a semi-transparent circle in the center of the user's view. By pointing the semicircle over an object and staying still for a few seconds, the mask interacts with it. A similar approach is used in the new Hololens in highlighting the object closest to the user's vision's center and then interacting with it through other means.

Similar to touch screens, this is a simple approach, but unlike touch screens, looking at an object to interact with it is both immersive and intuitive to use.

2.8.4. Machine Vision Interaction

As we mentioned before, Machine Vision has improved vastly since the beginning of AR and is widely used to solve the registration problem in the most immersive way possible to date. In addition there have been multiple Machine Vision applications that aim at gesture detection and interaction through Machine Vision. Machine Vision- based inputs owe their rapid improvement mostly to Kinect-Based systems, like the one developed by Chen et al. (2017)

The only problem is Machine Vision for interaction in AR environments is it is not widely used yet. We believe there are two causes for this. On one hand, AR in its current form is still in early prototyping stages and developers still care more about optimizing solutions to the registration problem or adding more features into their platforms. Second, Machine Vision developers don't have much incentive to optimize their algorithms for use in AR environments as there is no general case Machine Vision receivers. Instead, Machine Vision algorithms are tailored around a single application and optimized for that single application. This tends to change, as progress with both the Kinect and Hololens as well as other research like Mäkelä et al. (2017) have shown how natural Machine Vision interactions feel.

Because Machine Vision interactions are frequently replaced by a simpler, less immersive solution for the sake of simplicity, we noticed this is a key flaw of the industry and as such that was a core issue we wanted to solve in our approach.

In Holo-Board, we provide for support for Machine Vision interactions, so future developers may add Machine Vision interactions in Unity-based smartphone AR applications, which is currently non-existent.

3. Requirement Analysis

3.1. Introduction

Holo-Board is a Software Development Environment for Android applications that use Google Cardboard. Currently, Android SDKs for AR applications are built around handheld applications, not Cardboard. When developing a Cardboard SDK we have a few more issues to solve than simple Android applications. We already mentioned the most important of these issues being how we interact with AR objects, and this is a main focus of our application.

Our application should be an Application manager that hosts graphics, applications and programming tools specifically tailored around Cardboard apps, as well as provide an interface through which External input sources such as Machine vision.

In this section we will outline what requirements we set for completing our project and then we will outline the hardware and software we used in our development.

3.2 Requirements

Initially, we want Holo-Board to be a complete project any programmer can adjust to his needs, not just a collection of tools to be used by others. As such Holo-Board must:

- Be a full Android Application.
- All of its parts should be easily edited/reprogrammed.
- All of its parts should be independent of one another and replaceable as necessary
- Have some simple Demos which programmers can run to get an idea of how Holo-Board operates

As an application manager we also need:

- Scripts that act as Managers and mediators between smaller programs
- Clear cut classification of said smaller programs consisting of what they are and how they communicate with each other

In addition, to assist with Cardboard app optimization we need:

- Some way to handle 2 fully synchronized camera views in one screen (one for each eye)
- A single camera mode would be good for debugging
- Some manager that assists with showing canvas objects on both views, without the programmers having to set them twice

Finally, we need fluid support for interaction methods. Thus we also need:

- An input manager for non-generic inputs, like Machine Vision.
- Pre-built support for some type of Machine Vision interaction

- Preferably, using a controller as an alternative will be helpful, especially for development and debugging

3.3. Platform Information

Holo-board was fully developed in Unity. The version we used was Unity 2017.3. This version was selected due to compatibility issues with our version of ARToolkit with newer Unity releases.

For camera settings and Marker-based tracking ARToolkit was used. The version we used was ARToolkit 5.3.2 which was the latest stable version DAQRI published before shutting down ARToolkit's site. This version of ARToolkit was designed as a plugin to Unity version 5.X, but we found out it is still compatible with Unity 2017.X. We also used the standalone ARToolkit 5.3.2. library as it includes camera calibration tools and marker pattern generators for custom square markers as well as NFT markers.

Since we need to compile our application for Android smartphones we also used Android Studio. Since we wanted to develop our application for an Android 8.0 device we used Android Studio January 2018 version. Later on, we upgraded to the March 2018 version but compiling for Android 8.0 was problematic in that update so we rolled back and kept the January 2018 version throughout our development.

For future releases of Unity, it is recommended to switch plugins to ARToolkitX which was updated for use in Unity 2018.X, sadly after our project was completed. Holo-Board is also compatible with any phone which supports Android 3.X and above, even though our testing was done mostly in an Android 8.0 phone.

To develop Holo-Board a Dell Inspiron 15 3000 series was used, with Windows 7 OS. This laptop has a dual-core Intel Core I5 CPU @3,2GHz, 4GB RAM, a 0.5 Mpixel front camera used for testing and Onboard GPU. The OS is independent of our application since Unity is a cross-platform Engine.

Testing was done on two phones. The first was a Xiaomi Mi A1, having an Octa-core Snapdragon CPU @2.02GHz, Full HD screen and dual back camera for Full HD camera capture. It also is a mid-budget smartphone (250 euros) designed in 2017, upgradeable to Android 8.0. The second phone was a more dated Meizu M3S, a low budget (100 euros) phone designed in 2015 with an Octa-core processor, with 4 2GHz and 4 1GHz cores, no Full HD screen or camera. The Meizu M3S used Android 5.0.

We also wanted to integrate a controller in our application. A Dualshock 4 PS4 controller was selected because it is supported in all Android versions, with fixed keymapping for all.

For Machine Vision interactions we printed a Hiro marker, included in ARToolkit's library with dimensions 1,5cm*1,5cm The marker was then glued with a magnet behind it and attached to a ring worn on the index finger of the user.

4. Use Cases

Even though Holo-Board is primarily a tool for programmers, it is also designed as a full standalone End-user's experience. As a result, we not only have in mind use cases for the programmers that use Holo-Board, but also use cases for End-users that execute the demo application or implementations that want to follow our base architecture.

Below we will first present the use cases of a programmer inside the development environment and then an End-user's use cases when executing the demo app.

4.1. Programmer's Use Cases

The programmer decides what type of application to make:

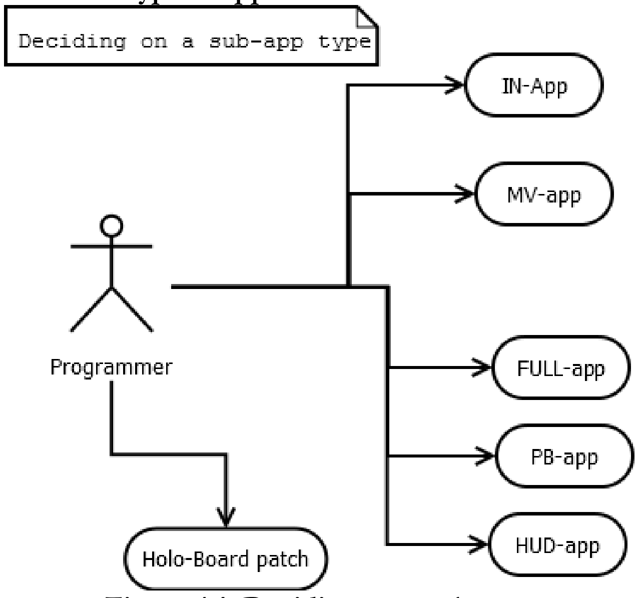


Figure 4.1. Deciding on a sub-app.

Before developing anything a programmer must decide what type of application he wants to make. Since Holo-Board is a full application we call all its smaller components sub-applications. A sub-application, or Sub-app for short, is an application with a specific purpose. In general a programmer may want to develop an application in 2 distinct categories, graphics or inputs, or develop a patch for Holo-Board.

A graphics programmer will have to decide between 3 options for his application: a Full application (FULL-App), a Heads-Up Display application (HUD-App) or a Position-Based application (PB-App). All three categories are graphics sub-apps with different capabilities and a distinct way of execution which will be explained below.

An input developer can develop an interaction sub-application classified either as a Machine Vision application (MV-app) or non-Machine Vision, simply put an Input application (IN-App). While both of these sub-apps are handled very similarly in Holo-Board it is important to distinguish them as MV-Apps may require more resources from Holo-Board and/or their integration into Unity may be more complicated.

Finally, since Holo-Board is not a perfect system by any means, a programmer may freely choose not to develop a sub-app but patch any of the already existing systems of Holo-Board.

Designing a HUD-app

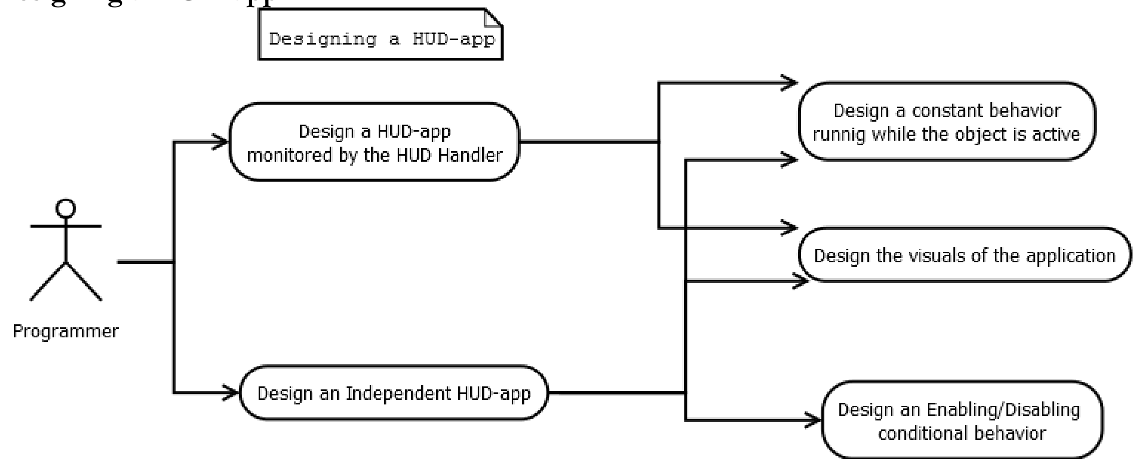

Designing a HUD-app
Programmer
Design a HUD-app monitored by the HUD Handler
Design an Independent HUD-app
Design a constant behavior runnig while the object is active
Design the visuals of the application
Design an Enabling/Disabling conditional behavior

Figure 4.2. Designing a HUD-app.

Heads-Up Display applications (HUD-Apps) are the simplest graphics applications of the three. These are 2D canvas applications that exist in an overlay of the real world's camera. Holo-Board has a HUD Handler (HUD-Han) script which can automate when these HUD-apps are visible or hidden.

If the programmer decides to use the HUD-Han then he needs to design the behavior of the application and the visuals. The programmer does not need to worry about when the application is considered active as that will be handles automatically, thus he can focus on optimizing the behavior itself.

On the other hand, if the user wants to execute his sub-app on a specific condition, and not when the whole HUD is shown, he can opt to not use the pre-build HUD-Han. In that case, the programmer also needs to program when and how his sub-app is executed, for example by adding a new button.

Designing a FULL-app

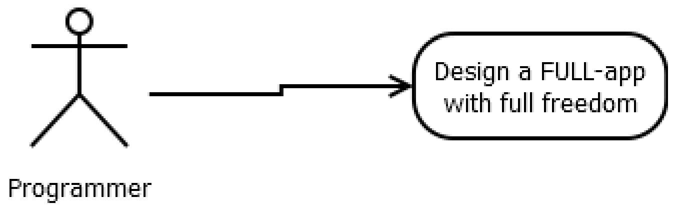


Figure 4.3. Designing a FULL-app.

FULL-Apps are the least restrictive sub-apps. When a FULL-App is executed, all HUD and Menu elements are hidden and a FULL-app has no restrictions to its execution, while all of Holo-Board's tools remain usable like MV-based inputs or ARToolkit.

Designing a PB-app

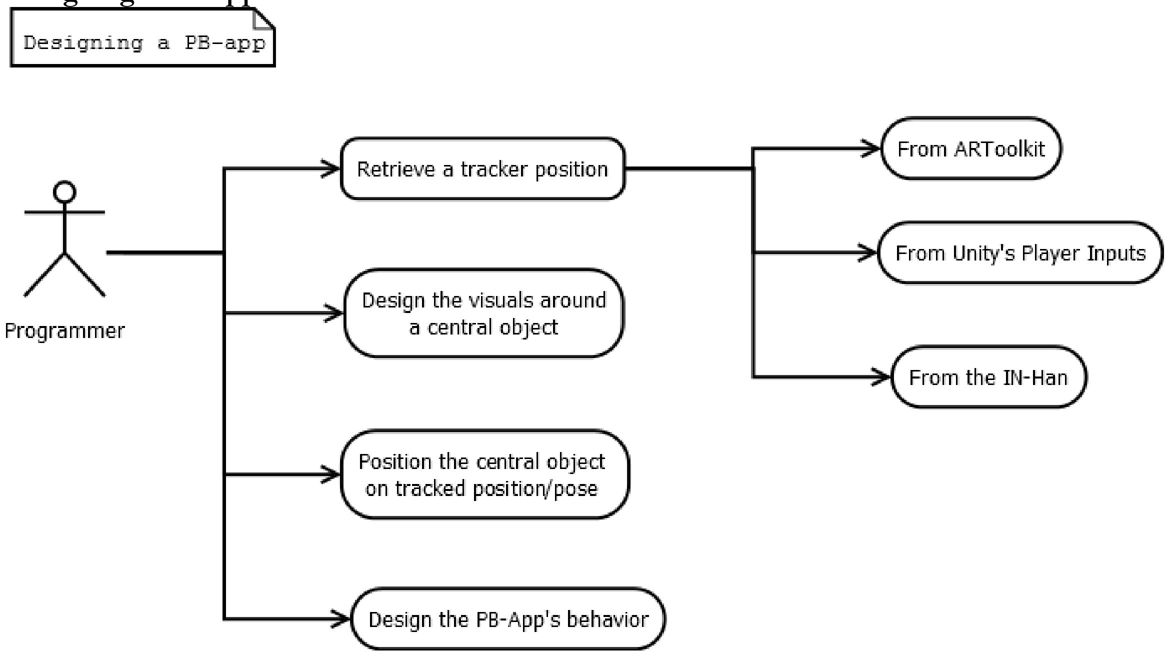

Designing a PB-app
Programmer
Retrieve a tracker position
Design the visuals around a central object
Position the central object on tracked position/pose
Design the PB-App's behavior
From ARToolkit
From Unity's Player Inputs
From the IN-Han

Figure 4.4. Designing a PB-app.

Position-Based applications (PB-Apps) are a more specific type of sub-app. PB-apps are graphics applications built around a specific point in virtual space. Through the registration method we detect where this central point refers to in the real world and position the whole PB-app there.

A programmer designing a PB-app must first decide on a tracker through which the registration is achieved. In the way Unity and Holo-Board work, the trackers are interchangeable at any time. In some cases when using standard controllers, Unity supports tracking through its basic Player Inputs. Two such controllers are Leap Motion or Oculus controllers.

In Holo-Board we provide two additional tools to receive tracker data from. First, a programmer may use ARToolkit's Marker-Based tracking, simply by creating an AR Tracked Object and leaving it up to ARToolkit. A second tool Holo-Board provides in the Input Handler (IN-Han). The IN-Han hosts a Dictionary of tracked positions which are written by IN-Apps and MV-Apps to be used by graphics applications.

After the programmer has decided on a tracker he must then design all the visuals of his sub-app around the tracked position. Finally he must then program the behavior of the sub-app.

A PB-App is free to be executed as part of a FULL-App or run at all times. It is even possible to link a PB-App to the HUD-Han as it is not restricted and it will be executed along with any HUD-Apps.

Designing an IN-app

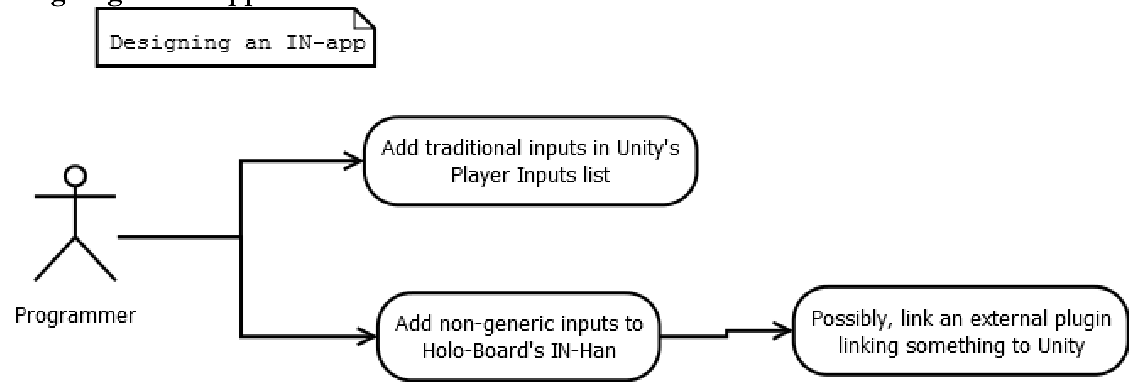

Designing an IN-app
Programmer
Add traditional inputs in Unity's Player Inputs list
Add non-generic inputs to Holo-Board's IN-Han
Possibly, link an external plugin linking something to Unity

Figure 4.5. Designing an IN-app.

Input applications (IN-Apps) are sub-apps that read user inputs without the use of Machine Vision, usually these inputs being the readings of a controller or sensor. In most cases, these controllers are supported by Unity's Player Inputs where there is plenty of documentation to assist the programmer elsewhere. When a controller or sensor is not supported by Unity's traditional input receivers, programmers can still use the Input Handler (IN-Han) Holo-Board provides. Usually, this will mean programming an external to Unity library and then simply pass necessary values to the IN-Han in a specific format.

Designing a MV-app

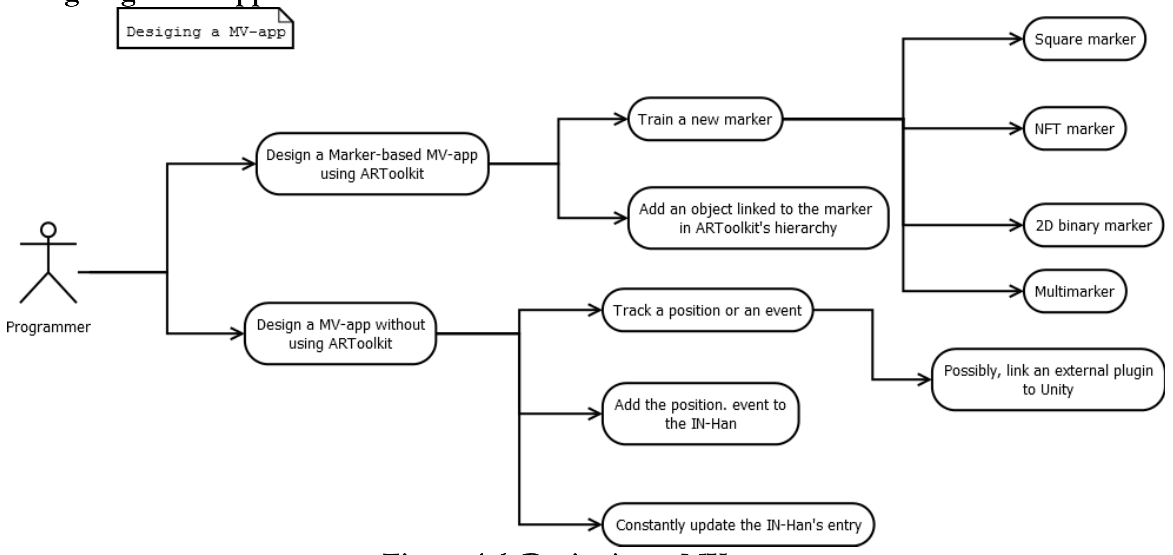

Desiging a MV-app
Programmer
Design a Marker-based MV-app using ARToolkit
Design a MV-app without using ARToolkit
Train a new marker
Add an object linked to the marker in ARToolkit's hierarchy
Square marker
NFT marker
2D binary marker
Multimarker
Track a position or an event
Add the position. event to the IN-Han
Constantly update the IN-Han's entry
Possibly, link an external plugin to Unity

Figure 4.6. Designing a MV-app.

Machine Vision applications (MV-Apps) are a specific type of input receiver. These sub-apps usually generate software-based inputs, not hardware, which Unity dislikes for its traditional inputs. This was the reason we developed Holo-Board's Input Handler.

Programmers have two options when developing a MV-App. The simpler solution is to use ARToolkit's Marker-Based tracking and retrain it. In that case the programmer can select a marker from a variety of types and use ARToolkit's pattern generators to scan the marker and produce a pattern data file. That file can then be imported to Unity and used by ARToolkit.

The second option is developing a new MV-app from scratch. It is possible this MV-app is not developed in Unity but is an external program using libraries that excel in Machine Vision, for example OpenCV. MV-apps need only connect with the IN-Han and send any tracked data in a specific format and the IN-Han will connect to any graphics applications. As long as a MV-app is running it should constantly send updated values to the IN-Han anytime they are updated.

Patching Holo-Board

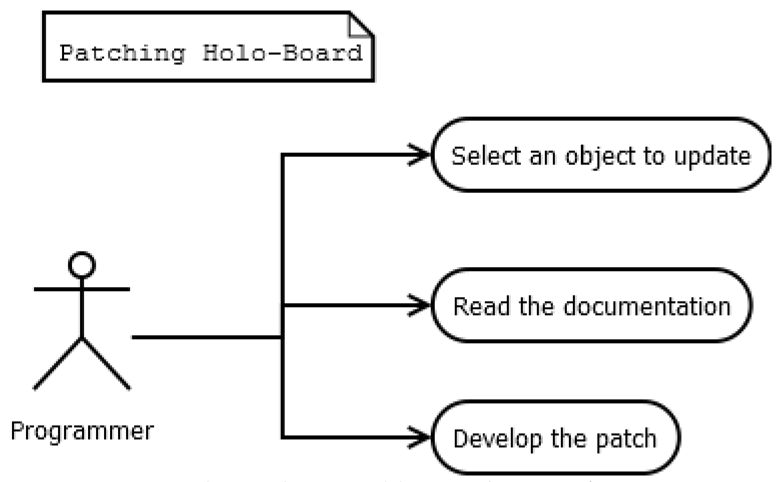


Figure 4.7. Patching Holo-Board.

Instead of sub-apps, it is also necessary for programmers to constantly keep Holo-Board itself up to date. If a programmer decides he wants to reprogram or patch any existing part of Holo-Board he is free to do so. In the next chapters we provide a full documentation of Holo-Board's tools, both how they are used and how they were programmed. Any programmer can use these as reference and develop his own version of Holo-Board's tools.

4.2. End-User Use Cases

The End-user's use cases are quite different from the programmer's. We consider these use cases from the moment a user executes the base demo app. If programmers develop their sub-apps and do not change the basic architecture of Holo-Board all of these use-cases remain the same.

The user executes the base Holo-Board app

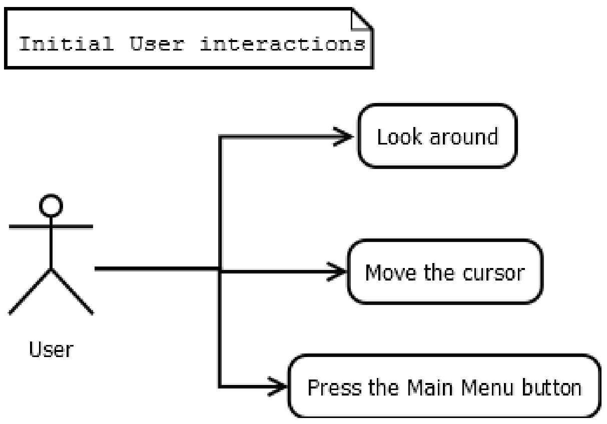


Figure 4.8. Initial user interactions.

When a user initially starts Holo-Board, he is greeted with a free view through the phone's camera. The only virtual objects visible are a semi-transparent button in the middle of the screen and a cursor. The cursor can be moved around the screen using either a Dualshock 4 controller or a Hiro marker. The cursor will follow the marker when visible, otherwise will move according to the DS4's left analog stick.

At any point, the user can click the semi-transparent button, either by moving the cursor on top of the button and holding it still for a few seconds or by pressing the PS button on the DS4 controller. This button will enable the Main Menu through which everything is executed.

The user opens the main menu

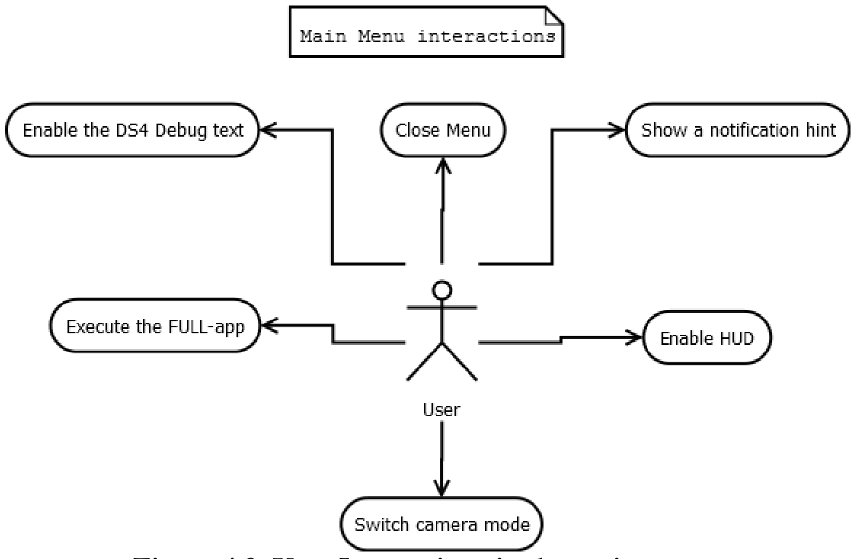


Figure 4.9. User Interactions in the main menu.

The Main Menu is presented as an overlay to the user's screen. For the Demo app we have a collection of six buttons each performing a specific functionality. Both the number of buttons and their functionality may be different based on the application, but for the demo app this is static.

The user can interact with any button similar to how he interacted with the central button. Each button is clickable by moving the cursor on top of it and holding still for a few seconds, and also each button is mapped to a DS4 button. Four of the buttons are color coded and positioned in a cross-shape, each mapped to one of the basic face buttons (Cross, circle, triangle and square) The other two buttons are positioned on the top left and top right corners and mapped to the two bumpers (L1 and R1).

As long as the main menu is visible, the central button remains on the screen but pressing it will have no effect.

Pressing the Close menu button

If the Close Menu button is pressed, all Main Menu buttons are hidden.

Pressing the DS4 Debug Text button

This button shows an overlay text indicating the values read from the DS4 controller in real time. This is a sub-app used when developing Holo-Board to map the correct buttons, but is still useful to get an idea of what values each button outputs.

Pressing the sample notification hint button

This button shows a simple hint text to the user using the Notification Text tool provided by Holo-Board.

Pressing the switch camera mode button

Using the switch camera button while on dual screen mode, which is the default, switches the perspective to a single camera mode. This is useful when we want to try Holo-Board but do not have a Cardboard mask or we want to debug something. Pressing the Camera Mode button again switches us back to dual screen perspective.

Pressing the Toggle HUD button

Pressing the HUD button toggles on all HUD-Apps linked to the HUD-Han. Pressing the same button while the HUD-Apps are active disables them.

Pressing the Execute FULL-App button

The Execute FULL-App button is the only button that switches the user's use case scenarios. It disables all Main Menu and HUD objects and enables the FULL-App giving it full control of the scene.

The user executes a FULL-app

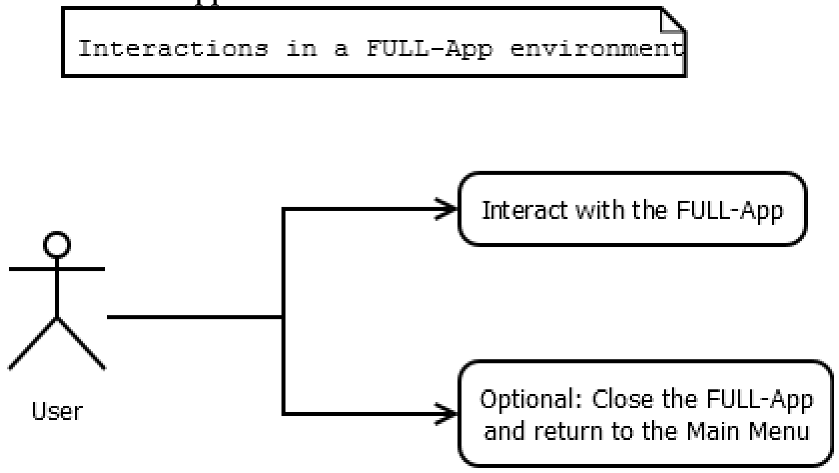

Interactions in a FULL-App environment
User
Interact with the FULL-App
Optional: Close the FULL-App and return to the Main Menu

Figure 4.10. FULL-App interactions.

A FULL-App is given full freedom as to how the user interacts with it. Thus, we cannot give specific use cases in this environment. The only interactions available in the Demo app is a button that returns us to the Main Menu screen.

5. Using Holo-Board for Development

As Holo-Board is an SDK usable by anyone it is important to have clear documentation about what can be done with Holo-Board and how.

Below we will provide a full documentation of Holo-Board. We will explain how every part of Holo-Board works and how programmers can use everything correctly.

5.1. Executing Holo-Board's Base App

Executing Holo-board can be done on any Android smartphone with Android 3.X or higher or inside Unity in debug mode.

5.1.1. Setting up for Smartphone Execution

To run on Android we just load the HoloBoard.apk file in the phone, install and run it. It is also recommended to have a Cardboard mask with an opening behind the camera and see the phone through that. To interact with the application we need one of two methods of inputs, either a Machine Vision marker or a Bluetooth DS4 controller.

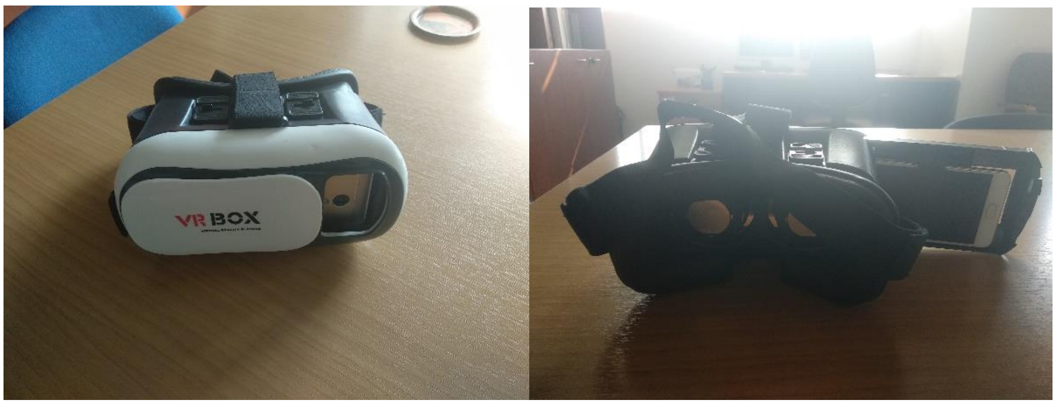


Figure 5.1. Cardboard mask with a camera opening front (Left) and back (Right).

To use a Bluetooth controller, it is enough to simply connect it via Bluetooth to the smartphone and Unity will automatically map it to the correct buttons. As there is no general mapping method all button mapping done and explained below is for official Sony PS4 Dualshock 4 controllers only. Alternatively, other controllers or a Bluetooth keyboard will probably work for moving the cursor, but the keymapping will be different for other buttons.

Figure 5.2. Dualshock 4 Controller.

For marker-based inputs it is recommended to print a Hiro marker of 2x2 cm size. For development that marker was mounted onto a ring to track a finger. The Hiro marker, as well as more markers are included in Holo-Board's Assets for future development.

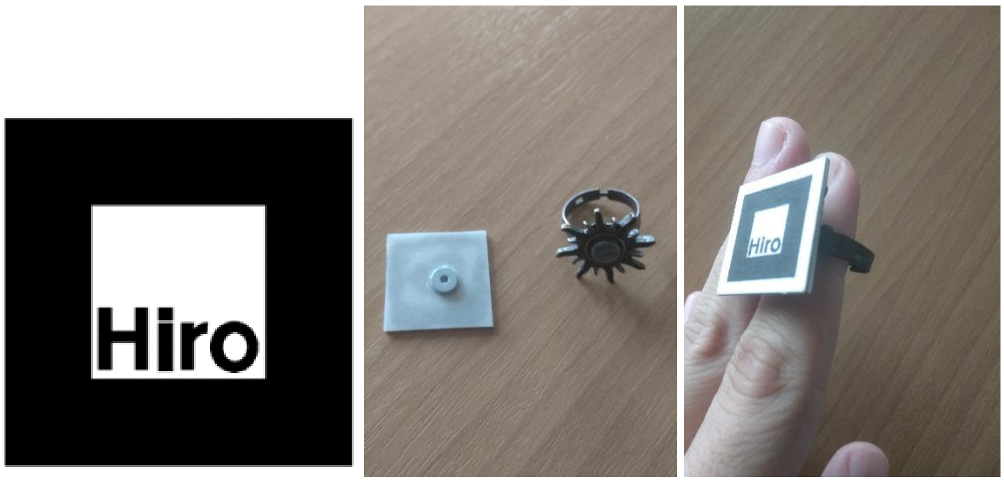


Figure 5.3. Hiro marker (Left). Our Hiro marker mounted on a ring (Middle and Right).

5.1.2. Setting up for Unity Debug Execution

Executing Holo-Board in a computer is done through Unity in debug mode. It is recommended to use Unity Version 2017.3 where Holo-Board was first developed on. Information on how to install Unity can be found on the official website.

When opening Unity open the pre-existing Holo-board project or create a new project, import all the Assets, and double click the HoloBoardMain.unity scene file. This is the central scene of Holo-board where everything is already set up. Then, by pressing the Play button, ARToolkit will open two windows to set the camera parameters with and execute.

For the execution, we can either use the same Machine Vision marker mentioned above or the keyboard's arrow keys (or WASD keys) to move the cursor. As the PS4 controller's mapping is not the same as in the smartphone, the left analog stick can still be used to move the cursor but the other buttons will probably not work.

5.1.3. Running the Demo Holo-Board App

When Holo-Board first runs it will show a message that the camera is loading and we can see a semi-transparent button in the center of the screen. If we see the Hiro marker through the camera feed we can see a cursor appear on top of it. We can move the marker on top of the button and hold it still for 2 seconds and the button will be pressed, showing us the whole menu. Similarly, we can press any other button to execute something and close the menu. Alternatively, we can use the left analog stick of the PS4 controller, the WASD keys or the arrow keys of the keyboard. In addition, we can also hold the PS button on the controller to click the central button and then use the face buttons (square, X, Circle and Triangle) and bumpers (L1, R1) to click the menu's buttons. The Menu's layout represents the shape of the PS4 controller as to make it easier to press the correct buttons instinctively. If at any point we try to use the touch screen Holo-Board will throw a warning, as pressing a button will desync the two screen views.

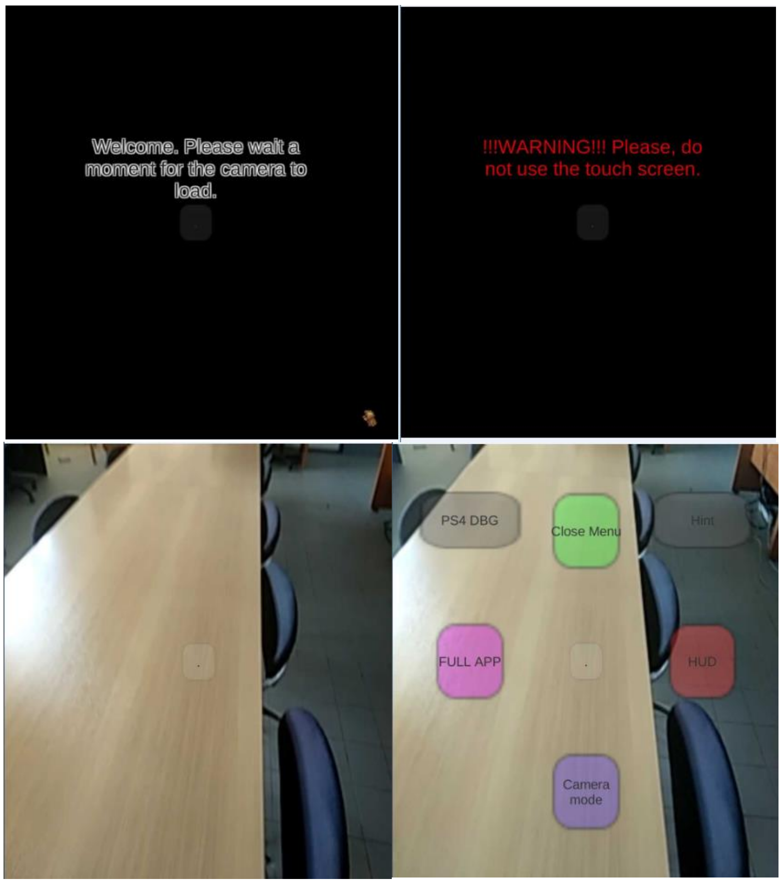

Welcome. Please wait a moment for the camera to load.
!!!WARNING!!! Please, do not use the touch screen.
PS4 DBG
Close Menu
Hint
FULL APP
HUD
Camera mode

Figure 5.4. Intro screen while the camera loads (Top Left). Warning message if the touch screen is used (Top Right). The camera is loaded (Bottom Left).The Main Menu (Bottom Right).

By clicking the "Camera Mode" button (assigned to X), we switch form a two camera display to a single widescreen one, which helps when debugging outside of the AR mask. Pressing the same button again while on single screen resets the two-camera perspective.

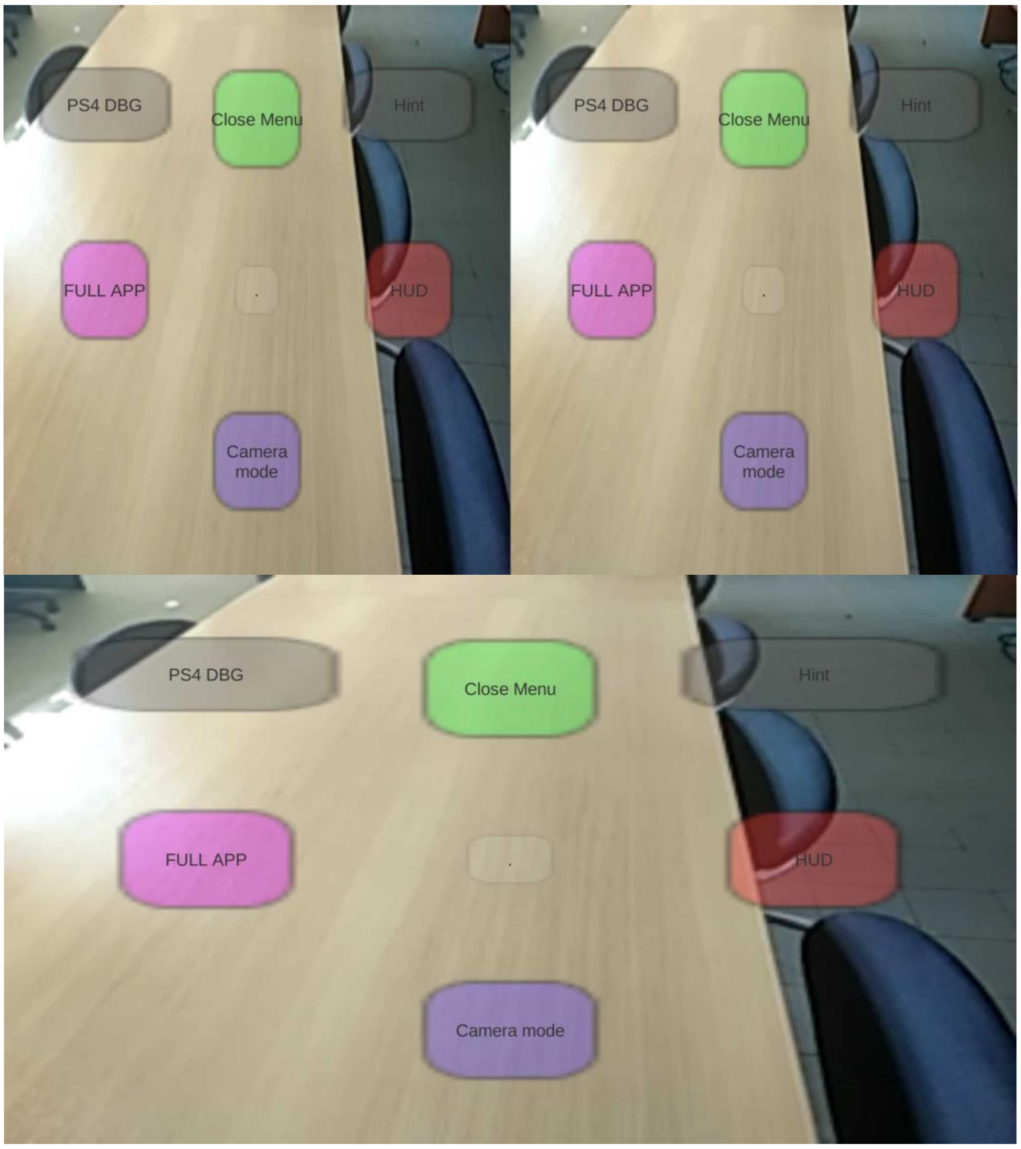


Figure 5.5. Dual camera view (Top). Single camera view (Bottom).

Using the two buttons in the middle (assigned to Square and Circle) executes two demo applications. The right one, dubbed HUD, enables some Heads-up objects such as a battery indicator and accelerometer value monitoring. The left one, dubbed FULL APP, is a skeleton interface that simply hides the menu and HUD to clear the screen for any other potential End-user app, leaving only one button that returns us to the main menu.

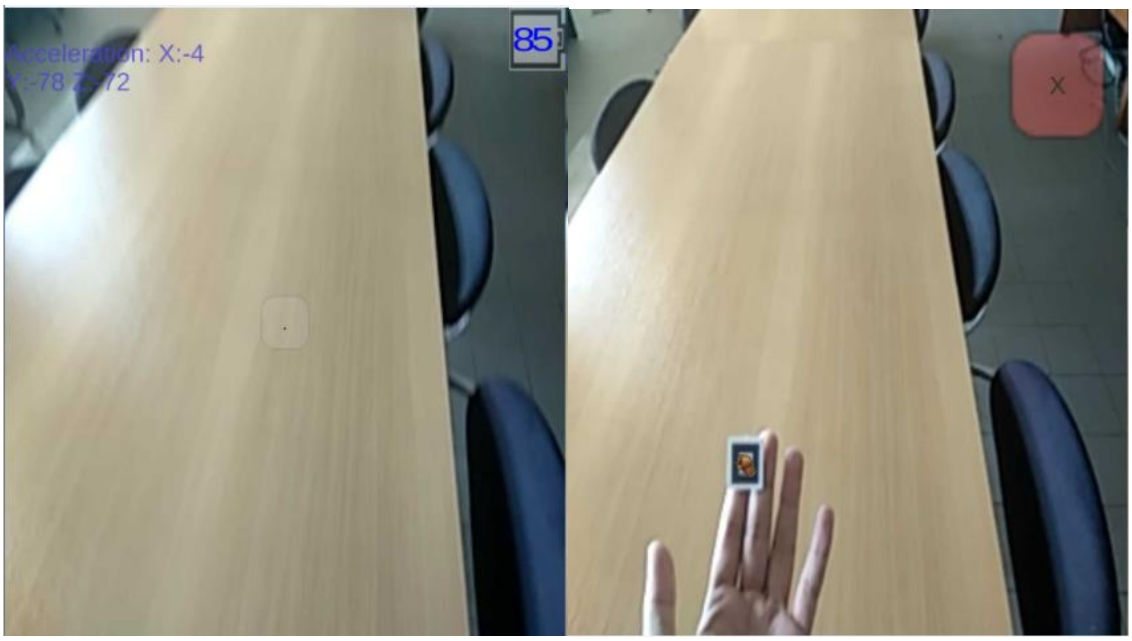
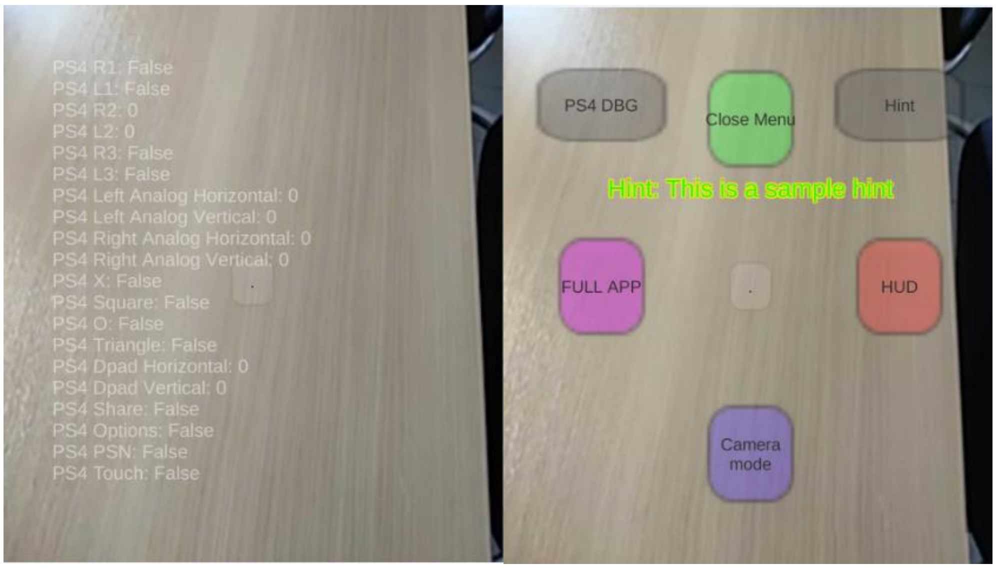


Figure 5.6. The HUD-App demo (Left). The FULL-App demo (Right).

Finally, the middle of the top three buttons, dubbed "Close Menu", (assigned to Triangle) simply hides the main menu from view while the other two buttons (assigned to R1 and L1) enable some debugging tools for the DS4 controller and notification text.



```
PS4 R1: False
PS4 L1: False
PS4 R2: 0
PS4 L2: 0
PS4 R3: False
PS4 L3: False
PS4 Left Analog Horizontal: 0
PS4 Left Analog Vertical: 0
PS4 Right Analog Horizontal: 0
PS4 Right Analog Vertical: 0
PS4 X: False
PS4 Square: False
PS4 O: False
PS4 Triangle: False
PS4 Dpad Horizontal: 0
PS4 Dpad Vertical: 0
PS4 Share: False
PS4 Options: False
PS4 PSN: False
PS4 Touch: False
```

Hint: This is a sample hint

Figure 5.7. Dualshock 4 debug text (Left). Sample hint text (Right).

5.2. The Basics of Holo-Board

5.2.1. Setting up the Basic Tools

Holo-Board is a complete Unity Project. As such the first step is installing Unity on your computer. It is recommended to use Unity Version 2017.3 where Holo-Board was first developed on. In future releases of Unity, ARToolkit may need to be changed to a newer version of ARToolkitX. Information on how to install Unity can be found on the official website. ARToolkit is already integrated into Holo-Board so it is not necessary to import it manually, but we use some tools provided in the standalone ARToolkit package which is downloadable from GitHub.

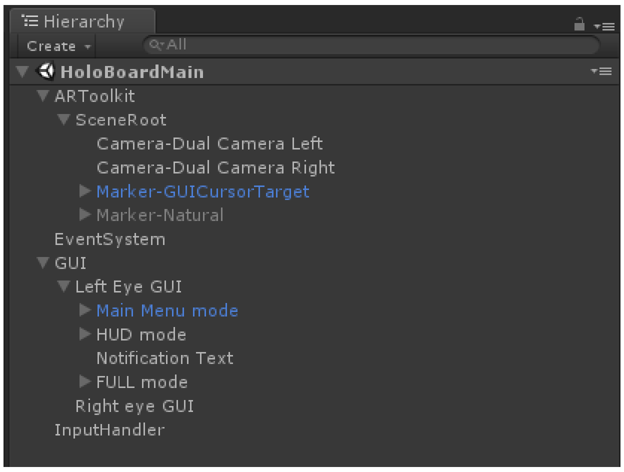
To compile for Android smartphones Android Studio is also required. For Holo-Board Android Studio version January 2018 was used. As the March 2018 update introduced errors in compiling for certain phones with Android 8.0, the whole development process was complete in version January 2018, but future versions should also work correctly. For more information on how to install Android studio visit the official site.

The final step is to setup the target smartphone in Developer mode. This consists of unlocking the hidden Developer settings menu and then enabling USB debugging. To unlock the hidden menu requires tapping the Kernel version in the Smartphone settings 8 times. If this is not the case, search online how to enable Developer mode for your smartphone's model specifically.

Finally, when opening Unity open the pre-existing Holo-board project or create a new project, import all the Assets, and double click the HoloBoardMain.unity scene file. This is the central scene of Holo-board where everything is already set up.

5.2.2. The Basic Hierarchy

When we open up the main scene, there are 3 key objects present: ARToolkit, GUI and InputHandler. The Event System is automatically created by Unity.



```
▼ HoloBoardMain
  ▼ ARToolkit
    ▼ SceneRoot
        Camera-Dual Camera Left
        Camera-Dual Camera Right
      ▶ Marker-GUICursorTarget
      ▶ Marker-Natural
    EventSystem
  ▼ GUI
    ▼ Left Eye GUI
      ▶ Main Menu mode
      ▶ HUD mode
        Notification Text
      ▶ FULL mode
      Right eye GUI
    InputHandler
```

Figure 5.8. The basic hierarchy.

Below ARToolkit is the whole tree necessary for anything related to ARToolkit. We will analyze all of ARToolkit's objects and scripts later.

The Input Handler is responsible for non-generic Unity inputs and only holds one script for its behavior. This object is not used directly but accessed through other objects. How this is used will be explained later.

Finally, the GUI holds two canvases for the overlay of the screen, one above each eye. Take note, that only the Left Eye GUI has children objects on its canvas and the Right

Eye GUI is empty. The Right Eye GUI is setup to dynamically mirror the Left Eye GUI at runtime, so programmers have to setup only one canvas and the second is synced automatically.

5.2.3. Holo-Board's Architecture

As Holo-Board is not based on a pre-existing architecture and is completely organized from scratch it is best we first specify what its components are and how they interact with each other.

Holo-board works as a distributed system that links multiple smaller standalone programs, referred to as Sub-applications, using some scripts that connect and manage how and when they work, referred to as Handlers.

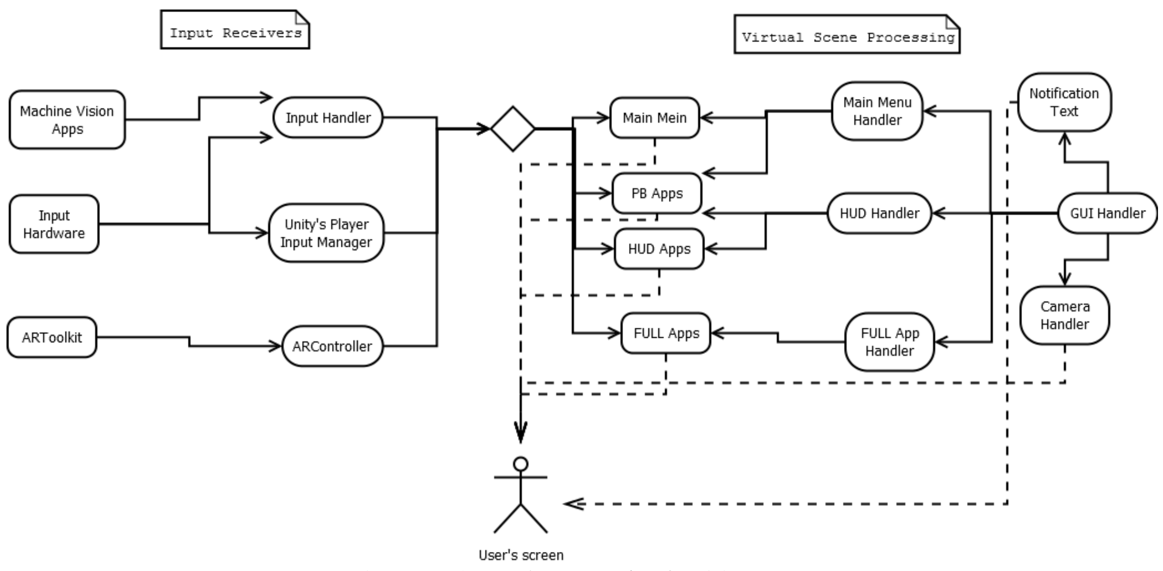


Figure 5.9. Holo-Board's Architecture.

Sub-applications are all smaller programs which are executed through Holo-board. They can be freely added or removed depending on the developer's needs without breaking the core Holo-board system. Because these Sub-applications must be linked correctly through the Handlers they are separated into different categories, each with different use cases and rule sets, explained in the previous section. For receiving inputs there are Input Applications (IN-Apps) and Machine Vision Applications (MV-Apps) while graphics applications are either Heads-Up Display Applications (HUD-Apps), Full Applications (FULL-Apps) or Position-based Applications (PB-Apps).

IN-Apps and MV-Apps are responsible for receiving inputs from the user and sending them to the Input Handler (IN-Han). Holo-board's graphics applications should not care where their inputs come from, so IN-Apps and MV-Apps read the inputs in any way they want (ex. using Machine vision or mapping a controller to certain inputs) and send them in a specific format to the IN-Hand. These apps can also be developed outside of the Unity engine and later be linked with the native handlers using middleware in order to use better tools such as OpenCV# or Native Android code.

HUD-Apps are applications that are shown on the user's screen as an overlay to the camera feed. They are simple 2D apps that any developer can just lay out on one canvas and then let the HUD Handler (HUD-Han) manage how and when they are executed, regardless if they must be shown over a single camera or dual cameras for Cardboard mode.

PB-Apps are based on ARToolkit's functionality and are 3D graphical applications that show up relative to a point in screen space. Currently, these work using ARToolkit's marker detection as base points but this can easily be replaced with any MV-App that recognizes a specific point in the camera's view.

Finally, FULL-Apps are complete applications where when executing them both the Main Menu and all HUD objects are disabled to give the FULL-app complete visual freedom. The only GUI elements left are a closing button and a MV cursor to be able to return to the main menu, but even these are optional. All non-visual tools remain intact and can be used freely in the FULL-App

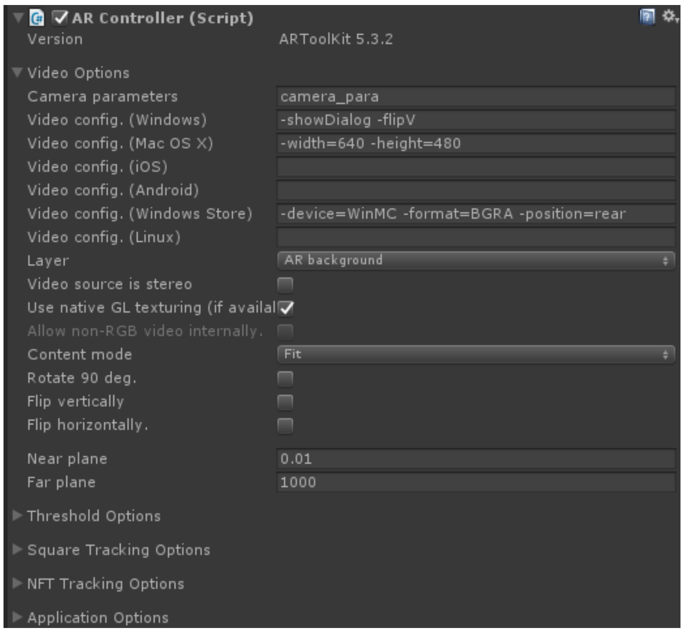
5.3. Using ARToolkit on Holo-Board

ARToolkit provides Holo-Board with a few useful tools regarding scene setup and Marker-based tracking. While these tools can be swapped at any time with better ones, ARToolkit provides enough freedom to repurpose them in a variety of ways.

Since DAQRI closed the official documentation page, we will provide documentation for a few key parts of Holo-Board. Most parts of ARToolkit can be adjusted through the Unity inspector, but in some cases we execute scripts provided in the standalone ARToolkit package downloaded from GitHub.

5.3.1. Camera and Scene Settings Through ARToolkit

Setting up all basic parameters of ARToolkit is done on the ARToolkit object. This object holds the AR Controller script trough which we can setup all the basic settings of ARToolkit. It also hosts all AR Marker scripts for tracked markers which will be explained below.

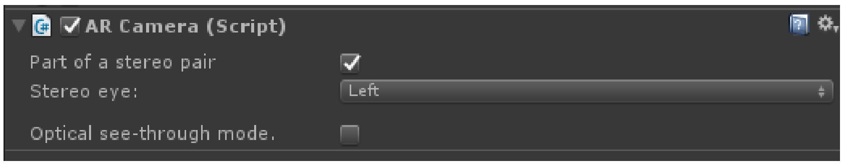


AR Controller (Script)	✓
Version	ARToolKit 5.3.2
Video Options	
Camera parameters	camera_para
Video config. (Windows)	-showDialog -flipV
Video config. (Mac OS X)	-width=640 -height=480
Video config. (iOS)	
Video config. (Android)	
Video config. (Windows Store)	-device=WinMC -format=BGRA -position=rear
Video config. (Linux)	
Layer	AR background
Video source is stereo	☐
Use native GL texturing (if availal	✓
Allow non-RGB video internally.	☐
Content mode	Fit
Rotate 90 deg.	☐
Flip vertically	☐
Flip horizontally.	☐
Near plane	0.01
Far plane	1000
Threshold Options	
Square Tracking Options	
NFT Tracking Options	
Application Options	

Figure 5.10. AR Controller script Inspector window.

As a child to the ARToolkit object, we have the Scene root, which acts as the center of the virtual scene. Everything related to ARToolkit is set as a child to that Scene Root.

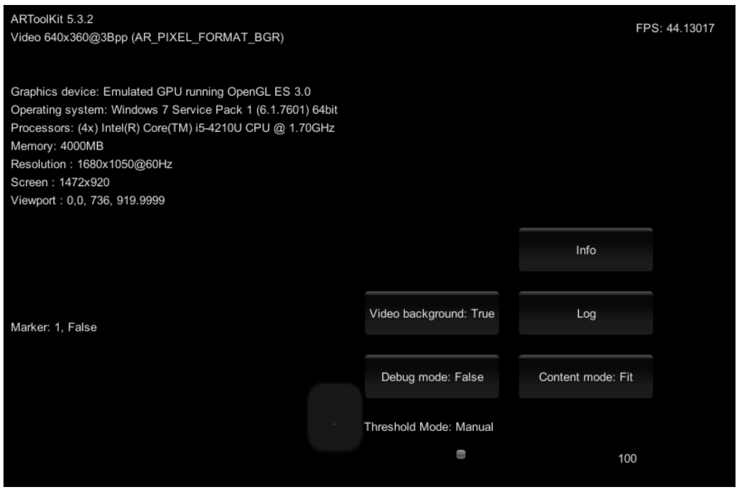
While ARToolkit is commonly used for its marker detection, it also helps us in setting up the cameras. Adding a camera to our virtual scene and attaching the AR Camera script to it will create an AR Background camera showing real-world feed to our scene and sync it with the virtual camera.



AR Camera (Script)	✓
Part of a stereo pair	✓
Stereo eye:	Left
Optical see-through mode.	☐

Figure 5.11. AR Camera inspector window.

Also, when debugging on a computer we can setup the correct camera parameters when we run debug mode. Similarly, while the application is running we can open a debug and settings menu provided by ARToolkit through which we can see the console, set camera parameters and adjust the thresholding when tracking markers amongst other things. This menu is enabled when pressing either Enter on the keyboard (computer) or the R3 button on the DS4 controller (smartphone).



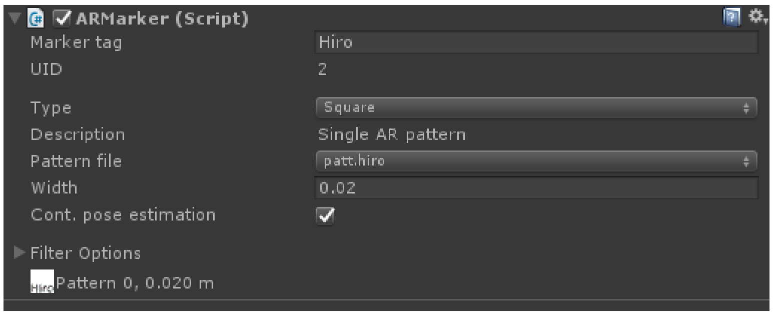
```
ARToolKit 5.3.2
Video 640x360@3Bpp (AR_PIXEL_FORMAT_BGR)                FPS: 44.13017

Graphics device: Emulated GPU running OpenGL ES 3.0
Operating system: Windows 7 Service Pack 1 (6.1.7601) 64bit
Processors: (4x) Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
Memory: 4000MB
Resolution : 1680x1050@60Hz
Screen : 1472x920
Viewport : 0,0, 736, 919.9999

Marker: 1, False
```

Figure 5.12. ARToolkit on runtime debug menu.

5.3.2. ARToolkit's Marker-Based Tracking

ARToolkit's main purpose is tracking markers using Machine vision. When a marker is found, the virtual cameras are positioned relatie to its position in the virtual space and any graphics objects below the marker are enabled. This is how traditional PB-Apps are made.

When we want ARToolkit to recognize a marker, we have to attach an AR Marker script to the ARToolkit core object. Through this script we select the type, pattern and size of the marker and then set a tag for this marker. Later, we will analyze how to add new pattern files to this list.

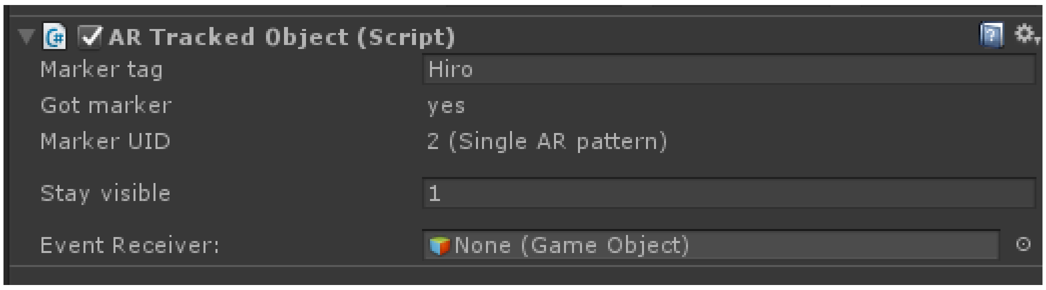


ARMarker (Script)

Marker tag	Hiro
UID	2
Type	Square
Description	Single AR pattern
Pattern file	patt.hiro
Width	0.02
Cont. pose estimation	✓

Figure 5.13. AR Marker's Inspector window.

After the AR Marker script is set we need to add a new empty object below the Scene Root. This object will hold the AR Tracked Object script and will serve as a root to anything tied to this marker. The only parameter we need to set is give it the same Tag as our AR Marker script and ARToolkit will automatically link them. We can then position virtual objects on this marker by adding them as children to the tracked object.


AR Tracked Object (Script)

Marker tag	Hiro
Got marker	yes
Marker UID	2 (Single AR pattern)
Stay visible	1
Event Receiver:	None (Game Object)

Figure 5.14. AR Tracked Object Inspector window.

5.3.3. Generating Pattern Files for ARToolkit

If at any point we want to use a custom marker it is necessary to extract pattern data files from it and import them to our project. In our project we used the standard markers of ARToolkit, but we also tested how to add new custom markers. Generating the pattern files is done using executables in the ARToolkit standalone library from GitHub

The first type of marker we can easily generate pattern files for is Square markers. Square markers, similar to our Hiro marker, are black and-white patterns with a thick black outline. The pattern can be any shape we want and the basic marker size is 8x8 cm, 4x4 cm of which are the central marker and the rest a pure black outline.

The first thing to do to generate a pattern file for a Square marker is design it and print it. Then in the standalone ARToolkit files in the bin folder execute the mk_patt.exe file. This will open a live feed through the camera in a new window. If a marker is detected, it will be highlighted and the pattern inside the marker will be shown on a corner of the window. When the pattern is detected as clearly as desired and with the correct rotation pressing enter will freeze the camera feed. To confirm generating the pattern files enter a file name and press enter. This will make a generic file with no suffix in the same directory mk_patt.exe is in. Take this pattern file and open Holo-Board's project's Assets folder and go into ARToolkit5-Unity/Resources/ardata/markers and paste it in there. The new marker should now be visible when we select an AR Marker script on the ARToolkit object.

The second type of marker we can use is Natural Feature Tracking markers, or NFT markers for short. NFT markers can be any Jpg image of any size and they can also be colored.

To generate an NFT marker pattern file we must go to the standalone ARToolkit's files in the bin folder and paste our marker file in there. Then, open a command line window and change directory to the same file, and then execute the genTexData file adding our image name as a parameter to the execution. It will ask us about adjusting some parameters, but if we do not need something specific we can keep to the default values. This will generate 3 new files with suffixes .iset, .fset and .fset3. Take these pattern files and copy them to Holo-Board's asset's StreamingAssets folder. To use an NFT marker, add an AR Marker script to ARToolkit, select NFT as the Marker Type and type the name of the new marker on the NFT dataset name.

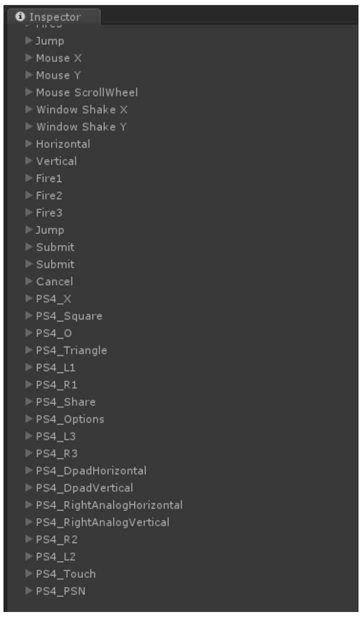
5.4. Holo-Board's Provided Tools

Outside of ARToolkit, Holo-Board also provides us with some new tools of its own. All of these tools were developed from scratch for Holo-Board but they can be reused, removed or replaced as necessary. Below, we will analyze them one by one.

5.4.1. Using a Dualshock 4 Controller

As a primary means of input other than Machine Vision, Holo-board also supports using a PS4 Dualshock 4 controller. The controller comes with Bluetooth wireless connection, and connecting DS4 controllers to any Android phones is already supported. By pressing the PS and Share buttons on the controller, it opens up the Bluetooth receiver which can then be picked up by the phone and pair to it. We have simply mapped they correct buttons in Unity's Player Inputs list, so they can be used by any application as generic Unity inputs.

As we did not find a full input mapping on the internet anywhere and instead it was done through trial and error, a text file with all the mapping done is provided in the Assets/Prefabs folder.



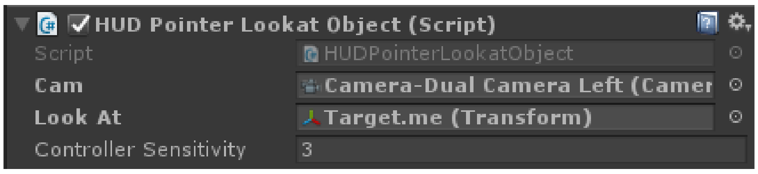
```
Inspector
▶ Jump
▶ Mouse X
▶ Mouse Y
▶ Mouse ScrollWheel
▶ Window Shake X
▶ Window Shake Y
▶ Horizontal
▶ Vertical
▶ Fire1
▶ Fire2
▶ Fire3
▶ Jump
▶ Submit
▶ Submit
▶ Cancel
▶ PS4_X
▶ PS4_Square
▶ PS4_O
▶ PS4_Triangle
▶ PS4_L1
▶ PS4_R1
▶ PS4_Share
▶ PS4_Options
▶ PS4_L3
▶ PS4_R3
▶ PS4_DpadHorizontal
▶ PS4_DpadVertical
▶ PS4_RightAnalogHorizontal
▶ PS4_RightAnalogVertical
▶ PS4_R2
▶ PS4_L2
▶ PS4_Touch
▶ PS4_PSN
```

Figure 5.15. The DS4 inputs in the Player Inputs list.

5.4.2. Machine Vision Based Cursor

Outside of the DS4 controller, Holo-Board also provides us with a Machine Vision based cursor and buttons. The cursor is provided as a prefab in the Assets/Prefabs folder. The cursor follows a 3D object from the 2D point of view of a camera and moves along the screen overlay's 2D canvas. If the 3D object is disabled, for example if it is out of sight, the cursor can be moved using the DS4;s left analog stick or the arrows on the keyboard. If the cursor stays still for over a few seconds, it is hidden from view until it is moved again.

To add a new cursor on the screen drag and drop the prefab GUICursor to the scene as a child to a screen canvas. Then take a look at the Inspector window and find the HUD Pointer Lookat Object script. The Cam object is the camera on top of which the cursor will be visible, the Look At object is the 3D object towards which the cursor will point and the controller sensitivity is a multiplier to the speed of the cursor if we move it using the DS4 controller or the keyboard arrow keys.



HUD Pointer Lookat Object (Script)	
Script	HUDPointerLookatObject
Cam	Camera-Dual Camera Left (Camer
Look At	Target.me (Transform)
Controller Sensitivity	3

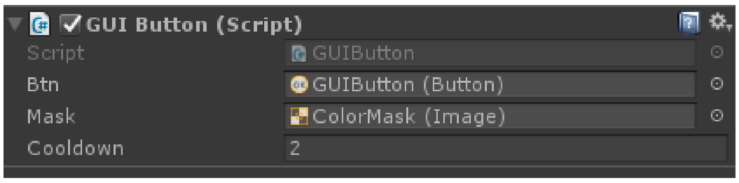
Figure 5.16. GUICursor's controller script.

The default cursor used in Holo-Board is on the Left Eye GUI canvas, the Camera is the Left Eye Camera and the Look At object is a target empty object which is a child of our Hiro Marker. For the Right eye, the Camera Handler automatically swaps the Cam object with the Right Eye Camera.

5.4.3. Machine Vision Based Buttons

Since our Machine Vision cursor is custom made, interactions with generic Unity buttons must be custom made as well. The MV cursor only has a position in space with no way of clicking a button. The MV button uses colliders to detect when a cursor is on top of it and, inspired by Kinect, simulates a click when the cursor stays on top of it for a few seconds.

To add a new button to the scene just add a new GUIButton object using its prefab in the scene. On the GUI Button script adjusting the Cooldown value changes how fast a click is simulated. All other functionality is set automatically.



GUI Button (Script)	
Script	GUIButton
Btn	GUIButton (Button)
Mask	ColorMask (Image)
Cooldown	2

Figure 5.17. The GUI Button script.

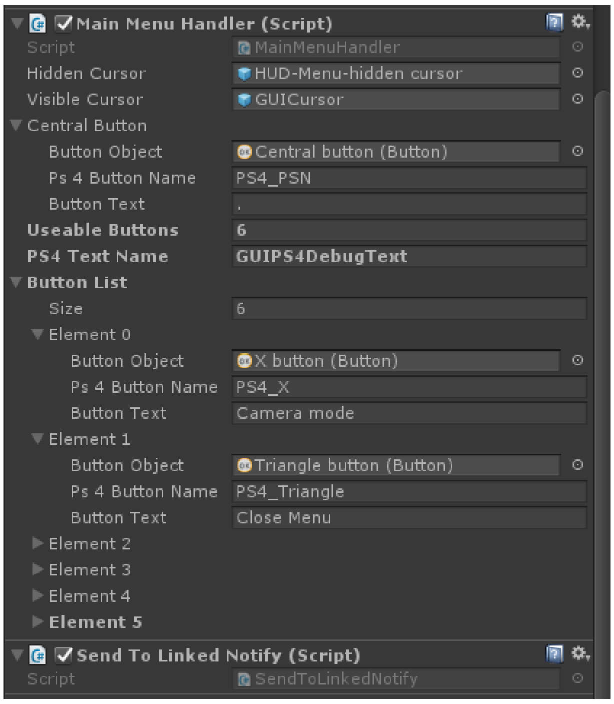
5.4.4. Adjusting the Main Menu

The Main Menu is a collection of the Main Menu Handler and a collection of cursors and buttons. All of its components can easily be reprogrammed using the Inspector window. The prefab of the Main Menu is the Main Menu mode file. On that object we have two scripts: Send to linked notify, which will be explained later, and the Main Menu Handler which is used to adjust the Main Menu settings.

When looking at the Main Menu Handler script on the inspector we first notice two cursor objects. The Visible cursor is the MV Cursor explained above while the Hidden cursor is an invisible cursor moved using DS4 buttons to simulate clicks in a similar behavior to the MV cursor. We also see a PS4 Text Name field which is used only when pressing the PS4 debug text button to find the PS4 debug text by name.

Below the cursors we see the Central button field with 3 subfields that hold relevant information to that button. The Central button is used to enable the main menu and is always visible in the middle of the user's viewpoint. We also have an array with the other six buttons present on the Main Menu and a field named Usable Buttons that tells us how many of these buttons we are using in our application. By reducing the value in the Usable buttons field we can show only as many buttons as we need for any app without altering the Main Menu object. Each button holds 3 fields of data: a reference to the button's object, a button text which alters the button's text at runtime and a PS4 button name that tells us using which PS4 input we can click that button using a controller.

To change what script is executed when pressing each button we can set its OnClick() function in the inspector like any traditional button.

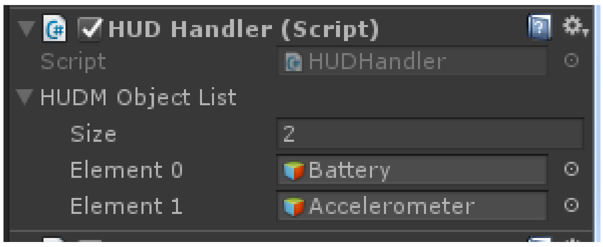


```
▼ Main Menu Handler (Script)
  Script                 MainMenuHandler
  Hidden Cursor          HUD-Menu-hidden cursor
  Visible Cursor         GUICursor
▼ Central Button
    Button Object        Central button (Button)
    Ps 4 Button Name     PS4_PSN
    Button Text          .
  Useable Buttons        6
  PS4 Text Name          GUIPS4DebugText
▼ Button List
    Size                 6
  ▼ Element 0
      Button Object      X button (Button)
      Ps 4 Button Name   PS4_X
      Button Text        Camera mode
  ▼ Element 1
      Button Object      Triangle button (Button)
      Ps 4 Button Name   PS4_Triangle
      Button Text        Close Menu
  ► Element 2
  ► Element 3
  ► Element 4
  ► Element 5
▼ Send To Linked Notify (Script)
  Script                 SendToLinkedNotify
```

Figure 5.18.The Main Menu Handler script's inspector window.

5.4.5. Using the HUD Handler

The HUD Handler script resides on the HUD mode object. This object serves as a parent to any GUI object not relevant to the Main Menu or the FULL App. On the Inspector window of the script we have a HUDM Object list that holds a list of GUI objects. When adding an object to that list it is enabled and disabled through the HUD Handler script. If we want more control over our objects we should not add it to this list, but it still should be a child of the HUD mode object.



HUD Handler (Script)

Script	HUDHandler
HUDM Object List	
Size	2
Element 0	Battery
Element 1	Accelerometer

Figure 5.19. The HUD Handler script's Inspector window.

5.4.6. FULL Mode Functionality

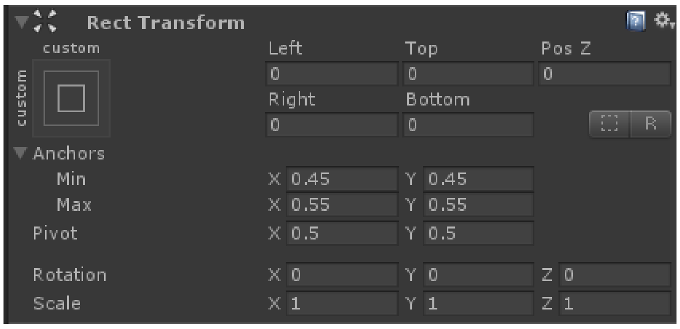
The FULL mode works in a similar manner to the HUD mode. The FULL mode holds a script that enables/disables all of its children when executed, but when the FULL mode is enabled it disables all HUD and Main Menu objects. Unlike the HUD mode, the FULL mode does not hold a list of objects to manage, instead all of its children are part of the FULL mode.

5.4.7. Layout Canvas Objects Correctly in the Scene

In Holo-Board, we have two canvases, one above each eye, but in order to keep them synced we use the Camera Handler script. This script allows us to design only the Left Eye canvas and on runtime the Right Eye canvas is created by duplicating its objects. While this automates a lot of work there are certain rules that must not be violated in order to duplicate the canvases correctly.

First of all, all canvas object must be children of the Left Eye GUI object. Only objects below the Left Eye GUI will be duplicated to the Right Eye GUI.

Secondly, the objects should be positioned using Anchors and zeroing out all pixel offset values. Pixel offsets pose problems not only in our application but in every application with adjustable resolutions, such as an Android application that runs on multiple smartphones with different resolutions. Anchors are percentage- based so they will occupy the same portion of the screen no matter what and will be at a specific position of any canvas they are set on.



Rect Transform

custom

	Left 0	Top 0	Pos Z 0
	Right 0	Bottom 0	
Anchors			
Min	X 0.45	Y 0.45	
Max	X 0.55	Y 0.55	
Pivot	X 0.5	Y 0.5	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

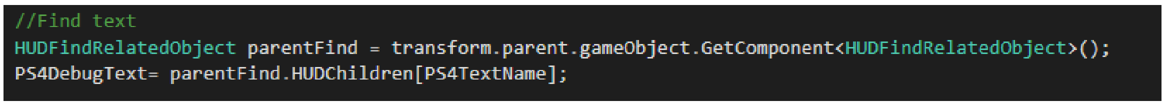
Figure 5.20. Position a GUI object correctly using anchors and zeroing out pixel offsets.

Finally, in order to duplicate references correctly, objects should not reference each other directly. For example, if object "A" on the Left Eye GUI points to another object "B" on the same eye, the duplicate “A” on the Right Eye will not point to duplicate “B” automatically.

How we solve this problem will be explained later, but in order for our solution to work, any objects on the canvas should follow the pre-existing basic hierarchy. Below the Left Eye GUI should be the Main Menu prefab, Notification text, HUD Handler and Full App Handler. Any other object should be a child of either the FULL Handler only if it is relevant to the FULL App otherwise it should be a child of the HUD handler, whether the HUD Handler monitors it or not.

5.4.8. Reference Other Objects

As mentioned above, objects should not reference each other directly, or else the two canvases will be de-synced when duplicating. Instead, both the Left Eye GUI and Right Eye GUI host the HUD Find Related Object script which has references to the Main Menu-Han, HUD-Han, FULL-Han objects as well as dictionaries containing the children of the Main Menu-Han and HUD-Han, searchable by name.



```
//Find text
HUDFindRelatedObject parentFind = transform.parent.gameObject.GetComponent<HUDFindRelatedObject>();
PS4DebugText= parentFind.HUDChildren[PS4TextName];
```

Figure 5.21. Using the HUD Find Related Object to find a HUD object from the Main Menu Handler through its parent, the Left/Right Eye GUI accordingly.

5.4.9. Using the Notification Text

The Notification text is a very specific GUI object. It can be used to show a message to the user. The text remains visible for a few seconds then disappears automatically. This is useful when we want to notify the user about anything, for example when the user uses the touch screen we show a warning message, because pressing buttons using touch will de-sync the two eye canvases.

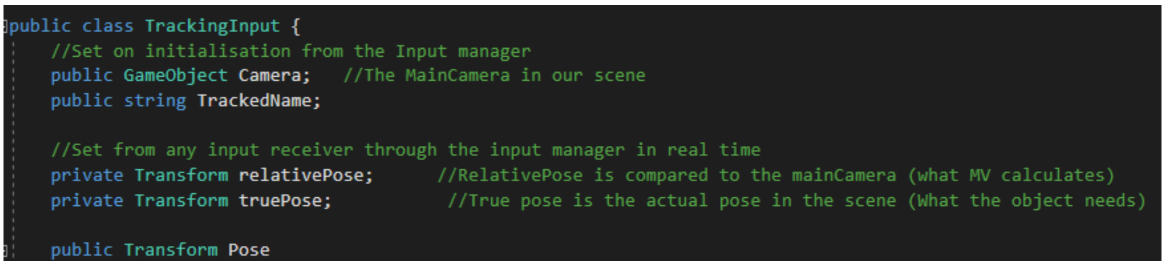
To send a message to the Notify text, the Main Menu, HUD and FULL Handlers have the Send to Linked Notify script attached to them. Simply call one of the Show Message, Show Hint or Show Warning methods present in there.

5.4.10. Using the Input Handler

In order to receive non-generic inputs Holo-Board uses its own Input Handler script. This Handler can receive inputs in one of two forms either a Tracking input or an Event input. A Tracking input is the result of tracking something on the scene and calculating its position, similar to ARToolkit’s marker tracking, while an Event input is checking if something is happening or not, for example doing a gesture.

The Input Handler consists of two dictionaries, one for each type of input. Each different input is an instance of a data holder class with specific data inside it.

Tracking inputs contain information relative to the position and pose of an object. Thus, the main information kept in there is a Transform object. Since we assume this information is relative to the user's viewpoint, we specify this as the relative pose and calculate the true pose in the virtual scene by triangulating this information with the position of our camera on the scene. This true pose will be the value we would give to and object to place it on the correct position in the virtual scene.



```
public class TrackingInput {
    //Set on initialisation from the Input manager
    public GameObject Camera;   //The MainCamera in our scene
    public string TrackedName;

    //Set from any input receiver through the input manager in real time
    private Transform relativePose;      //RelativePose is compared to the mainCamera (what MV calculates)
    private Transform truePose;           //True pose is the actual pose in the scene (What the object needs)

    public Transform Pose
```

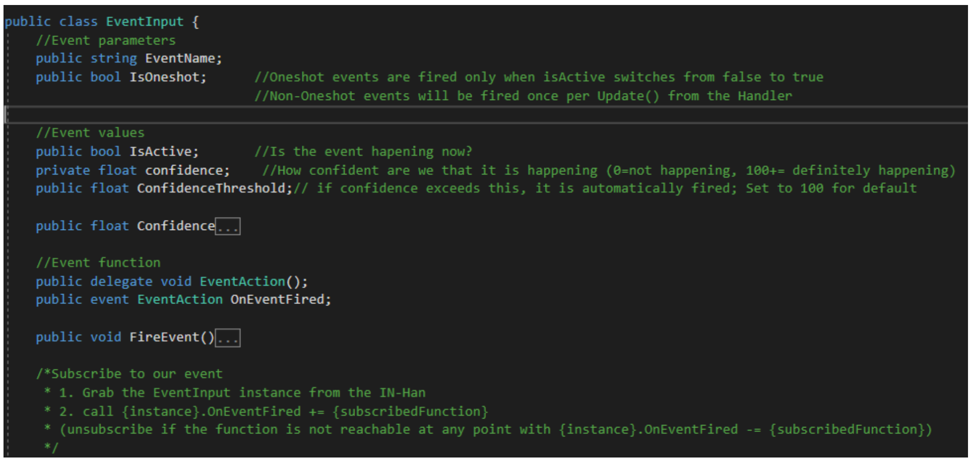
Figure 5.22. The Tracking Input data holder class.

To add a new tracking input or to update its value we can call the Set Tracking Input method on the Input Handler using the Input name as a parameter to specify which input we will add/update. Reading a Tracking input is done in a similar manner using the Get Tracing Input with its name as a parameter.

Event Inputs are as the name suggests simple events. When an application wants to know if something happens, it subscribes to an event, and when that something happens the event is fired.

In our case, we also extend the above functionality. The event input class has the traditional OnEventFired method which is a list of subscribed functions called when an event happens. Extending that functionality, we can specify if an event is continuous, like a grabbing motion, or one-shot, like a pinch. Continuous motions will be executed every time Update() is called on the Input Handler as long as the event is happening, while one-shot events will be fired only when the is Active value switches to true or the confidence threshold is exceeded.

In addition, since most Machine vision algorithms don't simply give us a Boolean if a gesture is happening but instead give a confidence percentage of how likely it is a motion is detected at any point, we can hold that information in the Confidence field. We can also set a Confidence Threshold value above which the event is automatically fired.



```
public class EventInput {
    //Event parameters
    public string EventName;
    public bool IsOneshot;      //Oneshot events are fired only when isActive switches from false to true
                                //Non-Oneshot events will be fired once per Update() from the Handler

    //Event values
    public bool IsActive;       //Is the event hapening now?
    private float confidence;    //How confident are we that it is happening (0=not happening, 100+= definitely happening)
    public float ConfidenceThreshold;// if confidence exceeds this, it is automatically fired; Set to 100 for default

    public float Confidence...

    //Event function
    public delegate void EventAction();
    public event EventAction OnEventFired;

    public void FireEvent()...

    /*Subscribe to our event
     * 1. Grab the EventInput instance from the IN-Han
     * 2. call {instance}.OnEventFired += {subscribedFunction}
     * (unsubscribe if the function is not reachable at any point with {instance}.OnEventFired -= {subscribedFunction})
     */
```

Figure 5.23. The Event Input data holder class.

5.4.11. Build and Run Correctly

Because our application uses ARToolkit and builds for android there are some details to set when building the project. When building for Android we must set the Package name by going to the Project Settings-> Player in Unity. The project name should have the format com.{Company name}.{Application name}. The same name should be changed in the Assets->Plugins->Android->AndroidManifest.xml file, used by ARToolkit.

6. Implementation

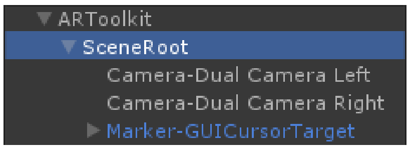
In the previous section we detailed how to use everything Holo-Board has to offer. In this section we will analyze why we made everything the way it is and how we developed each tool present in Holo-Board.

6.1. Integrating ARToolkit

Initially we wanted to find an appropriate SDK to base our application upon. When our project started, ARCore and ARKit were not as popular so our choice was between Wikitude, Vuforia and ARToolkit. All three SDKs have a plethora of tools to assist us and all three can be integrated to Unity.

Out of the three we selected ARToolkit for a few reasons. The main reason is that ARToolkit is Open Source, which is ideal since Holo-Board is a reprogrammable tool. The second reason ARToolkit was chosen is that we wanted an SDK that helped us design a basic Machine Vision interaction system and Marker-based tracking was enough to achieve that. In addition, our system will be using a distributed architecture so we will use ARToolkit but we will not tailor our application around it so it will be interchangeable at any point.

For starters, we downloaded the ARToolkit library as well as the Unity package from Github. Following the example of the sample projects we added the first objects to our scene.



```
▼ ARToolkit
  ▼ SceneRoot
      Camera-Dual Camera Left
      Camera-Dual Camera Right
    ▶ Marker-GUICursorTarget
```

Figure 6.1. The basic layout of ARToolkit.

First of all, we added an empty object named ARToolkit with the AR Controller script attached to it. By adding the package, the script was already set with correct parameters so we did not change anything.

Next we added another empty object called SceneRoot as a child to the ARToolkit object and added the AR Origin script to it. We made sure both our objects were initially positioned at the point (0, 0, 0) in the scene so our real scene root and Unity would have the same central point.

Finally, we added cameras to the scene removing the default camera provided by Unity. Since we have a Cardboard application we needed two cameras, one for each eye covering half the screen. By adding the AR Camera script to each camera and checking the box "Part of a stereo camera" and setting one as the left and the other as the right cameras, ARToolkit automatically set the viewpoint Rectangular fields correctly. The only thing we changed was these cameras would have a culling mask of UI and AR Foreground objects only. At runtime ARToolkit creates two additional cameras named Video Background that view only the AR Background layer from the camera feed and show it behind our virtual cameras.

After setting up the basic scene, we changed the build settings to Android and changed the parameters by following Unity's manual, as we explained in the previous section of the thesis, and our initial, empty application was buildable.

6.2. Tracking a Marker

As our next step, we wanted to track a marker using ARToolkit. By default, ARToolkit supports 4 types of markers: Square, Square barcodes, Multimarkers and NFT. Multimarkers were not useful for our application, while Square barcodes have fixed shape. NFTs would be ideal for our application, since they can be any shape or color we want but Square markers are simpler and also good enough for our basic application. A Square marker can be swapped with an NFT marker at a later date when a developer has a more specific application in mind and wants the marker to blend in some environment. Thus we chose to use Square markers.

ARToolkit itself provides us with two base square markers the Hiro and Kanji markers. In ARToolkit's files there is also a template empty marker on which we can add any shape we want as well as two more shapes simply named "one" and "two". Since we wanted to test how to add custom shapes to ARToolkit we decided to use the "two" marker.

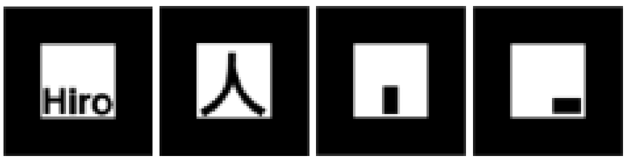


Figure 6.2. The four basic square markers in ARToolkit "Hiro", 'Kanji", "One", "Two" in order.

In order to generate and integrate the pattern files into ARToolkit we printed all four markers in 8x8cm size and followed the instructions mentioned in the previous section. We then added an AR Marker script to our ARToolkit object. By selecting Square in the marker type and the "patt.two" as the pattern file ARToolkit set a UID for our marker. We then added another empty game object as a child to the SceneRoot object that held the AR Tracked Object script. We gave both scripts the same marker tag and the "Got marker" field changed to "yes" indicating that the tag is correct. By adding a cube as a child to that marker and placing it on top of it we can then see that cube on top of the real world marker.

Later on in the development process we noticed that the "two" marker is too simple and would be falsely detected on random shadows in the environment frequently, even when the actual marker was in sight. Thus, the "two' marker was swapped with the Hiro marker.

6.3. Designing a Main Menu

Now that we can see virtual objects on the scene, the first thing we wanted to make was a Main Menu. The initial design was a menu visible on top of a marker somewhere in the world, for example on a wall or on a bracelet on the user's wrist. The user would then interact with any visible menu using gestures, or a controller.

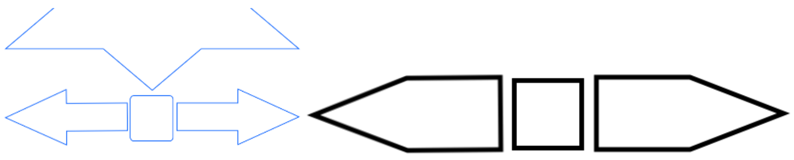


Figure 6.3. Initial Menu UI designs.

Eventually this menu design was scrapped because it was too impractical. Because the marker was supposed to be in the central square of the menu, doing a gesture over it would hide the marker, and thus close the menu itself. Also, by showing the menu on one hand like a wristwatch while holding a controller and then pressing buttons on the controller was uncomfortable.

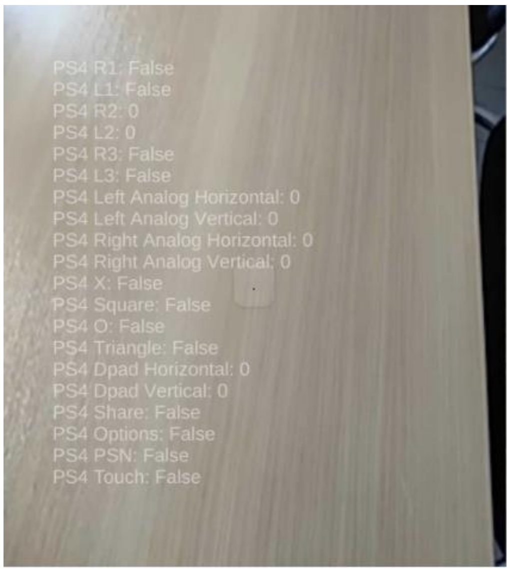
At this point we also noticed a key flaw of ARToolkit. If we want to track two markers in one scene, these markers are detected as a continuous scene, not two distinct collections of objects. If for example we want to show a menu over one marker and a pointer above a second, moving the pointer around would either move the whole menu with the pointer or show the menu correctly while freezing the pointer in a fixed spot.

6.4. Using a Dualshock 4 Controller

Before scrapping our first menu though, we attempted to design some ways to interact with it. One of our attempts was testing out Bluetooth controllers for Android smartphones. Initially we tried using a one-handed Bluetooth controller for Android, but after testing a couple of cheap controllers we found out they all have different and very random keymappings.

After finding out Dualshock 4 controllers support connecting to all Andorid phones via Bluetooth and testing one out, we selected that as our main controller. DS4 controllers are straightforward to use, even when navigating the basic menus of an android phone, moving the left analog stick highlights an object on screen and then moves accordingly, the square button was a click on the selected button and the X, Circle and Triangle buttons serve as the 3 basic Android buttons (back, home and active apps). Furthermore, it is a controller with 16 buttons and 2 analog sticks giving us plenty of keymapping options.

Following the instructions in a video tutorial we designed the DS4 debug text field and the keymapping for PC execution for the DS4 controller.



```
PS4 R1: False
PS4 L1: False
PS4 R2: 0
PS4 L2: 0
PS4 R3: False
PS4 L3: False
PS4 Left Analog Horizontal: 0
PS4 Left Analog Vertical: 0
PS4 Right Analog Horizontal: 0
PS4 Right Analog Vertical: 0
PS4 X: False
PS4 Square: False
PS4 O: False
PS4 Triangle: False
PS4 Dpad Horizontal: 0
PS4 Dpad Vertical: 0
PS4 Share: False
PS4 Options: False
PS4 PSN: False
PS4 Touch: False
```

Figure 6.4. The DS4 Debug Text.

When we executed the application on Android however, we noticed the keymapping was different. Since the correct keymapping for Android was nowhere to be found online, using trial and error we changed the keymapping to be correct for Android smartphones. We included the results of our research in a text file along with the project's prefabs. We also confirmed these keybindings were independent of Android version by testing on two smartphones running on different versions of Android.

We also noticed that ARToolit's debug menu, which was enabled with the Enter key on PC execution was mapped to the "joystick button 10" which, lucky for us was mapped on the DS4 controller as the R3 button. Through this we can set the threshold value for marker detection manually on any smartphone at runtime and change the camera resolution through a menu among some other tools, mostly useful for debugging.

6.5. Re-designing the Main Menu

As our initial design of the main menu was flawed we need a new one. Since our main problem was covering the marker when doing gestures in front of it, we want for the reverse approach. Instead of setting the menu on top of a marker in 3D space, we designed the menu as a 2D overlay to the camera and we would then use the marker as a position tracker to detect gestures.

Because doing gestures in midair is not precise, we wanted the buttons to be far apart from each other so users can easily press the correct buttons. By following the previous layout's example, we added a central button with 2 additional ones on opposite sides of the screen. The central button is used to enable the other two buttons which in turn could be used for anything. Because this time there was no marker in the middle of the menu, we also had the freedom of adding two more buttons vertically and make a cross- shaped menu.

Since the cross-shape resembled the layout of the four face buttons on the DS4 controller (X, Square, Circle and Triangle) we also made a simple script that mapped the DS4 buttons to the appropriate menu buttons. We also changed the color of all the buttons to be slightly transparent so it would not totally cover the user's field of view and also colored the four buttons according to their DS4 controller's counterparts so one could instinctively tell how they were connected. The central button was also mapped to the PS button which is in the middle of the DS4 controller and works as a menu button even in the Playstation 4 system.

After we added the cursor we also noticed that we could add four more buttons to the edges of the screen and they would still be distinct enough to press, but adding all four additional buttons to the screen covered too much of the actual field of view. Instead we opted to add two more buttons in the top-left and top-right corners. These two buttons were also bound to the L1 and R1 bumpers on the controller and were made visually thinner and wider than the more squared cross buttons. These buttons were also colored gray.

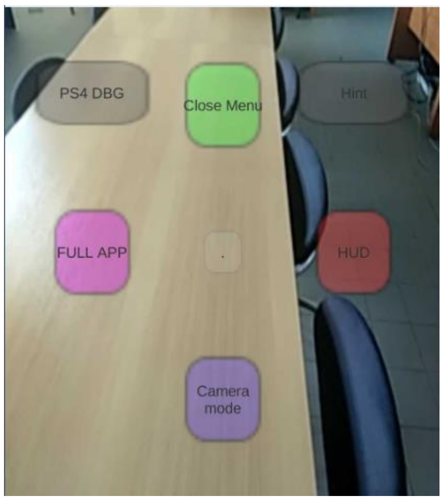


Figure 6.5. The final main Menu.

Keeping in mind our menu should be reusable in future applications, we also added some parameters to the Main Menu that can set how many buttons are usable, their names and the key bindings to the DS4 controller. All of these can be set from the Inspector at any point without altering any scripts or deleting any objects.

For starters, we kept the two horizontal central buttons empty for the two modes of executions we had planned in the initial version of the game, HUD and FULL-App modes. We also added the already made DS4 debug text to the L1 button for future use. Finally, we used one of the buttons as a close menu button. In the beginning we used the central button both for enabling and disabling the menu but this was changed for optimization after testing.

6.6. Machine Vision Cursor

Up until now, the only thing resembling Augmented Reality in our project is that we see the camera behind a simple menu. At this point we wanted a more immersive way to interact with our buttons than a controller.

Since our main menu is on top of the UI, when executing in Debug mode, the Windows cursor interacts with the buttons and clicking the mouse clicks a button. Connecting a Bluetooth mouse to our smartphone also enabled the same cursor in the app even when running on a phone. We can use ARToolkit's marker position as a reference point for the marker and Unity's "Camera.worldToScreenPoint()" function to convert a 3D point on screen to a 2D point in the UI. Using this we can, theoretically, move the cursor wherever we want.

Unfortunately, enabling and moving the cursor without appropriate hardware is highly rejected in Unity. Instead we designed our own cursor, using an image with the sprite of a button and moving it on the UI as mentioned before.

Figure 6.6. Our UI cursor sprite.

For a physical marker, initially we used a Hiro marker of the default size of 8x8cm glued to a cardboard to stay flat. This was not ideal when we wanted to move it as a pointer. Instead we reduced the size to 4x4cm and attached it to a wristband that could be worn on four fingers. Initially, we were hesitant about reducing the marker size as we thought it would increase the error rate for detecting the marker. After testing with the wristband we noticed that was not the case.

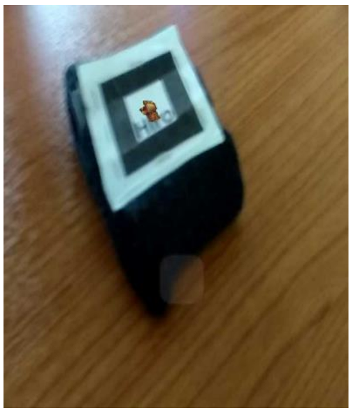


Figure 6.7. Hiro marker 4x4 cm on a wristband.

Even though using the wristband is a good solution for what we hoped for, ideally we would like to track the user's index finger as this will be a more natural motion. Following the success of the wristband, we reduced the size of the marker to 1,5x1,5 cm. While reducing the marker size made our application error prone when there are shadows in the background, this happened when the marker is not visible and not frequently enough to be a problem. This new smaller marker was attached to a ring using a magnet glued to its back.

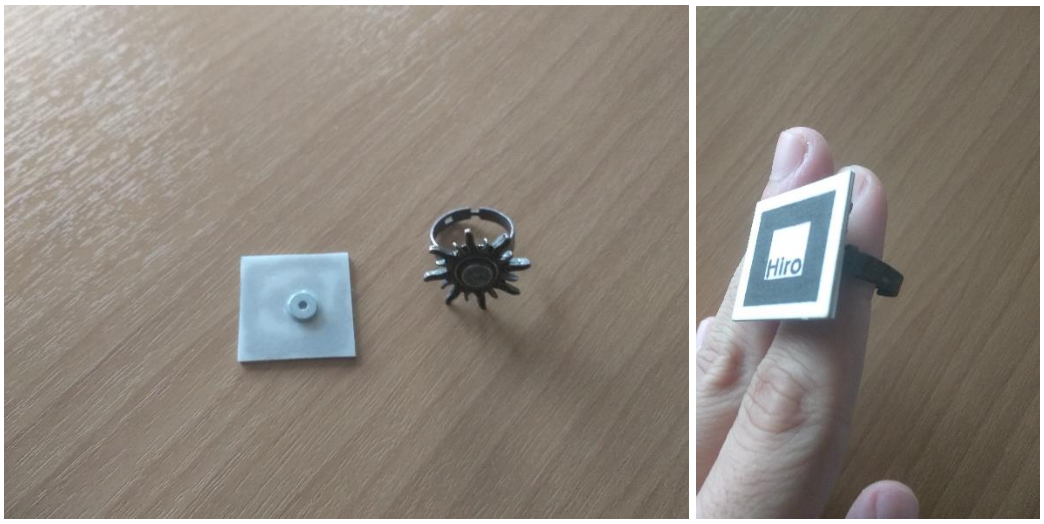


Figure 6.8. Hiro marker 1,5x1,5 cm on a ring.

When ARToolkit detects a marker on the screen it positions the virtual marker on the correct point in the screen. On the other hand when the marker is not visible the virtual marker is positioned in a generic point in space, specifically in the SceneRoot since we didn't move it away from there when we crated it. This behavior posed problems with the cursor flickering every time the marker was momentarily lost. To fix this problem we created an empty game object as a child to the marker object itself and made the pointer target that.

Unlike the base marker object, ARToolkit's marker disables all of its children when it is not visible, so instead of flickering when the marker was lost, the cursor simply stayed in place, and the momentary detection errors became unnoticeable.

Extending the above behavior when the target remains out of sight for over a few seconds we hide the cursor so it won't lurk on the field of view if we don't want to move it. In addition, if the marker is not visible we added functionality to move the cursor using the DS4 controller's left analog stick as an alternative (or the arrow/ WASD buttons on the keyboard).

6.7. Machine Vision Buttons

Since our cursor was custom made, currently it is a simple image moving through the screen. Next, we needed the buttons to notice it and act like it is an actual cursor. For that purpose, we added 2D colliders to both our cursor and buttons. When the cursor enters a button's collider we highlight it and after it exits we un-highlight it.

Next we needed a way to click our button. After testing, we noticed that clicking by checking the depth changes when doing a click motion was unstable. Sometimes buttons were clicked by accident, the clicking motion moved the cursor outside the button while other people curved their fingers to imitate a click, hiding the marker in the process.

As an alternative we imitated Kinect's clicking method. On the Kinect, clicking a button is done by holding our hand over a button for a few seconds. To indicate a click will be made, the button starts filling with color in a circular manner and when the button is filled with color it is clicked. We designed our buttons to work in the exact same manner and the result was satisfactory.

With these new buttons on the Main Menu, we re-designed the clicking method of the DS4 controller. This time, we added a second invisible cursor without visuals that moves on top of any button we press imitating the same clicking method the actual cursor uses.

6.8. Dual Camera Handling

A problem we ignored until now is that our application has two cameras on one screen. When designing the main menu, we used a canvas on top of one of the two cameras, and when we wanted to test in a Cardboard mask we had to duplicate and re-initialize all objects for the second camera.

After doing this more than a few times we searched for alternative solutions. One solution we found used in VR applications is setting the canvas in world space instead of screen space and setting it as a child to a camera. The same canvas would then be visible on top of both our cameras as they move in a similar manner. In our case this approach did not work for two reasons. First, our Machine Vision cursor was not translated correctly on the canvas when it was on 3D space. Instead of tracking the marker and pointing towards it while being on the 2D canvas, it just moved to the 3D point of the marker. Secondly, the more important problem was that ARToolkit automatically changes some camera parameters so our canvas was partially outside the field of vision and static in size no matter how we resized or moved the canvas and its children.

As an alternative solution, we reset our canvas on Screen space and made a custom camera and GUI Handler. This handler took all objects from the Left Eye GUI and duplicated them automatically when we pressed Play. We also added extra functionality to automatically set the target camera of our duplicated MV cursor to the Right Eye so it appeared on the correct half of the screen.

In addition, we added another functionality to the Camera Handler to switch from a two camera perspective to a one camera perspective for debugging purposes. This feature was used a lot in the development so we made one of the menu buttons the "Camera Mode" button that switched between the two modes.

6.9. The HUD Handler

After our menu was complete, the next step is to create an appropriate environment for Sub-applications to be executed upon. We want Holo-Board to not only support complete graphics applications, but also some simpler sub-apps with limited and semi-automated usage.

The first of these smaller Sub-Apps were Heads-Up display Apps. HUD-Apps are simple graphical applications that can show some simple information to the user at any time.

Examples of such apps are a battery indicator or a GPS feed in the corner of the user's field of view. Using the HUD Handler a user can see the world through Holo-Board's camera uninterrupted and whenever he wants to augment his view he can enable the HUD with the push of a button.

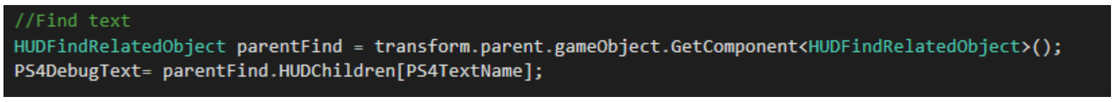
The HUD Handler was created as an empty object, sibling to the Main Menu. Its only functionality was getting a list of objects as input from the Inspector and enable/disable them with the press of a button from the Main Menu. One of the Main Menu's buttons was set to call the HUD Handler's toggle function.

After designing the above functionality, we noticed a key flaw of our duplications. If we referenced a canvas object from another using the Inspector, after duplicating both objects this reference was not duplicated. This is also very card to automate as it is very dependent to the objects themselves. We also noticed that these references were duplicated only if the objects had a parent/child relationship. To solve this, we designed a new system of GUI Communication.

6.10 GUI Object Communication

The basic concept of the GUI communication was simple. Our hierarchy consists of the parent canvas, two key Holo-Board objects (the Main Menu and HUD) and their children. When the application starts and after the duplication is complete, the Left and Right eye GUI objects keep a reference to each of their children, identifying each by the Handler script attached to each, and then makes a dictionary for each of their children list.

If an object wants to reference the Main Menu Handler or HUD Handler they can get the reference directly from their parent canvas, or if they want a child of these two they can make a search by name to the appropriate dictionary. All these references exist in the HUD Find Related Object Script accessed through other scripts.



```
//Find text
HUDFindRelatedObject parentFind = transform.parent.gameObject.GetComponent<HUDFindRelatedObject>();
PS4DebugText= parentFind.HUDChildren[PS4TextName];
```

Figure 6.9. Using the HUD Find Related object From the Main Menu Handler to access a child of the HUD Handler.

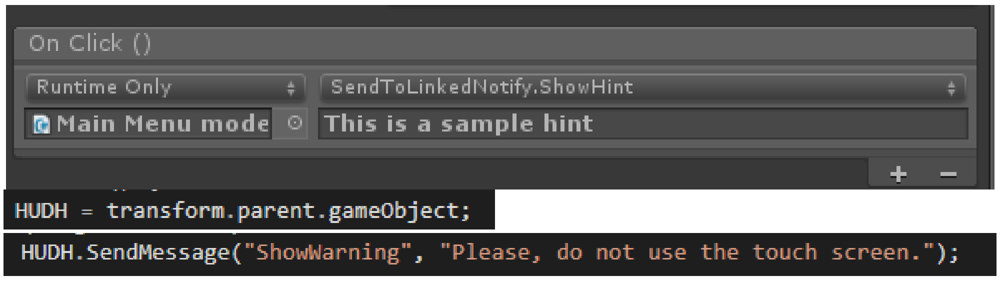
In order for this communication to work, it is necessary for future applications to keep our basic structure of the hierarchy. As children to the Left/Right Eye GUI objects should be only the key Holo-Board objects referenced directly in the HUD Find Related Object script while any other objects should be children to the appropriate Handler.

6.11. Notification Text

For our next step, we wanted to develop something to resemble the system notification popups present in both Computers and Smartphones. Through this we could show a message to the user for a short time whenever we wanted to inform him about something.

In order to achieve this we created the Notification Text object and placed it in the center of the Canvas. The Notification Text would receive a text to print via a function, a color for the text and a duration. It would then show that text in the middle of both eyes' view in the specified color and then hide it after the specified duration.

Accessing the Notification Text is done via a similar manner to the HUD Find Related Object, this time on the Send to Linked Notify script present on the Main Menu and HUD Handlers for simpler access from their children.


On Click ()
Runtime Only
SendToLinkedNotify.ShowHint
Main Menu mode
This is a sample hint

```
HUDH = transform.parent.gameObject;
HUDH.SendMessage("ShowWarning", "Please, do not use the touch screen.");
```

Figure 6.10. Sending a Notification Text request via Main Menu button on the Inspector (Top) or via script from a child to the HUD Handler (Bottom).

After testing, we decided to keep the Text duration static, add a second color outline to the text and pre-specify the colors of the text for three occasions: Normal text, Hints and Warnings. Normal text is Black and white, Hints have a vibrant color of yellow and green while warning a more aggressive red and black.

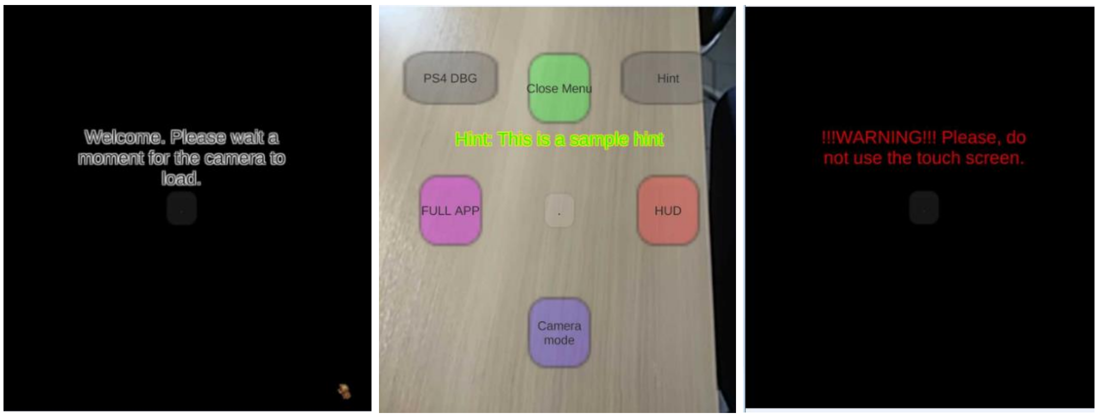

Welcome. Please wait a moment for the camera to load.
PS4 DBG
Close Menu
Hint
Hint: This is a sample hint
FULL APP
HUD
Camera mode
!!!WARNING!!! Please, do not use the touch screen.

Figure 6.11. Simple Notification Text (Left), Hint (Middle), Warning (Right).

6.12. FULL-App Handling

As we mentioned before, we designed the HUD Handler for simpler HUD-App objects, but Holo-Board should also support fully functional, unrestricted Applications.

To achieve that we also made the FULL-Handler. When executing a FULL Application, we disable both the Main Menu and all objects in the HUD, related and unrelated to the HUD Handler. This essentially clears all GUI objects unrelated to the FULL App itself.

The FULL-Handler, similar to the HUD-Handler should be the parent to all on-GUI elements relevant to the FULL-App, but unlike the HUD-Handler, the FULL-App may also be tied to any other object in the scene, both inside and outside the GUI. A FULL-App may control 3D objects in the virtual scene unrelated to the GUI hierarchy or even enable other markers in ARToolkit's hierarchy.

For the above reasons, we added the FULL-Handler to the HUD Find Related Object script as a reference, but the FULL-Handler should exclusively be responsible for any further execution, so we do not provide a dictionary of tis children.

We also added a function to the Main Menu to re-enable it, which we can call from the FULL-App to go back from a FULL-App execution to the basic Holo-Board.

6.13. Non-Generic Input Handler

As the final part of Holo-Board we wanted to develop a full Machine Vision Application that could track a finger and replace ARToolkit's square marker. Sadly, Unity itself is not made with Machine Vision in mind. Alternatively, we wanted to make an OpenCV# application for Machine Vision and execute it parallel to Holo-Board in real time.

This solution proved to be problematic in multiple forms. First, two separate applications are not permitted to access the camera at the same time. Second, gathering the data from Holo-Board then sending it to an external MV-app and reading it back is possible, but it will require the development of a new type of Middleware that connects Unity's C# with OpenCV# as OpenCV# is not supported in Unity yet. Finally, even if such a Middleware existed, developing an optimal Machine Vision algorithm for gesture detection from First-person could be a whole thesis on its own.

Instead, we decided to create an Input receiver for such inputs as if such Middleware existed and streamline the way we would receive Machine Vision Inputs in a Unity-based desired manner. If a future developer decides to design such a Middleware in the future, it will be useful to have a pre-defined structure to his output than doing it blindly.

As for the format we expect such inputs in, we were inspired by the Kinect as well as ARToolkit. In the end, we decided on two distinct types of inputs: Transformation data (position and pose) or Events. For both of these inputs we created a data holder class to keep the data in a specific format and functions that provide some additional control over the data when they will be used later. We also created two dictionaries where programmers can read/write inputs simply by using a reference by name.

Transformation data comes from tracking a specific point in space, similar to our markers, and calculating its relative position to the camera's viewpoint. These can be used to position objects somewhere in the virtual 3D space, similar to how ARToolkit handles its markers. The class holds a single Transform object that records the position, rotation and scale of the object. Because this is a result of Machine Vision, this transform position is relative to the camera, so when reading the data we triangulate the transform of the camera and the transform of the object relative to the camera to find the true transform position of the object in the virtual scene.

Event data tells us if something is happening or not, for example if a gesture is detected. Machine Vision techniques may provide this info in two ways: either a Boolean true or false if the gesture is happening at any given point or a confidence value telling us how likely it is that gesture is happening at any point. Our Event Input data holder supports both forms. If an input gives us a percentage of confidence we can also set a Confidence Threshold value above which the event is considered as happening through Holo-Board. Finally, an event may be a one-shot event, like a click or a continuous event like a waving motion that happens over a long time. In the data holder we can specify if it is continuous and continuous events will be fired every time the Update function is called on the Input Handler as long as that event is happening.

7. Conclusion

7.1. Summary

In our application we have designed a new SDK that programmers can use to design AR applications for Google Cardboard. We noticed how many unsolved problems still exist when developing a Cardboard-based AR applications. Outside of the touch screen, alternative input methods are scarce and unoptimized. Also, non-hardware inputs were openly rejected by Unity up until recently, and even now they have limited support, thus Machine Vision inputs are almost unusable. Also, even modern SDKs focus on handheld applications, as they are more popular than Cardboard apps, ignoring the problems of a Cardboard-based app completely. Finally, there is no organized structure to an AR application, as is the case for example in a website or a windowed application for a PC. Because of that, many parts that can be automated are still left unorganized.

On our part, we managed to solve a few problems presented above. We designed a system with specific architecture for organizing an AR application as well as tools that automate parts of the development process. We mapped a DS4 controller to Unity's input list for the inputs received from an Android smartphone, which was currently not available online. We also designed a Machine Vision based interaction system with a cursor and buttons. We have solved a few issues present in Cardboard app development, like GUI canvas layout and duplicating GUI objects on two cameras.

7.2. Future work

On the other hand, there are also quite a few issues we faced and could not solve in our current time frame. We will outline these parts here in hopes of future improvement from other developers.

7.2.1. Swapping out ARToolkit

In order to have tracking in our application, the camera feed we used was provided by ARToolkit and ARToolkit gives us a stream of screenshots instead of a video stream, which is limited on all phones at 30fps. For mask-based applications in VR it is proven that the optimal refresh rate is 120fps, so this is a serious delay issue that causes dizziness over a long period of use.

Also, the way we use ARToolkit is unconventional so we would prefer having a more dedicated Machine Vision tracker for gesture and object detection in order to optimize Holo-Board further. Ideally we wanted this to be included in the basic Holo-Board, but the focus shifted to more technical issues in the process. Regardless, ARToolkit is used as an assisting library, but no existing systems are built around it so it can be swapped with any alternative later on.

7.2.2. Evaluating Alternatives to our Tools

The other key issue we faced when developing is that we were not based on a predefined architecture. Any tools we developed were either inspired from other applications we found online or came to us during development. Now that we have a general idea of what we want to achieve we would like to test some alternative solutions to the tools we developed.

In addition, we would also like to further test Holo-Board and add even more development tools that we did not find in our design attempt. Our testing was done with only a few people, all of which had previously used a VR or Cardboard mask in the past. Further testing in a wider audience would be beneficial to detect further flaws that require optimization.

7.2.3. OpenCV# to Unity Middleware

Since Holo-Board has a receiver for Machine Vision applications, it would be beneficial if we could link our unity project to a library better tailored around Machine Vision. Currently, Unity does not support non-hardware inputs so using Machine Vision at all is only achievable through external libraries like ARToolkit. Ideally, we would want something easier to customize through Unity.

OpenCV is the most popular library for smartphones when it comes to Computer Vision, and it even has a C# version in OpenCV#. If we had some Middleware that could connect OpenCV# and Unity to send data from an OpenCV# program to our IN-Han it would open new horizons to Holo-Board. With the plethora of already existing OpenCV Machine Vision algorithms, and the simplicity of making new ones, we would have a vast library of Machine Vision tools to add to Holo-Board as necessary.

7.2.4. Custom-made Gesture/Object Detection Machine Vision Application

As we mentioned before, in Holo-Board we use ARToolkit for object detection and tracking. We also mentioned the way we use ARToolkit is a bit unconventional to what ARToolkit expects, and as such it is difficult to improve Holo-Board's detection in its current form. Ideally, ARToolkit will be replaced by a new Machine Vision system providing us inputs in the format our IN-Han can receive. This Machine Vision application can either be an OpenCV# application using the Middleware mentioned above, or even a standalone library linked to Holo-Board directly.

In our case, we wanted to design a system using a Deep Learning Neural Network for Gesture Detection similar to the one proposed in John et al. (2017), but designing such a system would be very time consuming.

7.2.5. Holo-Board End-User Apps

Finally, the ultimate goal of Holo-Board is to be a platform on top of which developers create their own End-User applications. As such, the ultimate improvement for Holo-Board would be for developers to start using it as a development platform.

8.References

[1]. Milgram, Paul, and Fumio Kishino. "A taxonomy of mixed reality visual displays." *IEICE TRANSACTIONS on Information and Systems* 77.12 (1994): 1321-1329.

[2]. Azuma, Ronald T. "A survey of augmented reality." *Presence: Teleoperators & Virtual Environments* 6.4 (1997): 355-385.

[3]. Vlahakis, Vassilios, et al. "Archeoguide: first results of an augmented reality, mobile computing system in cultural heritage sites." *Virtual Reality, Archeology, and Cultural Heritage* 9 (2001).

[4]. Choudary, Omar, et al. "MARCH: mobile augmented reality for cultural heritage." *Proceedings of the 17th ACM international conference on Multimedia*. ACM, 2009.

[5]. Shibata, Yoshitaka, and Katsumi Sasaki. "Tourist information system based on beacon and augumented reality technologies." *Network-Based Information Systems (NBiS), 2016 19th International Conference on*. IEEE, 2016.

[6]. Chen, Jen-Yang, et al. "Kinect augmented reality gear game design." *Applied System Innovation (ICASI), 2017 International Conference on*. IEEE, 2017.

[7]. Di Capua, Michele, et al. "Rapid prototyping of mobile applications for augumented reality interactions." *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 2011.

[8]. Mäkelä, Ville, et al. "" It's Natural to Grab and Pull": Retrieving Content from Large Displays Using Mid-Air Gestures." (2017).

[9]. John, Vijay, et al. "Real-time hand posture and gesture-based touchless automotive user interface using deep learning." *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017.

[10]. Ling, Haibin. "Augmented reality in reality." *IEEE MultiMedia* 24.3 (2017): 10-15.

Bibliography:

[1]. Mike Mc Shaffy, David Graham. Game Coding Complete, Fourth Edition, 2012

Resources:

[1]. ARToolkit GitHub: https://github.com/artoolkit

[2]. ARToolkit X website: http://www.artoolkitx.org/

[3]. ARToolkit alternative documentation: https://www.hitl.washington.edu/artoolkit/

[4]. ARCore website : https://developers.google.com/ar/

[5]. ARKit website: https://developer.apple.com/arkit/

[6]. Microsoft Holo Lens: https://www.microsoft.com/en-us/hololens

[7]. DAQRI AR mask: https://daqri.com/

[8]. Magic Leap: https://www.magicleap.com/

[9]. Vuforia SDK: https://developer.vuforia.com/downloads/sdk

[10]. Wikitude SDK: http://www.wikitude.com/download/?gclid=CI-LwfWhrNQCFSIL0wod4xgB4w
[11]. Pokemon GO: https://www.pokemongo.com/en-us/
[12]. The Ring brought to life in AR and various other AR projects: www.shek.it
[13]. Nerf Laser ops AR apps: https://apps.hasbro.com/
[14]. Unity tutorials: https://unity3d.com/learn/tutorials
[15]. Unity Manual: https://docs.unity3d.com/Manual/index.html
[16]. Android studio: https://developer.android.com/studio/
[17]. OpenCV library: https://opencv.org/