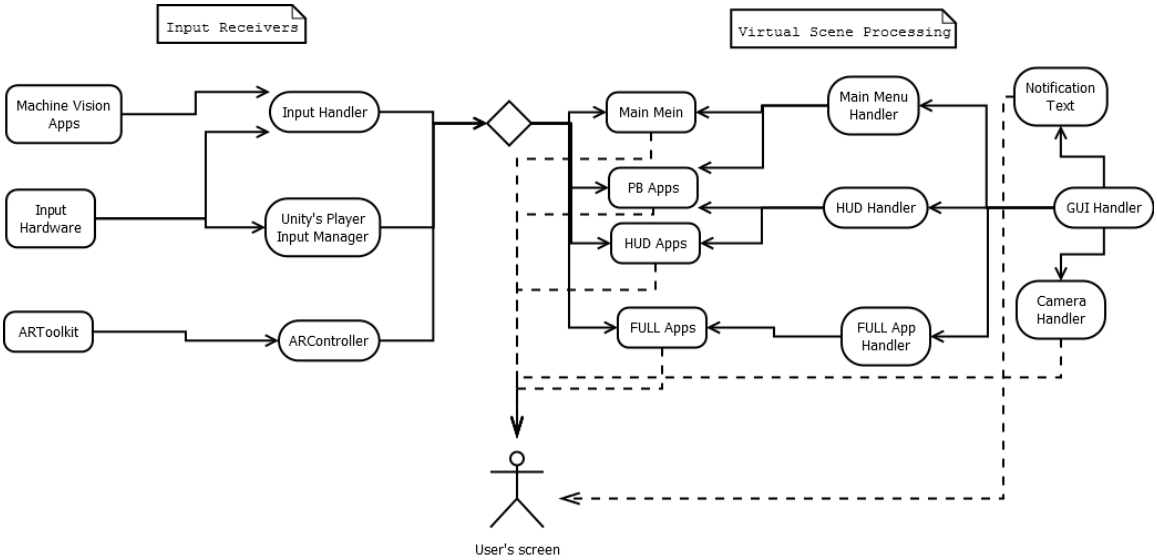
Holo-Board

Document version 1 – 04 Dec 2018 – This is a segment of my Bachelor's thesis that includes the Design document and the user manual for Holo-Board



Introduction

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.

Brief Description

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.

Target Hardware and software

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.

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:

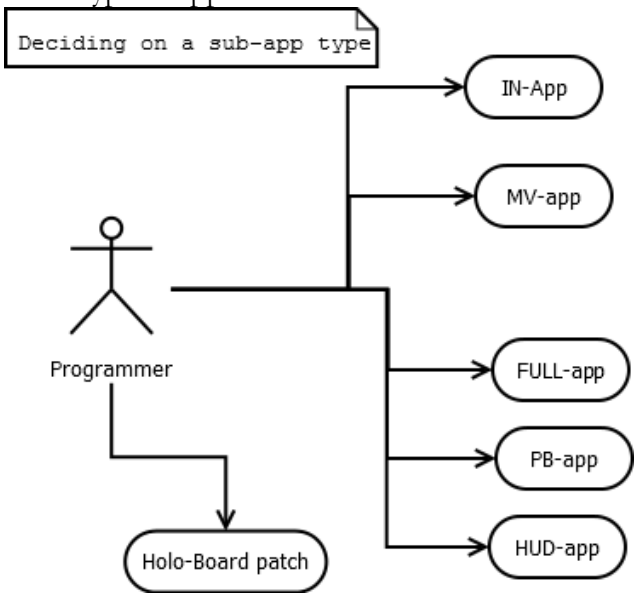


Figure 4.1. Deciding on a sub-app.

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

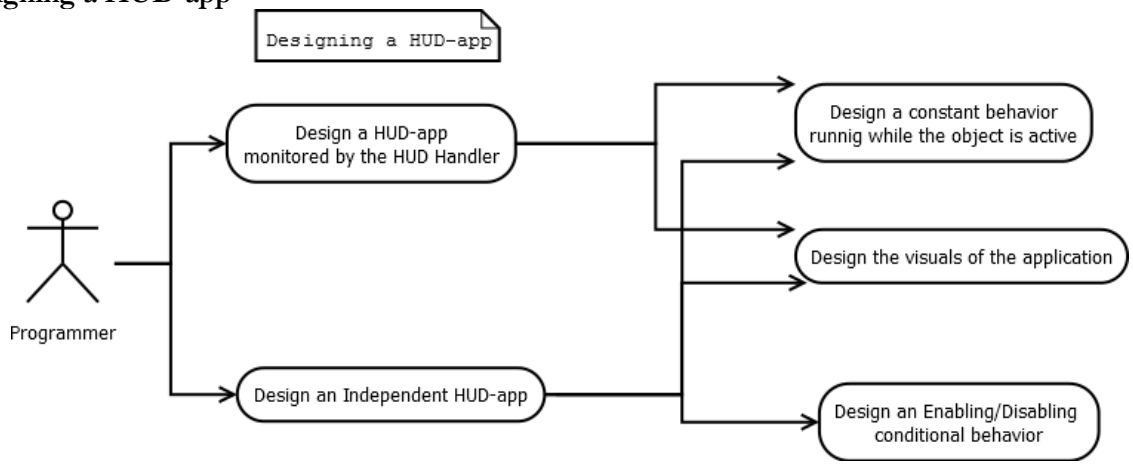


Figure 4.2. Designing a HUD-app.

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

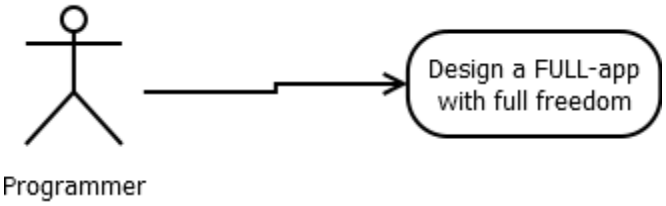


Figure 4.3. Designing a FULL-app.

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

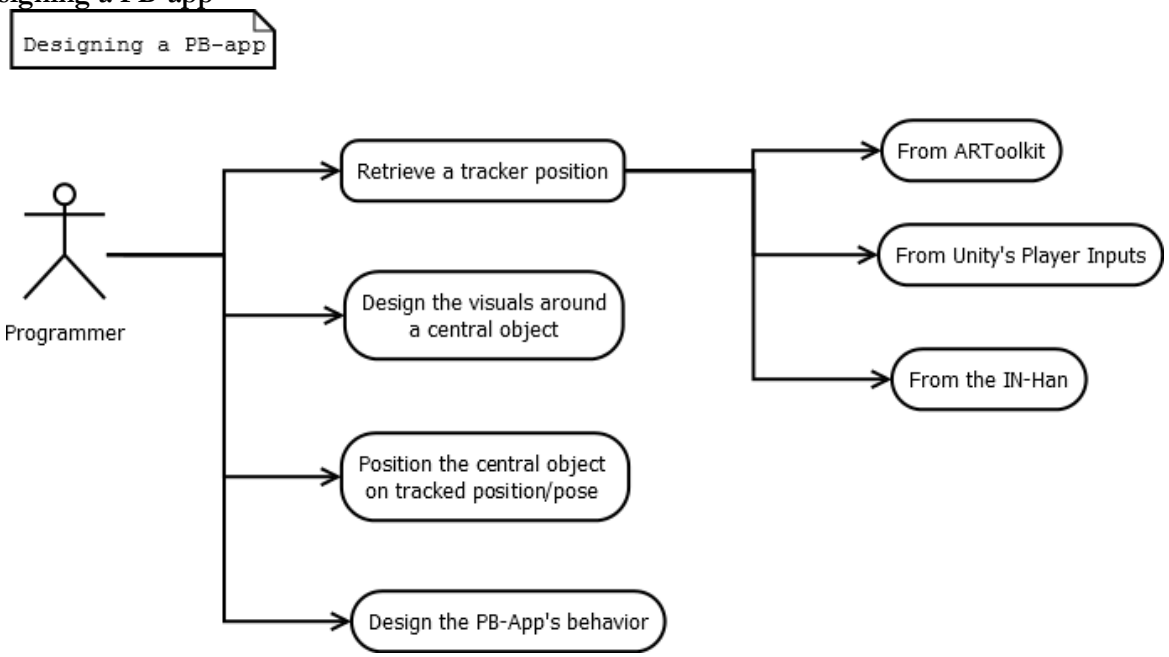


Figure 4.4. Designing a PB-app.

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

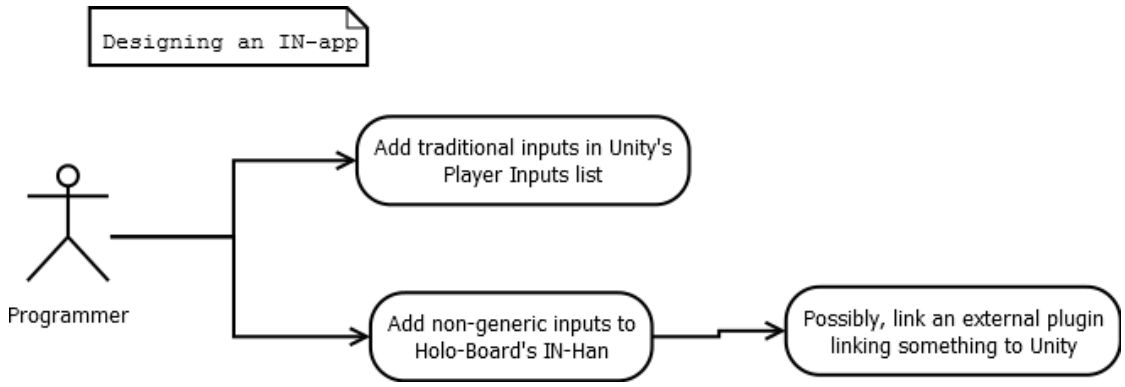


Figure 4.5. Designing an IN-app.

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

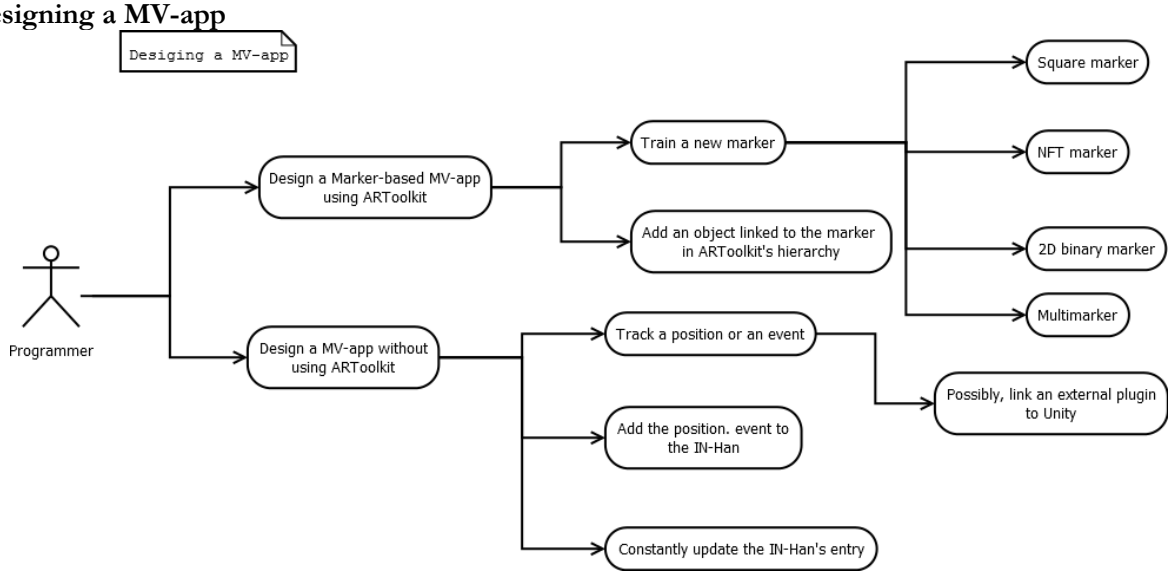


Figure 4.6. Designing a MV-app.

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

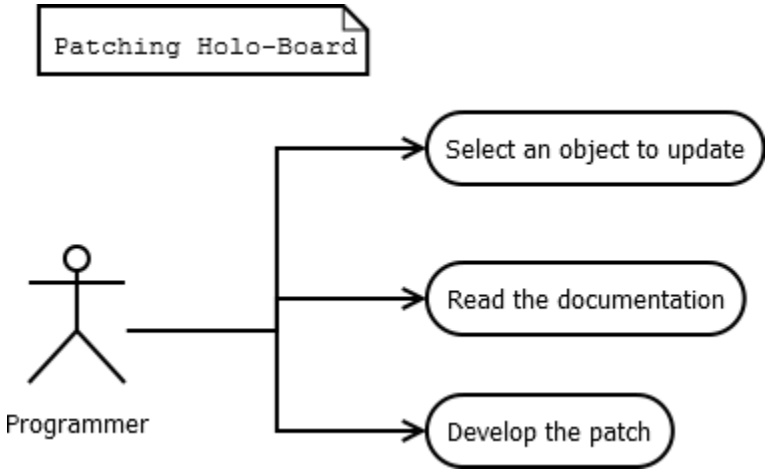


Figure 4.7. Patching Holo-Board.

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

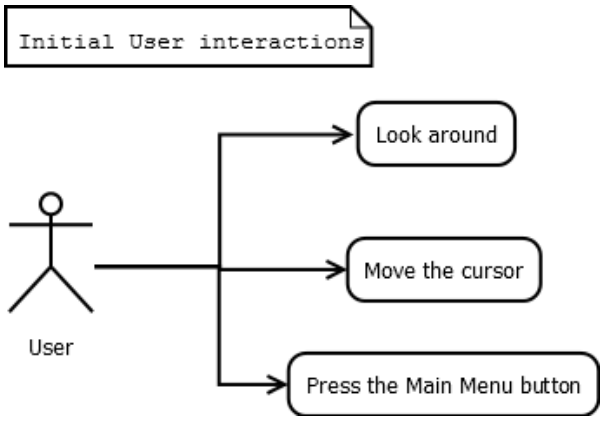


Figure 4.8. Initial user interactions.

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

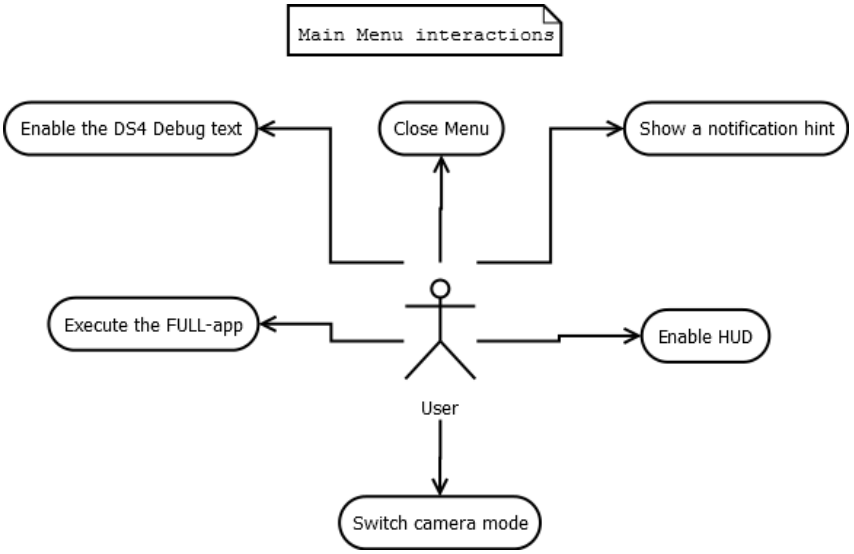


Figure 4.9. User Interactions in the main menu.

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

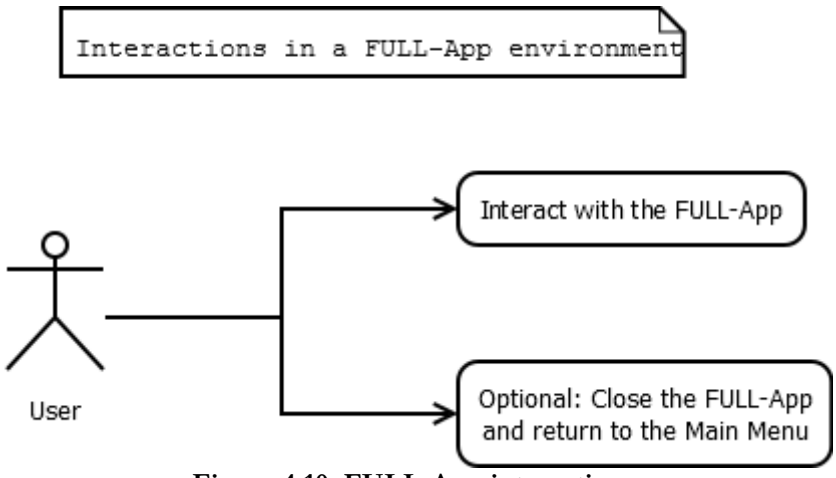


Figure 4.10. FULL-App interactions.

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.

Holo-Board manual

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.

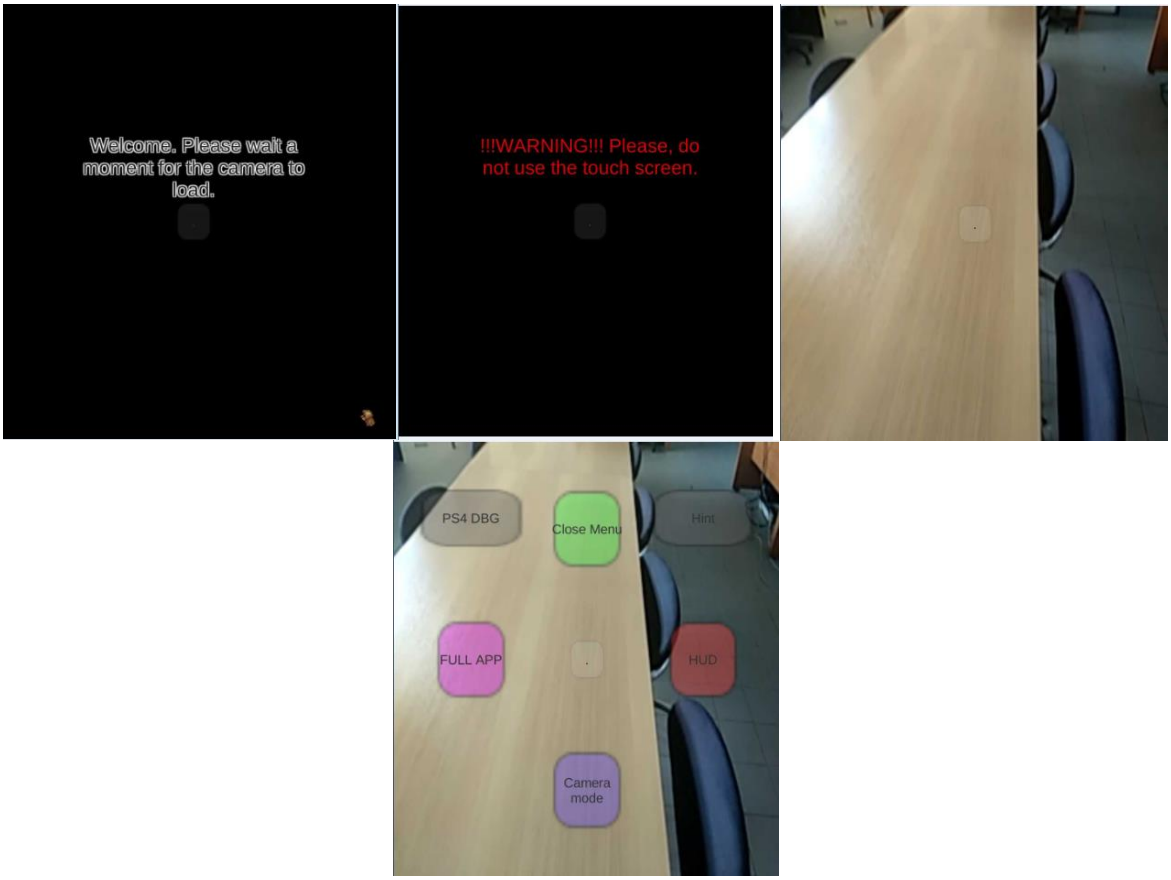


Figure 5.4. Intro screen while the camera loads (Top Left). Warning message if the touch screen is used (Top Right). The camera is loaded (Bottom Left).The Main Menu (Bottom Right).

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.

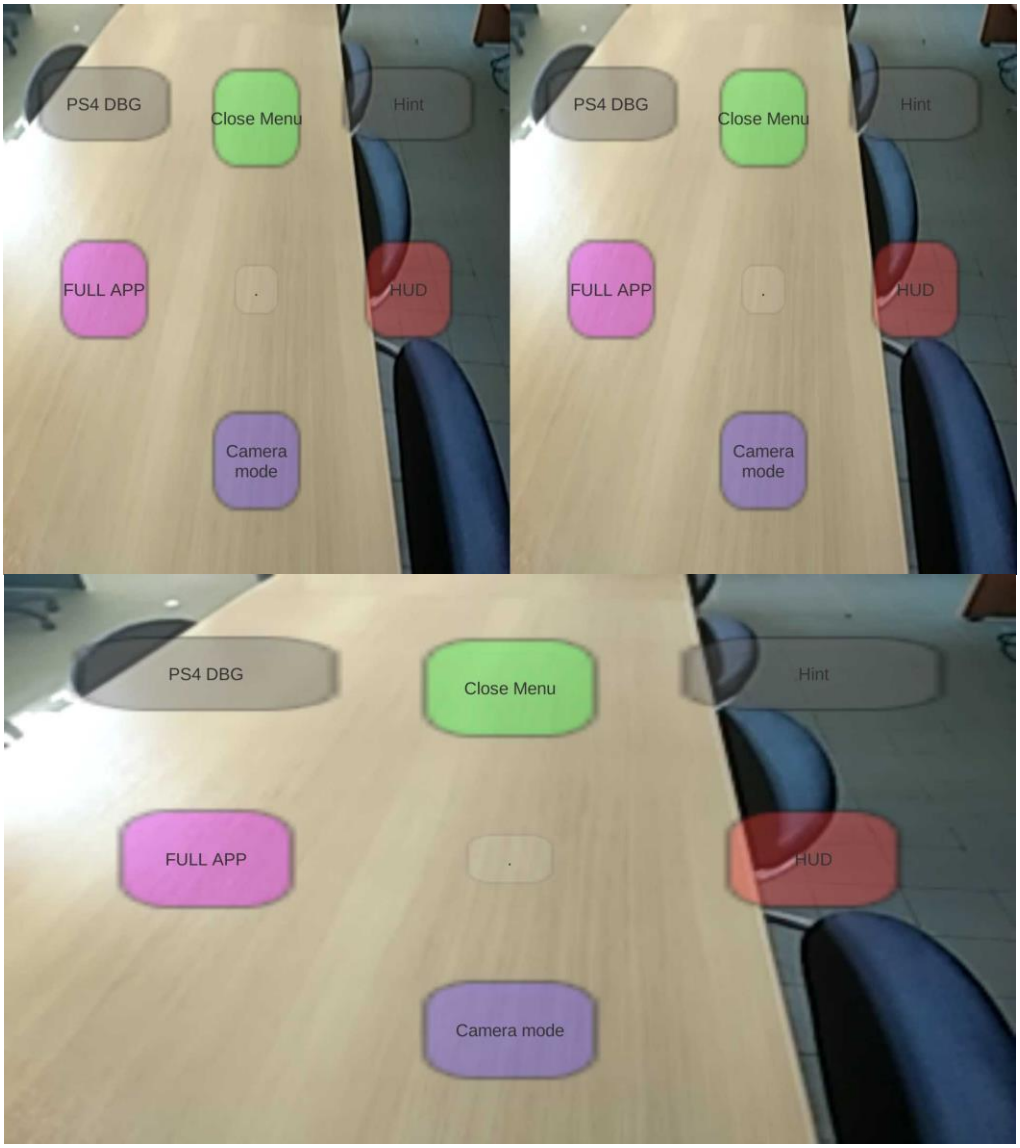


Figure 5.5. Dual camera view (Top). Single camera view (Bottom).

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.

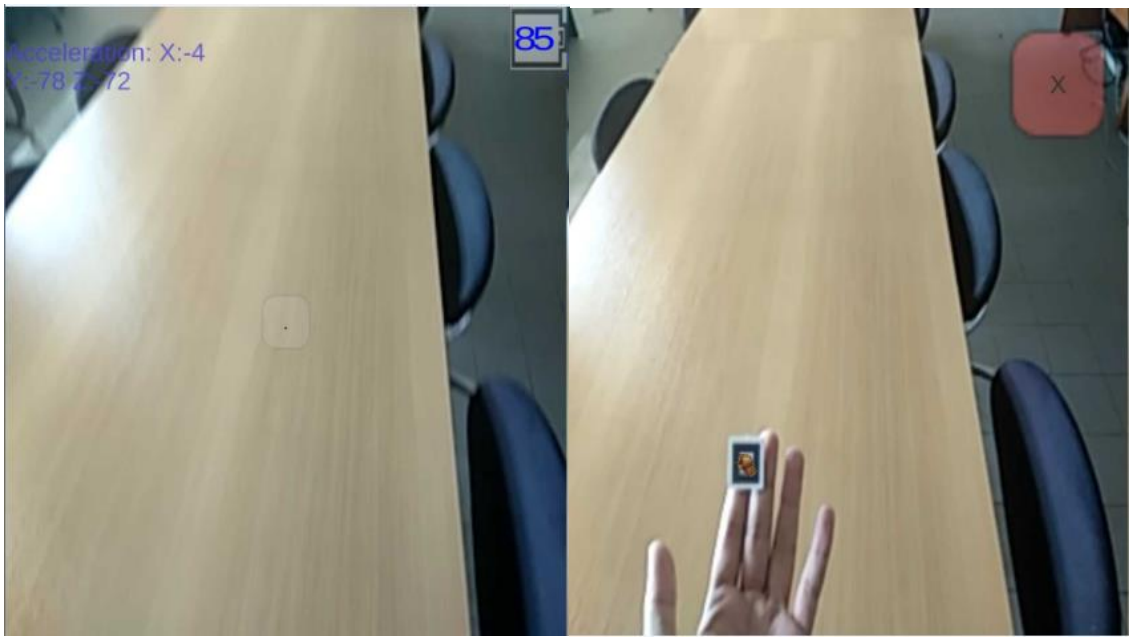


Figure 5.6. The HUD-App demo (Left). The FULL-App demo (Right).

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.

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.

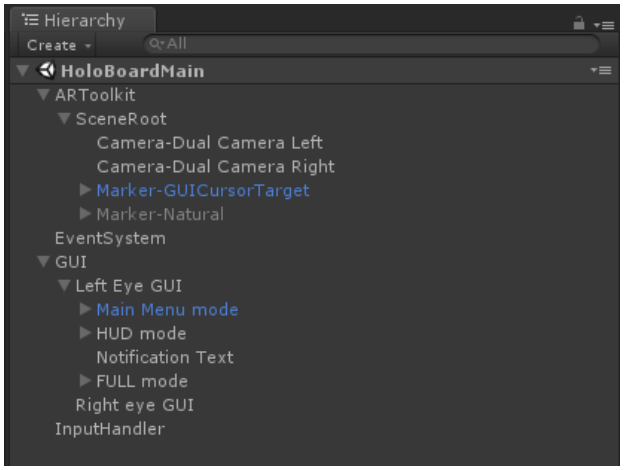


Figure 5.8. The basic hierarchy.

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.

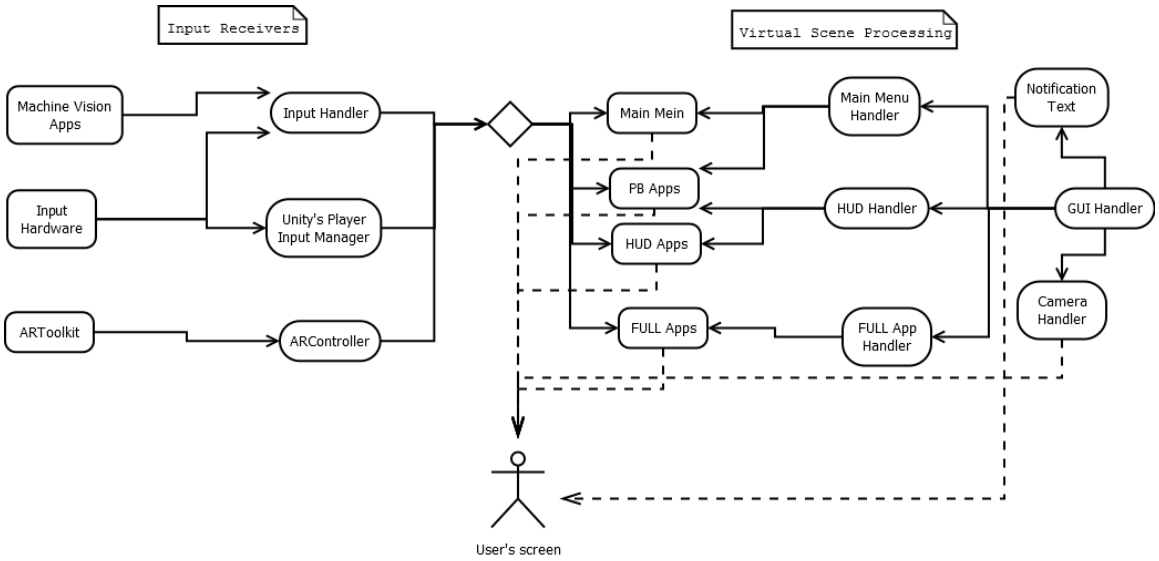


Figure 5.9. Holo-Board's Architecture.

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.

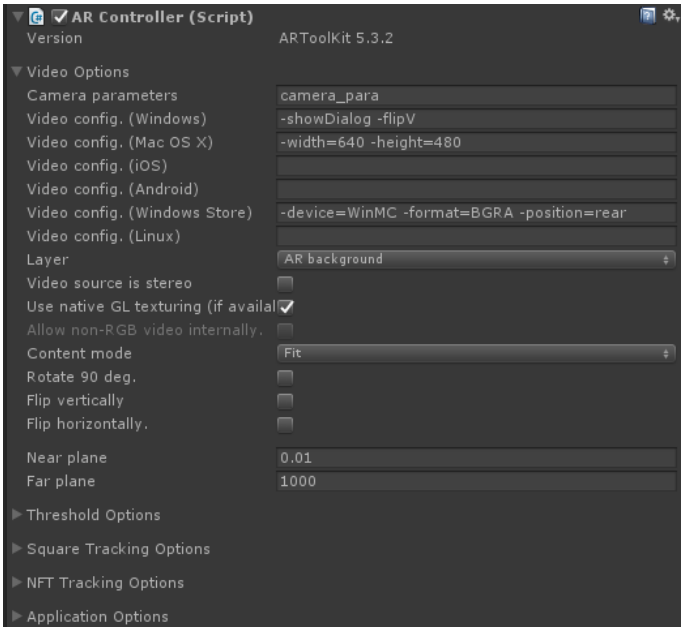


Figure 5.10. AR Controller script Inspector window.

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.

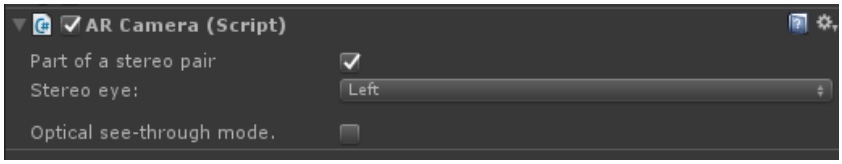


Figure 5.11. AR Camera inspector window.

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).

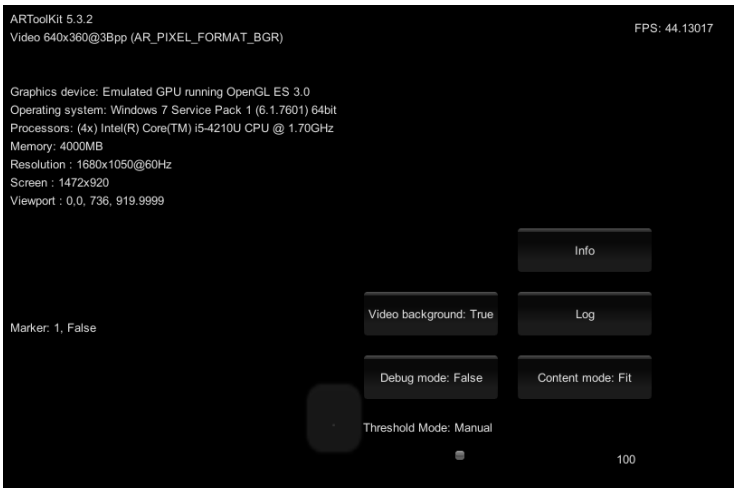


Figure 5.12. ARToolkit on runtime debug menu.

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.

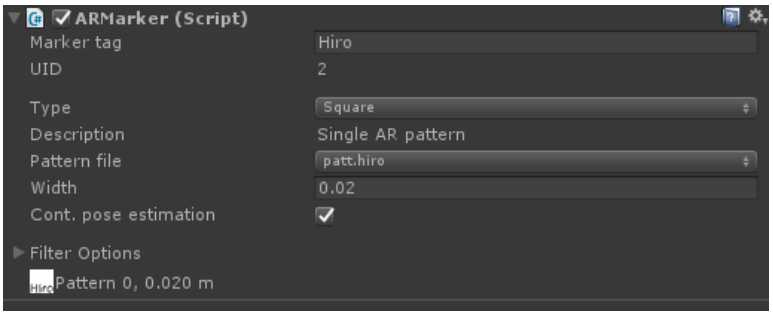


Figure 5.13. AR Marker's Inspector window.

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.

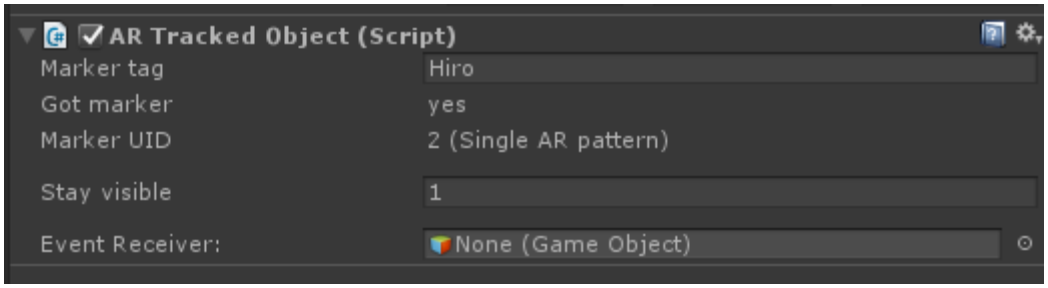


Figure 5.14. AR Tracked Object Inspector window.

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.

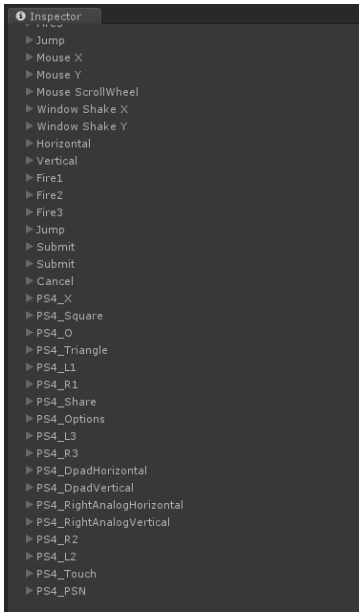


Figure 5.15. The DS4 inputs in the Player Inputs list.

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.

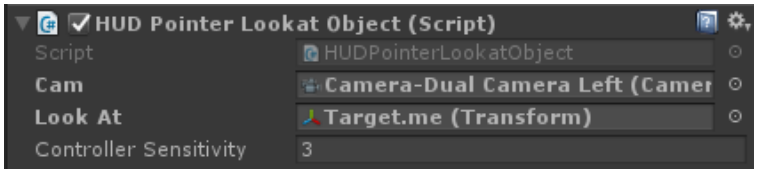


Figure 5.16. GUICursor's controller script.

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.

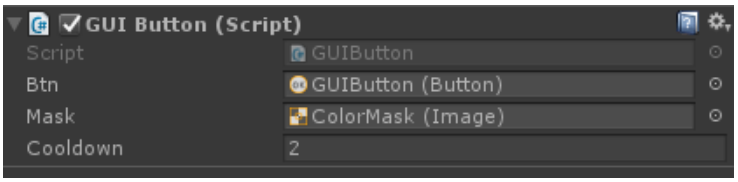


Figure 5.17. The GUI Button script.

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.

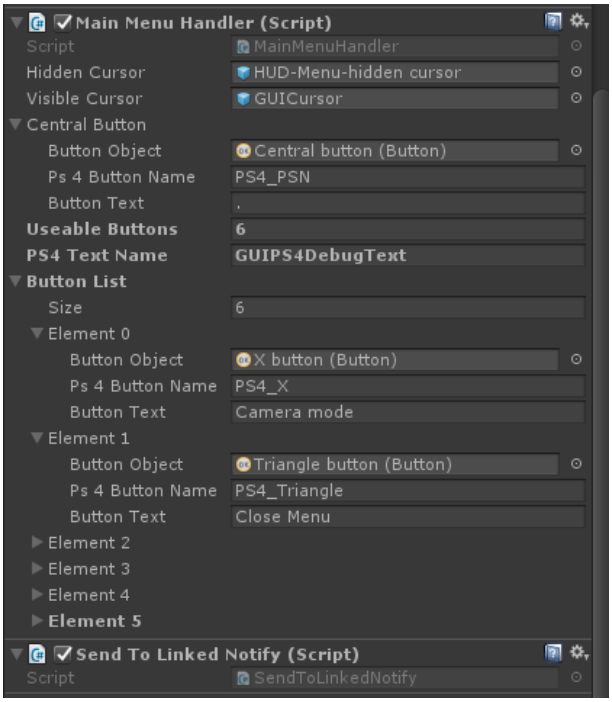


Figure 5.18.The Main Menu Handler script's inspector window.

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.

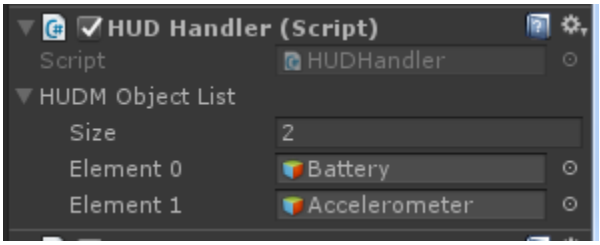


Figure 5.19. The HUD Handler script's Inspector window.

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.

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.