ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura
Campus di Cesena
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

DESIGN AND DEVELOPMENT OF A UNITY BASED FRAMEWORK FOR AUGMENTED WORLDS

Elaborata nel corso di: Programmazione Avanzata e Paradigmi

Tesi di Laurea di:
PIERLUIGI MONTAGNA

Relatore:
Prof. ALESSANDRO RICCI
Co-relatori:
Dott. ANGELO CROATTI

ANNO ACCADEMICO 2015 - 2016
SESSIONE I

Keywords:

Mixed-Reality

Augmented World

Unity

Hologram

To my beloved aunt,
family and friends,
whose words encourage me
every day ...

Abstract

EN The way we've always envisioned computer programs is slowly changing. Thanks to the recent development of wearable technologies we're experiencing the birth of new applications that are no more limited to a fixed screen, but are instead sparse in our surroundings by means of fully fledged computational objects. In this paper we discuss proper techniques and technologies to be used for the creation of "Augmented Worlds", through the design and development of a novel framework that can help us understand how to build these new programs.

IT Il modo in cui abbiamo sempre concepito i programmi sta lentamente cambiando. Grazie allo sviluppo delle recenti tecnologie wearable stiamo assitendo alla nascita di nuove applicazioni che non sono più limitate ad uno schermo fisso, ma sono invece disperse all'interno dell'ambiente, aumentandone le caratteristiche per via di una serie di oggetti virtuali. In questo documento vogliamo analizzare quali tecniche e tecnologie utilizzare per la creazione di "mondi aumentati", attraverso lo sviluppo di un framework che possa aiutarci a comprendere meglio i diversi punti di vista legati alla creazione di questi nuovi e innovativi programmi.

Contents

Chapter 1

Introduction

If computing companies have their way, 2016 will be the year in which augmented and virtual reality become widely popular. Different firms such as Facebook, Sony and Microsoft are getting ready to launch their set of high tech wearables. Google and Qualcomm are hard working on area learning, using computer vision technologies. Meta is ready to launch his new headset and is finding new ways to have a more natural interaction with holographic entities. Looks like we've finally reached a point that not so long ago we envisioned only in sci-fi movies.

What these new technologies seems to have in common is the usage of some sort of headset that, being equipped with different types of sensors, is able to alter the user perception of the world. This headset can either display new information on top of what the user is already seeing, in which case we talk about *Augmented Reality*, or render a new world entirely in front of his eyes; we call this *Virtual Reality*.

Virtual Reality is fully immersive: the headset must, by necessity, block out the external world. Using stereoscopy, a technique that fools the brain into creating the illusion of depth, it transforms a pair of images into a single experience of a fully three-dimensional world. At the moment, main examples of VR technology are Facebook's Oculus Rift[1] and HTC's Vive[2], and their primary business is the gaming industry.

Augmented Reality on the other hand, must maintain its user connected with the real world, allowing heads-up displays to be used instead of a closed headset. It makes use of interest points, fiducial markers or optical flows in the camera images to find the right way to overlay digital content on top of a video stream. Even though AR is nowadays mainstream and widely used inside mobile applications, we are intrigued by the possibilities brought by this technology, as recently seen with the debut of Microsoft's Hololens[3], that aims to liberate computing from a fixed screen, overlaying its user with useful additions inside his environment.

Augmented Reality is just the starting point of a new model for the creation

(a) Microsoft Hololens

(b) HTC Vive

Figure 1.1: Figure 7.3c shows an headset with see-trough lens. Figure 7.3d is a closed headset for virtual reality, provided with special gears to track movements in a closed fixed space.

of real-time applications that place the user in-between the virtual and physical world. As technology is moving forward, new programming scenarios open of which we need to find ways to meaningfully encapsulate their complexity, for developers to build new systems.

In this document we are going design, implement and use a novel programming framework for Augmented Worlds. Each chapter is part of a process that wants to extend the knowledge of what is on top of the model.

The first chapter is about what defines an Augmented World, describing the model through a list of its related concepts. Here we analyze critical points required for a software stack usable for the creation of these applications, while arguing about different programming abstractions.

In the following chapter we are going to design the framework starting from the main concepts of the model. We will talk about the framework's goals and have a bird-eye view of the system architecture, focusing on macro aspects of all required parts. Carrying on, we will talk about Holograms and HoloDoers, as basic units of work proposed for the envisioned framework abstraction.

Since our framework is based on the Unity videogame engine, it's necessary to introduce its functionalities. The third chapter will explain the basics for developing real-time applications with Unity, arguing about how the engine works under the hood. Here we talk about how to use the editor, the High Level API (HLAPI) for networking and Vuforia extension for Augmented Reality.

The fifth chapter will be about the implementation of the previously exposed framework's essentials, giving details about its basic units of work. Here we will explain how to properly use all parts of the framework as well as what they are meant for.

We will then conclude with more general thoughts, but only after giving a taste of our work through some practical example.

Chapter 2

Augmented World

The impressive leap in technology has reduced the gulf between digital and physical matter. In 1994 P. Milgram and F. Kishino defined the concept of *Mixed Reality* as "...anywhere between the extrema of the virtuality continuum."[4], extending from completely real to completely virtual environments, with augmented reality and virtuality ranging between.

Augmented Worlds are programs that span into Mixed Reality, increasing the functionalities of the physical environment "by means of full-fledge computational objects located in the space, that users can perceive and interact with by means of proper mobile/wearable devices"[5].

AW programs are meant to represent an extension of the real world, in which humans and artificial entities can interact and collaborate with each other, making use of augmented entities located in the environment. These entities exists independently from the actual presence of users, they can possibly have a virtual or physical body, be complex or rather simple, dynamic or static.

Augmented Worlds are multi-user systems, meaning that the state of the world is shared between multiple devices in a way that can enable collaborative actions, through the use of augmented entities, performed in the most natural way possible.

Another important aspect of this concept is that the user is part of the environment, making him visible to computable entities that could perform some kind of behaviour autonomously or upon interaction. Moreover actions executed in the physical environment can have an impact on the virtual environment as well and vice versa.

Augmented Worlds are immersive environments that evolve trough time, independently from his users. Behaviour based programming, artificial intelligence or even the BDI model[6] can be used to build strong autonomous entities that can alter the state of the world in a way that can benefit his inhabitants.

An AW program could be anything related to hands-free operations, like the cooperative building of a machine, where the user, equipped with an headset, is as-

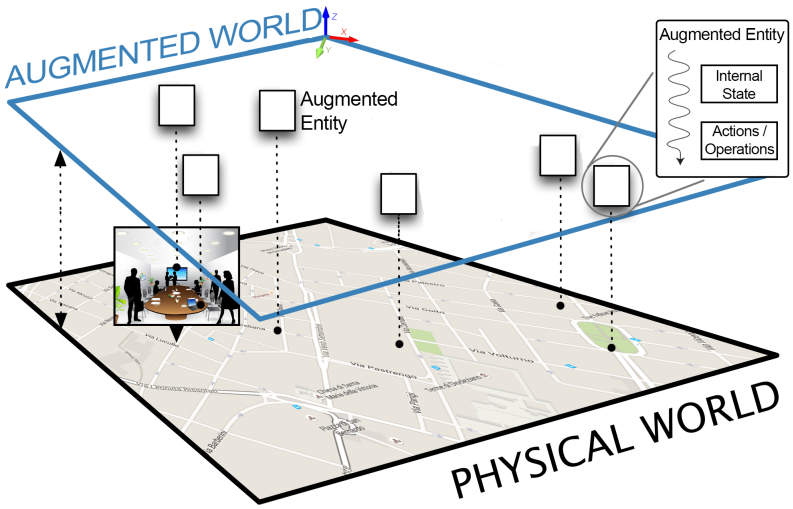


Figure 2.1: A simple representation of an Augmented World, focusing on spatial coupling aspect of augmented entities.

sisted by real time information about the process, making the assembly procedure easier to follow.

A more complex and intriguing example is a smart city or office fully equipped with sensors and actuators where coworkers can constantly perceive and influence the information layer around them using their mobiles and wearables.

What we've just described is now possible thanks to the enabling technology developed through the past years. With Augmented Reality we can render virtual beings on top of the user field of view, moreover using sensors and ad-hoc APIs, it's possible to tack the position and orientation of physical objects in real time. Dynamism and graphics can be obtained using a well structured 3D real-time engine, meanwhile networking systems and state synchronization[7] techniques can also be used to achieve casual consistency. These technologies are the base for building such systems, however we're still quite far from the real conception of how to use them to build an Augmented World Framework.

After this brief introduction, we have a feeling that the presented concept brings some interesting challenges. Head of the list the need of a clear model that pinpoints all the basic logical units of the system, so that a meaningful architecture for AW programs could be made. Follows a general understanding of the current state of the art for technologies and software that can support this idea, including wearables and relative APIs.

Following this chapter, we give major concern to problems and solutions relative to the development of real-time distributed systems, 3D real time engines and virtual/physical space coupling.

2.1 Main concepts

Here we see different aspects that feature an Augmented World program.

Space coupling What is meant to be represented is meant to have a position in the physical environment, space coupling means that representation of virtual entities is tight to one position in the real world.

When an augmented entity is instantiated at runtime, its local world coordinates must be specified, it would be then job of the system to dynamically bind this position to a specific point in the physical space. For this mechanism to work, the system needs to have some sort of reference of the environment, in order to acquire and apply the transformation matrix used to align the virtual world on top of what the user his seeing trough the device.

One way to make it possible is by using computer vision, where markers or points of interest could then be recognized directly from the camera video stream. Recently sensor fusion and smart terrain technology is used to obtain even greater results, making it possible to work in complex environments. One interesting result is shown by Google Project Tango[8], enabling applications to perform simultaneous localization and mapping within a detailed 3D environment.

The same must be possible for the other way around. Whenever a physical machine is an extension of an augmented entity, his physical position and orientation should constantly be streamed down to his virtual counterpart. This is necessary if we want to build a seamless mechanism where augmented entities can actively reason with the position and orientation of objects. For this mechanics to work, of course, we need some kind of sensor or system that can track one or more object positions in the environment. For large environments a gps and magnetometer might be used to obtain this data, while a bluetooth positioning system might be used for small indoor areas.

User modeling and interaction An augmented world is typically a multi-user application, where different users continuously influence the world and interact with each others.

When an user joins a particular augmented world, a new special entity is instantiated inside that world to which he's associated. This particular entity becomes the avatar of that user, meaning it becomes his medium of interaction. All user commands and actions must be performed through this entity, of which he has the authority.

The user's avatar should probably have a position relative to his physical location, in order reason based on this criteria. For example, special entities might be programmed to start a behaviour only when the user is near them or in a specific

area. Imagine the case where the user enters a dark room and suddenly an entity turns on the lights for him.

This being said, interaction between the user and the augmented world should be made through this particular entity, meaning that it encapsulate all meaningful actions that the user can perform inside the virtual world. An access control system could also be placed between user entities and regular AW objects, in order to have a more complex permission-based interaction system.

Physical embedding The virtual extends into the physical not only trough the user's device screen. Some augmented entities could also have a well defined body that resides inside our own world.

We are talking about physical embedding, intended as the practice that binds computational aspects of the Augmented World with devices and machines situated in the real environment.

The entity's physical body is intended to be the extension of some virtual object inside the Augmented World; communication between this entity and his body could be made through use of device specific commands and messages. Of course this devices should be configured in a way that enables data transfer from and to the Augmented World.

Real Time As we are creating a virtual world to be exposed upon our own, it's useful to clarify that the system expects to have some real time components.

Computational entities living in the virtual realm should be able to react actively to real world events and proactively start procedure based on their own belief. This entities are in fact "alive" meaning they should evolve and reason over time, this is the basic need for creating strong behaviour based entities.

Time consistency becomes a main point inside the model, since some of the entities own a representation that is constantly updated, their time long actions should be regulated by means of some internal notion of time, shared between all virtual entities to ensure a consistent evolution over time.

Actions like moving from point A to point B at constant velocity, animations and time-based events all need the notion of time to be uniformly updated and shared across the network.

2.2 Towards a programming model

Before starting with the design of some specific programming toolkit, lets focus on the main objectives of the Augmented World model, how can they be reached, what problems should be faced and what technologies we have at our disposition.

Starting with a fully OOP perspective we notice that most of the complexity of this model is left unhandled. We are able to shape all static aspect of augmented entities, but everything that is actively handled like events, user interaction and commands, time long actions, etc, would require the implementation of an ad-hoc application layer.

An agent oriented abstraction is indeed nearer to the problem we are facing, allowing augmented entities to be fully autonomous, encapsulating not only a state but also a behaviour. Interaction is handled in a fully asynchronous fashion similar to the real world case. An augmented world program can then be shaped in terms of autonomous agents situated in the virtual environment, making the bridge between the physical and virtual counterpart.

However we're still left with quite a lot of complexity to fill the gap between the agent oriented abstraction and the problem we are facing. The following concerns needs to be faced with the right approach to be efficient in concern of having a scalable system architecture and the creation of a ready to use framework.

State synchronization State synchronization between local and remote instance of the augmented entity should be made in a way that is transparent to the user, but still effective for both the network efficiency and a programming perspective.

Let us consider a star network topology where the central server holds the concrete instance of the augmented world. AW applications should constantly receive information from the central node so that their representation and belief of the world is at some extend consistent. From the other way around, user commands and actions should be sent by clients and handled directly on the server. For the developer it's then needed:

- a way to continuously stream data from server to clients.
- some sort RPC mechanism or messaging system with an efficient way to serialize data.

Visual representation Visualization of augmented entities is another main concern. As the state of the world is constantly updated, changes in one object graphics should be visible in real time, with time based movement and animations. This means that AW applications should be equipped or based on a real time graphic engine, on top of which augmented reality functionality could also be implemented.

For an excellent graphical effect we should also consider the possibility of implementing a mixed sorting layer that takes into account both virtual and physical objects of the scene, so that we are able to see only what we effectively have in

front of us. Different ways to achieve a mixed sorting layer are based upon advanced computer vision techniques and sensor fusion, that can constantly update the virtual world with occlusion masks.

From a programming perspective all that has been just discussed should happen in a way that is transparent to the developer. When an object is instantiated into the scene it will be rendered automatically by using its local position, its graphical properties and the camera field of view.

The handling of animations and other time long actions should be regulated by a fixed notion of time. This usually done by writing a function that is called before every frame is rendered on screen, in an update loop pattern[9] fashion.

Chapter 3

Envisioning the framework

The following section will be about the design of a simple framework for Augmented Worlds. This is not intended to be an optimal design for the model, but mostly an experiment to bring us closer to what might be like to work with such complex a system.

The design process will take into account the core features of the model. We will focus mostly on augmented entities, space coupling, state synchronization and entities visual representation.

3.1 Goals

We want to develop a software abstraction that is close to the Augmented World model; that can enable the creation of real-time 3D applications based on a multi-user shared augmentation of the real world.

The main goal is to provide the developer the right tools to build AW programs easily, by composing the environment with augmented entities and some basic units of work.

Asynchronous actions and object dynamics, should be a concrete functionality provided to the developer. Their mechanism and code should be hidden inside the core of the application, without him worrying about how such actions are actually performed.

The same applies also for Networking and state synchronization. The goal is to hide all complex networking and serialization aspects of communication from the developer and make him only worry on how to perform interactions between the user and augmented entities.

Space coupling is provided with the minimal configuration from the developer, that should only be able to specify references of the physical environment used by the system to bind virtual entities.

Another important aspect of this framework regards modularity. The software stack comes with a variety of subsystems that work together, each adding a major contribute to the provided functionality of the application. For example, different modules could handle physics, networking, collision, input; and of course some of them wight not be independent from each other.

The goal is to build a framework in which at least the functionality regarding graphics and network communication can be switched out without making the developer rewriting most of the application code. This should allow more flexibility to the developers, that can choose from a variety of supported system, as well as writing their own.

3.2 Topsight of the system

For his nature an augmented world program is based on a distributed system. Different clients all connects to the AW mainframe where the instance of the Augmented World is running.

The mainframe, being this a central server or a more complex cluster of machines, constantly informs clients of changes of their surroundings so that users can see a consistent augmentation of his environment. Therefore, the server holds the current state of the world, while clients are the window to the Augmented World for the user to see through.

The client's application should be able to create the effect of space coupling, by simply using some reference of the physical environment and the entities position, rotation and scale. This application should also be able to send commands to the server to inform when user wants to perform a particular action meant to change the state of the world in some way.

At the core of each instance must reside a well tight set of subsystems; this set is part of the *engine* of the application, needed to cover all aspects of the simulation. What subsystem is actually required may vary from server to clients, for instance, sound and graphics might not be concern of the server at all.

These are the required blocks needed by the AW system:

- A 3D dynamic model of the world: required to model space and to have a consistent measurement system for object positioning, orientation and scale. This is meant to be part of the same engine that regulates how the state of the world changes during time.

- A state synchronization system: to be used for both streaming data from server to client and message sending in general.

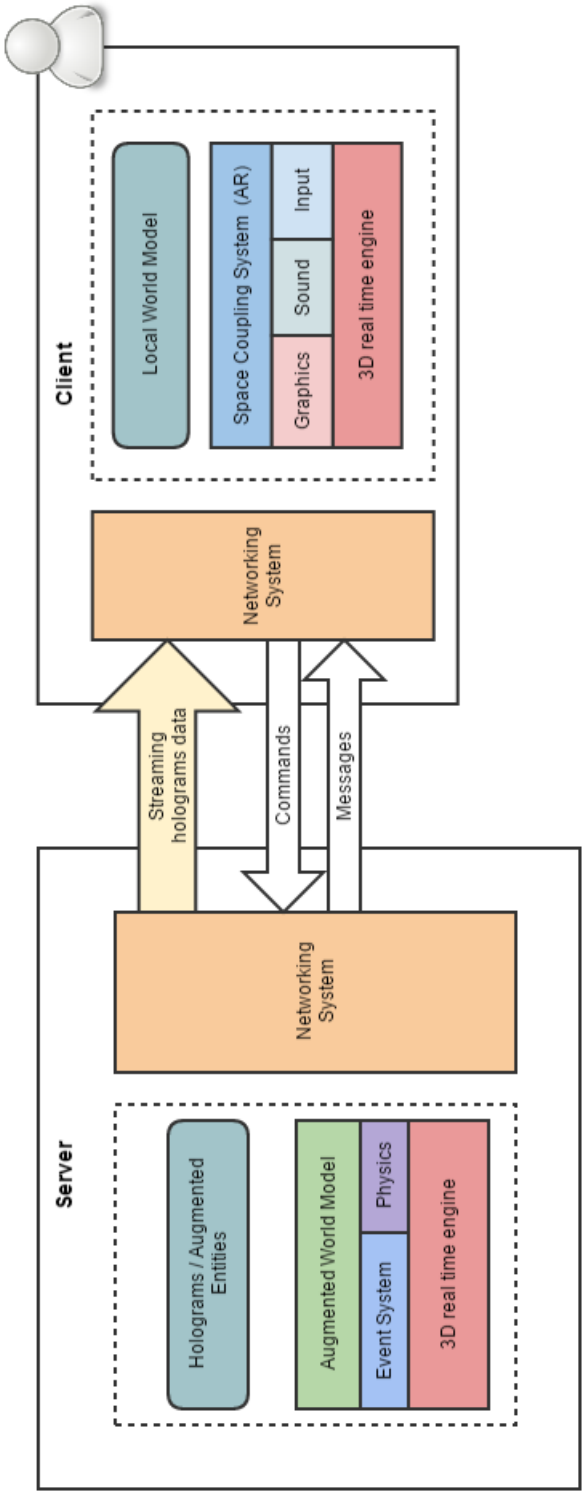


Figure 3.1: The image shows an abstract representation of the system. Each block can be provided by a unique software stack and it's an independent core functionality of the runtime engine. The figure doesn't necessary means that each part can't be located on both server or client side, it only shows what blocks are required on each process. The networking system is split between server and client, each managing a different side of communication.

- A space coupling system: that can correctly overlay onto the user perspective augmented objects on clients (for example, using Augmented Reality technology).

3.3 Hologram

Until now we've stated that the system is shaped in terms of augmented entities, being them static or dynamic. Let's focus on the static ones, that we call Holograms.

An Hologram is a three-dimensional image perceived by the user trough his device. It can be simply a non-interactive object, like compass always pointing in a direction, or either be a tool that the user can use to perform some kind of action.

An Hologram is instantiated at runtime, and it's position is bound the a specific point in the physical environment.

His visualization must be, at some degree, consistent between all devices connected to the same Augmented World. It encapsulates a flexible mechanism for state synchronization, and as well exposes methods in order to allow remote interaction.

Like in a standard OOP object, this particular augmented entity is completely static, it doesn't encapsulate a behaviour, but only a state. This state can be however continuously altered by some other real-time component in order to expose some kind of dynamism.

Applying separation of concerns, we divide an Hologram in three function specific parts.

View The view object should expose methods and functionality regarding only the visual representation of the entities, like changes in shape, material and animations. It could eventually encapsulate response to device specific reactions to user input: like key pressing, gestures, etc. This component is strongly tight to the graphic module used inside the application, since it is the software stack that actually renders the object onto the screen.

Model The model object encapsulates the state of the Hologram and exposes methods that alter his properties. It is meant to be instantiated only inside the server, but a local copy could also be present inside the client for fault management.

State Synchronization The last object, the state synchronization one, is based upon the networking system. It is the bridge between the local view and the remote model, regulating how data received from the server is interpreted and the view

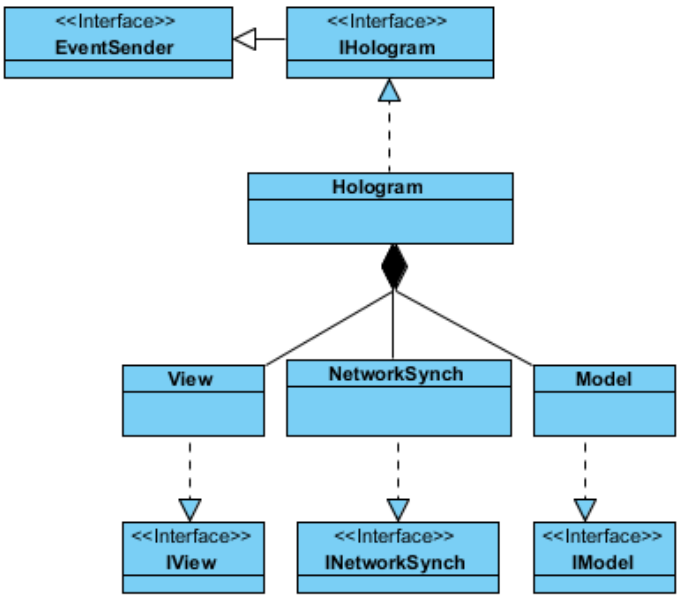


Figure 3.2: The image shows the conceptual diagram of the Hologram. An Hologram is composed by three different parts: the View, the NetSynch and the Model. This separation ensure that changes regarding these subsystems, like the state synchronization one, only affects part of the general application.

updated. It also encapsulates the behaviour in regard of received messages and user commands for that specific hologram.

We want to apply division of labour between concern specific parts of an Hologram. From a programming perspective, however, we want a seamless interaction with a specific Hologram, there should be no need to directly call methods of each Hologram components distinctly. The Hologram must then be able to direct calls of generic methods to his parts.

When an Hologram performs an action, some other entities might take notice. We recall that an Augmented World is inhabited not only by the user, but by autonomous entities as well. When something of matter happens in the environment, these special entities needs to take notice. In this perspective, an Hologram is an *EventSender*, thought as something that is able to release some sort of information in the environment, of which special entities might take notice at the time.

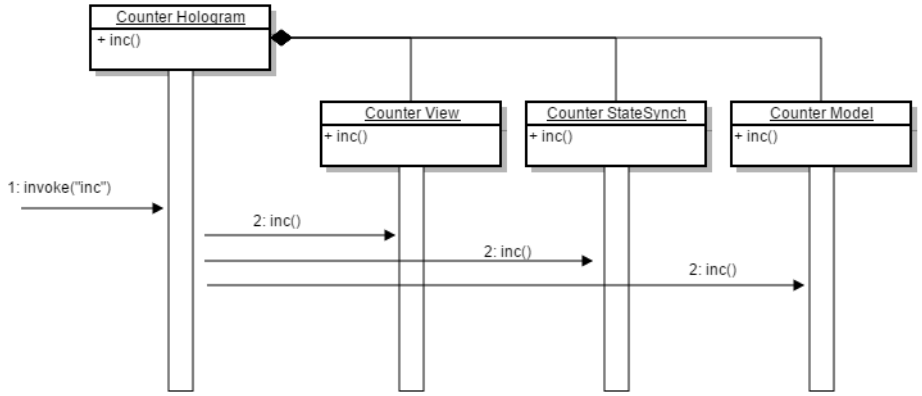


Figure 3.3: In this example, some external entity requires the Hologram to perform the action "inc()", that increments the value of some counter. Each part of the Hologram takes in action a different aspect of this call, the view might update his representation, the model increases the holding value, the NetSynch propagates the call to the server.

3.4 HoloDoer

By only relying on Holograms, it's not possible to shape entities that require some kind of dynamism, for this reason we introduce the HoloDoer. An HoloDoer is not a static entity like the Hologram, instead it is based on top of the real-time capabilities of the application, unlocking dynamism within a consistent self-evolution over time. His name remarks how it is intended purpose is handling one or more Holograms, in this sense, an Hologram might expose a dynamic behaviour when his visualization is continuously updated by an HoloDoer.

This entities are what makes the Augmented World truly "alive", ranging from simple units of work to full-fledged autonomous entities. It's important to notice that this entities don't require to be shared across all clients, they only resides inside a specific instance of the program, being this mostly the one hosted by the server.

In this perspective, an HoloDoer should be used to build autonomous entities residing in the Augmented World. These entities might be able to proactively react to changes in the environment, or even trigger some behaviour when an user or Hologram enters a specific area.

An event context is a purpose-specific set of listeners implemented by the developer or created dynamically at runtime. An HoloDoer registers itself to one or more event contexts to which it is interested; it can then react accordingly to events generated inside that context.

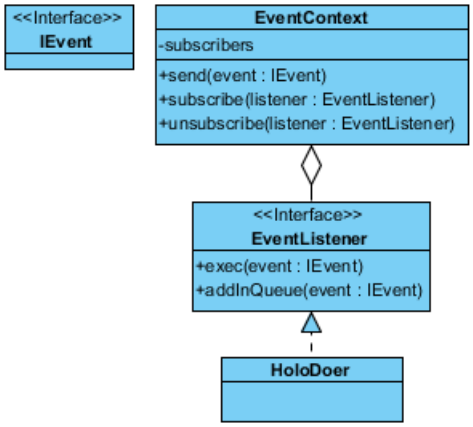


Figure 3.4: An HoloDoer conceptual diagram. This entity implements the EventListener interface, meaning it might be subscribed to some event context and react to specific events.

In other words, the HoloDoer is an "agent" that is meant to manage dynamic aspects around Holograms, it can act on its own and react to application based events from the event contexts it's registered to, being them generated by Holograms or other HoloDoers.

An Holodoer doesn't have a visualization, however it can make use of an Hologram to compose a fully autonomous augmented entity shared between different clients.

The HoloDoer object is also used as a base for more framework specific concepts, like Tracking Areas.

3.5 User modeling and interaction

When a user joins the Augmented World, a special kind of entity must be instantiated inside the server. This entity is bound to one definite user and it's destroyed at the end of the client specific connection with the server.

We call this entity an "user agent" as it is the virtual counterpart of the user, executing actions and commands on his behalf. This entity is strongly tight to the networking system, since it is the point where user sent commands and messages are interpreted and then eventually executed. This calls for a standardization of messages structure, ensuring that parameters sent over the network can be unserialized in some way.

The User Agent is dynamic and handles Holograms on the user behalf, moreover

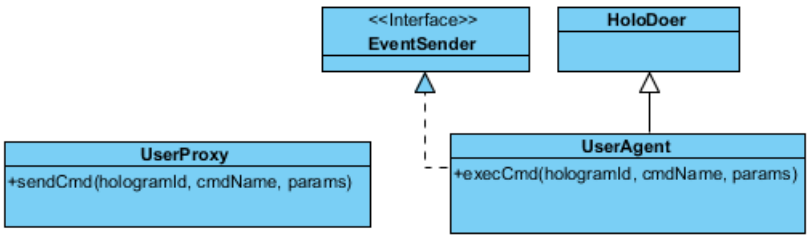


Figure 3.5: Structural diagram of object regarding user modeling. UserAgent is the object that place the user inside the Augmented World. UserProxy is a simple entity that is used for remote interaction.

it could also have an Hologram associated representing the user itself.

The User Agent is then meant to be part of the server application; on the client side an User Proxy object is required in order to control the remote instance. This object is of course tight to the adopting networking system, and is from the client point of view the one gateway for user specific messages to the Augmented World.

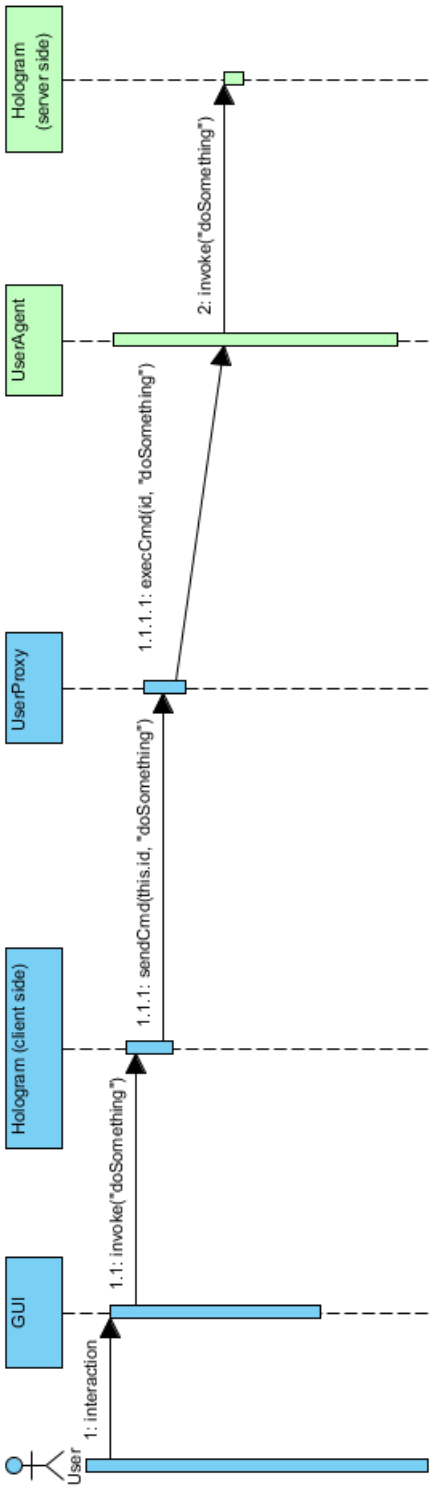


Figure 3.6: Example of interaction inside the framework. Notice that there's not guaranteed response from the server, effects might be seen from the client only if the action was successfully executed on the Hologram. Other mechanisms can be placed between the user agent and server side holograms to create a more complex permission based access system.

Chapter 4

The Unity Game Engine

Unity is a videogame engine well set in the industry, with an emphasis on portability, it allows development of 3d real time applications on top of different graphical libraries, including: Direct3D, OpenGL, OpenGL ES and other proprietary APIs for video game consoles. Unity gives support to most commonly used 3D assets and formats, moreover it has an increasing sets of functionality like: texture compression, parallax mapping, SSAO, dynamic shadows, bump mapping, fullscreen post-processing effects, etc. The engine is targeting more and more platforms, and is also becoming one of the central technology onto which most known Augmented and Virtual reality tools are based upon.

4.1 Basics

For building strong Unity-based applications, one must be first familiar with the provided Editor. The editor is a standalone program used not only to arrange objects into the scene, it is in fact an essential part of the developing process, providing useful functionality in the configuration of overall aspects of the program. Scripts order of execution, configuration of in-scene properties at start time, key-command binding, rendering properties, audio and image compression, are all features handled by this tool.

The main editor window is made up of several tabbed panels known in Unity as Views. There are several types of Views in Unity each one with a specific purpose.

The Project Window displays the library of assets that are available to use in the project. When you import assets into your project, they appear inside this view. This include prefabs (precomposed objects created directly from the scene), textures, materials, audios, meshes, etc.

Figure 4.1: The Unity Editor interface.

The Scene View allows you to visually navigate and edit your scene. This view has a 3D and 2D perspective, depending on the type of project you are working on. The Scene View can be used to select and position scenery, characters, cameras, lights, and all other types of Game Object.

The Hierarchy contains every GameObject in the current Scene. These objects can either be legacy Unity entities like basic shapes, lights, the camera, while other might be assets imported into the project. Objects can be arranged inside this view making use of Parenting; simply by dragging one object on top of another. As objects are added and removed from the scene, they will appear and disappear from the Hierarchy as well. By default the GameObjects will be listed in the Hierarchy window in the order they are made.

The Inspector Window shows all the properties of the currently selected object. These properties are public variables defined in the component script, and can be edited at both setup and run time. Because different types of objects have different sets of properties, the layout and contents of the inspector window will often vary.

The Toolbar provides access to the most essential working features. On the left it contains the basic tools for manipulating the scene view and the objects within it. In the center are the play, pause and step controls for running the application inside the editor.

4.2 Component based development

Unity is a component based engine, adopting a design pattern that was originally pioneered in order to avoid annoying class hierarchies. The idea is to package all functionality of Game Objects into separate behaviour-based scripts. A single GameObject is just the sum of his parts, being them legacy or user written components.

Unity comes with a vast set of legacy components used to extend one Game Object's functionality in terms of graphics, physics, user interface, audio, etc. User written scripts are mostly meant to be newborn components for GameObjects, following a well defined structure in concern of the engine execution lifecycle[10].

This reused-based approach to defining, implementing and composing loosely coupled independent components into systems is widely used in game engines and is one of the fundamental design pattern adopted to give the user the right flexibility to deal with most of the complexity brought by 3D real time applications[11].

GameObject components can be assigned inside the Inspector View, or by script, using the *AddComponent* call of GameObject. Each public property is directly rendered inside the Inspector with a special controller, to aid the developer in both configuration and debugging.

4.3 Runtime Engine Overview

Most of the complexity of Unity built games is hidden behind the inner runtime engine. This engine is the core of all Unity applications and even if his main structure is hidden from the public eye, we can still have a simplistic view at its possible implementation.

The Unity runtime is written in C/C++. Wrapped around the Unity core is a layer which allows for .NET access to core functionality. This layer is used by the user for scripting and for most of the editor UI.

This core is what really handles, in what is expected to be the most efficient way possible, the application main loop, including aspects regarding resource handling, front-end initialization and shoutdown, input decoupling and more.

At their heart, graphical real-time applications, such as videogames, are driven by a game loop[12] that performs a series of tasks every frame. By doing those tasks every frame, we put together the illusion of an animated, living world. The tasks that happen during the game loop perform all the actions necessary to have a fully interactive game, such as gathering player input, rendering, updating the world, and so forth. It is important to realize that all of these tasks need to run in one frame.

The most straightforward implementation of a game loop is to simply have a

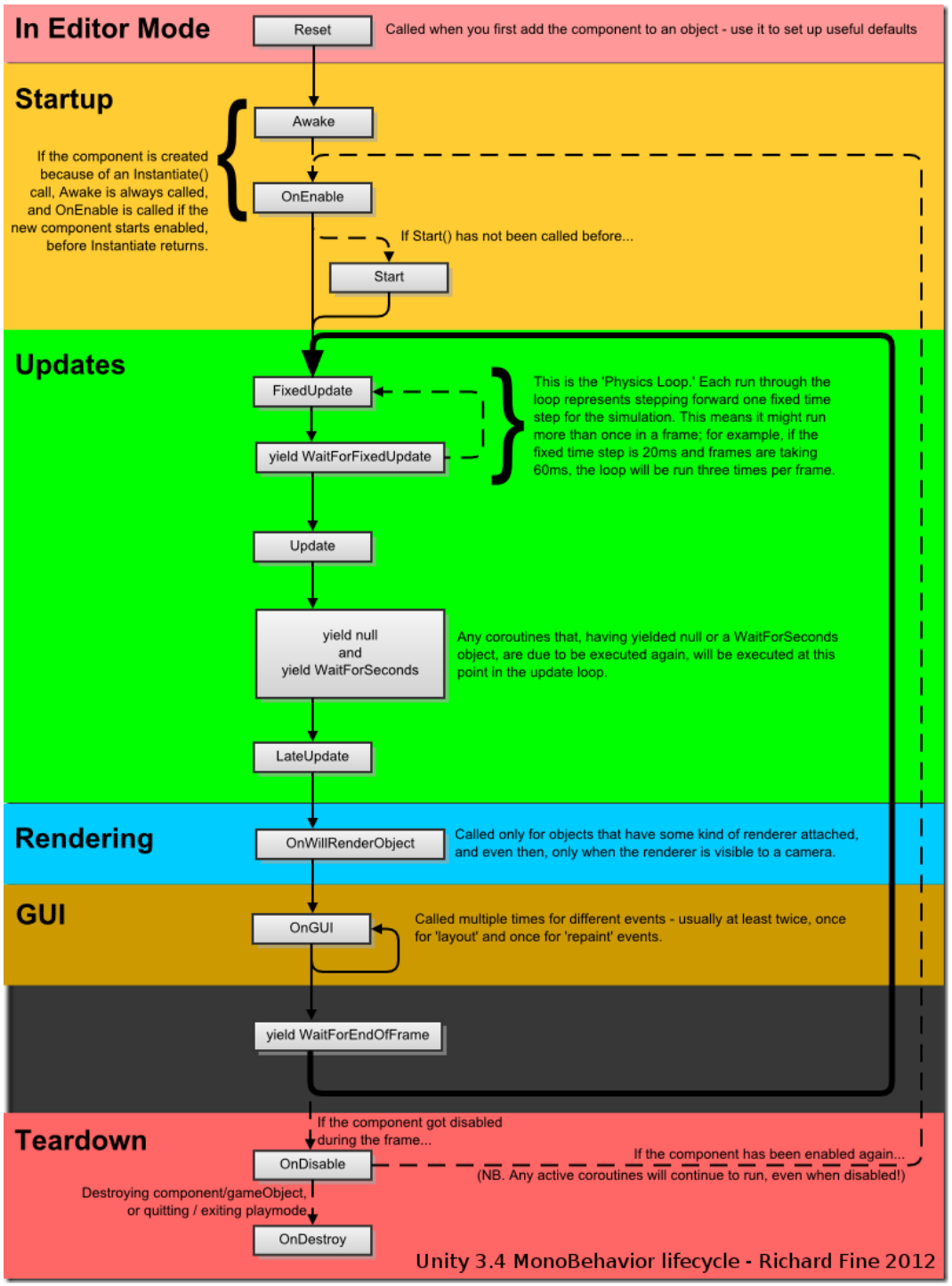


Figure 4.2: monobehaviour's lifecyle

main while block that sequentially calls all subsystem functionality to be executed in a single frame. This would also include the execution of all component-specific update functions for that exact frame rate. It's clear that without a proper structure and optimization the result would be quite unpleasant, even after adopting the fixed time step & variable rendering practice[13].

A more optimized solution can be earn by exploiting caching. The approach is to store all game objects inside a sequential array, so that calls to the components update function can be made following the memory linear traversal[14]. The idea here is that when you retrieve something from the RAM the likelyhood of requiring to fetch something nearby is high, so the data in that area is grabbed all at once. Of course this approach would have some impact on the complexity of game object deletetion, but it's often unnoticeable.

Another solution would be exploiting parallelism, in one multiprocessor game loop architecture. A way to take advantage of parallel hardware architecture is to divide up the work that is done by the game engine into multiple small, relatively independent jobs. A job is best thought of as a pairing between a chunk of data and a bit of code that operates on that data. When a job is ready to be run, it is placed on a queue, to be picked up and worked on by the next available processing unit. This can help maximize processor utilization, while providing the main game loop with improved flexibility[15].

Structure 4.1 A naive approach for building a game object. Note that the code isn't optimized to fully exploit caching.

```
class GameObject
{
public:
  Component *GetComponent( id );
  void AddComponent( Component *comp );
  bool HasComponent( id );

private:
  std::vector<Component *> m_components;
};
```

Structure 4.2 The naive approach for a system update would be to pass a list of game objects like so.

```
void Engine::Update( float dt )
{
  for(unsigned i = 0; i < m_systems.size( ); ++i)
    m_systems[i].Update( dt, ObjectFactory->GetObjectList( ) );
}
```

4.4 Vuforia support

Vuforia is a software stack for building AR applications with a large set of carefully encapsulated advanced computer vision features. Vuforia's recognition and tracking capabilities can be used on a variety of images and objects, like: single marker, multi-markers, cylinder targets, text and objects.

Vuforia provides tools for creating targets, managing target databases and securing application licenses. The Vuforia Object Scanner (available for Android) helps developers to easily scan 3D objects into a target format that is compatible with the Vuforia Engine. The Target Manager is a web app available on the developer portal that allows you to create databases of targets to use locally on the device, or in cloud. Developers building apps for optical see-through digital eyewear can make use of the Calibration Assistant which enables end-users to create personalized profiles that suit their unique facial geometry. The Vuforia Engine can then use this profile to ensure that content is rendered in the right position.

The Vuforia Extension for Unity comes as a simple unitypackage and allows developers to create AR applications and games easily using the Unity game engine. Installing the extension is just a matter of extracting the package inside the project and setup the provided prefabs into the scene. We'll briefly explain the basic steps on how to setup a simple Unity project.

License Key The first thing to do before starting using Vuforia, is to obtain a license key for the application to be used inside the project. The license key can be easily created using the developers portal, after subscribing as a developer.

Adding Targets In Vuforia, objects that can be identified by the computer vision system are called *targets*. For target binding, the extension provides a vast number of prefabs ready to be used inside the scene. Before starting using this components, however, we need to tell the scripts how these targets can be detected. We need to add a *Device Database* to our project, this can be done by

either creating a new database or using an existing one. To create a new database we need to use Vuforia Target Manager. After that, we just double-click on the downloaded package to import it into the project.

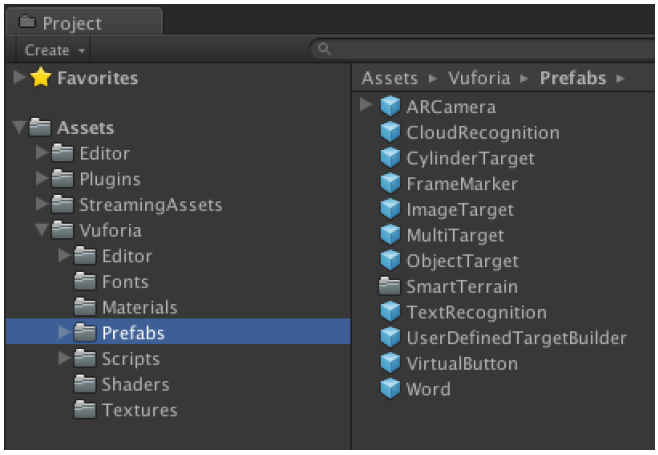


Figure 4.3: Assets imported into the project by Vuforia. ImageTarget, ObjectTarget, MultiTarget are all prefabs implementing a behaviour that automatically handles the recognition and binding of one target.

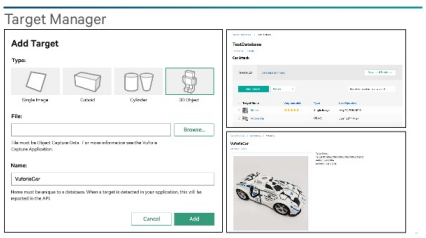
Add AR assets and prefabs to scene Now that we have imported the Vuforia AR Extension for Unity, we can easily adapt our project to use augmented reality. First of all we need to delete, or disable, the scene Main Camera, replacing it with ARCamera from the Prefabs folder instead. This object is responsible for rendering the camera image in the background and manipulating scene objects to react to tracking data. Remember that this object needs to be configured with a legit license key. The Database Load Behaviour script also needs the list of DataSets to be loaded when the application starts.

We now drag an instance of ImageTarget into the scene, the object that represents the marker inside the scene. By looking at the Inspector we see that the object has an ImageTargetBehaviour attached, with a property named DataSet. This property contains a drop-down list of all available Data Sets for this project. When a Data Set is selected, the Image Target property drop-down is filled with a list of the targets available in that Data Set. We can now select the DataSet and Image Target from StreamigAssets/QCAR, these are the same target we've generated in the TargetManager.

Add 3D objects to scene and attach to trackables We can now bind 3D content to our Image Target, we just need to place it as a child object of our

ImageTarget by dragging it on top of the parent inside the Hierarchy window. We can now test the application, the results should show the 3D content under the ImageTarget bound to the physical marker.

(a) An Image Target



(b) Vuforia Target Manager

Figure 4.4: After an image with a good entropy is added to the dataset using the manager, it can be easily used as an image target.

More informations on how to setup the Vuforia extension for Unity can be found on the website in the developer's library[16].

4.5 The High Level API (HLAPI)

There are tons of different ways for dealing with networking in Unity. For example one is free to choose between different legacy and currently holding networking systems, like Unet, the Low Levl API and others, while still being able to handle .Net sockets directly. However the problem of portability persist, and while working with Unity it is advised to use always engine specific functions, since they are already part of the application lifecycle.

The High Level API (HLAPI)[17] is a the current standard for developing multiplayer games. It was introduced with Unity 4 promising to make the code regarding networking more maintainable, compared to the previous Unet system.

It uses the lower transport layer for real-time communication, and handles many of the common tasks that are required for multiplayer games. While the transport layer supports any kind of network topology, the HLAPI is a server authoritative system; although it allows one of the participants to be a client and the server at the same time, so no dedicated server process is required.

The HLAPI allows developers to:

- Control the networked state of the game using a "Network Manager".

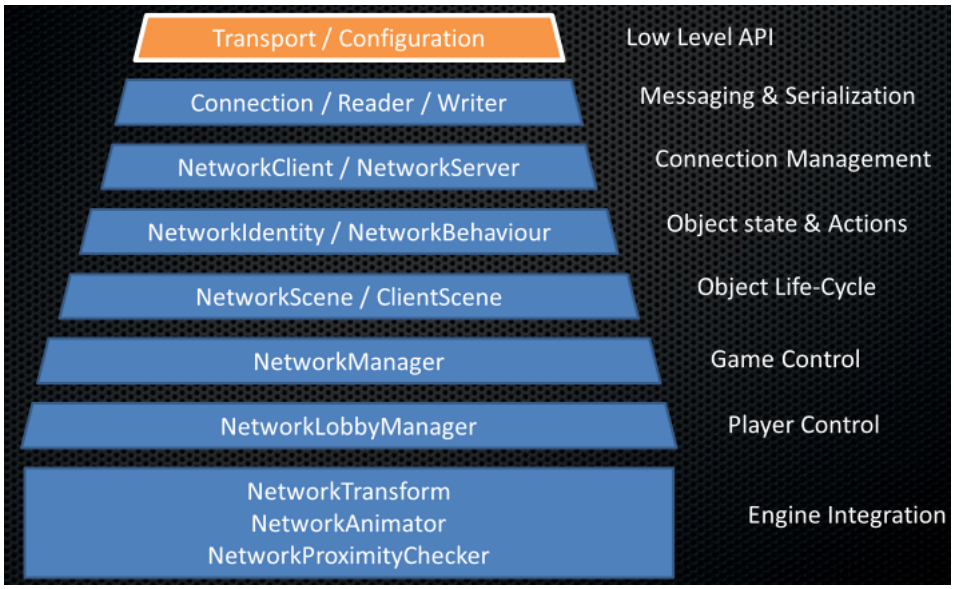


Figure 4.5: The Image shows the various layers of functionality of HLAPI, using as a base the Low Level API. Each layer is either a class, component or GameObject provided by Unity.

- Operate "client hosted" games, where the host is also a player client.
- Serialize data using a general-purpose serializer.
- Send and receive network messages.
- Send networked commands from clients to servers.
- Make remote procedure calls (RPCs) from servers to clients.
- Send networked events from servers to clients.

HLAPI uses functions that are embedded in the engine, it also provides ad-hoc editor extensions in order to ease the correct configuration of the scene. Part of the basic set of components and functionalities provided by the system are:

- A NetworkIdentity needed to give an unique reference to the object through the network.
- A NetworkBehaviour for writing networked scripts exposing:
  - An automatic synchronization mechanism for script variables.
  - Ways of performing remote procedures calls.

  - Message sending capabilities.

- A Configurable automatic synchronization of object transforms, provided by the NetworkTransform component.
- Support for placing networked objects into the Unity scenes.

Spawning entities In Unity, GameObject.Instantiate creates new Unity game objects. But with the networking system, objects must also be "spawned" to be active on the network. This can only be done on the server, and causes the objects to be created on connected clients. Once objects are spawned, the Spawning System uses distributed object life-cycle management and state-synchronization principles.

Players, Local Players and Authority In this networking system, player objects are special. There is a player object associated with each person playing the game, and commands are routed to that object. A person cannot invoke a command on another person's player object, but only on their own. So there is a concept of "my" player object. So, in contrast of common single player games, the "local player" object must first be instantiated by the server, and then added into our scene locally.

State Synchronization State Synchronization is done from the Server to Remote Clients. Data is not synchronized from remote clients to the server, this is a job for Commands.

There are two different ways to stream data from server to clients, first is an automatic process through the concept of SyncVars, the other is by writing custom serialization and deserialization callbacks.

SyncVars are member variables of NetworkBehaviour scripts that are synchronized from the server to clients. When an object is spawned, or a new player joins a game in progress, they are sent the latest state of all SyncVars on networked objects that are visible to them. Member variables are made into SyncVars by using the [SyncVar] custom attribute.

SyncVars can be basic types such as integers, strings and floats. They can also be Unity types such as Vector3 and user-defined structs. SycnVar updates are sent automatically by the server when the value of the variable changes, so there is no need to perform any manual dirtying of fields for SyncVars.

Remote Actions SynchVars are a way to stream data from the server to his clients. HLAPI also offers a way to send specific message from client to server and vice versa . These type of actions are sometimes called Remote Procedure Calls.

There are two types of RPCs in the networking system: Commands, which are called from the client and run on the server and ClientRpc calls, which are called on the server and run on clients.

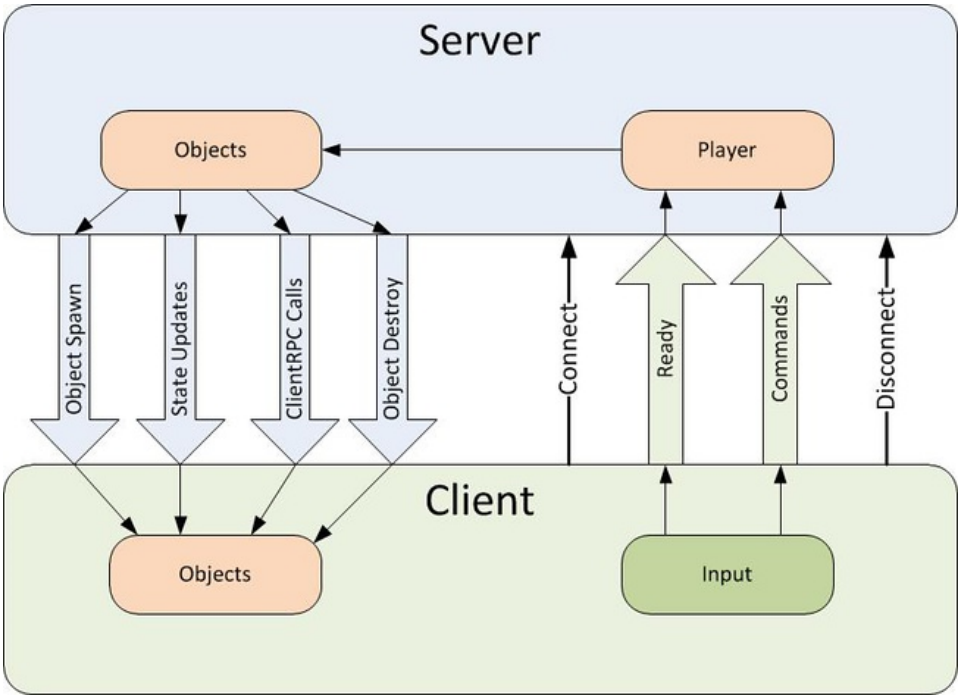


Figure 4.6: The diagram shows different types of interactions supported by HLAPI. Inside a NetworkBehaviour script the developer can use all provided attributes, as long the right conditions are met.

The arguments passed to commands and ClientRpc calls are serialized and sent over the network. These arguments can be:

- basic types (byte, int, float, string, UInt64, etc)
- arrays of basic types
- structs containing allowable types
- built-in unity math types (Vector3, Quaternion, etc)
- NetworkIdentity
- NetworkInstanceId
- NetworkHash128

- GameObject with a NetworkIdentity component attached

Arguments to remote actions cannot be subcomponents of GameObjects, such as script instances or Transforms. They can't be any other unserializable types.

4.6 Editor extension

Unity lets developers extend the editor with their own custom views and inspectors. It is possible, in fact, to create complex editor windows usable to automatize the creation and management of in-scene objects; or to use property drawers in order to define how properties are visualized for all instances of a particular script.

Property Drawers have two uses:

- Customize the GUI of every instance of a Serializable class. This can be done by attaching a new class that extends PropertyDrawer to a Serializable class by using the CustomPropertyDrawer attribute and pass in the type of the Serializable class that this drawer is for.

- Customize the GUI of script members with custom Property Attributes. This can be used to limit the range of values of a specific attribute of a component, or simply for changing how it is displayed inside the inspector. This can be done by writing a class placed inside the Editor that has the CustomPropertyDrawer attribute to which it's specified the name of the attribute to render.

By extending the editor it is not only possible to shape how custom components are rendered and used by the developer in the Inspector, but we're also able to write code that is directly executed inside the editor while defining the scene.

It is not hard to extend the editor main menu, adding new items and create custom windows. Making a custom Editor Window involves the following simple steps:

- Create a script that derives from EditorWindow.

- Use code to trigger the window to display itself.

- Implement the GUI code for drawing the content.

Scripts extending the editor needs to be placed under a subfolder of Assets named *Editor*. These scripts are then automatically executed during the editor lifecycle at the right moment, there's no need to compile them by hand, everything is handled automatically by the editor's runtime.

Example 4.3 Example of a custom window in C#

```
using UnityEngine;
using UnityEditor;
using System.Collections;

class MyWindow : EditorWindow {
    //The attribute tells the editor to place
    //the window in the top menu under the
    //location "Window" with the name "My Window"
    [MenuItem ("Window/My Window")]

    public static void ShowWindow () {
        EditorWindow.GetWindow(typeof(MyWindow));
    }

    void OnGUI () {
        // The actual window code goes here
    }
}
```

More information about how Unity editor can be extended can be found in the documentation[18].

Chapter 5

Development

In this chapter we're discussing the development of the AW framework, specifying the implementation of concepts previously exposed in the design section.

The objective is not to write a software stack that implements all Augmented World functionality from the ground up, but instead to use different technologies to address most complex requirements by binding them together, in order to have a taste of simple AW framework for writing 3D Augmented World programs.

Choosing Unity as a 3D real-time engine we have the advantage of a tool that encapsulates most of the core functions required by Augmented World programs, providing ready to use means for developing graphics, geometry of the environment and handling real-time behaviours.

Following we'll see how to exploit Unity advanced features to develop a framework on top of the engine that also extends the editor. We'll discuss the implementation of all basic AW framework concepts exposed in the previous chapters.

5.1 Hologram

Let's take a look at the framework's implementation of the Hologram concept. Since Unity is a component-based engine, it's only natural for the Hologram to be in fact an extension of the MonoBehaviour class.

We present *HologramComponent*, the concrete implementation of an Hologram, shaped by the sum of his parts. Its usage is based on top of the basic aspect of the component pattern. The functionality of an Hologram is sparse between the three main concern exposed in the previous chapter; these objects, referred by the component, might as well not be present or not implementing a specific function for that Hologram, simply because there might be no need.

In this perspective, the View, Model and NetworkSynch parts of the Hologram all become Unity components as well, having the advantage to be all accessible

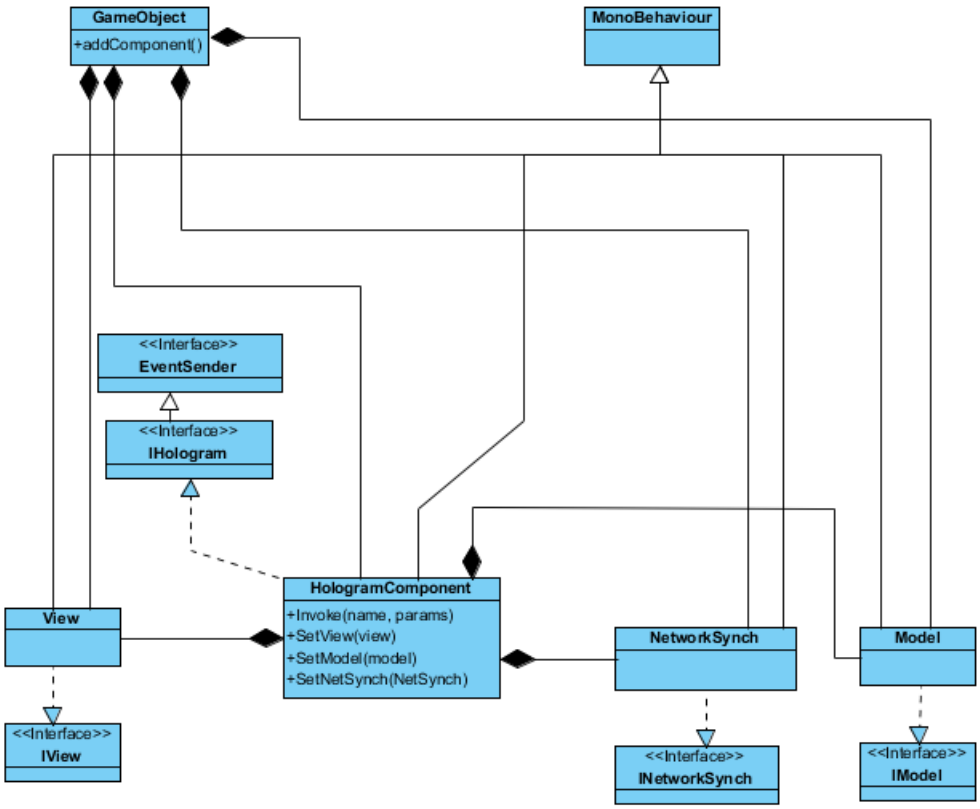


Figure 5.1: Class diagram of HologramComponent. Every module of Hologram is a Unity component as well.

from the same GameObject. The Hologram functionalities are shaped in a Unity-like fashion, the class isn't meant to be extended, its behavior changes by the effect of his parts.

In practice we think that by separating the functionality of an Hologram into different components assigned to the same GameObject we have two main advantages.

The first one is that it's possible to assign this entities trough the inspector and eventually handle its inspector-specific visualization trough an editor extension. This makes easier to the developer to track mistakes in the setup of the Hologram, in a more dynamic experience compared to the standard error message visualized in the debug console. Moreover it's possible to have a major flexibility in writing specific behaviour-based scripts that can be parameterized from the editor.

The second advantage is that there's no need for the developer to write structural specific code. Everything is handled in a more dynamic way, since there's no direct link between calls from an HologramCompoent to his parts.

In this Unity-based framework, the Hologram becomes a well defined GameObject to witch are assigned the View, Model and NetworkSynchronization components. Other legacy scrips might then be used to extend its functionality. For example, a mesh, collider and material component must be assigned to the GameObject and then be properly handled by the View if we want it to be rendered on screen.

The HologramComponent is meant to be the main point of interaction with this specific Augmented Entity, its parts shouldn't be accessed directly. When an HoloDoer, or some other entity, interacts with the Hologram, it is meant to be done directly through the *Invoke* method. This method propagates the call to the respective View, Model and NetSynch portions of the Hologram by checking out if the specified method is implemented on each subcomponent using reflection. This allows the developer to separate the payload of a method between the three main concerns, making the coding of an Hologram much cleaner and still quite flexible. Moreover this conceptual separation is critical if we want to build a framework that can work with different technologies, as said in the previous chapters.

An important aspect of an Hologram is that it is constantly rendered on top of the user perceived vision of the real world. This is intrinsically handled behind the scene by the chosen AR system. All HologramComponents are rendered applying the transformation matrix that is fetched directly by recognizing points of interest from the device video feed. The coordinates onto which the binding happens are specified by the *_world* GameObject's Transform, that is always used as a link from the virtual to the physical world. All HologramComponents should then be child of this GameObject, making their position relative to the physical point where space coupling happens.

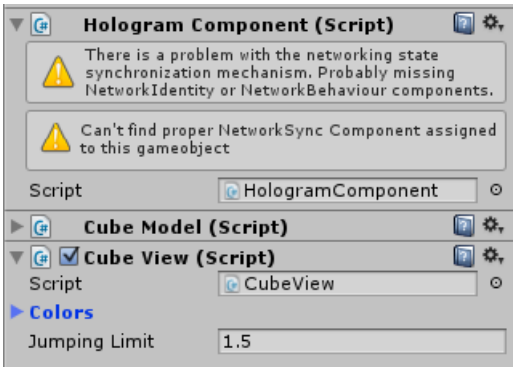


Figure 5.2: The image shows an error in the configuration of this Hologram regarding the HLAPI networking system.

Since this component might not be so much straight forward to use, and requires the copresence of his parts to work properly, an ad-hoc editor extension

comes to aid. The Inspector view of HologramComponent informs the developer about the correct configuration of that single augmented entity, in agreement to the currently holding framework specifics. For example, if Vuforia is the used AR technology, a message informing that the GameObject should have a parent with a TargetBehaviour is shown if that's not the case.

5.1.1 View

The View Component is meant to handle all aspects of an Hologram regarding his graphics. There's no strong requirement for this component, besides implementing the *IView* interface, that makes him recognizable by the HologramComponent.

What this script should be used for is to write essential graphic handling logic. Here we can rightfully find access to specific legacy components like: materials, mesh, animations, and so on. It should expose methods that change the Hologram visualization in some way and can make use of the standard Update method to perform some time-based animation.

Here is a list of legacy components that makes sense to handle inside the a View component:

- Any Mesh Renderer
- RigidBody
- Any Colldier
- Particle Effects
- Animation
- Materials

If the user needs to interact with this Hologram in any way, this component can also be used to catch input from the device (like the screen gestures, key pressing, etc), directly from the Update method. However for a correct separation it is advised to use a different component entirely, we'll argue about this in later chapters.

This component is of course tight to Unity graphics and it is not mean to be reusable outside the Engine.

5.1.2 Model

In its most pure form, an Hologram has memory, meaning it holds a state. This state needs to be consistent, accessed and updated, like any standard plain old object.

We encapsulate this simple concern inside a specific component, called Model. The Model holds data regarding the state of an Hologram, exposing methods that alter its state.

This object holds all structural information about an Hologram. Its instance has sense to be located only on the server process, since Hologram specific informations are private and centralized. This doesn't mean that redundancy can't be used to exploit this component for handling synchronization errors on clients, improving fault tolerance in regard to faulty networks.

This script needs to extend MonoBehaviour and implement *IModel*, however this is only necessary because we want the object to be treated like an Unity component, so that it can appear inside the Inspector. A Model component is meant to be used like a standard OOP object, MonoBehaviour functions, especially Update, need to be ignored. This way, by removing MonoBehaviour from the class declaration, the object can also be used in other environments not bound to the Unity engine.

In brief, this object is used by this framework as a server-side data space for a specific Hologram. The Model isn't accessed directly form other scripts, his state is regulated by calls performed by its HologramComponent.

5.1.3 Network Synchronization

One major concern about Holograms regard their synchronization mechanism. In our envisioned network architecture, one central node holds the main instance of the world, from which others update their view state.

This main instance is what really has the authority on all Hologram related data and functions. Clients can request for a specific action be executed on a Hologram by sending particular interaction messages, that we call *commands*. Client-side Holograms are continuously notified about updates to their remote counterpart, so that they can quickly react and update their visualization accordingly.

The behaviour surrounding message passing and data transfer between clients and server is encapsulated in a framework specific component. This component is the Network Synchronization one.

The Network Synchronization component is tight to the specific type of Networking System chosen for the application, whose functions are provided by some sort of Network Manager. This means that in the future, if the provided Networking System doesn't sweets the developer needs, it can be swapped out by only

affecting the code regarding NetSynch components.

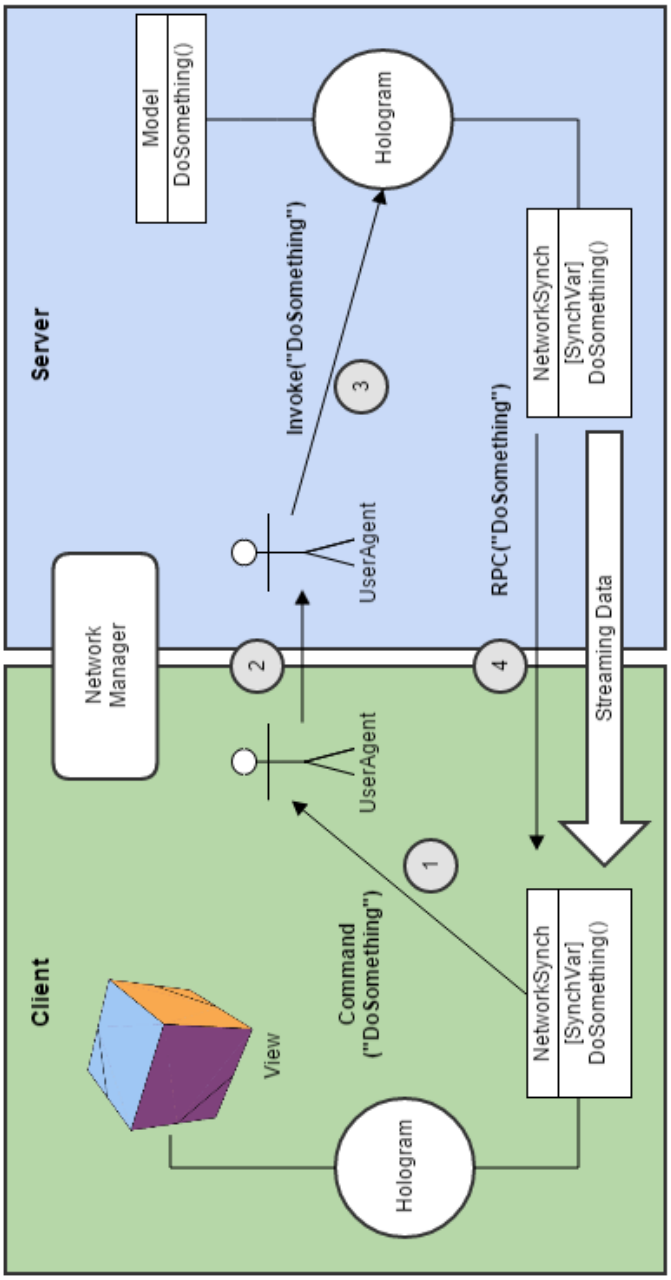


Figure 5.3: Bird's-eye view of client-server interaction in the framework. The figure shows the two main functions of the Network Synchronization component. The first one is the streaming of component specific data from the server to all clients by means of a SynchVar. The second one is the round trip of a command, that starts from a specific client instance and is eventually propagated to others by the server if it's necessary.

Currently the AW Framework only supports the Unity based High Level API for Networking. We choose to start from this Networking System because it's embedded in latest versions of Unity and already addresses most of the networking problems related to real-time distributed applications.

Being HLAPI on top of the real-time communication layer, it provides some standards that needs to be followed in writing the remote communication code. The following actions needs to be taken care of before writing the Network Synchronization component using HLAPI:

- Assign a Network Identity to the Hologram.
- Ensure that this GameObject is saved into a prefab so it can be instantiated at runtime.
- Create a component extending HLAPINetworkSynch abstract class.

HLAPINetworkSynch is the base class implementing functions that make use of the AWNetworkManager. It requires the import of the UnityEngine.Networking library, since it derives from the engine's NetworkBehaviour class. NetworkBehaviours are special scripts that work with objects having a NetworkIdentity component, these scripts are able to perform HLAPI functions such as Commands, ClientRPCs, SyncEvents and SyncVars.

Using HLAPI can make the process of writing the code for view-model synchronization quite easy, since the coordination mechanism for that Hologram is all limited to the same component. This means that both server and client side of the code are placed inside the same script. This is the standard when dealing with NetworkBehaviours.

Inside a script that derives from HLAPINetworkSynch we can make use of different synchronization mechanisms. Being this an extension of a NetworkBehaviour we can use SynchVar to dynamically update the state of a variable from server to clients. We can also make use of standard message passing, on top of which it is build the SendCmd function, used to remotely invoke methods on the specific server-side Hologram. Moreover, the attribute [ClientRPC] can be used in order to remotely invoke a method from server to all clients.

HLAPINetworkSych registers itself for remote messages from AWNetworkManager automatically as soon as a remote connection is established.

The method SendCmd() is used to remotely invoke a method on the same Hologram. The arguments of the call are automatically serialized in a stream of bytes, but they require to be basic types, implement *Serializable* or be built-it Unity math types (like Vector3, Quaternion, etc).

It is possible to use the method AskCurrentStateMessage() to receive specific information on the remote state of that Hologram. For this mechanism to work,

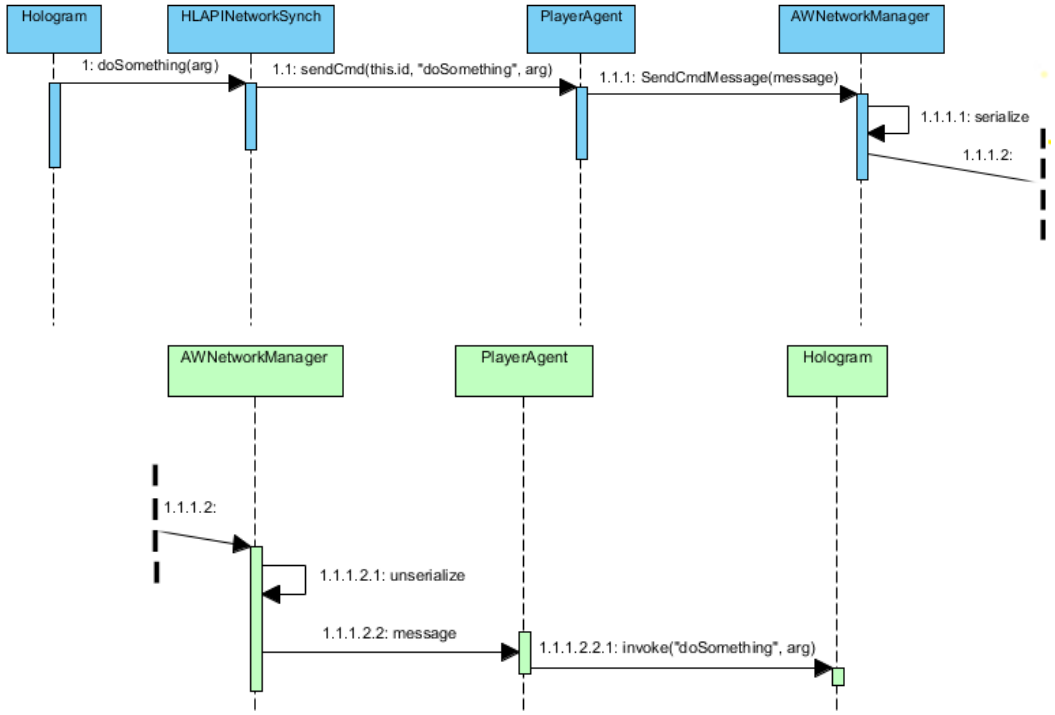


Figure 5.4: The figure shows the process behind the SendCmd call. The message is sent through the network as a bytestream by the AWNetworkManager. The message can't reach the Hologram directly, it has to pass trough PlayerAgent, of witch the client has the authority.

the developer must implement the abstract methods GetCurrentState and OnCurrentStateReceived.

GetCurrentState() is meant to generate a message holding a snapshot about the Hologram state of affairs. It returns a StateMessage that holds data with a key-value specification. OnCurrentStateReceived is the handler of the message, so it is meant to fetch data from the StateMessage and update the Hologram's view accordingly.

This mechanism is useful when a client connects to the server and needs to fetch all meaningful Hologram information at once. By doing this, its view can be updated with the remote Hologram's data, and the synchronization process can continue from that point forward. AskCurrentStateMessage() isn't meant to be used repeatedly, for a continuous update of the Hologram's representation it's advised for SynchVars to be used instead.

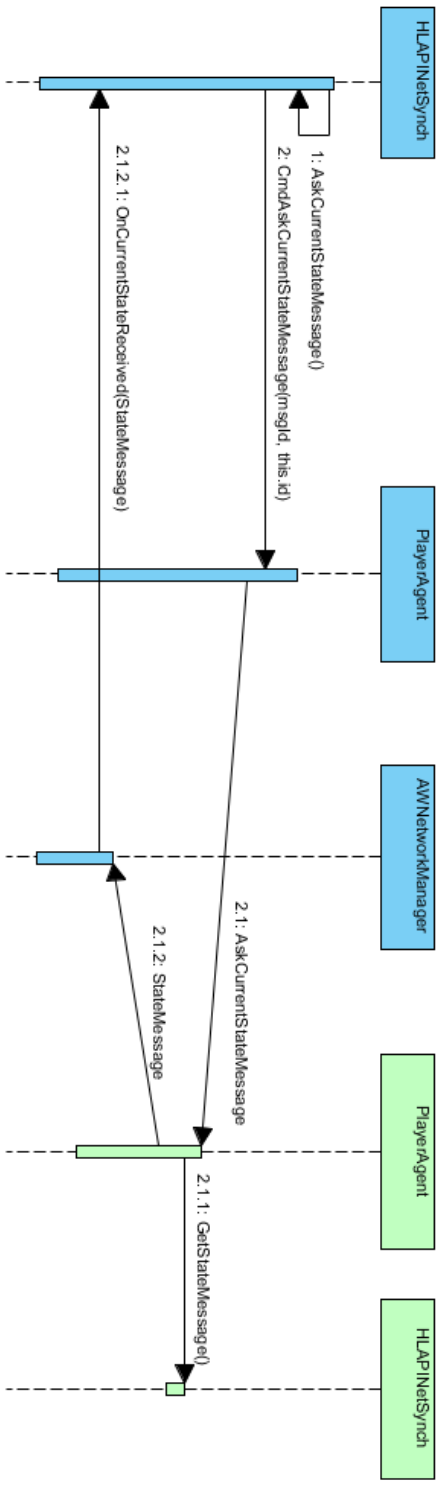


Figure 5.5: Interaction diagram for the AskCurrentStateMessage() call. Messages are serialized and unserialized into objects on both parts.

5.2 User shape and interaction

The User needs to be part of the model of the world. There are various reasons for this, already addressed in the previous chapters. First of all, autonomous entities might reason about the position and state of users. Secondly we need a client specific means of interaction, especially if we want to use HLAPI.

When a clients joins the scene, a special GameObject is instantiated at runtime by the AW Framework Manager. This GameObject is bound to a specific client instance, that owns the *authority* for that object. This means that the application can send Commands to that particular entity alone. By using Commands we can trigger server side execution of code for scripts deriving from Network Behaviour of which the client owns the authority.

For security reasons the User Agent is the only one GameObject a client can own the authority. The structure of this entity needs to be known by the AW Network Manager, so that it can be automatically instantiated and destroyed at runtime.

The framework already provides a standard GameObject inside the assets folder to be used for development and testing. This prefab is really basic, it only holds the required PlayerCommands script without any other specific component, besides the Network Identity one.

PlayerCommands expose the method Interact that can be used to remotely invoke a method of the specified Hologram, enabling remote interaction. This method doesn't directly use the attribute [Command], but instead it simulates its behaviour trough the use of an Interaction Message. This was necessary in order to enable the developer to extend the set of supported data types that can be used as an argument.

Example 5.1 PlayerCommands script - exposing the Interaction method.

```
/// <summary>
/// Interact with the passed object calling
//// the specified method name.
/// </summary>
/// <param name="go">GameObject reference
///     with NetworkIdentity.</param>
/// <param name="method">Method name.</param>
/// <param name="args">Serializable parameters</param>
public void Interact (GameObject go, string method,
                                    params object[] args){
```

```
    if (isClient) {
        netManager
            .SendInteractionMessage (go,
                method, args);
    } else if (isServer) {
        OnGameObjectInteraction (go, method, args);
    }
}

[Command]
void CmdInteract (GameObject go, string method)
{
    HologramComponent hc =
        go.GetComponent<HologramComponent> ();
    if (hc != null) {
        hc.Invoke (method);
    } else {
        Debug.LogWarning
            ("Can't find HologramComponent for "
            + go.name
            + " using SendMessage insetead");
        go.SendMessage (method);
    }
}

public static void OnGameObjectInteraction (GameObject go,
 string method, object[] args) {
    HologramComponent hc =
        go.GetComponent<HologramComponent> ();
    if (hc != null) {
        hc.Invoke (method, args);
    } else {
        if (args != null && args.Length > 0) {
            go.SendMessage (method, args [0]);
        } else {
            go.SendMessage (method);
        }
    }
}
```

Example 5.2 Message handling inside the AW Network Manager

```
//Message Handling
public const short INTERACTION_MSG_ID = 888;

public void SendInteractionMessage (GameObject go,
     string method, object[] args){
    NetworkIdentity ni =
        go.GetComponent<NetworkIdentity> ();
    if (ni == null) {
        Log ("No network Identity found on the object");
        return;
    }
    uint netId = ni.netId.Value;
    InteractionMessage msg =
         new InteractionMessage (netId,
             method, args);
    if (!client.Send (INTERACTION_MSG_ID, msg)) {
        Log ("Problem while sending msg:\n"
             + msg.ToString ());
    }
}

//Callback (Executed locally on the server)
public void OnGameObjectInteraction (NetworkMessage netMsg)
{
    InteractionMessage msg =
        netMsg.ReadMessage<InteractionMessage> ();
    Debug.Log ("Message received " + msg.ToString ());
    NetworkInstanceId netId =
         new NetworkInstanceId (msg.netId);
    GameObject go = NetworkServer.FindLocalObject (netId);
    PlayerCommands
        .OnGameObjectInteraction (go,
             msg.method, msg.args);
}
```

PlayerCommands is a required component to be assigned to the User Agent for the HLAPI networking system to work properly. Besides Interaction, it provides methods for requesting object spawning and destruction.

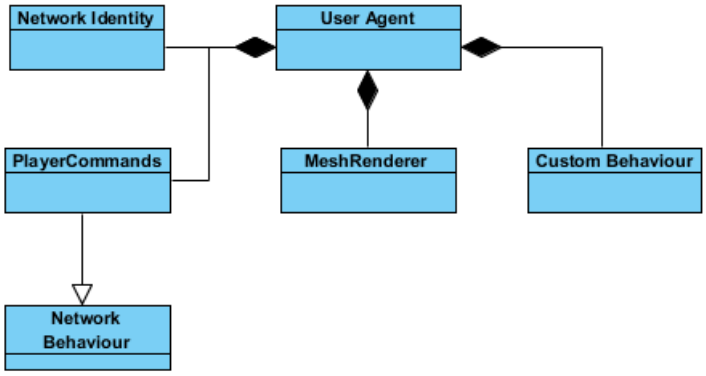


Figure 5.6: Roughly, an example of a custom prefab for an User Agent.

In its most basic form, the user doesn't have a geometry, it's merely an object with a position, to which we send commands. However this GameObeject can be extended freely by the developer through the use of composition. Specific applications might require the user to be rendered on screen, or to perform dynamic actions. This can easily be done by writing custom components.

5.3 Mono Event System

Inspired by event-driven programming, we want to have separation between a specific action happening inside our world, and the objects that are directly affected.

For example, an Hologram might perform an action that changes the state of the world in some way. The effects of this action might have an impact on the execution of other augmented entities, such as autonomous ones like HoloDoers. This effects might as well regard the whole scene, or just be limited to a small set entities in a particular geometric area.

For this reason Holograms are able to generate events, messages transmitted into the environment, that might be or be not perceived by other entities.

The AW Framework provides his own tools for event creation and handling, that is directly based on MonoBehaviour, making use of the implicit main loop.

A MonoEventContext is an entity where objects implementing the *IEventListener* interface can register to perceive context related events. In this sense, a context simply is a set of entities related by some logical criteria.

When we extend MonoEventContext providing some kind of dynamism, we can create event context that are related to particular geometrical properties. For example we can create a context where Hologram entering in a specific area can

automatically register.

A general context for the whole scene is ensured to be always present at runtime and can be accessed through the instance of AWConfig.

Being the event context based on MonoBehaviour, it uses the *FixedUpdate* method to send messages enqueued to his listeners at a fixed rate. These messages are then received by the entities that can trigger a behaviour or as well do nothing.

BaseEvent is the primary class for events dispatched by the MonoEventContext. In its basic form, it only provides reference to the sender. This class can be extended alike, for example including the position of the sender in a Vector3, so that the receiver can also reason on the distance from the object before handling the event.

5.4 HoloDoer

The digital world of augmented objects is, like our own, dynamic. The main difference is that in an Augmented World autonomous, possibly smart, entities work for us providing some sort of aid to users. This autonomous entities are called HoloDoer, marking the fact that are meant to handle Holograms.

Like Hologram, the HoloDoer concept is bound to be an extension of MonoBehaviour, enabling the developer the various advantages already discussed. Only this time we don't hide the exposed functions of MonoBehaviour like we did with the Hologram. The developer has all rights to use all functionality tight to the Unity engine life cycle.

The HoloDoerComponent class can be extended and used to handle one or more Holograms. Being it an EventListener, it can register to an EventContext for reacting to Hologram generated events through the use of a specific handler function.

The behaviour of an HoloDoer is specified in a Unity-like fashion, meaning that it is the result of the composition of more scripts. This entity can also act over time, making use of the *Update* or *FixedUpdate* functions. This means that its dynamic behaviour is the result of incremental changes gone trough the these methods.

In the end, using an extension of HoloDoerComponent alone, or the composition of more scripts, we can create fully autonomous entities.

Let's reason about how to have access to Holograms inside an HoloDoerComponent script. The most common way to access an Hologram in Unity would be trough a public variable set inside the inspector directly from the Editor. HoloDoers might find even more flexible the implementation of a perceive, then act pattern, where they can automatically get the reference of Holograms they are interested to, and then use them by performing an Invoke() call. Of course the

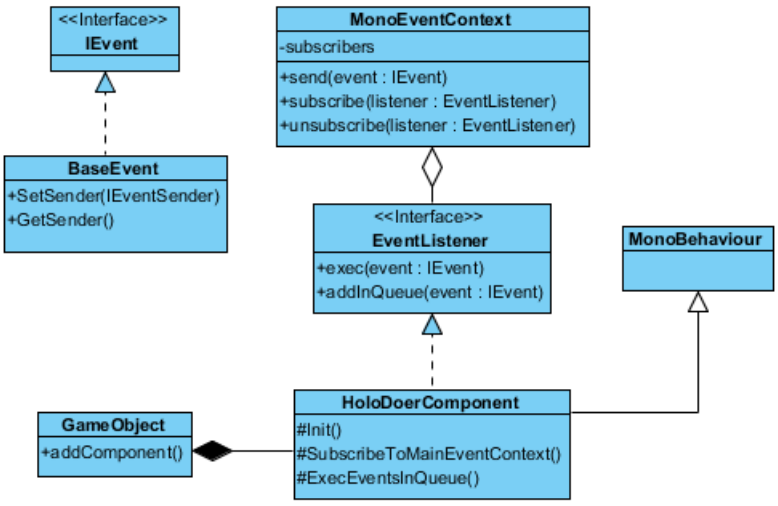


Figure 5.7: The class diagram shows the structure of an HoloDoerComponent, including the aggregation of EventListeners starting from a MonoEventContext.

HologramComponent class has no means to differentiate different types of Holograms, so GameObject tags should be used instead.

5.5 Tracking Area

A Tracking Area is a GameObject composed by the following parts:

- One or more trigger colliders.
- A TrackingAreaBehaviour component.

The job of this object is to track of all meaningful entities inside a specific geometric area, specified by the mesh assigned to the object collider. TrackingAreaBehaviour is in fact an extension of MonoEventContext, meaning objects and Holograms can subscribe to that context, and send events to all listeners in the area.

Moreover this component automatically generates events when an Hologram enter or leaves the area, specifically *OnTrackingAreaEnterEvent* and *OnTrackingAreaExitEvent*. This events can then be handled by HoloDoers in order to update the state of the world by any means.

The Tracking Area is provided to the developer by the framework and can be found inside the Prafabs folder. This ready to use GameObject can then be placed

inside the scene, all that is left to do is to define the boundary of the collider from scene view or within the Inspector.

The script EventContextCollector can be attached to Hologram and entities alike to provide an automatic mechanism that collects all tracking areas the object is into. Events can then be dispatched to all collected contexts using the Send() method of this component.

5.6 Editor functionality

Editor scripts are implemented to help the developer through the setup and configuration of the framework's parts. Essentially, these scripts are executed inside the editor life cycle and are able to reason upon the currently state of the scene, accessing directly to the Hierarchy.

Two are the main functionality provided by the Augmented World framework editor extension.

The Framework Configuration Window Accessible under Window >AW Framework Config, this view is what triggers the automatic configuration of the scene in accord to the selected properties.

At the moment, two are the main properties that influences the auto-configuration process. The first is the selection of the Augmented Reality system, if Vuforia is select, then the user is notified if some of the basic Vuforia prefabs, like ARCamera, are missing from the scene. The second regards the Networking System, for HLAPI the procedure automatically checks and place into the scene the AWNetworkManager configured to work with the Unity provided Networking System.

The HologramComponent Inspector View The standard component interface in the Inspector for the HologramComponent is overwritten by an ad-hoc editor script. This script makes use of Unity editor GUI to display messages that can help the developer trough the setup process of an Hologram, in regard to a specific AW framework configuration.

This is possible because this script already holds the reference of the component instantiated inside the editor. It is then possible to gain reference of the respective GameObject and reason on his properties.

Following are some alerts provided in the Inspector view of HologramComponent.

- A message is displayed if one of the View, Model or NetSynch component is missing.

- If Vuforia is used, a message informs when the Hologram is not a child of a Target Behaviour.
- If HLAPI is used, a message is displayed if the NetworkTransform or NetworkIdentity components are missing.
- If HLAPI is used, a warning is shown if the object isn't associated to any prefab.

5.7 The problem of serialization

To bypass some of the limitation regarding the usage of the HLAPI Command functionality, the framework uses his own bytestream serializer.

One big main annoyance of using HLAPI Commands was that it was impossible to serialize arguments that didn't fall in the specifics list of objects provided by Unity, being them: basic types, arrays of basic types, Unity math types, GameObjects with NetworkIdentity and some others.

To solve this issue, the framework uses it's own implementation of a command, making use of HLAPI Network Messages. Each time a command is sent from a client, it's specification and arguments are packed inside a InteractionMessage. Internally this object extends the provided abstract class MessageBase, it then needs to implement the methods Serialize and Deserialize, that specify how the message data should be converted to and from the bytestream.

We are now relying on the standard C# serialization process, that can automatically convert simple objects flagged with the Serializable attribute. However by doing this we are left with a big problem, the fact that all Unity basic types like Vector3, Quaternion, Color, etc are not flagged as Serializable. For this reason an extendible class called BinaryDataFormatter is provided to handle such cases.

BinaryDataFormatter uses a .NET BinaryFormatter in pair with a set of Surrogates, that specify how special not Serializable types should be converted into bytes and vice versa. For more information on how to use .NET formatters, surrogates and selector, check the documentation[19].

Chapter 6

Using the framework

In the previous chapters we found out about all tools provided by the AW Framework. Here we want to guide the reader through two different examples, in order to give a stronger sense of how this framework should be used.

Before starting with some concrete implementation, we are going to discuss how to properly setup the scene in order to make all parts of the framework work together. After we'll talk about how to properly code all entities, given that the scene is always correctly configured.

The First example will be about a simple Cube Hologram, that upon interaction performs some kind of action. The example will be about explaining how to write all required scripts regarding an Hologram as well as how to use an HoloDoer in order to perform some dynamic behaviour.

The second example is instead focused on the usage of Tracking Areas and events. It'll show how to setup a simple scene with a trackable area, to which Holograms can dispatch events into.

6.1 Creating the scene

Before starting working with the AW framework it's necessary to import all required scripts and assets inside the desired project, this can easily be done by extracting all contents of the framework's unitypackage.

The same must be done with Vuforia's assets if it is the desired technology for handling space coupling. The unitypackage can be downloaded on Vuforia developer's portal.

Now that we have all required prefabs and components in our project, we need to properly configure the scene for the framework to work correctly. For an automatic configuration, the developer can use the window located under Window >AW Framework Config, that will quickly setup all prefabs inside the Hierarchy

for the selected Networking and AR systems.

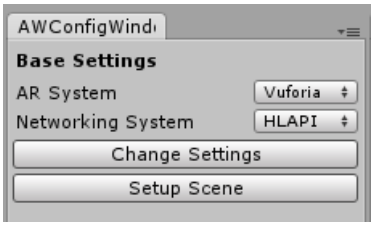


Figure 6.1: The configuration window of the framework. Implemented with a custom editor extension.

HLAPI - Networking Since at the right moment HLAPI is the only networking system available we'll start by discussing its proper configuration.

First of all, it is necessary to include into the scene the object *AWControlCenter* located inside the prefab folder of the project window.

The component *AWNetworkManager* should be assigned to this GameObject, becoming the one central point for network connectivity and related events. AWNetworkManager needs to know:

- the relative player prefab to instantiate at runtime.
- all prefabs of Holograms present into the scene.

A ready to use player object for HLAPI can be found inside the Prefabs folder, named PlayerAgentHLAPI.

The framework includes an user interface, optimized for mobile, usable to start a client or server instance of the application. Before starting it is advised to include into the scene the Canvas object located under the Prefabs folder, this UI can then be replaced in later times with an application specific one.

Vuforia - Space coupling Using Vuforia for target-based space coupling requires the scene to be set up as follows:

- ARCamera needs to be the scene default camera, configured properly with the developer's license key.
- A Target Behaviour must be used to mark a specific point in the physical space. This object needs to be parent of the *_world* GameObject inside the Hierarchy.

These are the simple actions that needs to be performed before starting using the framework. If some of the previous steps are missed, both the Inspector view of an Hologram, or the Debug view, will tell the developer what needs to be configured.

Summing up, these conditions about the state of the scene must be true, if HLAPI and Vuforia are chosen technologies:

- AW Control Center needs to be present with attached the following components:
  - AW Network Manager
  - AW Config
  - Mono Event Context
  - Screen Logger
- Spawn Info of AW Network Manager needs to have the reference of the Player Agent prefab.
- A Target Base like MultiTarget, ImageTarget, etc needs to be present, be parent of *_world* GameObject.
- AR Camera needs to be part of the scene and configured with:
  - The App License Key
  - the World Center
  - the Target Database to be imported in the application

At this point the user is ready to start defining all Holograms and HoloDoers of the project. Examples of correctly arranged scenes can be found in the provided demo folder.

6.2 Simple Cube Example

Let's consider this to be the "hello world" for Augmented Worlds. We want to build an application where the Hologram of a cube is instantiated into the world and shared by his users. Upon interaction the cube switches its colors and after a specific number of times, it performs some kind of animation.

Let's start by defining the Cube Model. The script implementing *IModel* would include a property that describes its rendering color with the RGBA standard.

```
using System.Collections;
using AWFramework;
using UnityEngine;

/**
 * It's the model of the cube
 **/
public class CubeModel : MonoBehaviour, IModel
{

    float xRotDegree = 0;
    float yRotDegree = 0;
    float zRotDegree = 0;
    public Color color;

    float YRotDegree {
        get {
            return this.yRotDegree;
        }
    }

    float XRotDegree {
        get {
            return this.xRotDegree;
        }
    }

    float ZRotDegree {
        get {
            return this.zRotDegree;
        }
    }

    public Vector3 GetEuclidRot (){
        return new Vector3 (this.XRotDegree,
             this.YRotDegree, this.ZRotDegree);
    }

    void SetEuclidRot(Vector3 rot){
        this.xRotDegree = rot.x;
```

```
        this.yRotDegree = rot.y;
        this.zRotDegree = rot.z;
    }

    public Color GetColor(){
        return this.color;
    }

    //Shared hologram methods

    public void SetColor(Color color){
        this.color = color;
    }

    public void SetColor(int colorInt){
        Color color =
            colorInt == 1 ? Color.red
                : Color.blue;
        SetColor(color);
    }

    public void Rotate(Vector3 axis, float degree){
        Vector3 newRot =
            GetEuclidRot() + (axis * degree);
        SetEuclidRot(newRot);
    }
}
```

Now that the cube model is clear, let's build the Game Object, or to be more specific, the Hologram, directly from the editor.

- Select GameObject >3D Object >Cube from the menu to create a new cube.
- Move the cube under the *_world* GameObject in the Hierarchy.
- Position the cube in order to be visible by the camera, for example at (0,0,0).
- Create a new prefab in the project window and store the newly created GameObject.
- Add the prefab to the AWNetworkManager spawning list.

Inspecting the cube shows us that it has a Cube Mesh and a Box Collider, these are already part of the Cube object created by the editor; we then need to:

- Add the HologramComponent script.
- Add a NetworkTransofrm (will automatically add a Network Identity).
- Add the newly created CubeModel script.

From this point on, we can start reasoning about the Cube View and all his relative graphical properties.

Switching colors Let's start writing the *CubeView* script, that for now will simply change the cube color upon user interaction.

```
using UnityEngine;
using System.Collections;
using AWFramework;

public class CubeView : MonoBehaviour, IView
{

    Renderer rend;
    HologramComponent hc;
    int colorInt = 0;
    //
    public Color[] colors;

    void Start ()
    {
        rend = GetComponent<Renderer> ();
        hc = GetComponent<HologramComponent> ();
        if(colors == null || colors.Length == 0){
            colors = new[] {Color.blue, Color.red};
        }
        SetColor(colors[0]);
    }

    void Update ()
    {
        //Temporary View Control
```

```
        //SetColor test
        if (Input.GetMouseButtonDown (0)) {
            RaycastHit hit;
            Ray ray =
                Camera.main
                    .ScreenPointToRay (Input.mousePosition);
            if (Physics.Raycast (ray, out hit, 100.0f)) {
                if (hit.collider.gameObject == gameObject) {
                    SwitchColor();
                }
            }
        }

        //Rotate Test
        if (Input.GetKeyDown (KeyCode.A)) {
            hc.Invoke ("Rotate", Vector3.up, 10);
        }
        if (Input.GetKeyDown (KeyCode.S)) {
            hc.Invoke ("Rotate", Vector3.down, 10);
        }
    }


    void SwitchColor(){
        colorInt = (colorInt + 1) % colors.Length;
        hc.Invoke("SetColor", colors[colorInt]);
    }

    //shared hologram methods

    public void Rotate (Vector3 axis, float degree)
    {
        transform.Rotate (axis, degree);
    }

    public void SetColor (Color color)
    {
        if (rend != null) {
            rend.material.SetColor ("_Color", color);
```

```
            Log ("color changed to"
                + color.ToString ());
        } else {
            Log ("renderer not found");
        }
    }

    public void SetColor (int colorInt)
    {
        Color color = colors[colorInt];
        SetColor (color);
    }
}
```

In the code, raycast[20] is used to detect interactions with the cube collider. Upon interaction the Hologram Invoke method is called, stating that the procedure *SetColor* should be called on both CubeModel and CubeView instances of that Hologram.

Right now we've obtained local synchronization between the Hologram's view and model, however what we really want is to update the state of the cube on the server and receive the data back as soon it is updated.

For gaining remote view-model synchronization we'll need to implement the cube *NetworkSynch* script.

```
using UnityEngine;
using System.Collections;
using AWFramework;
using UnityEngine.Networking;

public class CubeNetSync : HLAPINetworkSync, INetworkSync {

    CubeView view;
    CubeModel model;

    void Start(){
        view = GetComponent<CubeView>();
        model = GetComponent<CubeModel>();
    }
```

```
[ClientRpc]
public void RpcSetColor(Color color){
    view.SetColor(color);
}
[ClientRpc]
public void RpcRotate(Vector3 axis, float degree){
    view.Rotate(axis, degree);
}

//Shared hologram methods

public void SetColor(Color color){
    if(isServer){
        //send view update information to all clients
        ScreenLogger.getLogger()
            .ShowMsg("called from server "
            + color.ToString());
        RpcSetColor(color);
    }
    if(isClient){
        ScreenLogger.getLogger()
            .ShowMsg("called from client"
            + color.ToString());
        //send the command to the server
        SendCmd("SetColor", color);
    }
}

public void Rotate(Vector3 axis, float degree){
    if(isServer){
        ScreenLogger.getLogger()
            .ShowMsg("called from server: ROTATE " + degree);
        RpcRotate(axis, degree);
    }
    if(isClient){
        ScreenLogger.getLogger()
            .ShowMsg("called from client:ROTATE " + degree);
        SendCmd("Rotate", axis, degree);
    }
}
```

```
    public override
        void OnCurrentStateReceived (NetworkMessage msg){
        //empty for now
    }
    public override StateMessage GetCurrentState (){
        //empty for now
    }

}
```

This script derives from *HLAPINetworkSynch*, meaning it shares both server and client side code for networking.

The call of the method SetColor is split between the client and server implementation. When the user calls SetColor, a new Command is sent to the server, specifying its RGBA values. Upon command reception, Invoke is called on that Hologram on the server, triggering the server-side code of the script. When the server calls SetColor() the command's parameter is dispatched to all clients through a ClientRpc call.

The synchronization of the cube's state seems complete, however, what happens when the client joins the server at later time? From the client perspective, the script only updates the color property when an RPC call is performed from the server. We need a way to synchronize the cube representation as soon as a connection is ensured.

This can be done by implementing the methods *OnCurrentStateReceived* and *GetCurrentState*.

```
    //Called as soon a connection is enstabilished
    public override void OnStartLocalPlayer ()
    {
        base.OnStartLocalPlayer();
        AskCurrentState(); //Ask the server
        // for the current state message
    }

    /// <summary>
    /// Handler that processes the current
    /// state message received from the server
    /// after AskCurrentState is called.
```

```
    /// </summary>
    /// <param name="msg">Message.</param>
    public override
        void OnCurrentStateReceived (NetworkMessage msg)
    {
        StateMessage sm = msg.ReadMessage<StateMessage>();
        Debug.Log ("state message received "
            + sm.ToString() + " " + gameObject.name);
        Color c = (Color) sm.GetValue("color");
        view.SetColor(c);
    }

    /// <summary>
    /// Gets the current state of the object from the model.
    /// </summary>
    /// <returns>The current state message.</returns>
    public override StateMessage GetCurrentState ()
    {
        StateMessage sm = new StateMessage();
        sm.SetValue("color", model.color);
        return sm;
    }
```

Jumping We now want to make the cube perform a jumping animation after it changes color a certain number of times. For the sake of the example we'll not discuss details about the animation, instead we'll focus about how to trigger this behaviour in the expected manner.

The ability of counting doesn't really fit the cube model, that in accord to the framework specifics, only stores data about the Hologram structure. We want to decouple the behaviour from the model, for doing this we make use of an HoloDoer.

Let's create the script CubeDoer, deriving from *HoloDoerComponent*. This script will listen for incoming messages of the main event context, and when a certain number of events are received, it will call the Jump function by calling the Invoke method of the Cube Hologram.

```
using UnityEngine;
using System.Collections;
using AWFramework;
```

```
public class CubeDoer : HoloDoerComponent {

    public HologramComponent cubeHologram;
    public int counterLimit = 5;
    int counter = 0;

    // Use this for initialization
    void Start () {
        Init();
        SubscribeToMainEventContext();
    }

    // Update is called once per frame
    void Update () {
        ExecEventsInQueue();
    }

    public override void Exec (IEvent e)
    {
        Debug.Log("Executing event " + e.ToString());
        CountForJump();
    }

    void CountForJump(){
        counter++;
        if(counter >= counterLimit){
            counter = 0;
            cubeHologram.Invoke("Jump");
        }
        Debug.Log("Coutner: " + counter);
    }
}
```

For the sake of separation we create a new empty Game Object and consecutively add the CubeDoer script to it. We then assign the Cube Hologram reference of that script from the inspector. We also flag *Disable On Client*, since in this case the HoloDoer functionality are server-sided.

We now need to make the Cube Hologram generate an event when it changes color. For doing this we simply modify the Cube Model for sending a BaseEvent

to the main event context.

```
public void SetColor(Color color){
    this.color = color;
    SendCountEvent();
}

void SendCountEvent(){
    BaseEvent e = new BaseEvent();
    e.SetSender(GetComponent<HologramComponent>());
    AWConfig.GetInstance().MainEventContext.Send(e);
    Debug.Log ("Event sent " + e.ToString());
}
```

All that is left to do now, is to implement the Jumping method on both View and NetworkSynch components.

```
//CubeNetSynch

[ClientRpc]
public void RpcJump(){
    view.Jump();
}

public void Jump(){
    if(isServer){
        //send view update information to all clients
        ScreenLogger.getLogger()
            .ShowMsg("called from server: JUMP");
        RpcJump();
    }
}
```

```
//CubeView

public void Jump ()
```

```
    {
        //trigger the animation
    }
```

We can now test the application by running both a server and a client instance, after building the program using the engine Build Window. The effect should be the one described at the beginning, when the cube is clicked a certain number of times, it performs the jumping animation.

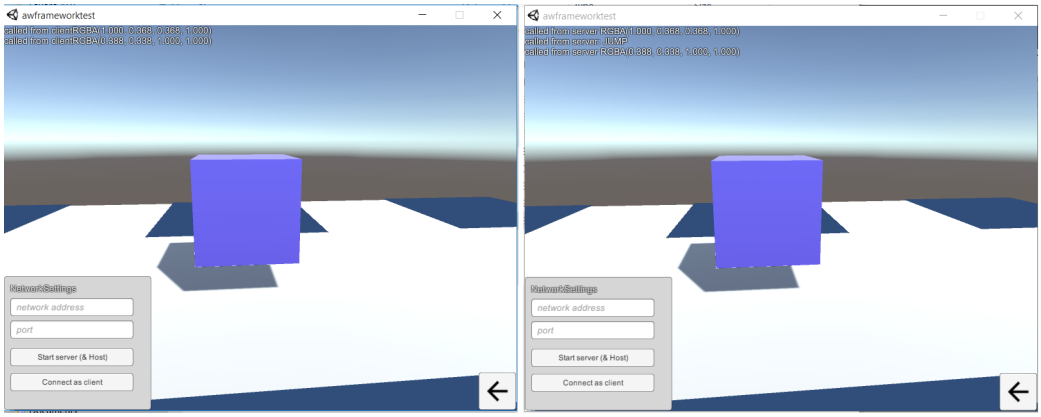


Figure 6.2: The example shows that the two instances of the application are consistent. The CubeDoer, after counting to 5, finally invokes Jump() on the Cube Hologram.

6.3 Tracking Area Example

This example is focused on explaining the correct configuration and usage of Tracking Areas.

Let's consider a simple scenario where an entity can move "almost" freely on a flat surface. We want this entity to be able to turn on and off a light upon user input, but only when it stands inside a specific area.

Let us start by arranging the scene in a way similar to the previous example.

- Place the Cube Hologram prefab at the center of the scene, this will become our "light".
- Add this Hologram to the AW Network Manager spawning list.
- Create a new cylinder object from the window GameObject >3D Object.

- Add to it the scripts Network Transform, Hologram Component and EventContextCollector.
- Store this object into a prefab and add it to the Network Manager spawning list as well.

The cylinder will be our moving entity, simulating a rolling barrel. This Hologram doesn't really need a Model component, all that we need to synchronize are its spatial properties, that are already stored inside the Transform component. We can then use NetworkTransform of HLAPI to automatically synchronize its position.

We then need only to write the NetSynch and View parts of the Cylinder Hologram, but before starting let's attach the *EventContextCollector* component to the Game Object as well.

The *EventContextCollector* is a (framework specific) component that automatically gathers all event contexts of Tracking Areas the Game Object is inside. It can then be used by other scripts for sending events directly into contexts he collected.

For creating the Tracking Area we can simply drag and drop the prefab from the AW Framework folder in the project window, or as well:

- Create a new Empty GameObject.
- Attach to it a Box Collider (or any other).
- Flag the collider as a Trigger
- Attach the TrackingAreaBehaviour script to the object.

This Tracking Area should somehow be "in the way" of the range of movement of our barrel. We can adjust position and geometry of the object/collider directly from the scene window.

Let's then start by writing the Hologram View script of the cylinder.

```
using System.Collections;
using AWFramework;

public class CylinderView : MonoBehaviour,
                            IView, IEventSender {

    HologramComponent hc;
    EventContextCollector ecc;
```

```
Vector3 force = Vector3.zero;

void Start () {
    hc = GetComponent<HologramComponent>();
    ecc = GetComponent<EventContextCollector>();
}

void Update () {
    Moving ();
    //pressing button
    if(Input.GetKeyDown(KeyCode.Space))
        hc.Invoke("SpacePressed", 1);
    if(Input.GetKeyUp(KeyCode.Space))
        hc.Invoke("SpacePressed", 0);
}

void Moving(){
    //handle input
    float x = Input.GetAxisRaw("Horizontal");
    if(x != 0){
        Vector3 dir = new Vector3(x,0,0);
        hc.Invoke("Move", dir);
    }
    //actually move
    transform.position += force * Time.deltaTime * 2.5f;
    force = Vector3.zero;
}

public void SendBtnEvent(int state){
    if(ecc != null){
        ButtonPressedEvent bpe =
            new ButtonPressedEvent(this, state);
        ecc.Send(bpe);
    }
}

/// <summary>
/// Gives the input to move for one frame.
/// </summary>
/// <param name="dir">Dir.</param>
```

```
    public void MoveLocal(Vector3 dir){
        force = dir;
    }
}
```

```
public class ButtonPressedEvent : BaseEvent
{

    int state = 0;

    public ButtonPressedEvent (IEventSender sender, int state)
    {
        SetSender(sender);
        this.state = state;
    }

    public int State {
        get {
            return this.state;
        }
    }
}
```

We use MonoBehaviour's Update method to perform the dynamic movement of the barrel through time, in response to the user input.

Only the server has rights to perform actions on Holograms, all the user can do to it is sending Commands. When the user presses the Right or Left input of the *Horizontal* axis, a command specifying the intended direction is sent to the server. The actual update of the position is then deferred to when the remote Hologram has actually moved. Using the NetworkTransform component, the position of the cylinder is updated automatically to all clients, without it, we need to write our synchronization code.

In the Update function still, we Invoke SpacePressed on the Hologram when the space input key is pressed or released.

We also implement the method SendBtnEvent that propagates the event *ButtonPressedEvent* to all context the collector has gathered.

The Cylinder Hologram's implementation ends with the Network Synchroniza-

tion script.

```
public class CylinderNetSync : HLAPINetworkSync,
                                INetworkSync {

    CylinderView view;

    public void Start ()
    {
        view = GetComponent<CylinderView> ();
    }

    public void Move(Vector3 dir)
    {
        if (isServer) {
            view.MoveLocal (dir);
        }
        if (isClient) {
            SendCmd ("Move", dir);
        }
    }

    public void SpacePressed(int state){
        if (isServer) {
            view.SendBtnEvent(state);
        }
        if (isClient) {
            SendCmd ("SpacePressed", state);
        }
    }

    //

    public override
        void OnCurrentStateReceived (NetworkMessage msg){
        //nothing
    }

    public override StateMessage GetCurrentState ()
    {
```

```
        return null;
    }
}
```

When a client Invokes Move() on the Hologram a command is sent to the server that eventually executes the server-side portion of the code. The server will then move the GameObject and NetworkTransform will synchronize its position to all clients.

When *SpacePressed* is called, the client will send the command to the server, and like before, the server will call the specific view method, this time making the Hologram generate a ButtonPressedEvent.

For the CylidnerHologram everything seems in order, but we still miss an important piece of the puzzle, what happens when an event is sent to the Tracking Area Context?

We need an entity handling the Cube Hologram, listening for events from the Event Context. This entity is an HoloDoer, we can easily update the CubeDoer script of the previous example to perform the right actions on our "lightbulb".

```
public class CubeDoer : HoloDoerComponent {

    public HologramComponent cubeHologram;
    public int counterLimit = 5;
    int counter = 0;
    public MonoEventContext[] subscribeTo;

    // Use this for initialization
    void Start () {
        Init();
        SubscribeToMainEventContext();
        foreach(MonoEventContext mec in subscribeTo){
            mec.Subscribe(this);
        }
    }

    // Update is called once per frame
    void Update () {
        ExecEventsInQueue();
    }
```

```
    public override void Exec (IEvent e)
    {
        Debug.Log("Executing event " + e.ToString());

        CountForJump();

        System.Type type = e.GetType();
        if(type.Equals(typeof(OnTrackingAreaEnterEvent))){
            Color c = Color.red;
            cubeHologram.Invoke("SetColor", c);
        }
        if(type.Equals(typeof(OnTrackingAreaExitEvent))){
            Color c = Color.blue;
            cubeHologram.Invoke("SetColor", c);
        }
        if(type.Equals(typeof(ButtonPressedEvent))){
            ButtonPressedEvent bpe = (ButtonPressedEvent) e;
            if(bpe.State == 1)
                cubeHologram.Invoke("SetColor", Color.yellow);
            else
                cubeHologram.Invoke("SetColor", Color.grey);
        }
    }

    void CountForJump(){
        counter++;
        if(counter >= counterLimit){
            counter = 0;
            cubeHologram.Invoke("Jump");
        }
        Debug.Log("Coutner: " + counter);
    }
}
```

From the Inspector, we now need to assign the reference to the Cube Hologram and the Tracking Area Event Context to the CubeDoer script.

Finally we can test the application, the results should show the cube changing color from yellow to gray upon pressure of the space key, but only when the cylinder is inside the tracking area. Moreover, the cube will switch from blue to red when the GameObject enters and leaves the area respectively.

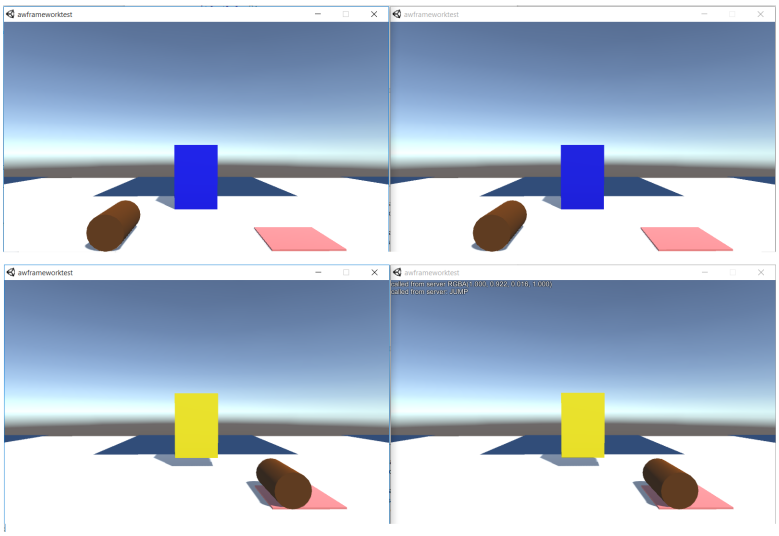


Figure 6.3: The image shows consistency between two instances of the application. The space input key shows effects only when the cylinder is inside the tracking area.

6.4 Building and testing

As mentioned before, our framework for now makes only use of tools provided by the Unity engine. The application can then be built directly from the editor, through the Building section; there's no need to further configure and setup different systems.

In our case, the client and server application shares the same code, meaning there's no separation at all between code from a server and client executable. This might not be the case in the future of course, where more scalable Networking System might be used instead. In that case, the client's application will still be build from the editor.

For the standalone pc build there's no need for further configuration, the developer just selects the main scene the program starts with. For mobile there's the need to configure the application package name. The developer can switch form one platform to another by simply using the button inside the Build Settings window.

For testing, the developer needs to run both instances of the application, we suggest the server to be always a standalone application. When the program starts the user can select if the running instance is either a server or client from the provided GUI that opens right after launch.

When the user choose either the server or the client instance, the AW Network Manager performs all its configuration steps, and then effectively instantiates all

Holograms into the scene.

The editor might also be used to run the application, so that the developer can debug its code and make use of the editor gizmos.

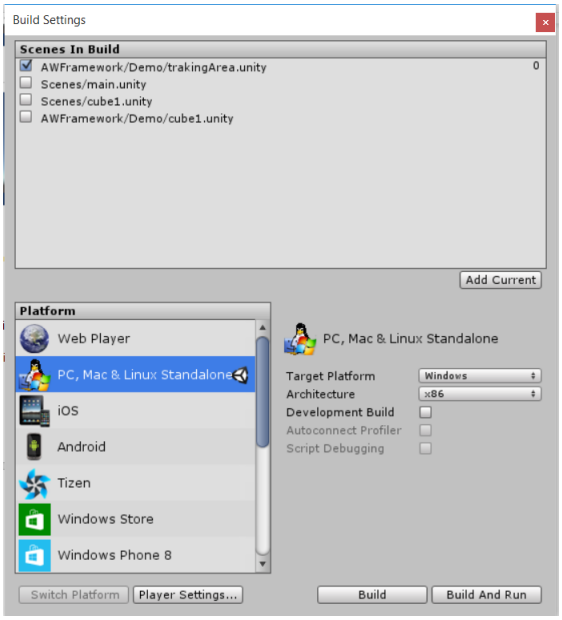


Figure 6.4: The Unity build setting window. Each required scene needs to be added to the build using the *Add Current* button.

6.5 Reposiory

All the framework's code, unitypackage, project, assets and demos can be found in the following repository.

`https://github.com/pievis/AWFramework`

Demo scenes can be directly opened and inspected:

- demo/scenes/cube1
- demo/scenes/trackingArea

Chapter 7

Wrapping things up

In the previous chapters we exposed a novel approach for building complex mixed-reality applications based on the Augmented World model. We addressed many concerns about the major aspects of these programs and we proposed ways on how to efficiently encapsulate these functions in order to obtain a well established structure that wraps together different technologies.

In this chapter we want to give a sense of closure to our work. Here we stop and briefly analyze the current state of the art of technologies currently showcased for the development of mixed-reality cooperative systems, comparing them to what we learned from the Augmented World model.

We'll discuss why we think it's important to have a concrete framework for developing such systems, carrying on with an afterthought about what to implement in the future in order to meaningfully extend its functionalities.

7.1 State of the art of technology

The way we've always experienced computation is changing. With the introduction of Mixed Reality, we are slowly undergoing the creation of a new paradigm, for now only carried by researchers and big companies.

This year, Hololens has finally been released to the public, available only to a small set of developers. Thanks to the recent publication of its documentation, we can start to reason about how a big firm such as Microsoft is addressing problems much likely similar to the ones presented in the Augmented World model, from a programming perspective.

From the beginning, Microsoft presented Hololens as a system embedded with some cooperative capabilities, where two or more users wearing the same headset could display and interact on the same augmented environment. During the Hololens Build 2016 conference, different applications showed this peculiar feature,

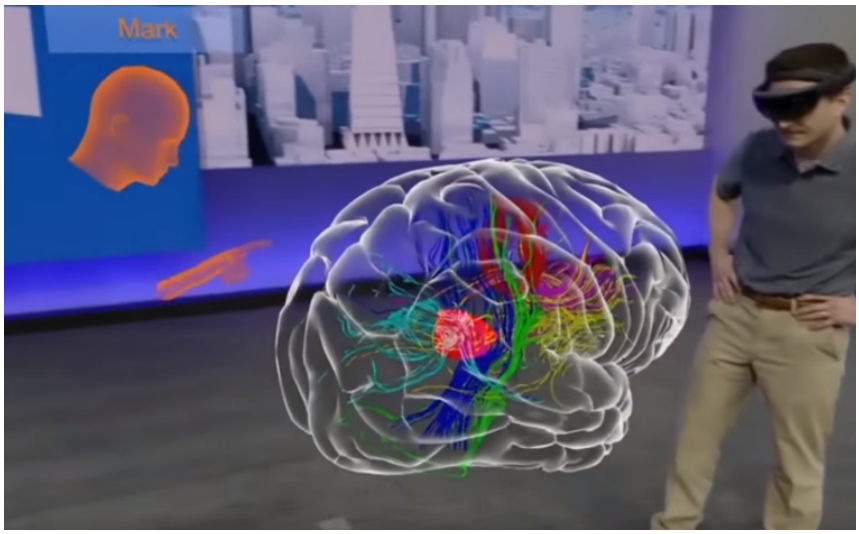


Figure 7.1: Showcase of a medical application for cooperative learning supported by Hololens. This application was shown during Build 2016 conference and displays a flying head as the avatar of some remote user.

most notably Autodesk Fusion 360, that provides a collaborative way of shaping 3D objects.

Besides some promotional material, there's no other way of knowing about what it is like to build Holographic applications. We have to resort at the still poor documentation.

Even Microsoft has chosen Unity as one of the enabling engine for building holographic applications, providing its own sets of scirpts and prefabs, that somehow needs the support of Visual Studio to be compiled.

The Hologram API is of course focused on shaping mixed reality experiences around the device, and it is not focused on the creation of multi-user programs. The main concern of cooperation is pushed past the gestures and voice recognition functionalities.

Shared experiences seems to be addressed inside the documentation[21], stating that Hololens APIs for Unity provides means for sharing the same environemnt. However no further explanation is given about specific Hologram state-synchronization code and user interaction at the time.

At the current state of the art, Microsoft may be not properly showing its plans about the future of shared experiences programming-wise. But it is of course a major concern. Object Theory[22] has in fact recently launched a Mixed Reality Collaboration Service for Hololens, providing a layer of functionalities for multi-user holographic experiences.

7.2 Why a framework?

Looking past what Microsoft has displayed so far, we notice that form a programming perspective, the creation of cooperative Mixed-reality systems can be performed by tightening up different entities to compose a coherent augmentation of the world.

However starting with the simple tools provided by Unity and third parties for the creation of such systems is just too naive. When the complexity of the application increases and move closer to the Augmented World model the need of a well structured framework is strong.

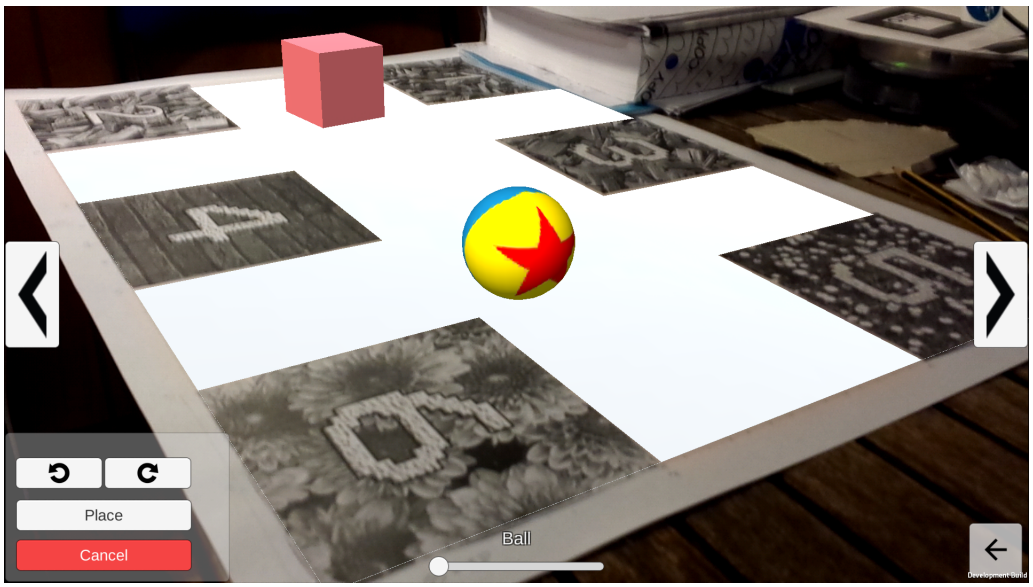


Figure 7.2: Example of a mixed reality application made by using Unity and Vuforia alone. The application enabled users to interact and manipulate objects inside a limited augmented surface. Code regarding cooperation, user shape and interaction, synchronization of entities was all ad-hoc, no framework was used at the time.

We remember that an Augmented World program is not simply a multi-user mixed reality experience. The world is made to exist independently form the presence of its user, and it's inhabited by autonomous entities that continuously change their surroundings.

When we want to build these in-between reality and virtuality experiences, we need a strong understanding of all aspects of a single Hologram. For this reason in the provided implementation we forced developers to split up the three main concerns regarding an Hologram.

In Unity, composition is a winner, everything is handled by loosely coupled references between GameObjects and components. This is good because it allows flexibility for behaviour building. However, in regard of networking, and Augmented Worlds in particular, we need to be strict on some of the model requirements. Not having total control over the state of an Hologram might cause inconsistencies and make life harder to the developer during the debugging process.

We put major regard upon the Network Synchronization behaviour of a Hologram. As we stated before, the developer should be able to choose what Networking System to use for building his program, because it has an enormous impact on the overall multi-user experience, including the number of user supported at a time.

If we want to go large scale, we sure need more concise ways for developing scalable multi-user and dynamic systems, maybe inspired by Massively Multiplayer Online Games architectures.

For these main reasons, we believe that the developers will sure appreciate a software stack that can remove some of the complexity of writing these applications, based on some interpretation of the Hologram concept.

Of course the main advantage remains the fact that these ready to use tools will make the development of the system faster, moreover in our case study we can use them to gain experience about building Augmented Worlds.

7.3 The future

At its current state, the framework might miss some of the main services that a software stack for Augmented Worlds should provide. Here we list some of the funtionalities that were left unhandled and need to find the right space inside the AW framework.

Wearables Of course to make the user fully experience an Augmented World, the framework needs a way to give support to some kind of wearables, mostly headsets. Fortunately the combination of Unity and Vuforia seems already capable of handling different kid of devices.

By providing the right configuration to the AR Camera component, Vuforia gives support to most common headsets, including gears that make use of the smartphone, like the Google Cardboard. This is possible because the component is already capable of splitting the image of the camera in two, creating the effect of stereoscopy on a single screen.

Vuforia also gives support to optical see-through devices, although Espeon Moverio[23] and ODG[24] are the only one supported at the time. Recently Vuforia has also announced a partnership with Hololens, but we don't know how it will effect the developing process through Unity at the moment.

Sensor bracelets like Myo[25] should also be considered in the future. This kind of devices could be used in order to provide a way of interaction in alternative to the device screen. This might call for the integration of specific APIs for such wearables, following a study of the device implementable features.

(a) ODG R-7

(b) Moverio BT-200

(c) Myo

(d) Google Cardboard

Figure 7.3: Small set of devices that the framework can already, or could, support in the future.

Decoupling user input At the current state of development, there's no framework provided entity to manage interaction between the user input and a specific Hologram. This means that the developer has to write his own user-interaction code, however we might be able to give support to this request by providing a context-aware mean for interaction.

In Unity, we are mostly used on handling inputs directly inside the Update method, by checking for specific external signal using the provided API.

Even in our example code, we wrote user-hologram interaction directly inside the View Component, using standard Unity input lookup mechanism. We used raycast, a technique based on detecting collisions between an imaginary line and an object, to check if the user was at the time tying to interact with the Hologram attached to our script.

There's no problem in handling interaction for a specific Hologram from the user device screen then, but what about other types of input? This is a major concern when we introduce headsets; at this point interaction with Holograms should be handled in new innovative ways. For example by using gestures, or vocal commands.

First of all then, we need a way to redirect a generic user input, even the simplest keypress, to a specific Hologram. For this we need to implement a user "selection" context, that dispatches events regarding inputs to the currently holding Hologram.

The behaviour about how to populate this context might be up to the developer, but the framework should at least provide some small set of scripts that automatically select the object the user is interested in controlling.

For example, a way of selecting the Hologram the user is interested to, is by implementing the "gaze" concept. Using raycast we can easily discover what the user is looking at by mean of the direction its camera is facing.

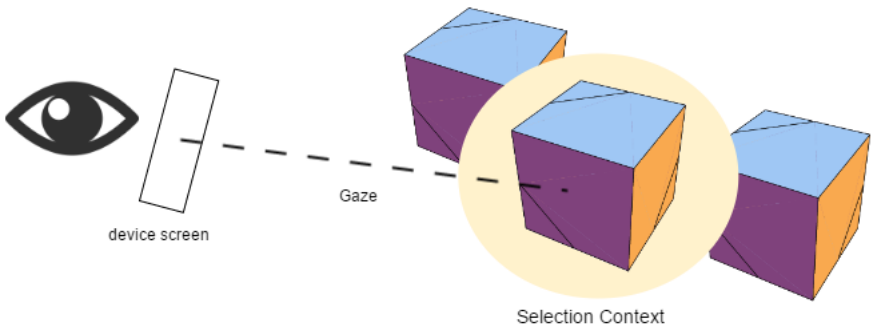


Figure 7.4: The figure shows a representation of the "gaze" concept. The user, looking trough his device directly to an Hologram, has selected its target for inputs.

The idea is to give the developer different layers for developing his own means of interaction. We should then provide some ready to use scripts, already implementing "gaze" and other types of selections, being them the extension of a basic type providing all standard functions in order to catch inputs and dispatch these messages to the currently holding Hologram.

When an input message reaches the Hologram, it can then be handled inside the View component thanks to specific handlers or inside the Update function directly.

Interaction is a major concern in the Augmented World model. The way of handling and controlling an Hologram should be natural to the user. We're excited to see what the future will bring in terms of APIs and technology that could effectively provide ways of handling virtual objects without any gimmick. In this field we might take note of progress made by Meta2's unique neuroscience-driven interface design principles to access, manipulate and share digital information, by using hands in the most natural way possible.

Figure 7.5: Proof of concept of Meta2 interaction handling. No other tools are required other than the user hands. Movements are tracked in a way that makes possible to move holograms like they where physical objects.

Localization and mapping The physical world, is for its nature, dynamic. Objects are often moved, introduced and removed from our environment. It is not feasible to remodel the representation of the world every time a big change is made in the room, like the case of moving furniture.

When our application needs to reason about physical boundaries, it is strong the need of having at our disposition some technique that can automatically map and model the ambient surroundings.

We can find two main ways of doing this, by either using the server, or by delegating it to clients.

From the server, we can track the position of objects from different sensors and cameras disposed inside the ambient. When a change is perceived, as long it is inside the "field of view" of this sensors, we can track and update the state of the world accordingly.

By delegating to clients, as long as they have the technology, we can make them continuously fetch information about the environment. While this information is being fetched, we could send the data to the server, so that it can increase its knowledge of the world and consequently generate the right collision meshes.

Some experiments about this functionality could be made using Vuforia's Smart Terrain technology, that given the right environmental conditions, is able to generate a small map of the ambient directly into the scene.

In the future, we might as well explore Google Project Tango[8] APIs for Unity. Project Tango enables apps to track a device's position and orientation within a detailed 3D environment while it is simultaneously mapped. This technology uses specialized depth sensors in combination with a gyroscope and the device camera

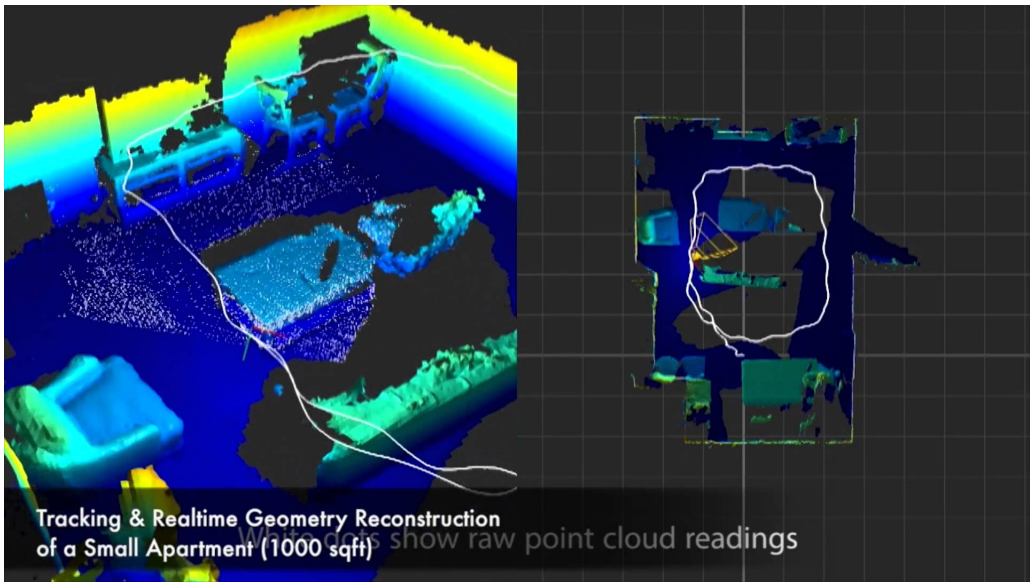


Figure 7.6: In this figure, Project Tango is tracking in real time the geometry of the space. It can detect spots he has already seen and update the user position accordingly.

to obtain its results.

Simultaneous localization and mapping it's another of the many features that should be taken in account when dealing with Augmented Worlds in the future.

Physical embedding Physical embedding was discussed as one of the main feature of the Augmented World model. However, this functionality was momentarily left unhandled to focus more on the virtual aspect of the model.

Physical embedding should be provided in a way similar to the Network Synchronization component, only this time the component should be able to interact using different communication protocols and different APIs, in regard to the physical device we want to take control.

Physical embedding should then be recognized as a part of an Hologram, so that its functions can be called through the Invoke method. The framework should provide an unique interface, called for example *EmbeddedSynchronization*, that is used to mark all scripts that require to send messages to the remote physical devices. These scripts of course needs an ad-hoc implementation of the device APIs made by the developer, before even starting to write this behaviour.

By doing this we've effectively decoupled this concern in an isolated behavioural script. However this could be the case where some hierarchical structure may provide some standard functions in a way similar to SendCmd for the NetworkSyn-

chronization component.

Chapter 8

Conclusion

As technology is moving forward, new programming scenarios opens. We've reached the point of developing software much similar to ones portrayed in old style sci-fi movies. With Augmented Worlds we want to bring reality and virtuality closer, envisioning a new generation of systems in which computation can take various form of augmentation.

In this paper we carried out an experiment about the creation of a software solution for building Augmented World programs, based on already existing technologies. With Unity as a strong 3D real-time engine, the complexity of the framework is reduced, imposing a reused-based approach on top of the use of Components. The Vuforia extension enables Augmented Reality functionality in the most transparent way possible, making it likely the best candidate for handling space coupling at the moment. HLAPI is another Unity-based technology that reduces complexity of writing real-time communication code for state synchronization.

The process of envisioning the framework was useful in order to clarify some of the essential pieces behind these programs. We've intrinsically imposed an abstraction, that makes use of Holograms and HoloDoers in order to shape the contents of AW applications. While developing the framework we learned how all subsystems are linked together, while still focusing on providing separation of concerns between the three Hologram functions.

We are still far away from the ideal Augmented World Framework suitable for large scale or complex systems, however we find peace knowing the software is not that far behind from the concept we had in mind.

The developing process went on smoothly without any unforeseen limit imposed by the used technologies, making the creation of this solution easier than expected. Moreover, we might not be that far away from the right interpretation on how to build these systems. In terms of design and programming Unity brings well known advantages to the implementation of real-time dynamic systems, giving the developer the flexibility to separate behaviour into scripts; and yet leaving the

definition of the right constraints to AW Framework specifics.

In the end we can learn a lot about Augmented World system by using this experimental framework alone, but the great given is the information acquired through the creation process.

Bibliography

[1] "Oculus rift specs." https://www.oculus.com/en-us/rift/.

[2] "Htc vive specs." https://www.htcvive.com/eu/product/.

[3] "Hololens hardware." https://www.microsoft.com/microsoft-hololens/en-us/hardware.

[4] P. Milgram and A. F. Kishino, "Taxonomy of mixed reality visual displays," *IEICE Transactions on Information and Systems*, p. 1321–1329, 1994.

[5] A. Croatti and A. Ricci, "Programming abstractions for augmented worlds," *AGREE! @ SPLASH 2015*, 2015.

[6] I. D. Bratman ME and P. ME, *Computational Intelligenge*, ch. Plans and resource-bounded pratical reasoning, pp. 349 – 355. ME, first edition ed., 1988.

[7] G. Fiedler, *Networked Physics*, ch. State Synchronization. online ed., 2015.

[8] "Project tango." https://get.google.com/tango/.

[9] B. Nystrom, *Game Programming Patterns*, ch. Update Pattern. gb, first edition ed., 2015.

[10] "Unity manual - execution order." http://docs.unity3d.com/Manual/ExecutionOrder.html.

[11] B. Nystrom, *Game Programming Patterns*, ch. Component. gb, first edition ed., 2015.

[12] B. Nystrom, *Game Programming Patterns*, ch. Game Loop. gb, first edition ed., 2015.

[13] G. Fiedler, *Game Physics*, ch. Fix Your Timestep! online ed., 2015.

[14] R. Gaul, "Component based engine design." http://www.randygaul.net/2013/05/20/component-based-engine-design/, 2013.

[15] J. Gregory, *Game Engine Architecture*, ch. Low-Level Engine Systems. CRC, second edition ed., 2015.

[16] "Vuforia dev portal - how to setup a simple unity project." https://developer.vuforia.com/library/articles/Solution/Compiling-a-Simple-Unity-Project.

[17] "Unity manual - the high level api." http://docs.unity3d.com/Manual/UNetUsingHLAPI.html.

[18] "Extending unity editor." http://docs.unity3d.com/Manual/ExtendingTheEditor.html.

[19] "Binaryformatter class." https://msdn.microsoft.com/en-us/library/system.runtime.serialization.formatters.binary.binaryformatter(v=vs.110).aspx.

[20] "Unity tutorials - raycast." https://unity3d.com/learn/tutorials/topics/physics/raycasting, 2015.

[21] "Shared holographic experiences in unity." https://developer.microsoft.com/en-us/windows/holographic/shared_holographic_experiences_in_unity.

[22] "Object theory website." http://objecttheory.com/.

[23] "Moverio smart glasses." http://www.epson.com/cgi-bin/Store/jsp/Landing/moverio-augmented-reality-smart-glasses.do.

[24] "Odg overview." http://www.osterhoutgroup.com/system.

[25] "Myo website." https://www.myo.com/.