ENGINEER REPORT
END-OF-STUDIES INTERNSHIP
SPECIALIZATION F4 : SCIENTIFIC COMPUTING AND MODELING

Collaborative and immersive scatterplot rendering

Presented by : Sandy Le Pape

Project advisor :
Matt Adcock

Referent teacher :
Vincent Barra

Defense date :
27^{th} of August 2019

Internship duration :
5 months and 2 weeks

School year 2018-2019

Acknowledgements

First of all, I would like to warmly thank my CSIRO supervisor, Matt Adcock, who trusted me throughout the project, although my school specialization was relatively far from the laboratory's field of application. I am also very grateful to him for taking the time to find funding for the internship. Moreover, he took an interest in my professional desires in order to propose a project, adapted for me and useful for the laboratory's objectives. I would also like to thank his colleagues Stuart Anderson, Simon Malnar and Mingze Xi, who were able to advise me on the various technical aspects of my work and these technologies, which were totally new to me. I also would like to thank Brian Chen, a dynamic and passionate casual, who offered me his help many times. As part of this internship, I will also thank the main creator of the data visualization tool *ImAxes*, Maxime Cordeil, tool which has been a strong basis for the project. I also thank Vincent Barra as the pedagogical referent teacher for his great reactivity and availability, and allowed me to initiate the contact with Olivier Salvado, Dadong Wang and then my tutor. My thanks go to Pascale Granet and Megan McDonald for managing all the internship papers too. I would like to thank ISIMA, my engineering school, which enabled me to acquire some of the mathematical and computer skills necessary for a full understanding of the project, as well as CSIRO, a fulfilling workplace with a deep respect for nature and the individual. Finally, I wish the whole the members of the team I joined for this internship all the best.

List of Figures

Abstract

In order to fully master the analysis of large datasets, many sectors would like to be able to interact in real time with the corresponding visualizations. In recent years, the field of *immersive analytics*, combinaison of virtual technologies and data analysis, has offered intuitive and more flexible interactivity with data than computer softwares. However, they rarely exploit the possibility of a group experience.

This internship allowed me to implement a networked solution and to make improvements concerning the perception of the data in an immersive environment, by working on the graphic rendering in particular. The end of the internship will be fully dedicated to the integration of my work into an existing data visualization tool.

Keywords: *immersive analytics*, virtual technologies, networked solution, rendering

Contents

Presentation of the organization

This internship was entrusted to me by Commonwealth Scientific and Industrial Research Organisation (CSIRO), the Australian federal government agency in charge of scientific research, whose mission is to meet the greatest scientific and technological innovation challenges for the Australian territory. Its headquarters are located in Canberra, the capital, in the Acton district. This city hosts a very small number of industry head offices, mainly due to the fact that the country's economy is more present in cities such as Sydney, Melbourne or Brisbane. CSIRO has its own campus, accessible to all, with *Synergy* and *Discovery* buildings as the main ones. The particularity of this campus is to mix buildings specialized in the cultivation of green species, chemistry and IT development.

Figure 1: *Synergy* building at CSIRO in Canberra

The first form of CSIRO was born in 1916 as an advisory council for science and industry in Melbourne. The Institute evolved over the next ten years to become CSIR (Council of Scientific and Industrial Research). All these years, the vision and objectives remained the same: to provide research support for the development of the farming, mining and manufacturing industries across Australia. The fields of application evolved

over the years, turning more towards animal and floral life, but also the treatment of materials after the Second World War. During this war, CSIR supported the Australian defensive forces until 1949 when the CSIR has been renamed CSIRO, refocusing on the country's environmental concerns. Over the years, the Institute's research areas expanded, conducting research on urbanization, water and nutrition. Since 2014, in order to clarify the strategic reading of the laboratory, CSIRO has been structured around three main axes: scientific research (containing nine units gathering the specialities discussed above), biological infrastructures and tertiary services.

Throughout history, CSIRO has been responsible for numerous scientific and technological breakthroughs, whose influence has spread far beyond the borders of the Australian country. The Institute's most famous invention, which few articles highlight as coming from CSIRO, is Wi-Fi. Now widely used in many parts of the world, its research has been based on the Fourier transform and the study of the behaviour of radio waves in the environment in which they propagate. Another invention, used throughout Australia, is the plastic banknote, designed by the laboratory. These banknotes have the particularity of having a much greater durability compared to former ones, are made from polymers (so more ecological) and are less likely to carry dirt. In the early 1990s, one-month contact lenses were developed. Other discoveries, including some vaccines, have repeatedly illustrated the talent of the people working in the organization.

Figure 2: The most significant inventions of CSIRO

CSIRO is composed of multiple organizations, including Data61, wishing to meet today's challenges by mastering Data Science in relation to technologies. Data61 was created

in 2015, as a result of CSIRO's strong desire to promote collaboration between industries, laboratories and universities. From Cybersecurity to Artificial Intelligence, including Image Processing, this ingenious combination of domains has provided valuable assistance in the analysis and prediction of data for the medical, agricultural and telecommunications sectors.

I did my internship at the Immersive Environments Laboratory in Canberra, an integral part of Data61. This laboratory is specialized in the fields of Virtual and Augmented Reality. This place is a real catalyst for innovation which has significant funding to obtain the latest technologies, in order to stimulate the desire of researchers to master the technologies of tomorrow. The laboratory team regularly hosts industry and the general public for events, functions and demonstrations.

Figure 3: Laboratory logo

Recently, the immersive environments laboratory focused on possible ways to improve the readability of large amounts of data, through new analytics and visualizations in 3D virtual worlds, referred to as *immersive analytics*. This work is carried out in cooperation with other Australian universities, the government and some interested companies. Thanks to the equipment it has in the field of Virtual and Augmented Reality and its recent premises, the laboratory hosts each year some students to become familiar with these technologies. This year, I am lucky enough to be part of them.

Introduction

At a time when data is increasingly available, heterogeneous and complex to describe, it is necessary to be able to process and visualize this data in different ways in order to bring out valuable information. To do this, graphical representations such as histograms or point clouds are used to detect general patterns after applying multivariate statistical or learning methods for example. However, the data sets used to construct these representations are large and have a myriad of forms, involving the use of data fusion techniques in order to compare them. The question is whether this fusion can be automated and trustworthy or whether humans will have to intervene at some point.

On the other hand, VR and AR technologies are constantly developing, between the opportunities they offer to increase the accuracy of surgical operations or allow a total immersion for video games. The simple-minded interactivity (use of joysticks or gestures) and the immersion that these technologies provide make it possible to enjoy new kind of features.

Combined with visualization, data exploration and analysis, these technologies allow us to take a new step forward in interactivity: with simple joysticks, it is now possible to easily manipulate graphics or any other virtual object anchored in the real world, as shown in FIGURE 4. However, the use of these new technologies brings new problems. Collaborative data visualization is complicated, especially in Virtual Reality because each player's world is generated from different computers, and therefore do not necesarily have the same reference point. Besides, each player has his own visualization of a scene from their precise point of view, and cannot be shared with the same angle easily to other players. This produces some interesting *computer graphics* challenges.

As CSIRO is very oriented towards environmental applications related to the country, the project focused on a concern in the energy sector. Let's consider the field of construction for instance: depending on the building under consideration (house, apartment), its age, its location on the territory, how is it possible, using data sets, to measure interactively the energy impact in recent years and to what extent can the user's reading be facilitated? This decision-making tool will be based on a map of Australia, complete

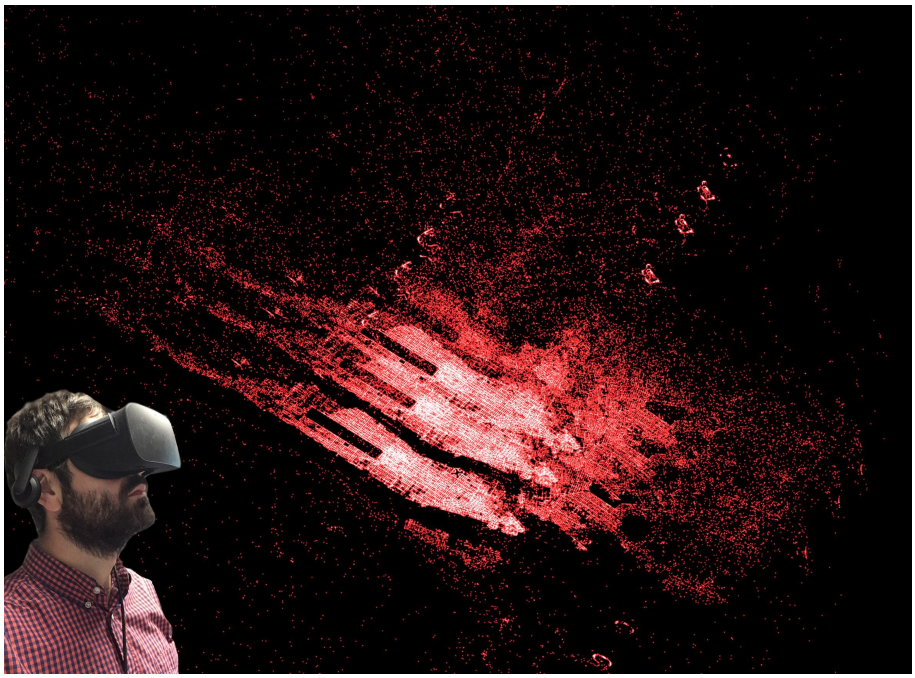


Figure 4: Example of an immersive data visualization [1]

data sets and visualization tools, and will display, depending on the location, a data visualization (bar charts, time series, etc.).

The purpose of this project is to visualize on a 3D map of Australia the impact of this use, based on energy consumption data sets. It will analyse the data collected in order to come up with some ways to optimise this consumption and potentially predict it. This collaborative project raises new issues in multi-user visualization. It is therefore also a question of finding innovative solutions to improve the quality of visualizations, by interacting directly with the graphics card. This dense and large-scale project, entitled *"Delivering the integrated data asset for the future of energy forecasting"*, bases itself on the NEAR project, a platform which underpins change of energy behaviour in Australia (its system is decribed in FIGURE 5). It involves the work of several CSIRO teams across the country.

In a first chapter, we will present all the softwares and tools used for the project, allowing the exploration and interactive analysis of multivariate data in a virtual environment. It was then decided that bringing a collaborative dimension to one of its tools, considered promising and designed for Virtual Reality, would be interesting (chapter 2). Finally, in a third chapter, we exposed research conducted to improve the data analysis capabilities and the data visualization rendering for each player.

As the laboratory's expectations were not really fixed at the beginning of the internship, the *Specifications* prepared in collaboration with my supervisor have been adapted

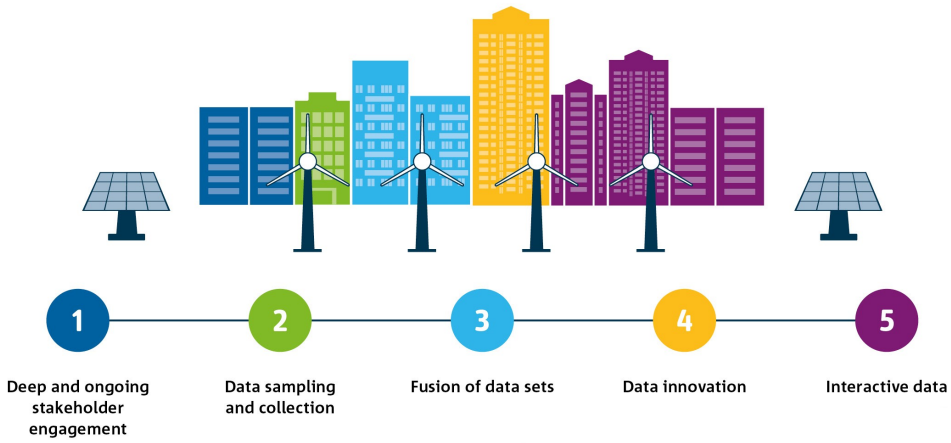


Figure 5: Unified system created by CSIRO for the NEAR program [2]

according to priorities and my professional objectives throughout the internship. The drawing of a forecast Gantt chart is therefore not possible. But here is the real Gantt chart (FIGURE 6):

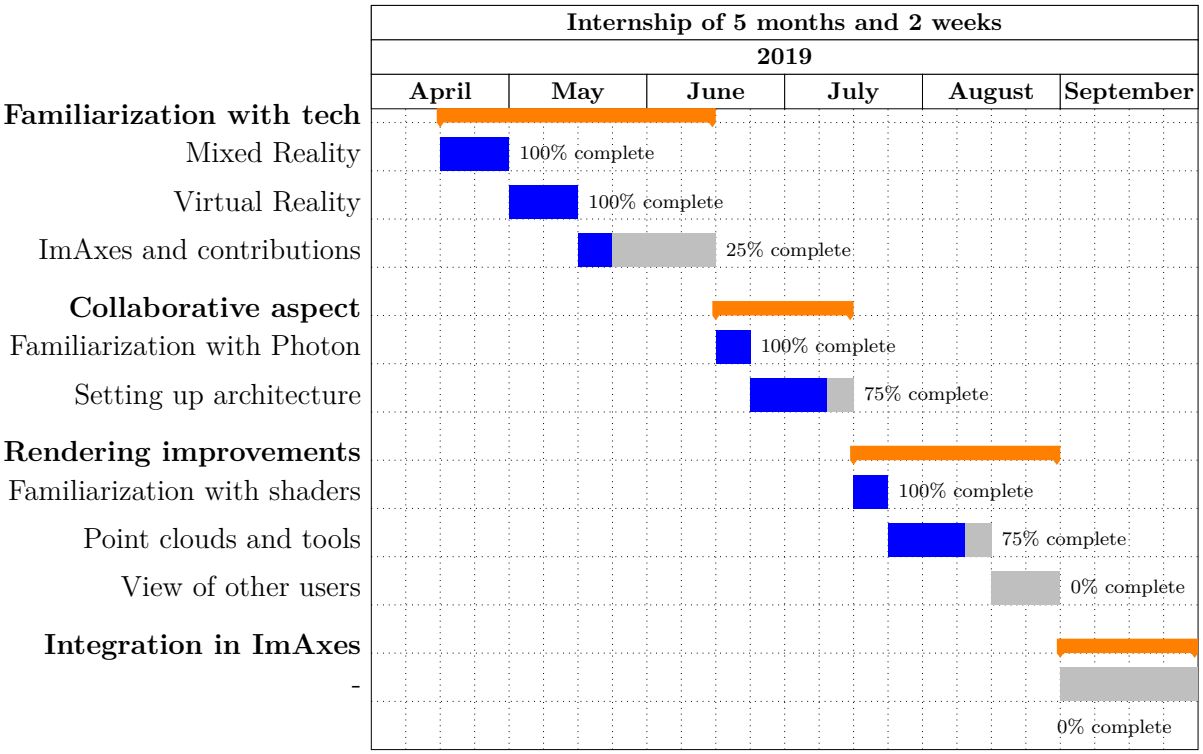


Figure 6: Real Gantt chart

Chapter 1

Overview of the technologies and first contributions

1.1 Presentation of Virtual Reality concepts and technologies

1.1.1 Virtual and Immersive Reality

Virtual Reality (VR) can be seen as the umbrella term for all immersive experiences, which could be created using purely real or synthetic world content, and even a hybrid of both. VR is mainly characterized by creating the illusion of immersing the user in an artificial 3D world, which was only visible from the outside through a computer screen. Thus, when the user holds a VR headset, their field of vision is substituted by another one allowing the visualization of the 3D virtual world (as illustrated in FIGURE 1.1). The player can interact by means of joysticks alike, called *motion controllers*.

(a) Real world base (b) 3D world substitution

Figure 1.1: Correspondence between real and virtual worlds

Throughout the project, the VIVE PRO headsets designed by Valve Corporation and

HTC have been used (FIGURE 1.2). Other VR headsets exist, mainly from Oculus and Microsoft competitors.

Figure 1.2: Exemple of a complete VIVE PRO VR toolkit

Most technologies offering VR experiences require the connection and the initialization of different devices represented on FIGURE 1.3:

- *base stations* also called *lighthouses* (A), helping to scale space in which the experience will take place, tracking the exact position of *controllers* and headset, etc.
- *controllers* (B), only tool enabling interaction with the virtual world, containing various buttons, triggers and analog pads that vary according to the models.
- the connection of cables (display port, usb port and power) from the headset (C) to the computer (E) through a box (D).

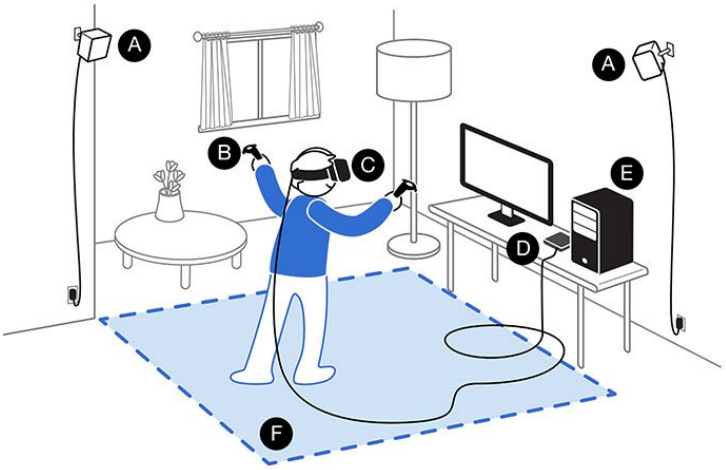


Figure 1.3: Game environment

Apart from that, other technologies can alter reality without immersing the player into another world.

1.1.2 Brief outline of Augmented Reality

As its name suggests, Augmented Reality (AR) does not substitute the field of vision unlike VR, but enriches it with virtual objects like virtual screen that informs the weather in real-time, etc.

Technologies to provide AR experiences are less expensive than VR ones. A simple smartphone with the dedicated applications could run AR games, PokemonGO and HoloChess (FIGURE 1.4) probably being the best known AR applications on this day. 3D glasses can also be used, like the Google ones. In this respect, a 3D movie (by wearing stereoscopic glasses) can be considered as an AR experience.

(a) HoloChess game conceived via ARKit (b) Game launched on a Smartphone

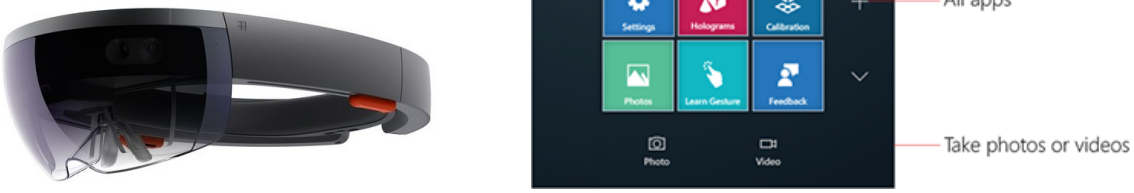
Figure 1.4: AR game HoloChess derived from Star Wars

Finally, Microsoft has been able to broaden what existed between VR and AR before.

1.1.3 Mixed Reality : between Augmented and Mixed Reality

Mixed Reality (MR or XR) is often considered as the result of the fusion between Virtual and Augmented Reality, in the sense that it allows interaction with virtual objects overlaid in the real world. Their placement, orientation and physics are based on the real world. This can only be offered with high-performance rendering that overcomes AR needs. The APPENDIX 1 helps to clarify the significant differences VR, AR and MR, notably thanks to the *Reality-Virtuality continuum*.

Microsoft HoloLens technologies (FIGURE 1.5) belonging to the WMR (for *Windows Mixed Reality*) have put light on MR potential. They are more intended for developers because the technology is still in a proof-of-concept phase and needs a few more evolutions so that the lay public could master it enough. The carried out work has mainly been done on Microsoft HoloLens 1. One of the strongest advantage of HoloLens headsets is its ability to operate as a *standalone* device, delivering a completely untethered experience.



(a) HoloLens headset

(b) Menu appearing when headset is worn

Figure 1.5: HoloLens headset and Home menu

MR applications can also be launched on WMR immersive headsets as well and just need a supplement of code for the *controllers* to work.

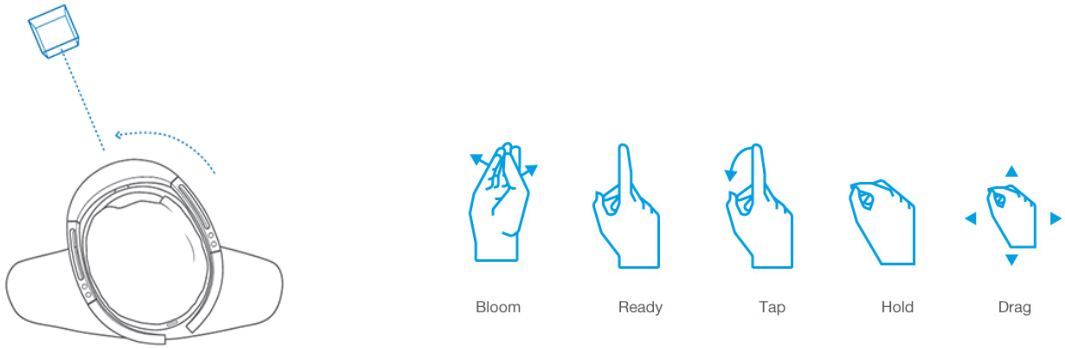
The primary functionality guiding the user in their interactions is a target called *gaze*. It allows the player to simulate a point according to the movements of user's head wearing a headset. The *gaze* has been conceived as a laser pointing straight ahead.

At the moment when the *gaze* intersects a hologram, an action can be taken. Three possibilities opens to us then:

- the *bloom* gesture, which is reserved exclusively to return to Hololens Home menu,
- the *air-tap* gesture, which enables selection (similar to a click on a computer),
- the *voice command* which offers infinite possibilities because Hololens voice recognition is advanced. Just pronounce a word in English and associate it within the code to make the action possible when the application is launched.

It is possible to create our own gestures, which would be a combination of primary gestures, but Windows strongly recommend not to do it because it violates the encapsulation of the technology. It fosters predefined composite gestures such as *tap and hold*

(equivalent to *click and drag* on a computer), or manipulation for rotation or scaling a hologram, as shown on (FIGURE 1.6).



(a) HoloLens *gaze* (b) HoloLens classical gestures

Figure 1.6: HoloLens basic interactions [3]

To be able to develop applications, some tools will be useful or highly recommended.

1.2 Presentation of development and modeling tools

1.2.1 Unity, multi-platform game engine

Unity [4] is a game engine developed by Unity Technologies, the first version of which was released in 2005. Its quick handling and its ease of importing a game on more than twenty platforms make it one of the most popular game engines. It also allows developers to implement applications for VR or MR devices mainly on C# [5].

The creation of a new Unity project brings up the editor which is divided into seven windows (FIGURE 1.7), described below :

- the menu bar: allows us to access all the game engine settings;
- Hierarchy: shows the tree structure of all GameObjects present in the scene;
- Project: shows the project directory containing all the folders, mandatory ones being Assets (containing all our 3D models, scripts, etc.), Packages and Project Settings;
- Inspector: specifies all the modifiable parameters as well as the Components relative to one item (GameObjects, Materials, etc.).
- Scene: displays the 3D space built for the game;

Figure 1.7: Windows of the game engine Unity

- Game: renders the scene from the camera's point of view.

The particularity of a game under Unity is its designed made of fundamental bricks:

- the *GameObjects*, corresponding to elements rendered in the game;
- the *Components*, corresponding to characteristics of the *GameObjects*, which are either offered by librairies or designed by ourselves via scripts.

The way Unity works is highly based on some central Components:

- to translate, rotate, scale a GameObject in the 3D space or access parent or child informations relatively to it, *Transform* Component is essential;
- to a GameObject to be receptive to Physics in a 3D world, *Rigidbody* Component contains basic Physics laws managed by Unity;
- to associate a mesh to a GameObject, apply a texture on it or obtain easily its bounds, *Mesh Filter*, *Mesh Renderer* and *Mesh Collider*) are used respectively.

It is possible to save the configuration of a GameObject with all its Components and settings attached : these objects are called *prefabs*.

To maintain functional code versions, Unity Collaborate (FIGURE 1.8) has been used, to the detriment of Git version control services, only used for sharing the code to the whole team within CSIRO via BitBucket and Confluence pages.

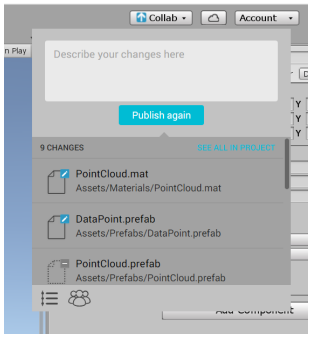


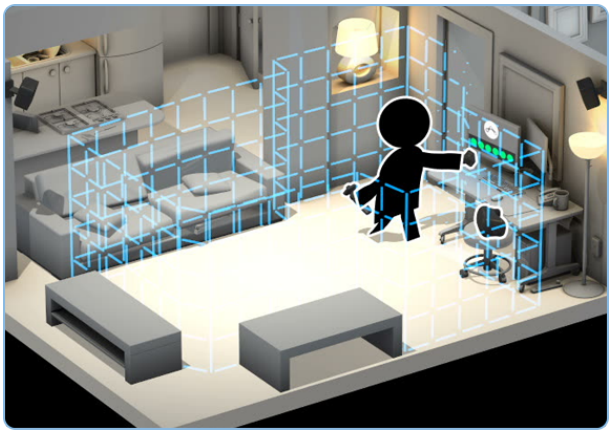
Figure 1.8: Window for pushing / pulling modifications

Microsoft Visual Studio 2017 has been used to code additional modules for *ImAxes*. It has high-quality tools, including a debugger, a must-have for codes of such magnitude.

Developing a VR application under Unity requires some elements to be installed.

1.2.2 Tools and packages of Virtual Reality

To begin with, SteamVR application is necessary if the user wants to live a VR experience on a Valve hardware. It includes the initialization of VIVE technology and the playing area has to be drawn (FIGURE 1.9(a)). The different Components of the technology appear in green in a small window if they are detected by the computer (FIGURE 1.9(b)).



(a) Step of delimiting the playing area

(b) SteamVR window

Figure 1.9: SteamVR

Another key tool is OpenVR, an open-source API offering a set of pure C++ virtual methods to override. It is widely used because it offers a simple interaction with a VR headset, whatever its brand. A code using OpenVR features can therefore be launched on any headset, without having to maintain a version of the code per hardware.

Finally, VRTK (for *Virtual Rality ToolKit*) project, also open-source, provides a set of scripts and prefabs which speeds up VR applications development, in particular by providing solutions for interactivity (with *controllers* and via buttons) or physics.

If we want to develop a MR application, we have to import different *packages* and setting different parameters in the menu bar if one do not want to be exposed to a considerable number of errors.

1.2.3 Tools and packages of Mixed Reality

Only the IP address associated with the hardware must be provided to Visual Studio to build the code on the device. Unity has implemented the holographic view feature, working either on the computer screen or on the device, by providing the IP address of the hardware. It saves a considerable amount of time during a test phase.

The central tool for MR development is an open-source project called MRTK (Mixed Reality ToolKit) and designed by Microsoft. It provides a set of methods, diverse prefabs (with useful Components, shaders, etc.) and features (FIGURE 1.10), allowing fast development of Unity applications on HoloLens or any other WMR headset.

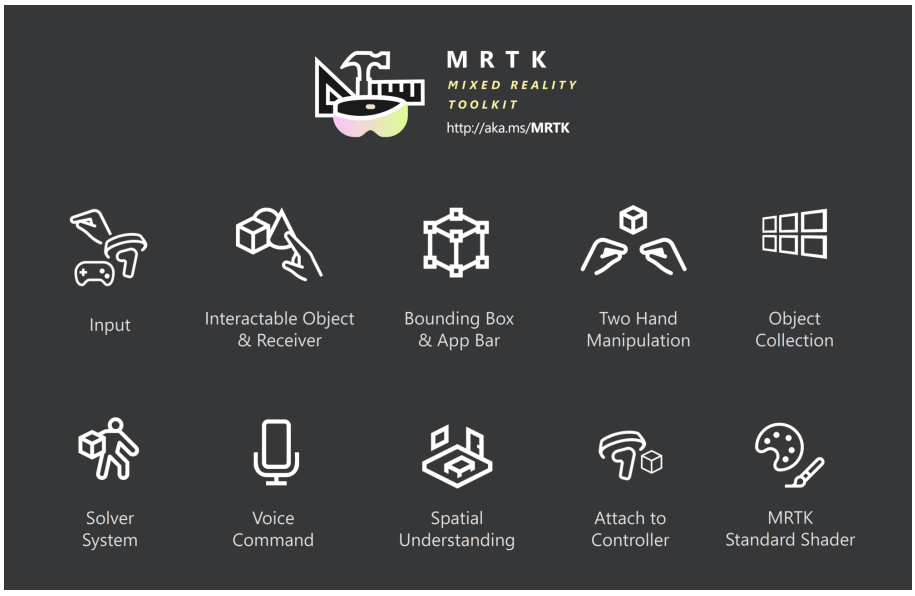


Figure 1.10: Panorama of MRTK features

Windows has launched HoloToolkit Academy [3], a series of projects using Unity, presented as *"fill-in-the-blank"* code exercises. These tutorials offer a first quality step to familiarize ourselves with the implementation of basic MR features for HoloLens v1.

Turning now to the visualization tools to which the project is based on.

1.3 State of the art of data visualization and exploration tools

1.3.1 ImAxes : interactive tool for multivariate data exploration

1.3.1.1 Presentation of the tool

ImAxes [6] is a tool for visualization and exploration of multivariate data in a VR environment. The motivation of this project has been to put oneself in the shoes of a typical user wishing to analyze an object by a set of dimensions that characterizes it. From a database, the application builds *histograms* associated with each dimension entered, as we can see in FIGURE 1.11.

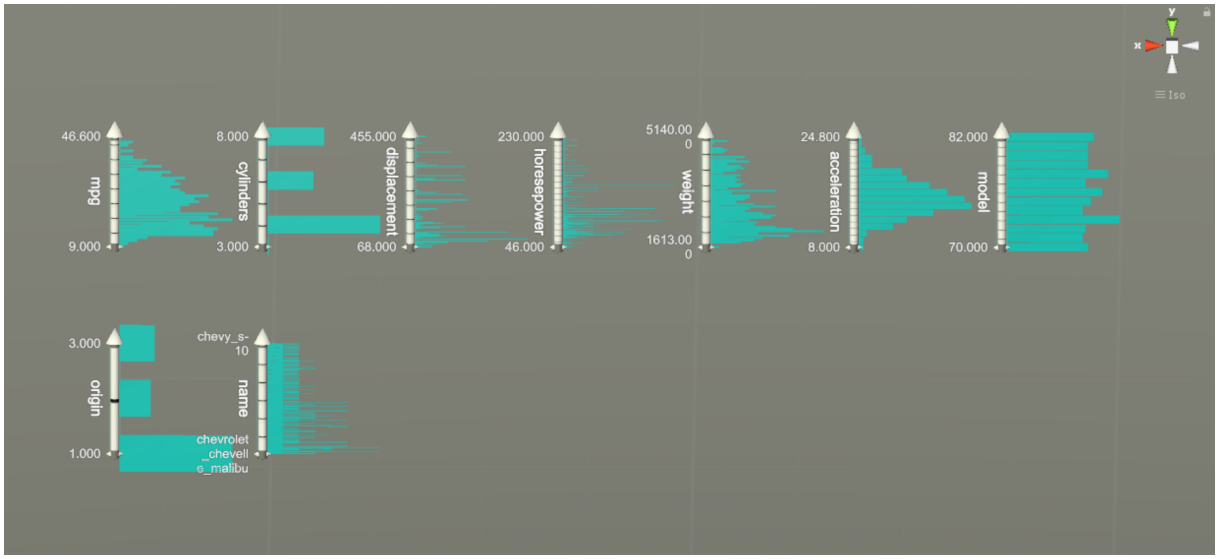
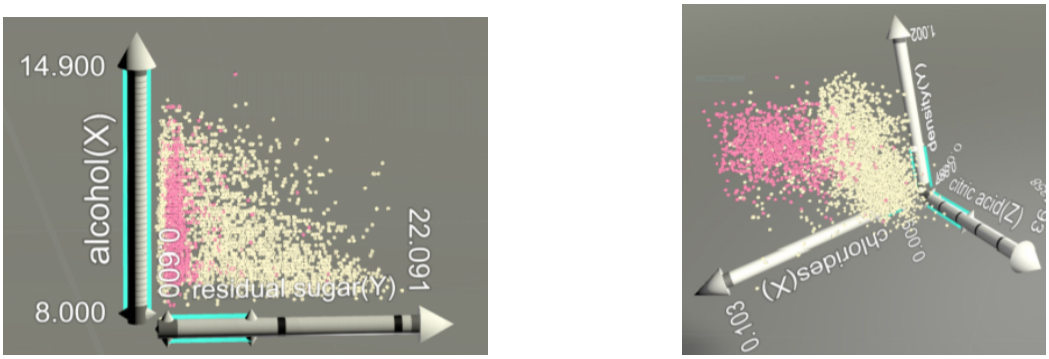


Figure 1.11: Set of histograms visualized when the application is launched [6]

ImAxes is a tool that is intended to be interactive: by pulling the trigger of a *controller*, it is possible to grab a histogram, which is cloned for further analysis. One can easily manipulate the histogram by holding it in its center, throwing it away, etc. If a second one is placed perpendicular to the previous one, a *2D point cloud* is formed (FIGURE 1.12(a)).

If we have a third axis orthogonal to the plane formed by the two previous axes, we can visualize a *3D point cloud* (FIGURE 1.12(b)).



(a) Construction of 2D point cloud

(b) Construction of 3D point cloud

Figure 1.12: Construction of point clouds [6]

The way axes are merged to build new visualizations entails orthogonal calculations, done via grammar rules. When an axis touches another one, a "physical skeleton" has to be coded so that new data visualization could be built accordingly. This has been translated into a list of rules describing each possible orthogonal or parallel axes configuration.

The other type of visualization proposed is called *parallel coordinate representation*. By grabbing one axis close to another in parallel, fine straight line connections are created for each correspondence in the plane formed by the two axes (FIGURE 1.13).

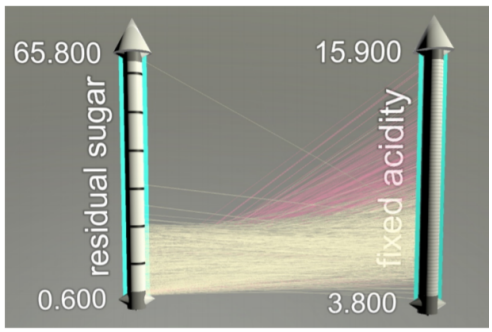
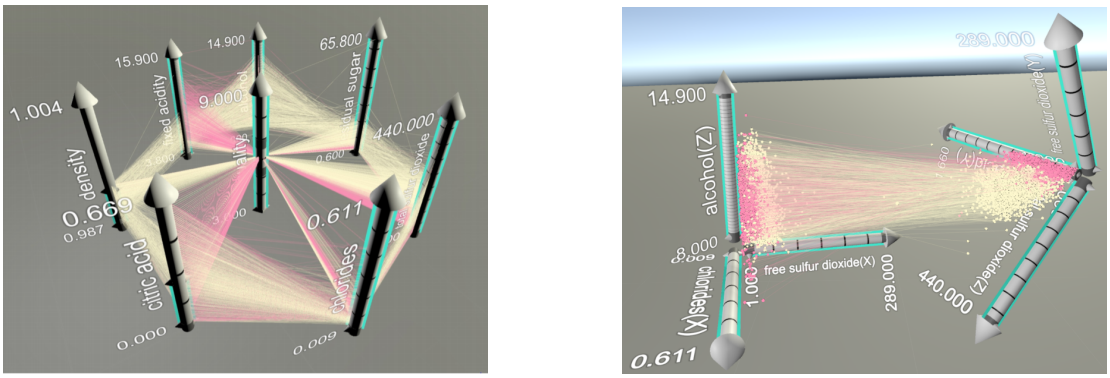


Figure 1.13: Parallel coordinates representations [6]

Combined in a circular way (FIGURE 1.14(a)) or with 3D point clouds (FIGURE 1.14(b)), this less known visualization brings out new ones, discovered by the researchers while not anticipated.

The same team of researchers with some others have developped another tool called IATK. You could find a presentation of it in APPENDIX 2, because it could be a perfect tool to combine with ImAxes.



(a) Circular representation

(b) Mixed representation

Figure 1.14: Emergent representations [6]

Finally, approaching a *controller* nearby an histogram pops out two types of sliders: the diamond-shaped one makes it possible to adjust the scale of each axis whereas the other one works as a cutting line (FIGURE 1.15).

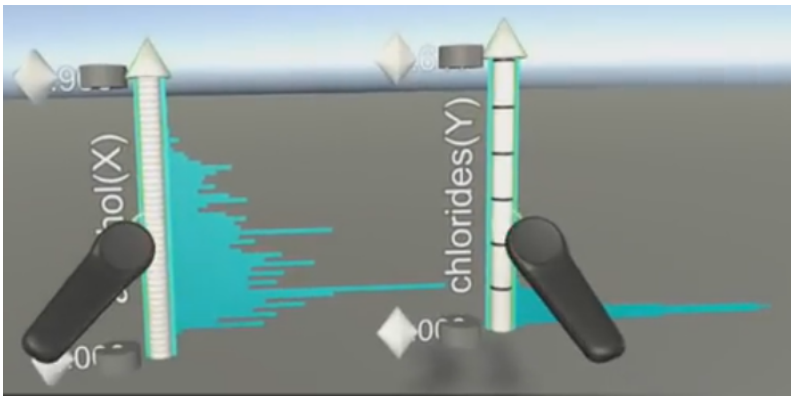


Figure 1.15: The two types of existing cursors

1.3.1.2 Contributions

In order to get started with the tool, some basic C# methods have been implemented to calculate, from a series of data extracted from the database, the variance and standard deviation. The data series actually corresponds to the abscissa of the histogram for a given dimension, and bars length represents the number of times the dimension occurs in the stipulated range of values.

Then, these values has been displayed via a basic Unity panel, appearing when grip button (for VR) has been pressed or *air-tap* gesture (for MR) has been performed on the histogram in question. The panel displaying time has been set to three seconds for each panel that has been requested, using `WaitForSeconds` Unity method to declare and start

a timer. One of the difficulties encountered was to implement where the panels should appear if several panels had been requested, without any overlap. The solution found has been based on panel *colliders*.

Initially, the objective was to adapt this tool for MR. However, a few weeks were enough to notice the difficulty of such a conversion, because all features available through the use of *controllers* must be translated into features to be triggered when certain actions were performed and some words spoken. A fast import allowed us to obtain a static image, e.g. an non-interactible application. Besides, since the use of HoloLens1 headsets is limited to one gesture, its computing and rendering power limited and HoloLens2 technology was not available at that time, we preferred not to pursue this further.

Now let's see how we will link this project to a map of Australia.

1.3.2 MapsSDK-Unity : tool for visualizing data on a world map

MapsUnitySDK [7] is a tool designed by Microsoft to visualize data on maps. Depending on the demo we are running (FIGURE 1.16), the features are different.

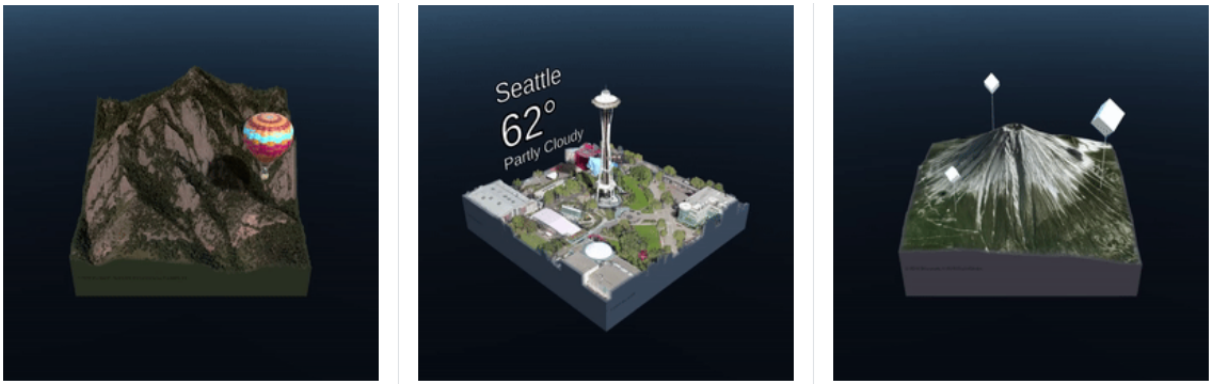


Figure 1.16: The different demos of the tool [7]

The part of the tool that interests us uses 2D Bing maps, whose precise location in longitude / latitude has to be provided. The desired localization is rendered with a relatively high degree of detail, but the rendering is not immediate (streaming can take a few seconds before updating). Using the *controllers*, it is possible to rotate the 2D map, but also to zoom in on a particular area. An illustration is provided FIGURE 1.17.

Pins are placed according to locations read in the database. They are displayed in a quantity that goes with the degree of zoom used (national information will be shown if we

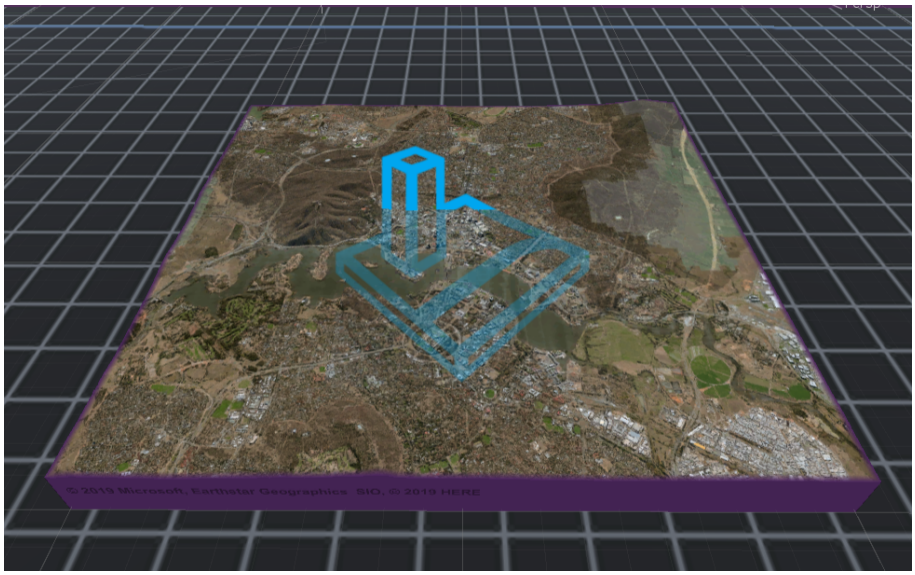


Figure 1.17: Scene obtained after indicating the Bing key, longitude and latitude of Canberra

visualize the whole country, and regional information if we visualize one state). However, details are not provided for any pins and could also lead to a world-scale map. Indeed, if we have a big database, chances are that at least one would be located in another country, causes a wrong scale.

In order to become familiar with the project and to overcome the lack of this demo, a database (temperature variations) has been provided to the application. A set of panels similar to those implemented for ImAxes have been displayed, showing name of the prescise locations as well as characteristic values of the chosen dimension.

In order to study energy data, some datasets available on the net have to be provided to the previous projects.

1.4 The datasets used

Although the elements presented below are visualization tools, we will only consider data they contain.

1.4.1 AREMI, data visualization in Australia

As indicated at the entrance of the website, AREMI (for Australian Renewable Energy Mapping Infrastructure) [8] is a platform containing a large spatial data on renewable

energy in Australia. AREMI is financed by the National Agency for Renewable Energies. As for its design, it was partly developed by Data61, based on the NationalMap project that the Australian government had previously initiated. An example FIGURE 1.18 gives you an idea of the interface.

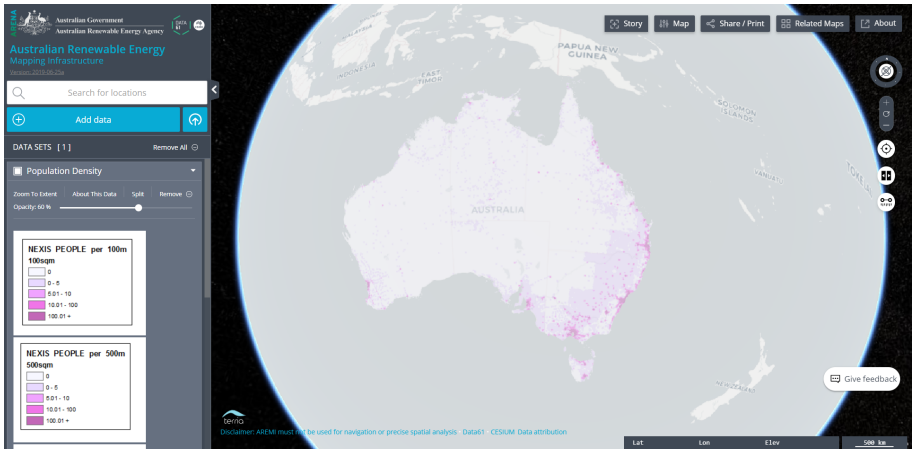


Figure 1.18: Population density in Australia (study provided by AREMI) [8]

One just has to indicate a location in Australia, then add as many databases from the catalogue as wished in order to visualize (by points, color gradients, etc.) the dimension on the map in spatial view. The database is rich, and sorted by theme (energy infrastructure, topography, population, etc.). For each database, a brief thematic and technical description is indicated, the organization that hosts the data accessed by the platform, and possibly the url addresses or Excel files if applicable.

The NEAR program will be complementary in a sense it will offer some different datasets and studies on them.

1.4.2 The NEAR program

The NEAR (Australia's National Energy Analytics Research Program) **??** is a platform prototype that aims to collect energy data, research and reports from all sectors and make them available across Australia. The principle has already been exposed in Introduction FIGURE 5. It is above all an interactive data visualization tool, allowing to considerably reduce the amount of data that is usually found in *Big Data* Excel files. When we enter the platform, a list of studies is available, based on datasets, and provides different studies in the form of graphs mostly, as in FIGURE 1.19.

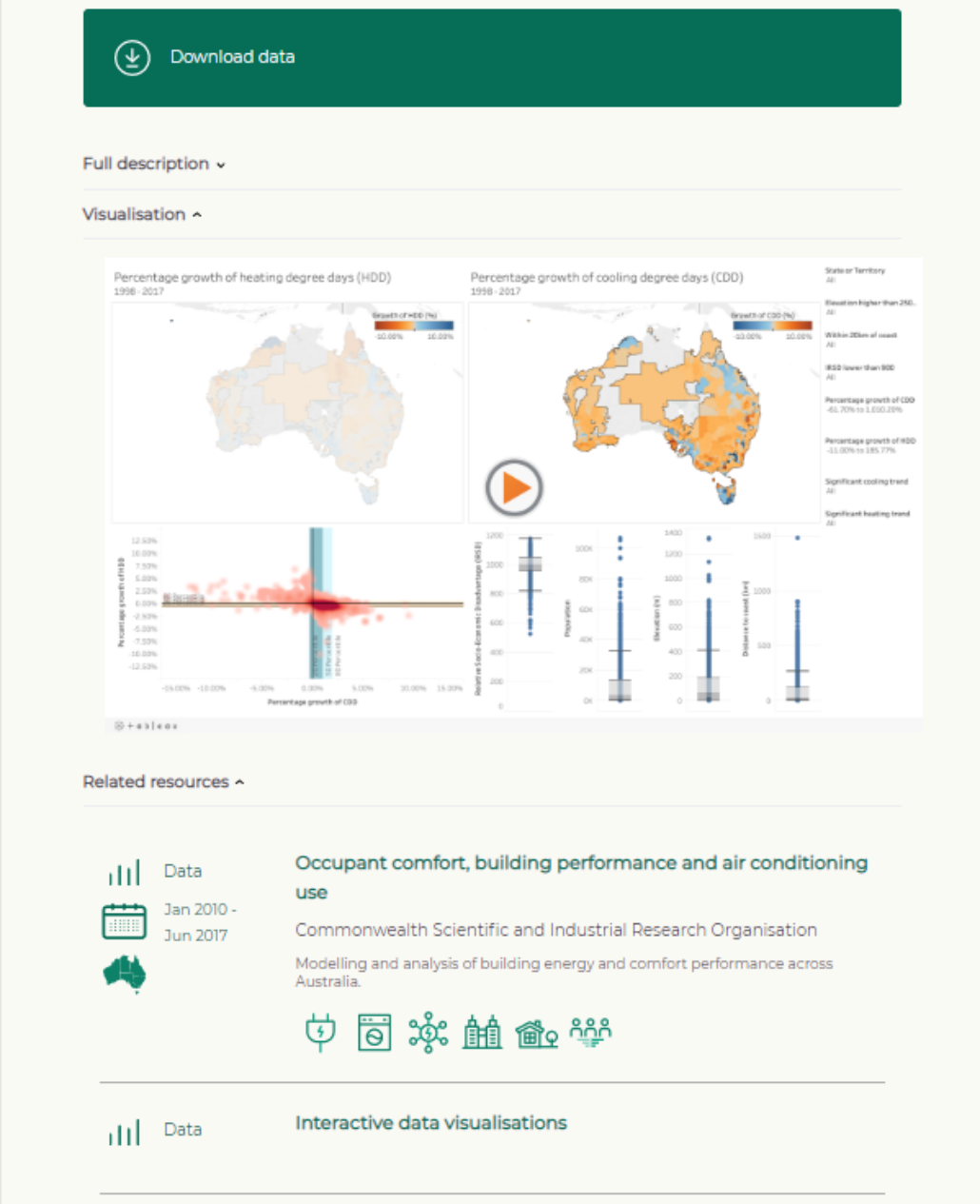


Figure 1.19: Heating and cooling needs in Australia (study provided by NEAR) [2]

Energy consumption data is predominant within this tool, and in this regard, NEAR has developed predictive tools to estimate electricity consumption per building. To do this, many studies have been conducted on the efficiency of solar panels or on demand rates by region to make the platform complete. The objective of these studies was to deduce the typical profile of a consumer.

Let's now see how a collaborative solution has been put in place inside Unity.

Chapter 2

Construction of a collaborative environment in Virtual Reality

2.1 Development of a networking application under Unity

2.1.1 Overview of the different networking possibilities

The selection criterion in order to choose the most suitable multiplayer solution for our application is its complexity. Generally, basic methods like connection, disconnection, message sending or receiving just use low-level API (LLAPI). But in our case, we want to assign particular messages to particular players or synchronize states, then complexity rises and high level API will be required (HLAPI). All HLAPI methods uses LLAPI ones to work. In light of what we need, the most famous HLAPI solutions are:

- UNet, based on Unity LLAPI and using Unity Cloud. Considered insufficient in terms of performance by the developer community, a new layer will soon be created, which has devalued UNet;
- PUN (for *Photon Unity Networking*) [9], based on Realtime LLAPI and using Photon Cloud. Photon is an independent, multi-platform network game engine. A version 2 has recently been released, which deprecated the 1;
- develop our own network architecture, independently of Unity, but the considerable time that would have been required was unreasonable for a 6-month internship.

PUN has been chosen for performance concerns. Indeed, although both solutions use server / client architecture, PUN supports in addition *peer-to-peer* sending of messages.

So two players can exchange data only by passing only through a relay server instead of having to pass through the host as UNet imposes, as illustrated on FIGURE 2.1.

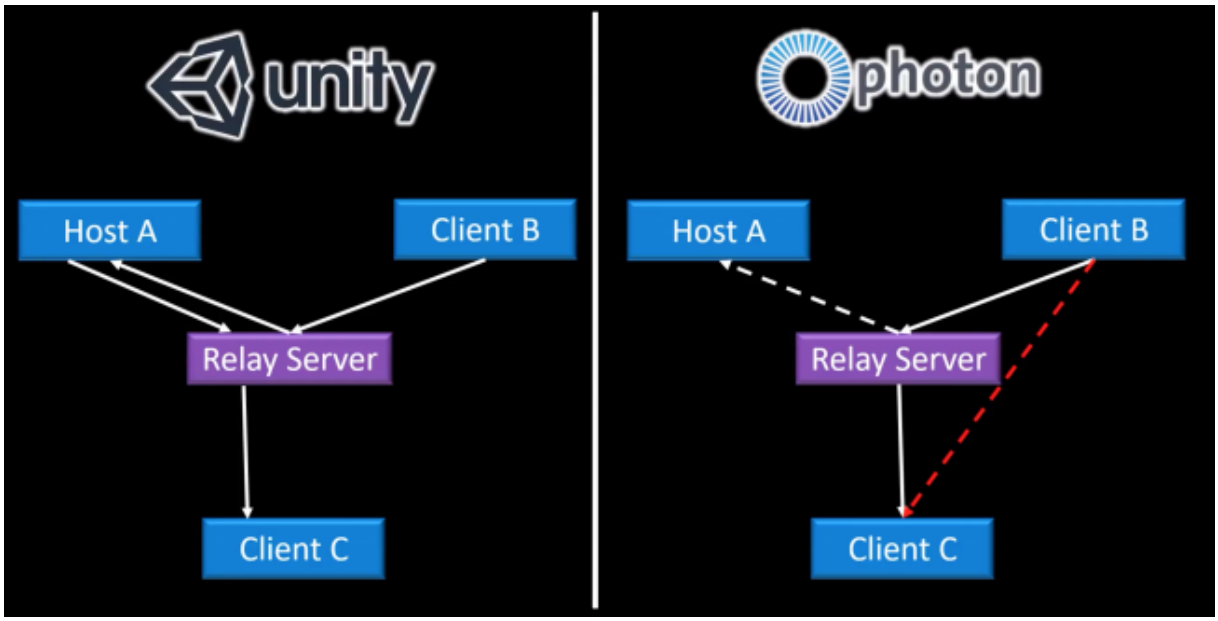


Figure 2.1: Network architecture comparison between UNet and PUN

Photon v1 was chosen for its ease of use and the tutorials that flow on the web. Concerning v2, Photon mechanics have been deeply modified and the documentation is poorer for the moment.

Throughout this project, it has been necessary to take a closer look at PUN fondamentals.

2.1.2 Fundamental concepts of PUN, API for networking development under Unity

The fundamental element that any GameObject must have in order to be viewed by all players is the *Photon View* Component. When attached to a GameObject, it enables each player to be associated with an identifier, unique on the whole network. This identifier is associated with the GameObject as soon as the application using the Photon server is launched.

Various parameters are available in this Component, the most important ones being:

- Owner, setting the degree of ownership transfer over the object in which the Component is attached. Possible options include Fixed (no ownership transfer possible),

Takeover (transfer via use of Photon methods) and Request (transfer via Photon methods overload);

- Observe, offering different modes of data transmission through pipelines. Only the OnReliableOnChange option will be used, allowing data to be sent only when there is a change between two updates.

Let's detail the key steps to enter a game room under Photon (see FIGURE 2.2). The first stage is the connection to a *lobby* listed on the Cloud. It allows you to consult all the available rooms you can reach. Photon has servers on serveral regions (including Australia) distributed across multiple hosting centers so that latency is minimized. Once this connection done, the creation of a *room* will be required. Its role is to host the multiplayer game taking place in a Unity scene where players will interact with GameObjects. The latter are qualified by Photon as *SceneObjects* because the Owner is initially the scene.

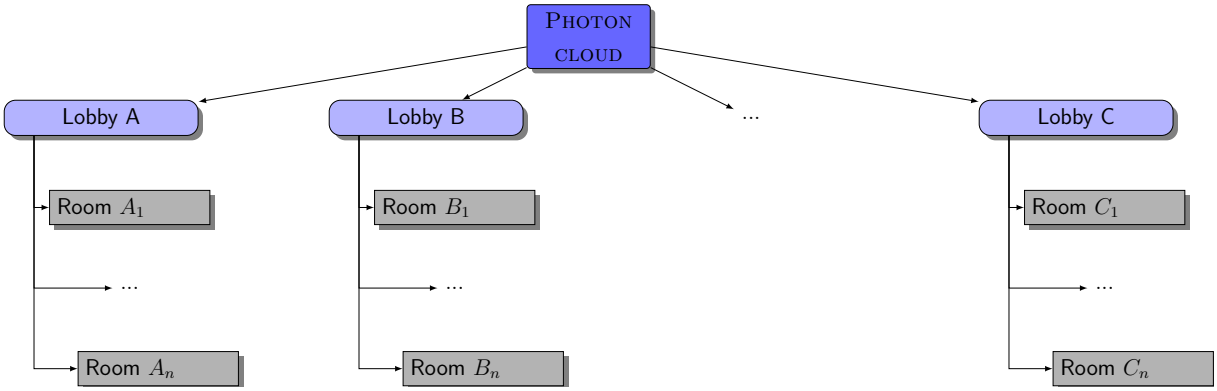


Figure 2.2: Lobbies and rooms in Photon PUN

The first to be physically present in the scene is a fortiori the user who creates the room. It is called *Master Client* because the only way clients connecting the room after him could sent information to the Cloud will pass through him, as illustrated in FIGURE 2.3. There is naturally one per room and it is thus given the network logic of the game. The information send will be movements coordinates for instance, and must be send to the server so that Photon Cloud send updates to all clients. According to the desired update frequency, messages should be sent to the Cloud by using :

- `OnPhotonSerializeView` method for information that needs to be constantly updated (simulate continuous movement, etc.);
- a callback method called `RPC` for punctual or constant information throughout the game (player name, fixed color, etc.). A RPC must be applied to a particular GameObject, which it targets with the *Photon View* Component;

- `SetCustomProperties` method allowing to add a property to the cube by using a HashTable.

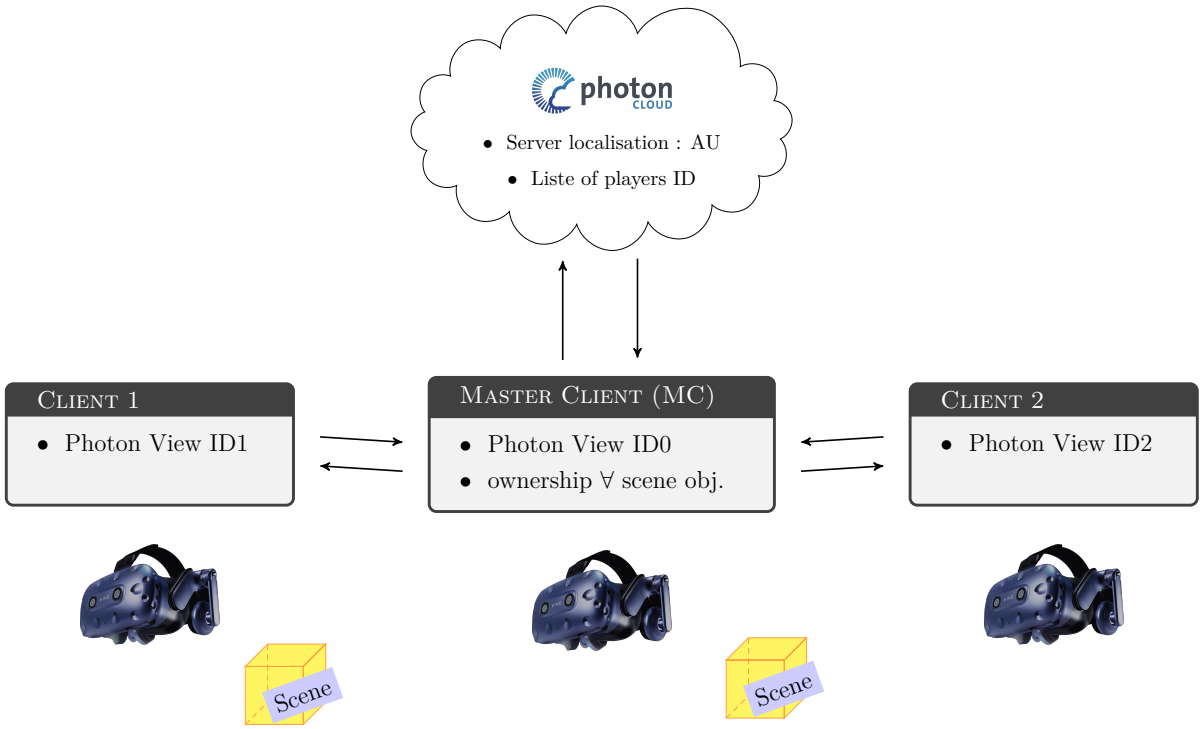


Figure 2.3: Ownership and Cloud updates in Photon PUN

Other elements will be discussed in the next section, if necessary. Let's now look at the global Photon architecture that has been used.

2.2 Used architecture and implemented features

We remember that the objective is to provide a collaborative solution to ImAxes. But expecting to convert this project directly is very complicated, because it was not designed for this purpose initially. So a new project has been opened.

2.2.1 Sequencing of the scenes

In a similar way to online video games, the application can be divided into two parts, as shown on FIGURE 2.4. The first step is the initialization of the Photon network, within a scene dedicated to its initialization. Each player must enter a player name and then join or create a room. This scene is local to each computer that will run the application, more details of which can be found in section 2.2.

The next scene is the game scene, which contains a set of GameObjects. It is then very common to load a separate Unity scene via the `PhotonNetwork.LoadLevel` method, allowing all users to load the same scene in which they will interact. This scene exists if at least one player creates a room. In PUN architecture, when the host leaves the room, its power is automatically confered to another client, so its lifetime will be that of the last client leaving the room. This scene is detailed in sections 2.3. and 2.4.

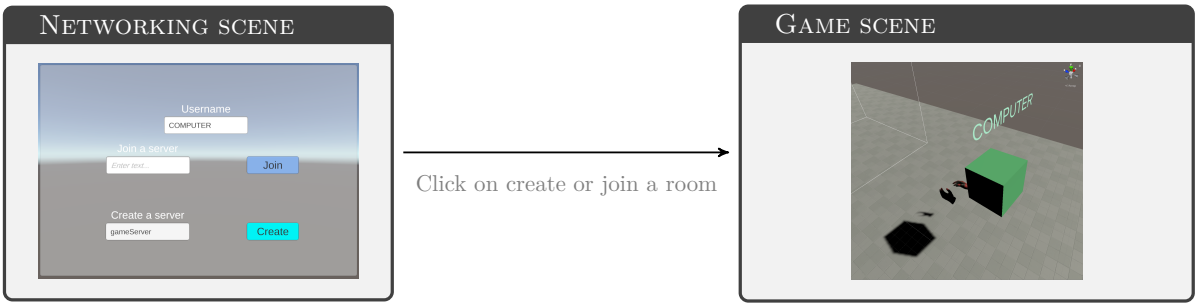


Figure 2.4: Bloc diagram of scene sequencing

Let's present the user interface launched in the first scene.

2.2.2 User interface of networking communication

The first step, in order to converge towards a collaborative application, is the connection to a room. It consists in informing the user of the evolution of the connection to the Photon server by a series of panels (FIGURE 2.5). To make all these different views following each other, it is necessary to enable or disable the corresponding panels, by the command `setActive` with true or false as unique argument.

(a) Connection in progress

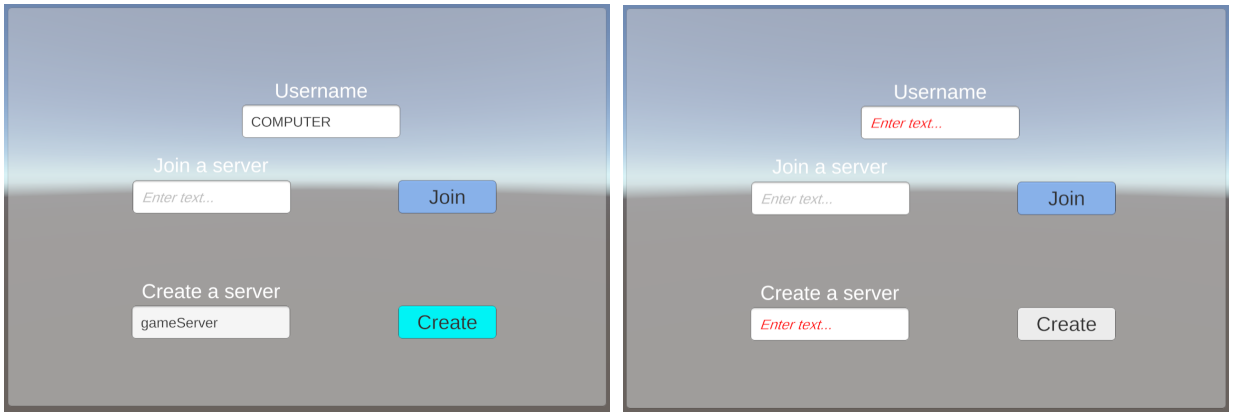
(b) connection failure

(c) Successful connection

Figure 2.5: Possible connection information

Once the connection is completed, an interface appears (FIGURE 2.6(a)), allowing the client to indicate the username he wants, then either to create his own room (by giving

him a name) or to join an existing one. The names provided must be greater than 1. If not, the subtext "enter text" will appear in red (FIGURE 2.6(b)) to inform the player of his/her clumsiness. Finally, all texts are displayed via *TextMeshPro* Components, of a higher quality than *TextMesh* Components in Unity by default.



(a) Menu to create / join a room

(b) Menu with red text error

Figure 2.6: Initial menu and errors indicated

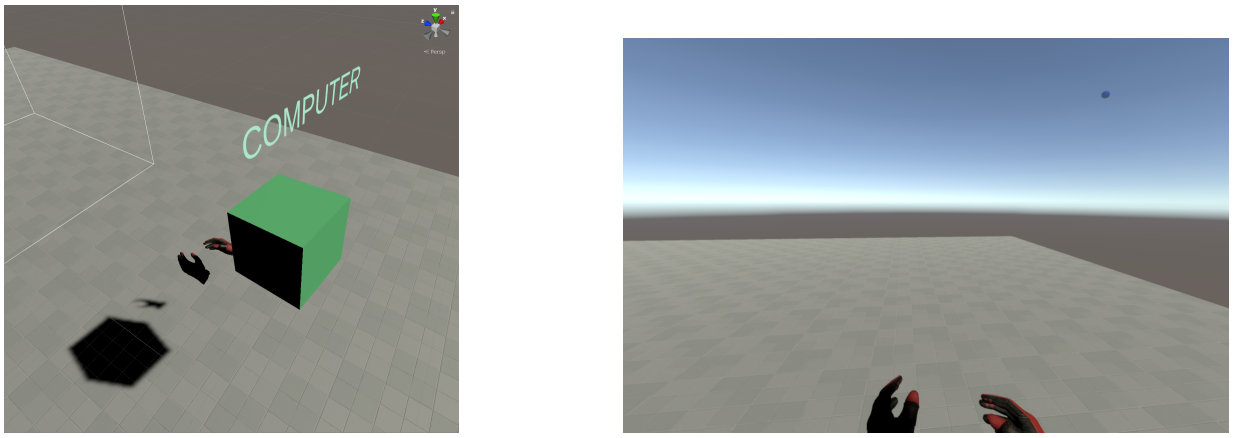
One of the downside is the blurred rendering when we launch the application, that has not been resolved yet. Another objective will be to create a drop-down menu to choose the room we want to join among the existing ones.

Once the room is created or joined, the player enters the 3D world in the form of a 3D model, that has to be explicited as well as its movements updates.

2.2.3 Modeling and behaviour of the different users

The aim is to physically visualize all the players in the scene when a room has been created. Thus, when a player connects to a room, an instantiation of body parts prefabs, roughly represented by a cube in place of a headset and 3D hand models for *controllers*) is performed (FIGURE 2.7). This instantiation necessarily passes through the PUN `PhotonNetwork.Instantiate` method so that Photon Cloud can instantiate them on the network and transmit this information to other players. These prefabs must be placed in the Resources folder.

The player's movements (resulting in the evolution of a position and rotation over time) is easily transmitted to others by attaching the PUN *Photon Transform View* Component to the corresponding GameObject, allowing the position, rotation and scaling to be sent continuously to the server as desired (it is possible, however, to code it ourselves via



(a) External point of view

(b) Egocentric point of view

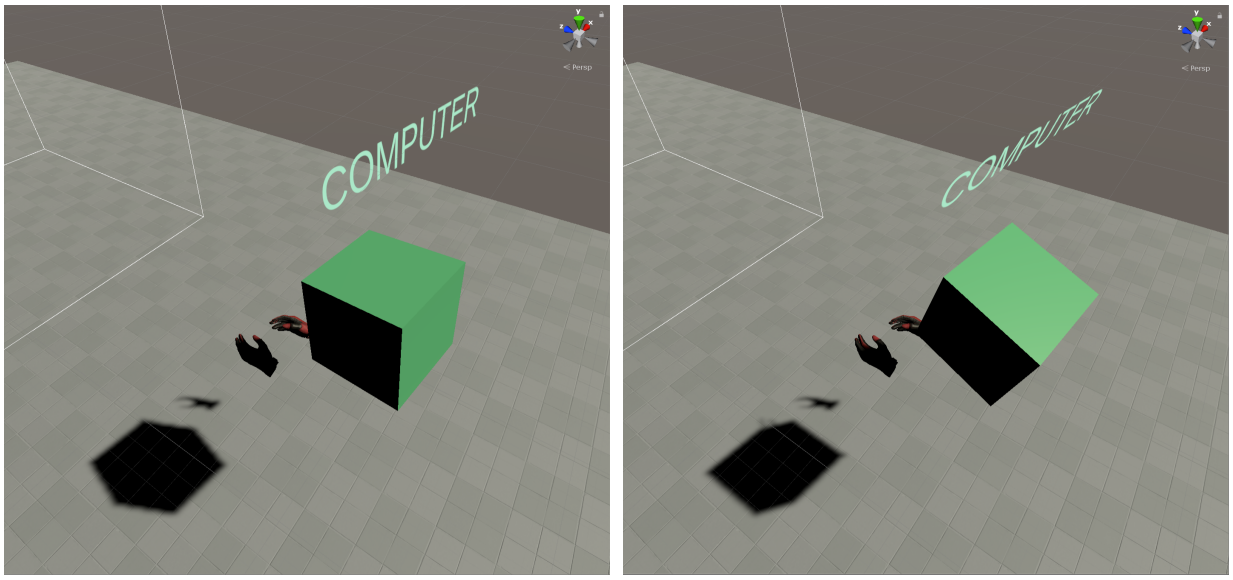
Figure 2.7: Different player modelling points of view

`OnPhotonSerializeView` method).

In order to distinguish the players, a random color has been assigned to them. As this colour was identical throughout the game session, a simple RPC has been sufficient to transmit the information. In the same RPC, the name of the user indicated in the graphical interface has been provided too. The command `isMine` is fundamental to code this type of functionality. Indeed, this keyword makes it possible to distinguish if the code is initiated by the player's computer or the computer of another one. So, to initialize the color of the other players in our world, we must make sure that the cube we color is not ours because the code running on the two computers connected to the two headsets is identical. The same goes for the player name, which requires a GameObject in each scene, in order to be transferred.

Finally, the orientation of the player's name in the game has been coded in such a way that the avatar can read the names of all his colleagues (FIGURE 2.8(a)), as well as his own name by looking up to the sky (FIGURE 2.8(b)). The orientation is calculated by introducing a difference of Quaternions (mathematical object representing a rotation in Unity) between the position of the player's name and that of the camera into the Unity method `LookRotation`. We put it in an `Update` method, to have the information each frame. These two GameObjects are found within the code thanks to `FindObjectWithTag` command, much less expensive than the `Find` command, its use being a bad reflex to take.

The whole point to code a collaborative solution is to see how each player could interact with GameObjects.



(a) When the player looks ahead (b) When the player looks up

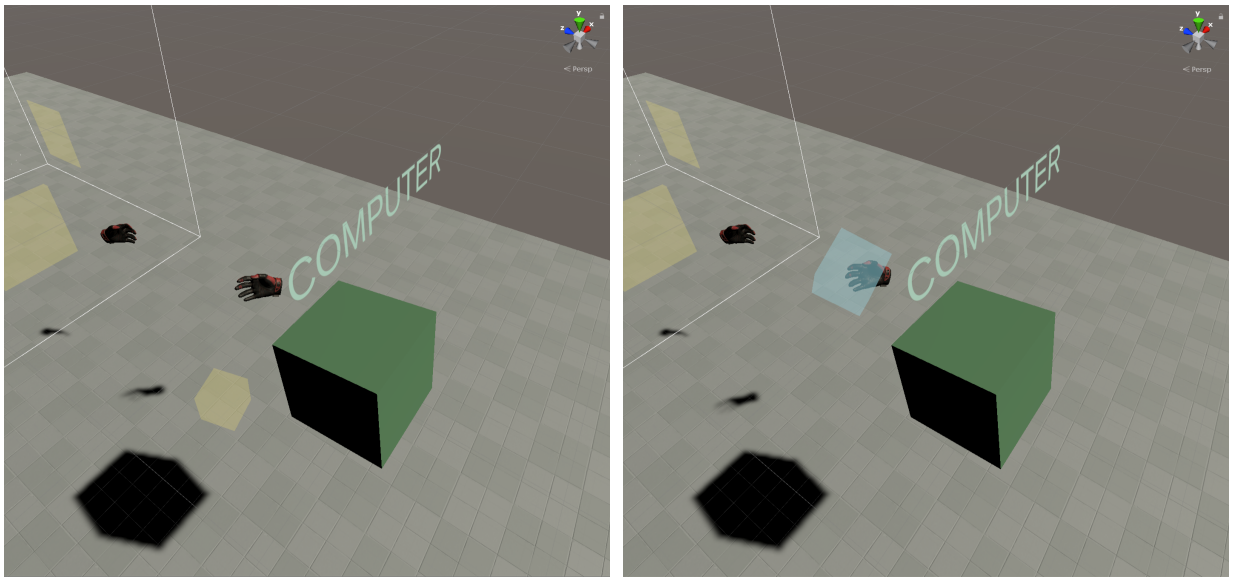
Figure 2.8: Orientation of the player name according to the player gaze

2.2.4 Interaction with the scene GameObjects

In order to avoid any cacophony, the *Master Client* will be the owner of all GameObjects as soon as he enters the scene he has created. If a client who subsequently arrives wishes to interact with one of these, he must send a request to the owner via the function `RequestOwnership`, automatically accepted by transferring his ownership via the method `TransferOwnership`. The *Takeover* option has been chosen for each *Photon View* Component, which means that a player cannot refuse to loose its ownership over a GameObject if requested by another player.

Interacting with a GameObject (here a cube) means being able to take it by hand and manipulate it as we see fit using the *controllers*. The cube is colored differently as soon as both cube and *controllers colliders* intersect (see FIGURE 2.9). The *collider* of each *controller* is a sphere located at their respective ends, and only GameObjects tagged *CubeInteractible* are receptive to *controllers* in the shared world.

However, cube anchor point was always the same (its center), and its movement when grabbed was very slightly jerky at times, sometimes starting from our hand towards infinity for one or two seconds. The origin of this inconvenience was the initialization in the `Update` method of the parent's object's position to the *controller* one. The use of `MoveTo` method brought a much more fluid handling of the grabbed object and a more realistic gravity when object is launched. Moreover, anchor points possibilities are infinite and very precise now (FIGURE 2.10(b) and (c)), instead of unique center anchor point before (FIGURE 2.10(a)). Quite often, we realize that a Unity function has already been



(a) Cube and hand *colliders* don't intersect (b) Cube and hand *colliders* intersect

Figure 2.9: Cube color switching according to *colliders*

implemented for many situations, and much better than what could have been done in many lines.

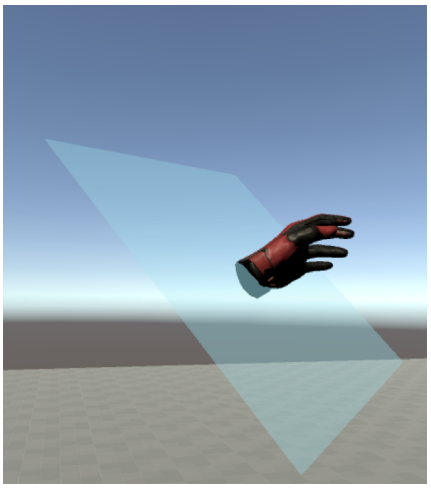
To improve the manipulation ease of the SceneObjects designed for analysis (planes, described on Chapter 3), when a player grab one of them, its gravity is activated and its kinematics disabled, and contrary is done when the same player let it. So planes will not automatically fall onto the floor when a user let it fall, and collisions between analysis planes will not result in weird reactions.

The cube is also subject to Unity's laws of gravity, which are at the root of a long misadventure.

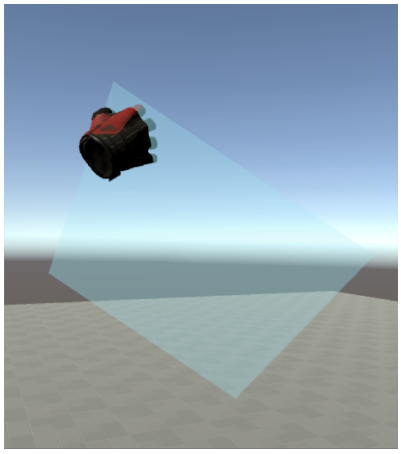
2.3 Unexpected difficulties encountered

2.3.1 Physical simulation of the GameObjects

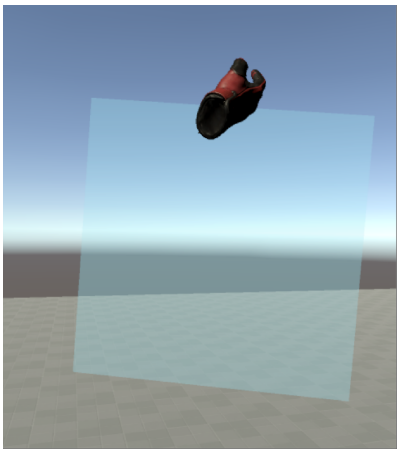
One of Photon's major current limitations is that it does not manage physics of the SceneObjects on the server itself (which would have allowed all users to enjoy a stable, unique and instantly updated physics). It is therefore Unity Physics module that is used. However, since each computer runs the same Unity application locally, physics management is also local, with its own temporality, built on the GPU's Nvidia PhysX



(a) Unique anchor point



(b) Anchor point at a plane corner



(c) Anchor point at a plane middle edge

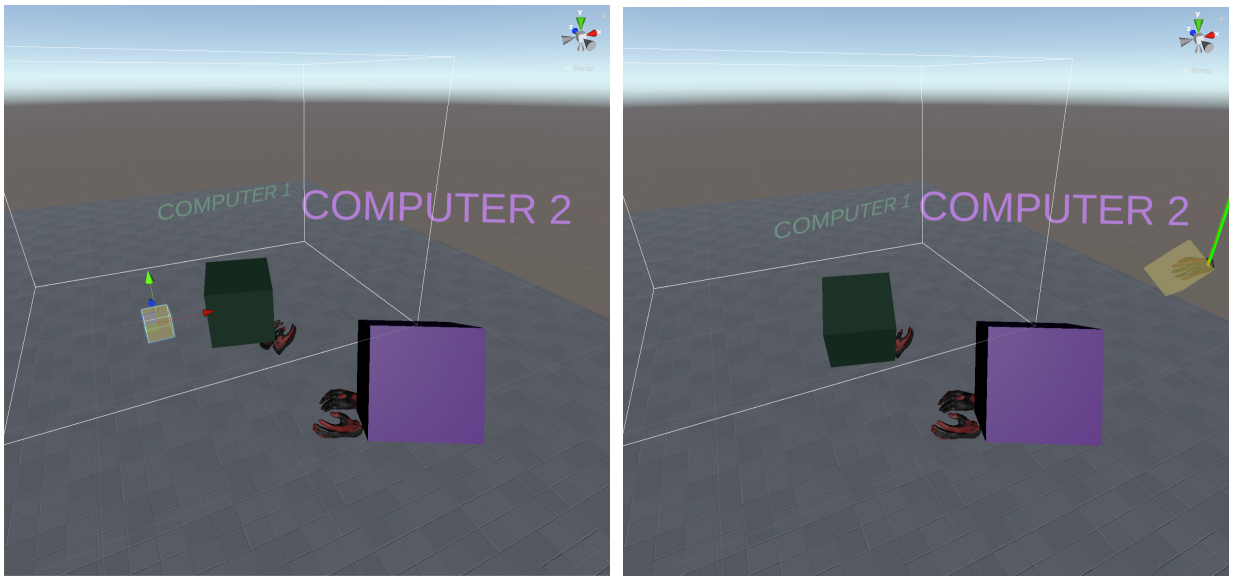
Figure 2.10: Difference of anchor points according to the function used

engine. That is why the engine is described as *non-deterministic*. But when a user wishes to grab a cube, its movement is subject to the physical laws of gravity, which must be updated for other users. And it is not possible to process this information as a player's movements for example, because it is not subject to particular physics.

A Component similar to *On Photon Transform View* exists (*On Photon Rigidbody View*) but too many deficiencies were visible, sometimes producing strange physical phenomena. Despite a certain fluidity, we could not see the displacement in height of a grabbed cube, only large jolts in height allowing us to see some evolution.

The first strategy tested was to build a 3D object without a *Rigidbody* Component, then to build a child object that had one. Its movement would be transmitted indirectly through the object without *Rigidbody* Component, which would be distributed all over the network. This strategy has been proved to be inconclusive: no movement was visible.

The second strategy was to transmit the numerical values that define a *Rigidbody* Component (which are velocity and angular velocity) in a Vector2D list to other users, then update them locally for each player. To do that, an overload of the `OnPhotonSerializeView` method (overview given on FIGURE 2.11) has been done. Then each player has been able to visualize continuous cube movements. However, lack of fluidity has been observed, that is why exchanging a cube like a balloon is not possible for example.



(a) Cube not grabbed (b) Cube grabbed by computer 1

Figure 2.11: Cube visualized on the computer 2 scene view screen

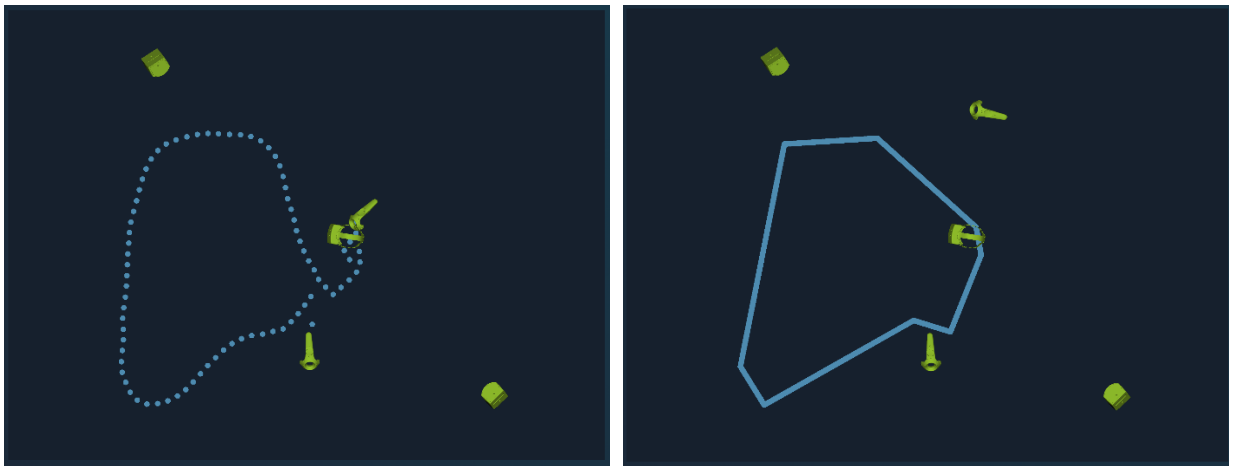
Apart from that, SteamVR has always been at the center of several issues, including the offset between each player's world.

2.3.2 Correspondence of the 3D world of each user

The other main difficulty was to simulate a world of unique geometry for all users. Indeed, as discussed at the beginning of the chapter, each player has his/her own HTC VIVE headset running on a dedicated computer. Then the loaded 3D scene is identical, but its delimitation is specific to each player because based on a geometry relative to the game area initialization drawn for the headset. This results in significant discrepancies between the visualization returned for the different players, where a player might see another one in front of him when he is actually behind him. In a nutshell, for this immersive application, an AR feature was desired (e.g. visualizing the precise position of another player within the world while putting the headset and vice versa).

One of the solutions that largely solved the problem was to draw game zones (required

when installing SteamVR) very similar for all headsets involved in the collaborative solution. SteamVR, after closing the loop (FIGURE 2.12.(a)), approximates the play area to a coarse geometric shape (FIGURE 2.12.(b)) : it only takes a few centimenters of differences to finally arrive at distinct areas accentuated by the clumsy approximation.



(a) Play area drawn (b) Play area approximated

Figure 2.12: Play area required by SteamVR

This solution allowed to obtain similar geometries for each headset, but some offset was still visible between *lighthouses* as illustrated in FIGURE 2.13. Consequently, one couldn't touch a headset by placing one's hand on its corresponding 3D modelisation in the virtual world.

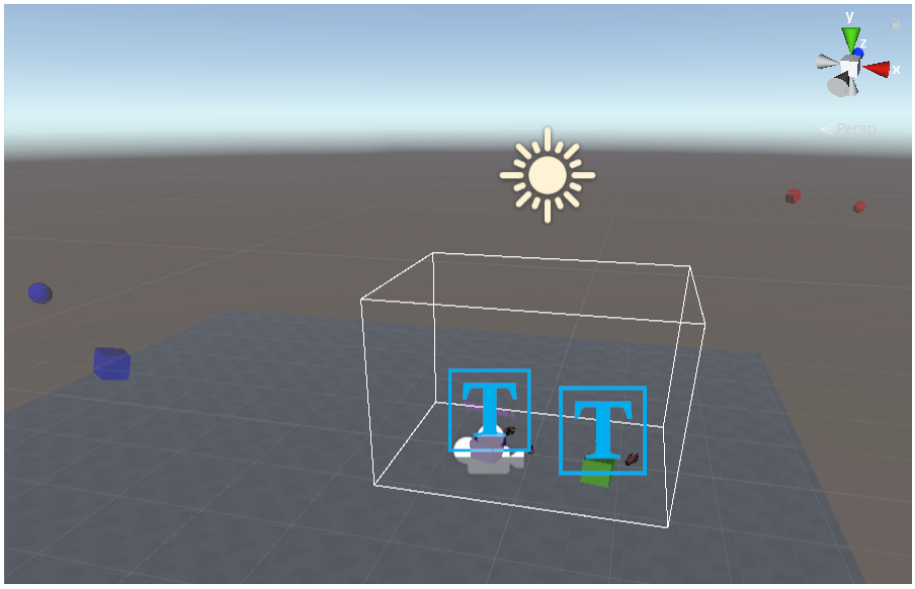
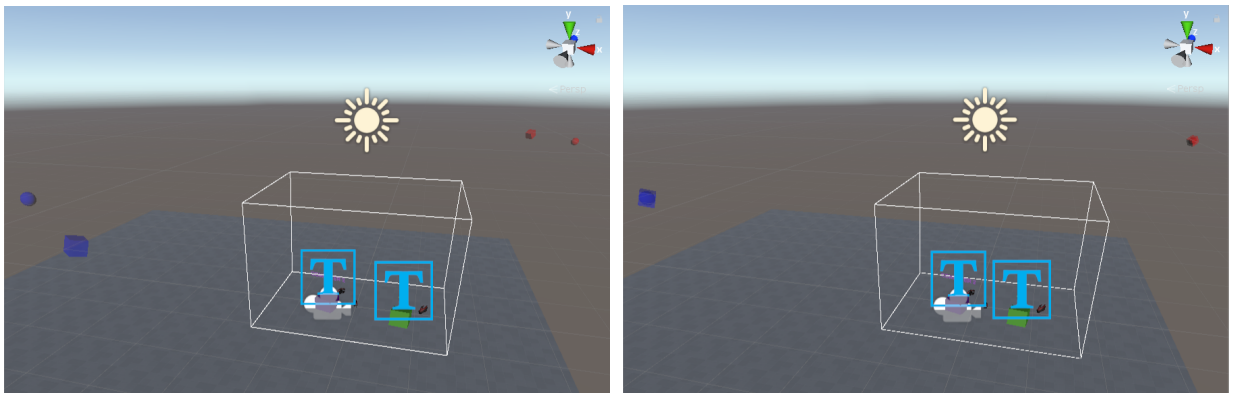


Figure 2.13: Offset persisting between cubes representing *lighthouses*

Research has already been conducted on this subject [10]. One of the goal was to

look for a solution adapted to a game taking place in a confined and closed space. They have worked on an algorithm that could predict a future collision, and modify the 3D virtual world of both players so that they physically deviate from their initial trajectory accordingly. As this research was not yet fully completed, it has been quickly dismissed.

A simpler solution has been chosen: it consisted in implementing a spatial correspondence program between the different local geometries on which the representation of the common scene was based. The *Master Client*'s world is the one taken by default (because the first to visualize the scene) and its geometry must be adopted by each new client. However, these calculations can only be made from a reference point, e.g. in the presence of a physical element having a stationary position in real space, regardless of how the headset is used. It is possible to create it ourselves but, in our case, it is easier to extract from SteamVR the position information of the two *lighthouses* fixed to the ceiling according to geometry drawn as FIGURE 2.12 shown. Every HTC VIVE headset in the laboratory uses the same two *lighthouses* actually. This information is only accessible via the serial number associated to each base station, which SteamVR initializes randomly per headset, except for one that does not vary (basis for the geometry). Thus, one base station will not be referenced by the same serial number depending on the headset considered and on the SteamVR room launched. Our only chance is then to compare the changing serial number in each world: if two serial numbers are the same for one base station, it means the stations have been identified the same way for the two worlds (case shown in FIGURE 2.14 with 3D models to differentiate which base stations pair belongs to, and colors to see if there is any pair inversion), otherwise they are opposite, and then shift of the client's 3D world is processed accordingly.



(a) Before world mapping (b) After world mapping

Figure 2.14: *lighthouses* position evolution

To finish, some equipment has been set up during the project, but brought more problems than anything else.

2.3.3 Wi-Fi equipment issues

The use of wires limited significantly the user's comfort in his / her movements during immersion. A few months ago, a device (shown on FIGURE 2.15) has been sold by VIVE, making the device wireless. Here are the devices requires, in addition to the Wi-Fi software to install on the computer:

- a Wi-Fi adapter to attach to the headset,
- a device composed of a WiGig box for wireless data communication at a speed of several Gigabits and a PCIe card (for *Peripheral Component Interconnect*) to be inserted into one of the motherboard slots so that the computer has an additional port;
- cables connecting the headset and battery to the adapter.

Figure 2.15: Components of an Wi-Fi wireless VIVE adapter

Tests were carried out for a month on a VIVE PRO and VIVE PRO EYE headsets but the device was quickly abandoned. Despite good fluidity, the sensors often took more than ten minutes to be detected and the battery lifetime was almost ridiculous (two hours at most). To restart the computer has always been the best solution to all problems.

The last section will expose how data visualization have been improved, in particular in a multiplayer environment.

Chapter 3

Collaborative data visualization rendering

3.1 Functioning of the graphics *pipeline* under Unity

3.1.1 Graphics rendering process and *shaders* principle

A *GPU* (Graphics Processing Unit) is responsible for the graphic layout of each pixel that makes up the screen. It has to pass through different steps to render an image, that constitutes the *graphics pipeline*. By default, the GPU identifies each object as a assembly of tiny polygons often called *primitives* that covers a bunch of pixels. A primitive is a connection of points called *vertices*, and can be described by its position, its normal, etc. We will only consider triangles because it is the most widely used primitive.

To have more direct control over the graphics pipeline and intervene in the default rendering, a program coded to specifically run on the GPU called a *shader* can be used. Among other things, it enables us to modify the geometry apparence of the mesh, light reflection or create special effects. A shader is commonly used by a *material* applied on an object. And a material requires a *texture* to be defined : a 2D image often called *sprite* that is frequently expressed in 3D model coordinates qualified as *UV data*.

A shader can be written in several languages, and supported by APIs providing a standardized set of functions allowing high accuracy in communicating data to the graphics card. These languages are :

- GLSL (OpenGL Shading Language), supported by OpenGL API,
- HLSL (High Level Shading Language) used by default in most of the game engines such as Unity or UnrealEngine4, and supported by the Direct3D subset of DirectX

API (whose pipeline is illustrated on FIGURE 3.1),

- Cg (C for Graphics), now deprecated, which compiler outputs DirectX or OpenGL shader programs.

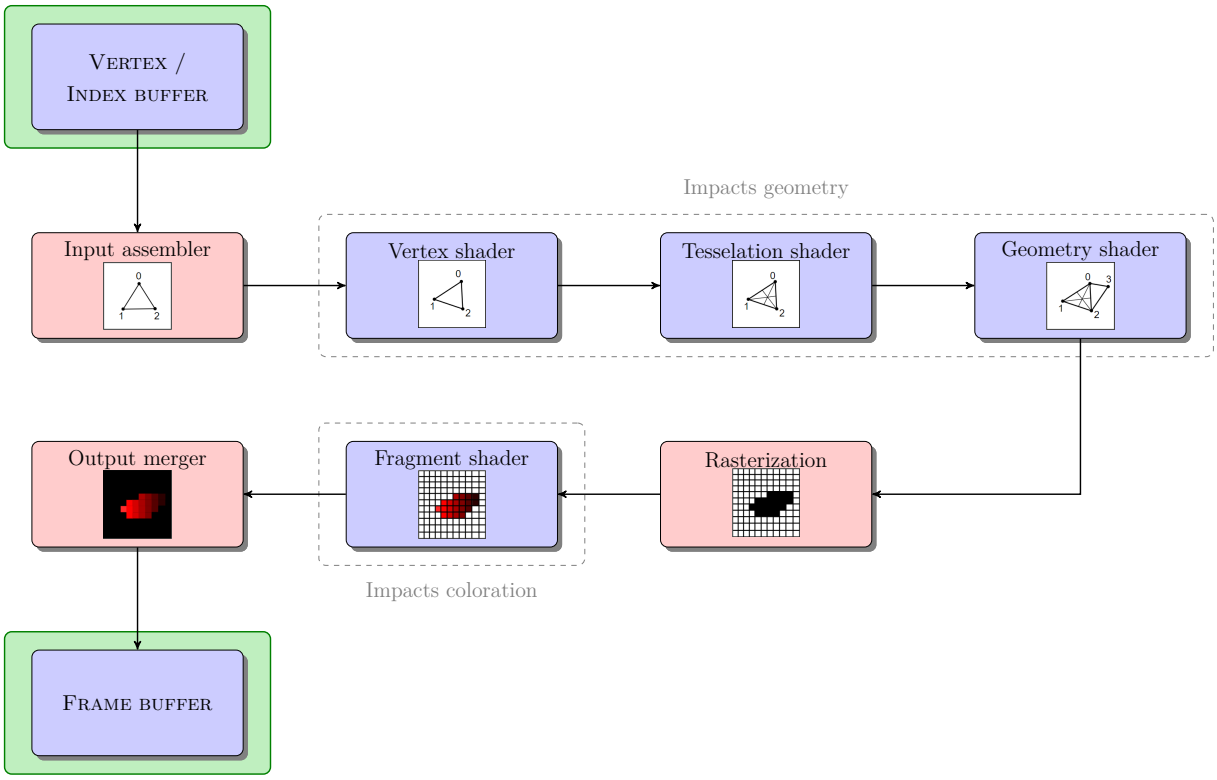


Figure 3.1: Direct3D Graphics pipeline

It is now time to introduce particular shaders and the language that formalizes the way to connect a shader to Unity.

3.1.2 Design of Unity *shaders* and particular ones

To be rendered, a GameObject has a Component *Mesh Renderer* containing at least one material, which can use a shader, as explained before. Materials can use at most one shader, but a shader can be used by as many materials as one want. The FIGURE 3.2 illustrates specifities of Unity3D.

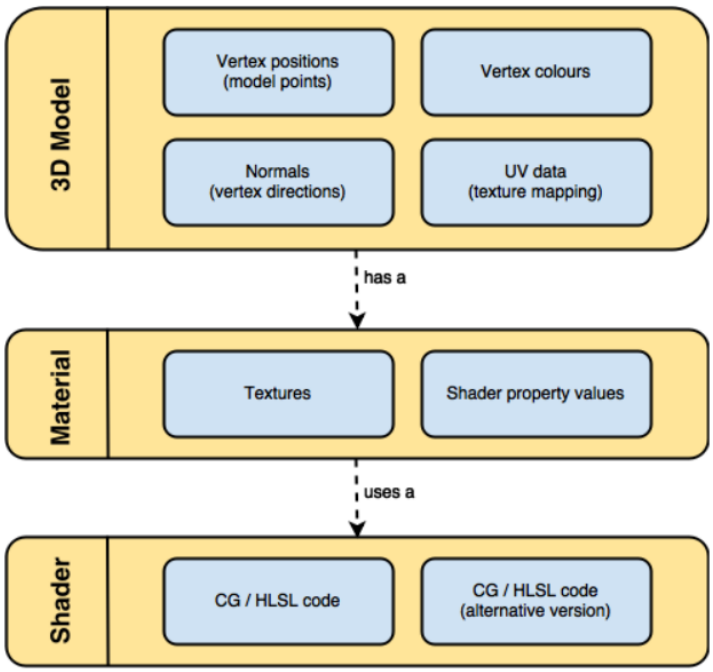


Figure 3.2: Unity3D rendering workflow [11]

All shaders in Unity have to be written by using a declarative language called ShaderLab. It acts as a wrapper that allows Unity to embed shaders more easily and manage their behaviour. ShaderLab imposes a very specific structure :

- a **Properties** section containing all real-time modifiable variables for a given material, viewable in the Inspector panel as for public class attributes for scripts;
- a **Subshader** section containing a line of setting up tags filled in as a key-value pair and the successive render waves called *passes* calculated on the GPU.

Actually, passes are very specific to Unity and must be understand as classical shader (as defined before) containers. So a Unity shader can accomodate several shaders.

Each pass contain a shader program embedded in CGPROGRAM snippets, containing at the beginning pre-compile directives, includes, and useful variables/structures/functions for it. One or more characteristic functions implements the shader, which must be stipulated in a macro at its the beginning, and vary according to the type of shader we want to implement. Unity supports several types of shaders, to be used according to our objective, the most used being:

- the *surface shader* if we want to focus on the realistic light effects of a texture,
- the *vertex & fragment shader* if the purpose is to deal with unrealistic special effects.

It is the latter type that we will use, illustrated in FIGURE 3.3. As its name shows, these type of shaders are composed of two main functions :

- *vertex* (in charge of the vertices definition) will tell the GPU at which position each point should appear. It takes as input a structure instance containing variables (position, normal and UV coordinates in our case) that can caracterize a vertex, and returns a structure instance containing all the variables (position clipped to the camera, color, world coordinates, etc.) calculated from the inputs.
- *fragment* (in charge of the pixels definition) will tell the GPU what color each pixel should take. It takes as input the output offered by the *vertex* function, and returns a vector with four Components representing a RGBA color.

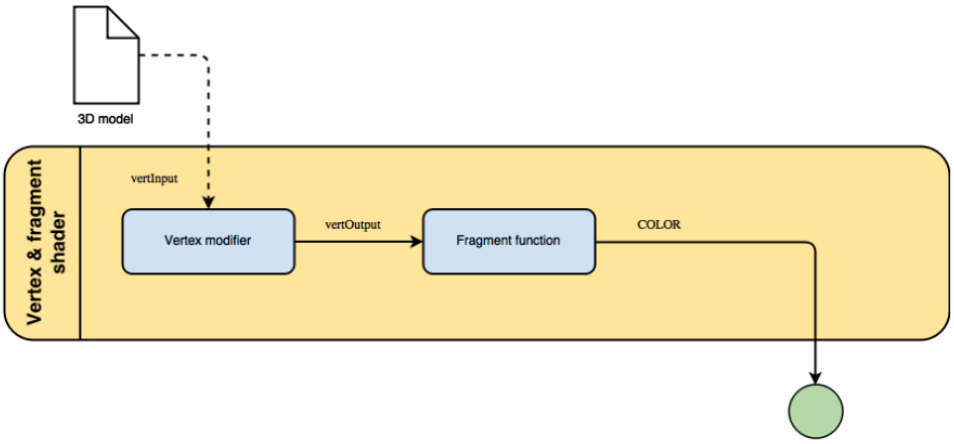


Figure 3.3: Unity3D *vertex & fragment shader* workflow [11]

The graphical pipeline also allows the implementation of more complex shaders, such as *tesselation shader* that enables us to subdivide a primitive or *geometry shader* whose principle is to generate new primitives by adding to points, lines or triangles new ones respectively [12]. We will dive into the latter one in the following section.

Some ways must be find then to supply shaders with energy data.

3.2 Graphical representation of data

3.2.1 Creation of a point cloud and rendering

Before launching head down into the rendering of a scene where several ImAxes graphics would appear, it was first decided to consider a 3D sphere with an applied texture that

uses a point cloud shader. This sphere is a non-draggable SceneObject at first. In the long run, the objective is to provide as an input of this shader, coordinates that would be calculated according to a database so that it can build the graphic shape of the cloud by itself.

By default, Unity uses traingles to represent 3D objects, whereas modern graphics hardware can mainly support triangles, lines and points (what interested us here) as basic primitives. The problem is that the appearance of points is not configurable in HLSL (in particular their size, near one pixel dimension). That is why a *geometry shader* is often implemented in order to convert each vertex of the concerned object into a concatenation of triangles primitives covering much more pixels which can then be seen by the player. The implementation consists in calculating in a *for-loop* the future triangle adjacent to the previous one thanks to a trigonometric circle (triangles represented in red on FIGURE 3.4).

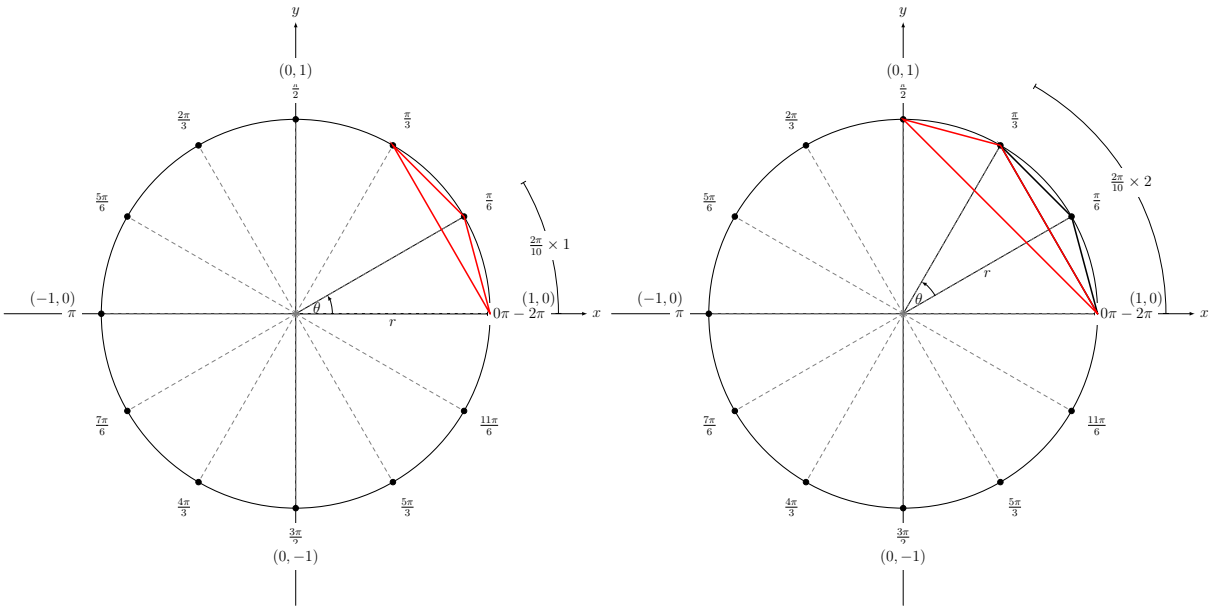


Figure 3.4: Computation of the two first polygon vertices

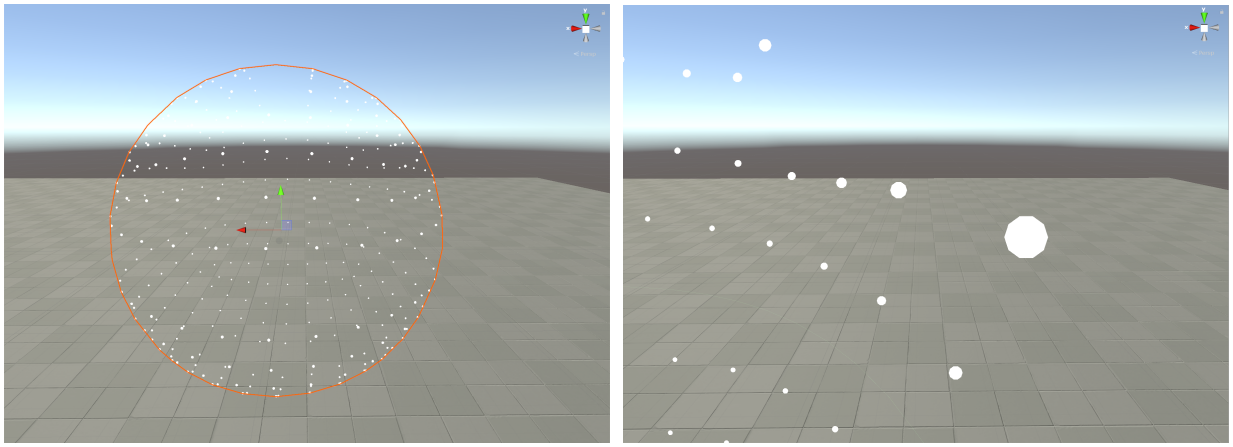
So, the formula used to compute the position of polygon vertices is:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} + \begin{pmatrix} \cos(\theta) \times xAspect \\ \sin(\theta) \times yAspect \\ 0 \\ 0 \end{pmatrix} \times polygonSize$$

where :

- $(x'\ y'\ z'\ w')^T$ are the coordinates of the vertex to be computed,
- $(x\ y\ z\ w)^T$ are the coordinates of the previous vertex,
- θ is the angle ($\equiv 0 \left[\frac{2\pi}{polygonNbSides}\right]$) formed between the two vertices inscribed within the trigonometric circle,
- $xAspect$ and $yAspect$ are parameters relative to screen size,
- $polygonSize$ is a magnification factor.

So, in our case, the simplest solution is to build a polygon with enough edges for the human eye to assimilate each point of the sphere cloud described just before (FIGURE 3.5(a)) to a round shape (FIGURE 3.5(b)). The number of edges for each polygon and their size can be adjusted.



(a) Spherical point cloud (b) Points drawn as polygons in the point cloud

Figure 3.5: Spherical point cloud and its caracteristics

As a conclusion, the *geometry shader* has modified the default entity considered by the *vertex* structure. It takes the most common primitive (point) as an input and outputs a list of triangles, called a *polygon strip*, forming a polygon. Then the *fragment shader* provides the color of the point to be visualized on the final rendering of the image according to the texture provided.

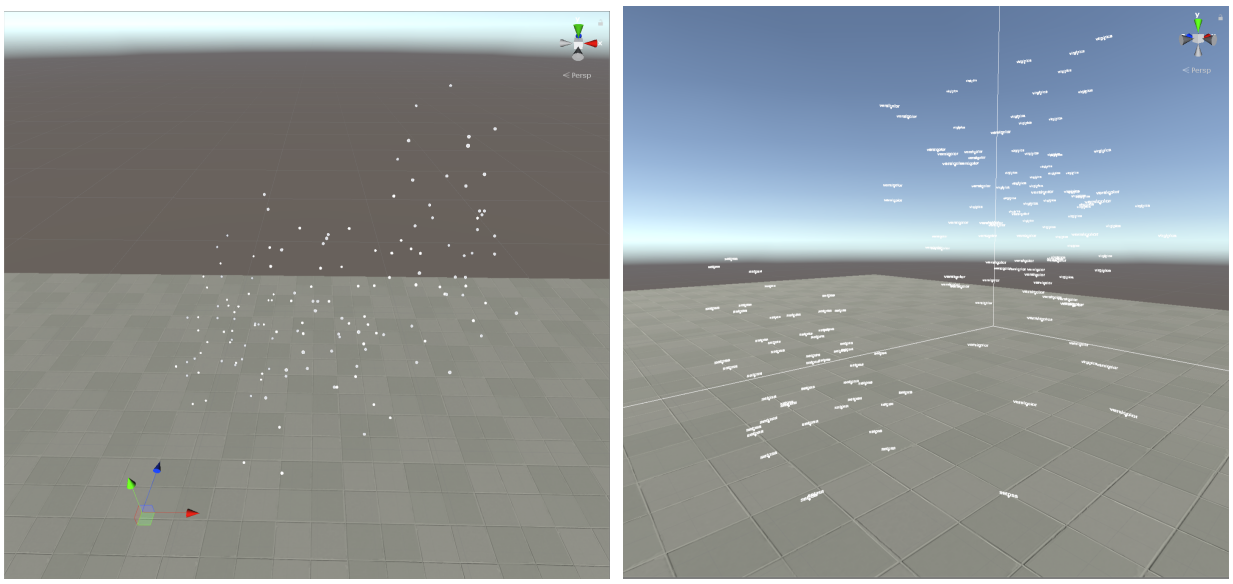
All this work cannot enable the dataset to be represented. The following subsection will introduce the method.

3.2.2 Data representation associated to this cloud

In order to get rid of this sphere, we must provide as the input of the graphic pipeline the point position that we want to visualize in the immersive world. A point will logically be represented by a little 3D Unity sphere. The texture that will be applied to this sphere will be that of the shader. The human eye will see a polygon instead of a sphere composed of small polygons too small to be seen on a human scale.

First, a basic class containing a method for reading Excel files was designed. Each line indicates the position of the data in space (the first three columns), then some characteristic dimensions according to the processed database (the following columns). The delimiters have been defined by regular expressions but it is possible to do without them.

Then, another script allows, from the reading of a file using the previous class, to instantiate for each of its lines a small sphere whose position will be the one indicated in the file. Then normalize with the minimum and maximum values of each axis to avoid obtaining a too sparse cloud (FIGURE 3.6(a)). Above each sphere will be indicated one of the dimensions, if desired, which will then be stored as a string in a *TextMeshPro* Component of a GameObject, child of the sphere (FIGURE 3.6(b)). Its rotation is implemented by the same code as for the player name. The main difference with the previous part is that we have to maintain a list of *MeshRenderer* Components containing the one of each sphere, all of which will benefit from the implementation of the shader.



(a) Point cloud built from an Excel database (b) Legend displayed next to each point

Figure 3.6: Data point cloud and its caracteristics

Now, some ways have been found to find patterns or layers inside the point cloud

thanks to some tools, widely used in *Computer Graphics*.

3.3 Improvements brought on visual data exploration

The MRTK code for shaders has been a solid basis and a valuable help for the following subsections. Here are some of the shaders offered by MRTK (FIGURE 3.7) :

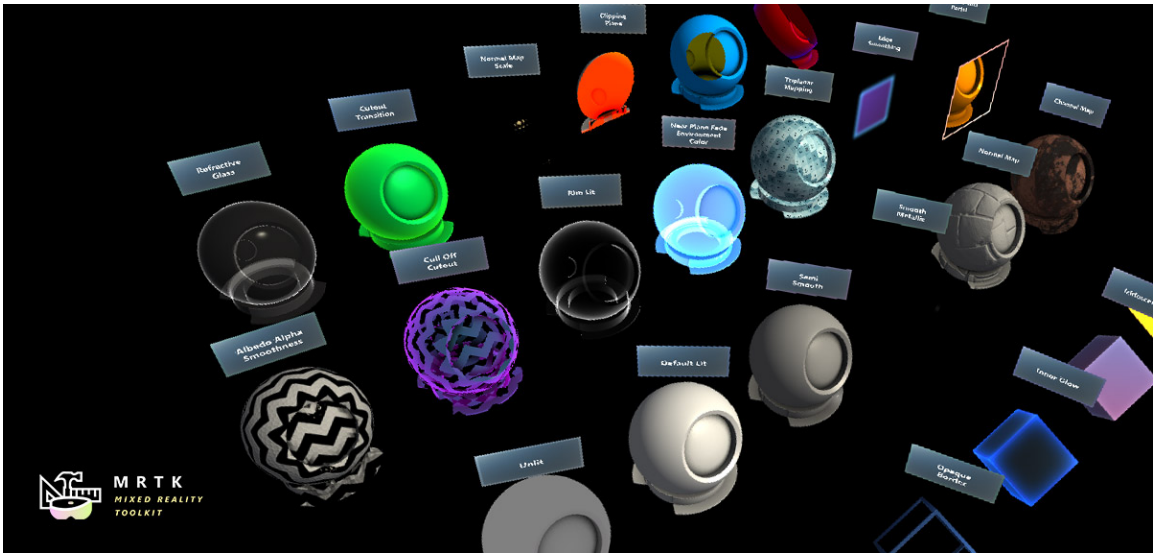


Figure 3.7: Different shaders available under MRTK

3.3.1 Exploration of the data by clipping planes

One of the first ideas we naturally have to explore a complex point cloud is *Clustering*, e.g. to gather a set of points into packets, allowing us to find some meaning. In *Computer Graphics*, one of the most famous methods is *clipping*. Its principle (shown on FIGURE 3.8) is to visualize different layers of points within the cloud by translating a plane on it. Usually, the points between the plane and its holder are not rendered (qualified as clipped), and those on the other side are not changed until they intersect it, hence we name it as clipping plane. In order to highlight points during intersection, an orange color is given to them, transmitted at the beginning of the pipeline.

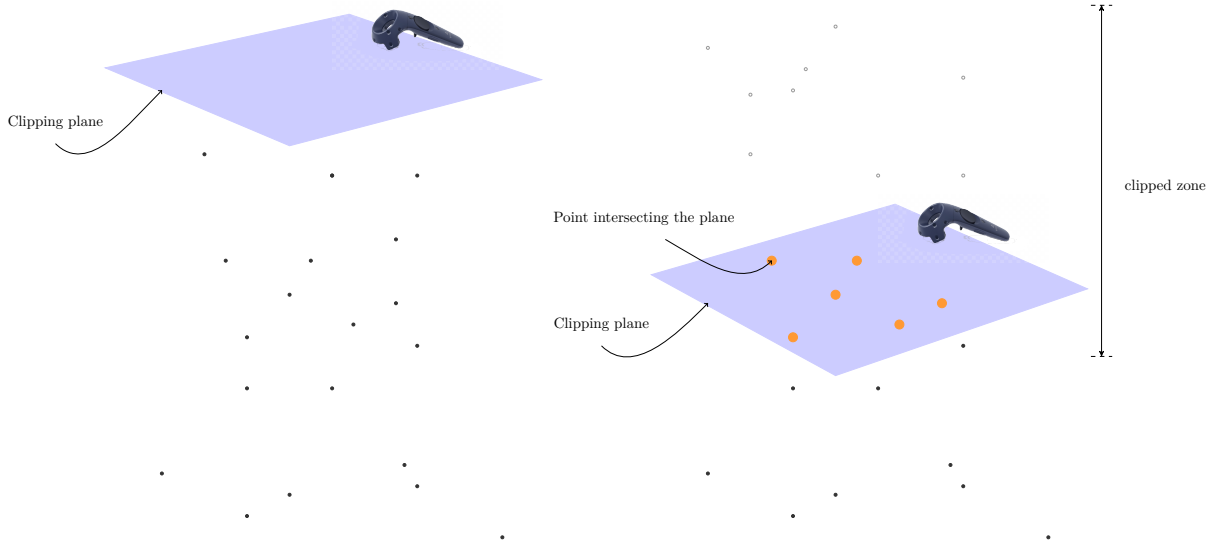


Figure 3.8: Illustration of clipping principle

The shader attached to the material of the plane is only intended to make it transparent, so that points behind it can be seen. A few lines regulating fog rate has been added to a classic *vertex and fragment shader* and tags for transparency has been provided. The clipping property is actually implemented in the point cloud shader. Indeed, the plane is only a tool that may result in a point to be rendered or not, so it makes sense to code this property in the previous shader.

To do this, the distance from each point of the cloud to the plane is calculated by projecting this point on the plane (FIGURE 3.9).

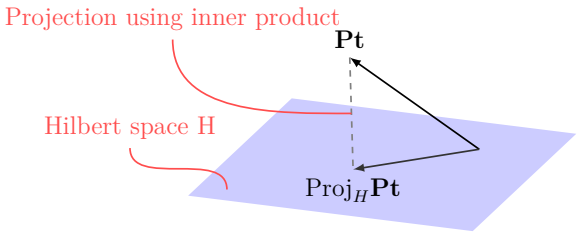
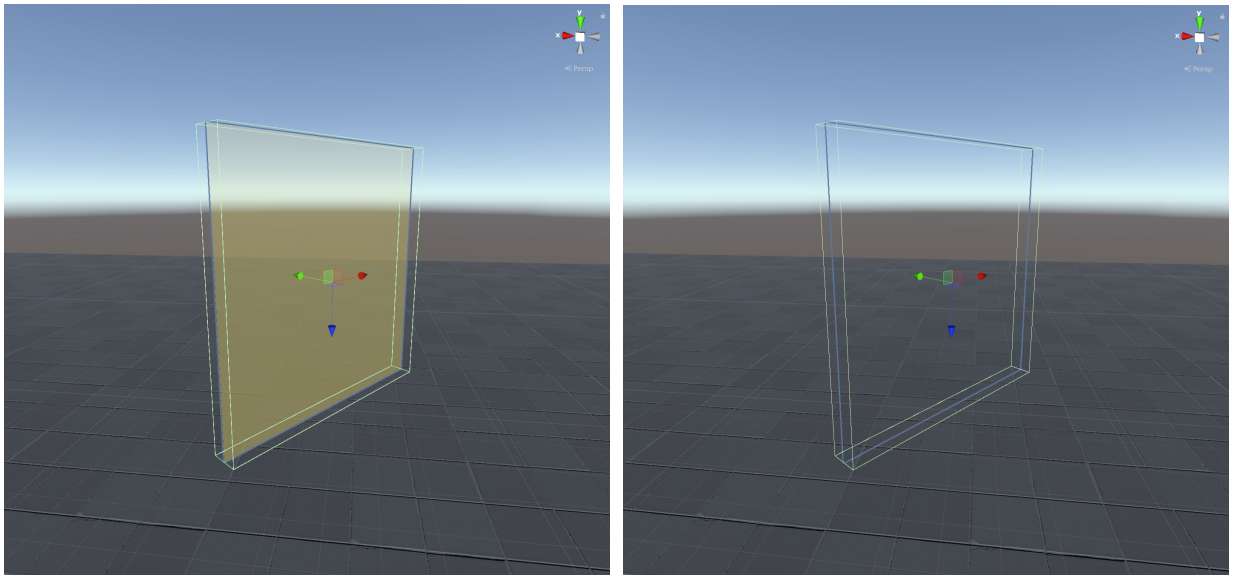


Figure 3.9: Distance from point to plane using projection

The following formula allows to calculate the distance d between a point $\vec{p}$ of the cloud and the plane center $\vec{c}$, thanks to its normal $\vec{n}$, without normalization:

$$\forall \vec{p} \in cloud, d = (\vec{p} - \vec{c}, \vec{n})_{\mathbb{R}^3}$$

Once the implementation of this plane completed, the consequence of an optimized rendering method used by Unity (and by an overwhelming majority of software as well) has been observed, commonly called *backface culling* (FIGURE 3.10). It consists in rendering only faces of the object that the user can see. Several methods exist to make both sides of the plane rendered. Among them, one of the most clever and less expansive in termes of resources method seems to "glue" two planes, so that their respective rendering would be opposite (according to the outward normal of the planes). No *Rigidbody* Component is attached to the second plane, a simple *Fixed Joint* Component is enough to follow the movement of the other plane subjected to gravity. It will also need a *Photon View* Component attached to be rendered in the multiplayer application.



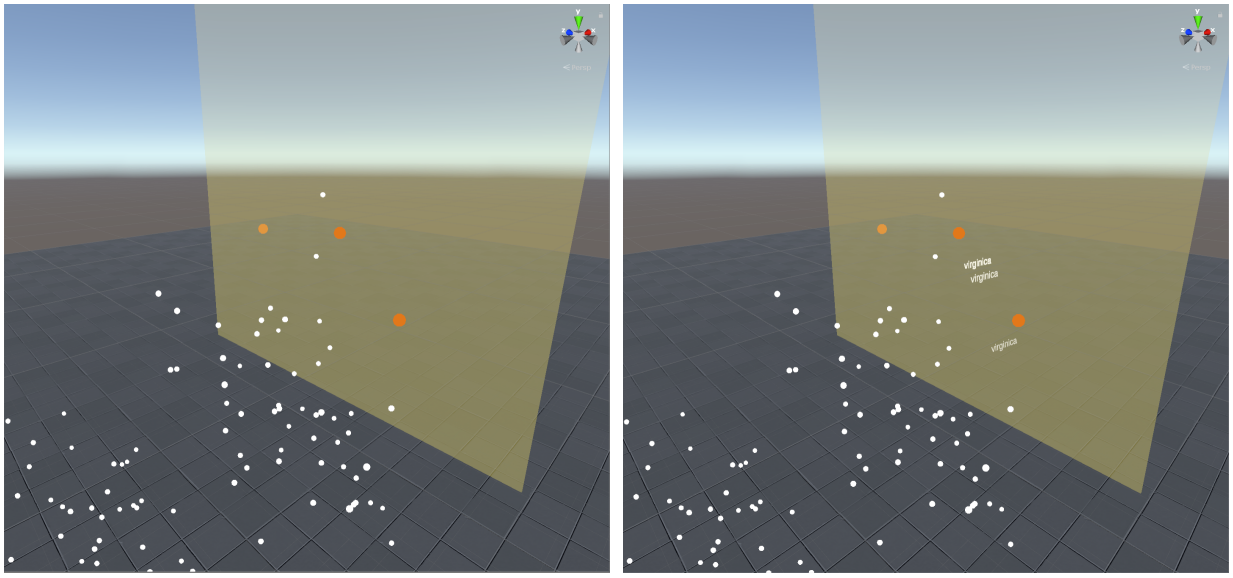
(a) Plane face visible by the player (b) Opposite face not rendered

Figure 3.10: Illustration of *backface culling*

To make this object more interesting for analysis, the points intersecting the plane have been enlarged and highlighted (FIGURE 3.11(a)). But theorically, a plane has no depth, so we cannot expect the distance between intersecting points and plane to be a perfect zero. Thus it is necessary to leave a certain margin along the normal plane axis, taken as the size of a polygon by default, e.g. 0.002 cm. This value is adjustable: it finally represents the thickness of the layer to be highlighted computed from the clipping plane.

In order to have a better reading of the data, the dimension that we want to show for each point (mentioned in section 3.2.2.) has been limited to the points included in the clipping plane. Regarding the appearance duration of the text relating to a point, we cannot hold a reasoning of the type: "if the point is no longer rendered (therefore that it is behind the clipping plane), then we no longer display the text" because this code and the shader are not linked explicitely, and providing string associated to each point to the

shader would be unreasonable. Thus, an alternative is to check if the time elapsed since the text appeared is greater than zero seconds by a Timer script. The results FIGURE 3.11(b) show what have been obtained.



(a) Legend disabled (b) Legend enabled

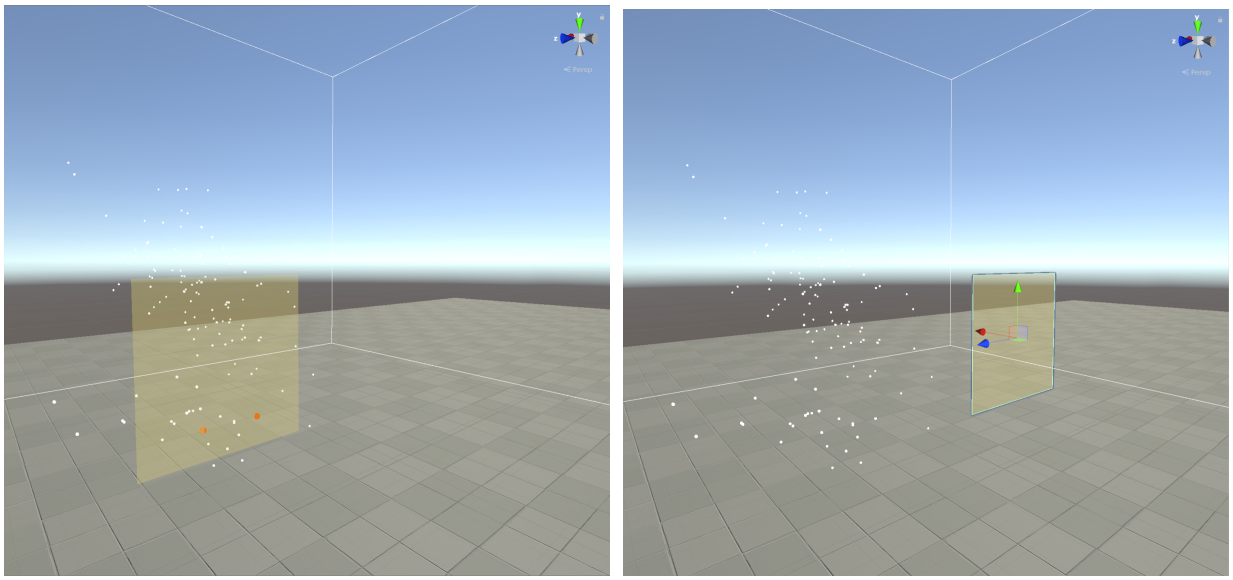
Figure 3.11: Clipping plane results

The first version of this functional code does not, however, allow the size of the clipping plane to be taken into account.

3.3.2 Finite clipping planes and encountered issues

After many tests carried out on the point cloud, we can point out that a translation of the plane several meters aside the point cloud will also make points disappear, which is not desired (FIGURE 3.12). Ideally, we want to provide the shader input with the plane's boundary coordinates, in order to verify if point coordinates belong to the plane ones or not.

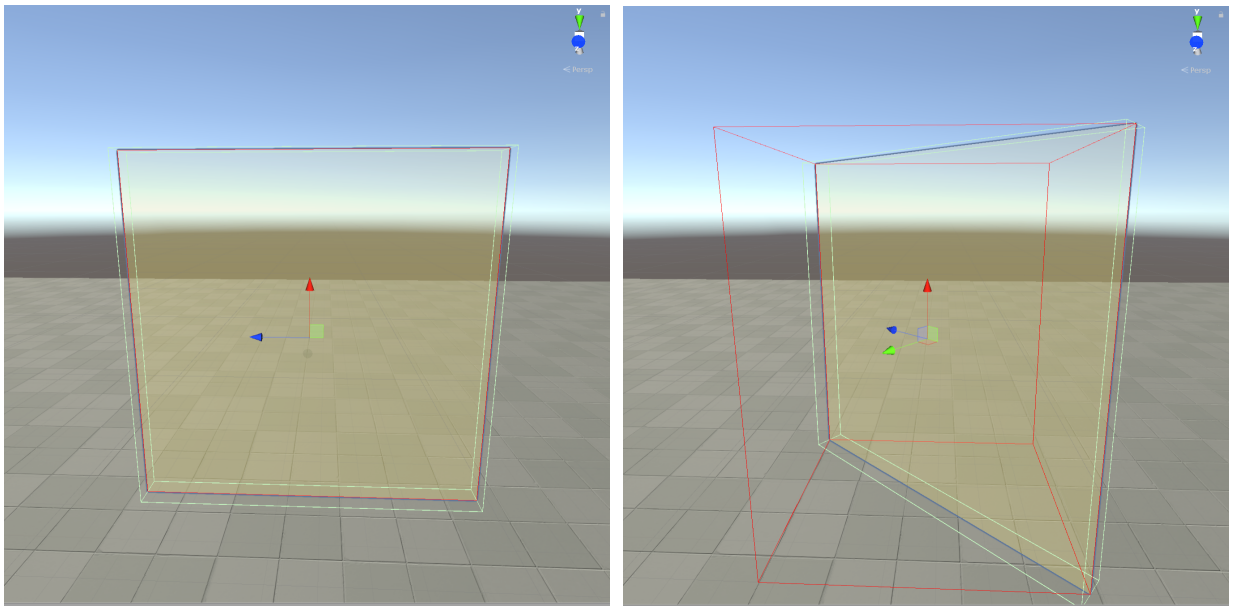
The first idea was to use the Components *MeshCollider*, *MeshRenderer* and *MeshFilter* to extract the boundaries of the basic 3D model to which it is attached using the element *bounds*. No significant results have been obtained. On the one hand, these scripts cannot change the boundaries accordingly when the plane is rotated or scaled, as shown below on (FIGURE 3.13). This behaviour is often refered in the litterature as AABB (for *Axis-Aligned Bounding Box*), a typical way to represent object boundaries. It is computationally and memory efficient because it is a coarse first-approximation, but doesn't fit the object very well, so it is not going to help us. On the other hand, the boundaries were



(a) Plane intersecting points (b) Plane not intersecting points

Figure 3.12: Issue raised with infinite clipping planes

often larger than the plane itself.



(a) Plane modified only by translation (b) Plane modified only by rotation

Figure 3.13: Illustration of AABB collision

After trying many methods to calculate the boundaries of a plane, none of them were working when a rotation was performed. Thus, rather than having planes, very thin cubes of thickness were modelled in order to overcome the difficulty. The boundaries obtained were correct, but still did not change according to rotation, this being due to the element *bounds*, which was not very adaptive. Thus, it was necessary to recalculate one by one the

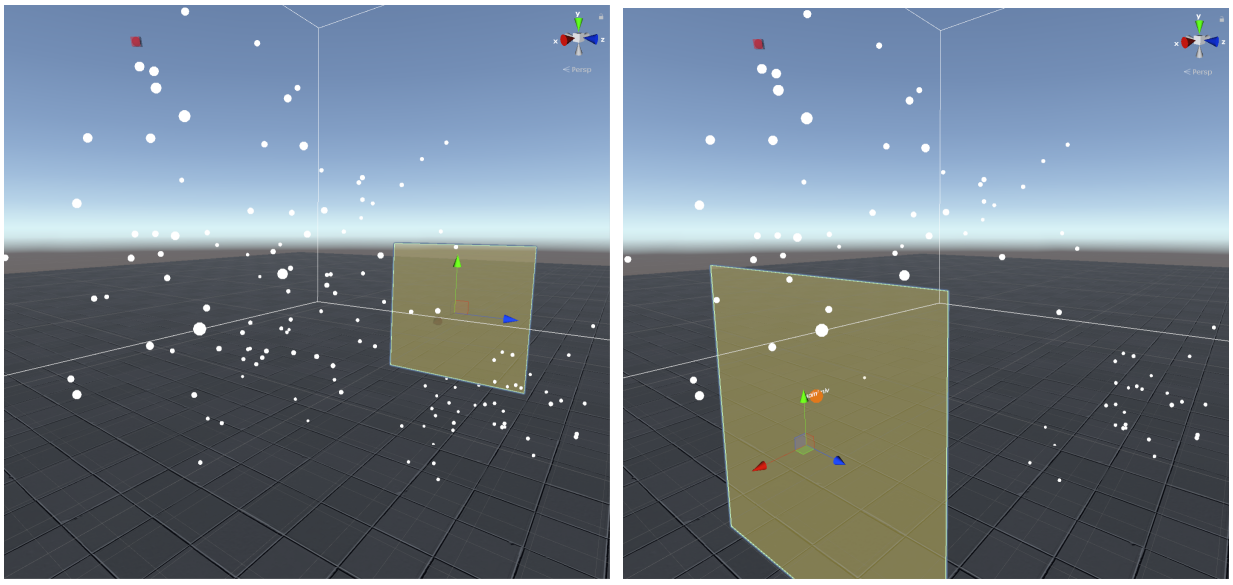
eight corners of the thin cube and taking into account the possible rotations that would animate it to obtain precise boundaries. The general formula to compute a corner is:

$$corner = R \times \begin{pmatrix} sgn(width/2) \\ sgn(height/2) \\ sgn(depth/2) \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

where :

- *sgn* is the sign function that offers $2^3 = 8$ possible combinaisons, that give the eight corners coordinates,
- R is the clipping plane rotation matrix computed with Unity method `SetLookRotation`,
- width, height and depth corresponding to the plane ones computed by using *localScale* parameter,
- $(x\ y\ z)^T$ are the plane center coordinates.

Then provide the extreme values for each world axis allowed to obtain the expected clipping result thanks to a discriminating condition stipulating that whether the plane normal must be collinear to one of the world repair vector (see FIGURE 3.14).



(a) Plane behind point cloud (b) Plane clipping all points on its trajectory

Figure 3.14: Clipping plane according to normal plane collinear to x axis

Conditions based on dot product to know if a point was behind or in front of the plane were not enough to succeed in having a finite clipping plane that can be rotated.

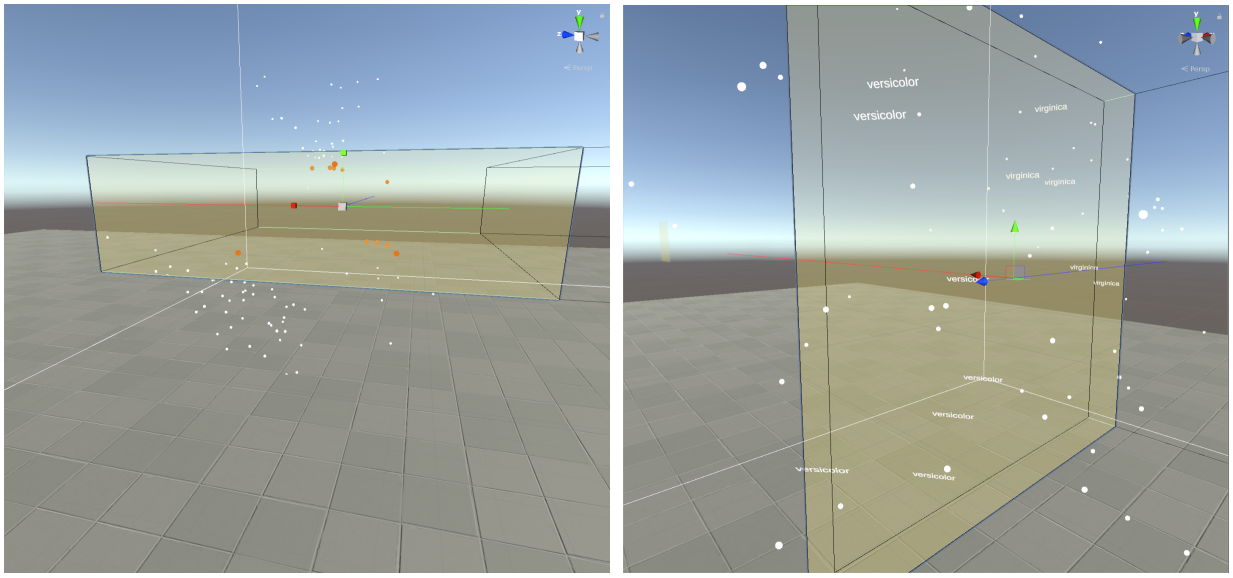
Indeed, it was only necessary to delete the points located in the area behind the plane, area whose main direction is the outward normal of the plane. So the trick is to have a clipping box that is a parent of a plane that renders one face of the cube. The depth of the box (corresponding to normal plane axis) will be very important and will largely exceed headset position. The player will not be aware of the trick inside the immersed environnement.

Implementing a clipping box is analogue to a clipping plane, except for the clipping condition (applied inside or outside the box) that implies a new formula to compute the distance:

$$d = \left\| \max\left(\vec{0}, \vec{d}\right) \right\|_{\mathbb{R}^2} + \min\left(\max\left(d_i\right)_{0, i \in [|1,3|]}\right)$$

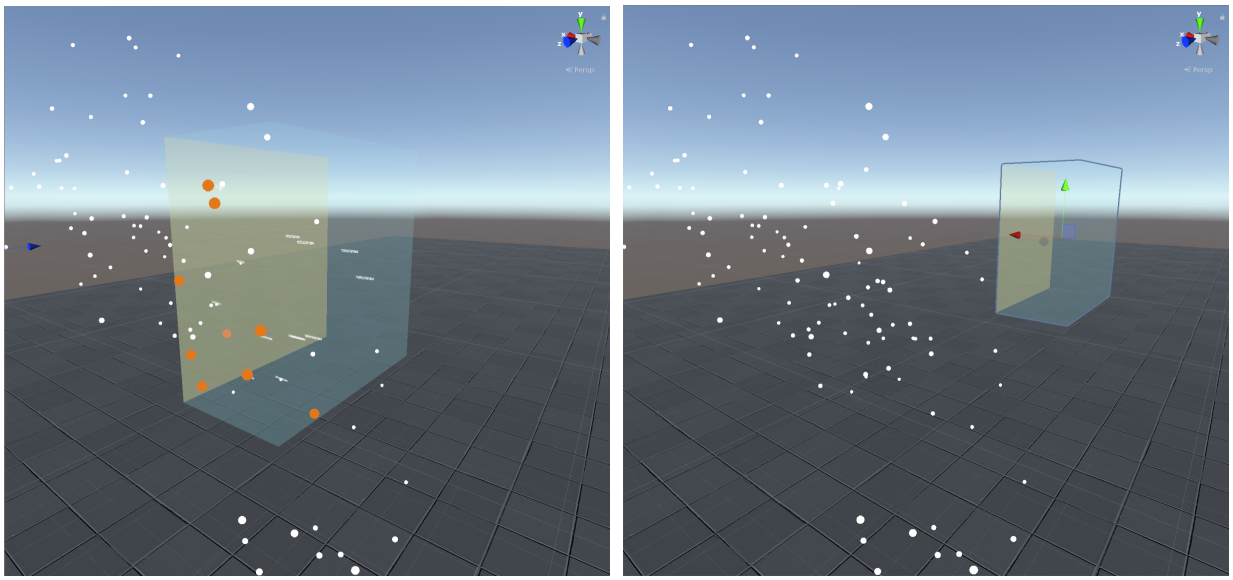
where $\forall \vec{p} \in cloud, \vec{d} = |R \times (\vec{p}, 1)| - boxSize$ is the distance vector from $\vec{p}$ to the plane with R its rotation matrix.

These are the results which have been obtained, in FIGURE 3.15 for the clipping box and FIGURE 3.16 for the clipping plane simulation.



(a) Thick clipping box (b) Thin clipping box with legend

Figure 3.15: Clipping box results



(a) Points are inside the box

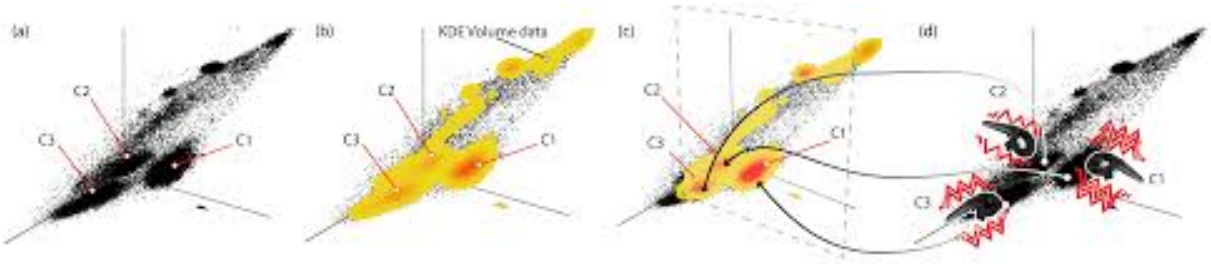
(b) Points are outside the box

Figure 3.16: Results of clipping plane with box attached

Now that a clipping plane has been implemented, let's think about how we could use two of them for analysis.

3.3.3 Implementation of a Highlight Planes prototype

The concept of Highlight Planes is based on a research paper [13] mostly written by the same authors as ImAxes. They have tried to explore various ways to interact easily with data in an immersive environment. Among the interesting techniques they have worked on, the *scaptics* (name inspired by *haptics* word) consist in brief of vibration returns according to the local density of the point cloud (FIGURE 3.17(a)). Another idea was to take inspiration from the cutting planes in order to make some patterns appear in the cloud (detecting bunch of points intersecting the plane or holes as in (FIGURE 3.17(b) and (c)).



(a) *Scaptics* : vibration of the *controller* according to the density of the cloud

(b) Highlighting a hole

(c) Highlighting a dense local area

Figure 3.17: Illustration of *scaptics* and *highlight planes* [13]

From these two techniques, an interesting hybrid was imagined, consisting in highlighting a certain number of points of the desired cloud by placing two planes in parallel. Points in the area between these two planes would therefore be highlighted, as illustrated on FIGURE 3.18.

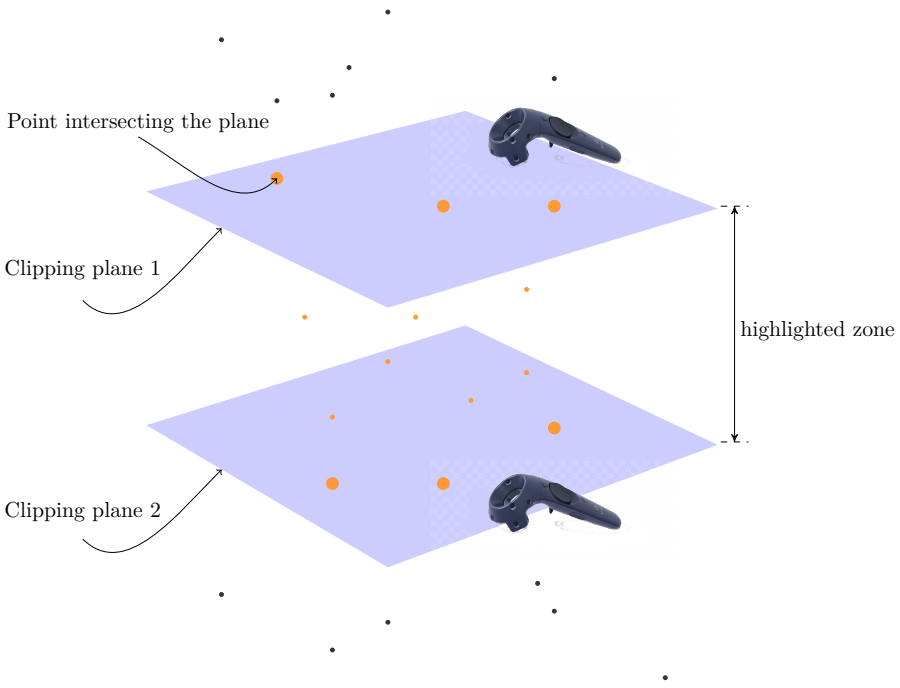
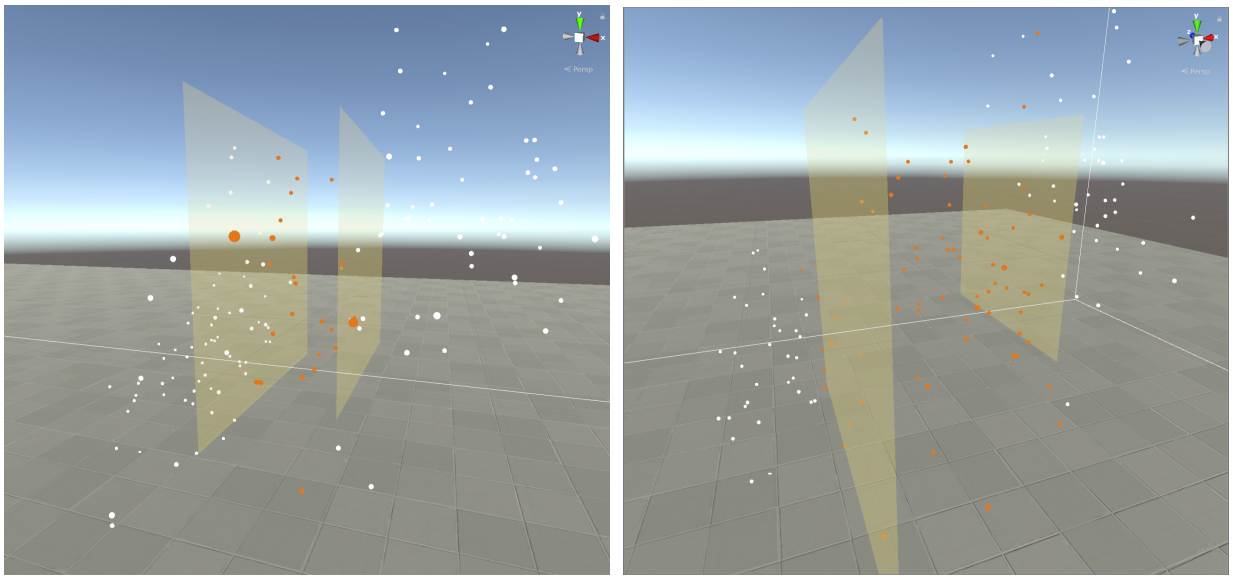


Figure 3.18: Illustration of highlight feature

The highlight was first achieved by using the previous second orange texture throughout the graphic pipeline, as shown on the obtained result on FIGURE 3.19 :



(a) Highlighting thin portion (b) Highlighting thick portion

Figure 3.19: First prototype of highlight planes

Then, the implementation of classical light reflection models quickly became essential for a better graphic rendering. The first one that has been implemented is the *Lambertian reflexion model* (FIGURE 3.20). It is accessible and gives mat aspect to materials, due to *diffuse* reflexion. Lambert's law states that the intensity of light scattered from a point on a reflecting surface follows a cosine formula, regardless the player's point of view.

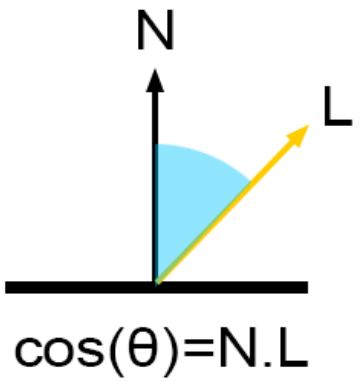


Figure 3.20: Lambertian reflexion model and useful vectors

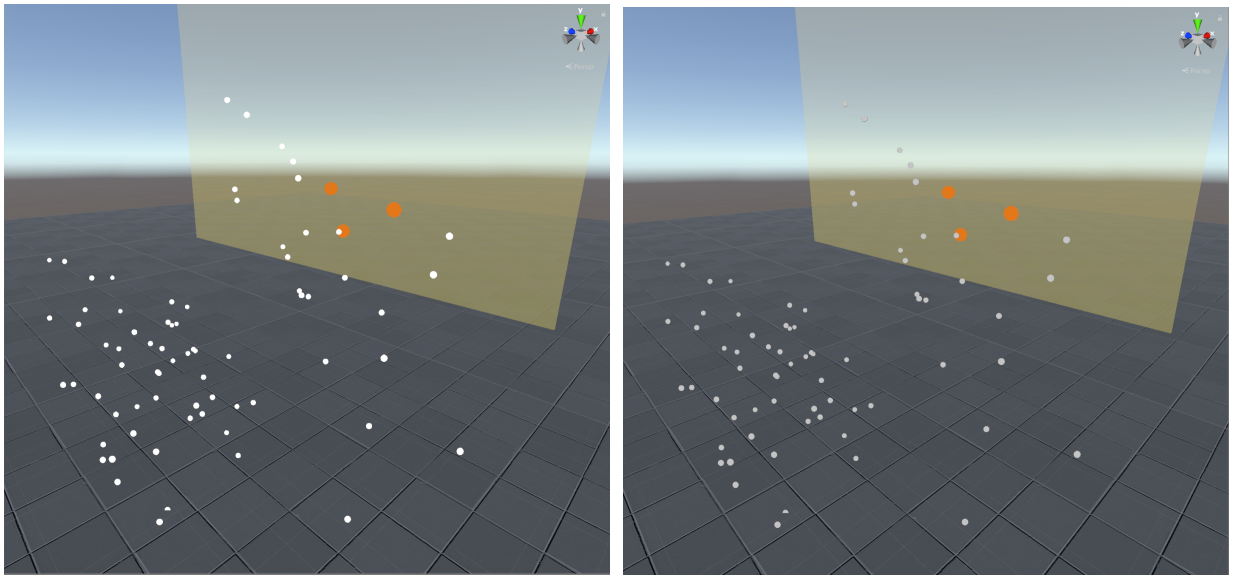
So a highlight effect has been indirectly created by appling the Lambertian model to all points which have therefore been attenuated, except the orange ones. The formula used is conditionned to never be under 0 :

$$I_{refl} = \max\left\{0, \left(\vec{l}, \vec{n}\right)_{\mathbb{R}^3} \times c \times I_{incid}\right\}$$

where :

- $\vec{l}$ is the normalized directionnal light vector,
- $\vec{n}$ is the vertex surface normal vector,
- c is the surface color value,
- I_{incid} is the incident light intensity and I_{refl} the diffuse one.

You will find the obtained results below (FIGURE 3.21):



(a) Diffuse light disabled (b) Diffuse light enabled

Figure 3.21: Results for Lambertian reflexion model

To give orange points the desired highlight, the *Blinn-Phong reflexion model* has to be implemented. It gives materials specular lighting (FIGURE 3.22). This model is dependant of the player's point of view but also the angle of the incident light.

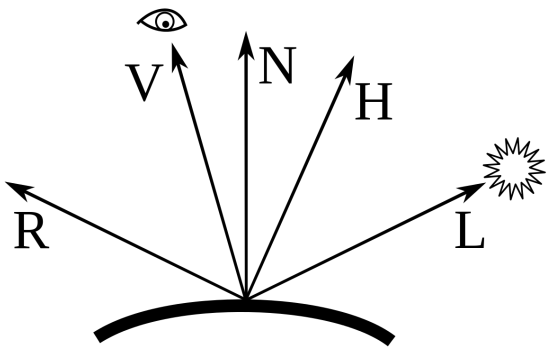


Figure 3.22: Blinn-Phong reflexion model and useful vectors

But it is more complex to implement it in a shader than the Lambertian one. This work is in progress.

As a conclusion, we can summarize by the diagram below (FIGURE 3.23) the coded graphics pipeline for this application :

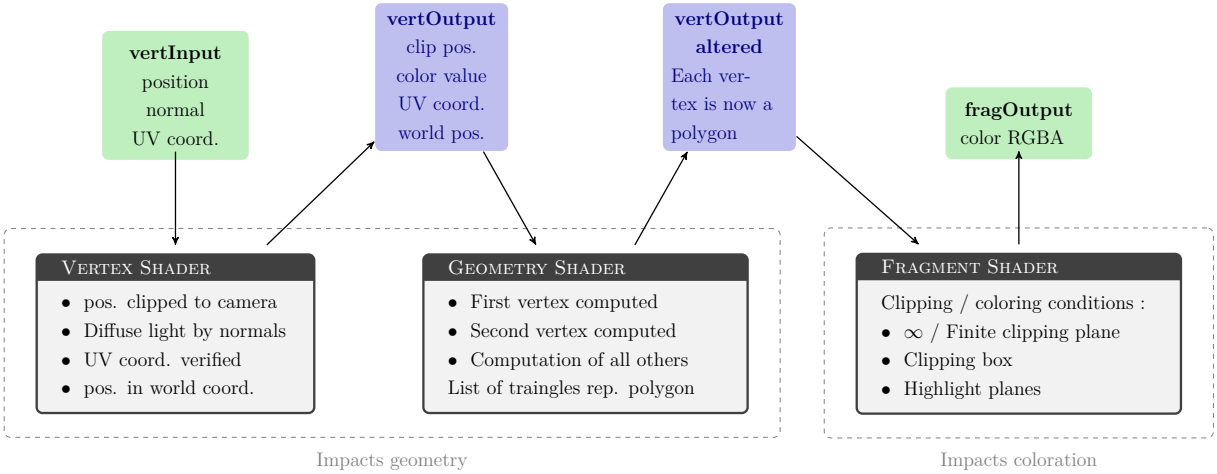


Figure 3.23: Graphics pipeline of the application

Let's now take a more detailed look on how each player can interact each other to share some interesting information.

3.3.4 Perception of a user's view by all

This section is at the origin of the chapter. When the team have thought a collaborative solution for immersive data visualization, the first concern was how a user could, at any given time, enjoy the exact image of another user in real time. Imagine that a user has deduced valuable information by a precise configuration of clipping planes in a cloud, only visible from his point of view. How could all other users enjoy this view without having to move to its exact position? This work is in progress.

It should be recalled that the internship ends at the end of September, so there will be one month left to finish the work started.

3.4 Future objectives and outlook concerning the internship

The time left will be an opportunity to deepen and refine the various points discussed in sections 2 and 3. The priorities will be as follows:

- implement a solution to visualize another player's screen, at a given time, in such a way we can benefit from his data visualization, with his angle of view.
- take a closer look at other tools such as IATK, and use them as a basis for importing additional *analytics* into the project. This could involve volumetric graphical rendering issues, interesting to find patterns subject to analysis in the data;
- refine communication between different users, and thus allow a player to refuse to grant object ownership to another one. For the moment, membership transfers are automatic as soon as the *controller* trigger is pulled when both the *controller* and object *colliders* are in contact;
- import the project into the ImAxes tool. Some colleagues have made improvements on their side. Jonathon Hart has merged ImAxes and MapsSDK-Unity tools. His work allows to visualize, on a 3D map of Australia, bar graphs emerging from specific locations when a visualization of ImAxes is sufficiently close to the map. And Simon Malnar has brought a multi-platform solution for this project. It would probably be more appropriate to introduce the collaborative solution in this project. As ImAxes was not designed for initial multiplayer use, it will be a difficult task to fully integrate multiplayer dimension. And in this respect, my contributions towards understanding some of the issues and potential solutions in network data visualization is a crucial first step on this path.

Conclusion

The original objective of the internship was to implement a solution to import the data visualization and analysis tool called ImAxes, running only in VR for the moment, for MR. In theory, it should allow the manipulation of a set of visualizations by HoloLens gestures and voice commands, and ideally enable a collaborative use of the application. However, the time spent was not sufficiently successful, the objective of the internship was not clearly defined from the beginning, and HoloLens2 headsets could not be obtained in time. For all these reasons, it was decided to reorient the internship more explicitly on the development of a high-quality graphics rendering multiplayer solution, functional for VR. In this respect, significant improvements have been made.

The improvement of my mastery in Unity as well as the understanding of virtual technologies, totally unknown to me until then, took a considerable amount of time and slowed down my progress at the very beginning of the internship. It was after consultation with the team in charge that an objective has been set, meeting both the needs of the laboratory and my professional objectives.

The current status of the project suggests many opportunities for improvement. Some research papers that are complementary to what has already been coded have not been used yet. In particular, it would have been possible, through a deep understanding of algorithmic structures of other projects, to obtain a larger number of *immersive analytics*. Besides, the abilities to analyse data (the highlight in particular) could be improved by working on more complex lighting effects.

I have considerably increased my mastery of programming in Unity, but also learned to find algorithmic and geometric solutions to fill the few gaps interviewed in the game engine. I was also aware of the importance and potential of Virtual and Mixed Reality technologies, having studied them in many aspects. It is certain that this internship will be an undeniable added value in order to gain exposure to techniques relevant to the world of video games and special effects that I have the ambition to integrate.

Bibliography

Scientific articles

[1] Benjamin Bach Christophe Hurter Maxime Cordeil Andrew Cunningham. "IATK: An Immersive Analytics Toolkit". PhD thesis. Monash University, University of South Australia, University of Edinburgh, Ecole Nationale de l'Aviation Civile, 2019.

[6] Tim Dwyer Bruce H. Thomas Kim Marriott Maxime Cordeil Andrew Cunningham. "ImAxes: Immersive Axes as Embodied Affordances for Interactive Multivariate Data Visualisation". PhD thesis. Monash University, University of South Australia, 2017.

[10] Evan Suma Rosenberg Mahdi Azmandian Timofey Grechkin. "An Evaluation of Strategies for Two-User Redirected Walking in Shared Physical Spaces". PhD thesis. USC Institute for Creative Technologies, 2017.

[13] Clement Robin Barrett Ens Bruce Thomas Tim Dwyer Arnaud Prouzeau Maxime Cordeil. "Scaptics and Highlight-Planes: Immersive Interaction Techniques for Finding Occluded Features in 3D Scatterplots". PhD thesis. Monash University, University of South Australia, 2019.

[14] Akira Utsumi Fumio Kishino Paul Milgram Haruo Takemura. "Augmented Reality: A class of displays on the reality-virtuality continuum". PhD thesis. ATR Communication Systems Research Laboratories, 1994.

Website pages

[2] CSIRO team. NEAR project. `https://near.csiro.au/assets/46008de4-56b8-64e4-aece-b7970f28e91a`.

[3] Microsoft. Mixed Reality Academy. `https://docs.microsoft.com/en-us/windows/mixed-reality/tutorials`.

[4] Unity Technologies. Unity documentation. `https://docs.unity3d.com/Manual/index.html`.

[5] Microsoft. Csharp documentation. `https://docs.microsoft.com/en-us/dotnet/csharp/`.

[7] Microsoft team. Maps-SDK Unity. `https://github.com/microsoft/MapsSDK-Unity`.

[8] Geoscience Australia Data61 and Clean Energy Council teams. AREMI plateform. `https://www.nationalmap.gov.au/renewables/`.

[9] Photon. PUN documentation. `https://doc.photonengine.com/en-us/pun/v1/getting-started/pun-intro`.

[11] Alan Zucconi. Shaders tutorial. `https://www.alanzucconi.com/2015/06/10/a-gentle-introduction-to-shaders-in-unity3d/`. 2015.

[12] Anthony Blechet. Shaders Laboratory. `http://www.shaderslab.com`. 2017.

Appendix 1 - Infography of VR / AR / MR differences

The concept of Mixed Reality is very often confused with AR because both associated technologies offer the vision of an enriched world. The key difference is that only Mixed allows interaction with the virtual objects that have enriched the world. Digitisation is sometimes so important that it can sometimes be confused with VR, but the principle of Mixed Reality is to fix the position of holograms in the space of real objects. You will find a summary FIGURE 1.

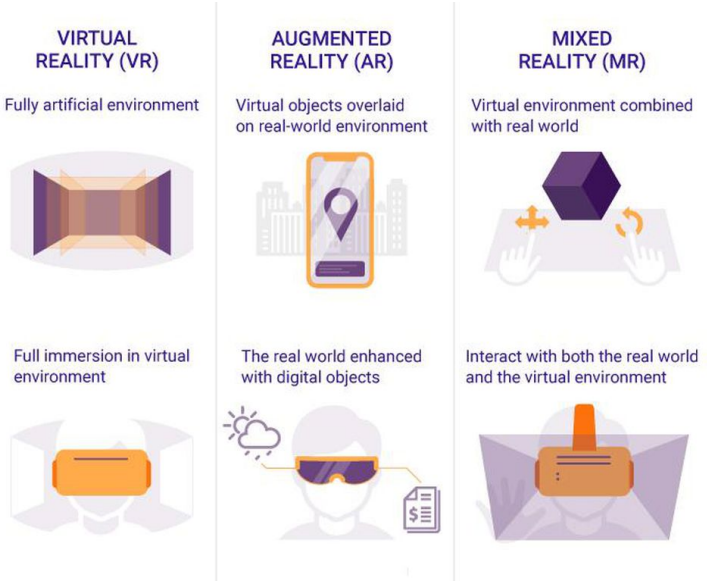


Figure 1: Differences between RV, RA and RM

To better understand where VR, AR and MR limits are, Milgram introduced in 1994 [14] the concept of reality-virtuality continuum illustrated FIGURE 2, bringing a broader definition to Mixed Reality qualifying it as any experience located on the continuum between Reality on the extreme left and Virtuality on the extreme right.

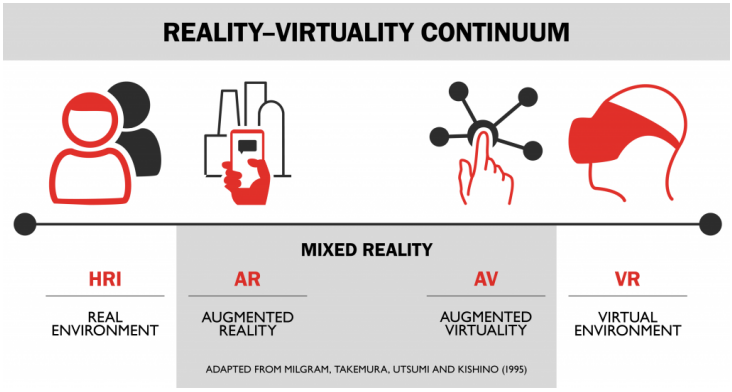


Figure 2: Milgram continuum

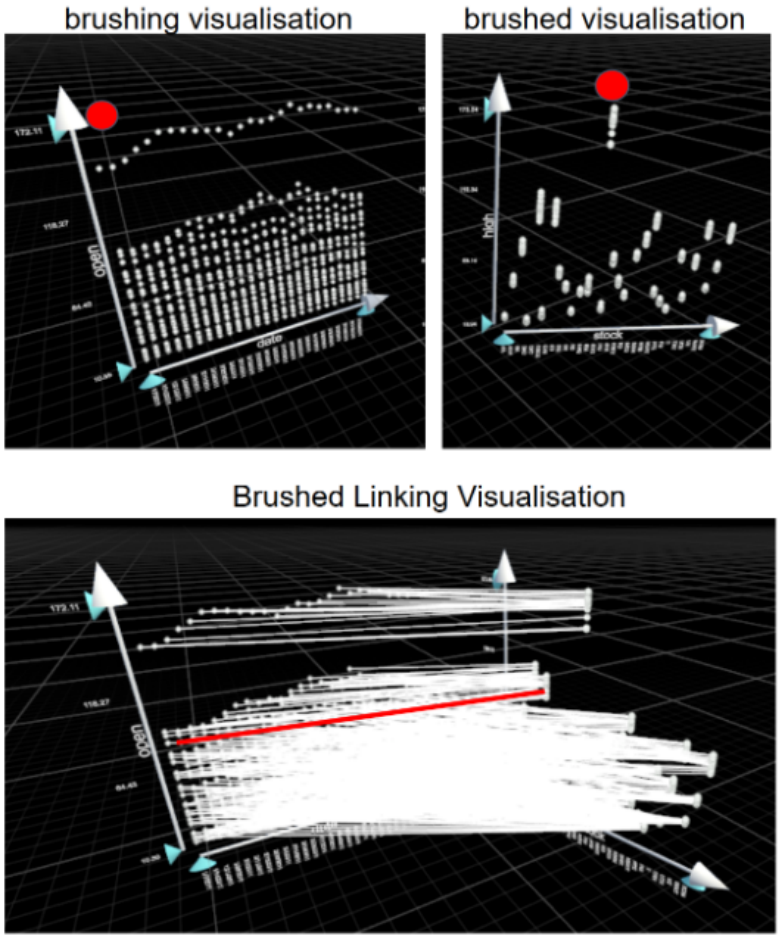
Appendix 2 - IATK : the advanced version of ImAxes

IATK (for Immersive Analytics Toolkit) [1] is a Unity project allowing the creation of VR data visualization. This open-source tool was designed by the ImAxes team and some other researchers. The package VRTK is needed if we want to run the application.

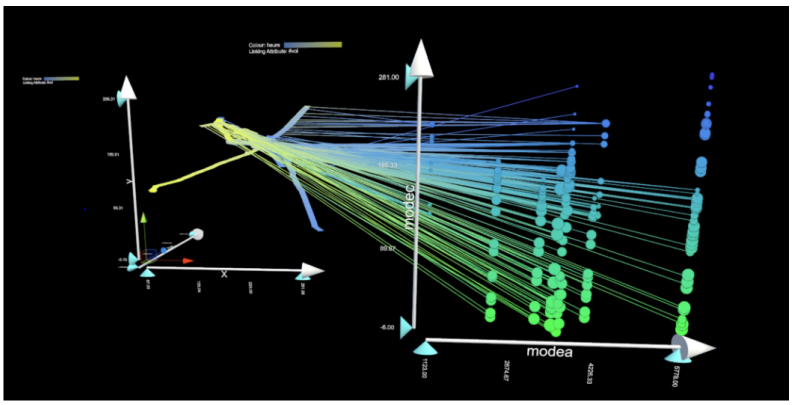
IATK can be seen as an advanced version of ImAxes, whose main improvements are:

- a possibility to provide as input a big data scaled database (order of magnitude around the million), and the possibility of scaling according to what we want to focus on;
- a wider range of analysis tools, such as a successful *brushing and linking* (FIGURE 1.3(a)) feature allowing points enhancing, more precise linked visualizations (FIGURE 1.3(b)), etc;
- a greater diversity of visualization (FIGURE 1.4) than ImAxes such as 3D histograms, superposition of different histograms, etc.;
- a GUI interface for a better management of the *immersive analytics* experience.

When a histogram is built, a high level of control over the visualization parameters is possible in the Hierarchy window. One can then launch the application to view the built visualization.

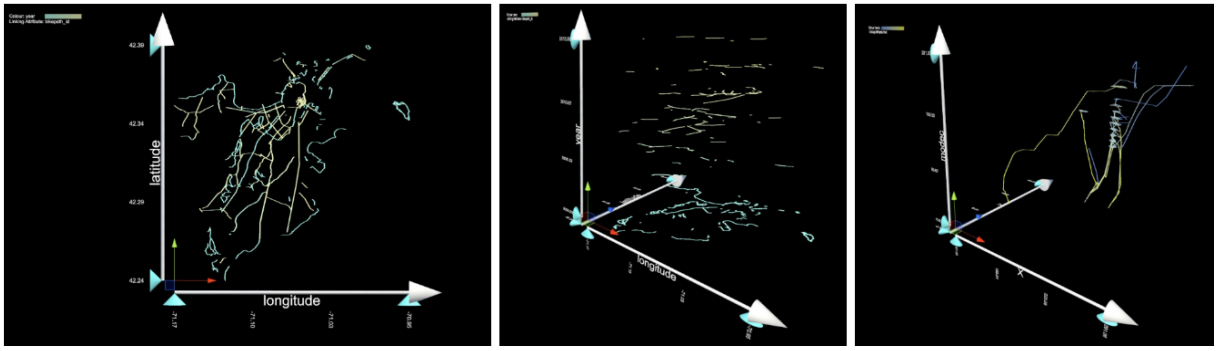


(a) Principle of *brushing and linking*

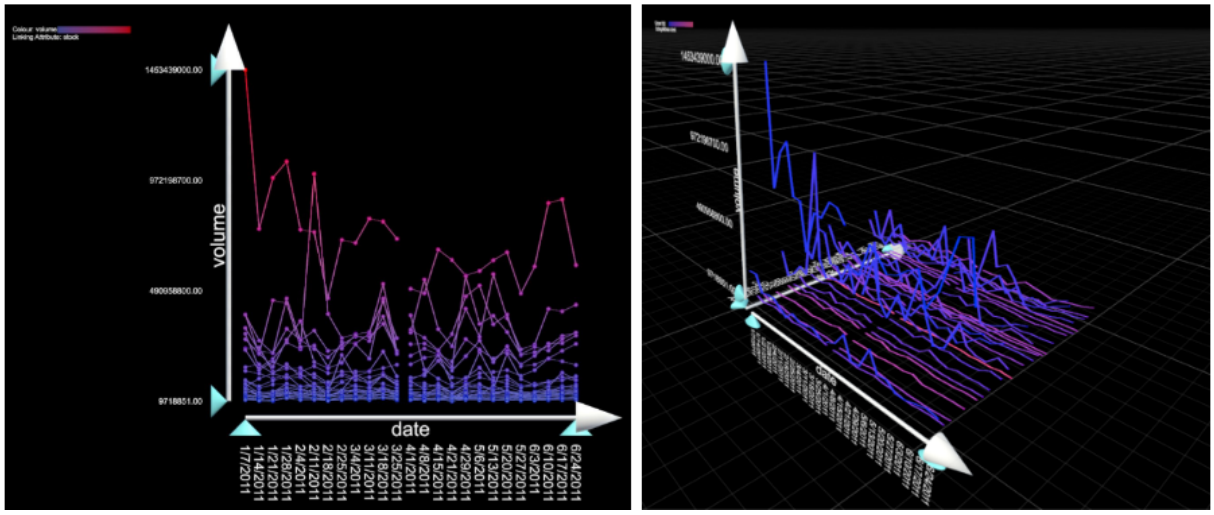


(b) Principle of linked representations

Figure 3: Improvements in the analysis available in ImAxes [1]



(a) Wire representation



(b) Dot representation

Figure 4: Innovating representations, not in ImAxes [1]

Appendix 3 - Network layers in UNet

The image below FIGURE 5 how the Unity UNet HLAPI is built : the root is the naturally the LLAPI and then successive layers are brought to add features, mainly enabling state synchronization between players including high-performance serialization, network classes for a better handling of notions and so on.

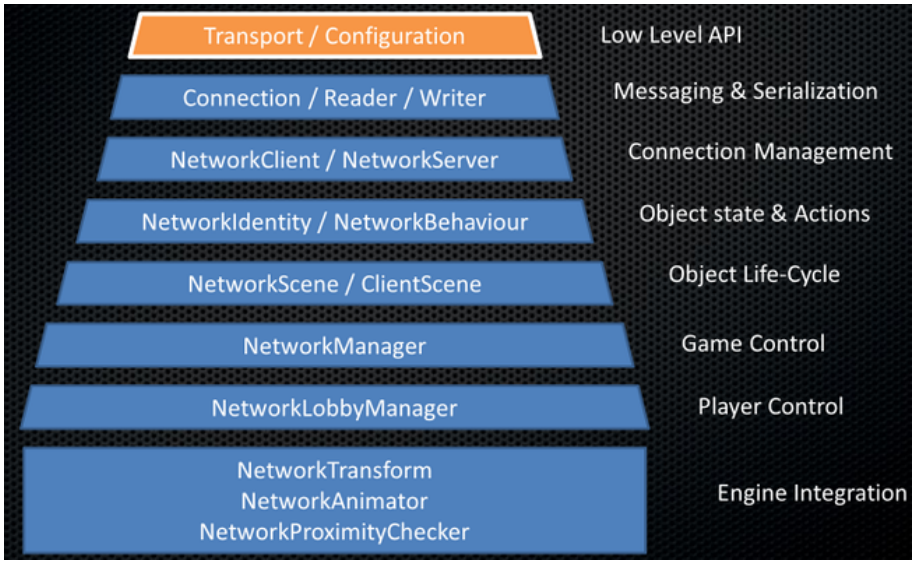


Figure 5: HLAPI UNet network layers

To indicate Unity we want to use its HLAPI, we just need to use namespace `UnityEngine.Networking`