Integrated Virtual Reality Game Interaction: The Archery Game

Domenico Sammartino

Submitted to

University of Hertfordshire

in partial fulfilment for the award of the degree of

Masters by Research

March 2015

Contents

List of Figures

Abstract

The aim of this research is to develop an innovative technology framework that allows a new type of human-computer interaction, where user's motion recognition is combined with immersive visualisation. The demonstrator program allows the user to visualise a virtual arrow on the top of a real physical crossbow used as game controller. Stereoscopic-3D Visualisation is implemented using two virtual cameras with a variable angle between them (Automated Toed-in). An algorithm is also used to calculate the maximum acceptable dimension of the stereoscopic arrow, in relation with the screen size and the data provided by the Motion Capture system about the user position. The Motion Capture data are transmitted to the video game using a network interface demanded also to depacketize and process the motion data. Tests prove that the Stereoscopic-3D effect is strong enough to visualise the virtual arrow in a realistic position on the top of the crossbow. The theory formulated may be utilised to develop a new generation of Stereoscopic applications easy to use and immersive.

Chapter 1

Introduction

The field of user-computer interaction may be considered one of the most relevant topics in Computer Science nowadays. Indeed, it is common to read about Augmented Reality and Intelligent Interfaces in specialised journals and the everyday news. Furthermore, the game industry is now moving towards a direction where the player or the real environment[1] become part of the game.

Nowadays touch-less devices are growing between end users, due to the innovation which they bring among the different types of interactions between humans and machines. For example, it is more comfortable to swipe a hand in front of a tablet to turn the page of an ebook[2] or to use hand gestures to control a SmartTV[3]. Theoretically, the next step for touch-less devices is to allow users to perform more complicated actions, such as driving a robot using military hand and arm signals[4] or for an artist to carve his\her next masterpiece.

Speech Recognition is another example of touch-less interaction; it is commonly implemented in many electronic devices (especially smart-phones and tablets), and allows the user to interact with the device using voice commands. Through these, it is possible to perform different types of tasks such as sending emails, dialling a number, setting an appointment on the calendar and so on. Apple Siri and Google Now are the most important examples of this technology.

A wider scenario is offered by **motion capture** devices. These systems are able to capture the movements of body parts, and to use them to interact with the user interface. The most well-known motion capture device is **Microsoft Kinect**[28] (codename in development *Project Natal*), developed by Micorsoft as a game controller for XBOX

[1] RoomAlive is a proof-of-concept realised by Microsoft, which transforms any room in an immersive and augmented space, where the player can move and interact freely with virtual objects

[2] digital book publication in electronic form, commonly used on devices such as smart-phones or tablet.

[3] New generation of TVs or set-top boxes integrated with broadcasting media functionality or the possibility to install application on it to improve the device functions

[4] form of visual communication used by a military force when the soldiers need to remain undetected

360 and XBOX One. Equipped with an RGB camera, depth sensor and multi-array microphones, this device is able to provide full-body motion capture, facial and voice recognition. Following the XBOX edition, Microsoft released a version to be used on Microsoft Windows - dedicated especially to developers - that has also been used to implement gesture recognition[4] on the PC.

Used principally in film making, nowadays motion capture is widely used in gaming and, more recently, with the introduction of the **LEAP Motion Controller**, has also been introduced as a PC input device. Previously, motion gestures on PCs were possible principally using webcam-based software (much less accurate), or using the Kinect (which was something outside the competence of the common user).

The next step for new generation devices is to be able to analyse and understand how the user interacts with the outside world and use this data to create a new interaction model between the user and the machine, thus creating a more natural and instinctive user experience. Moreover, this new kind of model could make things easier for those classes of users with disabilities, or those who have difficulty using standard models of interaction. An example would be a simultaneous translator from sign language to voice, using a motion capture system that is able to translate sign language into spoken words. The google Project *Tango* is an example of this. This proof-of-concept project aims to make a portable device capable of analysing the room in which the user is located, and to perform certain operations in relation to it.

The introduction of *motion-sensing* game controllers such as PlayStation Move™, Nintendo Wiimote™ and Microsoft Kinect™ has led to a new generation of game-play which aims to create a fully immersive game experience for the player. Furthermore, the comeback of **HMD**[5] technology introduced a new concept of game-play in which the player feels immersed in the virtual environment. **Stereoscopic 3-D** (S3D) visualization, instead, was particularly appreciated in the early 2000s, and reached its peak with James Cameron's 'Avatar', which led to a revolution[31] in this particular field[6]. Nowadays, S3D is also commonly integrated into video-games, but only in terms of graphic-effects improvement.

Do (Research) What People Want

The starting point of this research project was a motivation to expand the theory behind an only Stereoscopic-3D Game, and to apply S3D to a wider scenario. Moreover, the theory on which this research is based can also be used to develop an Augmented Reality application, which is considered a hot topic nowadays. There are many examples related to this kind of application that are trying to change the way people look at reality; starting from the experiment by the supermarket chain Tesco[25] - to improve

[5]Head-mounted display: helmet provided with a small display optic in front of the user's eye
[6]in terms of shooting and displaying stereoscopic movies

the e-commerce user experience - , to the marketing campaign delivered by Tissot[20] to publicize their products.

It is important to propose a research study on an innovative topic, especially in order to give an important contribution to human knowledge and the evolution of technology.

Improvements

Comfort and user experience are of supreme importance to mainstream consumers. Nowadays, many computer games producers equip their products with a stereoscopic-3D feature, but they are only concerned with the "special effects", and not with serious improvements to the game-play. Moreover, most recent TV or PC-Monitors are Stereoscopic-3D ready, and the related technology[10] is continuously improving.

Having a stereoscopic-3d effect that causes discomfort and does not add anything to the user experience is unreasonable. For this reason, the focus of this project was to study this field in depth, in order to formulate a better theory for obtaining an S3D effect that is not as disturbing to the user, and which either fundamentally improves the overall experience or is necessary to enjoy the content.

A Matter of (right) Movements

The possibility of using a stereoscopic-3d interface as a new way of interacting with a pc is exceptional only if combined with a revolutionary input interface. Raise their hands those who have never dreamt of interacting with a pc like Tom Cruise in Minority Report. The good news is that this kind of technology exists nowadays. LEAP Motion Control and Microsoft Kinect are only two examples of hand-gesture input peripherals. The moment in which this kind of technology will be embedded in any device is not too far off in the future[9][19].

Motion Capture is the most natural answer if someone wants to develop an easy to use and immersive interface. For this reason, it is necessary to use a Motion Capture system that is as precise as possible, and to make it work in all kinds of environments.

Why a Game

Despite the fact that video-games have always been considered as a hobby limited to younger people only, nowadays they are starting to become more relevant in the entertainment industry. In United States, video-games are considered to be one of the fastest growing industries, and generated as much as $20.77 billion in profits in 2012[6].

Moreover, video-games are also used for educational or training purposes, (i.e. serious games[7]) and the introduction of a Stereoscopic and a Motion Capture component can

[7] a game designed not only for an entertainment purpose. A flight simulator to train pilots is considered a serious game.

open even wider scenarios of utilization.

Summary

Starting from the main motivation of studying a relevant topic, this research focuses on two main topics. Although Stereoscopy and Motion Capture are two technologies that are evolving continuously, they are not expanding as fast among mainstream consumers. The purpose of this research is also to demonstrate that these two technologies are more than adequate for developing good Augmented Reality software, in particular video-games that can also be used for educational purposes[8].

[8] lets imagine a game that allow kids to truly interact with an environment using hand gestures and moving freely

Chapter 2

Background Knowledge

This chapter examines and analyses the knowledge at the basis of this research project. Starting from an introduction to binocular vision in humans and the perception of depth, the analysis will then move on to a brief history of Stereoscopic 3D Visualisation, its function in computer graphics and the long-term side effects (mainly relating to the nervous system and sight) that may arise as a consequence of prolonged use of this type of technology.

The second part of the chapter will then focus on analysing the Motion Sensing devices available on the market today, comparing them with the device used in this research project and highlighting its benefits and shortcomings.

The game engine Unity3D will be the focus of the third part of this chapter, in which the reasons for this choice will be explored.

Finally, the chapter will close with a very brief analysis of the physical aspects of using a real object, in this case a bow, in a videogame. In particular, the mathematical model used in the simulation of the arrow's movements will be set out and explained.

2.1 Binocular Vision and Stereoscopy

2.1.1 Binocular Vision

"What is real? How do you define real? If you're talking about what you can feel, what you can smell, what you can taste and see, then real is simply electrical signals interpreted by your brain. This is the world that you know." (Morpheus' answer to Neo in The Matrix, 1999)

Sight is by far the most important of the senses: through sight, humans are able to immediately gain consciousness of what is around them, subsequently completing the information captured in this way through stimuli coming from the other senses. Sight also enables humans to learn things faster, as can be exemplified by prehistoric cave paintings used to recount historic events, or by paintings in churches used to explain

Figure 2.1: Altamira prehistoric cave - Spain

passages of the Bible or lives of the Saints.

What we see is determined by light being reflected off objects and passing through the pupil, which is then "corrected" by dioptric correction. The light is then reproduced on the retina and subsequently interpreted by the brain. What we see, therefore, is none other than the result of electric impulses driven by the quantity of light that we receive from our surroundings.

The aforementioned principle is valid for all species capable of vision, apart from characteristics such as colour perception and the dimensions of the visual field. In fact, some animals are unable of perceiving the full colour spectrum, while in others the position of the eyes is such that the visual field is larger. In humans, the eyes collaborate in a quasi-absolute way: the two ocular images combine in the brain to produce a single image. In many animals, however, the vision of each eye is independent form the other, resulting therefore in a wider visual field. In the case of eyes that present the same type of perception, the term used is **binocularity**[8], while in the case of eyes with different perceptions, the reference term is **two-edged** vision[1].

Stereopsis is the ability of superior mammals and primates to perceive the depth of space through the binocular system, i.e. using both of the images captured by each eye, in order to obtain a final, single image. Depth perception, furthermore, is assisted also by other monocular elements: information that can be gathered and elaborated by even just one eye (occlusion, perspective, variations in contrast and motion-induced parallax), as well as experience gained by the individual in evaluating the position of an object in space based upon the individual's own experience of the object's properties. Stereopsis, therefore, allows an individual to have greater perception, although there is a limit of 6 to 8 metres from the observer in which stereopsis can function.

Therefore, humans are capable of seeing one singular imagine despite the eyes each producing a separate image at the same time. This is possible because of a mechanism known as **sensorial fusion**[2] between the two distinct images. At the same time, the

[1]Some eye conditions such as strabismus, may cause binocular vision in humans as well.

[2]Motorial fusion, on the other hand, allows the individual to keep both images inside the fovea, due

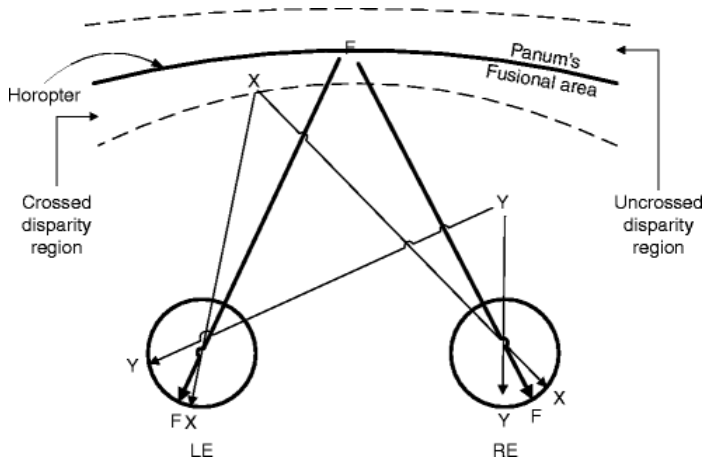


Figure 2.2: Panum's Area and Horopter. Original Source: Robert Earl Patterson Ph.D., Binocular Vision and Depth Perception, Springer Berlin Heidelberg, pp 121-127.

differences between the two images are analysed- a mechanism that allows the individual to perceive depth, as well as greater ease in the recognition of the object[17].

An object projects a pair of retinal points, one for each retina, formed by the object point that is observed in any particular moment. These retinal points are otherwise known as correspondent, as they are both fused into a single point by the visual system. The point in which the ocular axes converge is known as fixation. For each point of fixation, there is a curve known as the **horopter**, composed of all the points of the visual space in which fixation occurs. In order to recognise observed objects, it is necessary that these are perceived as singular- therefore, cerebral cortex continuously adjusts the muscles involved in this operation, so as to ensure that the images fall on correspondent retinal points.

The points F_{LE} and F_{RE} in figure 2.2 represent the correspondent retinal points of the fixation point F, which is perceived as singular. If we consider another point, X, we can notice that it forms another pair of retinal points that, compared to the previous two points, are positioned internally in the right eye and externally to the left eye. The distance between F_{LE} and X_{LE} and between F_{RE} and X_{RE} is known as retinal disparity. Due to the pairing of the correspondent retinal points, point X will be visualised as a singular image. However, the different position of the F points to that of the X points will result in the individual perceiving point X as being to the right of point F. Furthermore, point X lies in a visual area known as **Panum's fusional** area, which extends in front of and behind the horopter. In this area, fusion still takes place and the point will be visualised as a singular image, albeit more distant than point F.

Points that lie outside of Panum's fusional area, on the other hand, are visualised as double. This phenomenon is known as **Diplopia**, or double vision. The images of these points are formed on retinal points that, being unpaired, are known as **disparate** and

to the continuous alignment of the ocular axes.

do not lead to fusion.

Humans do present a **physiological** form of Diplopia, but do not realise this as the brain focuses on the object that the individual is observing, thus ignoring the double images.

In reality, as an object is three-dimensional it is perceived as an ensemble of object points that project their image in a corresponding set of retinal points. As there can be only one point of fixation, the other points fall on the Horopter or Panum's fusional area, and are fused into singular points. However, they generate retinal disparities that, once analysed by the visual system, generate the perception of the objects three-dimensionality. Of course, it is essential that the points fall in Panum's fusional area, i.e. that they are perceived in absence of Diplopia.

2.1.2 Stereoscopy

Despite having always been a natural phenomenon, the first formal definition of Stereoscopy was given in 1832 by Charles Wheatstone: *"...the mind perceives an object of three dimensions by means of the two dissimilar pictures projected by it on the two retinae..."*[27]. Wheatstone demonstrated his assertion by constructing a machine capable of simulating the perception of depth in an image- in this way, the first stereoscope was born. The machine consisted of two cabins placed on either side of the operator, on which were placed two images slightly differing from each other, as seen by the human eye. A system made of mirrors and prisms sent the images back to the eyes, allowing the individual to perceive the optical effect.

Figure 2.3: Mirror Stereoscope

A stereoscopic system must imitate the parallax angle that forms between the eyes in

order to aid the brain in interpreting the perception of depth in the stereoscopic image; Wheatstone's machine was based on this theory, as well as all the other systems created since.

William Gruber's **View-Master**, commercialised for the first time in 1938 by Sawyer's Photographic Service, was the first attempt at presenting this kind of machine to the greater public. Over the course of subsequent years many models have been presented and are still available to this day, but are fundamentally the same: the system is made up of a pair of binoculars and a space in which disks are placed. These disks contain slides that form the stereo image, and contain 14 images in total, although the individual will only see 7 of them: in fact, each image visible to the individual is made up of two slides, one for each eye.

Figure 2.4: View Master Binoculars and disks

In the computer era stereoscopy has largely been revisited in particular by the film and video game industry, bringing about a distinct revolution in stereoscopic techniques.

Many stereoscopic models exist, ranging from images visible to the naked eye to complex digital systems. This chapter will now cover the most relevant models of the present day.

Anaglyph

This system, widely used by the film industry in the 1960s, is made up of a pair of monochromatic stereoscopic images, each with a different dominant colour and mounted on the same support. The images must be viewed using a particular filter, so as to enable each eye to observe its relevant image. The coloured filters used most widely are red, green, blue and their complementary colours: cyan, magenta and yellow. The choice of colours to be used is determined by the fact that any given colour cannot be present on both images. The overall effect of the Anaglif is less than satisfactory, as it

is impossible to view images in full colour and the brightness of the images is highly limited. On the other hand, the model is extremely cheap to put together.

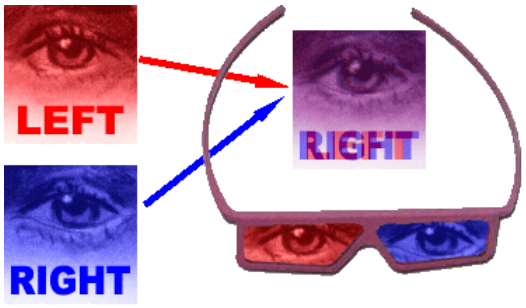


Figure 2.5: Anaglyph System

Polarised Light

A highly complex system used in the film industry, the Polarised Light model uses to projectors with polarised lenses and particular glasses used to view the images. The glasses used also have polarised lenses in order to enable each eye to view its relevant image. This system is evidently more expensive than the Anaglif, but produces much better results. The same effect can be obtained by projecting the images on a canvas on which a polarised filter has been applied. The greater part of 3D televisions currently available on the market is based on this technique.

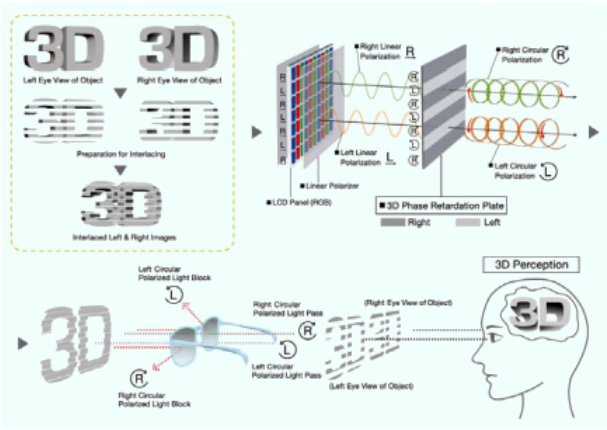


Figure 2.6: Polarised Light

Shutter Glasses

Invented in 1922 by Laurens Hammond and William F. Cassidy and known at the time as Teleview, this system entails an alternated shuttering of each eye so that a single frame is viewed by the correct eye. The first prototype of this model used a pair of glasses equipped with an automatic shuttering system. Towards the end of the 20th century, this system was revisited and applied to an electronic digital system using

glasses with LCD lenses. These are then coordinated by a processor, and the system can be used both at the cinema and at home, and linked to PC's, consoles and more. Using an appropriate type of software, the processor sends the left-hand and right-hand images alternately to the projector or display. In order for this to be possible the display must have a frequency of 120Hz- exactly double that of a regular display. The glasses are then synchronised with the projector, so as to free each eye at the right moment.

Auto-stereoscopy

With this system, the images can be viewed in 3D without the aid of external instruments positioned in front of the observer's eyes (such as glasses, stereoscopes, etc.). In fact, the support is already equipped with the appropriate technology that will hide a certain image from the eye that is not supposed to view it. The most widely used specimen of this system is the parallax barrier.

2.1.3 Stereoscopic Video Games in the past

The need to create more realistic and impactful video games has spurred developers to experiment with new and innovative types of gameplay.

The major breakthrough came in 1969 with Ivan Sutherland's Sword of Damocles. Despite not being an actual video game, it is considered to be the first ever attempt at creating a 3D virtual reality system. The experiment was composed of a helmet equipped with two small CRT screens, one for each eye, connected to an enormous mainframe, and the helmet displayed a simple stylised cube in stereoscopic 3D.[26] The system was named the "Sword of Damocles" because the helmet was fixed on to the ceiling due to its excessive weight- just like the Greek prince Damocles' famous sword.

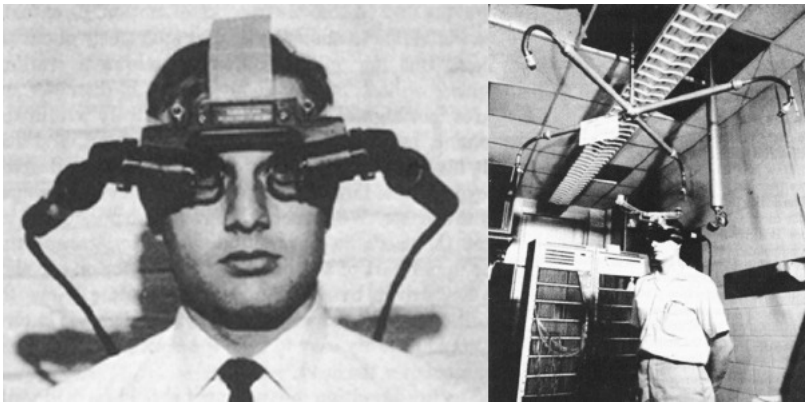


Figure 2.7: The Sword of Damocles

In 1982, 3D Stereoscopy enters arcades with the SubRoc3D arcade game produced by SEGA. The game was equipped with a binocular visor, in which a rotating disk was placed, alternately shuttering each eye. The game environment was made up of

Figure 2.8: SEGA SubRoc3D console, separate from its cabinet

hand-drawn elements placed in a 3D environment.[24] The aim of the game was to sink the largest number of enemy ships, using a submarine.

The TomiTronic3D and GCE Vectrex 3D (1983 and 1982, respectively) were the first attempts at bringing 3D video games into the homes of consumers. The former was the pioneering portable 3D console, while the latter the first 3D peripheral for consoles.

Figure 2.9: The TomiTronic3D (left) and GCE Vectrex 3D (right)

In 1995, Nintendo attempted to create its own "portable" 3D games console: Virtual Boy. This, sadly, revealed itself to be one of the company's biggest failures. Made up of a tripod-based helmet connected to a joypad, the game was discontinued by Nintendo itself after a year after several customers complained of severe headaches caused by Virtual Boy games sessions.

2.1.4 Stereoscopic Video Games Today

In 2010, Sony releases new firmware for the PlayStation3, adding a stereoscopic support. In the same year, several 3D-supported games are released and 3D gaming sees its golden moment.

Figure 2.10: Nintendo Virtual Boy

The next year, in 2011, Nintendo releases its latest portable console, the Nintendo 3DS, which allows individuals to play stereoscopic games without auxiliary instruments.

Despite a couple of booming years, 3D technology has now somewhat slowed down due to decreasing interest from the greater public. The 3DTV market is winding down and the Nintendo 3DS is currently being singled out as the company's latest flop.[21]

The problem lies in the fact that 3D visualisation has been understood until this moment as a graphic effect that makes video games "cooler", adding nothing to the actual game-play. Furthermore, the graphic quality of the game remains poor as the console needs to go through the same process twice, elaborating a separate image for each eye. Another obstacle to the diffusion of stereoscopy in video games lies in the quality of 3D displays, as in order to achieve the best possible 3D experience the monitor must be as extended as the individual's visual field. The size of a monitor is directly proportional to its cost, and very few gamers are willing to spend the amount of money required for the best possible type of 3D system.

2.1.5 Medical Consequences

A common perception is that 3D visualisation is harmful to eyesight, and generates painful headaches. This kind of reaction in the human body is the result of the optical illusion created by stereo images.

In nature, accommodation[3] and convergence[4] are closely linked, as they enable the individual to visualise and focus on objects that are placed at a certain distance. Each

[3]the process by which the eye increases optical power to maintain a clear image (focus) on an object as it draws near

[4]simultaneous inward movement of eyes toward each other to maintain the observed object always in sharp

time an individual observes an object, the effort required of the eye muscles is directly proportional to the object's distance from the observer.

In the visualisation of stereo content, the link between the two aforementioned mechanisms is interrupted, as the eyes both adjust to the level of the support that visualises the image rather than where the object is projected by the stereo effect. At the same time, however, both eyes converge on the 3D object. This condition is known as discordance between accommodation and convergence, and is the principal cause of 3D-related visual strain.[12]

For the same reason, it is clear that stereoscopy is possible only for individuals that do not suffer from visual conditions that cause this type of discordance, such as strabismus.

2.2 Motion Sensing Devices

In 1984 the first mouse appeared with the Macintosh 128K. This innovative input system shook the world of IT, as it was now possible to use hand movements to interact with a computer.

For the same reason, the introduction of the graphic board revolutionised the world of digital graphic design as it allowed designers to use familiar movements – such as those required to draw with a pencil – in order to develop digital designs.

Motion sensing devices have the same goal: to use movements that are natural to human beings in order to interact with digital devices. Examples of such movements are turning the page of a book, or moving a steering wheel in order to simulate driving.

Nowadays, the videogame industry has reached a level where the graphic models used in videogames are extremely close to reality. Therefore, the only way to create new games able capable of attracting more consumers is to offer new experiences in game-play. Many companies have turned to motion sensing devices to achieve this aim.

The following paragraphs are dedicated to a description of some of the devices that are at the moment changing the world of video games, as we know it.

2.2.1 Microsoft Kinect

Released by Microsoft in 2010, the Kinect is a revolutionary input device capable of capturing body movements and of using them to interact with digital content. Initially only available for the Xbox, the Kinect was then released in 2012 with a kit that allowed it to be used with a PC.

The device is equipped with an RGB camera, a depth sensor and an array of microphones. The depth sensor is an infrared laser combined with a CMOS sensor, capable of capturing 3D data under any light conditions. The sensitivity of the sensor is adaptable in relation to the gameplay, the individual gamer's physical conditions

and any obstacles present in the device's field of action. The microphone allows a localisation of sounds, as well as noise reduction.

Figure 2.11: Microsoft Kinect

Unfortunately the device does not capture the movements of all body parts, as it is unable of recognising individual body parts such as the fingers. Furthermore, it cannot capture movements made by objects, but only those made by the human body.

2.2.2 Nintendo Wii Remote

Released in 2005 along with the Wii console, the Wii remote is considered to be the greatest revolution in console-based gameplay.

The remote is formed of an infrared camera placed at the top, and an accelerometer on the inside. These allow for the manipulation of objects on the screen through the movement and pointing of the device.

The controller is capable of capturing movements on three axes through the use of the accelerometer and of moving a pointer on the screen through the camera and a sensor strip placed along the display. The sensor strip is formed by 4 LEDs in line that allows the Wii Remote to locate its physical position.

In subsequent years, the device has been upgraded several times through a communication portal placed at the base of the controller. Examples include the Wii Motion Plus, which captures even more complex hand movements, the Wii Vitality Sensor, that reads the individual's vital stats through an oximeter, and the Numchuck, a type of joystick equipped with an internal gyroscope that allows the expansion of the movement area.

This controller is capable of capturing the movement of a single hand (or of both, in the case of the Numchuck) but not of the entire body. Therefore, the gameplay of an

FPS (First Personal Shooter) could be easily enhanced, while this would not work in the case of a football game.

Figure 2.12: Nintendo Wii Remote

2.2.3 Optical Tracking

In order to develop a device capable of recognising the movement of individual fingers or of the head, or capable of achieving the greatest precision possible, the two devices discussed above are not suited to the task. In order to achieve this, it is necessary to turn to the family of devices known as Optical Tracking Systems.

Largely used by the film industry (e.g. Avatar) and in the development of certain video games (such as Splinter Cell: Blacklist), these devices are the most accurate currently available on the market.

The devices are based on a simple principle: a set of three or more cameras are installed in a quasi-circular space, and markers made of a particular reflecting material are placed on the object or individual that is to be traced. During the capture session, the cameras trace the movement of the markers and transmit these movements (under the form of X, Y and Z coordinates) to the computer, which elaborates them for the relevant application.

There are two types of optical tracking systems: active and passive. The former is the most common and cheapest, as the markers are small spheres made of a particular material that allows the light reflected from the cameras to be once more received by them. The latter works on the same principle, although the markers themselves emit a form of light (usually infrared) that is then captured by the cameras.

An example of these devices is Natural Point's OptiTrack camera set- this device was also chosen for the development of this project.

Figure 2.13: Ubisoft's state-of-art performance capture studio in Toronto, Canada (Ubisoft)

2.3 Unity3D: a Game Engine

Unity3D is a cross-platform engine developed by Unity Technologies. Born in 2005 as a development tool for games on OSX (Apple's Operating System), today it is one of the most widely used tools in the development of video games.

Developed in C/C++, it supports code written in C-sharp, Javascript and Boo and allows for these to be simultaneously used in the same project. This tool allows the development of video games that can function on PlayStation 3, Xbox 360, iOS (Apple's mobile operating system), Android, PC Windows, Linux, OSX and web applications.

Unity has its own internal tool for the construction of virtual video game environments, but also supports models imported from other programs such as 3DStudioMax, Maya, Blender and many others. It also supports the widely known physical engine NVIDIA PhysX, for the simulation of more complex physical effects and the ulterior enrichment of the game play.

Unity3D comes in two versions- free and pro. The former has all the standard characteristics needed to develop a video game with this engine, while the pro version also includes support at diversified levels of detail for better features, better audio filters, audio and video import and reproduction, a support with dynamic lighting and much more.

Last but not least is the worldwide community that offers help and advice as well as contributing to continuously improve this software.

Unity's philosophy is based on the following principles:

- **Scene:** the virtual location in which the game is set. A scene can be an entire level or a single part of it as it is often better to distribute a level in multiple scenes in order to increase the loading quality.

- **Assets:** The most basic of the blocks that make up the video game. These are

the resources used to build the game such as texture, scripts and XXX. Assets are assigned to game objects and prefab.

- **Game object:** The active objects present in a scene. It is possible to create empty ones (that only have the possibility to transform- i.e. to be rotated or moved), or to choose from the pre-made ones.

- **Prefabs:** Sometimes a developer needs to reuse the same game objects in many scenes and many times. Prefabs allow the memorisation of complete objects for them to be reused at a later time.

- **Components:** particular objects that can be linked to the game objects in order to change their behaviour. Functionality can be added and their appearance can be changed. Common components in Unity are particle emitters, cameras and lights.

The interface is composed of five principal views:

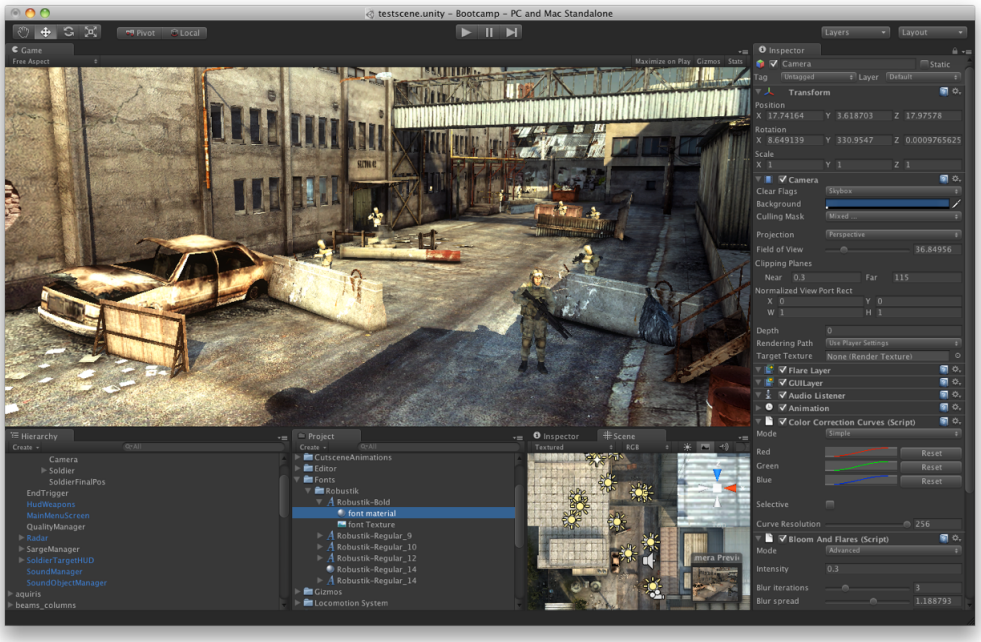


Figure 2.14: Unity3D interface

- **Scene:** The section in which it is possible to add, move or remove the active objects in a scene. In other words, this is the visual editor in Unity.

- **Game:** Here the entire video game can be previewed through the principal camera (if present). When test mode is on, the entire game can be tested, although the more advanced effects are not active.

- **Hierarchy:** This lists all the active objects present in the actual scene. Through this list, it is possible to quickly select an object or add a particular activity or component to an active object.

- **Project:** A global vision of the project file. From here, it is possible to select the activities and components imported from other programs, or downloaded from other sources.

- **Inspector:** After having selected something in the hierarchical or project view, in this section it is possible to view and modify its properties. If a script is selected, it is possible to preview the written code on the inside. From here it is also possible to add something to a selected object.

2.4 Physical Model of a Bow

Used since ancient times by the human race to fight and hunt, nowadays the bow is used principally as a recreational activity. Normally, when an individual throws an object with his or her arms, the final speed of the object is determined by the kinetic energy transferred from the muscle. People with longer arms or more muscle are naturally more capable of throwing the object further and at a higher speed. A force-speed relation limits the power generated in this way and, even when the muscles contract at the right time and release the object at the right speed, the force lost in the throw is more than half of the total force.

A bow aids this process as the muscles are able to work at a slower pace and most of the work is done by the bow in the form of potential elastic energy. This energy is then transferred to the object (arrow) when the bow is released, giving it a much higher speed and allowing it to travel a much greater distance.

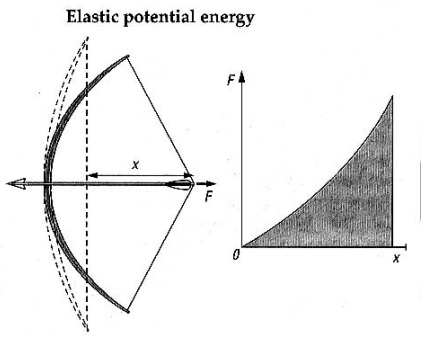


Figure 2.15: physical model of a bow, and efficiency curve

As illustrated in Figure 2.15, the force (**F**) applied to the arrow is correlated to the distance by which the chord is moved from its resting position to a position of tension

(point x in the figure), and is tightly correlated to the potential energy stored by the bow as the chord tenses. This relationship is described as such:

$$E_{pot} = \frac{1}{2} * k * x^2 \tag{2.1}$$

In Equation 2.1 **k** is a constant known as the **elastic constant**, and is linked to the type of bow used (a long bow will have a different k to that of a short one). The greater k is, the more energy will be needed to tense the chord (for example, Ulysses' bow must have had a k tending to infinity).

As described in Equation 2.2, when the chord is released, the potential energy is transferred to the arrow in the form of **kinetic energy**, minus a small part that dissipates through friction with the air (this is illustrated as μ). In the end (Equation 2.4), the initial speed of the arrow will be the following (**m** being the arrow's mass):

$$\mu * E_{pot} = E_{kin} \tag{2.2}$$

$$\frac{1}{2} * \mu * k * x^2 = \frac{1}{2} * m * v_{arr}^2 \tag{2.3}$$

$$v_{arr}^2 = \sqrt{\frac{\mu * k * x^2}{m}} \tag{2.4}$$

Chapter 3

State Of The Art

3.1 Stereoscopic Human Interfaces

Tele-operation is widely applied to robotics in order to operate in a remote context trough the utilisation of a robot. In their paper[7], Ferre et al. illustrate the possibility of adopting a Stereo-3D visualisation system to obtain a better representation of the remote operation space. Indeed, when the operator seeks a precise robot guidance, in order to manipulate objects as well (for example thorough the utilization of robotics arms), it is necessary to obtain a better perception of the remote environment. Please note that a monoscopic system is better, where either only a supervision of the robot's movements and a remote programming procedure are required.

In humans, each eye has a field of view angle of 120°, while the region where the images are captured by both eyes is 50° wide. The images captured in this smaller area are correctly merged by the brain, generating a depth perception of the individual's surroundings. This merging mechanism is realised through the combined work of the different parts of the eye. Reproducing this mechanism is the largest obstacle in developing a 3D-Visualization system capable of reproducing a realistic depth perception. Indeed, it is fundamental to keep in mind that no 3D visualization system is as effective as than as binocular human eyesight. Taking this as assured, it is still possible to design a binocular camera set-up that partly reproduces the natural human stereoscopic vision.

The solution is to use two cameras with a focal length equivalence of 50 mm, placed at a distance of 6-7 cm from each other and with a small angle (less than 25°) between them. The angle between the two cameras is a key point in determining where the images are properly merged, and the dimension of the effective working area. The effectiveness of the system is calculated using Equation 3.1, that calculates the maximum projection disparity achieved by the aforementioned system.

$$projectionDisparity(d) = 2 * focallenght * \tan\left(\frac{\alpha}{2}\right)\left|\frac{H}{d} - 1\right|, \quad \text{with} \quad \alpha > 0 \quad (3.1)$$

Using the same formulation, it is possible to calculate the maximum projection disparity in the case of a parallel axis camera system ($\alpha = 0$).

$$projectionDisparity(d) = focallength * \frac{O}{d}, \quad \text{with} \quad \alpha = 0 \tag{3.2}$$

Please note that H represents the distance between the cameras and where their axes intersect, while O represents the interaxial distance between the two cameras. α is the angle between them and d the distance between the camera setting and a generic observed object.

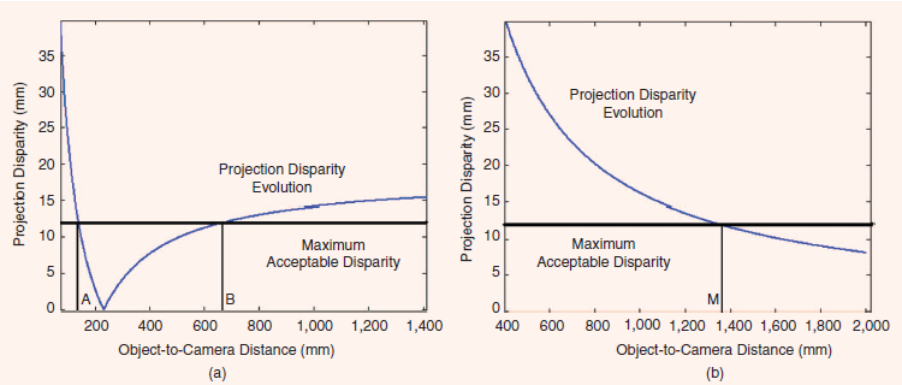


Figure 3.1: It is possible to calculate the effective working area, according to the maximum projection disparity. In the case of $\alpha > 0$ (a) the objects placed at a distance between A and B are correctly merged. On the other hand, in an $\alpha = 0$ configuration (b), the object must be placed further than 1200 mm (point M) to be correctly fused.

Figure 3.1 illustrates the results of the above formulas, compared to a maximum acceptable disparity defined as 12 mm. This value depends on the user-to-screen distance (in this case 45 cm) and the human thresholds (1.5°) for image disparity. It is possible to observe that the effective working area dimension varies considerably between the two configurations. A system with crossed axes is better if used when the object is near the cameras, meanwhile a parallel axes configuration is better for objects that are placed further away. The best solution is to have a system that is able to adapt the angle between the two cameras by itself, depending on the distance from the observed object.

The above theory is also applicable to the context of virtual reality, albeit taking some considerations into account. The biggest obstacle is represented by the different typology of cameras used to develop the ArcheryGame video-game. The virtual cameras utilised in Unity3D are able to sharply visualise all the objects located inside the Field of View, whether they are placed really close or at a distance up to infinity. In Chapter 4 & 5 we will see how to apply this theory to calculate the effectiveness of a Toed-in stereoscopic system realised in Unity3D, which is able to vary the α angle between the two cameras.

3.2 Immersive 3D User Interface for 3DTVs

Stereoscopic-3D visualisation can also be applied to designing Immersive 3D User Interfaces, which can be applied to many fields. An Immersive 3D environment allows users to see 3D objects and still have the perception of their own body, while interacting with the virtual components. The floating object produced with Stereoscopic-3D visualisation can compose an interface that allows users to interact with those objects as naturally as in real life.

Du et al. [5] analyse the realisation of this kind of User Interface, solving some important technical issues particularly relating to the user experience.

Nausea is eliminated by applying a constraint directly related to Martin Banks' "1/3 dioptre" law[16][3]. This law is expressed by Equation 3.3, where z is the perceived distance from the virtual object and d is the user-to-screen distance.

$$\left|\frac{1}{z} - \frac{1}{d}\right| \leq 2.3 \tag{3.3}$$

The perceived dimension of the object is also an important issue when realising an immersive 3d user interface. If too small, users may experience strong difficulties in interacting with the floating object. At the same time, if too big, the object may not be displayed correctly by the visualisation system.

Equation 3.4 illustrated by Du et al. allow the calculation of the maximum acceptable dimension (WxH) of a floating object, in relation with:

- Screen Size w
- Watching Distance d
- Interpupillary Distance p
- Screen Height h

$$\begin{cases} W = \dfrac{z(w+p)}{d} - p \\ H = \dfrac{z * h}{d} \end{cases} \tag{3.4}$$

The solution takes into account the realisation of a User Interface where the Stereo-3D floating object is only defined by its height and width. In our case, the virtual arrow is also characterised by its depth, which does not influence the user experience if maintained proportionally to the other two dimensions.

In realising the Archery Game, the above formulation was used in order to establish the actual dimension of the floating object at a certain distance z calculated by the OptiTrack System. In this way, it may be possible to enhance the Stereoscopic effect in relation to the perceived distance, and to scale the object dimension when it changes.

3.3 A Bow as an Interaction Device

The idea of an Archery Game was taken from two past related works that explored the realisation of a Virtual Reality game where the user can interact with the virtual world through the utilisation of a real bow. Both works rely on the utilisation of optical tracking to capture the bow movements performed by the user, as well as on stereoscopic 3D visualisation to visualize the real environment.

In *Deer Hunter*[23], the authors use a system formed by two cheap security cameras located on the left and right sides of the player. These cameras are equipped with a IR emitter for night vision operation. At the same time, a set of three IR reflective markers are placed on the bow, that can be seen by the two cameras. Using global thresholding they are able to distinguish those pixels that contain the information relative to the reflective markers. With triangulation it is possible to calculate the exact spacial position of each marker. Through network communication, the information relative to the position of the bow and its orientation are sent to a Virtual Environment Client, which will use them to interact with the virtual world. The game has also been designed to work in multi-player, where two users located in different rooms (as well as at greater distances) can compete inside the same Virtual Environment.

Bow Operated Virtual World[1] expands the previous work, using a Panorama Screen and implementing new bow actions as well. Indeed, the user is now able to move freely inside the Virtual Environment. The Panorama is realised using a curved screen that completely covers the user's field of view. A set of different projectors is needed, which uses an active shutter glasses system to visualise the Stereo-3D images.

These two projects were used as the starting point for developing the new ArcheryGame. Despite the fact that both projects aim to realise an immersive game, they do not focus on the Stereo-3D component, but merely implement a simple system that is able to visualise the images giving a moderate depth perception to the user. The ArcheryGame 2.0 analysed in this master thesis aims to focus particularly on the Stereoscopic component, bringing the user further inside the Virtual Environment.

Chapter 4

Proposed Investigation

The aim of this research is to develop an innovative technology framework that allows for a new type of human-computer interaction, where the recognition of the user's movements is combined with immersive visualisation. The ultimate goal is a simple to use and more effective touch-less type of human-computer interaction, compared to what is currently available. The proposed new concept is applied to an innovative game, where players use their own body movements in the surrounding 3D space and provide commands to the game. The game of reference is the 3D Archery Game, which is presented in the next section. In this game, it is possible for a user to interact with and play the game by using hand gestures while effectively benefiting from S3D visualization. The proposed research and development activity aims to demonstrate the feasibility of this new type of interaction. In order to achieve this goal, the following objectives were set out:

- **Motion capture.** Develop an accurate motion capture technology setup that provides positional input to the game engine.
- **3D Visualization.** Develop an accurate 3D visualization method, which allows the integrated system to display 3D visualization information consistently to the motion capture.
- **Integrated System.** Develop an integrated system capable of managing motion captured positional information, 3D visualisation, and computer graphics objects visualised according to the 3D visualisation method developed in the previous point.

The following sections describe the proposed idea and the motivation behind this research.

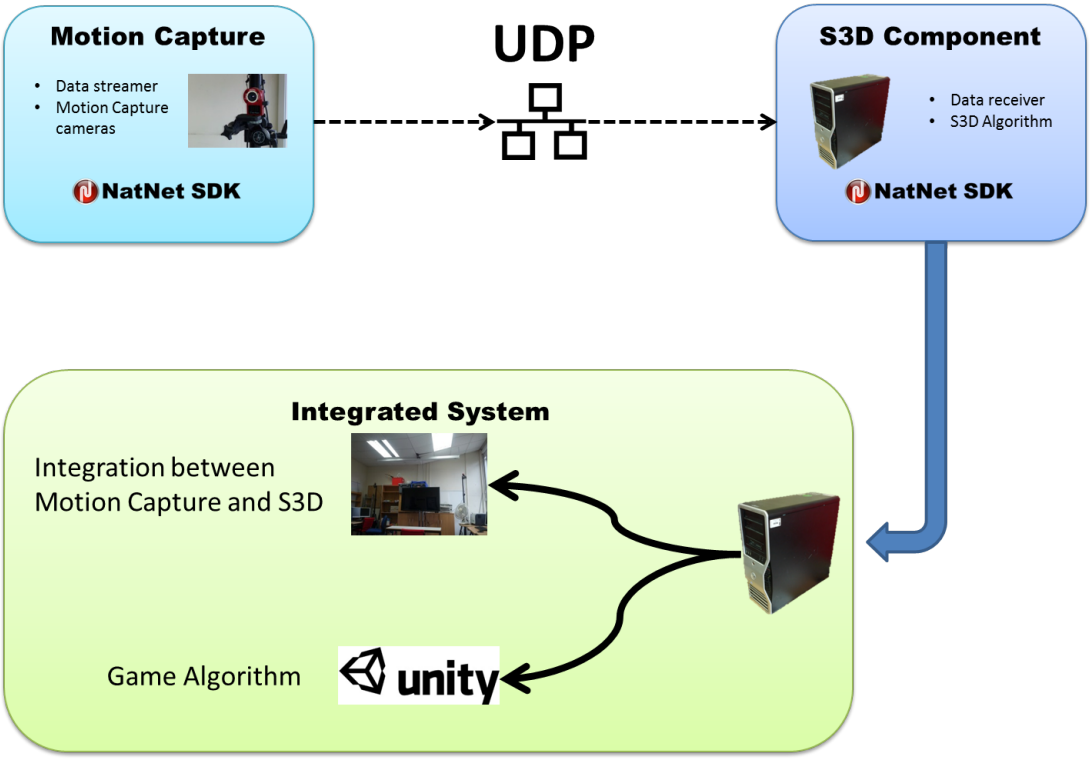


Figure 4.1: Proposed solution diagram

4.1 3D Archery Game: Potential and Shortcomings

To test the main idea, it was decided to develop an improved version of an Archery Game (formerly know as ArcheryGame1.0), developed by the author in 2013. The game consists of the possibility for the user to shoot virtual arrows using a **real bow** - the bow is then tracked by the motion sensing system. Moreover, the 3D Visualisation system allows the user to see the **virtual arrow** superimposed on the real bow.[22]

Potential	
New type of interaction	The system allows the user to interact with the virtual environment using a real object. In this case, a bow that can be chosen based on the player's requirements
Full-immersive gameplay	The 3D effects transform the room in which the system is installed in the game environment. The player and the environment around it are part of the game.

Idea with infinite potential	The ArcheryGame is only an example of the basic idea. Following the same principle, it is possible to develop other types of applications, not only related to the field of video games.

Shortcomings	
Limited interaction	The user is only able to rotate the field of view. The virtual arrow cannot follow all the movements of the bow, which means that the arrow will not be seen exactly superimposed on the bow.
No system integration	The motion capture and the stereoscopic systems are not able to communicate between each other. The arrow is rendered based on an approximate position of the user.
Basic tracking performances	The motion capture does not track position, but only the rotation angles (yaw, pitch, roll). Tracking is limited only to one object at any one time.
Basic stereoscopic effect	The game is not designed for the S3D. The stereoscopic visualisation is achieved by converting the 2D rendering in 3D, using a feature integrated into the GPU. Due to this, the user is not able to customize the S3D effect, but only to (slightly) change the depth perception.
Game affected by lags	The game runs on the same machine that has to elaborate the motion capture data. The workload may then become very high.

4.2 Proposed Solution

A Virtual Reality game is defined as *a game which is playable using only the body movements of the player or through the utilisation of a real object (like a bow), which is not directly connected in any way to the pc on which the game runs.* In this way, every object can be used as a game controller -from a bow to a blade- without any limitations

in player fantasy or requirements.

A video game, especially an Archery Game, needs a game controller that must be reactive and as accurate as possible in reflecting the player's actions in game. The principal features that must be taken in account to choose the Motion Capture system to be adopted as game controller are **accuracy** and **reliance**.

The previous section highlighted the limitations present in the first version of the game. In order to make a game playable in stereoscopic-3D visualization, it is necessary to redesign the motion capture and the stereo-3D module, and to implement an effective connection between them.

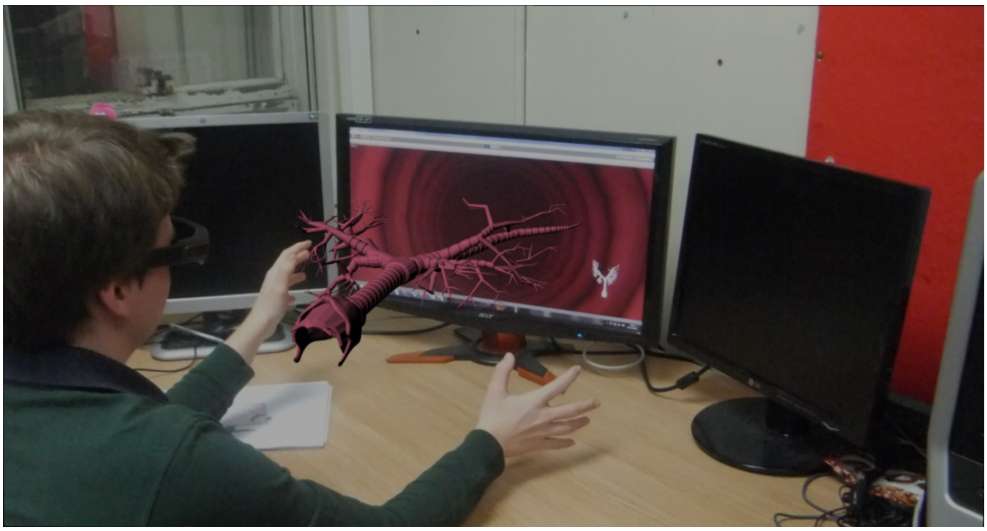


Figure 4.2: Augmented Reality medical application using Motion Capture and Stereoscopy-3D. Concept picture

4.2.1 Motion Capture Module

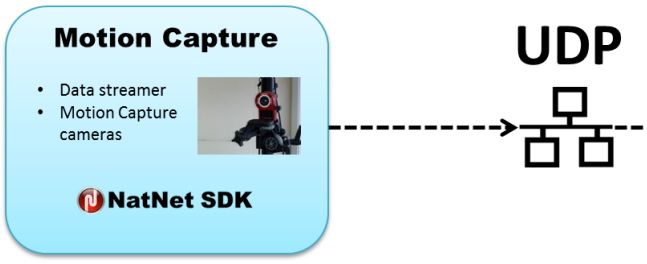


Figure 4.3: Motion Capture Module diagram

Motion capture devices (also known as **motion sensing devices**) are capable of recording the movements of objects or people.

The first version of the ArcheryGame was implemented using an Optical Tracking system, capable of detecting the movements of a real bow on which some passive

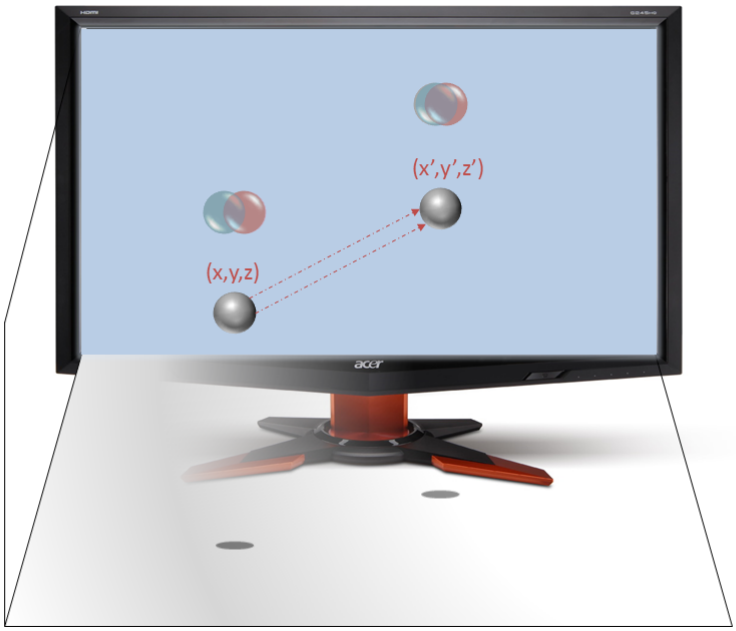


Figure 4.4: The virtual space is superimposed onto the real one, delimited by the area in front of the screen.

markers are installed. The first procedure proposed to detect player movements was not complete and had some shortcomings.

The Virtual World superimposed onto the Real World

A motion capture system is a technology that is able to continuously detect the position of an object moving inside a real space. To develop an Augmented Reality game that uses this technology to allow the user to interact with the virtual object, it is necessary to translate the real environment -in which the user operates- into the virtual one, where the interface components are located (Figure 4.4).

The movement of an object is characterized by the change of its positional coordinates along the axes (x,y,z) and by the rotation along them (pitch, yaw, roll). It is not possible to identify the virtual object inside the real world using the same coordinates of the virtual environment. Indeed, we need to align the real and the virtual world,and also to align both coordinate systems. To do so, we must identify a unique reference system on which we can translate the coordinates of both the real and virtual space. There are two ways of obtaining this: the first one is to create a virtual space which is based on the same reference system of the real space, while the second one is to translate into real space coordinates the virtual space coordinates.

An example is shown in Figure 4.4, where the real world is represented by the area around the screen. The objects (two little grey balls) are visualised through the 3D

Visualization system, in order to appear inside the real world.

This problem, related to the reference system, is strictly connected to the kind of motion capture technology that will be used to track the user's movements.

Three motion capture systems have been considered, the features of which were analysed in order to choose the most appropriate system for the game. Those systems are **LEAP Motion Controller**, **Microsoft Kinect** and **Natural Point OptiTrack**. The following table gives a brief view of the features of interest. It should be noted that hFoV and vFoV stand for Horizontal and Vertical Field of View, and represent the view angle of the embedded cameras. FPS stands for Frame per Second, and it is the frequency on which the images are captured by the system.

Device	**Full-Body**	**Marker-less**	**Range**	**Frequency**	**Accuracy**
LEAP	No	Yes	150°hFoV	290FPS	1\100th mm
Kinect	Yes	Yes	57°hFoV 43°vFoV	30FPS	1-2mm
OptiTrack	Yes	No	46.2°hFov 34.7°vFoV	100FPS	sub-millimiter

The LEAP Motion and Kinect are Marker-less systems. They are simpler in terms of installation and configuration. It is not necessary to place markers on the player and, usually, they are already calibrated to be used out-of-the box. On the other hand, they are able to capture the human body rather than objects. The Microsoft Kinect full body capture is limited to 20 articular joints, which means that the fingers are not recognised. Leap motion, on the other hand, is able to capture the fingers and most common hand gestures, but is not capable of full body recognition.

After an accurate analysis, it was found that the OptiTrack system satisfied the project requirements. It allows for tracking both body movements and objects manipulated by a user, by placing IR reflective markers on the object to be tracked. The shortcomings of this system include the necessity to have at least 3 cameras -OptiTrack is usually expensive- that must be installed around the tracking area. Furthermore, the calibration process is not simple. Nonetheless, the system is highly configurable and can easily be adapted for several applications.

Kinect and Leap motion have their own reference system, which requires specific mapping between real and virtual coordinates. Using the OptiTrack system it is relatively simple to do so.

The main theory behind this is very simple: the origins of the virtual and real world must coincide, as well as the direction of the axes.

The OptiTrack system maps the captured area to a virtual space. This mapping is based on metric units. In principle, is straightforward to map the OptiTrack virtual space onto the Unity3D virtual space, if fixed reference points are provided.

It was decided that the common coordinates system origin was to be located at the bottom of the **visualization screen** in use. This origin of the coordinates system is supposed to be estimated by placing markers on the monitor. Once this position is known, all objects can be related to the common coordinates system.

Each object visualised is characterised by two sets of information. The first describes the object's centre position (along the x, y and z axes), while the second one describes the rotational position (pitch, yaw, roll).

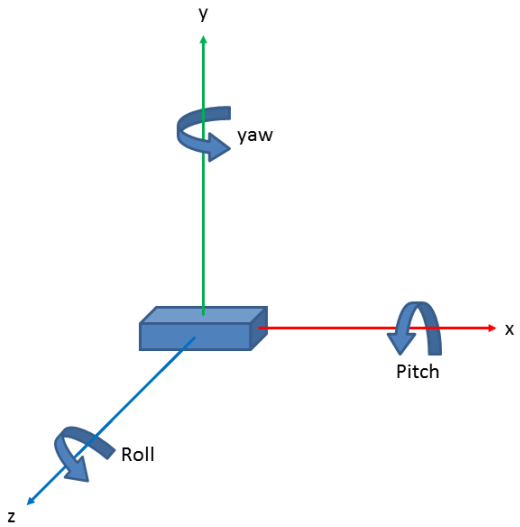


Figure 4.5: An object is characterised by the (x, y, z) coordinates and the rotation along the axes (pitch, yaw, roll). The reference system is Right-Handed.

Using the OptiTrack data

The script that receives and processes the data in Unity3D from the OptiTrack System, was provided by the system vendor (Natural Point). It was a generic example able to track the movement of a single Rigid Body or a Skeleton[1]. This research aims to use three Rigid Bodies and, in order to do so, it was necessary to analyse in depth the communication mechanics and the different typologies of frame transmitted from the OptiTrack. Given that no documentation was provided to guide the improvement of this communication script, the analysis below was produced by the author and can be used as a starting point for future improvements.

Once the real object is correctly tracked by the optical system, the data must be analysed and then processed by the game engine, in order to visualize the game interface.

The communication channel has been designed to work on network bases, using **User Datagram Protocol** (UDP) and a socket connection between the server on which the optical tracker is installed and the client, where the game engine runs. The server sends each packet through the networks, using a multicast address, allowing several clients to receive the data. If a multicast connection is not available, it is possible to set up a unicast connection specifying the server IP address on the client side.

UDP is appropriate for data transmission where it does not matter if one packet is lost during the communication, rather than the data streaming arriving to the receiver as close to live as possible. Moreover, the OptiTrack server allows to transmit up to 100 frames per second, and the loss of one or two frames does not influence the quality of the motion capture.

The OptiTrack server uses a proprietary library to stream and receive the data through the network. This library works as an interface of communication (using the scheme explained above) and depacketisation of the data received by the client. The Figure 4.6 represents the exchange of data between various kinds of client and the server. Two-way communication allows the two parties to exchange information. In particular, the client asks who can transmit and the server sends the data to the requesters. The game engine Unity3D utilised to develop this project is not present in the list of NatNet SDK supported clients. It was necessary to design and develop a custom client application in order to receive and read the frames sent by the server[2].

The server sends the frame through the network using a buffer, and its dimension depends on the number of markers that are present inside the captured scene. Using an index, which is updated every time the application reads a set block of information, it is possible to go through the buffer and read all the data inside. Inside a UDP packet are stored not only the data to be transmitted, but also the address information to reach

[1] A Skeleton is a set of Rigid Bodies grouped together to simulate the movement of a more complex object, such as a human body

[2] The complete implementation of the client will be explained in Chapter 5

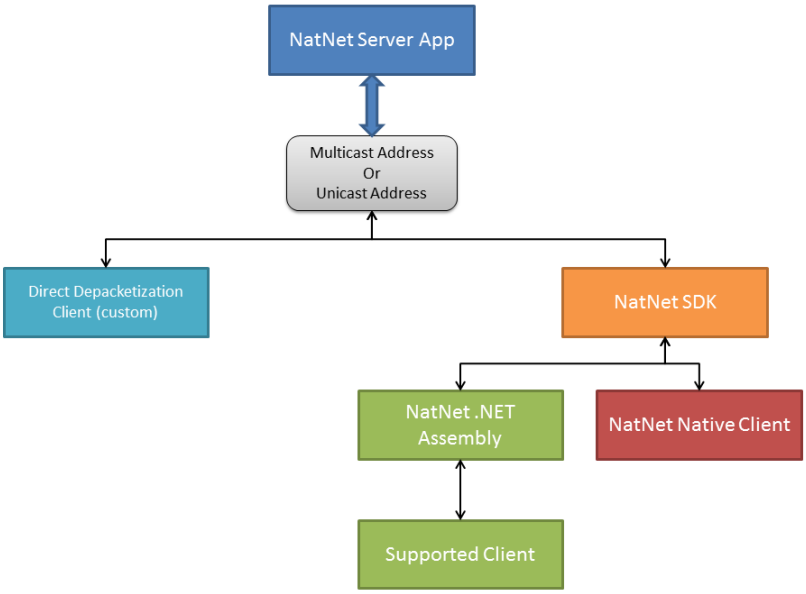


Figure 4.6: Typologies of client/server communication using the NatNet SDK. The library is provided as support for some clients, otherwise it allows the programmer to develop a custom depacketization client.

Offset - Octet		0							1							2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source Port																Destination Port														
4	32	Length																Checksum														
		Message (Data)																														

Figure 4.7: The UDP header is formed by 4 fields of 2 bytes each (16 bits). The offset index starts from 0 and the length of the message can be maximum 65,535 bytes (8 byte UDP header + 20 byte IPv4 header + 65,507 byte data).

the client. For this reason is important to discern between the two typologies of data so as to analyse exactly from which index position the marker information starts. Figure 4.7 illustrates the UDP packet structure. The first 16 bits of the frame are composed by the header, which contains the **Source** and **Destination** port, the total **length** of the message and the **Checksum**.

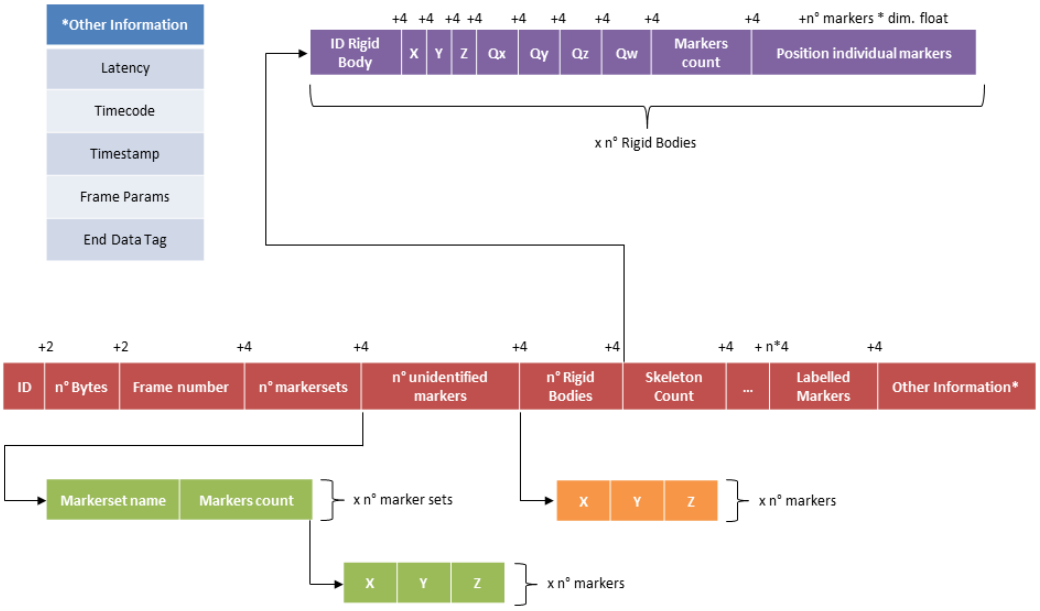
The communication between the Server and the Client happens using two different ports. The **Command** port is used to exchange the command requests between the two parts, while the **Data** one is used to transmit the motion capture data. Every OptiTrack frame carries an ID Message, which denotes what kind of data are inside the packet. The complete list of those IDs are listed in the following table.

MESSAGE	**ID**
PING	0
PING_RESPONSE	1
REQUEST	2

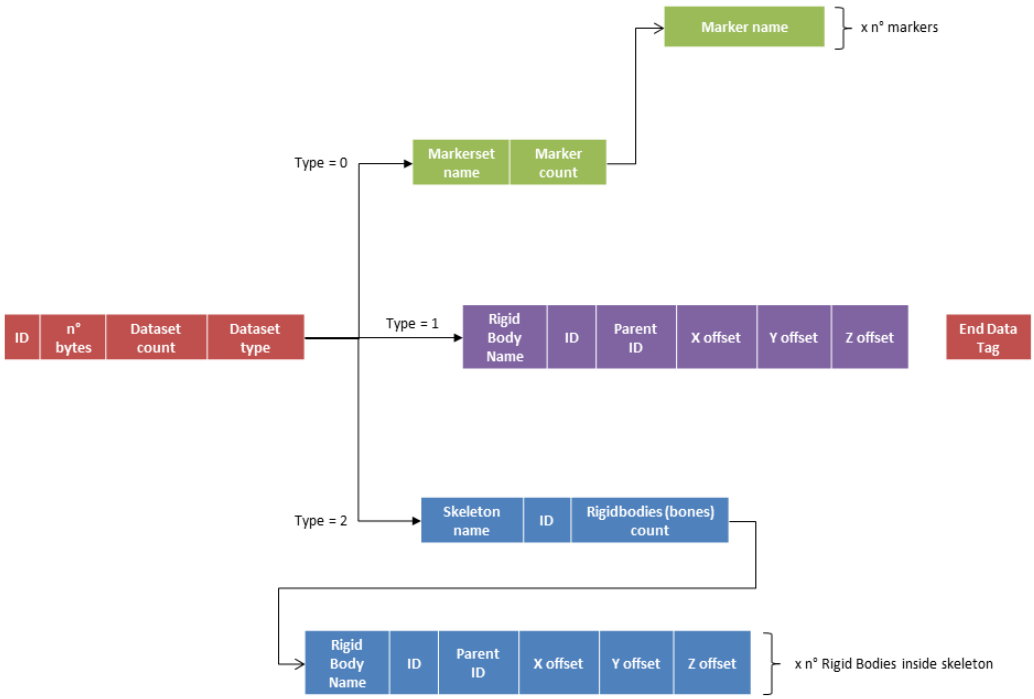
RESPONSE	3
REQUEST_MODEL_DEFINITION	4
MODEL_DEFINITION	5
REQUEST_FRAME_OF_DATA	6
FRAME_OF_DATA	7
MESSAGE_STRING	8
UNRECOGNISED_REQUEST	100
UNDEFINED	999999.9999

The server and the client communicate using these IDs. For example, if the client sends ID=6 the server will answer sending an entire frame of data starting with ID=7.

The principal ID Messages needed to be handled by the client are 5 and 7. The complete structure of those two frames is illustrated in Figure 4.8. When the client requests a data frame, the server fills the entire buffer with the requested information. The client then allocates part of the memory to store this buffer and to start the depacketization, using an offset pointer. This variable goes through the allocated memory, by only updating its value adding the exact dimension of the data just read. Usually, each information occupies 4 bits, otherwise it is longer than the number of bits stored. For example, after reading the ID Message at the beginning of the payload, the pointer is updated adding 2 bits to read the next set of information. Instead, if it is necessary to read the name of a Rigid Body, the pointer is updated by adding the corresponding number of bits as the characters which compose the name (ex. "hand" is 4 bits).



(a) When the MessageID is 7, the buffer is filled with the information regarding the markers and Rigid Bodies coordinates.



(b) When the MessageID is 5, the buffer is filled with the definition of the data which will be transmitted from the server.

Figure 4.8: The UDP payload stores the data sent by the OptiTrack server. Depending on the type of ID Message received, the buffer is filled with different types of information.

4.2.2 Stereoscopic-3D component

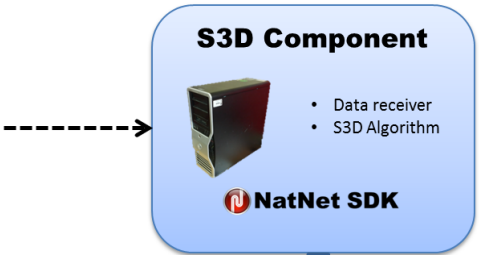


Figure 4.9: S3D component diagram

After studying how a stereoscopic image is generated and visualized, the following step was to design a system inside Unity3D able to generate a stereoscopy output, and then subsequently to visualise it through the utilisation of either a passive or an active system. Unity3D implements the possibility to use the integrated Stereo-3D render[3], converting any mono image into a stereo one. However, this solution does not give a wide range of customisation for the user. Using Nvidia3D Vision, the only parameter which is easily modifiable is depth perception. At the same time, the documentation on this render engine is not exhaustive, and fails to give a complete explanation of the entire rendering process and how the stereoscopic image is generated.

In order to obtain a virtual object which is superimposed on a real one, it is necessary to have a depth perception which allows the generated stereoscopic object to reach a considerable distance from the screen. If this is too elevated it may provoke eyesight distress or sickness. For this reason, the utilization of NVidia3D was been abandoned in the first moment. This decision was followed by a strategy to develop a way to visualize a video game developed in Unity3D in Stereo-3D from scratch.

The first and easiest way consists of realising a system of two cameras in Unity3D, and to visualise them in a **Side-by-Side** (SBS) configuration. The monitor software then merges the two images in one to obtain a stereoscopic image. The two cameras are placed with the axes parallel between them, in order to have the simplest system possible (Figure 4.10).

Furthermore, how the image is rendered has to be to take in account. The integrated stereoscopic render adapts the **aspect ratio** of the final output images. This value represents the proportional relationship between the height and width of a picture, and is directly correlated to the chosen rendering resolution. It is expressed by two numbers separated by a colon (i.e. 4:3). This issue is easily solved by assigning a custom aspect ratio to each camera, in order to obtain a final picture which is rendered correctly.

As mentioned above, a parallel camera axis system is the simplest to realise, rather than a system with converged axes. In this research, both systems have been analysed

[3]NVidia3D Vision

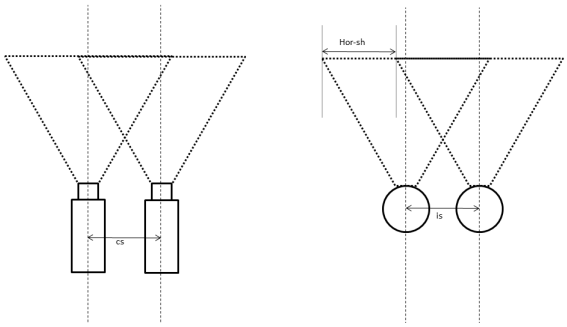


Figure 4.10: Parallel Stereoscopic-3D configuration

in order to find the best configuration possible to develop the User Interface.

Parallel Axis Configuration

The main concern of this system is to create a horizontal shift between the two images (left and right). In this case, the object appears to be placed at an infinite distance and is properly merged at greater distances from the camera. This configuration is the best choice if the operation area (in this case the scene on Unity3D) is extensive, and if the objects to be visualised are placed at greater distances. As shown on Figure 4.10 on page 38, the distance between the cameras is equal to the horizontal shift between the two pictures (parallax). Moreover, if it is equal to the distance of the user's left-right irises, and the angle of view of the camera lens and the display screen are also equal, all the conditions to obtain a distortion-free representation are fulfilled[29]. However, it is almost impossible to obtain a similar system, as the shooting and the visualization conditions often vary.

Convergent Axis Configuration

A crossed axis configuration (or Toed-in) allows to shoot stereoscopic images, changing not only the separation distance between the two cameras, but their axis angle as well. The subject to be visualised in 3D should be captured in correspondence to the convergence point of the two camera axes. By changing this point, it is possible to move it back and forth from the viewer. At the same time, the camera separation may be changed to adjust the stereoscopic effect.

This system allows a greater manipulation of the stereoscopic effect, and for this reason it has been widely used in this work. This configuration has been adapted to be used with the same theory used to design a stereoscopic 3D configuration for robotic telemanipulation.

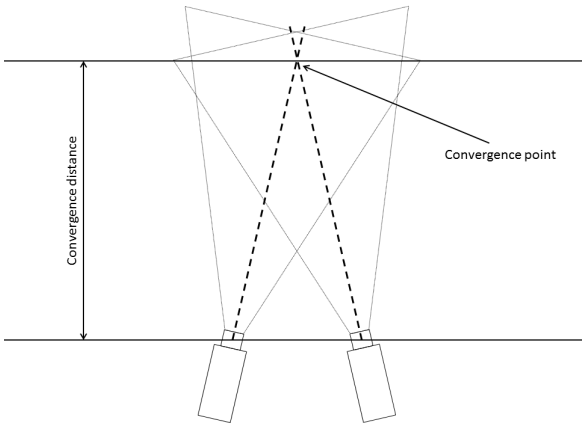


Figure 4.11: Toed-in Stereoscopic-3D configuration

Realising a Toed-in stereoscopic system in Unity3D

As seen in the previous section, a toed-in cameras configuration is the best choice to design a stereoscopic system that is adaptable to every scene size. To build this kind of system, Ferre et al's theory (illustrated in Chapter 3: State of the Art) on designing a stereoscopic system configuration for robotic telemanipulation was used[7].

This configuration is reliable for rendering a stereoscopic subject in order to obtain an Augmented Reality interface, because it reproduces human vision and provides an effective reproduction of the rendered scene, simplifying the subsequent operation of rendering the stereoscopic object at a determined distance from the user.

Before doing so, it is important to consider the virtual cameras used in Unity3D. The system analysed is based on the utilisation of real cameras, which have a specific **Focal Length**. It is possible to define Focal Length as the distance in mm from the optical centre of a lens to a point where a subject at infinity appears in sharp focus. Usually, this is the distance between the lens and the surface of a film or a digital camera's sensor.

For example, a 28mm lens gives a wide-angle view, while a 300mm telephoto lens offers a much narrower angle of view that brings far-away subjects much closer.

Since Unity3D cameras do not have a proper lens system[4], a definition of Focal Length does not exist. On the other hand, the magnification factor of the image is given by the **Field of View**, expressed in degrees along the Y axis.

Instead of completely eliminating the definition of field of view and thus modifying the theory used to develop the toed-in system, it was decided to find a correspondence of this measure inside the Unity3D camera system. Through some experiments, it has been observed that for higher Field of View values the camera performs an equivalent zoom. Moreover, on Unity3D the Field of view decides the dimension of the **Far Clipping Plane**, which is the furthest point that the camera can catch. It also changes the

[4]like most virtual camera systems, they are based on the pin-hole model

dimension of the **Near Clipping Plane** which is, on the other hand, where the camera view begins. These two planes delimit the dimension of the camera's **virtual frustum**, which is the limited area inside which the subject is rendered.

As Focal Length, the horizontal distance between the two Clipping Planes was used, as only the objects placed inside this place are rendered. Using the trigonometric notations and the Figure 4.12 on page 42 as reference, it is possible to calculate the height in mm of each Clipping Plane.

$$\overline{BO}3 = \tan(\beta) * \overline{AO}\,, \quad \text{with} \quad \beta \quad \text{in radians}\,. \tag{4.1}$$

$$\overline{BC} = \overline{BO} * 2 \tag{4.2}$$

The value of $\overline{BC}$ (Equation 4.2) represents the height in mm. To find the width (Equation 4.3) it is necessary to first divide the image plane height by the screen vertical resolution (in pixels), and then multiply the result by the horizontal resolution.

$$width_{mm} = \left(\frac{\overline{BC}}{h_{screen}}\right) * w_{screen} \tag{4.3}$$

$$hFoV_{rad} = \arctan\left(\frac{width_{mm}}{aspectRatio}\right) \tag{4.4}$$

$$dist = \frac{width_{mm}}{\tan(hFoV_{rad})} \tag{4.5}$$

$$dist_{tot} = dist_{far} - dist_{near} \tag{4.6}$$

It is important to keep in mind that this measure is not a real focal length, but the dimension of a virtual space between the two planes of the camera, where the object is always seen in sharp focus.

The Figure 4.11 on page 39 represents the design of a Toed-in system in Unity3D, using the theory illustrated in [7]. It is characterised by the distance between the centres of the cameras (O), the angle between the camera axes (α) and the distance between point I and the intersection of the camera axes (H). All these parameters are related between them by Equation 4.7

$$O = H\sin(\alpha)\,, \quad \text{with} \quad \alpha > 0 \tag{4.7}$$

Defining as d the distance between a point P of the central line and point I, it is possible to calculate the projection disparity (Equation 4.8) between the camera projections of the point located on the central line. This measure is useful for calculating

an acceptable distance between the two cameras, in order to avoid vision issues in the user. This measure must be less than the maximum acceptable disparity, which is calculated keeping into account the user-to-screen distance and the human threshold.

$$projectionDisparity(d) = 2 * focallenght * \tan\left(\frac{\alpha}{2}\right)\left|\frac{H}{d} - 1\right|, \quad \text{with} \quad \alpha > 0 \quad (4.8)$$

Immersive User Interface

We define an Immersive User Interface the typology of human-computer interaction that allows the user to be part of the virtual environment represented through the utilization of stereoscopic-3D visualization. In an immersive environment, the user is able to see the 3D object, the background contents and his own body, and it would be natural for the user to use his/her own body to interact with the objects floating out from the screen. The idea was inspired by the study carried out by Du et al. [5], from which the equations illustrated in this section have been taken.

An Immersive User Interface has to take into account various variables to be considered truly immersive; above all, the 3D floating object must be perceived by the user at an acceptable distance from the screen, which allows the user to operate with it.

The main goal of this research is to develop an Augmented Reality videogame, using the principle of a Stereoscopic-3D interface. For this reason, the real environment around the player and the characteristics of the visualization system (for example the screen size) must be taken into account.

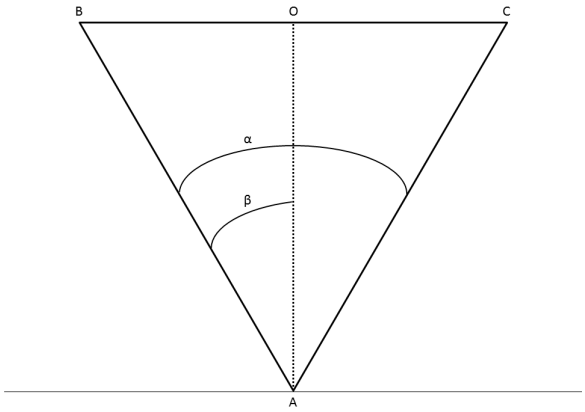
In real world, when we observe a real object, we are able to estimate our distance from it by analysing its dimension. For example, we can understand how far we are from a skyscraper (let's say the Shard in London), depending on whether it occupies just a few centimetres, or all our field of view.

Using the same principle, it is possible to determine the distance between the user and the floating object using this measure to calculate the dimensions of the object. Using the Figure 4.14 on page 44 as reference,with Equation 4.9 it is possible to define the dimensions (Width, and Height) of the virtual object as AB.

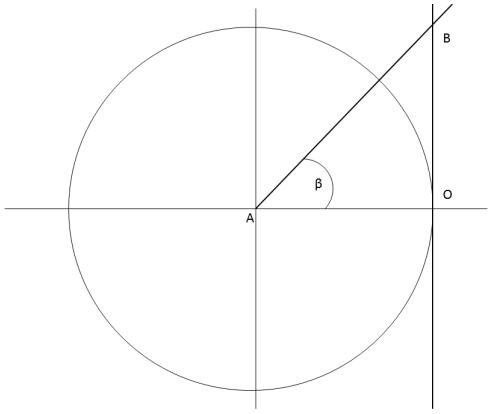
$$\begin{cases} W = \dfrac{z(w+p)}{d} - p \\ H = \dfrac{z * h}{d} \end{cases} \quad (4.9)$$

Where:

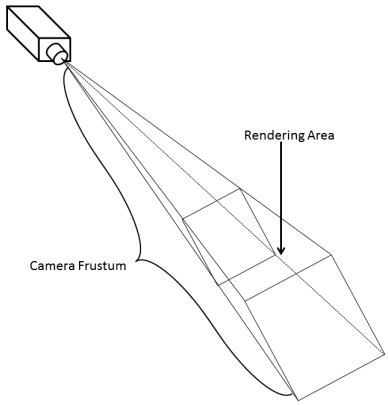
- **z** is the perceived distance to the virtual object
- **w** is the screen width (in cm)



(a) Setting A as the camera position and AO the distance to the near clipping plane. BC also represents the field of view extension with an angle α



(b) BO represents the mid field of view with angle β. Knowing the distance AO it is possible to calculate the field of view extension by trigonometric notation.



(c) Representation of the virtual camera frustum. The zone delimited by the two planes is the rendering area.

Figure 4.12: It is possible to represent the camera frustum as a triangle (a) where α represents the field of view angle. To calculate the dimension of the clipping plane it is necessary to represent it as a goniometric circle (b).

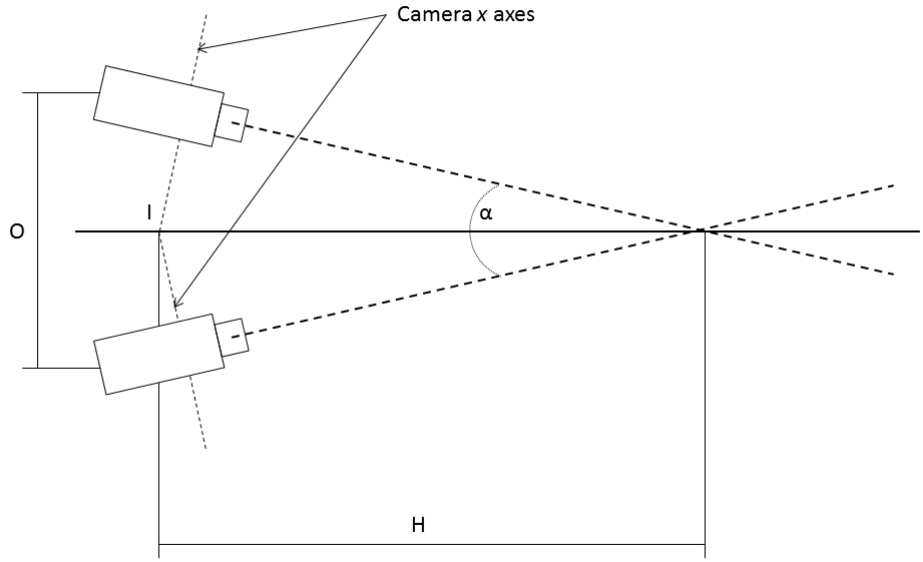


Figure 4.13: Illustration of the Toed-in system designed on Unity3D. The figure is a simplified version of the one seen in Chapter 3.

- **p** is the inter-pupillar distance ($p = 0.065m$)
- **d** is the user-to-screen distance
- **h** is the screen height (in cm)

It is possible to use the screen height dimension and the vertical resolution to calculate the best user-to-screen distance (Equation 4.10).

$$d_{best} = \frac{\frac{h}{v}}{2\tan\left(\frac{1}{120}\right)} \approx 3400\frac{h}{v} \tag{4.10}$$

Using these results in addition to those previously formulated to realise the Toed-in configuration, it is possible to obtain a stereoscopic system which allows the user to visualise an object floating out of the screen at a certain distance. The above formula must be completed using the user's and real environment information captured by the Motion Capture system, which has been illustrated in the previous section.

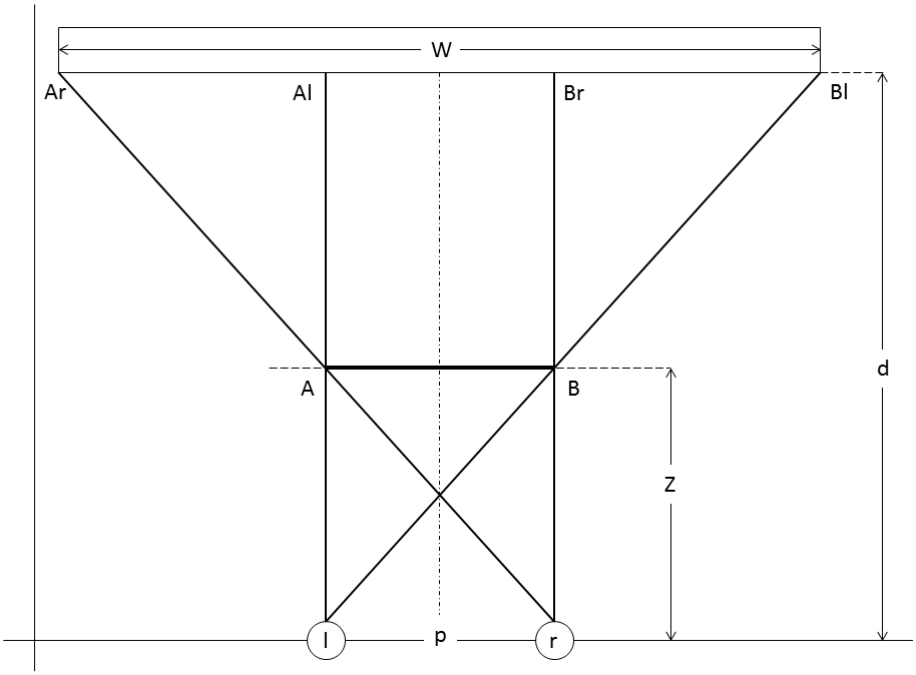


Figure 4.14: Calculating the dimension AB of the floating object, using the screen size (W), the user-to-screen distance (d) and the user-to-object distance (z). p represents the interaxial eye separation.

4.2.3 Integrated System

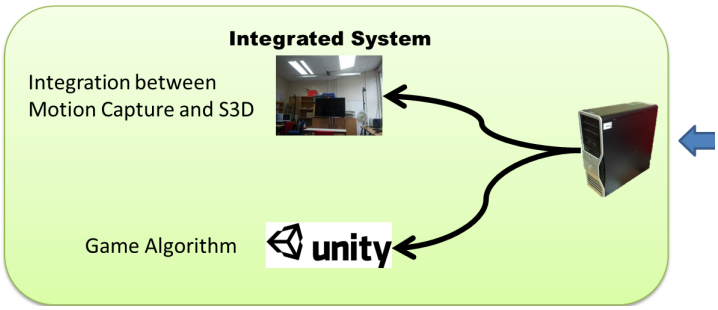


Figure 4.15: Integrated System diagram

An integrated system is defined as the union of the two previously illustrated technologies, in order to obtain a smart interface that is able to adapt itself to the player's requirements.

The main goal of this work is to develop an Augmented Reality video game that must be adaptable and fully-immersive.

The Stereo-3D system must obtain from the OptiTrack the information regarding the player's position and the real environment around him/her. In particular, it is necessary to settle a new system of coordinates where the monitor is the origin. Once all this information is available, all the calculations needed for the video game interface are performed by the Stereo-3D algorithm.

The last obstacle is how the information regarding the position of the markers is transmitted to the game engine. The coordinate system in Unity3D is right-handed, while the OptiTrack is left-handed. Moreover, the information sent by the Motion Capture system needs to be converted from quaternion to Euler angles, in order to detect the exact orientation of the tracked object.

Quaternion to Euler Angles conversion

Quaternion are commonly used to calculate rotations in a 3-Dimensional space. In particular, they are stored inside the OptiTrack frame with the notation: (qx, qy, qz, qw)[5].

The process followed to perform the conversion takes some conventions into account:

- input and output units are both in radians
- Euler order is YZX
- Euler angles are about global axes
- The results are right-handed
- Y = Heading(YAW)
- Z = Attitude(Pitch)
- X = Bank(Roll)

The first step consists of testing the singularity at the north and south pole, by calculating a variable *test* using Equation 4.11. If this is higher than +90°there is a singularity at the north pole (Equation 4.12). If it lower than -90°, it is at the south pole (Equation 4.13).

$$test = qx * qy + qz * qw \tag{4.11}$$

$$\textbf{North Pole} \begin{cases} Y = 2 * 2 \arctan \dfrac{\sqrt{qx^2 + qw^2} - qx}{qw} \\ Z = \dfrac{\pi}{2} \\ X = 0 \end{cases} \tag{4.12}$$

$$\textbf{South Pole} \begin{cases} Y = -2 * 2 \arctan \dfrac{\sqrt{qx^2 + qw^2} - qx}{qw} \\ Z = -\dfrac{\pi}{2} \\ X = 0 \end{cases} \tag{4.13}$$

[5]The complete explanation of the Quaternion to Euler Angle conversion, and the reason for which it is preferable to use a 4D system to describe the rotation of a 3D object, can be found at the following website[2]

in the other cases (-90≤$test$≤90) the conversion is as such:

$$a = 2 * qy * qw - 2 * qx * qz \tag{4.14}$$
$$b = 1 - 2 * qy^2 - 2 * qz^2 \tag{4.15}$$
$$c = 2 * qx * qw - 2 * qy * qz \tag{4.16}$$
$$d = 1 - 2 * qx^2 - 2 * qz^2 \tag{4.17}$$

$$\begin{cases} Y = 2 \arctan \dfrac{\sqrt{a^2 + b^2} - a}{b} \\ Z = \arcsin(2 * test) \\ X = 2 \arctan \dfrac{\sqrt{c^2 + d^2} - c}{d} \end{cases} \tag{4.18}$$

Once the triplet (X,Y,Z) is obtained using Equation 4.18, it is possible to perform a conversion from radians to degree, and to apply the result to the virtual object on Unity3D. This operation is only used to deduct the rotational angles of the tracked object, to be applied onto the virtual one. The conversion itself changes the reference system from left-handed to right-handed, so in this case it is not necessary to perform further operations.

Right or Left Hand?

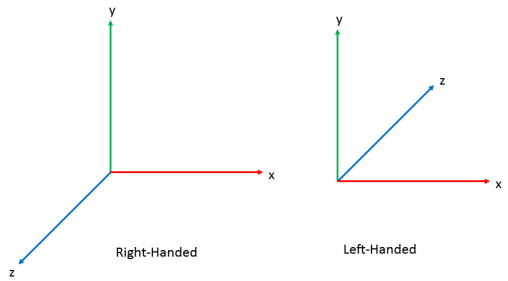


Figure 4.16: The substantial difference between the two systems is the version direction sign of the z axis.

If the issues regarding the rotation angles are settled, the difference between the coordinates system must still be faced for the translation of the point inside the space.

As mentioned before, the OptiTrack is a Left-Handed system, while Unity3D is Right-Handed. The difference is illustrated in Figure 4.16, where the sign of the vector

alongside the z axis must be changed if the translation is converted from the first to the second system, and viceversa.

In this research the Right-Handed convention was adopted to describe the movement along the axes. This reference system is used by the game engine and it is, therefore, more simple to convert the triplet from the Motion Capture once they are integrated inside the game engine.

Chapter 5

Implementation

5.1 Stereoscopic-3D Visualization Component

5.1.1 Preliminary Idea: Realising a Stereo Camera system in Unity3D

To understand how the configuration was implemented, it is necessary to briefly explain how the camera system on Unity3D works. The GameObject "camera" in Unity3D renders inside a rectangle (knows as **Viewport rect**), which covers the entire screen area (Figure 5.1). The viewport rect is characterised by four values that indicate the coordinates (0-1) where the camera should be visualised on the screen.

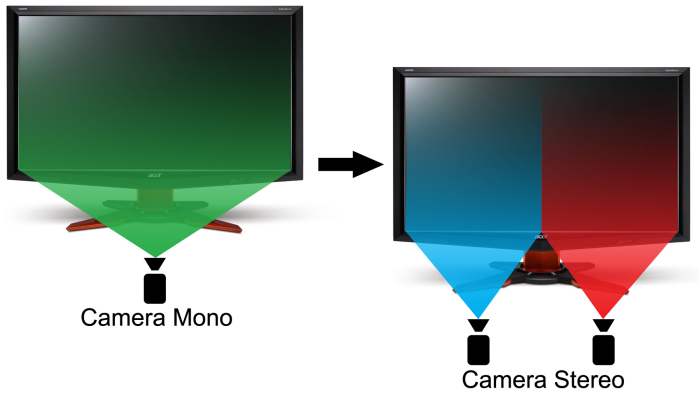


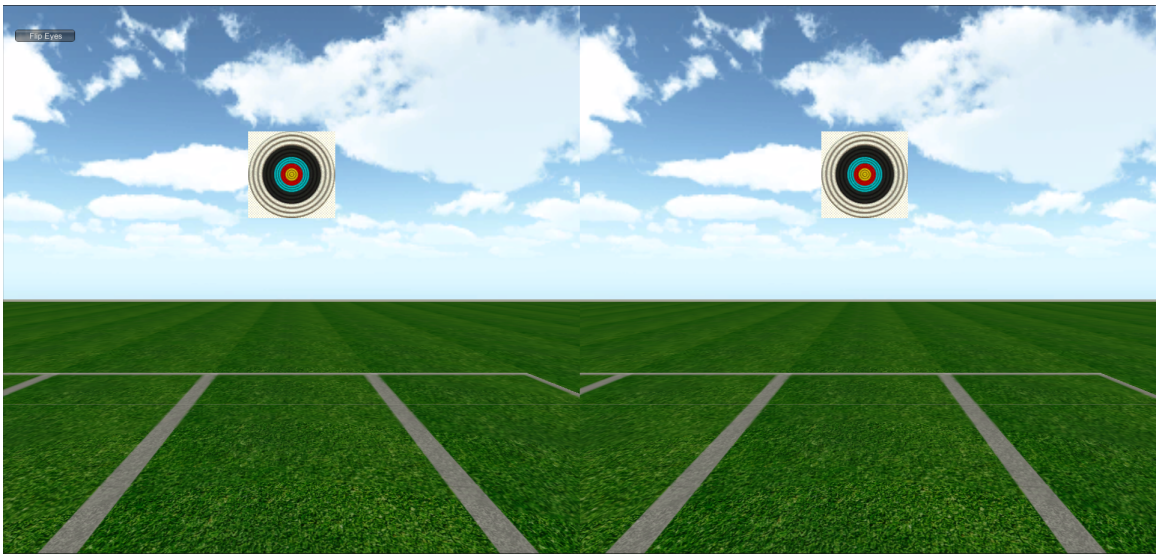
Figure 5.1: Stereo Camera in Unity3D

- **X:** The initial horizontal position on which the camera view will be drawn.
- **Y:** The initial vertical position on which the camera view will be drawn.
- **W(Width):** Width of the camera output on the screen.
- **H(Height):** Height of the camera output on the screen.

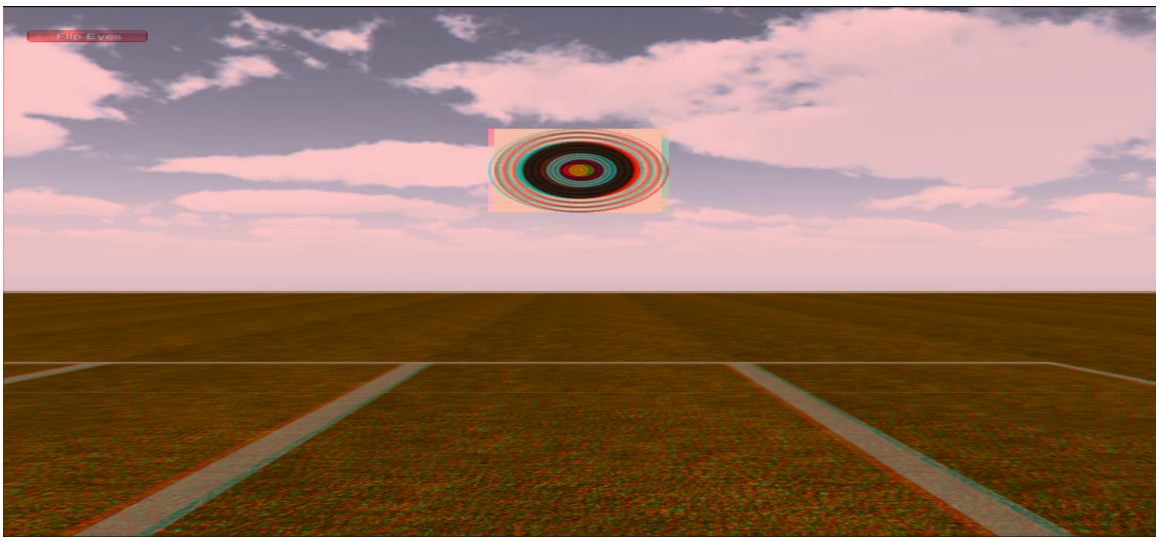
By changing the above values it is possible to render more than one camera on the screen, establishing the exact starting point and size of each camera. The following settings produce the result illustrated in Figure 5.2.

	Left Camera	**Right Camera**
X	0	0.5
Y	0	0
W	0.5	0.5
H	1	1

Table 5.1: ViewPort settings for left and right camera



(a) Side-by-Side image generated by Unity3D



(b) Final Output generated by the TV 3D software

Figure 5.2: The two cameras produce an unique view (a). The TV 3D Software then produces the stereoscopic image (b)

```
public class cameraAspect : MonoBehaviour {
        float AspRat = 0.0f;
        // Use this for initialization
        void Update () {
                //Calculate the aspect ratio of the monitor in use
                AspRat = (float)Screen.currentResolution.width /
                (float)Screen.currentResolution.height;
                //assign it to the linked camera
                this.gameObject.camera.aspect = AspRat;
        }
}
```

Code 5.1: CameraAspect.cs

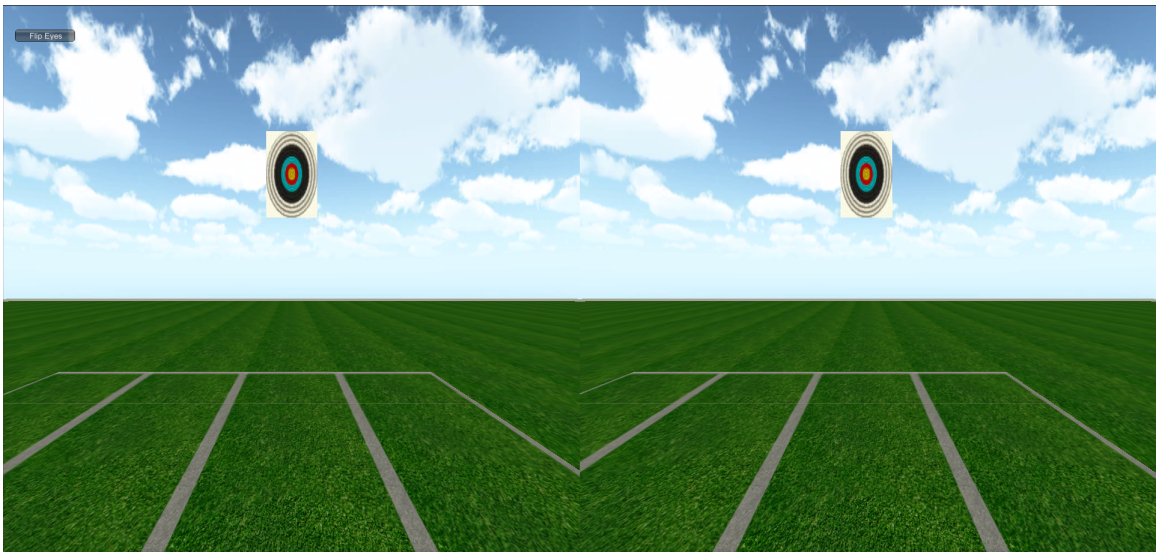
What is observed in Figure 5.2(b) is a stretching of the final merged image, caused by the **aspect ratio** of the two previously generated ViewPort Rects. The chosen resolution for this project is 1920x1080, and the monitor used has an aspect ratio of 16:9[1]. Consequently, the two camera ViewPort Rects should render using an Aspect Ratio that, summed together, will result in 16:9. When the two views are merged by the TV software, the pixels are stretched in order to cover a wider area.

To avoid this phenomenon, the aspect ratio of each camera is forced to 16:9. This is done by calculating the aspect ratio of the monitor in use, and then initialising each camera with this value.

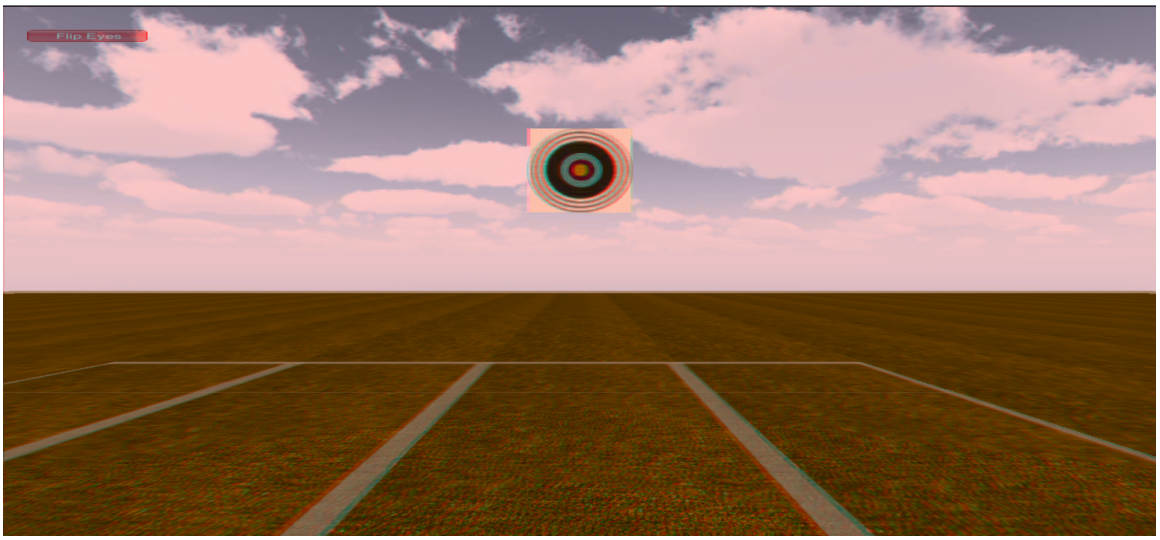
In Unity3D, it is not possible to assign a custom Aspect Ratio to the camera using the common options provided by the Inspector, but through a specific script linked to the camera, (Code 5.1) which calculates and assigns the value.

The result of the above operation is illustrated in Figure 5.3. The images represented on the two camera ViewPort Rects appear stretched, and this is caused by the fact that the two images now are rendered with a 16:9 aspect ratio (double than before). The deformation is caused by the visualisation area which is also 16:9, so the images are stretchered horizontally. The final picture (b) is no longer deformed, given the result ratio is the same as that of the visualization area.

[1]It is the international standard format for HDTV



(a) The two images are horizontally stretched, to fit the visualization area



(b) After the merging, the deformation is gone

Figure 5.3: How a stereoscopic image is visualised in Unity3D

5.1.2 Implementing a Toed-in Stereoscopic-3D System

The next step is to realise a convergent axis system on Unity3D. As explained in Chapter 4, a toed-in configuration allows to move the stereoscopic object back and forth, by only changing the convergence point between the two camera axes.

Starting from the results explained in the previous section, it is necessary to create a script which allows the change of the angle between the two cameras, in a smart way, rather than setting a unique angle. This is the best solution for implementing a system that can be adapted to user requirements, as eyesight varies from person to person.

Code 5.2 illustrates the implementation of the Toed-in system. The script calculates the angle between the cameras using the triangle theorems. Knowing the camera-to-camera and the object-to-camera distances, it is possible to calculate an angle β. Then

```
        // Update is called once per frame
        void Update () {
    //To calculate the angle between the two camera the process is:
    //1- Calculate the middle distance between the two cameras
        cameraDist = (Mathf.Abs(this.transform.position.x - cameraR.transform.position.x))/2;
    //2- Calculate the distance between the object and the cameras
        objectDist = Mathf.Abs(target.transform.position.z - source.transform.position.z);
    //3- Imagine having a triangle between the two cameras. Using the two lengths we have:
         alpha=arctang(c/b)
        angleAlpha = Mathf.Atan(cameraDist/objectDist);
    //4- then beta= pi/2 - alpha
        angleBeta = Mathf.PI/2 - angleAlpha;
        angleBeta = angleBeta * Mathf.Rad2Deg;
    //now we need to assign the value to each camera, first subtracting 90 degree from the previous
         result
    //because we need to know by how much to rotate the camera inwards to obtain this angle
        this.transform.Rotate(Vector3.up, (angleBetha - 90)-this.transform.rotation.y, Space.Self);
        cameraR.transform.Rotate(Vector3.up, -((angleBeta - 90)-cameraR.transform.rotation.y),
            Space.Self);
        }
}
```

Code 5.2: ToedInCamera.cs

subtracting this angle from 90, it is possible to calculate the actual axis angle of each camera.

Up to this point we have a system that is able to adapt itself in order to maintain the virtual object on the point of intersection of the two axes.

The following table lists various angles α calculated at a certain distance between the camera and the object. Applying the formula seen in Chapter 4, it is possible to calculate the relative Projection Disparity of each angle, and to formulate an evaluation of the system.

object dist	angle α
0.68	11.93
1.1	6.63
2.1	3.48
3.1	2.36
4.1	1.78
5.1	1.43
6.1	1.20
7.1	1.03
8.1	0.90
9.1	0.80
12.1	0.60
15.1	0.48

The result is illustrated in Figure 5.4. It is possible to notice that using a interaxial distance between the two cameras of 0.25, objects placed further have a disparity value which is acceptable, while objects too close to the camera cannot be correctly merged.

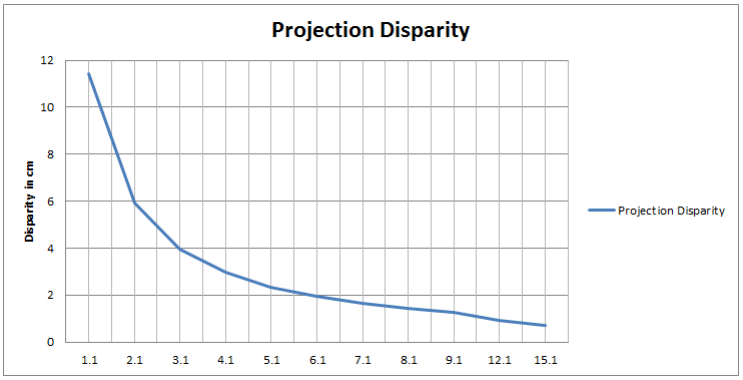


Figure 5.4: The calculation of the Projection Disparity allows to understand what is the range in which the stereoscopic-3D object can be correctly seen by the user.

It is possible to perform the same calculation using a settled distance between the cameras and the object, and changing the interaxial distance. However, the objects near the camera will still not be merged correctly.

To overcome this issue, it is necessary to utilise a further theory that allows the visualisation of a stereoscopic object by changing its dimension depending on the object-to-camera distance.

5.1.3 Immersive User Interface

The previous section explained how stereoscopic-3D visualization is realised in Unity3D. This methodology is enough to visualize objects which are placed at a medium distance from the camera. This research aims to visualize a virtual arrow superimposed on a real bow. For that reason the arrow is virtually placed as close as possible to the cameras, and the 3D effect may be annoying or generate discomfort in the user's eyesight.

The theory analysed in Chapter 3 regarding the realisation of an Immersive User Interface for 3DTV illustrates the possibility of developing an UI by changing the dimension of the visualised object in relation to the theoretical distance from the user.

This theory can be implemented only by using some additional information from the motion capture system, as the user-to-screen and the user-to-object distance must be recovered from the real position data. Otherwise, it is possible to deduct this information or to insert it manually inside the system. However, the principal aim of this work is to obtain a smart interface which is able to adapt itself.

Given the formula explained in Chapter 4

$$\begin{cases} W = \dfrac{z(w+p)}{d} - p \\ H = \dfrac{z * h}{d} \end{cases} \tag{5.1}$$

While the variable p is fixed at 0.065 m, the other values need to be calculated in relation to the typology of screen used, in particular the monitor resolution and actual dimension in cm. The first step consists of calculating this information, knowing that the monitor resolution is 1920x1080, the aspect ratio is 16:9 and the diagonal is 24 inches[2].

$$\begin{aligned} &Base(w) = 16x \\ &Height(h) = 9x \\ &Diagonal = \sqrt{Base^2 + Height^2} \approx 18.37 \\ &w = \left(\frac{24}{18.37}\right) * 16 = 20.90inch \Rightarrow 53.086cm \\ &h = \left(\frac{24}{18.37}\right) * 9 = 11.70inch \Rightarrow 26.718cm \end{aligned}$$

The value of z and d are calculated using the measures calculated by the OptiTrack

[2]These calculations can be transposed to be used in other typology of visualization screens, including video projectors

```
private float calculateRatio() {
aspectRatio = widthRes / heigthRes;

return (diagonalInch / Mathf.Sqrt (Mathf.Pow (aspectRatio, 2) + 1));
}
public void calculateMonitorDim() {
heigthCm = calculateRatio () * 2.54f;
widthCm = aspectRatio * calculateRatio () * 2.54f;
}
```

Code 5.3: Calculation of the screen actual dimension in cm

```
public float[] calcObjectDim() {

float[] dim = new float[2];

dim[0] = (z * (scr.getWidth() + p)) / d - p;
dim[1] = (z * scr.getheigth ()) / d;

return dim;
}
```

Code 5.4: Snippet for the calculation of the object maximum dimensiona at a certain position z.

system between the user head, the crossbow and the monitor (all equipped with some markers, as will be explained later on the chapter). As an example, we are going to establish that those distances are z=50 cm and d=84cm[3].

Substituting all the values on the Equation 5.1, we obtain

$$W = 28.96cm$$
$$H = 15.90cm$$

which represent the maximum dimension of the virtual object to be seen at the distance of 50cm from the user. These measurements allow us to find a scaling variable to be applied directly on the Unity3D virtual object. This allows us to visualise a very close stereoscopic-3D object, without moving the convergence centre of the two cameras. In relation to its distance from the two cameras, the object will be consequently scaled.

[3]using the best distance formula illustrated on the previous chapter

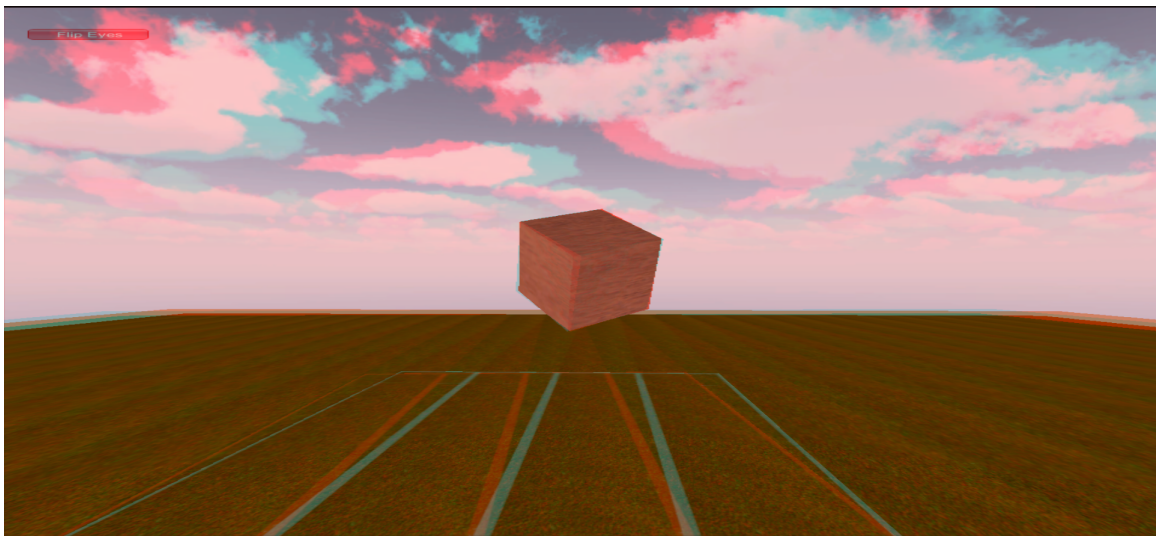


Figure 5.5: Formula Application at a distance of 5 cm from the object

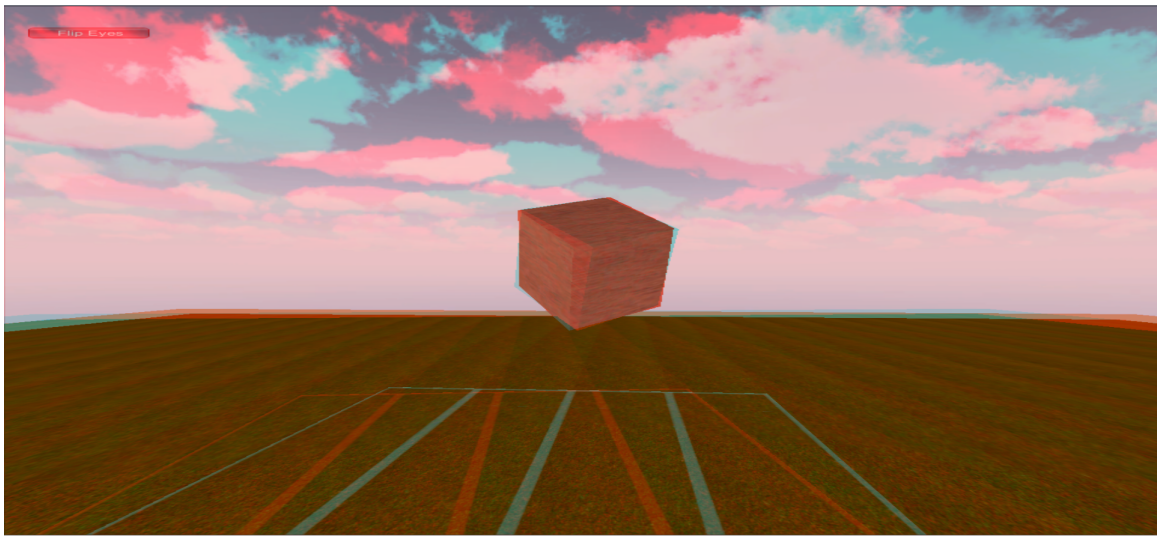


Figure 5.6: Formula Application at a distance of 2 cm from the object

5.2 Motion Capture Component

5.2.1 Tracking Tools and Unity3D communication

The communication between the OptiTrack and an external application happens normally through the utilisation of a library (NatNet SKD) provided by the OptiTrack producer. At present, Unity3D is not among the officially supported applications. This means that it is necessary to develop a C-sharp script in order to send and receive packets from the server located over the network.

In Chapter 4 the structure of the communication and the packets exchanged between the client and the server were analysed in detail.

The server was already implemented by the company that produced the OptiTrack system. It is an application called TrackingTools that allows to stream the captured data over the network, using a multicast or a local address. This application is structured in order to continuously stream the data, once the relative option is activated. A handshaking procedure is necessary to synchronize the client to the streaming channel, otherwise the data received will not be time synchronised with the server, causing lags and glitches.

The *handshake* between the client and the server includes three different stages (Figure 5.7): the first is related to the tracked object which opens the socket communication with the server, then it moves to the second stage where the OptiTrack Client sends a request of connection to the server. Finally, the server answers by sending back the first frame. After this handshaking, where the connection is initialised, there is no necessity for the client to send other requests as the server will keep sending data until the application is closed by the user.

Figure 5.8 represents the activity diagram between the scripts, which compose the server. Inside the Unity3D scene it is possible to have more than one object tracked at one time, and for this reason it is necessary to have a unique script which receives the data, and more than one script that use these data to perform the multiple translations. It is impossible to implement a unique script assigned to every scene object, which include the communication part as well, otherwise we will have the same number of open sockets as there are active objects in the scene.

The following two sections provide an analysis of the structure of the scripts that compose the communication platform: **OptiTrackUDPClient** and **TrackedObjectScript**.

5.2.2 The OptiTrackUDPClient class

This is the class demanded to connect directly with the Tracking Tools server. It has been developed following the guidelines provided by the NatNet SDK documentation.

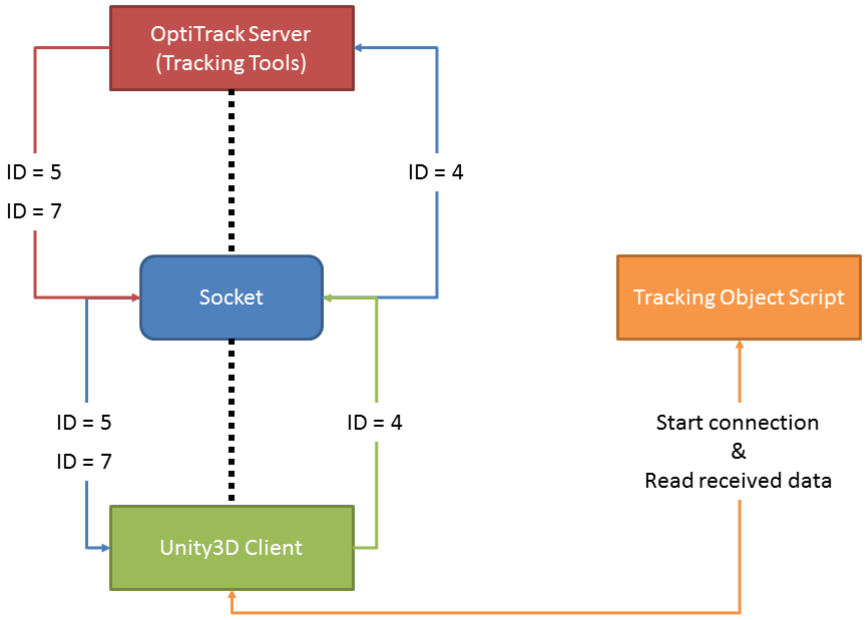


Figure 5.7: Communication scheme between the tracked object and the server (Tracking Tools). The Unity3D Client receives and processes the data from the server, then the Tracking Object Script reads this information directly from it.

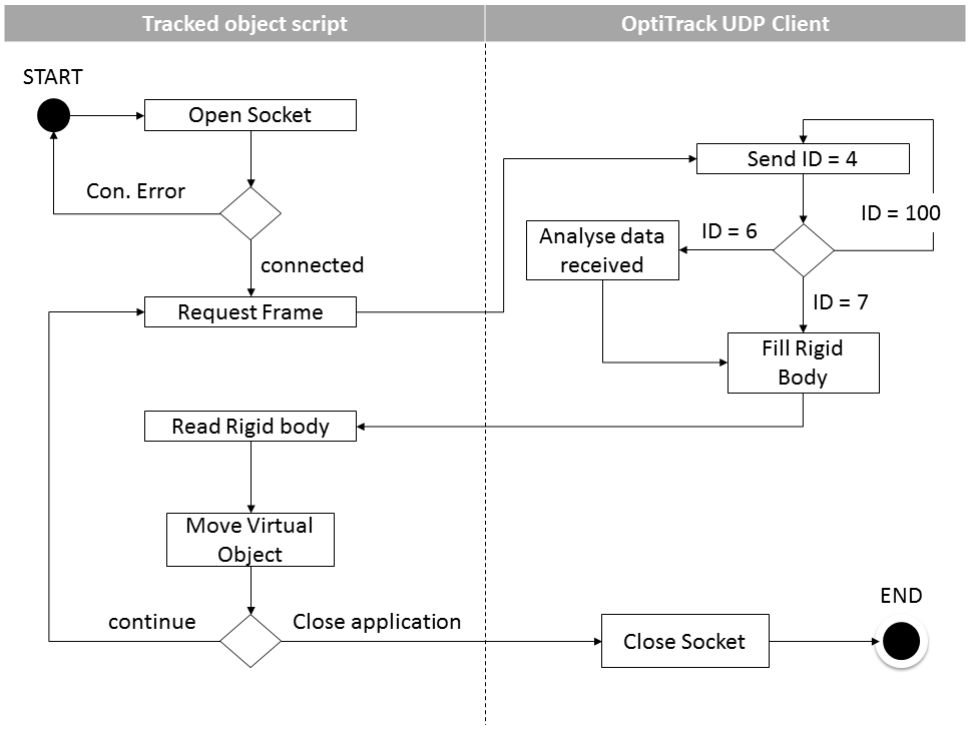


Figure 5.8: The Client side is composed by two or more different scripts. The tracked object script, which is assigned directly on the virtual object on the Unity3D scene, and the OptiTrack UDP client, which communicates directly with the Tracking Tools server.

The complete structure of the class is illustrated in Figure 5.9.

The first four variables **dataPort**, **commandPort**, **multicastIPaddress** and **localIPaddress** are allocated with the connection information.

- dataPort: 1511
- commandPort: 1510
- multicastIPaddress: 239.255.42.99
- localIPaddress: 147.197.232.84

Through the dataPort the data are received from the server, while the client sends the request ID from the command port.

The socket connection through the multicast or the local address is initialised by the **Connect()** method. The same method starts the asynchronous connection between the two parts, by initialising the method **AsyncReceiveCallBack**. In this way, the OptiTrack UDP Client will always receive the packet from the server without any request from the Tracked Object Script.

RequestDataDescription and **RequestFrameofData** are two methods that use the command port to send an ID value to the server in order to start the handshaking. These methods are not directly utilised by OptiTrack UDP Client, but by the Tracked Object Script. In this way, it is possible to stop and start again the reception of the data, without closing the socket connection.

ReadPacket first and then **ReadRB** are the methods dedicated to retrieve the Rigid Body information from every received packet. ReadPacket goes through the memory buffer, where the packet is stored, using a pointer that is incremented after reading each portion of information. Once the pointer reaches the memory allocation where the Rigid Bodies information is stored, it calls the ReadRB method by forwarding the memory address where this information is stored. The Rigid Bodies are a complex set of information, composed by three or more markers, which need an adequate method to be read correctly. Once this information is retrieved, it is stored inside an apposite RigidBody object, which is accessible from other scripts (this is a characteristic that will be analysed further in the next section).

QuaternionToEuler, **RadiansToDegrees** and **QuaternionNormalise** are the methods related to the conversion of the rotational angles from quaternion to euler. These methods have been developed using [2] as a reference.

Lastly, once the application is closed by the user, the socket connection is interrupted by the **CloseConnection** method. Without this, the connection between the client and the server will never be interrupted unless a time-out occurs.

Inside the class, a number of RigidBody data structures must also be initialised. These must be as many as the virtual objects that must be tracked inside the Unity3D

scene. These data structures are necessary to store the information of each set of markers tracked in the real scene. For this reason, it is not possible to implement a unique OptiTrack UDP solution, but should be adapted to the number of set of makers to take into consideration.

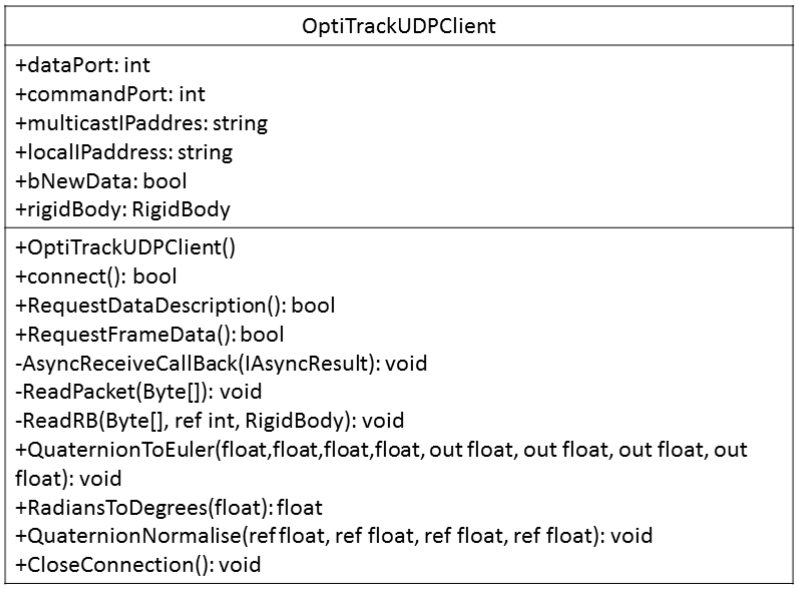


Figure 5.9: UML diagram of the OptiTrackUDPClass.

5.2.3 Tracking Object Script

The last component of the communication platform is the script assigned to the Unity3D scene object to move using the OptiTrack movements.

Using the OptiTrack it is possible to track one or more sets of markers, which are called Rigid Bodies. These data structures are characterised by a set of variables that describe the orientation and the spatial position of the real object, and through them it is possible to move a virtual object inside the Unity3D game scene.

Figure 5.10 represents the C-sharp structure of a RigidBody object. It is characterised principally by the translation coordinates and the rotation angle (yaw, pitch, roll), used to move the virtual object inside the game scene.

The data received from the server are processed and then stored inside this object by the OptiTrackUDPClient. The TrackedObjectScript simply initialises a reference to the client, and then uses those data directly. To simplify the concept lets make an example.

Inside the OptiTrackUDPClient class there is a RigidBody called hand, which is linked to the set of markers placed on the hand of the user.

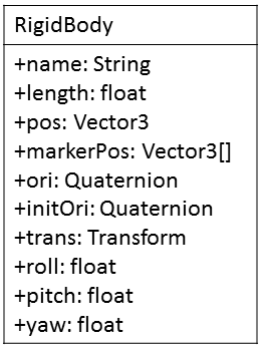


Figure 5.10: UML structure of a RigidBody Class

```
RigidBody head = new RigidBody();
```

Inside the TrackedObject class we have a reference of the UDP client class, that we use to open the connection and then receive the data.

```
public class TrackedObject {

        OptiTrackUDPClient udpClient = new OptiTrackUDPClient();

        public void startConnection() {
                if(udpClient.Connect())
                  udpClient.RequestDataDescription();
        }
        public void takePos() {
                float x = udpClient.hand.pos.x;
                float y = udpClient.hand.pos.y;
                float z = udpClient.hand.pos.z;
        }
}
```

Code 5.5: The code take the position of a generic tracked object inside the OptiTrack space.

A general TrackedObject script is structured as shown in the above code. The data are taken directly from the UDP Client using the reference udpClient, and the position information is stored inside some variables directly accessing the RigidBody *hand* created in the OptiTrackUDPClient class.

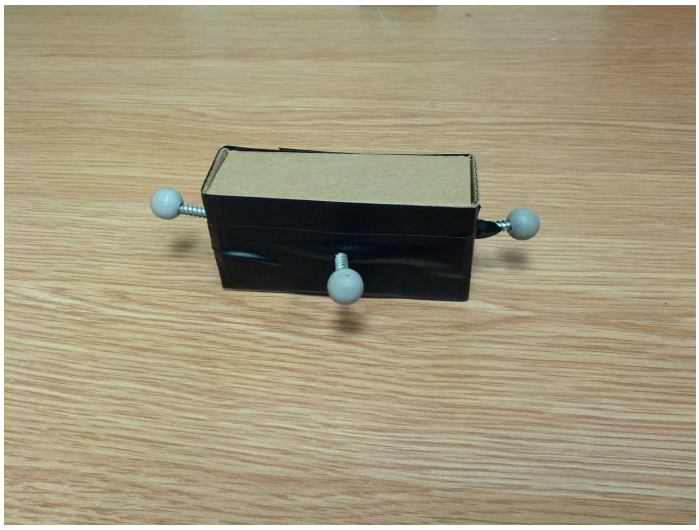
5.3 Implementing The ArcheryGame2.0 Game

5.3.1 Integration between the modules

The S3D algorithm needs the user-to-screen and the user-to-crossbow distance to generate the stereoscopic output.

Both the screen and the user are equipped with reflective markers that identify their position inside the real space.

Figure 5.11 shows the solutions to track both the user and the screen. The user must wear a pair of glasses on which three markers are placed in line, while a cube with three markers is placed on the top or on the bottom of the screen.



(a) Monitor tracker set, to be placed at the bottom or at the top of the screen. The markers are placed to form a triangle.

(b) Head tracker set, to be worn by the user. The markers are placed in line.

Figure 5.11: The two tracker sets have been built from scratch. In particular (b) were a pair of sunglasses from which the lenses were removed, and the markers installed with some screws.

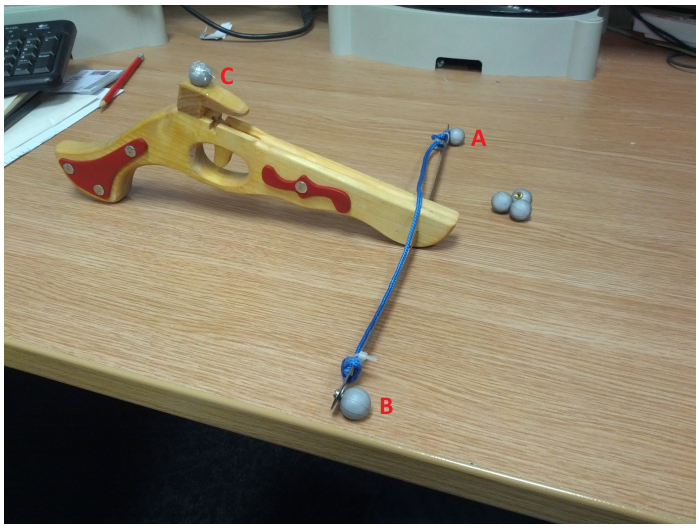


Figure 5.12: The markers on the crossbow are placed to prevent the OptiTrack from losing the object while the player is moving. Moreover, the variation of the distance between A and B is used as a trigger event to shoot the virtual arrow.

The crossbow is also identified by a set of three markers (Figure 5.12), but they are placed in a particular way (we will explain further in the next section).

In Euclidean three-space, the distance between two points A and B is

$$\overline{AB} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2 + (z_A - z_B)^2} \tag{5.2}$$

The following script uses the above formulae to calculate the user-to-screen distance. The same procedure is applicable to determine the crossbow-to-player distance.

```
public class distCal {
        public GameObject screen;
        public GameObject user;

        public float getDistCm() {
                float distx = Mathf.Abs(screen.transform.position.x - user.transform.position.x);
                float disty = Mathf.Abs(screen.transform.position.y - user.transform.position.y);
                float distz = Mathf.Abs(screen.transform.position.z - user.transform.position.z);

                return Mathf.Sqrt(distx+disty+distz);
        }
}
```

Code 5.6: Snippet for the calculation of the distance between the user and the screen. The code can be modified to be generic.

Once these distances are calculated they can be considered in Code 5.4, to calculate the stereoscopic 3D effect.

The virtual objects that compose the game environment inside Unity3D, and are linked with a real tracked object, must be placed at the origin of the coordinates system of the game engine. This allows the virtual objects to move according to the OptiTrack-given coordinates. An example: if the crossbow has real coordinates in the OptiTrack coordinates (12, 34, -98), the respective virtual object inside Unity3D will have the same coordinates. In this way ,it is possible to reproduce exactly the spacial

positions of the real objects inside Unity3D.

```
public class TrackedObject : MonoBehaviour {
        float minimum = -1000f;
        float maximum = 1000f;
        float sensitivity = 5.0f;
        float posX = 0f;
        float posY = 0f;
        float posZ = 0f;

        public void moveObject() {
                float trX = Mathf.Clamp ((x - posX) * sensitivity, minimum, maximum);
                float trY = Mathf.Clamp ((y - posY) * sensitivity, minimum, maximum);
                float trZ = Mathf.Clamp ((z - posZ) * sensitivity, minimum, maximum);

                this.transform.Translate (trX, trY, -trZ);

                posX = x;
                posY = y;
                posZ = z;
        }
}
```

Code 5.7: Completation of the Code 5.5.

Code 5.5, shown in the previous section, can be completed with Code 5.7. Once the position is received from the OptiTrack, it can be assigned to the generic object by multiplying each coordinate value by a **sensitivity** variable. Depending on his/her own preferences, the user can decide which value to assign to sensitivity.

5.3.2 Building the game components

As mentioned above, every tracked object is identified inside the OptiTrack space by a Rigid Body, which is composed by a set of three markers each.

The markers are applied to each tracked object in order to compose a geometrical figure.

The **glasses** that must to be worn by the player are equipped with three markers placed in line between them, as shown in Figure 5.11b. The **crossbow** and the **monitor** marker sets are instead placed in order to form two triangles, with different sizes between them (Figure 5.11a and 5.12). The OpiTrack software streams the centroid coordinates of each rigid body on the network.

Figure 5.13 shows in detail the position of the markers of each game component. The OptiTrack is able to distinguish each trackable object by its different shape and size.

5.3.3 Implementing the Game Mechanics

For every game frame, a runtime script analyses the distance between markers A and B on the crossbow (see the Figure 5.12 as reference). As the distance between the two markers decreases, the script calculates the force that will be used to shoot the arrow. Once the player pulls the crossbow trigger and the distance between the markers increases again, the arrow is shot. The next arrow is recharged automatically after firing.

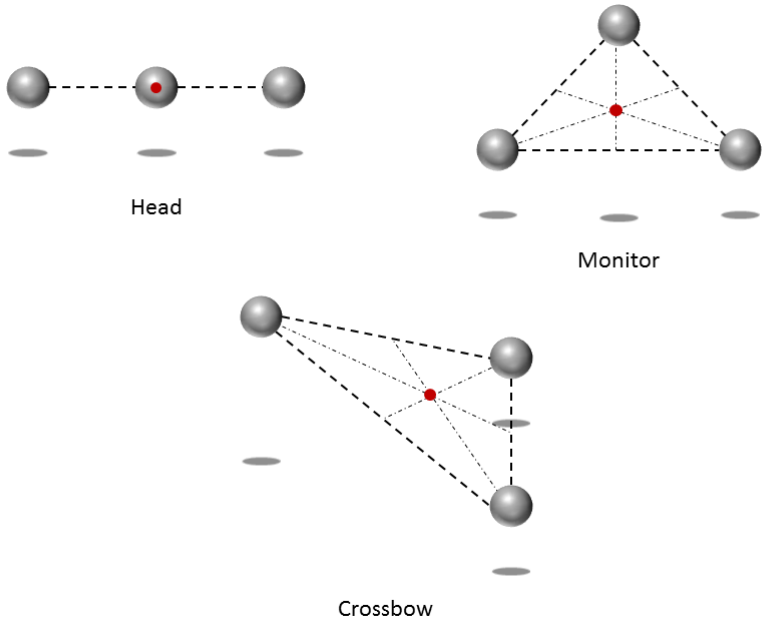


Figure 5.13: Every tracked object is equipped with reflective markers, placed in order to compose different geometrical figures. The OptiTrack system transmits the information relative to the marker sets centroid (red point).

5.4 Testing System Functionality

The aim of each test is to verify:

- the reliability of the S3D algorithm, using monitors of opposite sizes.
- the capability of the Motion Capture system to track the user position.
- the integration between the two above mentioned modules is working.

The expected result is that the interface adapts intelligently to each different context, without the intervention of the user, but using the data provided by the screen in use and by the motion capture.

Systems Configuration

Each module component is listed in the following table:

S3D	
PC:	Intel Xeon E5503
	NVidia Quadro FX 4800
	4 Gbyte RAM
	Broadcom NetXtreme GigaBit Ethernet

Software:	Windows 7 Professional 32-bit Unity3D v4.6
Screen:	LG 55LM620T-ZE 55” 3DTV Acer GD245HQ 24” 3D Monitor

Motion Capture	
PC:	Intel Core i7-870 ATI Radeon HD 5700 12 Gbyte RAM Intel 82578DM Gigabit Network Connection
Software:	Windows 7 Enterprise 64-bit OptiTrack Tracking Tools v2.5.3 32-bit
Motion Capture System:	3x OptiTrack V100:R2 Cameras 1x OptiTrack OptiHub USB

For the motion capture, it was necessary to use a strong hardware configuration, as a decisive calculation power was needed to correctly process and transmit, without lag, the data to the computer that had to visualise the video game in 3D.

The test were performed in two different working spaces. The first one is illustrated in Figure 5.14, where the cameras are installed in order to capture a small area of $1m^3$ in front of the 24-inch monitor.

The second space, illustrated in Figure 5.15, has the cameras installed around a 55-inch 3DTV, capturing an area of $9m^3$ in front of it.

The room in which the system is installed must also satisfy certain specific requisites. There cannot be any obstacles capable of obscuring the cameras, and it must be completely isolated from external sources of light. Sunlight is a natural source of infrared light, which could disturb the system by generating “noise” that can easily be mistaken for undesired data, thus compromising the accuracy of the motion tracking.

Testing Procedure

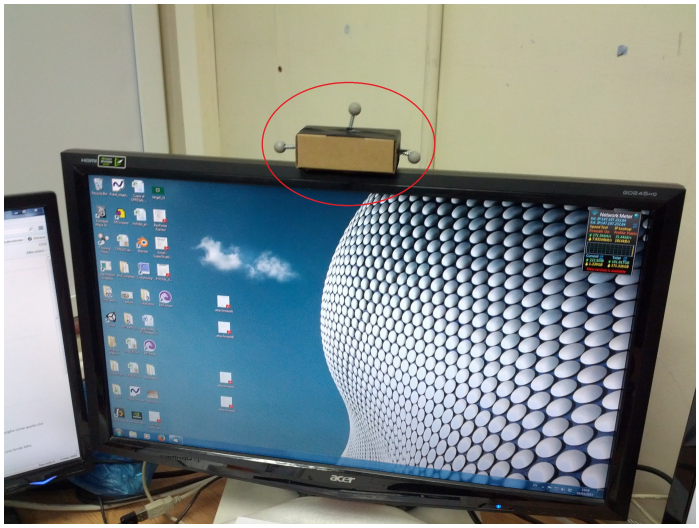
The real environment is prepared by placing the Monitor Tracker Set (Figure 5.11a) in front of the screen or on top of it, as shown in Figure 5.16.

The user is asked to stand in front of the monitor [4], and to wear the tracked glasses as well as the stereoscopic glasses. Once his or her position is correctly recognised by the motion capture, the game starts and the virtual environment is visualised according

[4] inside the camera’s field of view

Figure 5.14: Small Captured Area

Figure 5.15: Big Captured Area



(a) Monitor tracker set placed on the top of the screen.

(b) Monitor tracker set placed in front of the screen using a tripod.

Figure 5.16: The Monitor Tracker Set must be placed on or in proximity of the monitor.

to the information provided by the user and the screen position. It is then possible to visualise the stereoscopic virtual arrow using the tracked crossbow. The user can modify the depth perception and the arrow's position using the keyboard, to fit the virtual environment to his or her point of view.

The Stereoscopic-3D Visualisation is obtained using the "merging software" provided by the TV to visualise Side-by-Side contents.

Result Analysis

The stereoscopic effect is distinctly better than in the first version of the ArcheryGame, as it allows users to visualise the object with a standard error of 1-2 cm from the real object position, as opposed to the 10-15 cm of the previous version. Due to the Toed-in system, which can automatically change the angulation of the cameras, the stereoscopic arrow is constantly maintained on the same point in which the camera axes intersect. Furthermore, the scaling algorithm maintains the dimensions of the arrow in proportion to the distance between the user and the screen, and to the dimensions of the screen itself.

In this way, the video game adapts itself automatically and it is unnecessary for the user to introduce external data regarding the hardware in use and the dimensions of the room.

It was observed that it is impossible to define a standard distance, adequate for all users, between the cameras that compose the stereoscopic image. This is because the eye distance varies from person to person. It was also observed that the user's point of view significantly influences his or her capacity of perceiving the 3D effect, making it necessary to modify the game environment in relation to the user's height.

This evidence has been highlighted by asking two individuals of different heights to test the system while keeping the same configuration. The difference between the perceived position by the two testers is proportional to the difference between their respective heights. This issue can be easily overcome by providing the system with an automatic height calculator, which keeps users' different heights in consideration.

Despite the fact that the 3D effect satisfied the requirements set, the motion capture was unable to position the arrow with precision, as it always resulted in appearing 1 or 2 centimetres off in relation to the position of the bow. This error varies from person to person, in relation to the user's height and his or her perception of the stereoscopic effect. It was not possible to define a mathematical relationship capable of explaining this phenomenon. The problem can be overcome by calibrating at the beginning of the game, during which the user is able to move the arrow so as to place it exactly on top of the bow. Once this operation is carried out, the arrow will follow the movements of the bow and will be visualised in the correct position.

Chapter 6

Conclusions

6.1 Summary and Achievements

The initial objective of this research was to develop an innovative model of human-machine interaction, through the utilisation of Stereoscopic-3D Visualisation and Motion Capture technologies. The targeting application was a video game that must be playable only using Stereoscopic-3d Visualisation: The ArcheryGame. The main goal was to allow the user to visualise a virtual arrow through a Stereoscopic-3D screen, as it is seen on the top of a real crossbow. Motion Capture technology was used to identify the user's exact position inside the room.

Initially, a previous version of the ArcheryGame[1] was analysed, to identify the potential and the shortcomings of the solution. It was decided to completely modify the video game, only maintaining the concept, and to use a crossbow instead of a bow.

Based on the preliminary analysis, the three principal objectives listed below were identified:

- Develop an accurate motion capture technology setup that provides positional input to the game engine.
- Develop an accurate 3D Visualisation method, which allows the integrated system to display 3D visualisation information consistently to the motion capture.
- Develop an integrated system capable of managing motion captured positional information, 3D visualisation, and computer graphics objects visualised according to the 3D visualisation method developed in the previous point.

A first analysis of the state of the art allowed the identification of some related works necessary to master this research field. Two papers, in particular, were fundamental for

[1]Developed by the same Author for a Bachelor Thesis

the identification of a Stereoscopic-3D theory to visualise the virtual arrow in relation to externally provided information.

The developed system was divided in two modules: the Motion Capture Module and the Stereoscopic-3D Visualisation Module, both of which were deployed on two different machines. The presence of two computers allows for a division of the elaboration load, thus avoiding the lags and system crashes encountered in the first version of the game.

The game was developed entirely on the cross-platform game engine Unity3D, while a set of 6 infra-red cameras was used to track the user's position in the real environment. A network bridge allowed the communication between the two modules, using UDP data packets.

The Stereoscopic-3D visualisation was obtained using a system formed by two virtual cameras, placed Side-By-Side, which were able to modify their axis angle in relation to the position of the virtual arrow from them. The stereo visualisation of the arrow was compromised when it was too close to the cameras. For this reason, an algorithm was developed to scale the arrow dimension in relation to its distance from the cameras. This solution allowed the system to adapt itself in relation to the screen size used, and the user's distance from it.

Users testing the system found that the stereoscopic-3D effect allowed them to visualise the arrow as if it was at a distance of few centimetres from them. The game play resulted to be enjoyable, and the 3D effect added a fundamental feature to it. People found very difficult to play the video game with the Stereoscopic-3D effect turned off. Some issues were found during the tests, which should be tackled by further research:

- The arrow is not seen on the top of the crossbows by all the users.
- The arrow launch algorithm suddenly stopped working due to out-of-date Motion Capture libraries.
- The network communication bridge sometimes did not work properly.
- When moving the crossbow, movement of the arrow was slightly delayed.

6.2 Future Works

An automated calibration procedure was needed to compensate the error between the Motion Capture data and the visualised arrow. After calibration, the arrow should be visualised in accordance with the user's movements. The arrow launch algorithm issue must be solved using the cameras' up-to-date software, and this will also solve the occasional communication breakdown.

Subsequent tests using the LEAP motion device and the OptiTrack allowed for the development of simple interaction interfaces. The user is thus able to interact with a simple Stereoscopic User Interface, using hand movements as input and the stereoscopic effect to visualise the components of the interface. Indeed, the possibility of visualising a virtual object with good precision in a real environment can bring to a new way of interacting with digital contents, from entertainment to professional applications.

The theories described in this work lay the foundations for the development of new video games based on 3D stereoscopy, where the latter is no longer seen as merely an additional effect that enhances the gaming experience, but as a fundamental component capable of opening a new gaming sector in which it becomes an essential part of the game play.

Some example of such future developments could be a new generation of video games in which the stereoscopic object is capable of performing certain actions in relation to the users' behaviours and his or her position in the real environment such as, for example, a fencing simulator. Indeed, it would be possible to use a real environment as a playground, where everyday objects can be used as interaction device. On-line shopping can also take advantage of a similar technology, where users can have a preview of the object that they want to buy, avoiding wasting their money or embarking on annoying refund procedures.

Exploration and medical research would take an obvious benefit from such technology. Exploration of dangerous environments, such as volcanoes or broken-down buildings, can be done by telemanipulated robots, where the operators are able to operate inside the remote environment with extreme precision. Risky surgeries can be supported by visualisation systems that help the surgeon to keep the patient's situation easily under control.

Bibliography

[1] Jens Rosenkjaer Andersen and Bjarne Kondrup Mortensen. Bovw - bow operated virtual world. Final Report. Aalborg University, january 2007.

[2] Martin John Baker. Conversion quaternion to euler, 2015. [Online; accessed 05-February-2015].

[3] Marty Banks. Basic visual perception concepts related to 3d movies, 2009. presentation slides at NAB'09, Las Vegas, USA.

[4] K.K. Biswas and S.K. Basu. Gesture recognition using microsoft kinect. In *Automation, Robotics and Applications (ICARA), 2011 5th International Conference on*, pages 100–103, Dec 2011.

[5] Lin Du, Peng Qin, Jianping Song, Wenjuan Song, Yan Xu, and Wei Zhou. Immersive 3d user interface for 3d tvs. *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, pages 1–4, 2011.

[6] Entertainment Software Association™. The transformation of the video game industry, 2012. [Online; accessed 24-October-2014].

[7] M Ferre, R Aracil, and M Sanchez-Uran. Stereoscopic human interfaces: Advanced telerobotc applications for telemanipulation. *IEEE Robotics and Automation Magazine*, 2008:50–57, 2008.

[8] Ian P. Howard. Binocular vision and stereopsis, 1995. Oxford University Press.

[9] Shel Israel. Why apple bought primesense, November 2013. [Online; accessed 24-October-2014].

[10] Michael Karagosian and Younghoon Lee. 3d grows up: Stereoscopic technology continues to advance, December 2013. [Online; accessed 24-October-2014].

[11] C Keskin, A Erkan, and L Akarun. Real time hand tracking and 3d gesture recognition for interactive interfaces using hmm. *ICANN/ICONIPP*, 2003:26–29, 2003.

[12] P.T. Kovacs and Balogh T. 3d display technologies and effects on the human vision system, 2011. 2nd International Conference in Cognitive Infocommunications, Budapest.

[13] S Livatino, F Banno, and G Muscato. 3d integration of robot vision and laser data with semi-automatic calibration in augmented reality stereoscopic visual interface. *IEEE Transactions on Industrial Electronics*, 8(1):69–77, 2012.

[14] S Livatino, L De Paolis, and et al. Stereoscopic visualization and 3d technologies in medical endoscopic intervention. *IEEE Transactions on Industrial Electronics*, 62(1):525–535, 2015.

[15] S Livatino, G Muscato, S Sessa, and V Neri. Depth-enhanced mobile robot teleguide based on laser images. *Elsevier Journal of Mechatronics*, 20(7):739–750, 2010.

[16] Hoffman D M, Girshick A R, Akeley K, and Banks M S. Vergence-accomodation conflicts hinder visual performance and cause visual fatigue. *Journal of Vision*, pages 1–30, 2008.

[17] Mario Muratori. Stereoscopic visualization technique, 2007. RAI Innovation and Research Centre, Turin.

[18] Paul Oliver. *Succeeding with your Literature Review: A Handbook for Students*. McGraw-Hill Education, 2012.

[19] David Pierce. Google's insane all-seeing project tango tablet is coming to consumer next year, jun 2014. [Online; accessed 24-October-2014].

[20] Retail Gazette. Tissot launches augmented reality display at harrods, 2011. [Online; accessed 23-October-2014].

[21] Leadbetter Richard. Tech focus: Stereo-3d - year one, 2011. [Online; accessed 08-March-2015].

[22] Domenico Sammartino. Stereoscopic-3d games : Archery game. Bachelor Thesis, july 2013.

[23] Robert Stepien and Jon Frydensbjerg. Deer hunter: An immersive virtual reality simulation gaming system with a bow as interaction device. Master Thesis. Aalborg University, June 2002.

[24] The International Game Museum. Sega subroc-3d, 2009. [Online; accessed 08-March-2015].

[25] The Telegraph. Tesco trials augmented reality, 2011. [Online; accessed 23-October-2014].

[26] Virtual Worldlets Network. Sutherland's sword of damocles, 2009. [Online; accessed 08-March-2015].

[27] Charles Wheatstone. *Contributions to the Physiology of Vision.–Part the First. On Some Remarkable and Hitherto Unobserved, Phenomena of Binocular Vision*, pages 371–394. The Royal Society of London, 1838.

[28] Wikipedia , The Free Encyclopedia. Kinect, 2014. [Online; accessed 15-October-2014].

[29] H Yamanoue. The differences between toed-in camera configurations and parallel camera configurations in shooting stereoscopic images. *2006 IEEE Internation Conference on Multimedia and Expo*, pages 1701–1704, 2006.

[30] A Zocco, S Livatino, and L DePaolis. Stereoscopic-3d vision to improve situational awareness in military operations, 2014. Lecture Notes in Computer Science, Springer. Dec. 2014.

[31] Ray Zone. *3-D Revolution : The History of Modern Stereoscopic Cinema*, pages 387–390. The University Press of Kentucky, 2012.

Appendices

Appendix A

Code

A.1 OptiTrackUDPClient.cs

```
//#define DEBUG

using System;
using System.Net;
using System.Net.Sockets;

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

/**
* @class OptiTrackUDPClient
*
* OptiTrackUDPClient is a class for connecting to OptiTrack Arena Skeleton data
* or Tracking Tools RigidBody data and storing in a general state class object
* for access by Unity characters.
*
*/
public class OptiTrackUDPClient
{
public int dataPort = 1511;
public int commandPort = 1510;
public string multicastIPAddress = "239.255.42.99";
public string localIPAddress = "147.197.232.84";

public bool bNewData = false;
public Skeleton skelTarget = null;
public RigidBody rigidbody = null;

public RigidBody monitor = new RigidBody ();
public RigidBody hand = new RigidBody ();
public RigidBody head = new RigidBody ();

public int[] offsetAr = null;

Socket sockData = null;
Socket sockCommand = null;
String strFrame = "";

/**
*       @brief Constructor.
*/
public OptiTrackUDPClient () {}

/**
*       @brief Destructor.
*/
~ OptiTrackUDPClient()
{
if(sockCommand != null)
sockCommand.Close();
if(sockData != null)
sockData.Close();
```

```
}

/**
*       @brief Method that connects to Arena/Tracking Tools.
*/
public bool Connect() {
IPEndPoint ipep;
MyStateObject so;
#if DEBUG
Debug.Log("[UDPClient] Connecting.");
#endif
// create data socket
sockData = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
sockData.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.ReuseAddress, true);
ipep = new IPEndPoint(IPAddress.Any, dataPort);
//ipep = new IPEndPoint (IPAddress.Parse (localIPAddress), dataPort);
try
{
sockData.Bind(ipep);
}
catch (Exception ex) {
#if DEBUG
Debug.Log("bind exception : " + ex.Message);
#endif
}

// connect socket to multicast group
IPAddress ip = IPAddress.Parse(multicastIPAddress);
sockData.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership, new
    MulticastOption(ip, IPAddress.Any));
//sockData.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership, new
    MulticastOption(ip, IPAddress.Parse(localIPAddress)));
so = new MyStateObject();

// asynch - begin listening
so.workSocket = sockData;
sockData.BeginReceive(so.buffer, 0, MyStateObject.BUFFER_SIZE, 0, new
    AsyncCallback(AsyncReceiveCallback), so);

// create command socket
sockCommand = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
ipep = new IPEndPoint(IPAddress.Any, 0);
try
{
sockCommand.Bind(ipep);
}
catch (Exception ex){
#if DEBUG
Debug.Log("bind exception : " + ex.Message);
#endif
}
// asynch - begin listening
so = new MyStateObject();
so.workSocket = sockCommand;
sockCommand.BeginReceive(so.buffer, 0, MyStateObject.BUFFER_SIZE, 0, new
    AsyncCallback(AsyncReceiveCallback), so);

return true;
}

/**
* @brief Method that request the data description.
*/
public bool RequestDataDescriptions()
{
if(sockCommand != null)
{
Byte[] message = new Byte[100];
int offset = 0;
ushort[] val = new ushort[1];
val[0] = 4;
Buffer.BlockCopy(val, 0, message, offset, 1*sizeof(ushort));
offset += sizeof(ushort);
val[0] = 0;
Buffer.BlockCopy(val, 0, message, offset, 1*sizeof(ushort));
offset += sizeof(ushort);

IPEndPoint ipep = new IPEndPoint(IPAddress.Parse(localIPAddress), commandPort);
int iBytesSent = sockCommand.SendTo(message, ipep);
#if DEBUG
```

```
Debug.Log("[UDPClient] sent RequestDescription . (Bytes sent:" + iBytesSent + ")");
#endif
}

return true;
}

/**
* @brief
*/
public bool RequestFrameOfData()
{
if(sockCommand != null)
{
Byte[] message = new Byte[100];
int offset = 0;
ushort[] val = new ushort[1];
val[0] = 6;
Buffer.BlockCopy(val, 0, message, offset, 1*sizeof(ushort));
offset += sizeof(ushort);
val[0] = 0;
Buffer.BlockCopy(val, 0, message, offset, 1*sizeof(ushort));
offset += sizeof(ushort);

IPEndPoint ipep = new IPEndPoint(IPAddress.Parse(localIPAddress), commandPort);
int iBytesSent = sockCommand.SendTo(message, ipep);
#if DEBUG
Debug.Log("[UDPClient] Sent RequestFrameOfData. (Bytes sent:" + iBytesSent + ")");
#endif
}

return true;
}

/**
* @brief Async socket reader callback - called by .net when socket async receive procedure,
    receives a message
*  Una volta chiamata la classe da headTRacker o gli altri, in automatico fa partire la
    connessione Asyncrona...e dal metodo successivo parte la catena di lettura della frame
* @param
*
*/
private void AsyncReceiveCallback(IAsyncResult ar)
{
MyStateObject so = (MyStateObject)ar.AsyncState;
Socket s = so.workSocket;
int read = s.EndReceive(ar);
#if DEBUG
Debug.Log("[UDPClient] Received Packet (" + read + " bytes)");
#endif
if (read > 0)
{
// unpack the data
ReadPacket(so.buffer);
if(s == sockData)
bNewData = true;        // indicate to update character

// listen for next frame
s.BeginReceive(so.buffer, 0, MyStateObject.BUFFER_SIZE, 0, new
    AsyncCallback(AsyncReceiveCallback), so);
}

}

private void ReadPacket(Byte[] b)
{
int offset = 0;
int nBytes = 0;
int[] iData = new int[100];
float[] fData = new float[500];
char[] cData = new char[500];

Buffer.BlockCopy(b, offset, iData, 0, 2); offset += 2;
int messageID = iData[0];

Buffer.BlockCopy(b, offset, iData, 0, 2); offset += 2;
nBytes = iData[0];
#if DEBUG
Debug.Log("[UDPClient] Processing Received Packet (Message ID : " + messageID + ")");
#endif
```

```
if(messageID == 5)                // Data descriptions
{
strFrame = ("[UDPClient] Read DataDescriptions");

Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("Dataset Count: {0}\n", iData[0]);
int nDatasets = iData[0];

for(int i=0; i < nDatasets; i++)
{
//print("Dataset %d\n", i);

Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("Dataset # {0} (type: {1})\n", i, iData[0]);
int type = iData[0];



if(type == 0)    // markerset
{
// name
string strName = "";
while(b[offset] != '\0')
{
Buffer.BlockCopy(b, offset, cData, 0, 1); offset += 1;
strName += cData[0];
}
offset += 1;
strFrame += String.Format("MARKERSET (Name: {0})\n", strName);

// marker data
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("marker count: {0}\n", iData[0]);
int nMarkers = iData[0];

for(int j=0; j < nMarkers; j++)
{
strName = "";
while(b[offset] != '\0')
{
Buffer.BlockCopy(b, offset, cData, 0, 1); offset += 1;
strName += cData[0];
}
offset +=1;
strFrame += String.Format("Name : {0}\n", strName);
}
}
else if(type ==1)   // rigid body
{
if(major >= 2)
{
// name
char[] szName = new char [MAX_NAMELENGTH];
strcpy(szName, ptr);
ptr += strlen(ptr) + 1;
printf("Name: %s\n", szName);
}

int ID = 0; memcpy(&ID, ptr, 4); ptr +=4;
printf("ID : %d\n", ID);

int parentID = 0; memcpy(&parentID, ptr, 4); ptr +=4;
printf("Parent ID : %d\n", parentID);

float xoffset = 0; memcpy(&xoffset, ptr, 4); ptr +=4;
printf("X Offset : %3.2f\n", xoffset);

float yoffset = 0; memcpy(&yoffset, ptr, 4); ptr +=4;
printf("Y Offset : %3.2f\n", yoffset);

float zoffset = 0; memcpy(&zoffset, ptr, 4); ptr +=4;
printf("Z Offset : %3.2f\n", zoffset);
}
else if(type ==2)   // skeleton
{
InitializeSkeleton(b, offset);
#if DEBUG
Debug.Log("InitializeSkeleton");
#endif
}
```

```
}   // next dataset
#if DEBUG
Debug.Log(strFrame);
#endif

}
else if (messageID == 7)          // Frame of Mocap Data
{
strFrame = "[UDPClient] Read FrameOfMocapData\n";
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("Frame # : {0}\n", iData[0]);

// MarkerSets
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int nMarkerSets = iData[0];
strFrame += String.Format("MarkerSets # : {0}\n", iData[0]);
for (int i = 0; i < nMarkerSets; i++)
{
String strName = "";
int nChars = 0;
while (b[offset + nChars] != '\0')
{
nChars++;
}
strName = System.Text.Encoding.ASCII.GetString(b, offset, nChars);
offset += nChars + 1;

Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int nMarkers = iData[0]; strFrame += String.Format("Marker Count : {0}\n", nMarkers);



rigidbody.markersPos = new Vector3[nMarkers];


for(int j=0; j < nMarkers; j++) {
rigidbody.markersPos[j] = new Vector3(0F,0F,0F);


Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
rigidbody.markersPos[j].x = fData[0];


Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
rigidbody.markersPos[j].y = fData[0];


Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
rigidbody.markersPos[j].z = fData[0];

strFrame += String.Format("\tMarker {0} :
     [x={1},y={2},z={3}]\n",j,rigidbody.markersPos[j].x,
rigidbody.markersPos[j].y,rigidbody.markersPos[j].z);
}

}

// Other Markers
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int nOtherMarkers = iData[0]; strFrame += String.Format("Other Markers : {0}\n",
     nOtherMarkers);
for(int j=0; j < nOtherMarkers; j++) {
Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
float x = fData[0];

Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
float y = fData[0];

Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
float z = fData[0];

strFrame += String.Format("\tMarker {0} : [x={1},y={2},z={3}]\n",j,x,y,z);
}

//nBytes = iData[0] * 3 * 4;
//Buffer.BlockCopy(b, offset, fData, 0, nBytes); offset += nBytes;

// Rigid Bodies
//RigidBody rb = new RigidBody();
```

```
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int nRigidBodies = iData[0];
offsetAr = new int[nRigidBodies];
strFrame += String.Format("Rigid Bodies : {0}\n", iData[0]);
for (int i = 0; i < nRigidBodies; i++)
{
offsetAr[i] = offset;
strFrame += String.Format("Offset rigid body {0}: {1}\n", i+1, offsetAr[i]);
ReadRB(b, ref offset, this.rigidbody);
strFrame += String.Format("ID : {0}\n", rigidbody.ID);
strFrame += String.Format("pos: [{0},{1},{2}]\n",
    rigidbody.pos.x,rigidbody.pos.y,rigidbody.pos.z);
strFrame += String.Format("ori: [{0},{1},{2},{3}]\n", rigidbody.ori.x,
rigidbody.ori.y,rigidbody.ori.z,rigidbody.ori.w);
}

//store the two rigid bodies - I didn't find a better solution

int of1 = offsetAr[0];
int of2 = offsetAr[1];
int of3 = offsetAr[2];
ReadRB(b, ref of1, head);
ReadRB(b, ref of2, hand);
ReadRB (b, ref of3, monitor);

strFrame += String.Format("Head pos: [{0},{1},{2}]\n", head.pos.x, head.pos.y, head.pos.z);
strFrame += String.Format("Monitor pos: [{0},{1},{2}]\n", monitor.pos.x, monitor.pos.y,
    monitor.pos.z);
strFrame += String.Format("Hand pos: [{0},{1},{2}]\n", hand.pos.x, hand.pos.y, hand.pos.z);

// Skeletons
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int nSkeletons = iData[0];
strFrame += String.Format("Skeletons : {0}\n", iData[0]);
for (int i = 0; i < nSkeletons; i++)
{
// ID
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
skelTarget.ID = iData[0];
// # rbs (bones) in skeleton
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
skelTarget.nBones = iData[0];
for (int j = 0; j < skelTarget.nBones; j++)
{
ReadRB(b, ref offset, skelTarget.bones[j]);
}
}

// frame latency
Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;

// end of data (EOD) tag
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
#if DEBUG
Debug.Log(strFrame);
#endif
// debug
String str = String.Format("Skel ID : {0}", skelTarget.ID);
for (int i = 0; i < skelTarget.nBones; i++)
{
String st = String.Format(" Bone {0}: ID: {1}    raw pos ({2:F2},{3:F2},{4:F2})  raw ori
    ({5:F2},{6:F2},{7:F2},{8:F2})",
i, skelTarget.bones[i].ID,
skelTarget.bones[i].pos[0], skelTarget.bones[i].pos[1], skelTarget.bones[i].pos[2],
skelTarget.bones[i].ori[0], skelTarget.bones[i].ori[1], skelTarget.bones[i].ori[2],
    skelTarget.bones[i].ori[3]);
str += "\n" + st;
}
#if DEBUG
Debug.Log(str);
#endif

if(skelTarget.bNeedBoneLengths)
skelTarget.UpdateBoneLengths();


}
else if(messageID == 100)
{
#if DEBUG
```

```
Debug.Log("Packet Read: Unrecognized Request.");
#endif
}

}

// Unpack RigidBody data
private void ReadRB(Byte[] b, ref int offset, RigidBody rb)
{
int[] iData = new int[100];
float[] fData = new float[100];

// RB ID
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int iSkelID = iData[0] >> 16;                    // hi 16 bits = ID of bone's parent skeleton
int iBoneID = iData[0] & 0xffff;                 // lo 16 bits = ID of bone
rb.ID = iData[0]; // already have it from data descriptions

// RB pos
float[] pos = new float[3];
Buffer.BlockCopy(b, offset, pos, 0, 4 * 3); offset += 4 * 3;
rb.pos.x = pos[0]; rb.pos.y = pos[1]; rb.pos.z = pos[2];

// RB ori
float[] ori = new float[4];
Buffer.BlockCopy(b, offset, ori, 0, 4 * 4); offset += 4 * 4;
rb.ori.x = ori[0]; rb.ori.y = ori[1]; rb.ori.z = ori[2]; rb.ori.w = ori[3];

QuaternionToEuler(rb.ori.x, rb.ori.y, rb.ori.z, rb.ori.w, out rb.yaw, out rb.pitch, out
    rb.roll);
rb.pitch *= -1.0f;
rb.yaw = (RadiansToDegrees(rb.yaw));
rb.pitch = (RadiansToDegrees(rb.pitch));
rb.roll = (RadiansToDegrees(rb.roll));

// RB's markers
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int nMarkers = iData[0];

// RB's makers data
Buffer.BlockCopy(b, offset, fData, 0, 4 * 3 * nMarkers); offset += 4 * 3 * nMarkers;
float[] markerData = fData;

// RB's marker ids
Buffer.BlockCopy(b, offset, iData, 0, 4 * nMarkers); offset += 4 * nMarkers;

// RB's marker sizes
Buffer.BlockCopy(b, offset, fData, 0, 4 * nMarkers); offset += 4 * nMarkers;

// RB mean error
Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;

}

void InitializeSkeleton(Byte[] b, int offset)
{
int[] iData = new int[100];
float[] fData = new float[500];
char[] cData = new char[500];

string strName = "";
while(b[offset] != '\0')
{
Buffer.BlockCopy(b, offset, cData, 0, 1); offset += 1;
strName += cData[0];
}
offset += 1;
strFrame += String.Format("SKELETON (Name: {0})\n", strName);
skelTarget.name = strName;

Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("SkeletonID: {0}\n", iData[0]);
skelTarget.ID = iData[0];

Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("nRigidBodies: {0}\n", iData[0]);
skelTarget.nBones = iData[0];

for(int j=0; j< skelTarget.nBones; j++)
{
```

```
// RB name
string strRBName = "";
while(b[offset] != '\0')
{
Buffer.BlockCopy(b, offset, cData, 0, 1); offset += 1;
strRBName += cData[0];
}
offset += 1;
strFrame += String.Format("RBName: {0}\n", strRBName);
skelTarget.bones[j].name = strRBName;

// RB ID
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
int iSkelID = iData[0] >> 16;                    // hi 16 bits = ID of bone's parent skeleton
int iBoneID = iData[0] & 0xffff;          // lo 16 bits = ID of bone
//Debug.Log("RBID:" + iBoneID + "  SKELID:"+iSkelID);
strFrame += String.Format("RBID: {0}\n", iBoneID);
skelTarget.bones[j].ID = iBoneID;

// RB Parent
Buffer.BlockCopy(b, offset, iData, 0, 4); offset += 4;
strFrame += String.Format("RB Parent ID: {0}\n", iData[0]);
skelTarget.bones[j].parentID = iData[0];

// RB local position offset
Vector3 localPos;
Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
strFrame += String.Format("X Offset: {0}\n", fData[0]);
localPos.x = fData[0];

Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
strFrame += String.Format("Y Offset: {0}\n", fData[0]);
localPos.y = fData[0];

Buffer.BlockCopy(b, offset, fData, 0, 4); offset += 4;
strFrame += String.Format("Z Offset: {0}\n", fData[0]);
localPos.z = fData[0];
skelTarget.bones[j].pos = localPos;

//Debug.Log("[UDPClient] Added Bone: " + skelTarget.bones[j].name);

}
skelTarget.bHasHierarchyDescription = true;
}

// Convert a quaternion (TrackingTools type) to euler angles (Y, Z, X)
// Y = Heading  (Yaw)
// Z = Attitude (Pitch)
// X = Bank     (Roll)
// From Martin Baker (http://www.euclideanspace.com/maths/geometry/
//rotations/conversions/quaternionToEuler/index.htm)
// conventions:
//  - input and output units are both in radians
//  - euler order is YZX
//  - euler angles are about global axes
//  - euler + angle is right-handed
public void QuaternionToEuler(float qx, float qy, float qz, float qw, out float y, out
    float z, out float x)
{
float test = qx * qy + qz * qw;
if (test > 0.499)                              // singularity at north pole
{
y = 2.0F * Mathf.Atan2(qx, qw);
z = Mathf.PI / 2.0F;
x = 0.0F;
return;
}

if (test < -0.499)                             // singularity at south pole
{
y = -2.0F * Mathf.Atan2(qx, qw);
z = -Mathf.PI / 2.0F;
x = 0.0F;
return;
}

float sqx = qx * qx;
float sqy = qy * qy;
float sqz = qz * qz;
```

```
y = Mathf.Atan2(2.0F * qy * qw - 2.0F * qx * qz, 1.0F - 2.0F * sqy - 2.0F * sqz);
z = Mathf.Asin(2.0F * test);
x = (float)Mathf.Atan2(2.0F * qx * qw - 2.0F * qy * qz, 1.0F - 2.0F * sqx - 2.0F * sqz);
}

public float RadiansToDegrees(float dRads)
{
return dRads * (180.0f / Mathf.PI);
}

// normalising a quaternion works similar to a vector. This method will not do anything
// if the quaternion is close enough to being unit-length. define TOLERANCE as something
// small like 0.00001f to get accurate results
public void QuaternionNormalise(ref float x, ref float y, ref float z, ref float w)
{
// Don't normalize if we don't have to
float tolerance = 0.00001f;
float mag2 = w * w + x * x + y * y + z * z;
if (mag2 != 0.0f && (Math.Abs(mag2 - 1.0f) > tolerance))
{
float mag = (float)Math.Sqrt(mag2);
w /= mag;
x /= mag;
y /= mag;
z /= mag;
}
}
//close the socket connection. remeber to call it with OnApplicationQuit(), otherwise the
    connection
//will remain always open until the system automatically close it for a timeout
public void closeConnection() {
sockData.Close ();
sockCommand.Close ();
}
}

public class MyStateObject
{
public Socket workSocket = null;
public const int BUFFER_SIZE = 65507;
public byte[] buffer = new byte[BUFFER_SIZE];
}
```

A.2 handTracking.cs

```
//#define DEBUG

using UnityEngine;
using System.Collections;

public class handTracking : MonoBehaviour {
public OptiTrackUDPClient udpClient = null;
private bool bSuccess;

public string multicastIPAddress;
public string localIPAddress;
public int dataPort;

public float sensitivityX = 5F;
public float sensitivityY = 5F;
public float sensitivityZ = 5F;

private float minimumX = -1000F;
private float maximumX = 1000F;

private float minimumY = -1000F;
private float maximumY = 1000F;

private float minimumZ = -1000F;
private float maximumZ = 1000F;

private float positionY = 0F;
private float positionX = 0F;
private float positionZ = 0F;

private Rigidbody rigidBody = new Rigidbody();
```

```
void Start ()
{
// Make the rigid body not change rotation
if (this.rigidbody)
this.rigidbody.freezeRotation = true;

Skeleton skelPerformer = new ArenaSkeleton();
RigidBody rbPerformer = new RigidBody();

// Connect to a Live Arena session
udpClient = new OptiTrackUDPClient();
udpClient.localIPAddress = localIPAddress;
udpClient.multicastIPAddress = multicastIPAddress;
udpClient.dataPort = dataPort;
udpClient.skelTarget = skelPerformer;    // <-- live data will be stuffed into this
    skeleton object
udpClient.rigidbody = rbPerformer;
bSuccess = udpClient.Connect();
if(this.bSuccess) {
udpClient.RequestDataDescriptions();
}

}

void Update ()
{
#if DEBUG
Debug.Log ("hand pos: [" + udpClient.hand.pos.x + "," +
udpClient.hand.pos.y + "," +
udpClient.hand.pos.z + "]");
#endif
if (this.bSuccess) {

float trX = (udpClient.hand.pos.x - positionX) * sensitivityX;
float trY = (udpClient.hand.pos.y - positionY) * sensitivityY;
float trZ = (udpClient.hand.pos.z - positionZ) * sensitivityZ;

transform.Translate (trZ, trY, trX);

positionX = udpClient.hand.pos.x;
positionY = udpClient.hand.pos.y;
positionZ = udpClient.hand.pos.z;
}
}
void OnApplicationQuit() {
udpClient.closeConnection ();
}

}
```

A.3 ToedInCamera.cs

```
//#define DEBUG

using UnityEngine;
using System.Collections;

public class ToedInCamera : MonoBehaviour {

float cameraDist = 0.0f;
float objectDist = 0.0f;
float angleAlpha = 0.0f;
float angleBetha = 0.0f;

private float oldCameraDist = 0.0f;
private float oldObjectDist = 0.0f;

float rotationA = 0.0f;
float rotationB = 0.0f;

bool appRotation;

public Camera right;
public Camera left;

public GameObject source;
public GameObject target;
```

```
//debug purposes
string str = "";

void Start()
{
appRotation = true;
}

// Update is called once per frame
void Update () {


//To calculate the angle between the two camera the precess is:
//1- Calculate the half distance between the two camera
float distx = Mathf.Pow ((left.transform.position.x - right.transform.position.x),
    2.0f);
float disty = Mathf.Pow ((left.transform.position.y - right.transform.position.y),
    2.0f);
float distz = Mathf.Pow ((left.transform.position.z - right.transform.position.z),
    2.0f);

cameraDist = Mathf.Sqrt (distx + disty + distz);

cameraDist /= 2.0f;
#if DEBUG
str = "";
str += "Camera Dist/2: " + cameraDist + '\n';
#endif
//2- Calculate the distance between the object and the cameras
distx = Mathf.Pow (target.transform.position.x - source.transform.position.x, 2.0f);
disty = Mathf.Pow (target.transform.position.y - source.transform.position.y, 2.0f);
distz = Mathf.Pow (target.transform.position.z - source.transform.position.z, 2.0f);

objectDist = Mathf.Sqrt (distx + disty + distz);
#if DEBUG
str += "Camera-to-object distance: " + objectDist + '\n';
#endif
if (oldCameraDist != cameraDist || oldObjectDist != objectDist)
{
appRotation = true;
oldCameraDist = cameraDist;
oldObjectDist = objectDist;
}
//3- Imagine to have a triangle between the two cameras, so using the two length we
    have: alpha=arctang(c/b)
angleAlpha = Mathf.Atan(cameraDist/objectDist);
#if DEBUG
str += "Angle Alpha: " + angleAlpha*Mathf.Rad2Deg +'\n';
#endif
//4- then betha= pi/2 - alpha
angleBetha = Mathf.PI/2 - angleAlpha;
angleBetha = angleBetha * Mathf.Rad2Deg;
#if DEBUG
str += "Angle between the two cameras: " + angleBetha;
Debug.Log(str);
#endif
//now we need to assign the value to each camera, first subtracting 90 degree from
    the previous result
//because we need to know how much to rotate inwards the camera to obtain that angle
if (appRotation)
{
left.transform.rotation = Quaternion.Euler(0.0f,90-angleBetha, 0.0f);
right.transform.rotation = Quaternion.Euler(0.0f, angleBetha - 90, 0.0f);

rotationA = left.transform.rotation.y;
rotationB = right.transform.rotation.y;

appRotation = false;
}

}

private Vector3 Vector3(double p1, float angleBetha, double p2)
{
throw new System.NotImplementedException();
}
}
```

A.4 StereoAlgorithm.cs

```
#define DEBUG
using UnityEngine;
using System.Collections;

public class StereoAlgorithm : MonoBehaviour {

monitor scr;
float z = 30.0f;
float d = 84.0f;
protected float p = 0.65f;
public GameObject cube;

//debug
string str = "";

public formula() {
scr = new monitor (Screen.height, Screen.width);
}

public float[] calcObjectDim(float zedPos) {
float[] dim = new float[2];

dim[0] = (zedPos * (scr.getWidth() + p)) / d - p;
dim[1] = (zedPos * scr.getheigth ()) / d;

return dim;
}

}

class monitor {
private float heigthCm = 0.0f;
private float widthCm = 0.0f;

private int heigthRes = 0;
private int widthRes = 0;

private float diagonalInch = 24.0f;
private float aspectRatio = 0.0f;

public monitor(int resH, int resW) {
heigthRes = resH;
widthRes = resW;
calculateMonitorDim ();
}
private float calculateRatio() {
aspectRatio = widthRes / heigthRes;

return (diagonalInch / Mathf.Sqrt (Mathf.Pow (aspectRatio, 2) + 1));
}
private void calculateMonitorDim() {
heigthCm = calculateRatio () * 2.54f;
widthCm = aspectRatio * calculateRatio () * 2.54f;
}

public float getheigth() {
return heigthCm;
}

public float getWidth() {
return widthCm;
}

}
```