

École polytechnique de Louvain

Development of an interface for combining a motion-tracking system with a virtual-reality headset

Authors: **Jérôme DELFORGE, Victor FISCHER**
Supervisors: **Philippe LEFÈVRE, Laurent OPSOMER**
Readers: **Philippe LEFÈVRE, Laurent OPSOMER, Charles PECHEUR**
Academic year 2021–2022
Master [120] in Computer Science and Engineering
Master [120] in Biomedical Engineering

Abstract

In order to understand the underlying mechanisms of our body, the System and Cognition (COSY) division of the Institute of NeuroScience explores dexterous manipulation, motor controls and sensory-motor coordination. For the purpose of bringing their research a step ahead, the members of the laboratory recently acquired a virtual reality headset. The goal of this master thesis is to combine this headset with a motion tracking system to make a fully operable set-up for future experiments. Virtual reality is a powerful tool to generate sensory conflicts and study their consequences on the human brain. The interface developed in the framework of this thesis is polyvalent, easy to handle and compatible with the other equipment of the laboratory. This document is intended to be the groundwork for future experiments conducted in the COSY lab.

Acknowledgements

With these few lines, we would like to thank all the people who helped us directly or indirectly in the realisation of this project.

Firstly, we address a special thanks to our supervisors Pr. Philippe Lefèvre, for his guidance and advice, and Laurent Opsomer, for his close follow-up of the project and all the knowledge that he transmitted to us.

We also thank Simon Vandergooten for his time and answers when Laurent was not available. And more generally, we would like to thank all the members of the COSY lab for their enthusiasm regarding our project.

We would like to express our gratitude to Joel Mitchelson from the Codamotion support service for helping us to get started with RTNet SDK and for answering all the questions we had.

We also thank the four friends of ours who agreed to take part in our pilot experiment, Sébastien, Quentin, Guillaume and Benoit.

Finally, we would like to thank our friends and families who supported us throughout the year and long before.

Contents

1	Introduction	4
2	State of the art	6
2.1	Motor control	6
2.2	Load Force, Grip Force, and Safety Margin	7
2.3	Internal models	8
2.4	Sensory feedback	9
2.4.1	Visual feedback	9
2.4.2	Vestibular feedback	10
2.4.3	Cutaneous feedback	10
2.4.4	Proprioceptive feedback	10
2.5	Adaptation	11
2.6	Multimodal information	11
2.7	Gravity estimation and verticality perception	12
2.8	Different environments	13
2.8.1	Rotating chair	13
2.8.2	Parabolic flight	14
2.9	Virtual Reality	14
2.9.1	Analysis of visual feedback	15
2.9.2	Influence of sensory and acoustic feedback	15
2.9.3	Influence of visual delay	17
2.10	Conclusion	17
3	Materials	18
3.1	Oculus Quest 2	18
3.2	Vizard	19
3.3	Codamotion cameras	20
3.3.1	CODA CX1 sensor unit	20
3.3.2	Mini hub unit	21
3.3.3	Drive boxes and LED markers	21
3.3.4	RTNet SDK	21
3.4	Manipulandum	22
3.5	Rotating chair	22
4	Interface	23

5	Implementation	25
5.1	Before getting started	25
5.2	The plug-in	26
5.2.1	Project settings	27
5.2.2	The <i>Coda</i> class	27
	Creating the plug-in	27
	Coda::run - Starting the cameras	29
	Aligning the cameras	31
	Coda::acquire - Performing an acquisition	33
	Coda::saveData - Saving the data	35
	Interacting with our plug-in from Vizard	37
5.2.3	The <i>Sensor</i> class	38
	Creating the sensor	38
	Retrieving the position	41
	Retrieving the orientation	42
5.3	Using the plug-in in Vizard	46
5.3.1	Importing the plug-in	46
5.3.2	Creating a sensor	46
5.3.3	Aligning virtual with real	46
	From Codamotion to headset	48
	From headset to VR	49
	The full transformation matrix	49
5.4	Final product	50
5.4.1	Summary	50
5.4.2	Codamotion.py	51
5.4.3	Example of a Vizard script	53
6	Usage	54
6.1	Usage of the set-up	54
6.1.1	Before the experiment	54
	Installing the drive boxes and LED markers	54
	Installing the cameras	55
	In Vizard	56
6.1.2	During the experiment	57
6.1.3	After the experiment	58
6.2	How to implement an experiment	58
6.2.1	Launching a new world	58
6.2.2	Creating a virtual environment	59
	Modelling	59
	Lighting	62
	Orienting	62
6.2.3	Flow control	63
	Events	63
	Actions	65
	Controlling objects	67
	Tasks	68

6.2.4	Proximity sensors	69
7	Pilot experiment	71
7.1	Introduction	71
7.2	Materials and methods	71
7.2.1	Participants	71
7.2.2	Experimental setup	71
7.2.3	Virtual environment	72
7.2.4	Experimental task	72
7.2.5	Data collection	73
7.3	Results and discussion	73
7.3.1	Analysis of the forces	73
	First impression	73
	Measure of the GF-LF delay	74
	GF when LF is max	74
	Comparison of all subjects	75
7.3.2	Analysis of the positions	76
7.4	Conclusion	78
7.4.1	Discussion of the experiment	78
7.4.2	Issues faced	78
8	Conclusion	80
8.1	Achievements	80
8.2	Limitations	80
8.3	Research perspectives	81
8.4	Conclusion	82
	Bibliography	83
A	Create a new project	86
B	Project settings	89
C	Complete diagram of the interface	92

Chapter 1

Introduction

Object manipulation is part of our everyday life: when grabbing a cup of coffee, when holding our smartphone, etc. Even if these actions are very common, they are still not totally understood. Indeed, how our brain is capable of adjusting forces with so much precision when grasping an object? How does it compute the force needed to keep it from falling?

To understand with precision all the underlying mechanisms that are present in our body, lots of scientists conduct experiments to unravel the mysteries of dexterous manipulation. Moreover, this understanding may help in other scientific fields, such as robotics or prosthesis.

To improve knowledge on this subject, our promoters Philippe Lefèvre and Laurent Opsomer bought a virtual reality (VR) headset. This purchase will contribute to the long-term vision of their laboratory in the System and Cognition (COSY) division of the Institute of NeuroScience, at UCLouvain. Among other topics, the researchers of this laboratory study dexterous manipulation, motor control and sensory-motor coordination.

In this regard, virtual reality is a great way to generate sensory conflicts and study their consequences on object manipulation. How would our brain react if our eyes tell us that everything is fine, but our inner ear tells us that the up is down and vice versa? Or if the object that we manipulate goes in the opposite direction to the movement we give it? In short, VR is a powerful tool to explore new fields of research and open new possibilities for the future of neuroscience.

With that in mind, our main task consisted in connecting this VR headset with a system of motion-tracking cameras. These cameras can track the positions of LED markers that can be placed anywhere (on the body, on objects, etc), depending on the goal of the experiment.

Our objective was to create a system that collects the positions monitored in real-time and integrate them into the virtual environment (VE) displayed in the headset. For example, we could stick markers on an object and recreate this object in the VE. Thanks to our interface, the virtual object would move and rotate following the physical object, or in another way programmed by the researcher to trick the subject's brain!

As this project is meant to pave the way for future experiments with this equipment, one of our biggest concerns was to provide a robust code working in all situations. We tried to make our system as polyvalent as possible while still being accessible and easy to handle. In the same spirit, we built parts of this document like tutorials that can be followed by future users of our interface.

This report is divided into several chapters. In chapter 2, we set the whole context by explaining the key concepts underlying the research in dexterous manipulation. In chapter 3, we present the materials and software that we used for this project. Chapters 4 and 5 contain the details of the implementation of our system. We explain how to use it in chapter 6 as well as how to build an experiment using VR. Finally, we detailed in chapter 7 the small pilot experiment that we conducted to test the proper functioning of our interface.

Chapter 2

State of the art

In this section, we summarise important concepts to apprehend this master thesis. We present the background knowledge needed to understand dexterous manipulation and sensory-motor coordination, as well as what virtual reality can bring to this research field.

2.1 Motor control

Our everyday movements are regulated by our central nervous system (CNS). To control our movements, it integrates multiple sources of information and then recruits muscles to complete a task, such as walking, grabbing an object, etc. Motor control is very interdisciplinary and gathers lots of different aspects such as multisensory integration (see section 2.6), signal processing, coordination, biomechanics and cognition [6]. Motor control is divided in four parts, as shown in figure 2.1 [28].

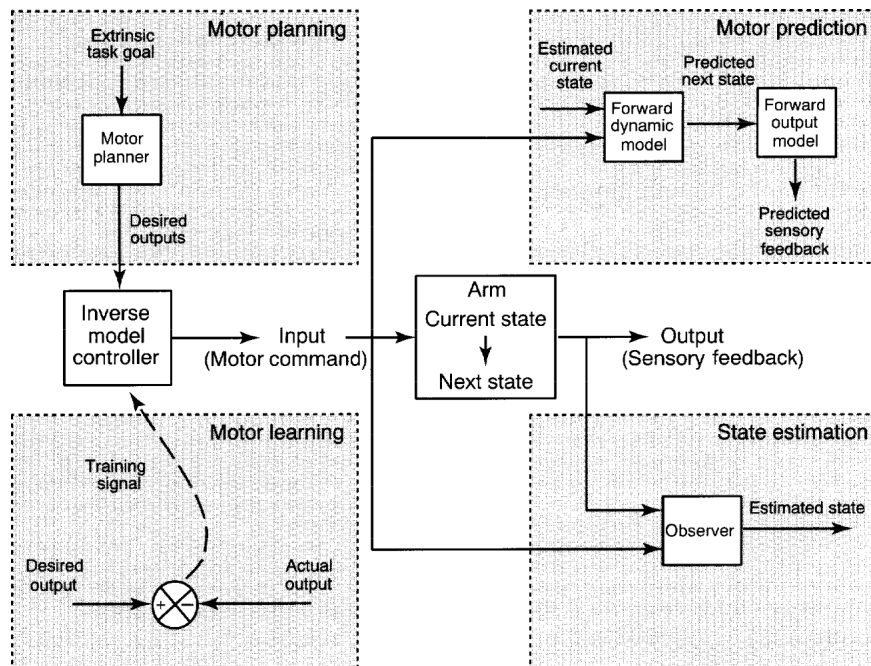


Figure 2.1: Schematic motor system [28]

The first one is motor planning, which gives the desired states or outputs to realise a specific goal. For example, there is an infinite number of ways for a hand to reach a target. Motor planning is the task of selecting the best possibility among them, the aim is to minimise the cost function in order to make as little effort as possible.

Secondly, motor prediction, as its name suggests, predicts the consequences of our actions. It uses internal models (see section 2.3) to predict the next state, based on an estimation of the current state.

This estimation of the current state is done thanks to the third part: state estimation. Indeed, the state of the system is not directly available to the controller. This estimation is based on the stream of motor commands and sensory feedback.

Finally, motor learning adapts the internal models to different situations and changes such as the growth of the body for example.

2.2 Load Force, Grip Force, and Safety Margin

Forces are an essential part of motor control. A force can be the result of a motor command, or can represent a sensory feedback. The two forces widely studied in object manipulation are the grip force (GF) and the load force (LF).

The GF is the normal force that we apply to the surface of an object when we are holding it. The LF is the tangential force to the surface of an object, varying according to the object's weight and acceleration. The safety margin is defined as the difference between the GF/LF ratio and the slip force/LF ratio, where the slip force is the minimal force required to prevent an object from slipping.

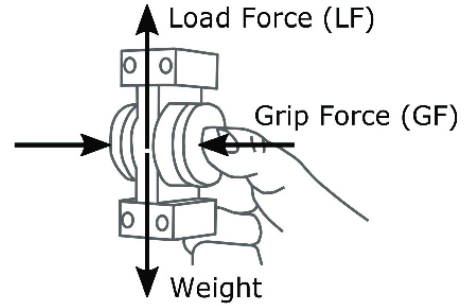


Figure 2.2: GF and LF [23]

$$\text{Safety margin} = \frac{GF}{LF} - \frac{\text{slip force}}{LF}$$

The GF should exceed the slip force with a safety margin depending on the context (uncertainty, object fragility, task goal,...) [2]. The GF is also modulated in parallel with the LF. Indeed, the LF varies during object manipulation (according to the object's weight and acceleration) and the GF needs to be sufficient to avoid slipping (with μ the friction coefficient): $\|\vec{LF}\| \leq \mu \|\vec{GF}\|$

This phenomenon happens regardless of movement frequency and the gripping surface's texture. Nevertheless, some conditions may change the grip force gain, and therefore the safety margin. For example, it has been observed that if a subject sees a ball falling into a cup they are holding (and thus expect an incoming increase of LF), there is an increase of GF before the impact, to prevent slipping [11].

The CNS is thus capable of synchronising these two forces during object manipulation to ensure a stable grip, with a delay of -10 to 15ms (-10 because the CNS anticipates a potential increase in LF). This delay is smaller than sensorimotor feedback (30 to 250ms), meaning that this coordination is planned by the CNS. It relies on sensorimotor feedback and internal models, which are mechanisms employed by the CNS to predict sensory and physical consequences of actions. The GF is thus regulated by a low-level mechanism (quick response) and a high-level motor command (slower, for voluntary adjustments) [22] [14].

2.3 Internal models

During the beginning of the 1990s, computational studies proposed a new concept saying that the CNS was using internal models that mimic the dynamic behaviour of motor control, planning and learning [29]. They combine different sources of information to predict the correct motor command to be adopted. At the end of the 1990s, and until now it is considered obvious, but at that time, it was a topic of debate [15].

Internal models can be used by a feedforward controller (for anticipation of the desired action) and they are continuously updated based on sensory feedback (involved in reflexes for example) [17]. There are two types of internal models, as illustrated on figure 2.3.

The first one is the inverse model, which uses the desired and actual state to estimate the accurate arm motor commands needed to reach the desired state. The second one is the forward model, which was discovered and has since been studied through research on coordination between reaching and grasping [15]. It calculates the predicted next state from the motor command of the inverse model and the estimated current state. This predicted trajectory is then sent to the GF controller, which generates the most accurate hand motor command possible [5].

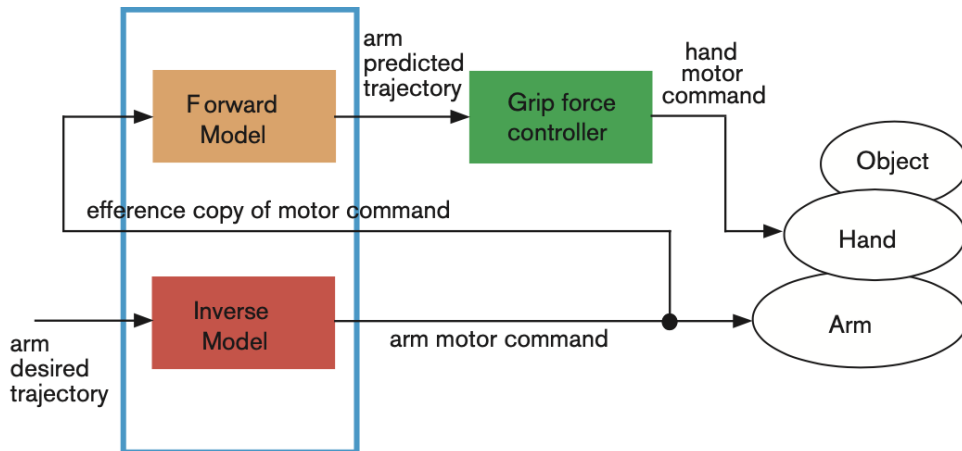


Figure 2.3: Computational model based on both internal models [15]

In the COSY lab (and in the scientific world), lots of experiences are performed to understand how internal models work. Virtual reality will help scientists to go further and deeper in their comprehension. For instance, by testing whether visual feedback about body orientation is taken into account to predict the adequate GF required for a stable grasp during arm movements.

2.4 Sensory feedback

Without our nerves, we are deprived of sensory feedbacks, which are required to adapt the command that the CNS sends to our muscles to make the correct task. Among all the information that reaches our brain, somatosensory (involving cutaneous and proprioceptive), vestibular and visual feedback are particularly important for motor control. That is why most research focus on these feedbacks, soon to be helped by VR.

2.4.1 Visual feedback

The control of GF during manipulations relies on internal models, for which eyesight is an important source of information. In a study conducted by S. Toma, V. Caputo and M. Santello in 2020 [24], subjects were asked to perform vertical movements with a manipulandum, with its position displayed on a monitor (figure 2.4). The movement of the displayed object was either congruent (in the same direction as the real object) or incongruent (in the opposite direction).

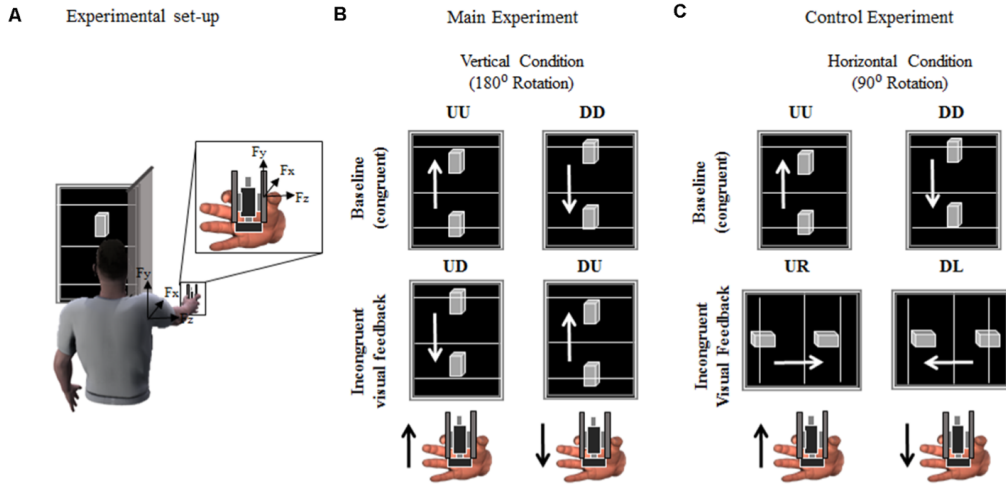


Figure 2.4: Experimental setup, main and control experiment [24]

Participants were divided into two experimental groups: the main one and the control one. In both cases, the congruent condition was the same, the displayed object followed the same direction as the real object. However, for the incongruent condition, the object movements were shown vertically to the main group, and horizontally to the control group (see figure 2.4).

Among the conclusions of the article, it claims that visual feedback significantly influenced time-to-peak GF. In the UD/DU (bottom of figure 2.4.B), the GF peaks are shifted respectively later/earlier than the baseline. These observations seem to confirm the concept of visual gravity [19].

Moreover, a smaller effect of visual feedback on GF modulation for UD than DU incongruent trials suggests that visual and somatosensory feedback are differently integrated into the prediction, depending on the direction of object displacement.

The main advantage of integrating virtual reality into scientific experiments is the possibility to modify the visual feedback. Not only on a screen such as the one used in the previous study, but in a completely immersive way and in 3 dimensions. For example, when monitoring the position of a real object, researchers could alter the direction or amplitude of the movements. With a virtual environment, it is also possible to induce visual cues modifying the visual perception of gravity, or deviate the attention of the subject from its task.

2.4.2 Vestibular feedback

The vestibular system is situated in the inner ear, it induces the sense of balance and the sensation of movement. Thanks to its structure, it indicates rotational movements and linear accelerations to the neural system. It is also involved in lots of brain functions like spatial orientation, oculomotor control or head rotation. It is thus very important and crucial in our case, particularly to estimate the orientation of the body relative to gravity [9].

2.4.3 Cutaneous feedback

The cutaneous feedback and its implications in internal models are widely studied. In 2003, Augurelle et al. wanted to determine the contribution of feedforward and feedback control during cyclic arm movements, with tactile feedback of the thumb and the index anaesthetised [2].

Their results showed that for half of their trials (with subjects who have anaesthetised fingers), the subjects let the object slip, due to an insufficient GF/LF ratio. Indeed, the GF control and adaptation were a lot impacted and the safety margin was higher. They concluded that the cutaneous feedback is necessary for the internal models to update motor commands [14].

2.4.4 Proprioceptive feedback

The proprioception gives information about the body itself, in particular it refers to the sensation of body position and movement in space. This sense is essential in our everyday life, to control our movements and body.

For example, Mr Ian Waterman lost this ability in the 1980s [25]. Due to an infection that attacked the sensory neurons from his neck, he lost the touch, movement and position sensations, and was thus completely unable to coordinate his movements. Every sensory receptor in the body plays a role in the proprioceptive feedback.

2.5 Adaptation

In motor control, we usually use the term "adaptation" to refer to the modifications required to accomplish a task following a change. However, sometimes we need to learn this adaptation. For example, if someone wants to lift an empty box that looks heavy, he will produce a much higher GF than what the LF requires. At the second lift, an accurate force scaling could be expected. But if we change suddenly the box weight (not the visual appearance), he will surely generate inappropriate forces.

To learn and maintain this ability to generate correct motor commands, the motor system needs to predict the sensory consequences of these motor commands. It is based on an estimation of the expected weight of the object, influenced by multiple sources of information (sensory feedback) and other parameters, such as the sensorimotor memory [10]. This estimation is part of the internal models used to predict the consequences of our movements.

2.6 Multimodal information

As said before, the CNS combines several sources of information. This phenomenon is called multimodal information, or multisensory integration.

In 2007, A.B. Le Seac'h and J. McIntyre [16] assumes that the CNS combines several sensory feedback (2.4) and cognitive cues for its perception of the vertical (estimation of the direction of gravity). More precisely, they propose that body axis and/or visual clues are elements that defined the reference frame of the subject used for movement planning and control.

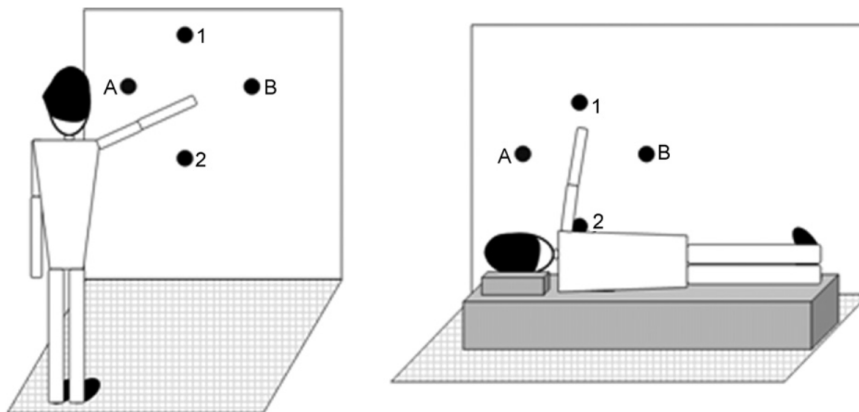


Figure 2.5: Experimental setup [16]

To verify their assumptions, they asked eleven volunteers to perform horizontal and vertical arm pointing movements. The participants were doing these movements in two different positions (upright and lying) and with or without vision, as illustrated in figure 2.5. The results showed asymmetries in the movement direction when the subjects performed the movement in the body axis direction, when they were lying with eyes closed. From this observation, they concluded that the CNS uses an internal model of the effect of gravity based on multimodal information, i.e., information coming from different sensory systems and then combined.

Once again, through VR, researchers could study the concept of multisensory integration by inducing discrepancies between the different sensory feedbacks and analysing their consequences on the behaviour of the CNS.

2.7 Gravity estimation and verticality perception

In our everyday life, all our actions and behaviours are influenced by Earth's gravity, without us being aware of it. For example, our body has learned to take gravity's effect into account, and our body structure (muscles, bones, etc) has evolved consequently [16]. In their paper [8], C. Dakin and A. Rosenberg summarised experiences about gravity estimation (finding the direction of the gravity vector) and verticality (relative to gravity) perception, which are essential components of motor control.

In the research on perception of vertical, subjective vertical tasks are common tasks given to participants, where they are asked to orient themselves or an object according to gravity. The results of these tasks help to understand the contribution of the different sensory cues to the estimation of our body orientation relative to gravity.

Three sensory cues play an essential role in gravity estimation: the vestibular, visual and somatosensory feedbacks. On one hand, as balance is one of its main functions, the influence of the vestibular system on verticality perception is quite straightforward. The movements of the otolithic membrane when the head is tilted induce the bending of the sensory hair cells, causing a sensation of imbalance. On the other hand, studies also showed the importance of visual feedback. Indeed, the presence of tilted or moving elements in the visual field tends to bias the perception of vertical. Finally, somatosensory information is the third feedback that influences verticality perception. For example, people tend to better align a glass of water with gravity than a metal bar. This also highlights that the context in which the subjective vertical task is performed has an influence on the accuracy of the result.

In virtual reality, subjective vertical tasks are easy to implement. We actually did implement one at the beginning of our master thesis. The goal was to align a rectangle with gravity. The VR allowed us to suppress the possible visual cues of vertical, as the background was black and the only seen object was the rectangle, controlled with a joystick. The subjects would be tilted with the rotating chair to study how they would estimate the direction of gravity only based on the vestibular feedback. No subject performed this experiment, but it could be a study topic!

This is an example where the object to align with gravity is controlled with an interfacing controller. However, as our system includes motion tracking cameras, it is also possible to make the subjects hold a real object and display its position and orientation in the virtual environment. This makes it possible to study the influence of the somatosensory feedback on the perception of the vertical. The influence of the visual feedback could also be studied by integrating tilted or moving elements in the subject's field of view.

2.8 Different environments

Some studies try to put participants into unusual situations. Indeed, we do not know exactly how they will react, and the results may help to find new underlying mechanisms of the human body. In this section, we present two experimental set-ups used to put the subjects in unfamiliar situations. With our interface, these set-ups could be combined with a VR headset.

2.8.1 Rotating chair

A rotating chair is a seat attached to a rotation axis, which can therefore be tilted to different orientations. The COSY lab is equipped with a rotating chair that can rotate on a transversal axis, i.e. forward or backward, at 360°.

In this recent paper written in 2021, L. Opsomer, F. Crevecœur, J-L. Thonnard, J. McIntyre and P. Lefèvre [22] performed an experiment with a rotating chair at two different orientations: right side up and upside down. Thirty-six healthy participants were given the task of performing vertical rhythmic arm movements with a manipulandum at a pace of 1Hz. The movement was delimited by 2 targets 32cm apart.

The results show that the coordination between the GF and the LF operates in an Earth-centred allocentric reference frame (reference with object-to-object relationship, independently of the participant's point of view [4]), where the gravity defines up and down. This coordination relies not only on sensory feedback but also on predictive mechanisms which are strong enough to still perform accurate predictions of the sensory feedback in an upside-down position.

However, the GF increases when the participant is in an upside-down position, due to the unfamiliar situation of the subject, even if the relationship GF-LF improved with practice [22].

2.8.2 Parabolic flight

Parabolic flights reproduce hypergravity and microgravity conditions in an aircraft by alternating upward and downward movements, like a parabola, as we can see in figure 2.6 [12]. Simulating the effects of free fall on the human body is a very effective way to understand how the human body reacts and adapts to space missions. Parabolic flights are useful to test the effects of short-term exposure to altered gravity, since the periods of exposure last approximately 30 seconds.

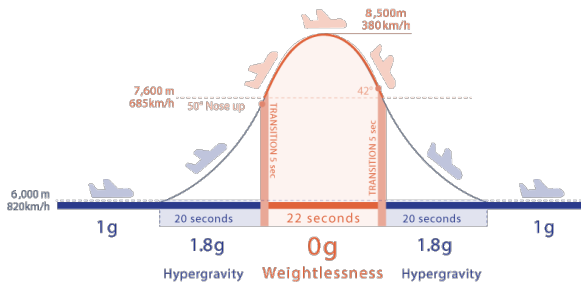


Figure 2.6: Parabolic flight [12]

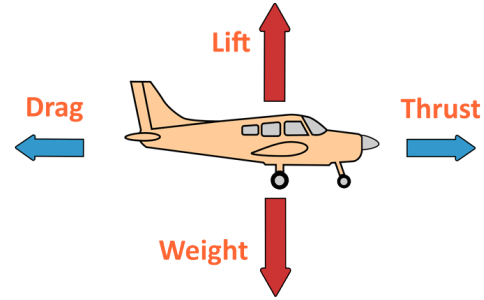


Figure 2.7: Forces acting on a plane [1]

Despite passengers perceiving the gravity to be zero, the only changing factor is the acceleration, the gravity is still present. During the up and down movement of the aircraft, the passengers perceive a force perpendicular to the aircraft's floor, due to the aircraft lift and thrust that produce the appropriate accelerations along the aircraft's vertical and longitudinal axis 2.7 [1].

The situation in parabolic flight is unfamiliar to most people, and is thus interesting to study. In 2005, O. White, J. McIntyre, A-S. Augurelle, and J-L. Thonnard analysed the coupling between GF and LF during cyclic vertical movement in parabolic flight (1g, 1.8g and 0g). The results highlighted that the internal model predicts the load force by taking into account the dynamic characteristics of the upper limb, the object and the environment [27].

2.9 Virtual Reality

Virtual reality is *a simulated experience that can be similar or completely different from the real world* [7]. For now and in our case, we use a VR headset that allows us to evolve in complete immersion in a universe.

Currently, VR is known for its video games but it can also be used for scientific needs. VR makes it possible to simplify experiments because changing the virtual environment is much simpler and less costly than changing the real environment. Moreover, it becomes easier to combine different devices, and thus generate sensory conflicts and study the body's response. In this section, we will review papers studying the sensorimotor system using VR, that can give ideas for future experiments.

2.9.1 Analysis of visual feedback

In their paper [14], F. M. Grover, S. M. Schwab, and M. A. Riley study the influence of visual feedback during the manipulation of objects using a virtual reality headset.

The subjects were asked to maintain an object inside a target moving along a tube, with visual feedback of the object and feedback on their performance. The study manipulated the visual gain, with three different values (see figure 2.8) under two conditions. For the first one, the target followed an unpredictable trajectory driven by a stochastic process (called the stochastic trajectory) while for the second one, the target followed a deterministic but random and unpredictable trajectory (called chaotic trajectory).

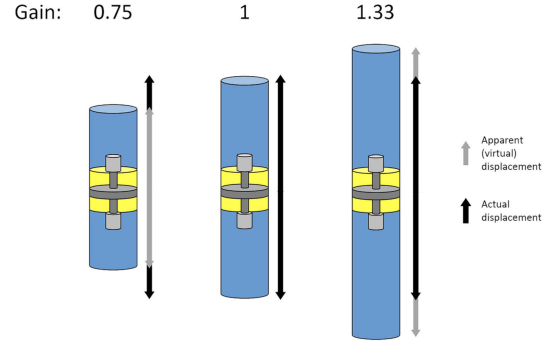


Figure 2.8: Gain manipulation [14]

This article concludes, following the results, that dynamic visual feedback impacts time-varying modulation of GF, GF control and GF-LF coupling. Indeed, GF-LF ratio is higher in the 1.33 gain condition than in the 0.75 gain condition. It confirms that an increase in visual gain means greater object displacement, velocity, and acceleration. From there, the subject will believe in a greater LF variation, and will therefore adapt its GF accordingly.

2.9.2 Influence of sensory and acoustic feedback

This study conducted by J. L. Maldonado Cañón, T. Kluss, and C. Zetzsche [18] investigated the learning and adaptation processes in motor control during a contact event. To do so, they studied the influence of sensory feedback on human movement behaviour in the case of a missing, noisy, or incomplete sensory feedback between a VE and a physical environment (PE).

The setup of the experiment is shown on the left of figure 2.9. It is composed of several elements :

- A mirror: reflects the image shown by the monitor and occludes the Phantom and the participant's hand
- A monitor: shows the VE
- A Phantom: a device that generates haptic stimuli, equipped with a stylus (able to move in the 3D space)
- Headphones: produce sound to mask the noise of the Phantom

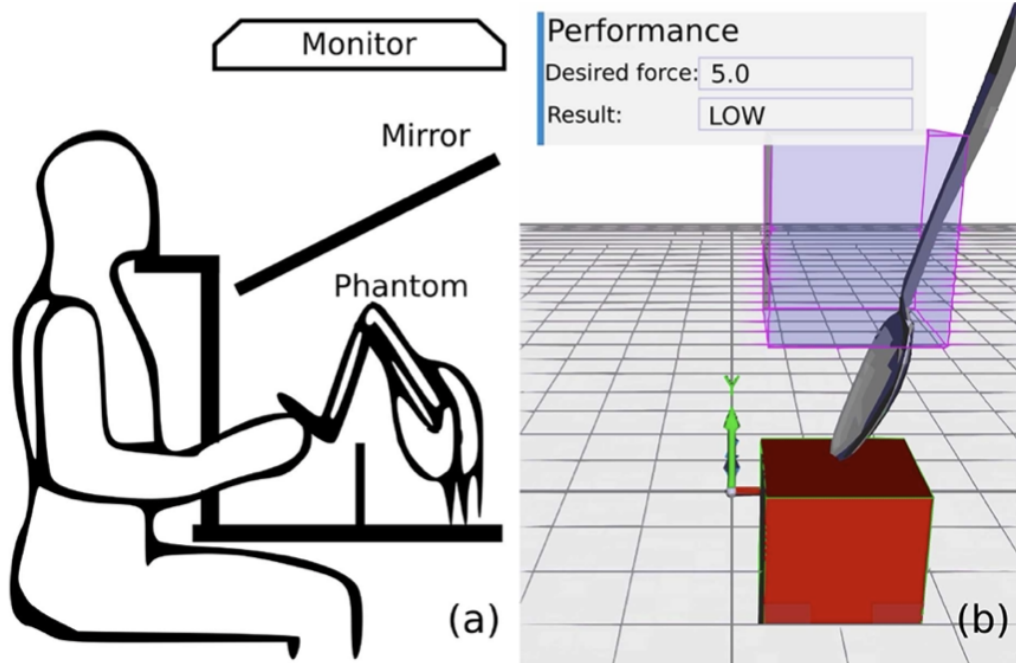


Figure 2.9: Experimental setup and 3D virtual environment [18]

The task given to the participants was to move the spoon (in the PE, it is the stylus) until it touches the contact box with a target force in the VE. At the end of each trial, the result ($score - target$) was given: low – good – high.

When the spoon touched the box, a sound and a small vibration adapted to the force of the impact were emitted in order to reproduce the same condition as an impact. Participants performed three blocks of 50 trials. The blocks differ in the sensory information that was delivered at the moment of contact :

1. Haptic condition (H): force was rendered
2. Acoustic condition (A): impact sound was generated
3. Haptic-acoustic condition (HA): force was rendered and audio was generated

Results suggest that participants were more frightened to try high velocities when haptic feedback was available, compared to acoustic feedback. Indeed, the contact at high velocity in H condition may be perceived as harsh while in A condition, the movement can be stopped smoothly.

This paper shows how motor control combines different sources of information to adapt its kinematic parameters, particularly the haptic and auditory feedbacks. It concludes that the auditory information is sufficient for the learning of the movement of a tool (here a pencil) toward an object with a specified contact force/velocity. However, in the presence of the haptic information, the subject learns faster and is more precise in his movements.

2.9.3 Influence of visual delay

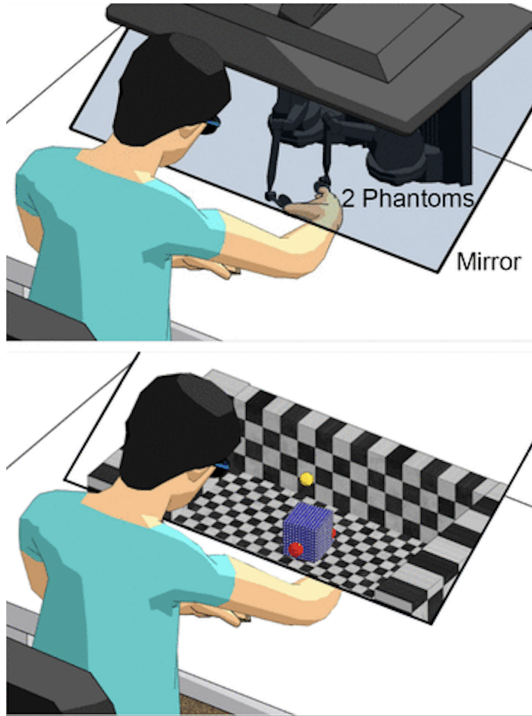


Figure 2.10: Experimental setup [26]

In their study [26], V. Van Polanen, R. Tibold, A. Nuruki, and M. Davare wanted to assess how object lifting changes force scaling and weight perception of an object. The setup of the experience is shown in figure 2.10. The mirror allowed the participants to see the VE. Thirty participants performed two tasks: lifting objects with and without visual delay, and they were asked to indicate which of both cubes felt heavier. The force feedback was applied by two Phantoms and was present in both conditions.

The results of the experiment showed significant differences between force profiles in delay and no-delay lifts. Visual delay led to a slower force generation and increased grip forces. Moreover, the experience showed that a cube lifted with delay is perceived as 18% heavier than without delay.

The paper concludes that the visual delay does not only alter force scaling, but also increases the perceived heaviness of lifted objects. It confirms the theory that visual cues play a major role in object manipulation, even if haptic information is, according to the article, sufficient to lift an object. Moreover, they found that visual and haptic feedback influence independently the force scaling.

2.10 Conclusion

It is with this context in mind that we designed our interface. All this research on the human body made us understand what will our interface be used for. Throughout this chapter, we have reviewed the key concepts necessary to understand what VR could bring to help the research in motor control. In addition to that, it gives ideas for future experiments that could be conducted to discover how the human body works.

Chapter 3

Materials

This chapter presents the materials that we used for this master thesis. We worked with a VR headset called Oculus Quest 2, within which we displayed virtual environments created with the software Vizard. We combined this software with a motion tracking system called Codamotion. Besides that, the laboratory is also equipped with a manipulandum and a rotating chair.

3.1 Oculus Quest 2

The VR headset we have been working with is the Oculus Quest 2 (or Meta Quest 2), produced and marketed by the company Meta. Its technical characteristics are summarised in table 3.1.

The Oculus Quest 2 comes with two controllers called Oculus Touch, one for the right hand and one for the left hand. They both have one joystick, 2 buttons and one "menu" button on their upper face, as well as two buttons on the back. It is possible to detect when a button is pressed, but also when a finger lay on it without pressing it.

We can connect the headset to a computer with the Oculus Link technology, whether via a cable or through Wi-Fi. However, as the connection in the lab is not fast enough to support the Air Link, we only used wired Oculus Link. Once connected with Oculus Link, the subject wearing the headset finds himself in the Link hub, and any software can be launched from the computer, including scripts for experiments.



Figure 3.1: The headset and its controllers [20]

Size	22.4cm x 45cm
Weight	503g
Resolution	1834 x 1920 pixels per eye
Frame rate	90fps
Degrees of freedom	6

Table 3.1: Oculus Quest 2 characteristics [21]

3.2 Vizard

In order to create virtual environments where the subjects will evolve and perform the experiments, we have been using Vizard, a software developed and marketed by WorldViz. On their website [30], it is described as a "comprehensive virtual reality software for researchers". Vizard is Python-based and has additional libraries to display virtual environments in the Oculus Quest 2.

Vizard offers the possibility to create custom 3D environments designed for experimentation and display them in a VR headset at a refreshing rate of 60Hz. In this perspective, it includes several features to control physics, objects or events either by the subject or the experimenter. Here is a non-exhaustive list of the most interesting features for the COSY lab experiments, some of them are detailed in section 6.2:

- Creating 3D objects and controlling their position, orientation, size, etc, including custom models taken from the internet or elsewhere.
- Controlling the position and orientation of the subject within the virtual environment.
- Creating plug-ins in C++ that can be used to interface with hardware devices.
- Creating proximity sensors and targets.
- Linking objects so that they share the same transformation matrix.
- Controlling the flow of an experiment with event triggering, tasks, timers, etc.
- Importing video and audio files.

From the list above, the most interesting feature for us is the software development kit (SDK)¹ provided by Vizard that allows us to create plug-ins in C++ that can be launched in Vizard and used to interact with hardware devices such as a motion tracking camera. It can also be used to create and/or access 3D models of objects, but we did not use it for that purpose.

¹"A software development kit (SDK) is a collection of software development tools in one installable package." Wikipedia, Software development kit. https://en.wikipedia.org/wiki/Software_development_kit

3.3 Codamotion cameras

The motion tracking cameras we have been working with are the CX1 CODA sensor units, produced and marketed by Codamotion Ltd [3]. Each unit is composed of three cameras, meaning that one unit is sufficient for monitoring 3D data. Those units are connected either to a Codamotion Active Hub or to a host PC connected to a Codamotion Mini Hub. In our case, we used a host computer connected to a Mini Hub.

A Codamotion RTNet Server is running on the host PC and collects data from the cameras. Once sent to the server, data can be retrieved either with a software called Odin or with an application developed with an SDK called RTNet, which is available in C++ or Java. For this project, we used the C++ version of the SDK with Visual Studio 2019.

3.3.1 CODA CX1 sensor unit

The CX1 unit can monitor up to 56 markers. Its sampling rate depends on the number of monitored markers and can be set from 1Hz to the limits defined in table 3.2. The maximum measurement time depends on the number of markers and on the sampling rate.

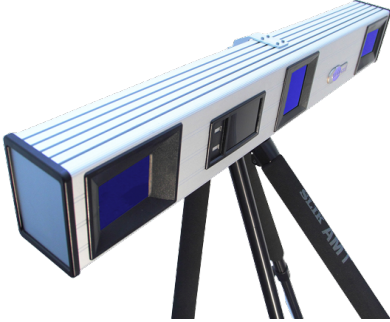


Figure 3.2: Coda CX1 unit [3]

A typical experiment in the lab requires a minimum of 4 markers and a sampling rate of 200Hz, this gives a maximum measurement time of 600s. In terms of resolution, the angular resolution of one camera is about 0.002 degrees, the lateral position resolution is approximately 0.05mm at 3m distance and the distance resolution is about 0.3mm. The markers can be detected within a volume going from 2m to 6m in front of the camera, with a width and height of approximately 1.6 times the distance.

Number of markers	Max sampling rate (Hz)
56	100
28	200
12	400
6	800

Table 3.2: Sampling rates of the CODA CX1 unit

3.3.2 Mini hub unit

As mentioned before, the CX1 cameras are connected to the host computer via a mini hub unit. Up to two CODA CX1 units can be connected to one mini hub and if so, they are automatically synchronised. The mini hub provides 12V DC power to the CX1 units and sends the monitored data to the host computer.

3.3.3 Drive boxes and LED markers

The Codamotion cameras monitor the positions of LED markers. Four of those markers can be connected to a drive box which provides power to the LED. The drive boxes are synchronised with the Coda scanner units and flash the markers alternatively within each frame. This way, marker identification is maintained.



Figure 3.3: Drive box and LED markers [3]

3.3.4 RTNet SDK

Real-Time Network SDK is the SDK provided by Codamotion to interact with the Codamotion system from the host computer. In this system, represented on figure 3.4, the RTNet Server links the mini hub and the monitoring software, whether Odin or another application developed with RTNet SDK.

In the SDK, the main programming interface to interact with the server is the class *RTNetClient*. It is used to connect to the server, initialise devices, launch and stop acquisition, etc. The connection between the client and the server is done using the TCP/IP protocol² over an Ethernet connection. The measurement data itself is received via a UDP socket³ represented by the RTNet class *DataStream*.

²"The Internet protocol suite, commonly known as TCP/IP, is the set of communication protocols used in the Internet and similar computer networks." Wikipedia, Internet protocol suite. https://en.wikipedia.org/wiki/Internet_protocol_suite

³UDP is one of the communication protocols of the Internet protocol suite. It is often used for time-sensitive applications as it is faster than other standard protocols. https://en.wikipedia.org/wiki/User_Datagram_Protocol

The RTNet server can have five different states:

- Shut down: The server is started but has not received any hardware configuration and has not initialised any device.
- Started: A hardware configuration has been chosen and the associated monitoring devices are initialised.
- Prepared: The system is ready for acquisition. Diode offset calibration has been done and parameters such as acquisition time or number of markers to monitor should have been set by the user.
- Continuous Acquisition: The system is streaming data continuously over one or more RTNet Data Stream. The data is not stored on the server so it is up to the receiving application to store it.
- Continuous Buffered Acquisition: The system is acquiring data but instead of being automatically sent over data streams, it is stored in buffers on the server for later retrieval. However, during the acquisition, the latest frame of data can be requested by the client.

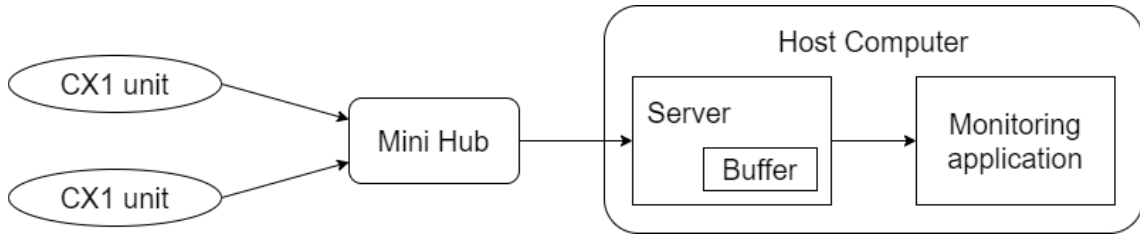


Figure 3.4: Codamotion architecture

3.4 Manipulandum

The manipulandum (see figure 2.2) is an object of 260g that records the grip force and the load force. It is equipped with a tridimensional accelerometer (Analog Devices, ref. ADXL78) and tridimensional force/torque sensors (Mini 40 F/T transducers, ATI Industrial Automation) [22]. It is held between the right thumb and index finger. The manipulandum is cables-connected to a computer, allowing the recording and visualisation of the forces in real-time.

3.5 Rotating chair

The lab is equipped with a rotating chair. This chair can rotate around the transverse axis, allowing subjects to sit at different orientations. They are attached to the chair with security bells, and foot straps are used for the feet [22].

Chapter 4

Interface

With this material and software programs at our disposal, we can have a clear view of how to implement our interface. As a reminder, the main challenge will be to combine RTNet SDK, used to retrieve the positions data, with Vizard, used to model the virtual environment.

As said in section 3.2, Vizard provides an SDK allowing us to create plug-ins¹ made to interact with hardware devices. How lucky we are, that is exactly what we need! We are thus going to create a plug-in in C++ with the Vizard SDK, that exploits the functionalities of the RTNet SDK. It will be launched from Vizard and should be built to control as many parameters as possible from Vizard, so that we do not need to modify the plug-in during the experiments. The figure 4.1 represents the main architecture of our system.

The interface between Vizard and the hardware is done through a *sensor* object. Such sensors can provide raw data such as position or orientation and can represent a controller or a headset for example. In our case, we are going to use sensors to represent the LED markers tracked by the CX1 units, or a set of these markers. Indeed, we may need to represent complex and orientable objects such as the manipulator. As the orientation will be calculated from the positions of the markers, we may also choose to retrieve the data of each marker separately and "combine" them in one object in Vizard. But this, in addition to being more complicated to implement and use, would duplicate the data transfer between the plug-in and Vizard.

To visualise the sensors provided by our plug-in, Vizard offers the possibility to link them to 3D shapes via the well-named *link* object. Via this link, the 3D shape will adopt the position and orientation of the associated sensor. And as those data are updated in real-time by the plug-in, the shape in the virtual environment will move according to the movements of the markers in the reality.

¹"In computing, a plug-in is a software component that adds a specific feature to an existing computer program." Wikipedia, Plug-in. [https://en.wikipedia.org/wiki/Plug-in_\(computing\)](https://en.wikipedia.org/wiki/Plug-in_(computing))

However, as the coordinate system used by the Codamotion cameras is not aligned with the one of the virtual environment, we will have to create a transformation matrix to project the positions of the markers at the right place in the VE. We can then apply this transformation matrix to the links in Vizard, so we do not have to manage this in our plug-in. If needed, we can also apply an additional transformation matrix to modify the visual feedback compared to the reality. Knowing that, the list below resumes the features that our plug-in should contain:

- Starting the cameras with the right parameters,
- Collecting positions data of any marker and sending them to Vizard in real-time,
- Creating groups of markers representing one object, collecting their position and orientation and sending them to Vizard in real-time,
- Saving the collected data in an external file for post-processing,
- Stopping the cameras at the end of the experiment.

On top of that, here are the mandatory actions that need to be done in Vizard:

- Connecting to the VR headset,
- Importing the plug-in and interfacing with it,
- Creating the transformation matrix between real and virtual environments,
- Creating the sensors needed and linking them to a 3D shape,
- Apply the transformation matrix to the links.

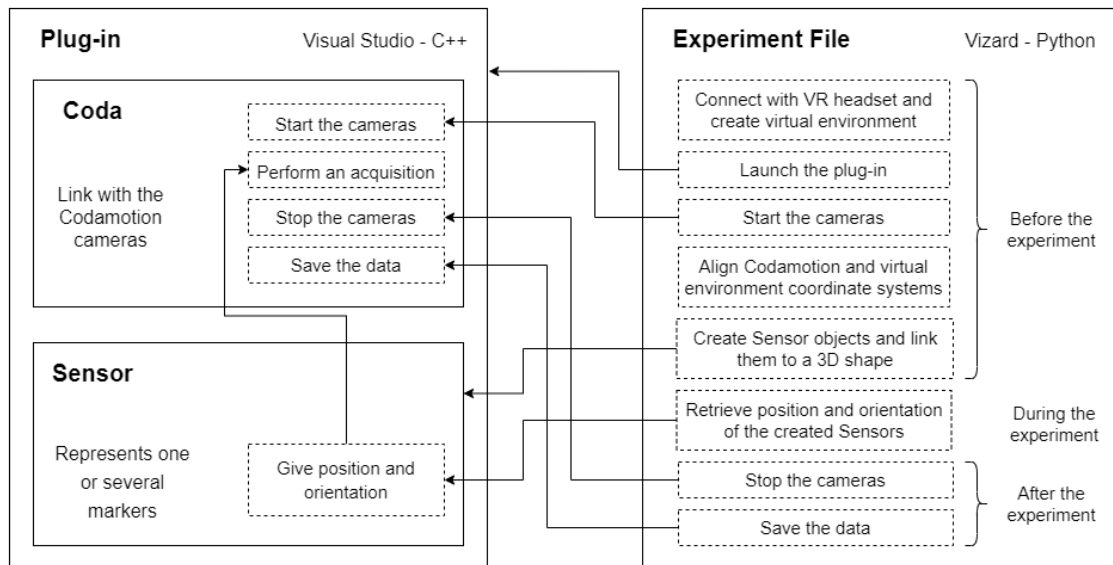


Figure 4.1: Architecture of the interface

Chapter 5

Implementation

The plug-in that we developed was written in C++ using Visual Studio, it can be launched directly in Vizard via a built-in command. Our goal was to build it in such a way that everything from starting to stopping the cameras can be controlled from Vizard. In this way, future experiments can be carried out without modifying the plug-in. We also tried to make it as complete as possible, thinking of as many functionalities that the researchers would need as possible.

The project is divided into two main parts. On one hand the plug-in, and on the other hand the python scripts written in Vizard, used to design experiments. In this chapter, we are going to present our interface in an iterative way, as we built it throughout the year. As we used the Vizard documentation [31], and particularly the topic about the Vizard SDK, to build our interface, some parts are very similar to it.

In this chapter and the following one, parts of code will be shown. For greater clarity, the background of the C++ code will be in blue: light blue for the header files and dark blue for the CPP files, and python code will be in green.

5.1 Before getting started

At this point, you may wonder what is a plug-in and why is it the best solution for us? A plug-in is a small software that can be executed by another software. It can be used to extend the functions of an application, to reduce its size or allow its customisation. In our case, our plug-in will allow us to interface with the Codamotion cameras from Vizard and collect their data in real-time under the form of objects (sensors) easily integrable to a virtual environment. A plug-in can have many file extensions, however, as Vizard imposes it, ours will be a .dle file.

The functions of the RTNet SDK are included in a dynamic-link library called *codaRTNetProtocol.dll*. Dynamic linking, in opposition to static linking, means that the functions, resources or data stored in a file can be used by one or several other executable files. In our case, it means that we will be able to call the functions of the RTNet SDK from Vizard.

Although dynamic linking is mandatory in the context of a plug-in, *codaRTNetProtocol.dll* should not be considered as such. If a plug-in can be seen as an extension to a specific software, a DLL is rather an interface that can be used by many software programs. Those concepts are illustrated in figure 5.1, in the context of our application.

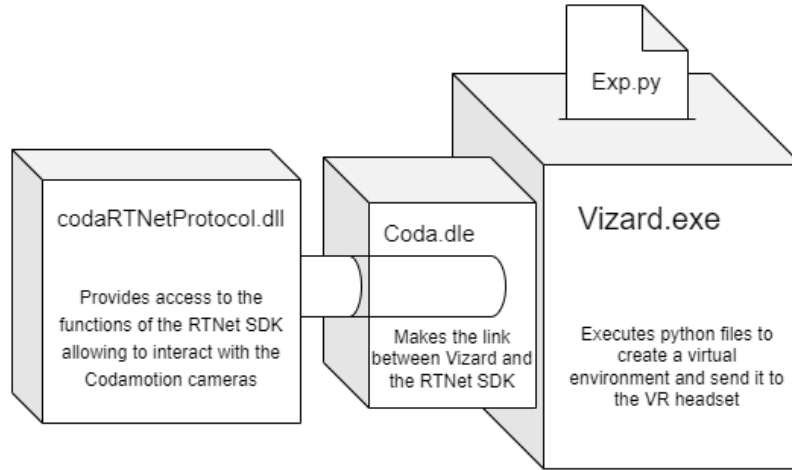


Figure 5.1: Architecture of the execution environment

The very first step in the development of an experiment is thus to create a project folder. This folder will contain the python experiment file that will be run with Vizard, and must also contain the *codaRTNetProtocol.dll* file, as it will be called by Vizard. This file can be found and copy-paste from the *CODARTNETSDK-Win64-2.00.03.00/bin* folder (or the latest version). Although we recommend it, our plug-in does not have to be in the same folder, as long as you mention the good file path when launching it from Vizard. Another solution is to put it in the *WorldViz/Vizard7/plugin-ins* folder with the other Vizard plug-ins.

5.2 The plug-in

This section will explain how to create a plug-in such as the one we developed. However, except if a big adaptation needs to be done, future users can use our project and bring the needed modifications.

Firstly, we need to create a Visual Studio project. Screen captures of the following procedure are available in appendix A. Open Visual Studio and select **Create a project**, then select **Windows Desktop Wizard** and click **Next**. Later on, the Desktop Wizard will give us the option to create a DLL project, and that is what we need because we want to create a plug-in. Click **Create** after giving a project name and a project location. Set the application type as **Dynamic Link Library** and select the **Empty Project** option, then click **Ok** to create the project. We can directly create our main class by selecting **Project** → **Add class**.

5.2.1 Project settings

Before implementing anything, we need to change a few settings as we will combine two different SDK. Screen captures of the project settings are available in appendix B. First, change **Debug** to **Release** in the main toolbar. Then open the project settings in **Project** → **Properties** and adjust the following ones:

- General: **Output Directory**: set the folder location where the plug-in will be output.
- Advanced: **Target File Extension**: set to .dle,
- C/C++:
 - General: **Additional Include Directories**: add the location of both the Vizard SDK include directory (Vizard7/bin/Include) and the RTNet SDK include directory (CODARTNETSDK-Win64-2.00.03.00/include, or the latest version),
 - Code Generation: **Runtime Library**: set to Multi-threaded DLL,
- Linker:
 - General:
 1. **Enable Incremental Linking**: Yes
 2. **Additional Library Directories**: add the location of the Vizard SDK library (Vizard7/bin/Libs) and both RTNet SDK libraries (CODARTNETSDK-Win64-2.00.03.00/lib and CODARTNETSDK-Win64-2.00.03.00/lib/win64-msvc-19, or the latest version)
 - Input: **Additional dependencies**: add codaRTNetProtocol.lib and codaRTNetProtocolCPP.lib,
 - Manifest File: **Generate Manifest**: No.

The exportation of our plug-in can be done by clicking **Build** → **Build solution**. The .dle file will be created in the folder mentioned as Output Directory. Other files will also be output, but the plug-in works as well without them.

5.2.2 The *Coda* class

The Vizard SDK provides a virtual base class called *viz::Extension*, the main class of our plug-in must derive from this class in order to make it compatible with Vizard.

Creating the plug-in

We named our main class *Coda* as it will be used to interact with the Codamotion cameras. We can see this class as the link between Vizard and the RTNet Client, used to communicate with the RTNet Server. This class derives from the *viz::Extension* class and the only method that has to be overwritten is *getName*, the newly created header file will thus look as follows:

```

1 #include <viz/Extension>
2
3 class Coda : public viz::Extension
4 {
5 public:
6     Coda(void);
7
8     virtual const char* getName() const { return "Coda"; }
9
10 protected:
11     virtual ~Coda(void);
12 };

```

Coda.h base

As this extension will be accessed from Vizard, we must create a DLL export function in the CPP file. As we will see later, this function will also be used to define the python interface for our plug-in, but for now, our CPP file looks like this:

```

1 #include "Coda.h"
2
3 extern "C" __declspec(dllexport) viz::Extension *
4     CreateVizardExtension(viz::Data & data);
5 viz::Extension* CreateVizardExtension(viz::Data& data)
6 {
7     return new Coda();
8 }
9
10 Coda::Coda(void)
11     : viz::Extension()
12 {
13 }
14
15 Coda::~Coda(void)
16 {
17 }

```

Coda.cpp base

The data exchange between the plug-in and Vizard is done thanks to the *viz::Data* class. This class is a container holding the values passed as arguments to the *viz::Extension* methods in Vizard. The fields of the *viz::Data* class are listed in the tab 5.1 and can be accessed with their string value. For example, the "command" value can be accessed with:

```

1 int command = data.getInt("command");

```

This class works as a container, it can contain any kind of information and its purpose is only determined by the function in which it is used.

Name	Type
command	int
message	string
x	float
y	float
z	float
w	float

Table 5.1: viz::Data fields

In order to interface with the Codamotion cameras, we created three functions:

```

1 int run(int conf, bool doAlign, int 0, int X, int Y);
2 vector<float> acquire(vector<int> markers);
3 void saveData(const char* filename);

```

Coda functions

Coda::run is used to start the cameras and optionally make the alignment, *Coda::acquire* is a function that gives the last monitored position of one or more given markers and *Coda::saveData* is used to save the data in a file after an acquisition.

We also added two class attributes: the RTNet client and the data stream. The client is the class that we will use to connect to the RTNet server and control it from our plug-in, and the data stream is where the monitored data will come from.

```

1 RTNetClient cl;
2 DataStream stream;

```

Coda::run - Starting the cameras

The *Coda::run* function will be used to start the cameras, at the very beginning of the acquisition. It takes a few arguments: firstly the camera configuration to use, then a boolean to indicate if we want to perform an alignment and if so, the markers to use for the alignment.

The first thing to do is to establish the connection with the Realtime Network server:

```

1 AutoDiscover discover;
2 this->cl.doAutoDiscoverServer(discover);
3
4 if (discover.dwNumServers == 0)
5 {
6     fprintf(stderr, "ERROR: no server found using auto-discovery");
7     return 1;
8 }
9
10 this->cl.connect(discover.server[0].ip, discover.server[0].port);

```

Coda::run (1)

RTNetClient::doAutoDiscoverServer will find every Codamotion Realtime Network servers available. If none is found, we raise an error, otherwise, we connect to the first server available. The server is now in the *shut down* state (see section 3.3.4).

After connecting to the server, we have to find the different configurations available, once again if no configuration is found, we raise an error.

```
1 HWConfigEnum configs;  
2 this->cl.enumerateHWConfig(configs);  
3  
4 if (configs.dwNumConfig == 0)  
5 {  
6     fprintf(stderr, "ERROR: no hardware configuration available");  
7     return 1;  
8 }
```

Coda::run (2)

Here, we talk about hardware configuration: whether we use a CodaHub or a MiniHub, how many cameras we use, etc. These configurations need to be previously defined, but it has already been done by the lab members. At the time we developed this interface, the different configurations available were the following:

- 0: Use two CX1 units,
- 1: Use three CX1 units,
- 2: Use four CX1 units,
- 3: Use only the CX1 unit connected to port 1,
- 4: Use only the CX1 unit connected to port 2,
- 5: Use two CX1 units in diagnostic mode,
- 6: Use two CX1 units in low correction mode.

In order to control this parameter from Vizard, we set it as a parameter to the *Coda::run* function named *conf*. However, the three configurations that we mainly work with are configurations 0, 3 and 4.

Now that the server is launched, we can start the system with the desired configuration. But before that, as they may remain some settings from previous experiments, we have to make sure that the system is fully shut down before starting it again. If we do not do that, an error can occur between two executions, for instance, if we change the configuration. After that, the server will be in the *started state*.

```
1 // ensure that the system is shut down  
2 cl.stopSystem();  
3 codanet_sleep(50); // wait a bit  
4  
5 // starts the system with the good configuration  
6 cl.startSystem(configs.config[conf].dwAddressHandle);
```

Coda::run (3)

After starting the system, we have to indicate a limit to the number of frames we want to record. For that we use *RTNetClient::setAcqMaxTicks*, with *DEVICEID_CX1* as first argument because we use CX1 units. In our case, we can simply use *CODANET_ACQ_UNLIMITED* as second argument to indicate that we want to record indefinitely, or until we tell the system to stop.

```
1 // record indefinitely or until we tell the system to stop
2 cl.setAcqMaxTicks(DEVICEID_CX1, CODANET_ACQ_UNLIMITED);
```

Coda::run (4)

Aligning the cameras

Now, we can perform an alignment if required. Performing an alignment means that we indicate to the Codamotion system which markers will be used to define its internal coordinate system. This operation requires 5 markers: one for the origin, two markers to indicate the origin and direction of the X-axis, and two markers for the origin and direction of the Y-axis¹. However, as the data exchange between our plug-in and Vizard is limited by the *viz::Data* container, we can reduce the number of markers to 3. This means that we use the same marker for the origin of the coordinate system, the origin of the X-axis and the origin of the Y-axis. These three markers should thus be placed in a right-angle configuration, at least 50cm away from each other.

Concretely, we firstly create an alignment request with *DeviceOptionsAlignment::align*, then we perform the alignment procedure with *RTNetClient::setDeviceOptions*. After that, we retrieve the alignment's information with *RTNetClient::getDeviceOptions* and we verify if the alignment has been executed correctly, if not we send an error and exit.

```
1 //perform alignment if requested
2 if (doAlign == true)
3 {
4     // alignment request structure
5     DeviceOptionsAlignment align(0, 0, X, 0, Y);
6
7     // perform alignment
8     // this will cause the system to look for alignment markers
9     // - they must be in position by this time
10    cl.setDeviceOptions(align);
11
12    // retrieve information
13    DeviceInfoAlignment info;
14    cl.getDeviceInfo(info);
15
16    // print alignment diagnostics
17    DWORD marker_id_array[5] = { 0, 0, X, 0, Y };
18    print_alignment_status(marker_id_array, info);
```

¹It is actually not required for the Y-axis to be perpendicular to the X-axis. It is used by the Codamotion cameras to define the XY plan, and the Y-axis is defined based on this plan and the direction of the X-axis. However, for simplification, we will consider that this axis has the same direction as the Y-axis.


```

19 // exit early if alignment failed
20 if (info.dev.dwStatus != 0)
21 {
22     fprintf(stderr, "Exiting due to alignment errors\n");
23     codanet_sleep(500);
24     exit(-1);
25 }
26 }

```

Coda::run (5)

As performing the alignment is a particular operation, that can also fail if the cameras or markers are not placed correctly, we strongly recommend not to include it directly in the experiment script, but rather creating another script specifically for the alignment. That way, if the alignment fails, we will avoid further errors because the script tries to launch the VE with the wrong alignment. Moreover, *print_alignment_status* will indicate the reason why the alignment failed, for example if a marker is occluded, so it can be directly fixed.

It also means that when doing a real acquisition, we are not going to use the O, X and Y arguments of the *Coda::run* function. We can thus take advantage of this to use them for other purposes. You may wonder why can't we simply create other arguments? Remember that every variable shared between Vizard and our plug-in should fit in a *viz::Data* object (see table 5.1), so places are limited.

Hence, we are going to use the O argument to indicate how many markers we want to monitor. *DeviceOptionsCodaMaxMarkers* allows restricting the number of markers and greatly fasten the retrieving of the data once we want to store them (see section 5.2.2). The only disadvantage is that it restricts the markers in numerical order, i.e., if we limit the number of markers to 4, we can only use markers 0, 1, 2 and 3. That is why we make sure that using this option is not mandatory, you can avoid it by setting O to 0 (with *doAlign=false*).

```

1 if (doAlign == false && O != 0)
2 {
3     codaRTNet::DeviceOptionsCodaMaxMarkers limit_marker_number(0);
4     cl.setDeviceOptions(limit_marker_number);
5 }

```

Coda::run (6)

The last few steps to go before the system is ready to acquire data are:

```

1 this->cl.prepareForAcq();
2 this->cl.createDataStream(this->stream, 7000);
3
4 codanet_sleep(50);
5 this->cl.startAcqContinuousBuffered();

```

Coda::run (7)

The function *RTNetClient::prepareForAcq* performs any pre-acquisition configuration and enters the system in the *prepared* state. *RTNetClient::createDataStream* creates a UDP socket on the client and connects it to the server for streaming of data, 7000 is an arbitrary ID for the socket. The *codanet_sleep(50)* is there to ensure that the socket is up before starting acquisition. Finally, *RTNetClient::startAcqContinuousBuffered()* sets the server into *continuous buffered acquisition* state, meaning that the acquired data are put in a server-side buffer for later saving.

Coda::acquire - Performing an acquisition

While the data is being continuously stored in the buffer, we must also send it to Vizard so that it can be used in the virtual environment. However, while a standard acquisition rate needed for post-processing is 200Hz, the refresh rate of Vizard is only 60Hz, it is thus not necessary to send every monitored data in the virtual environment.

Taking that into account, we had to choose between two methodologies: either we send every monitored frame to Vizard and the filtering is done within Vizard, or Vizard itself requests an acquisition to the plug-in when it is needed. For several reasons, we kept the second option. First of all, Vizard SDK already implements framewise data requests. Secondly, as said before not every acquisition is needed in Vizard, and limiting the number of data sent will lighten the load on the data stream between the plug-in and Vizard. Finally, implementing the data sharing this way will facilitate the implementation of the *Sensor* class detailed in section 5.2.3.

We have thus created the function *Coda::acquire* that takes a list of marker numbers as an argument and returns the position values of these markers. We use a vector of floats to store the X Y Z positions in the marker order of the argument, meaning that the X position of markers[0] is stored in pos[0], its Y position in pos[1], etc.

After creating this positions vector, we can retrieve the last frame of data from the buffer using the function *RTNetClient::monitorAcqBuffer*:

```
1 vector<float> pos(3 * size(markers));
2
3 this->cl.monitorAcqBuffer(); // request latest frame
```

Coda::acquire (1)

This function will send the latest frame to the data stream, we can retrieve it using:

```
1 RTNetworkPacket packet;
2 this->stream.receivePacket(packet, 1000);
```

Coda::acquire (2)

The second argument of *DataStream::receivePacket* indicates the delay in microseconds to wait for a new packet, this operation should not take longer than 1ms.

Finally, after verifying that the packet is readable, we can decode it and save the position in our vector. Here, "decode" actually refers to the process of finding the offset of every marker value within the data packet. After that, we can use a series of get methods to retrieve the data of each marker. We have its position as well as its validity and its intensity. Its intensity refers to the light intensity of the detected LED marker, on a scale from 0 to 255. *PacketDecode3DResultExt::getIntensity* returns an intensity value for each camera (3 per CX1 unit) and if these values are under 10, the marker is considered undetected.

The function *PacketDecode3DResultExt::getValid* returns a non-zero flag if the marker was detected by at least one measurement unit, 0 otherwise. When storing the position, if this validity flag is 0, we store a 0 position value for this marker. This way, when retrieving its position, we know that we can't use this marker. Indeed, as the precision of the cameras is in the order of the millimetre, a [0.0, 0.0, 0.0] position for a marker is very unlikely to happen. Moreover, this position should have already been taken by the origin marker used for the alignment.

```

1  if (packet.verifyChecksum())
2  {
3      // decoder objects
4      PacketDecode3DResultExt decode3D;    // 3D measurements (CX1)
5
6      // decode results
7      if (decode3D.decode(packet))
8      {
9          for (int i = 0; i < this->nbmarkers; i++) {
10             float* newpos = decode3D.getPosition(markers[i]);
11             BYTE valid = decode3D.getValid(markers[i]);
12             BYTE* intensity = decode3D.getIntensity(markers[i]);
13
14             if ((int)valid != 0) {
15                 pos[3 * i] = newpos[0]/1000;
16                 pos[3 * i + 1] = newpos[1]/1000;
17                 pos[3 * i + 2] = -newpos[2]/1000;
18             }
19             else {
20                 pos[3 * i] = 0;
21                 pos[3 * i + 1] = 0;
22                 pos[3 * i + 2] = 0;
23             }
24         }
25     }
26 }
27 else
28 {
29     fprintf(stderr, "checksum failed\n");
30 }
31 return pos;

```

Coda::acquire (3)

You can notice that we divide the position by 1000. It is due to a difference in scale between Codamotion and Vizard. While Codamotion gives us the positions in millimetres, Vizard asks them in meters, so we decided to convert them to the Vizard's scale.

You may also have noticed that we inverted the sign of the z-axis, this is once again due to a difference between the Codamotion and Vizard's conventions: Codamotion uses a right-handed base for its vector orientation and Vizard uses a left-handed base, once again we decided to keep the Vizard's convention so that it will be much easier to manipulate the position data within Vizard later on.

Coda::saveData - Saving the data

At the end of an acquisition, one could like to save the recorded data in a file to perform post-processing on it. In this regard, we created the function *Coda::saveData*, which takes as argument the name of the file in which the data will be saved.

During the acquisition, the data is stored in a buffer, all we need to do is recover every packet from this server, decode them and write their content in a file. We begin by mentioning that the devices that we are working with are CX1 units, opening the file and retrieving the total number of frames with *RTNetClient::getAcqBufferNumPackets*.

```
1 WORD device = DEVICEID_CX1;
2
3 FILE* file;
4 if (fopen_s(&file, filename, "wt") == 0)
5 {
6     // despite the name, this actually returns the number of frames
7     DWORD numframes = this->cl.getAcqBufferNumPackets(device);
```

Coda::saveData (1)

After that, we can loop through every frame (Codamotion frames, not Vizard) and request the packet at each one of them with *RTNetClient::requestAcqBufferPacket*. This function will send the requested packet to every open data stream, meaning that we can retrieve it through our previously opened stream. *DataStream::receivePacket* is used to receive the last packet sent to this stream. If there is no packet available, which should normally not happen, it will wait for up to the time mentioned as timeout in microseconds. 1ms is sufficient here, a too big waiting time could considerably slow down the whole saving process.

```

1  for (DWORD iframe = 0; iframe < numframes; iframe++)
2  {
3      // request packet for this frame
4      cl.requestAcqBufferPacket(device, iframe);
5      DWORD timeout = 1000;
6
7      // retrieve until timeout
8      codaRTNet::RTNetworkPacket packet;
9      stream.receivePacket(packet, timeout);
10
11     if (!packet.verifyChecksum())
12         printf("checksum failed");

```

Coda::saveData (2)

We can finally decode the packet and write its components in our file.

```

1      // decoder objects
2      PacketDecode3DResultExt decode3D; // 3D measurements (CX1)
3
4      // decode & print results
5      if (decode3D.decode(packet))
6      {
7          // find number of marker positions available
8          DWORD nmarkers = decode3D.getNumMarkers();
9
10         fprintf(file, "FRAME: %05u ", decode3D.getTick());
11
12         // loop through the markers
13         for (int i = 0; i < this->nmarkers; i++)
14         {
15             DWORD imarker = this->markers[i];
16             float* pos = decode3D.getPosition(imarker);
17             BYTE valid = decode3D.getValid(imarker);
18             BYTE* intensity = decode3D.getIntensity(imarker);
19
20             fprintf(file, "MARKER: %02u COORDS: %05.1f %05.1f
21                        %05.1f VALID: %03d I: ", (imarker + 1), pos[0],
22                        pos[1], pos[2], (int)valid);
23             for (DWORD iI = 0; iI < decode3D.
24                 getNumCamerasPerMarker(); iI++)
25                 fprintf(file, "%03d ", (int)intensity[iI]);
26         }
27         fprintf(file, "\n");
28     }
29     fclose(file);
30 }

```

Coda::saveData (3)

Here, we save the data of one frame per line. In each line, we register the frame number, and the four markers with their numbers, its position as X Y Z coordinates, its validity and its intensity. Be aware that the position stored here is the one monitored by the cameras, so the one in the Codamotion coordinate system. One could change this section of code if he needs to store other data or in another format.

Interacting with our plug-in from Vizard

To interact with our plug-in, Vizard SDK includes a few built-in functions, of which the function *command* is the one we are going to use. This method is called when the *command* function of the python extension object is called within Vizard, and we can give arguments that can be retrieved in a *viz::Data* container (see Table 5.1). In our case, we will use the *command* function for three different actions: start the cameras for an acquisition, start the cameras for an alignment, and stop the cameras and save the data. The method will thus look like this:

```
1 void Coda::command(viz::Data& data) {
2     int command = data.getInt("command");
3     switch (command) {
4         case 1:
5         {
6             //run without alignment, "y" is the number of monitored
6             //markers
7             this->run(data.getFloat("x"), false, data.getFloat("y"),
7             //0, 0);
8             break;
9         }
10        case 2:
11        {
12            //run with alignment, "y", "z" and "w" are alignment
12            //markers
13            this->run(data.getFloat("x"), true, data.getFloat("y"),
13            //data.getFloat("z"), data.getFloat("w"));
14            break;
15        }
16        case 3:
17        {
18            //stop acquisition if it is still in progress and save
18            //data
19            if (cl.isAcqInProgress())
20                cl.stopAcq();
21            const char* file = data.getString("message");
22            this->saveData(file);
23            break;
24        }
25    }
26 }
```

Coda::command

As you can see, the instruction is given through the int field *command*. Hence, command 1 correspond to starting cameras for an acquisition, we thus call the *Coda::run* function with the *conf* argument given via the "x" field of the data object and the max markers argument via the "y" field, and we set *doAlign* to false (see section 5.2.2).

If we want to perform an alignment, we can use the command 2, it will also call the *Coda::run* function but with *doAlign* set to true, and thus we give the O, X and Y arguments for the alignment via the "y", "w" and "z" fields of the data structure.

Finally, command 3 is made for stopping the acquisition and saving the data. We verify if an acquisition is still in progress and if so, we stop it. Then we call *saveData* with the name of the file given via the "message" section of the data structure, if no filename is given, no data is saved (see section 5.2.2).

Once that is done, the interaction from Vizard can be done that way:

```
1 ext = viz.add('Coda.dle') # Launch the plug-in
2
3 # Start the acquisition with the first camera for 4 markers
4 ext.command(1, x=3, y=4)
5
6 # stop the acquisition and store the monitored positions in test.
7   txt
8 ext.command(3, "test.txt")
```

In the *command* function, we don't need to specify the names of the arguments as long as we give them in the order of the *viz::Data* container. If one of them is skipped because not used, we need to specify the following, even though it can be avoided by giving values such as an empty string to the arguments not used in the plug-in, they will simply not be retrieved.

5.2.3 The *Sensor* class

Once we have created our main class, we can create a sensor class. This class will be used to represent a marker or a set of markers within Vizard. Using such a class presents some advantages as many aspects of the sensor are automated by the Vizard SDK. For example, if we overwrite the functions *getPosition* or *getOrientation* from the *viz::ExtensionSensor* class, they will be automatically called at each frame and thus the position and the orientation of the associated Sensor object in Vizard will be updated in the virtual environment.

Creating the sensor

Our sensor will represent at least one marker, and it also needs to access the *Coda::acquire* function. The strategy adopted is that we primarily provide one marker, and then we can add the other markers via the *command* function.

We are also going to use *viz::ExtensionSensor::update*, a built-in function that will be called at the beginning of every Vizard frame. It is in this function that we will call *Coda::acquire* and store the position and orientation values in protected variables. After that, they will be sent to Vizard via the *viz::ExtensionSensor::getPosition* and *viz::ExtensionSensor::getOrientation* functions. The main advantage of this strategy is that we only have to call *Coda::acquire* once per frame.

We have thus a header file that first looks like this:

```
1 #include <viz/Extension>
2 #include <viz/ExtensionSensor>
3 #include "Coda.h"
4
5 class Coda;
6
7 class Sensor : public viz::ExtensionSensor
8 {
9 public:
10
11     Sensor(int m, Coda* c);
12
13     vector<int> markers;
14     int nbmarkers;
15
16     Coda* coda;
17
18     virtual const char* getName() const { return "Sensor"; }
19     virtual void command(viz::Data& data);
20     virtual void update(const viz::Event& e);
21     virtual bool getPosition(float* pos, int flag);
22     virtual bool getOrientation(float* ori, int flag);
23     virtual int getSourceType() const { return VIZ_LINK_POS |
24         VIZ_LINK_ORI; }
25
26 protected:
27     virtual ~Sensor(void);
28     float m_pos[3];
29     float m_ori[4];
30 };
```

Sensor.h base

The function *viz::ExtensionSensor::getSourceType* is used by Vizard to know which information is provided by the sensor, it can either be position, orientation, scale or a combination of them. In our case, we only need the position and the orientation values.

In our CPP file, the constructor of the class will look as follows:

```
1 #include "Sensor.h"
2
3 Sensor::Sensor(int m, Coda* c)
4     : viz::ExtensionSensor()
5 {
6     this->coda = c;
7     this->nbmarkers = 1;
8     this->coda->nbmarkers += 1;
9     this->markers.push_back(m);
10 }
11
12 Sensor::~Sensor(void)
13 {
14 }
```

Sensor.cpp base

As said previously, we are going to use the *command* method to implement the possibility to include several markers in our sensor:

```

1 void Sensor::command(viz::Data& data) {
2
3     int command = data.getInt("command");
4     switch (command)
5     {
6         case 1:
7         {
8             // add a marker to the set
9             int newmarker = (int)data.getFloat("x");
10            this->coda->markers.push_back(newmarker);
11            this->coda->nbmarkers += 1;
12
13            this->markers.push_back(newmarker);
14            this->nbmarkers += 1;
15            break;
16        }
17    }
18 }

```

Sensor::command (1)

The next question to ask ourselves is how are we going to call that sensor class from Vizard? We need to implement the python command that we will use to create a sensor object, this is done within the *CreateVizardExtension* method in our main class, via the *Python_SetExtensionCode* function:

```

1 viz::Extension* CreateVizardExtension(viz::Data& data)
2 {
3     const char* code =
4         "class _Coda(viz.VizExtension):\n"
5         "    def addMySensor(self, marker):\n"
6         "        return _Sensor(self.addSensor(marker).id)\n"
7         "
8         // Define sensor interface
9         "class _Sensor(viz.VizExtensionSensor):\n"
10        "    def getName(self):\n"
11        "        return \"Sensor\"\n"
12
13        Python_SetExtensionCode(code);
14
15        return new Coda();
16 }

```

Coda::CreateVizardExtension

Within the python interface, we defined a function *addMySensor* in the *Coda* class to create an instance of our *Sensor* class. This function will itself call the *addSensor* built-in function in python, that call a built-in function in C++ called *viz::Extension::createSensor*, that we thus need to implement in our *Coda* class:

```

1 viz::Referenced* Coda::createSensor(viz::Data& data) {
2
3     //At first only one marker is given, the other one will be
4     given with the command function
5     int sensNum = data.getInt("command");
6     this->markers.push_back(sensNum);
7     return new Sensor(sensNum, this);
8 }

```

Coda::createSensor

Thanks to this, we can now create a sensor for our markers in Vizard via the *addMySensor* function:

```

1 sensor = ext.addMySensor(1) #create a sensor for the marker 1
2 sensor.command(1, x=2) #add the marker 2 to the set of markers

```

Retrieving the position

The sensor as it is now does not provide the position of the markers. We can have two cases, either it represents only one marker, and we only need the position of this marker, or it represents a set of markers, and we want to know the centroid of all these markers.

In order to calculate the position of the sensor, we are going to use the built-in function *viz::ExtensionSensor::update* which will be called at the beginning of every frame. In this function, we are going to call *Coda::acquire* for the marker set, calculate its centroid and store the resulting position in a protected variable of the class.

```

1 void Sensor::update(const viz::Event& e) {
2
3     // mean position of all the markers
4     float posx = 0;
5     float posy = 0;
6     float posz = 0;
7     int k = 0; // nb of markers in the computation
8
9     // retrieve the positions of all the markers at the latest
10    frame
11    std::vector<float> newpos(3*(this->nbmarkers));
12    newpos = (*coda).acquire(this->markers);
13
14    for (int i = 0; i < this->nbmarkers; i++)
15    {
16        // acquire return a position of 0 if the marker is occluded
17        // a marker should never be at the exact [0, 0, 0] position
18        as it is the position of the origin marker of the coda
19        alignment
20
21        if (newpos[3*i] != 0 && newpos[3*i + 1] != 0 && newpos[3*i
22            + 2] != 0)
23        {

```

```

20         k += 1; //increment number of markers
21         // add position of this marker to the mean position
22         posx += newpos[3*i];
23         posy += newpos[3*i+1];
24         posz += newpos[3*i+2];
25     }
26 }
27
28 // update position
29 m_pos[0] = posx / k;
30 m_pos[1] = posy / k;
31 m_pos[2] = posz / k;
32 }

```

Sensor::update (1)

You may have noticed the *if* condition, its purpose is to detect whenever the recorded position of a specific marker is equal to zero, which is caused by an occlusion of this marker as we saw in section 5.2.2. In that case, the position of this marker is not added to the centroid computation. This prevents the object associated with the sensor to jump suddenly in the virtual environment towards the origin of the 3D space. Instead of that, it will only deviate a bit from its true position, which is much less disturbing for the subject. Of course, the real solution is to make sure that every marker is visible by at least one camera at any time of the experiment.

Now that we have computed the position of the sensor, we have to send it to Vizard using the dedicated function *ExtensionSensor::getPosition*. We simply set the position in the *pos* argument, the *flag* parameter is optional and will not be used in our case.

```

1 bool Sensor::getPosition(float* pos, int flag) {
2     pos[0] = m_pos[0];
3     pos[1] = m_pos[1];
4     pos[2] = m_pos[2];
5     return true;
6 }

```

Sensor::getPosition

Retrieving the orientation

If we have our sensor representing a set of markers, we may also want to use its orientation in Vizard. We will compute this orientation from the positions of the markers, but first, we need to indicate which markers will be used as axis marks. For that to be controllable from Vizard, we will use once again the *command* function. Hence, we add to our header file a new variable to store the marker numbers and a boolean to verify if axes have already been given or not:

```

1 int axes[4] = { 0 };
2 bool isAxes = false;

```

Sensor.h new variables

And add the command to provide the marker numbers:

```
1 case 2:
2 {
3     // add an axis for sensor orientation
4     int ox = (int)data.getFloat("x");
5     int x = (int)data.getFloat("y");
6     int oy = (int)data.getFloat("z");
7     int y = (int)data.getFloat("w");
8     if (!isAxes) {
9         this->axes[0] = ox;
10        this->axes[1] = x;
11        this->axes[2] = oy;
12        this->axes[3] = y;
13        this->isAxes = true;
14    }
15    else {
16        fprintf(stderr, "You can not define more than one axis
17            system.");
18    }
19    break;
20 }
```

Sensor::command (2)

As we want at least 2 axes, we store 4 marker numbers, the origin and direction of the X axis, and the origin and direction of the Y axis.

As said earlier, we will compute the orientation in the *Sensor::update* function, so we modify our *for* loop to store the positions that we will need for these computations. First of all, we initialise the position arrays at the beginning of the function.

```
1 float posX[3] = { 0 };
2 float posOX[3] = { 0 };
3 float posY[3] = { 0 };
4 float posOY[3] = { 0 };
```

Sensor::update (2)

Then we add an *if* condition in the *for* loop to store the right position values in these arrays. The example below is for the posOX array corresponding to the axes[0] marker, but we will have four of them.

```
1 if (this->markers[i] == axes[0])
2 {
3     posOX[0] = newpos[3*i];
4     posOX[1] = newpos[3 * i + 1];
5     posOX[2] = newpos[3 * i + 2];
6 }
```

Sensor::update (3)

After the *for* loop, we can effectively compute the orientation using these positions. As Vizard imposes that we provide the orientation as a quaternion², we used the Irrlicht 3D Engine SDK [13] that provides useful tools to work with 3D vectors. Hence, we firstly create our vectors:

```

1 irr::core::vector3df vecX(posX[0] - pos0X[0], posX[1] - pos0X[1],
  posX[2] - pos0X[2]);
2 irr::core::vector3df vecY(posY[0] - pos0Y[0], posY[1] - pos0Y[1],
  posY[2] - pos0Y[2]);
3
4 vecX.normalize();
5 vecY.normalize();
6 vector3df vecZ = vecX.crossProduct(vecY);

```

Sensor::update (4)

After that, we can construct the orientation matrix of our object, transform it into a quaternion and store these values in our protected variables.

```

1 // creates orientation matrix based on the vectors
2 float mat[16] = { vecX.X, vecX.Y, vecX.Z, 0, vecY.X, vecY.Y, vecY.Z
  , 0, vecZ.X, vecZ.Y, vecZ.Z, 0, 0, 0, 0, 1 };
3 matrix4 m;
4 m.setM(mat);
5
6 // retrieve quaternion form of the orientation from matrix
7 irr::core::quaternion quat(m);
8 // update orientation
9 m_ori[0] = quat.X;
10 m_ori[1] = quat.Y;
11 m_ori[2] = quat.Z;
12 m_ori[3] = quat.W;

```

Sensor::update (5)

For more precision, we decided to give the possibility to add a second axis set for the object. If given, we would compute the mean of the orientation vectors. We implemented it by simply duplicating every part of the above code. *Sensor::command* can be adapted by duplicating the axes flag so that we can give two axes sets:

```

1 int axes[4] = { 0 };
2 int axes2[4] = { 0 };
3 bool firstAxes = false;
4 bool secondAxes = false;

```

Sensor.h duplication of the variables

²Quaternions are a system used in 3D rotations <https://en.wikipedia.org/wiki/Quaternion>.

```

1 case 2:
2 {
3     int ox = (int)data.getFloat("x");
4     int x = (int)data.getFloat("y");
5     int oy = (int)data.getFloat("z");
6     int y = (int)data.getFloat("w");
7     if (!firstAxes) {
8         this->axes[0] = ox;
9         this->axes[1] = x;
10        this->axes[2] = oy;
11        this->axes[3] = y;
12        this->firstAxes = true;
13    }
14    else if (!secondAxes) {
15        this->axes2[0] = ox;
16        this->axes2[1] = x;
17        this->axes2[2] = oy;
18        this->axes2[3] = y;
19        this->secondAxes = true;
20    }
21    break;
22 }

```

Sensor::command (2 bis)

In the *Sensor::update* function, we add four other arrays at the beginning and four other if conditions in the *for* loop. Finally, we duplicate the vectors computed from those positions and before constructing the orientation matrix, we combine the two sets of vectors as follows:

```

1 if (secondAxes)
2 {
3     vecX = vecX.operator+(vecX2);
4     vecX = vecX.operator/(2);
5     vecY = vecY.operator+(vecY2);
6     vecY = vecY.operator/(2);
7     vecZ = vecZ.operator+(vecZ2);
8     vecZ = vecZ.operator/(2);
9 }

```

Sensor::update (6)

That way, if the second command has been called two times and a second axis mark has been given, we combine them, otherwise we only use the first system of axes. Finally, as for the position, we have to send this information to Vizard via the *Sensor::getOrientation* function:

```

1 bool Sensor::getOrientation(float* ori, int flag) {
2     ori[0] = m_ori[0];
3     ori[1] = m_ori[1];
4     ori[2] = m_ori[2];
5     ori[3] = m_ori[3];
6     return true;
7 }

```

Sensor::getOrientation

5.3 Using the plug-in in Vizard

5.3.1 Importing the plug-in

Now that we have an operational plug-in, we can import it into Vizard and use it to create a virtual environment. There are at least two required importations:

```
1 import viz
2 import oculus
```

viz is the basic Vizard library, and *oculus* is the Vizard plug-in that allows us to connect to an Oculus VR headset.

Hence, the first thing that we need to do, before creating our virtual environment, is to connect to our headset:

```
1 hmd = oculus.Rift()
2 view = viz.MainView
3 viewlink = viz.link(hmd.getSensor(), view)
4
5 if not hmd.getSensor():
6     sys.exit('Oculus Rift not detected')
```

After that, we can import our plug-in using *viz.add* and start the cameras with a predefined configuration and a number of markers to monitor:

```
1 ext = viz.add('Coda.dle')
2 ext.command(1, x=config, y=nbmarker)
```

5.3.2 Creating a sensor

We can now create sensors in Vizard that can provide position and orientation information. To visualise them, we can link 3D objects to them that will automatically adopt their position and orientation. This is done thanks to the *link* class of Vizard:

```
1 sensor = ext.addMySensor(1) #we create a sensor for the marker 1
2 viz.link(sensor, vizshape.addSphere(0.1, color=red)) #we stick a
   sphere to the sensor, its position will be updated every frame
```

If you do that, you will observe something strange, the sphere is not at the right place! In fact, the Codamotion cameras and the virtual environment may not have the same coordinate system. To see the objects where they should be, we will have to align both coordinate systems.

5.3.3 Aligning virtual with real

A coordinate system, or reference frame, is characterised by its origin, orientation and scale, which are specified by a set of reference points. In our case, we will have to find the transformation matrix that links the Codamotion "real" coordinate system with the Vizard "virtual" coordinate system. As the headset is the only thing that links the virtual environment with reality, we are going to use its position and orientation to create our transformation matrix.

In concrete terms, we fix three markers on the VR headset to have its relative coordinate system. With this, we find the transformation matrix between the Codamotion coordinate system and the VR headset, then the matrix between the headset and the virtual coordinate system, and finally we multiply the first one by the second one to find our full transformation matrix.

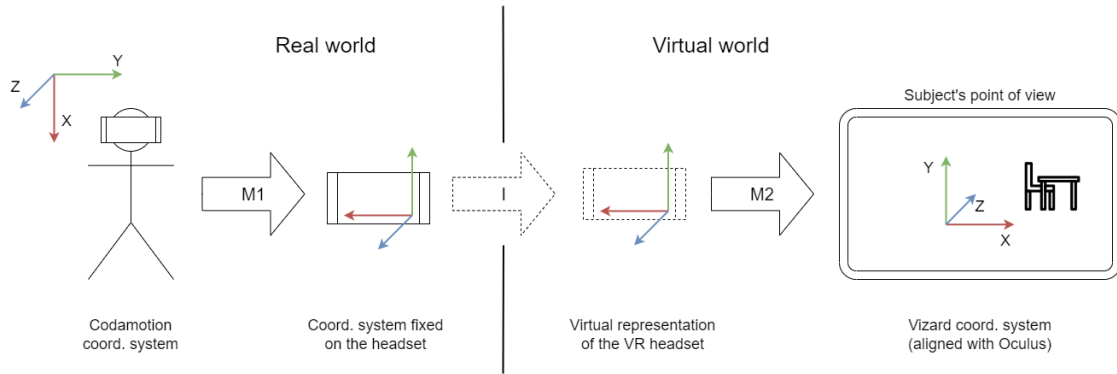


Figure 5.2: We need to find M1 and M2

As said in section 5.2.2, there are two big differences between the coordinate systems of Codamotion and Vizard. Firstly, the first one represents positions in millimetres, while in the second one they are represented in meters. Secondly, Codamotion uses a right-handed base for its vector orientation, while Vizard uses a left-handed base. But as we have seen previously, we solved these problems in the *Coda::acquire* function by dividing the monitored positions by 1000 and inverting the Z axis.

There is actually one more matrix in our computations: the one between the headset coordinate system that we built up with our markers and the coordinate system of the representation of the headset in the virtual environment. Fortunately, we can place our markers such that the first one matches the second one and the transformation matrix is reduced to the identity matrix.

Hence, we have to place a marker in the bottom-left corner of the headset (while wearing it) that will represent the origin of our coordinate system. The X axis will be defined by this marker and another one placed in the bottom-right corner of the headset, and the Y axis by the origin marker and one placed in the top-left corner. The Z axis will thus point forward. There is a small offset between the centre of the positions of the markers and the headset, but it can be corrected in the first matrix.

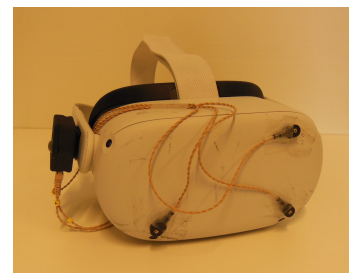


Figure 5.3: Markers on the headset

From Codamotion to headset

To get our first matrix, we are going to build a sensor object representing the markers that we put on our headset. In the code below, we represented the origin, X axis and Y axis markers of our headset by the letters O, X and Y.

```
1 #retrieve position and orientation of the head
2 headPos = [0.0, 0.0, 0.0]
3 head = ext.addMySensor(0)
4
5 head.command(1, x=X)
6 head.command(1, x=Y)
7 head.command(2, "", 0, X, 0, Y)
8
9 while (sum(headPos) < 0.0001 or np.isnan(headPos[0])):
10     head.command(3)
11     headPos = head.getPosition()
12     headOri = head.getQuat()
```

The while loop is there to ensure that we do not acquire the information while one of the markers is not visible and that *getPosition* does not return NaN values such as we have observed sometimes. In short, to make sure that we have the correct information to work with.

You may also have noticed the command 3 that we did not implement in section 5.2.3. It is here due to a particularity in the functioning of Vizard with the sensors. Indeed, *Sensor::update* is called at the beginning of every frame. However, it is true when Vizard has already launched the virtual environment, which is not done yet when we want to construct the matrix to orient that same environment. This is why we have to implement this command to call *Sensor::update*³:

```
1 case 3:
2 {
3     viz::Event e = viz::Event(1);
4     this->update(e);
5     break;
6 }
```

Sensor::command (3)

This way, *Sensor::update* will compute the position and orientation of our sensor, and we will be able to retrieve them with *getPosition* and *getQuat*⁴.

³*Sensor::update* cannot be called directly from python using a built-in function such as *getPosition* for example.

⁴There is no *getOrientation* in Vizard, but instead the three functions *getQuat*, *getEuler* and *getAxisAngle*. They all call *Sensor::getOrientation*, but return the orientation in different formats.

We can now create our first matrix. Vizard has a dedicated class for transformation matrices called *Transform* which comes with a few useful functions. After the creation of the matrix, *postQuat* is used to define the rotation and *postTrans* define the translation. This will construct the matrix from headset to coda, so we have to invert it afterwards.

```

1 #Transform matrix from coda to headset
2 M1 = viz.Transform()
3 M1.postQuat(headOri)
4 M1.postTrans(headPos)
5 M1.postTrans([-0.015, 0.01, -0.06]) # Corrects the offset between
   the center of the markers and the center of the headset (in
   metres)
6 M1.invert()

```

The second translation is there to correct the offset of the position between the computed centroid of the markers and the true centre of the headset. The offset should be approximately equal to the one mentioned here, but may need to be measured again for better precision, if the markers are removed and put back between experiments. A solution to delete the offset in the X and Y-axis could be to place a fourth marker in the top-right corner, so that their centroid is also at the centre of the headset on the XY plan. And a solution to avoid the imprecisions due to the displacements of the markers could be to 3D-print a framework made to host the markers and that can be attached to the headset. That way, the markers remain at the same place from one utilisation to another.

From headset to VR

The transformation matrix of the headset in the VR environment can easily be retrieved using the function *getMatrix*:

```

1 #Transform matrix from headset to VR environment
2 M2 = viz.Transform()
3 M2.set(hmd.getSensor().getMatrix())

```

The full transformation matrix

The last step is to multiply the two matrices to have our full transformation matrix:

```

1 M = M1*M2

```

Now we can apply this matrix to the link between our sensor and the 3D shape representing the marker or the set of markers, and it will appear at the right position.

```

1 sensor = ext.addMySensor(1)
2 link = viz.link(sensor, vizshape.addSphere(0.1, color=red))
3 link.postQuat(M.getQuat()) #orientation
4 link.postTrans(list(M.getTrans())) #translation

```

After that, if a rotation, translation or scaling needs to be done for an experiment, we can either multiply the M matrix with another matrix, or use the *post* functions (*postTrans*, *postQuat*, etc) on the objects individually.

5.4 Final product

5.4.1 Summary

The code below gathers all the functions available in our plug-in and what they are used for:

```
1 # Launch the plug-in, return an Extension object
2 ext = viz.add('Coda.dle')
3
4 # Start the cameras without making an alignment
5 ext.command(command=1, x=config, y=nbmarker)
6 # Start the cameras with making an alignment
7 ext.command(command=2, x=config, y=0, z=X, w=Y)
8 # Stop the cameras and save the acquired data
9 ext.command(command=3, message=filename)
10
11 # Create a sensor based on one marker
12 sensor = ext.addMySensor(marker)
13
14 # Add a marker to the sensor
15 sensor.command(command=1, x=marker)
16 # Define an axis system for the sensor
17 sensor.command(command=2, x=X1, y=X2, z=Y1, w=Y2)
18 # Update the position and orientation of the sensor (when the VE is
19   launched, this is automatically done at every frame)
20 sensor.command(command=3)
21
22 # Return the position of the sensor (when the VE is launched, this
23   is automatically done at every frame)
24 pos = sensor.getPosition()
25 # Return the orientation of the sensor as a quaternion (when the VE
26   is launched, this is automatically done at every frame)
27 quat = sensor.getQuat()
28
29 # Return the orientation of the sensor as euler angles
30 euler = sensor.getEuler()
31 # Return the orientation of the sensor as axis angles
32 axisAngle = sensor.getAxisAngle()
33 # Return the transformation matrix of the sensor
34 matrix = sensor.getMatrix()
```

Functions of our interface

Figure 5.4 represents a diagram giving an overview of what has been done in this chapter. A more complete diagram including all the functions of RTNet SDK and the python file that we will present in section 5.4.2 is available in appendix C.

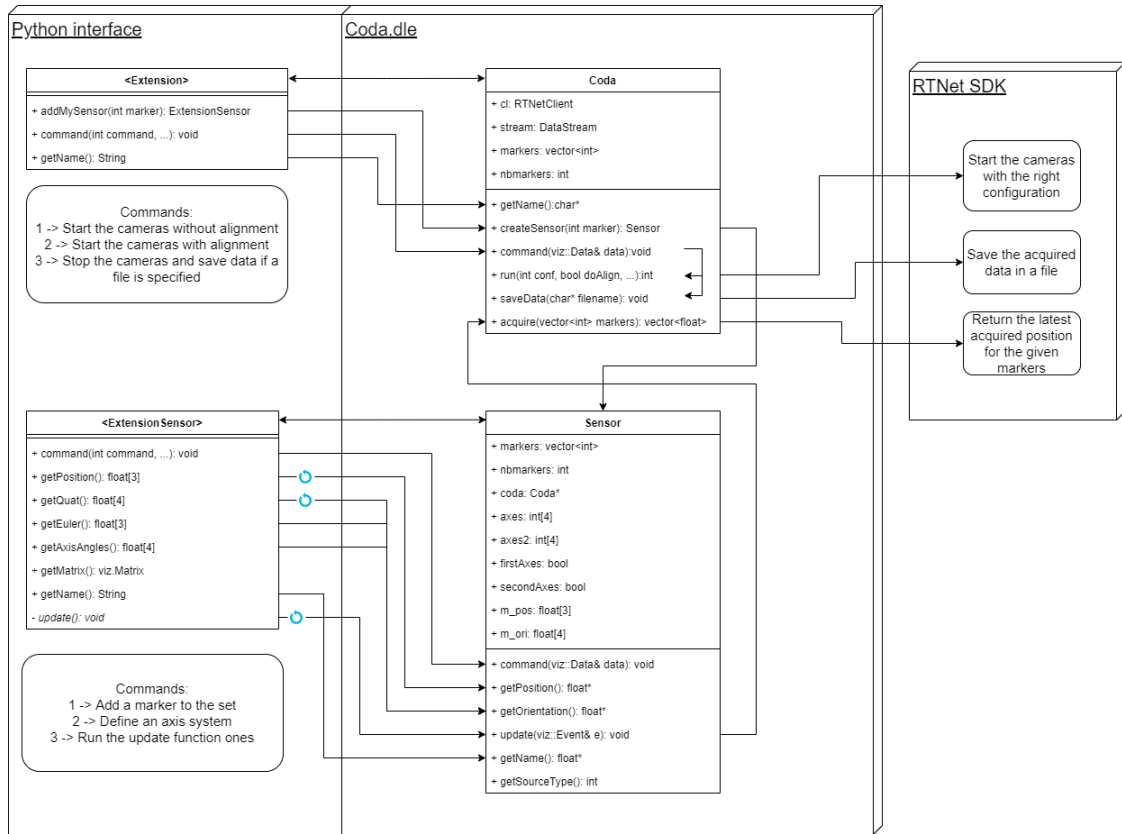


Figure 5.4: Diagram of our interface

5.4.2 Codamotion.py

To facilitate the usage of the plug-in in Vizard, we created a python file called *Codamotion.py* containing a class and diverse functions automating the few steps that are detailed above.

```

1 class MarkerSet:
2     """A class representing a sensor and its shape"""
3
4     def __init__(self, ext, markers, M, shape):
5         """Constructor.
6         inputs:
7             ext: the plug-in
8             markers: int[] containing the numbers of the
9                 markers in the set
10            M: the transformation matrix from real to virtual
11                coordinate system
12            shape: the 3D shape to link to the sensor"""
13
14     def addAxes(self, X1, X2, Y1, Y2):
15         """Add an axis for the sensor
16         inputs:
17             X1 and X2: markers for origin and direction of the
18                 X axis
19             Y1 and Y2: markers for origin and direction of the
20                 Y axis"""

```

```

17     def update(self):
18         """Perform one update (only useful if VE has not been
19             launched with viz.go() yet)"""
20
21     def align(ext, hmd, file = "matrix.csv", O = 4, X = 5, Y = 6)
22         """Perform the alignment between the Codamotion coordinate
23             system and the Vizard coordinate system and return the
24             resulting transformation matrix. Also save it in a file.
25             inputs:
26                 ext: the plug-in
27                 hmd: the virtual representation of the headset, obtained
28                     with oculus.Rift()
29                 file: the file in which to save the matrix
30                 O, X and Y: the numbers of the markers on the headset,
31                     representing the origin, the x-axis and the y-axis of
32                     the headset coordinate system respectively. """
33
34     def getMatrix(file = "matrix.csv")
35         """Return the matrix saved in a file.
36         input:
37             file: where the matrix is saved."""
38
39     def getMarkersFromDriveBox(driveBox)
40         """Return an array of int containing the numbers of the markers
41             connected to the drive box given as argument.
42         input:
43             driveBox: the letter of the drive box as a string in
44                 capital letters, it can go from A to F"""
45
46     def manipulandum(ext, driveBox, M)
47         """Return an object of the class MarkerSet representing the
48             manipulandum as a blue box.
49         input:
50             ext: the plug-in
51             driveBox: the letter of the driveBox used for the
52                 manipulandum
53             M: the transformation matrix from real to virtual
54                 coordinate system."""

```

Codamotion.py

5.4.3 Example of a Vizard script

With this following script, supposing that you use the drive box A for the manipulandum, you can have a virtual representation of the manipulandum as a box that follows your movements and register its position in "Test.txt" from the moment you launch the script until you press "q" on the keyboard. We also suppose that alignment has already been done and saved in the file matrix.csv.

```
1 import viz
2 import vizshape
3 import vizact
4 import oculus
5 import Codamotion as coda
6
7 file = "Test.txt"
8 config = 3
9
10 # Connect to the headset
11 hmd = oculus.Rift()
12 view = viz.MainView
13 viewlink = viz.link(hmd.getSensor(), view)
14
15 if not hmd.getSensor():
16     sys.exit('Oculus Rift not detected')
17
18 # Launch the plug-in and start the cameras
19 ext = viz.add('Coda.dle')
20 ext.command(1, x=config, y=4)
21
22 #Retrieve the alignment matrix
23 M = coda.getMatrix("matrix.csv")
24
25 # Creates a Sensor for the manipulandum with the markers of the
    drive box A
26 manip = coda.manipulandum(ext, "A", M)
27 manip.addAxes(3, 2, 3, 1)
28 manip.addAxes(1, 0, 2, 0)
29
30 # Function to stop the cameras and save the data, then quit the VE
31 def stop(ext):
32     ext.command(3, file)
33     viz.quit()
34
35 # Define the "q" key as command to stop the acquisition
36 vizact.onkeydown("q", stop, ext)
37
38 # Launch the VE
39 viz.go()
```

Example of code using Codamotion.py

Chapter 6

Usage

This chapter is dedicated to the usage of our interface, what should or should not be done, and what possibilities Vizard offers to build up an experiment. On top of that, we made a ready-to-use zip folder containing all the necessary files to create an experiment. This folder contains a template Vizard file, our plug-in, the *codaRTNetProtocol.dll* file, *Codamotion.py* and *Alignement.py*, that will be presented in section 6.1.1. This folder can be found on the desktop of the lab's computer.

6.1 Usage of the set-up

6.1.1 Before the experiment

Here are the few steps that must be followed to make the system ready for use.

Installing the drive boxes and LED markers

We can distinguish three groups of markers: one for the Coda coordinate system, one on the headset, and the last one that is monitored and displayed in the headset.

The Coda coordinate system (3 markers, 1 drive box)

As mentioned in section 5.2.2, these markers must be placed in a right-angle configuration with one marker as the origin, one defining the X axis, and one defining the Y axis. If the rotating chair is used, they can be placed as illustrated in figure 6.1. It is important to remember which markers are used as they will have to be indicated for the Codamotion alignment. In our case, we used drive box C with the marker 10 as origin, 9 for the X axis and 8 for the Y axis. They are set as default markers in our python script but can be changed.



Figure 6.1: Coda axis system

The headset coordinate system (3 markers, 1 drive box)



Figure 6.2: Axis on the headset

These markers are used for the alignment of the Coda coordinate system and the VR coordinate system detailed in section 5.3.3. The axis that they define must be aligned with the axis of the representation of the headset in Vizard, that is, the positive X axis going right and the positive Y axis going up from the point of view of somebody wearing the headset. In our case, we used drive box B with the marker 4 for the origin, 5 for the X axis, and 6 for the Y axis. Once again, they are set as default values in our script but can be changed.

The monitored markers (At least 1 marker and 1 drive box)

If it was not the case for the previous groups of markers, these are going to be monitored during the experiment and saved for post-processing. We recommend using the lowest markers in numerical order as indicating a low number as *limitMarker-Number* will allow you to significantly accelerate the data saving process (see section 5.2.2). When setting up your markers, if you group them as one object and want to display its orientation, make sure to have at least three markers in a right-angle configuration. However, the best alternative is to set four markers in a rectangle-shaped configuration so that you can define two different axis marks. For the particular case of the manipulandum, four LEDs are embedded on the front side and can be connected to a drive box with two special cables.

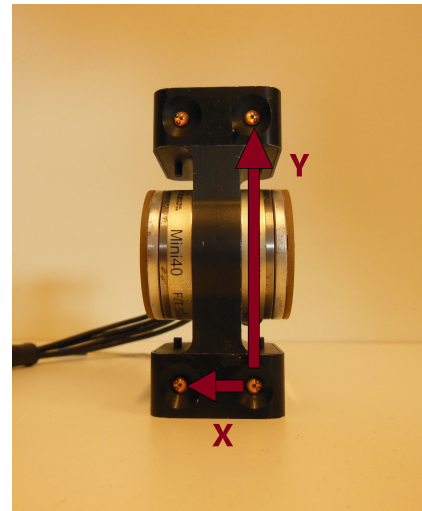


Figure 6.3: Axis on the manipulandum

Installing the cameras

At least one CODA CX1 unit is necessary for the position monitoring of the markers, but two can be used for better precision. They must be placed in a particular range of distance away from the markers that are specific to each unit, usually between two and four meters. Every marker and drive box should be visible from the cameras. Make sure that each used camera is connected to the Mini Hub and that this one is switched on, if so, the green LED of the CX1 unit should be turned on. Finally, run the RTNet server as an administrator.

In Vizard

In Vizard, before the experiment, the most important thing to do is the alignment. There are two alignments to perform, one for the Codamotion system and one for the virtual environment. The first one can be reused if the cameras and markers do not move between the experiments, but the second one should be done every time the headset is turned on. However, we recommend to always perform both of them at the same time, to make sure everything works correctly.

Regarding the coordinate system of the virtual environment, it may be interesting to understand how and when it is fixed. When creating the virtual environment, Vizard aligns it with the internal coordinate system of the Oculus Quest itself. This one, however, is fixed when turning on the headset, the Y axis is aligned with the gravity, going up, and the Z axis is going in the direction of the look, i.e., straight ahead. This internal coordinate system is fixed every time we light up the headset, and also every time it comes back from sleep mode, for example when the headset is taken off and then put back on¹. This is why we recommend to always perform the alignment before making any acquisition.

To facilitate this process, we made a python script (called Alignment.py) that can be launched once before the experiment. It will save the transformation matrix in a file that can be reused for the rest of the experiment. However, as we did not manage to save several Codamotion alignments, if the coordinate system of the cameras is modified, for example when changing the rotating chair orientation, this alignment script should be run again.

It works as follows: first we enter the name of the file where we want the transformation matrix to be saved and the Codamotion configuration to use. Then we connect to the headset and we start the plug-in, via the plug-in we make the alignment of the Codamotion system, and finally we make the alignment with the virtual environment.

```
1 import viz
2 import oculus
3 import Codamotion as coda
4 import time
5
6 # Enter here the name of the file in which to save the
7   transformation matrix and the config number to use
8 matrixfile = "matrix.csv"
9 config = 3
10
11 # Connect with the Oculus Quest
12 hmd = oculus.Rift()
13 view = viz.MainView
14 viewlink = viz.link(hmd.getSensor(), view)
15
16 if not hmd.getSensor():
17     sys.exit('Oculus Rift not detected')
```

¹The automatic sleep mode when the headset is not being worn can be deactivated.

```

17 # Launch the plug-in
18 ext = viz.add('Coda.dle')
19 # Start the cameras and make the alignment
20 ext.command(2, x=config, y=11, z=10, w=9)
21 time.sleep(1) # Make sure the system has started and Coda alignment
                is done
22 # Perform the alignment between Codamotion and the virtual
                environment
23 M = coda.align(ext, hmd, matrixfile)
24 # Stop the cameras without saving the data
25 ext.command(3)

```

As long as the Codamotion and the headset coordinate systems are not modified, the matrix generated by this script can be used in your experiments. This matrix can be retrieved using these few lines:

```

1 matrix = np.loadtxt("matrix.csv")
2 M = viz.Transform()
3 M.set(list(matrix))

```

Regarding the numbering of the markers, there is something to know about the alignment. You may notice that we previously mentioned using markers 8, 9 and 10 for the Coda alignment, but in the script above the markers 9, 10 and 11 are used. For some reason, Codamotion numbers the markers starting from 0 everywhere, except for the alignment, where they are numbered starting from 1. This is why we use markers 9, 10 and 11 in our script. However, if we want to display their position in our VE, we would need to create sensors for the markers 8, 9 and 10, which are the same, but numbered differently.

6.1.2 During the experiment

To create an experiment, the easiest way to proceed is to make a script that represents a block of acquisition in the experiment. Vizard offers the possibility to easily modify parameters between each run to affect the visual perception of the subject between each block. Hence, from the perspective of the researcher and considering that the cameras and markers have already been installed, an experiment would be controlled as follows:

1. Install the subject,
2. Perform the alignment,
3. Run the experiment script,
4. Start the acquisition of the manipulandum data if needed,
5. Let the subject perform the task,
6. Stop the experiment script,
7. Change the parameters as needed and restart from 2 if the Codamotion coordinate system has changed, 3 otherwise.

When installing the subject, he or she may need to perform a few operations using the Oculus controllers in the headset before it is ready for the experiment :

1. Denying access to data from the computer
2. Allowing oculus link

However, if it has been done before by someone else and the automatic sleep mode is deactivated, the subject should directly end up in the oculus link environment when putting the headset on.

Regarding the usage of the manipulandum, it may be possible to synchronise its launching with one of the cameras. It has already been done in the laboratory, but the triggering always came from the manipulandum controller. The problem here is that it is Vizard who launches the cameras via the plug-in. We believe that the easiest way to synchronise Vizard, the cameras and the manipulandum controller is to trigger the manipulandum from the plug-in, but we did not have the time to investigate more than that. The main advantage would be that the Codamotion data would be synchronised with the manipulandum data and would also be synchronised between the blocks.

6.1.3 After the experiment

After the experiment, in addition to tidying up everything, one should pay attention to put the drive boxes on charge and to correctly shut down the Oculus Quest 2, as it may progressively lose battery while on sleep mode.

The acquisition data files should be in the folder and under the name that you indicated for data saving.

6.2 How to implement an experiment

As it is a specialised software, Vizard offers several tools to create experiments. In this section, we will talk about the most interesting ones for the kind of experiments that could be conducted in the COSY laboratory. What we explain here are introductions to the most relevant features, many other possibilities are explained in the Vizard documentation [31].

6.2.1 Launching a new world

After having launched Vizard², you can create a new script by clicking File → New Vizard File. The following lines are mandatory if you want your script to work:

```
1 import viz
2 viz.go()
```

²For Vizard to be allowed to have access to the RTNet DLL file, it must be opened as administrator, otherwise it will fail to launch the plug-in.

The first line gives you access to the whole Vizard library. And on the second line, `viz.go()` is the command used to launch the Vizard rendering environment, in other words, it starts the virtual world. On the contrary, stopping the virtual environment is done with `viz.quit()`.

In order to control the viewpoint from where we see our world, we use the `viz.MainView` object. It can be seen as the camera that records images of our world and displays them on the screen of our computer.

As we use an Oculus Quest 2, we need to connect to it. In Vizard, all the headsets of the Oculus family are supported using the *oculus* plug-in provided by Vizard. The connection is done this way:

```
1 import oculus
2 hmd = oculus.Rift()
3 viz.link(hmd.getSensor(), viz.MainView)
```

We firstly import the oculus library, then create an object of the class `oculus.Rift()`³. This class provides an interface to connect to our headset and render our virtual environment in it. This class will automatically detect our headset if it is connected to our computer via Oculus Link (see section 3.1). The sensor object providing the orientation and position values associated with the headset can be retrieved with `getSensor`, meaning that if we link it to the main view, it will follow the movements of our head.

You can now start the script by clicking on the green arrow at the top of the screen. However, if you do it now, you will end up in an empty environment. In the next section, we will see how to fill in our virtual world.

6.2.2 Creating a virtual environment

Modelling

A virtual environment is composed of 3D shapes. In Vizard, they can either be generated using the *vizshape* module, or imported from external files. In both cases, these objects will belong to the class *node3d*. In this section, we will refer to these objects as shapes, models or nodes in an equivalent manner.

The *vizshape* module has several functions to create simple shapes such as spheres, boxes, cylinders, etc. For example, you can create a red sphere of 1cm radius to represent a marker:

```
1 sphere = vizshape.addSphere(0.01, color=viz.RED)
```

Or a blue box to model the manipulandum:

```
1 manip = vizshape.addBox([0.05,0.1,0.05], color=viz.BLUE)
```

³hmd is for Head Mounted Device.

However, in order to have a realistic environment, more complex shapes are needed. Fortunately, Vizard allows the importation of 3D models of most of the standard formats, although the most recommended are glTF 2.0 and FBX. They also provide their own native format called OSG.

Vizard already provides a few of these models, but there also exist many websites referencing a wide range of different models (SketchFab, CGTrader, Turbo Squid, etc). However, from our experience, finding free and quality models is not as easy as it seems. The last possibility, for the most courageous ones, is to create their own models using a modelling software such as 3D Studio Max, Maya or Blender.

Importing those models is done via the function *add Child*, for example, you can add a plant to your virtual world with:

```
1 plant = viz.addChild("plant.osgb")
```



Figure 6.4: 3D model of a plant.

The class *node3d* provides a lot of functions to control our models, including their position, orientation and scale in the virtual environment:

```
1 plant.setPosition([5, 0, 2]) #the position is given in X Y Z  
   coordinates in metres  
2 plant.setEuler([90, 0, 0]) #the orientation can be modified using  
   euler angles, axis angles or quaternions  
3 plant.setScale([0.1, 0.1, 0.1]) #tiny plant
```

Vizard also provides models representing a whole scene, here is an example with an Italian piazza:



Figure 6.5: 3D model of a piazza.

However, as an Italian piazza may not be the best environment to conduct a scientific experiment, an easy way to model an empty room is by creating 4 walls, a floor, and a ceiling with 6 flattened box shapes. For more realism, it is also possible to apply textures on those boxes, and with a few decorative elements, it gives a result such as illustrated in figure 6.6.



Figure 6.6: Simple room as 3D environment

Another option to create a more realistic environment is using cube maps. They are composed of 6 pictures representing the textures of the 6 sides of a cube that Vizard uses to create a dome centred on the user. The code below generates the environment of the figure 6.7.

```
1 env = viz.addEnvironmentMap('Townhall_L.jpg') #create a cube map
   with the textures of a town hall
2 sky = viz.addCustomNode('skydome.dlc') #create the dome object
3 sky.texture(env) #apply the textures to the dome
```

However, as Vizard is not a wizard, it does not generate a fully 3D environment from only 6 pictures. If the user moves the environment will "move" with him, i.e., it will stay centred on the user. For example, in the example below, it is impossible to go behind the table and see the back of the chairs. If this example is striking, this feature is more useful to display landscapes such as a plain field with mountains far away that keep the same position relative to our body if we move around.



Figure 6.7: A town hall environment

Lighting

In order to correctly see our environment, we need to illuminate it. Vizard features three types of lights: directional, point and spot. In order to add a light to our environment, we use *viz.addLight*. After that, the type of this light is defined by the parameters that we set. That is to say, we need to define the direction for the light to be directional, the position to be a point light and both of them to be a spot light.

Directional lights are infinite waves of lights going always in the same direction. This light does not attenuate with distance, and its rays always remain parallel.

```
1 myLight = viz.addLight()  
2 myLight.direction(1, 0, 0)
```

Point lights originate from one point in space and illuminate in every direction. The intensity of point lights decreases with the distance. We can control the position and intensity of the light.

```
1 mylight = viz.addLight()  
2 mylight.position(0, 1, 0)  
3 mylight.intensity(2)
```

Finally, spot lights are a combination of directional and point lights. They diffuse a cone of light in a given direction, with the spread parameter controlling the width of this cone.

```
1 mylight = viz.addLight()  
2 mylight.position(0, 1, 0)  
3 mylight.direction(0, 0, 1, 1)  
4 mylight.spread(90)  
5 mylight.intensity(2)
```

Orienting

As part of the lab's experiments, it could be interesting to control the orientation of the virtual environment. However, modifying the position and orientation of every node independently would be a mess. This is where Vizard introduces its system of hierarchical relationships between models. Literally, we can define a node as the parent of another node. A node can have several children but can only have one parent, such that the relations between our nodes will create a tree called the scene graph.

The main difference between these relations and the *link* class is that a link can be created between a model and a sensor for example, where these hierarchical relationships are only possible between nodes. Yet they work in a similar way, if the transformation matrix of a parent node is modified, the matrices of all its children will be modified accordingly.

There are two ways to set a node as the parent of another, either at the creation, or afterwards with *setParent*:

```
1 plant = viz.addChild('plant.osgb')
2
3 #defining relationship when creating the object
4 plant2 = viz.addChild('plant.osgb',parent=plant)
5
6 #defining relationship afterwards
7 plant3 = viz.addChild('plant.osgb')
8 plant3.setParent(plant)
9
10 #mama plant has now 2 baby plants
```

When using `addChild` without parent argument, the object is set as child of the virtual world, represented by `viz.WORLD`. Therefore, `viz.WORLD` is actually the root node of the scene graph⁴.

In Vizard, every *node3d* object has its own internal coordinate system. The effect of the hierarchical relationship is to set the parent's coordinate system as default coordinate system for the node. On this basis, calling `setPosition` on the node will change its position with regard to its parent coordinate system⁵. In the same way, if the parent rotates, the child will not only rotate as well but also move in the world to stay at the same position within its parent coordinate system.

Thanks to that, we can create an invisible object or an object of size 0 that will be the parent of every node constituting our environment. That way, we only have to rotate this parent node to effectively rotate the whole environment.

6.2.3 Flow control

During an experiment, it could be useful to trigger events or actions, whether it is controlled by the subject or by the researcher. For that, Vizard introduces three concepts: events, actions and tasks.

Events

User interaction is mainly done through events. For example, it could be a keyboard entry, clicking on a mouse button, interacting with something in the virtual environment or indirectly controlled events such as timer expiration. All those events can be linked to a function so that it will be called whenever the event occurs.

For example, we want to trigger something when pressing a key. We firstly create a function for this purpose:

```
1 def HandleKeyboard(key):
2     # do something with key
3     print(key)
```

⁴Using `setPosition` or `setEuler` on `viz.WORLD` does not work.

⁵A parameter of `setPosition` can be modified to work in the world's coordinate system.

Now, in order to be notified of an event, we need to create a callback function for a particular event. Vizard will call this function when the event occurs.

```
1 viz.callback(viz.KEYDOWN_EVENT, HandleKeyboard)
```

Now, whenever we press a key, Vizard will call `HandleKeyboard` with the pressed key as argument.

However, this method is not very handy. We may want to create one function for a particular key or pass other arguments to the function. The *vizact* library includes a few methods for registering callback functions. Let's say that we want to turn off the cameras and close the program when pressing "q". We thus need to create a function for that:

```
1 def stop(ext, filename):
2     ext.command(3, filename)
3     viz.quit()
```

And we register it as a callback function using *vizact.onkeydown*:

```
1 vizact.onkeydown("q", stop, ext, "Test.txt")
```

vizact.onkeydown takes as argument the key to press, the function to call and the arguments to pass to the function. The *vizact* module includes many similar functions that can cover a wide variety of events.

As inputs that can trigger events, we may also use the Oculus Touch controllers. Within Vizard, they are represented by sensor objects and can be retrieved separately using two simple functions:

```
1 import oculus #The oculus plug-in
2 hmd = oculus.Rift() #connecting to the headset
3
4 # getting the touch controllers
5 rightTouch = hmd.getRightTouchController()
6 leftTouch = hmd.getLeftTouchController()
```

Hence, the touch inputs can be retrieved using the *vizact.onsensordown* function. For instance, we can let the choice of closing the program to the subject by pressing B on the right controller (which may not be a good idea, but this is for the example):

```
1 vizact.onsensordown(rightTouch, oculus.BUTTON_B, stop, ext, "Test.
    txt")
```

Actions

Besides managing events, the *vizact* module is useful to animate objects. In Vizard, each node has its own queue of actions, when an action is finished, the next one in the queue starts. For example, in order to make our environment more realistic, we want to add humans in the background. We can animate avatars using built-in animations included in *vizact*. Avatars come with a few static states, the basic one is an idle animation. Using *state* we can make an avatar repeating a state animation forever.

```
1 dude = viz.add('vcc_male.cfg')
2 dude.state(1) #looping idle animation
```

If we want our avatar to be more active, we can make it walk from one point to another.

```
1 dude = viz.add('vcc_male.cfg')
2 walk = vizact.walkTo([1,0,5]) #create the action
3 dude.addAction(walk) #add the walk action to the avatar's queue
4 #After walking, the avatar will be automatically looping idle
  animation
```

We firstly create an action, here *walkTo* is an action for avatars that will move the avatar towards the given point with a walking animation. After that, we need to add the action to the node's queue using *addAction*. In the example below, we queue up more actions making the gentleman walk to [1, 0, 5], run the idle animation 2 times, cheer up once and finally walk to [-2, 0, 4].

```
1 dude = viz.add('vcc_male.cfg')
2
3 #create actions
4 walk = vizact.walkTo([1,0,5])
5 idle = vizact.animation(1)
6 cheerUp = vizact.animation(3)
7 walk2 = vizact.walkTo([-2, 0, 4])
8
9 #add actions to the queue
10 dude.addAction(walk)
11 dude.addAction(idle)
12 dude.addAction(idle)
13 dude.addAction(cheerUp)
14 dude.addAction(walk2)
15
16 #After walking, the avatar will be automatically looping idle
  animation
```

As this is a bit long, we can group these actions into a sequence. It will also allow us to repeat this sequence as many times as we want.

```
1 dude = viz.add('vcc_male.cfg')
2 #create actions
3 walk = vizact.walkTo([1, 0, 5])
4 idle = vizact.animation(1)
5 cheerUp = vizact.animation(3, 0.5, 0.5)
6 walk2 = vizact.walkTo([-2, 0, 4])
7 #create the sequence of actions
8 seq = vizact.sequence(walk, idle, cheerUp, walk2, idle, viz.FOREVER)
9 #add the sequence to the queue
10 dude.addAction(seq)
```

That way, our man will go back and forth between points [1, 0, 5] and [-2, 0, 4], with an idle and a cheer animation each time. With *viz.FOREVER*, we indicate that we want the sequence to run forever, if we want it to run only 5 times we need to replace it by 5.

We can also add randomness to our animations. We call that dynamic animation, in opposition to static animations that are the same every time they are played. In the code below, we create a dynamic parameter that will choose among a pool of animation ID's. We also want the avatar to walk to a random point in a 2m square around [1, 0, 5].

```
1 dude = viz.add('vcc_male.cfg')
2 #create dynamic parameters
3 xrange = vizact.randfloat(0, 2)
4 zrange = vizact.randfloat(4, 6)
5 animPool = [1, 3, 4, 5, 9]
6 animChoice = vizact.choice(animPool, vizact.RANDOM)
7
8 #create actions
9 walk = vizact.walkTo([xrange, 0, zrange])
10 idle = vizact.animation(1)
11 randomAnim = vizact.animation(animChoice)
12 walk2 = vizact.walkTo([-2, 0, 4])
13
14 #create the sequence of actions
15 seq = vizact.sequence(walk, idle, randomAnim, walk2, idle, viz.
16     FOREVER)
17
18 #add the sequence to the queue
19 dude.addAction(seq)
```

These dynamic parameters open up a wide range of possibilities. For example, we could randomise the position or movements of a target, the size or texture of an object to influence how the subject would apprehend its weight, or the orientation of the entire environment.

Controlling objects

The act of controlling an object is a problem of applying an action to that object following an input from the subject. It is not difficult if the input signal is instantaneous such as a button pressed, we can control the triggering of the action with an event and a callback function. However, if the control over the object must be maintained over time, it is more difficult.

To display the virtual environment, Vizard actually has a main loop that is run every frame. In this loop, it computes the actual states of every object, retrieves potential inputs as well as the orientation and position of the headset, and renders an image to display depending on all those parameters. It is also in this loop that the update function of the sensors from our plug-in is called, as well as `getPosition` and `getOrientation`. The *vizact* module actually provides tools to perform process that need to be controlled frame by frame in a callback function.

From those tools, the one we are interested in here is the function *onupdate*. It will create an event function that will be automatically called at every frame. All that remains is for us to retrieve the input value and update the object's property accordingly. For example, we want the subject to control the orientation of an object with the joystick of the right Touch controller. The goal would be for him to align this object with gravity to see how he perceives it without visual cues. Hence, the function to control that would look like this:

```
1 def spinObject(obj, controller):  
2     (y, x) = controller.getStick()  
3     spin = vizact.spin(1, 0, 0, 50*x, viz.getFrameElapsed())  
4     obj.addAction(spin)
```

In this function, we first retrieve the position of the joystick with *getStick* that return x and y positions between -1 and 1. After that we create a spin action, the first three parameters indicate on which axis the object should spin, here X, the fourth parameter is the angular velocity and the last parameter is the amount of time it should spin. We can indicate it in seconds or use `viz.FOREVER` to make it spin forever. Here we use *viz.getFrameElapsed* that returns the time between the beginning of the previous frame and the beginning of the current frame.

Finally, we add the action to the object. That way, if we fully push the joystick up, the object will spin at 50 deg/s, -50 deg/s if we fully push the object down, and the angular speed will linearly range between 50 and -50 deg/s between those positions. To take effect, we have to register this function with *onupdate*.

```
1 vizact.onupdate(0, spinObject, obj, hmd.getRightTouchController())
```

The first parameter is the priority of the function. Each action has a priority in the Vizard main loop represented by an integer, a function with priority -5 will be executed before a function with priority 0, which will be executed before a function with priority 5.

Tasks

In an experiment, the subject may need to complete a particular condition before going further. The *viztask* module allows pausing the execution of a function while waiting for a condition to be met, without stopping the Vizard update loop. This is done through the **yield** keyword.

We first need to create a function with one or more yield statements in it, then we add this function to the scheduler. At every frame, the scheduler will check if the condition is met, and block the function if not. A condition can be waiting for an event to occur, an action to finish, a few seconds to pass, etc. In the following example, we will create a red ball that will become green after five seconds.

```
1 ball = vizshape.addSphere(1, color=viz.RED)
2
3 def redToGreen():
4     yield viztask.waitTime(5)
5     ball.color(viz.GREEN)
6
7 viztask.schedule(redToGreen)
```

We can also run the task repeatedly, let's say we want the ball to alternate between red and green, we simply use a loop statement:

```
1 ball = vizshape.addSphere(1, color=viz.RED)
2
3 def redAndGreen():
4
5     while True:
6         yield viztask.waitTime(5)
7         ball.color(viz.GREEN)
8
9         yield viztask.waitTime(5)
10        ball.color(viz.RED)
11
12 myTask = viztask.schedule(redAndGreen)
```

And if we want this task to stop further in the experiment, we can kill it using:

```
1 myTask.kill()
```

6.2.4 Proximity sensors

Another intuitive way to interact with the virtual environment is by using proximity sensors. They will check for proximity between Vizard objects based on position and trigger events. Proximity sensors are managed with the vizproximity library and are composed of three classes: *Sensor*, *Target* and *Manager*. The Manager detects when a Target enters the range of a Sensor and sends a proximity event.

For example, we want to create a circular sensor that the subject will have to touch with his finger. To detect the position of its finger, we stick a LED marker at the tip of it, and we will retrieve its position with our plug-in. We could decide that the subject cannot see his fingertip by making the sphere invisible, but we still need the shape to make it a vizproximity target.

```
1 #launch plug-in, alignment should have already been done and saved
   in "matrix.csv"
2 ext = viz.add('Coda.dle')
3 ext.command(1, x=3, y=1)
4
5 M = coda.getMatrix("matrix.csv")
6
7 #create the sensor for marker 0 and link a sphere to it, this is
   not the same kind of sensor as a vizproximity sensor
8
9 sensor = ext.addMySensor(0)
10 fingerTip = vizshape.addSphere(0.1, color=viz.RED)
11
12 link = viz.link(sensor, fingerTip)
13 link.postQuat(M.getQuat())
14 link.postTrans(list(M.getTrans()))
15
16 #creates the manager
17 manager = vizproximity.Manager()
18
19 #creates the target from the fingertip
20 target = vizproximity.Target(fingerTip)
21
22 #creates a proximity sensor from circle by linking it to a bounding
   sphere
23 circle = vizshape.addCircle(0.2, color=viz.RED)
24 proxSensor = vizproximity.addBoundingSphereSensor(circle)
25
26 #register the circle as a sensor and the fingertip as a target
27 manager.addSensor(proxSensor)
28 manager.addTarget(target)
```

Now that we have our sensor and our target, we need to define what to do when we touch the target, for example, we could make the circle become green, and when we leave our finger, the circle becomes red again. We firstly create two functions and then link them to enter and exit events with the manager.

```
1 #function called when we enter the sensor
2 def enterProximity(e):
3     circle.color(viz.GREEN)
4
5 #function called when we leave the sensor
6 def exitProximity(e):
7     circle.color(viz.RED)
8
9 #register them as enter and exit functions
10 manager.onEnter(proxSensor, enterProximity)
11 manager.onExit(proxSensor, exitProximity)
```

Now we would like that when the subject touches the sensor, it disappears and appears again after one second, at a random place in front of the subject. We can use the viztask library with a *waitEnter* condition:

```
1 #x and y range
2 xrange = vizact.randfloat(-0.5, 0.5)
3 yrange = vizact.randfloat(1.5, 2)
4
5 #function called when we enter the sensor
6 def proximityTask():
7     while True:
8         yield vizproximity.waitEnter(proxSensor) #send an event
9             when proxSensor detects a target
10         circle.visible(viz.OFF)
11         yield viztask.waitTime(1)
12         circle.setPosition([xrange, yrange, 0.3])
13         circle.visible(viz.ON)
14 viztask.schedule(proximityTask())
```

Chapter 7

Pilot experiment

7.1 Introduction

Once all the features of our interface were implemented, we conducted a small pilot experiment. The aim was not really to get results, but rather to find the eventual improvements that we can bring to our work. This is why we performed a superficial analysis of the recorded data, without going into too much detail.

To assess the exactitude of the setup, we asked four participants to perform rhythmic arm movements with the manipulandum (see section 3.4) in a virtual environment. The participants sat in a rotating chair (see section 3.5), right side up and upside-down, and the environment will also be rotated at 180°. It should also be noted that as we had very few participants, so the results may not be considered scientifically relevant.

7.2 Materials and methods

7.2.1 Participants

Four healthy voluntary students participated in this pilot experiment. They are all right-handed and were naive concerning the purpose of the experiment. Before these four subjects, our promoter OPSOMER Laurent also performed the task, but his results were not taken into account. Indeed, at that time, some issues still needed to be fixed and this pre-experiment allowed us to find some of the problems that remained. As this pre-experiment differed from the main experiment on some points, we considered the results no longer comparable.

7.2.2 Experimental setup

The participants were installed in the rotating chair, with the Virtual Reality headset attached to their head and the manipulandum in their right hand. The manipulandum and the headset were cable-connected to a computer. And those cables were attached with straps to the participant's forearm and to the structure of the chair, so that they did not interfere with the subjects' movements.

7.2.3 Virtual environment

The subjects were launched in a virtual white room with some objects in the background (chair, table, plant, lights) as shown on figure 7.1. Those objects were meant to give visual cues about the direction of gravity. A virtual blue rectangular parallelepiped was following the position of the manipulandum in real-time.

Two discs 40cm apart served as targets, they were at arm's length from the subject. Each target was in red during the movement of the participant, and were turning green when he reached the target. After 5 round trips, the targets disappeared and the subject was asked to continue the same movement.



Figure 7.1: Virtual environment

7.2.4 Experimental task

The subjects were asked to perform rhythmic, continuous and vertical arm movements. The movement pace was set by a sound at frequency 1Hz (one back and forth per second) during the whole experiment, and movement amplitude by the distance between the two targets. The subjects performed the task in two orientations: right side up and upside down (possible thanks to the rotating chair). For half of the blocks, the virtual environment was rotated at 180°, the goal being to give the visual perception of an inverted gravity.

In summary, the participants performed 12 blocks of 25 movements (3 blocks of each condition: head up/down and environment upwards/downwards). The subjects had the possibility to test the setup before the recording of the data, and a short break was made between each block. The order of the tasks was randomised, with the exception that the subjects did all the blocks right side up, and then all the blocks upside down.

7.2.5 Data collection

The manipulandum (see section 3.4) was used to record at 200Hz the grip force and the load force applied on the manipulandum. Before each experience, the manipulandum needs to be calibrated: we need to determine what is the zero-state, with no touching. It is necessary to ensure that the forces recorded will have the same scale and that there are no parasitic data.

CODA cameras (see section 3.3.1) were used to track the positions of CODA active markers located on the manipulandum. The data were then analysed in Python.

Three markers were placed on the structure of the rotating chair and three others on the headset to perform the alignment. Moreover, four markers were used on the manipulandum to track its position. Codamotion data were acquired at 60Hz

7.3 Results and discussion

The results are divided into two parts. The first part is devoted to the analysis of the force dynamics (GF and LF), while the second part will analyse the kinematics of the manipulandum.

7.3.1 Analysis of the forces

First impression

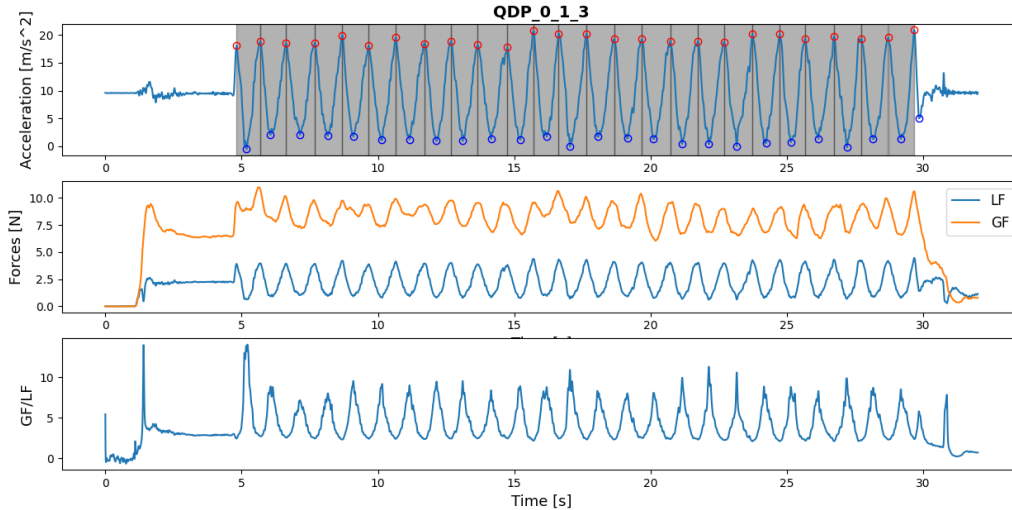


Figure 7.2: Typical result

The figure 7.2 shows a typical result of our experiment for one block (in this case 3rd trial of head up, environment down). The first graph shows the acceleration of the manipulandum in function of time. The second graph represents the GF and the LF, and the last one shows their ratio.

In the third graph, we have a good correlation between the GF and LF, meaning that this subject has properly coordinated its gripping force with the load of the manipulandum. However, looking at the other results, we noticed that this was not the case for everyone, for some of them the ratio went up to 200! This is explained by the fact that the subject gripped a little late, i.e. he gripped very hard at times when the apparent weight was light.

Measure of the GF-LF delay

To try to understand this relation between GF and LF, we used cross correlation¹ to calculate the delays between GF and LF peaks.

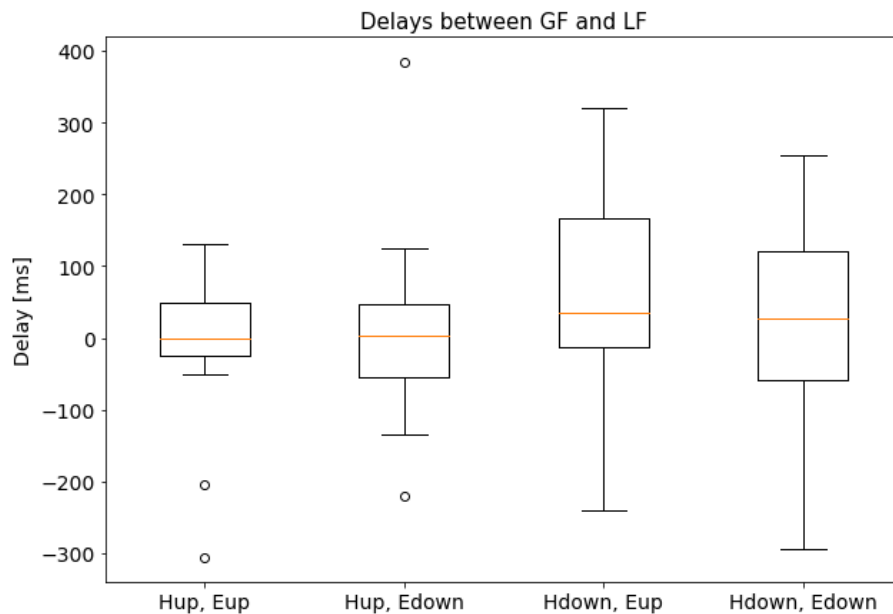


Figure 7.3: Boxplots of the delays

The results are shown as boxplots in figure 7.3, with one block per condition. The means of the four conditions seem similar, however, the variability is higher when subjects are upside-down, while some outliers are present in the right side up position.

GF when LF is max

The graph 7.4 has been obtained by calculating the mean ratio between GF and LF at the peaks of LF for the different conditions. When minimal, the LF will tend to 0 and calculating the ratio $\frac{GF}{LF}$ is not very interesting. However, at LF peaks, we could see if the subject gripped the manipulandum too much or not enough depending on the condition, but it does not seem to be the case here.

¹"In signal processing, cross-correlation is a measure of similarity of two series as a function of the displacement of one relative to the other." Wikipedia, Cross-correlation. <https://en.wikipedia.org/wiki/Cross-correlation>

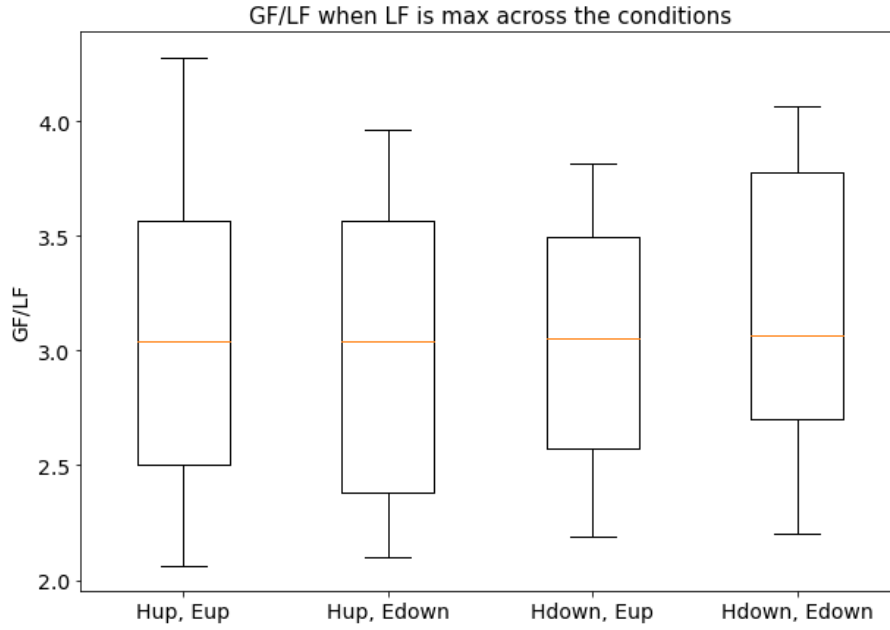


Figure 7.4: GF/LF at LF peaks

Comparison of all subjects

The graph 7.5 shows the mean of the amount of GF exerted by all subjects during all their trials depending on the LF. We can notice a difference between the physical conditions, the subjects tend to apply a bigger GF for the same amount of LF when they are upside-down. However, there are no clear differences when we flip the virtual environment, in either of the two physical conditions.

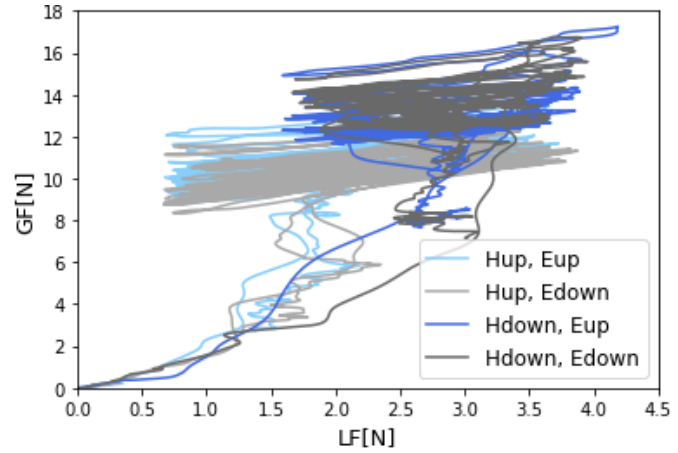


Figure 7.5: GF/LF of all subjects, all trials

Furthermore, we note in this graph that the LF does not oscillate around the same value between Hup and Hdown. This observation is verified by the graph 7.6, which represents the mean of the LF depending on the conditions. We observe that the Load Force is higher when the subjects have their head upside down, and by looking individually at the data, it is clear that this trend is not due to outliers, but is constant in all trials.

This difference in LF could be explained by the fact that part of the LF may have been taken up by the cables attached to the subjects' arm. What is less obvious is why is it only the case when subjects are upside-down, and why is it the case for all the subjects. Unfortunately, we did not manage to find a convincing answer.

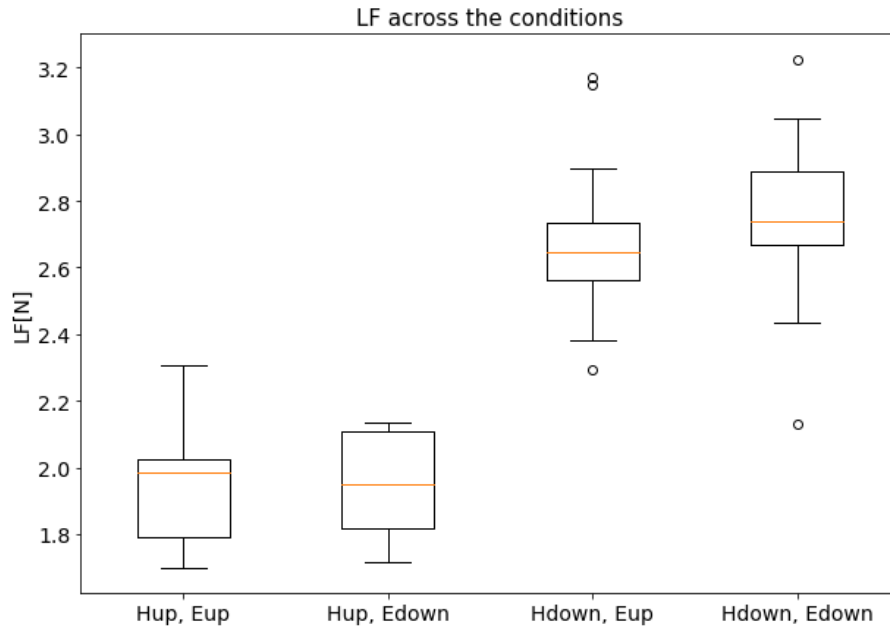


Figure 7.6: LF across conditions

7.3.2 Analysis of the positions

Figure 7.7 shows the typical trajectory of the manipulandum on the X-axis over time, during one block of acquisition. The axis system used for these graphs is the one of the Codamotion system that we defined on the structure of the chair. That means that the X-axis is going upwards, the Y axis is going left from the subject's point of view, and the Z axis is going backwards. The most interesting one for us is the position on the X-axis, as it is the direction of movement. We can see on this graph that the movement goes from 0 to 40cm, the positions of the targets.

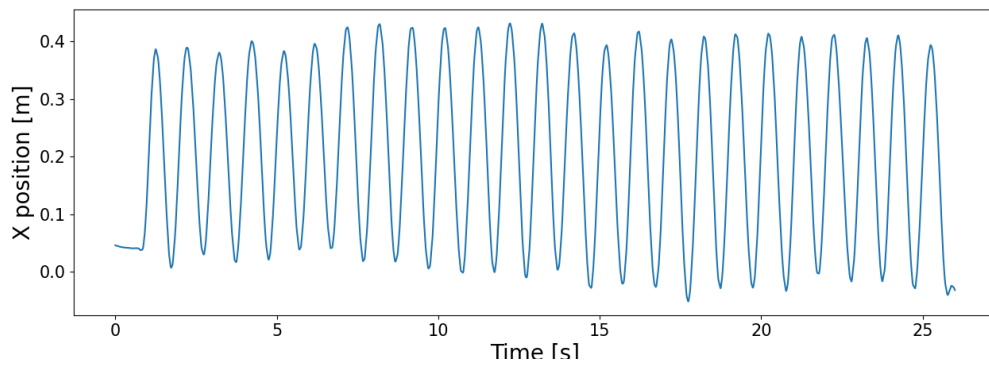


Figure 7.7: Position on the X-axis

Figure 7.8 shows the velocity of the manipulandum computed based on its position. This particular example is from an acquisition when the subject and the environment were right side up.

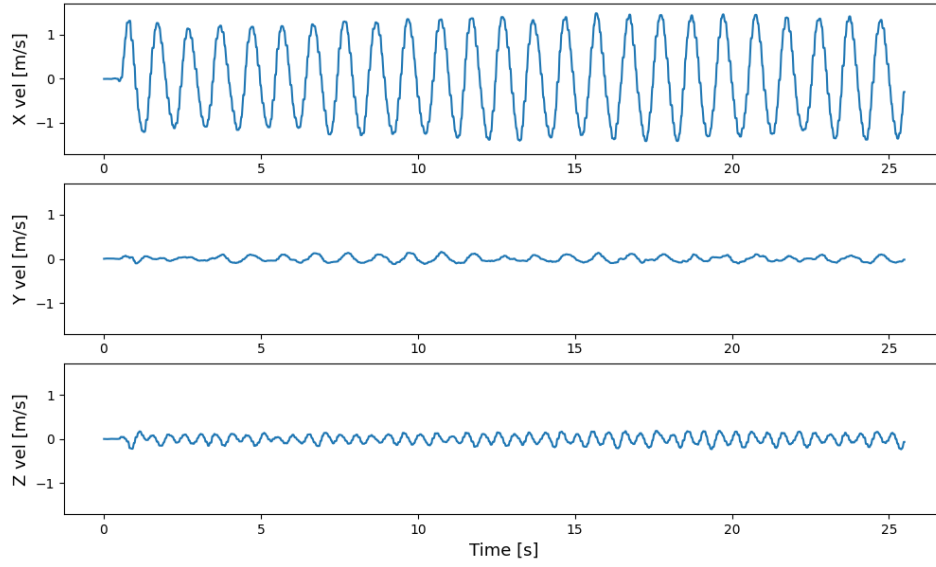


Figure 7.8: Velocity of the manipulandum

Figure 7.9 shows the absolute value of the up and down peaks of velocity, together with the linear regression. For the RU condition, the two solid lines cross each other but have roughly the same slope while for the UD condition, the linear regression is parallel and does not cross each other.

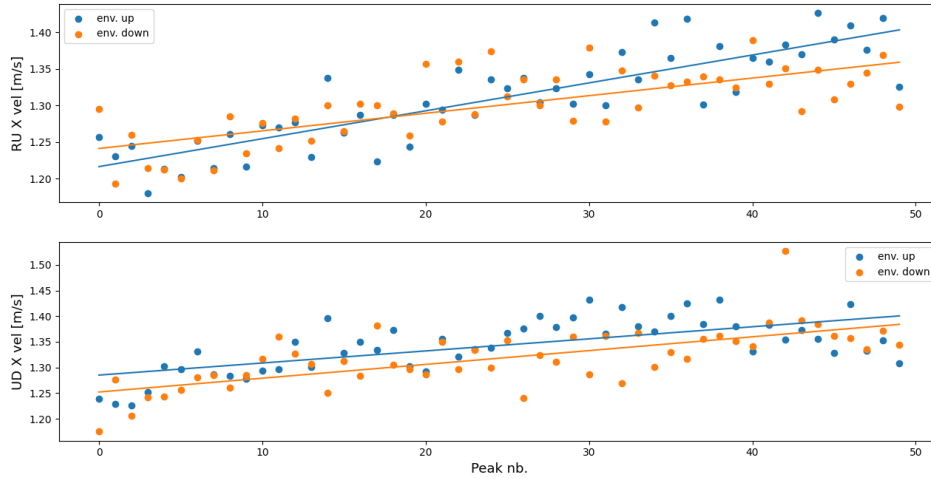


Figure 7.9: Absolute values of the peaks of velocity

For both physical conditions, the slope of the linear regression teaches us that the peak of velocity tends to be higher as the trials progressed. This means that the subjects are going faster at the end of a trial than at the beginning.

7.4 Conclusion

7.4.1 Discussion of the experiment

The main conclusion from this experiment is that, if there is a difference in object manipulation when subjects are right side up or upside-down, as it has already been seen in the literature, we could not highlight a difference when the visual environment is flipped over. However, a more complete experiment and a deeper analysis of the results should be performed in order to confirm these results.

Nevertheless, during this experiment, we have observed that subjects tend to focus more on their movement and the manipulandum than on their surroundings in the virtual environment. As the movement of the virtual object was always consistent with the real manipulandum, the subjects were not disturbed when not focusing on the environment.

One of the reasons for this is that the environment that we modelled was not striking enough. It was mostly white, with too few objects. One of our subjects did not even notice that the environment was sometimes inverted. And another thought that it was simply another environment, but not necessarily upside down.

We could have resolved this problem by making the environment busier and with more contrasts. We could also add moving humans in the background. The brain is more easily attracted by movements and by human forms. Seeing humans upside down would be more disturbing than a table or a plant staying still. Finally, another solution would be to force the subject to divert his attention from the manipulandum by making it disappear after a few back and forth movements. That way, the subject would not be able to follow the object anymore and should thus focus on his environment.

7.4.2 Issues faced

We did not push the analysis of our results further, as the goal of this pilot experiment was first and foremost to test and confirm the correct functioning of our system. That is why we choose to experience an extreme situation: totally upside down. We did not know how the headset would react, if the alignment would be done correctly, and if the captors inside the headset would not completely bug. Fortunately, everything went well on that level. Nevertheless, the experiment allowed us to raise a few issues that we resolved afterwards.

Firstly, at that moment, the saving of data was done during the acquisition. It was necessary because saving the data afterwards was taking several minutes, and it was not viable during an experiment. However, this forced us to save data at the refresh rate of Vizard (60Hz). That meant losing a lot of data compared to the 200Hz acquisition rate of the Codamotion cameras. It was also increasing the number of operations to perform in real-time and it was slowing down the whole process.

Secondly, due to the way we requested data from the Codamotion system, the computation of the orientation of objects was not perfect. When testing with the manipulandum, it was oscillating when moved around and the feeling was very unnatural. This even forced us to disable the following of the orientation for the pilot experiment.

Finally, in this previous version of our interface, we had not yet implemented the saving of the alignment. Meaning that we had to make the alignment at the beginning of every block of acquisition. As the Codamotion alignment is automatically saved, we only had to align the virtual environment, but that was still uncomfortable. Indeed, we had a few blocks where the subject moved his head during the alignment despite the instruction not to do so. Hence, the virtual environment was slightly tilted compared to the expected position.

All those issues were solved after the experiment, making our interface much more pleasant to use now. The data saving time of a 30s acquisition for 4 markers at 200Hz is approximately 4 seconds and can thus be done right after the acquisition. The orientation is now perfectly accurate to reality with no visible delay. And we can now save the alignment matrix and reuse it afterwards, so that there is no surprise during the experiment.

Chapter 8

Conclusion

8.1 Achievements

In this report, we present the development process of our interface for combining a motion tracking system with a virtual reality headset. We began by explaining the key concepts underlying the research in motor control. Then we portrayed the materials that were used and how our system was going to be implemented. After that, we described in detail how we implemented our interface and how it can be exploited with Vizard to create an experiment. Finally, we presented the pilot experiment that we conducted to assert the proper functioning of our system.

With the necessary hindsight, we believe that this equipment is sufficiently robust to be integrated as an integral component of the COSY lab's experiments. As explained in section 7.4.2, the pilot experiment made us understand with better precision how the interface will be used and thus what characteristics and features were essential to be viable as a research tool. This is the main reason why we decided to postpone the submission of our thesis. It allowed us to correct and improve the shortcomings of the system and make it flowing and pleasant to use.

8.2 Limitations

In order to measure the performances of our interface, we primarily planned to compute the total delay between the acquisition of a marker's position and the moment it is displayed in the headset. However, such a delay includes the data transfers through several pieces of hardware and software and their underlying process. Given the difficulty to compute this delay with precision and the lack of time, we decided to leave that out. However, when testing the interface, there is no visible delay, even when doing fast movements. We believe that a more limiting factor will be the 60Hz refresh rate of the virtual environment.

Of course, our interface is not perfect and could be improved in many ways. For example, the launching of the experiment could be synchronised with the starting of the manipulandum's data acquisition, which is done on another computer in the lab. But that is something to be discussed with the company that manufactured the manipulandum to see if an integration with RTNet SDK is possible. To go even further, we could try to retrieve the data of the manipulandum in real time via the plug-in and use them to influence the behaviour of the manipulandum in the VE.

Another example is the representation of the hands and arms of the subjects in the virtual environment, which we did not implement. There is a *Hand* class included in Vizard, primarily intended to be used with haptic devices such as VR gloves. It could be interesting to bypass its use to make it compatible with our plug-in, or simply use its hand 3D model to make it hold the manipulandum.

In addition to that, we could have improved the object's fluidity in the virtual environment when one or several markers are occluded. For now, when it happens, we simply remove this marker's position from the computations, which leads to a lack of precision in the position of the object. And its orientation is not even computed if the marker is part of its axis system. To improve that, we could implement a history of the markers' positions previously monitored and compute the object's theoretical current position and orientation based on it when one or several markers are occluded.

Finally, for a better precision when performing the alignment of the virtual environment, a frame could be 3D printed and used to host the LED markers around the headset. That way, they would always be at the same place with respect to the headset and the offset between the centroid of the markers and the centre of the headset could be precisely measured.

8.3 Research perspectives

This interface has been produced for the System and Cognition division of the Institute of NeuroScience at UCLouvain. Even though the laboratory that initiated the project is the one studying dexterous manipulation, several members from other laboratories of the COSY division showed an interest in the possibilities that virtual reality can bring to their own research. In this regard, we tried to make our interface as versatile as possible, even though we used the materials of the dexterous manipulation lab (manipulandum, rotating chair, etc).

We believe that VR can bring the research in neuroscience a step forward by opening the field to numerous possibilities. It allows researchers to easily create an optimal experiment environment and induce contradictions between the different sensory feedbacks. It can also be coupled with other "sensory disruptors" such as the rotating chair or even parabolic flights. With a lot of imagination, we may even see experiments conducted in the International Space Station! On top of that, Vizard, when well handled, is a powerful tool for designing VR experiments and is surprisingly effective in stimulating scientific creativity.

In an obvious manner, the main sensory information that we can alter with VR is the visual feedback. Discrepancies can be created at the level of the movement itself, for example by changing its direction or inducing a delay. Or they can be introduced via the virtual environment, for instance by tilting it in different orientations. That way, we could introduce a conflict between the vestibular feedback and the visual perception of gravity.

To explore gravity perception, VR is a useful tool to create subjective vertical tasks, such as explained in section 2.7. For instance, we could isolate the visual feedback by tilting the environment while the subjects remain straight. Or on the contrary, isolate the vestibular feedback by tilting the subjects with the rotating chair while the VE remains aligned with gravity.

In the field of dexterous manipulation, sensory conflicts could also be created with haptic feedback. For example, modifying the visual representation of the weight of an object is very easy in VR. It could be done by changing its size or its texture to play with its apparent density. By still modifying the object's appearance, we could create a visual clue of slipperiness, for example by displaying the manipulandum as a block of soap. Another manner of inducing conflicts with the haptic feedback could be to display obstacles in the range of motion where there is none in the real world, suggesting to the subject that an impact will happen while it is not true.

Moreover, our installation could be used in other fields of research. For now, we only described experiments where the subject is static. However, as the motion-tracking cameras have a wide range of detection and several of them can be used at the same time, we could perform experiments where the subject needs to move in the virtual environment. This could allow researchers to explore the sense of spatial orientation in different situations and study the influence of parameters such as ageing.

In addition to all these ideas, this interface could help in the comprehension of neurodegenerative diseases (Alzheimer's, Parkinson's, etc), or for people who have faced a serious accident (stroke, etc).

8.4 Conclusion

In addition to being the final product of our master thesis, we hope that this document can provide a solid groundwork for future students and researchers that will use our equipment. We are looking forward to seeing how it will be integrated into the research work of the COSY lab and what discoveries can be done with it.

Bibliography

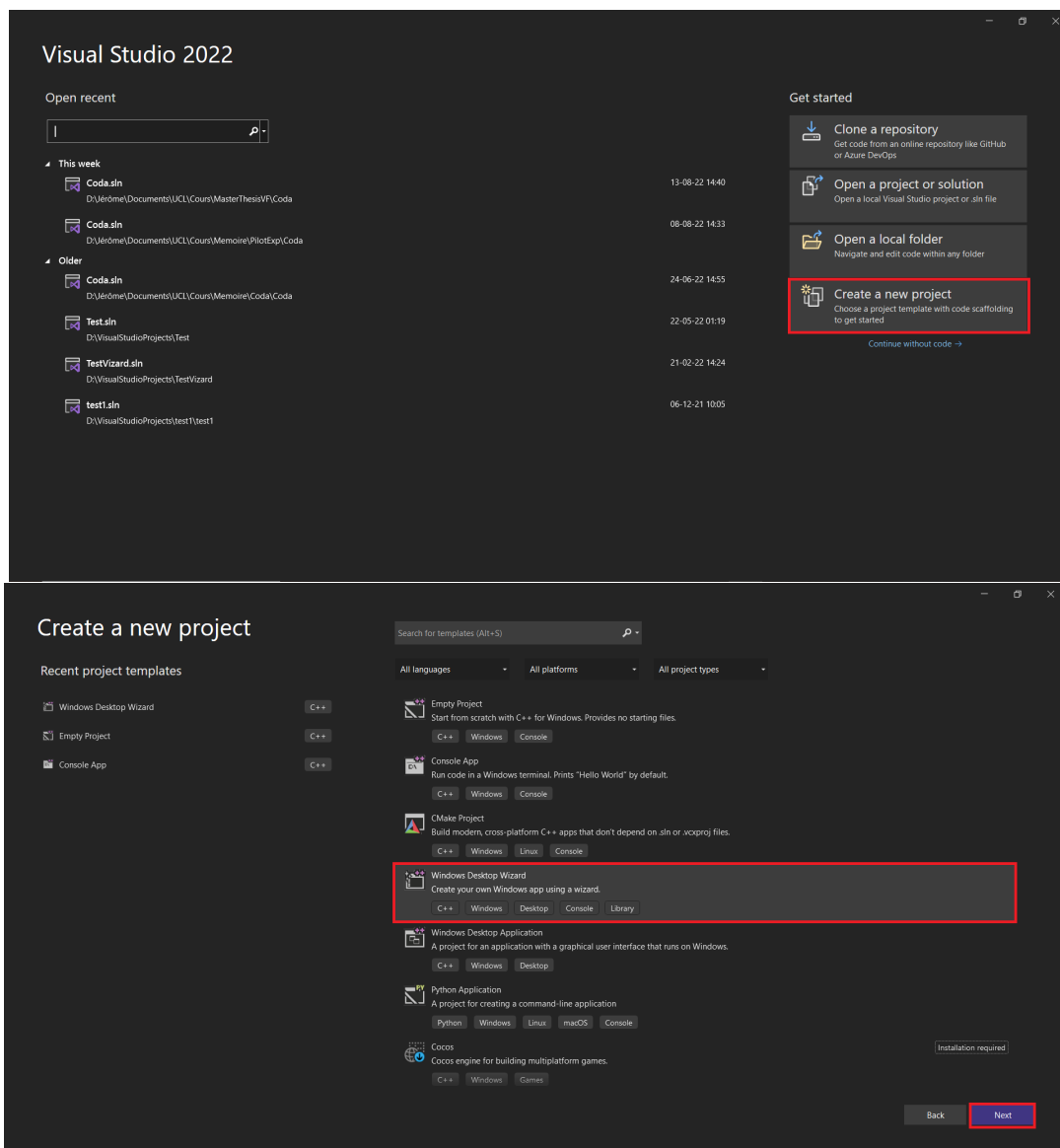
- [1] Benyamin Amiri, *Four forces of flight: Forces acting on a plane: Homa pilot*, Apr 2021.
- [2] Anne-Sophie Augurelle, Allan M. Smith, Thierry Lejeune, and Jean-Louis Thonnard, *Importance of cutaneous feedback in maintaining a secure grip during manipulation of hand-held objects*, *Journal of neurophysiology* **89** (2003), no. 2, 665–671.
- [3] Codamotion, *3d measurement*, <https://codamotion.com/3d-measurement/>.
- [4] Desirée Colombo, Silvia Serino, Cosimo Tuena, Elisa Pedroli, Antonios Dakanalis, Pietro Cipresso, and Giuseppe Riva, *Egocentric and allocentric spatial reference frames in aging: A systematic review*, *Neuroscience & Biobehavioral Reviews* **80** (2017), 605–621.
- [5] Wikipedia contributors, *Internal model (motor control)* — *Wikipedia, the free encyclopedia*, 2022.
- [6] ———, *Motor control* — *Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Motor_control&oldid=1064831277, 2022, [Online; accessed 10-August-2022].
- [7] ———, *Virtual reality* — *Wikipedia, the free encyclopedia*, 2022, [Online; accessed 27-May-2022].
- [8] Christopher J. Dakin and Ari Rosenberg, *Gravity estimation and verticality perception*, *Handbook of clinical neurology* **159** (2018), 43–59.
- [9] Brian L Day and Richard C Fitzpatrick, *The vestibular system*, *Current biology* **15** (2005), no. 15, R583–R586.
- [10] J. Randall Flanagan, Sara King, Daniel M. Wolpert, and Roland S. Johansson, *Sensorimotor prediction and memory in object manipulation.*, *Canadian Journal of Experimental Psychology/Revue canadienne de psychologie expérimentale* **55** (2001), no. 2, 87.
- [11] J. Randall Flanagan and Alan M. Wing, *The stability of precision grip forces during cyclic arm movements with a hand-held load*, *Experimental brain research* **105** (1990), no. 3, 455–464.

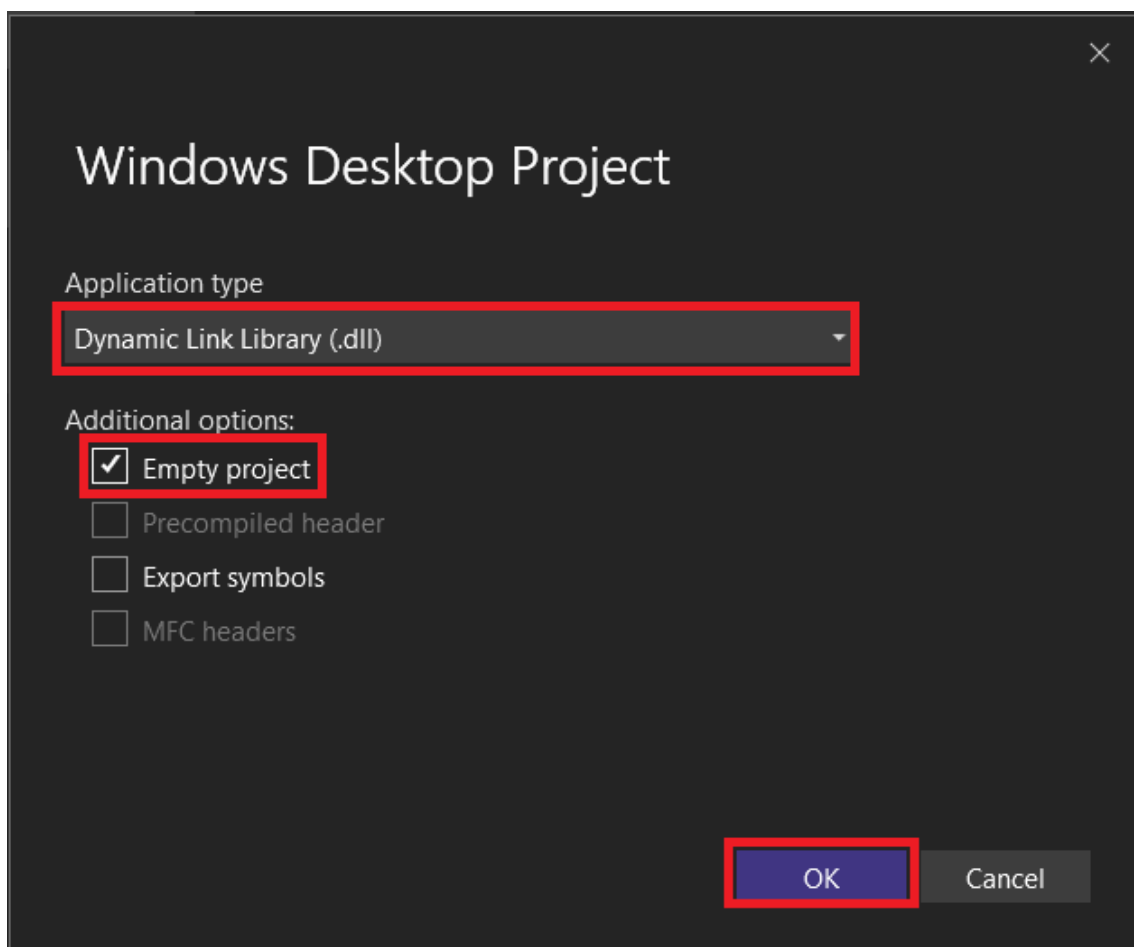
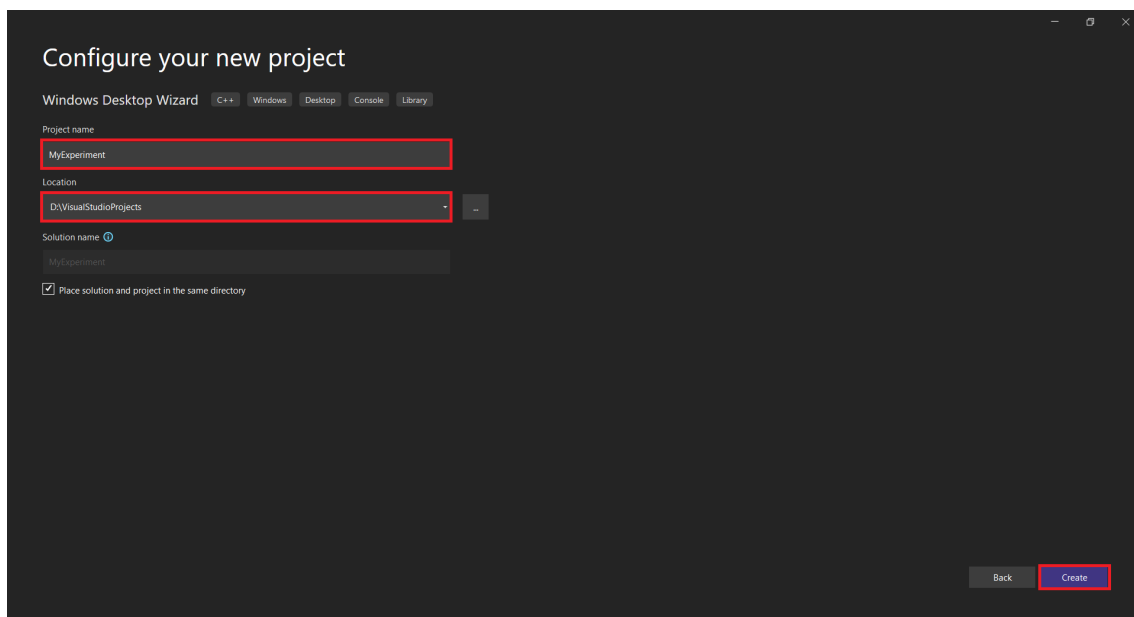
- [12] Air Zero G, *How zero-gravity parabolic flights work*, <https://www.airzerog.com/zero-g-flights-how-it-works/>.
- [13] Nikolaus Gebhardt, *Irrlicht documentation*, <https://irrlicht.sourceforge.io/>.
- [14] Francis M. Grover, Sarah M. Schwab, and Michael A. Riley, *Grip force-load force coupling is influenced by altered visual feedback about object kinematics*, *Journal of Motor Behavior* **52** (2020), no. 5, 612–624.
- [15] Mitsuo Kawato, *Internal models for motor control and trajectory planning*, *Current opinion in neurobiology* **9** (1999), no. 6, 718–727.
- [16] Anne B. Le Seac’h and Joseph McIntyre, *Multimodal reference frame for the planning of vertical arms movements*, *Neuroscience letters* **423** (2007), no. 3, 211–215.
- [17] Rodrigo S. Maeda, Tyler Cluff, Paul L. Gribble, and J. Andrew Pruszynski, *Feedforward and feedback control share an internal model of the arm’s dynamics*, *Journal of Neuroscience* **38** (2018), no. 49, 10505–10514.
- [18] Jaime Leonardo Maldonado Cañón, Thorsten Kluss, and Christoph Zetsche, *Adaptivity of end effector motor control under different sensory conditions: Experiments with humans in virtual reality and robotic applications*, *Frontiers in Robotics and AI* **6** (2019), 63.
- [19] Joseph McIntyre, Myrka Zago, Alain Berthoz, and Francesco Lacquaniti, *Does the brain model newton’s laws?*, *Nature neuroscience* **4** (2001), no. 7, 693–694.
- [20] Meta, *Oculus quest 2*, <https://store.facebook.com/fr/fr/quest/products/quest-2>.
- [21] ———, *Oculus quest 2 tech specifications*, <https://store.facebook.com/fr/fr/quest/products/quest-2/tech-specs#tech-specs>.
- [22] Laurent Opsomer, Frédéric Crevecoeur, Jean-Louis Thonnard, Joseph McIntyre, and Philippe Lefèvre, *Distinct adaptation patterns between grip dynamics and arm kinematics when the body is upside-down*, *Journal of Neurophysiology* **125** (2021), no. 3, 862–874.
- [23] Laurent Opsomer, Vincent Théate, Philippe Lefèvre, and Jean-Louis Thonnard, *Dexterous manipulation during rhythmic arm movements in mars, moon, and micro-gravity*, *Frontiers in Physiology* **9** (2018), 938.
- [24] Simone Toma, Veronica Caputo, and Marco Santello, *Visual feedback of object motion direction influences the timing of grip force modulation during object manipulation*, *Frontiers in human neuroscience* **14** (2020), 198.
- [25] John C Tuthill and Eiman Azim, *Proprioception*, *Current Biology* **28** (2018), no. 5, R194–R203.

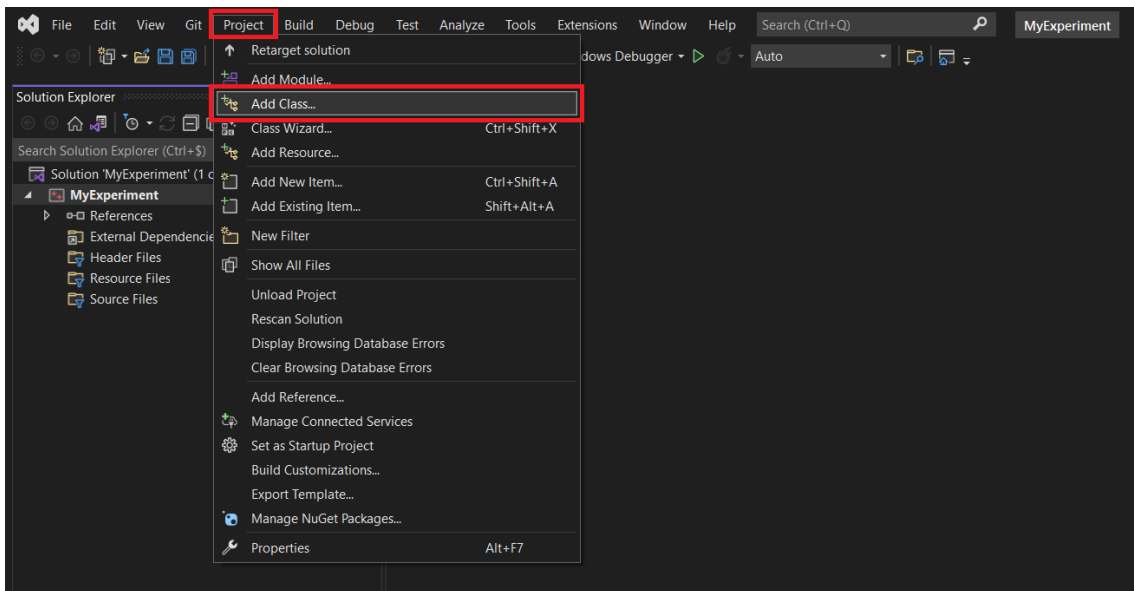
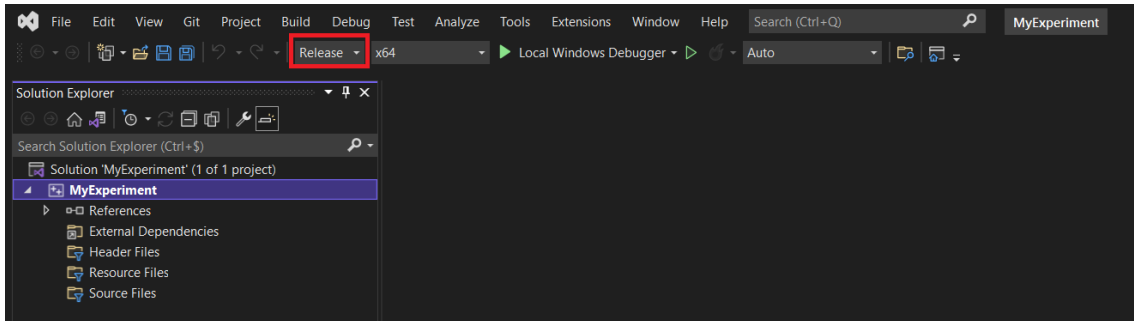
- [26] Vonne van Polanen, Robert Tibold, Atsuo Nuruki, and Marco Davare, *Visual delay affects force scaling and weight perception during object lifting in virtual reality*, Journal of neurophysiology **121** (2019), no. 4, 1398–1409.
- [27] Olivier White, Joseph McIntyre, Anne-Sophie Augurelle, and Jean-Louis Thonnard, *Do novel gravitational environments alter the grip-force/load-force coupling at the fingertips?*, Experimental brain research **163** (2005), no. 3, 324–334.
- [28] Daniel M. Wolpert, *Computational approaches to motor control*, Trends in cognitive sciences **1** (1997), no. 6, 209–216.
- [29] Daniel M. Wolpert, Zoubin Ghahramani, and Michael I. Jordan, *An internal model for sensorimotor integration*, Science **269** (1995), no. 5232, 1880–1882.
- [30] WorldViz, *Vizard / Virtual Reality software for researchers*, <https://www.worldviz.com/vizard-virtual-reality-software>.
- [31] ———, *Vizard documentation*, <https://docs.worldviz.com/vizard>.

Appendix A

Create a new project

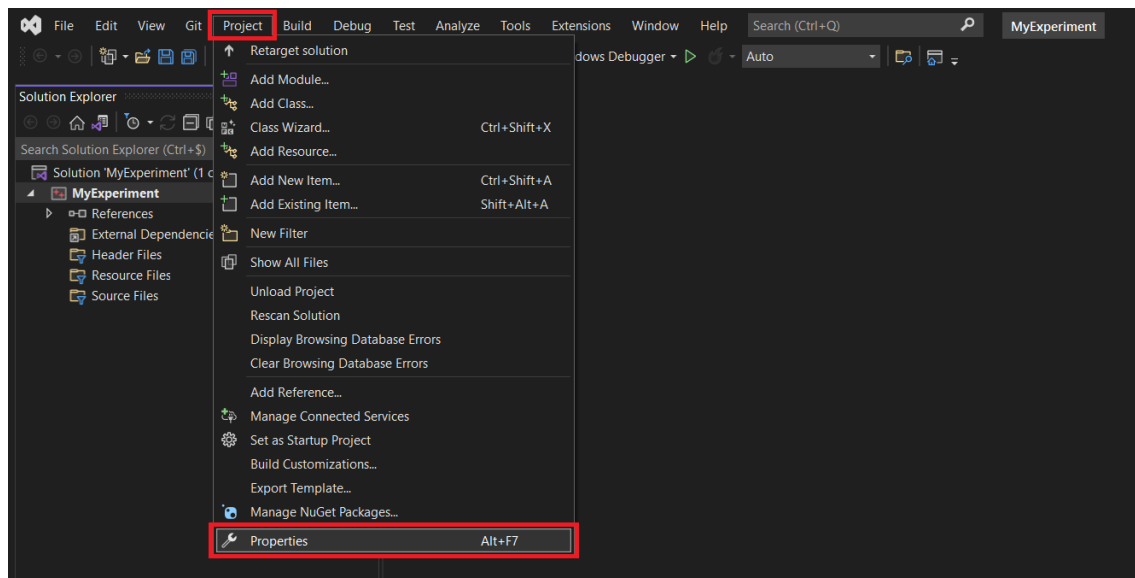






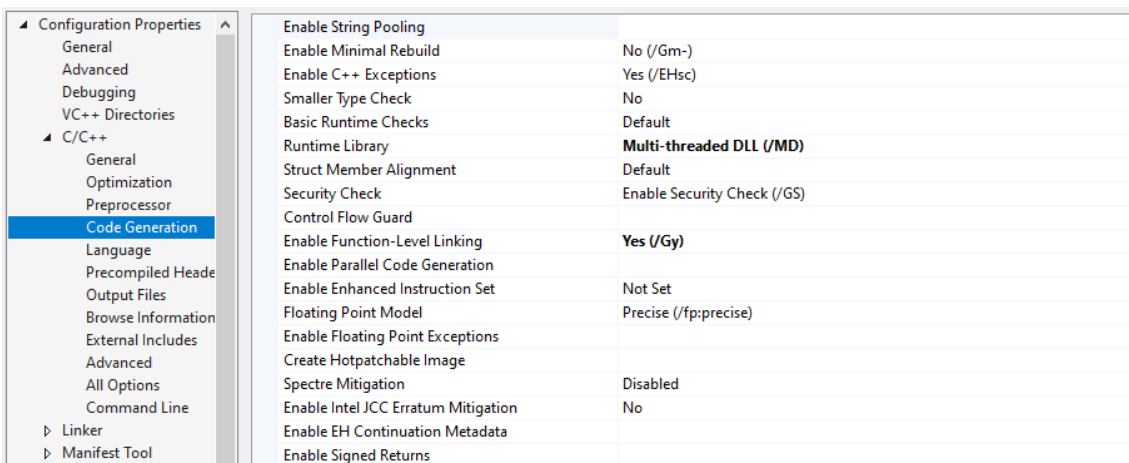
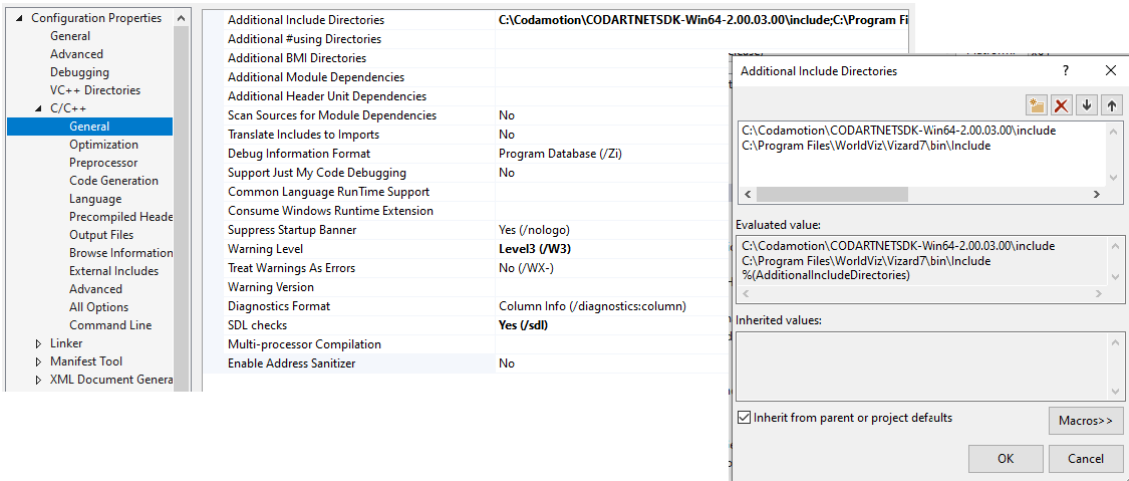
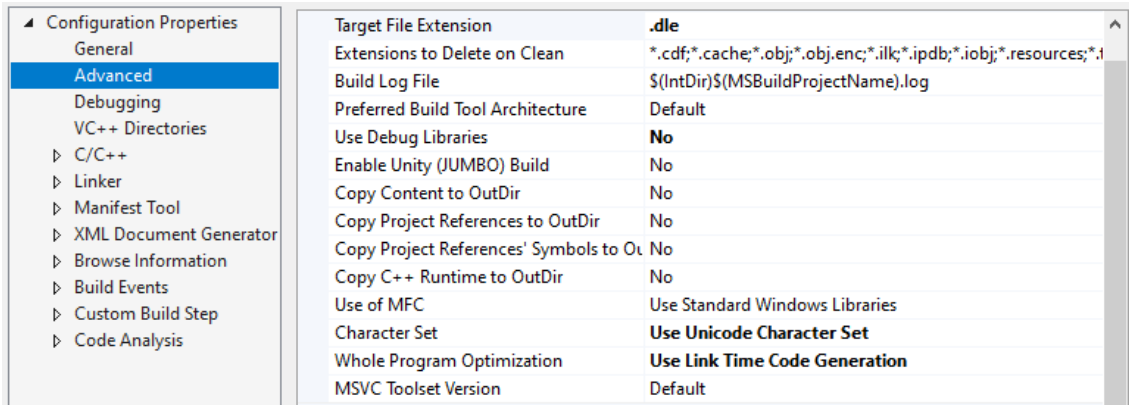
Appendix B

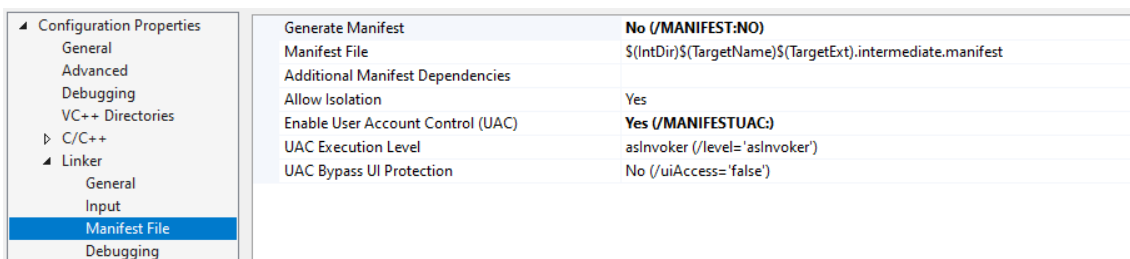
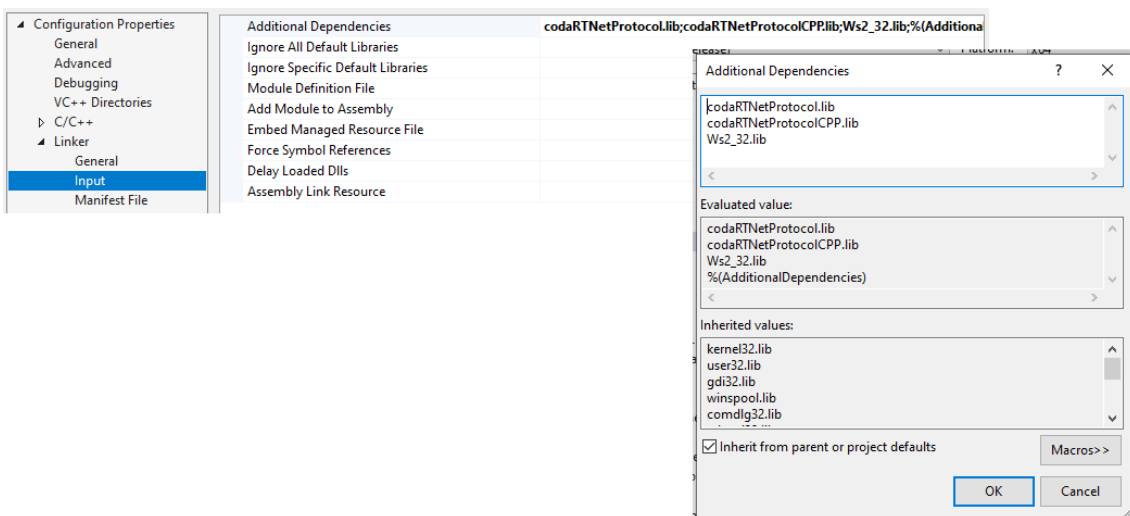
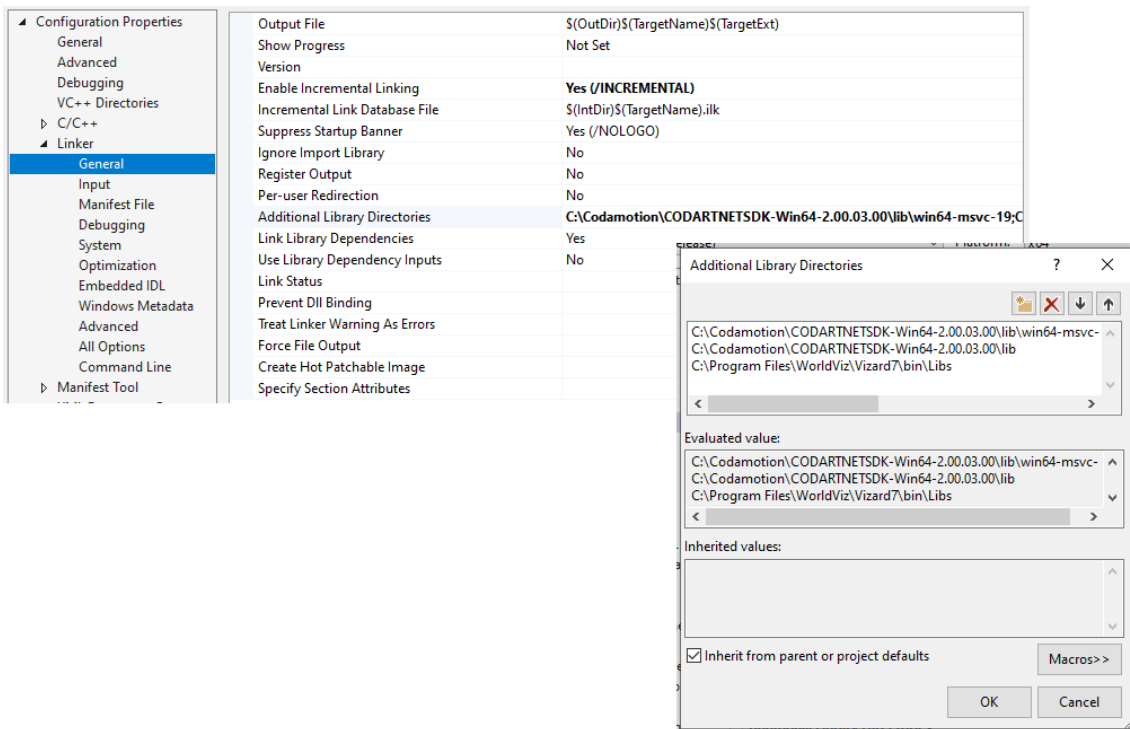
Project settings



Configuration Properties	
General	
Advanced	
Debugging	
VC++ Directories	
C/C++	
Linker	
Manifest Tool	
XML Document Generator	
Browse Information	
Build Events	
Custom Build Step	
Code Analysis	

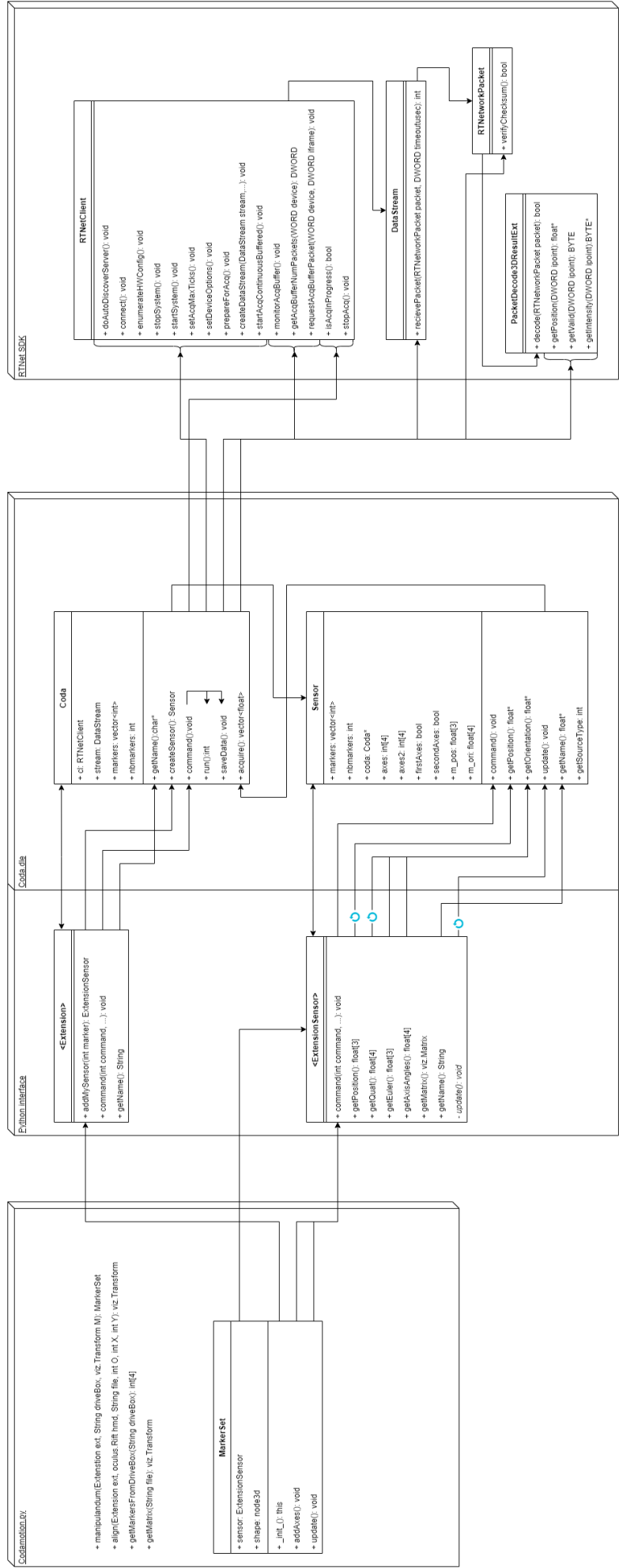
General Properties	
Output Directory	MyExperiment
Intermediate Directory	\$(Platform)\\$(Configuration)\
Target Name	\$(ProjectName)
Configuration Type	Dynamic Library (.dll)
Windows SDK Version	10.0 (latest installed version)
Platform Toolset	Visual Studio 2019 (v142)
C++ Language Standard	Default (ISO C++14 Standard)
C Language Standard	Default (Legacy MSVC)





Appendix C

Complete diagram of the interface



UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl