SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE ANTONIO RUBERTI (DIAG)

Medical procedures on virtual patients

MEDICAL ROBOTICS

Professor:
Marilena Vendittelli

Students:
Luca Distefano,
Anna Fumagalli,
Giuliano Giampietro,
Jamila Naffati,
Catia Romaniello,
Vincenzo Maria Vitale

Academic Year 2022/2023

Contents

Abstract

Simulating a virtual environment for research purposes is gaining progressively more consideration among the scientific community. Particularly, on medical branch it has the potential to be an effective tool for surgical tracking and planning, introducing digital twins of the human body on which simulating the previously named procedures. Moreover, thanks to the development of new haptic devices, the virtual experience becomes increasingly realistic.

In the case of the project detailed below, the aim is to interact with virtual objects- specifically on the 3-D model of a section of the abdomen- getting the tactile feeling of palpation through Weart TouchDIVER haptic device.

1 Introduction

Frenetic technological advancement invests every field, and the medical field is no exception bringing increasingly better patient outcomes. Part of the development mentioned before revolves around training strategies, surgical planning, as well as image diagnosis, including the use of digital phantoms.

The phantom referenced is a 3-D model that represents a *digital twin* of a human body, realized using advanced imaging techniques such as computed tomography (CT) and magnetic resonance imaging (MRI). Those virtual anatomical models mimic the appearance and behavior of tangible human tissues and organs.

Simulating medical procedures in a virtual - and controlled- environment in which those realistic models are integrated, allows surgeons to plan and practice surgeries safely without endangering the patient, exploring different medical techniques. In addition, it consents them to train repeatedly on complex operations until they feel confident enough to perform those operations on real patients and, as it follows, can be of support also for educating new medical professionals. The surgery can thus become safer, more efficient, with reduced prognosis, or more manageable for surgeons with different levels of experience, including would-be surgeon students.

To help medical professionals understand complex anatomical structures and visualize potential challenges that may arise during a procedure, Virtual Reality (VR) can come to the rescue. Virtual Reality technology is a computer-generated environment that simulates a real or imagined experience for the user. It is typically experienced through a headset or other immersive devices that provide a sense of presence in a virtual environment. The sense of immersion can enhance the user's experience and improve learning outcomes. Virtual reality has had enormous developments since its inception, and it continues to evolve with the advancements of hardware, software, and applications. This compelling technology has the potential to revolutionize the

way we experience the world around us, and its continued development promises to provide exciting opportunities for entertainment, education, training, and healthcare.

Due to the immersive experience VR offers, it is well-suited for medical practices in a 3D virtual environment with a digital phantom in it.

The smooth interaction with virtual patients VR allows, together with more realistic simulated procedures, can help physicians refine their skills. In terms of cost-effectiveness and accessibility, this technology to train medical professionals is preferable over traditional methods.

Furthermore, virtual reality provides professionals with simulated scenarios that seldom occur in real-life situations, such as rare or complex medical conditions. Training on a broader range of possibilities gives the specialists dexterity and adaptability, ultimately leading to better patient outcomes.[1]

It is in this context that the project is framed: the goal is to interact with a digital phantom of a section of the abdomen - including its internal organs and tissues- in a 3D virtual environment, receiving haptic feedback on the physical hand through the three thimbles of the Weart TouchDIVER haptic device 3.2.

In general terms, haptic devices are hardware tools that make the experience of the interaction between the user and the digital phantom more immersive, providing sensory feedback that mimics real-world interaction as a sense of touch. In the medical field, they find use in both simulations and actual medical procedures. The motivations behind using the mentioned devices in simulations are straightforward and concern their realism. But they also find a place during effective operations, for example, in laparoscopic procedures, during which the surgeon works with small instruments and cameras to operate on the patient through small incisions. The haptic feedback can therefore assist the surgeon in sensing the texture and strength of the patient tissues, providing a more accurate sense of the position of the instruments and the pressure to be applied[2].

Indeed, the environment the project under analysis fits has to be considered *Mixed* rather than exclusively virtual. *Mixed reality* merges virtual and real environments with the tactile feedback haptic devices provide: this combination offers multiple benefits for various applications, especially for medical issues. For instance, it grants realistic training for medical students who can receive back from the virtual patient -or phantom -sensory feedback that simulates real-world scenarios. Moreover, physiotherapy and rehabilitation settings exploit *Mixed Reality* simulating activities such as lifting weights or performing exercises during which users sense resistance and forces.

As mentioned above, the haptic device adopted in the project is the TouchDIVER of Weart[3]. It is a haptic glove that connects digital and physical worlds by adding tactile sensation to living the VR experience in an immersive manner so that the

returned cue is as close as possible to the real one. This tool involves collaboration with a VR Headset.

The device returns forces, vibration, and thermal signals through the three thimbles[1] that are an integral part of it.

As a virtual reality (VR) headset, the choice fell to the Oculus Rift S developed by Facebook [4]. It has inside-out tracking, meaning that the sensors are in the headset itself, and no external ones that need settings in the room are required. It uses five cameras to track the movement of the hands and head of the user and comes with two touch controllers that allow for precise hand tracking and intuitive interaction with virtual objects. Regarding the simulation software, the project develops in CoppeliaSim [5] powered by COPPELIA Robotics, formerly V-REP (Virtual Robot Experimentation Platform). CoppeliaSim, due to its specialization in robotics, is mainly used to simulate robots and environments - including complex robotics systems- for research, education, and industrial purposes. Probably the main plus of this software is the wide range of robot models it involves, including industrial, humanoid, and wheeled ones.

Finally, a review of the objective. The closure of the robotic hand- also developed in the project and detailed in the body of the report- is controlled by the TouchDIVER device, while the position of the physical wrist is mapped thanks to the tracking of the VR Visor controller (positioned as shown in Figure 1).

Figure 1: Pairing the TouchDiver with Oculus controller

The objective tasks to be performed by the hand of the interaction with the digital phantom of an abdomen [2] are mainly three:

1. Palpation: the aim is to perform a grasp of objects -with the capability of moving them- that belong to a digital phantom of an abdomen, receiving force feedback returned as pressure on the thimbles of the haptic device;

[1]The thimbles are to be worn on the thumb, index, and middle fingers

[2]The *digital phantom* of an abdomen used for the simulation has recognizable tumor lesions

2. Texture feeling: objects with different textures are wanted to be returned to the physical hand via the haptic device with vibration at different frequencies based on the surface that is touched;

3. Temperature feeling: organs can take on slightly different temperatures but, above all, some types of tumors appear warmer to the touch[6]. Therefore, this project considers it worthwhile to obtain haptic temperature feedback.

The project in question spans a context in which medical-technological research is advancing: several articles testifying to the cohesive use of haptic devices and virtual reality.

To cite a recent article, haptic devices, *Augmented Reality* (AR)[3], and artificial intelligence helps reduce physical contact in medical training during the COVID-19 pandemic [7].

Moreover, the article *A Review of Simulators with Haptic Devices for Medical Training*, by Escobar-Castillejos et al.[8] explores and reviews the use of haptic simulators for medical training. This paper aims to provide a state-of-the-art review of the latest medical simulators that use haptic devices, focusing on stitching, palpation, dental procedures, endoscopy, laparoscopy, and orthopedics. These simulators are reviewed and compared from the viewpoint of the used technology, the number of degrees of freedom, degrees of force feedback, perceived realism, immersion, and feedback provided to the user. In particular, the article emphasizes potential benefits regarding haptic devices used with Virtual Reality in palpation training for medical professors, such as a more consistent and standardized experience, reducing the chance of injury to the patients, and allowing trainees to improve their skills. Nevertheless, it reports some limits to the use of haptic devices and VR for palpation training, such as the cost and limitations of current technology, and cites several simulator environments specific for palpation training.
As mentioned above, a few medical simulation platforms that use haptic technology and VR are available, but these are specialized and limited in content. On the other hand, the CoppeliaSim environment chosen for this project is a multi-purpose simulation platform that can be customized to simulate a wide range of medical -and robotics -scenarios and procedures. Moreover, CoppeliaSim open-source architecture makes it more accessible and flexible than some proprietary simulation platforms, meaning that researchers and developers can customize and develop it to meet their specific needs.

Another innovation that CoppeliaSim may introduce for medical training and

[3]Augmented reality or computer-mediated reality refers to the enrichment of human sensory perception by information, usually electronically manipulated and conveyed, that would not be perceivable with the five senses.

education is the ability to provide feedback on the learner's performance. Thanks to sensors and motion tracking, the platform can provide real-time feedback on the trainee's accuracy, speed, and technique, letting them enhance their skills over time.

2 Theoretical Background

This chapter introduces basic concepts to better frame the project and answer some of the questions that arose when developing it. Typical fingertips trajectories are inspected at first. Secondly, robotics notions about inverse kinematics are described. An analysis of the human perception of haptic feedback and how it can be implemented in devices, closes this chapter.

2.1 Finger mechanics: kinematics and tracking

Deciding how to map TouchDIVER tracking data to the position of the fingers of the virtual hand represented a challenge. The device returns a normalized float value for each fingertip that characterizes the *time of flight* needed to go from the fully open configuration -corresponding to a return value of zero- to the closed one -giving one instead. As the Weart team did to implement its demonstrations, the chosen method is to assign a desired path to each fingertip.

Some scientific articles were reviewed to choose an appropriate shape for the curve generated by hand closure. The authors examined the typical paths described by the fingertips while reaching for an object, grasping it, and other related operations. The task that most assumes importance in the project is grasping. A specific article was crucial for defining the opportune path: Stereotypical Fingertip Trajectories During Grasp [9]. In this work, Kamper et al. analyzed the kinematics of movement of the digits during reach-and-grasp tasks for a collection of five objects of different volumes across twenty trials on a sample of ten healthy subjects. The position of the fingertips was determined- after recording joint angles in motion - by applying direct kinematics. As the authors state, the infinite number of trajectories that might be pursued by moving a finger from one point in space to another makes the hand the pivot for a wide range of everyday actions. This freedom carries a greater complexity of the needed control scheme, as a trajectory among the possible ones must be adopted and executed at a very high rate in real-time applications.

Even admitting the simplification introduced by assigning a desired trajectory to each fingertip, this is still a widely used choice, also employed by Calle et al. in *Design and kinematic analysis of a biphalange articulated finger mechanism for a biomechatronic hand prosthesis*[10].

More importantly, if the tasks performed are not too different from each other is reasonable to assume that the paths will also not differ much, as illustrated by the figure 2.

That is an example of the index fingertip location across twenty trials for one subject. The data are in polar coordinates; the concentric circles indicate loci of equal length r (cm); straight lines indicate loci of equal angle θ. The metacarpophalangeal

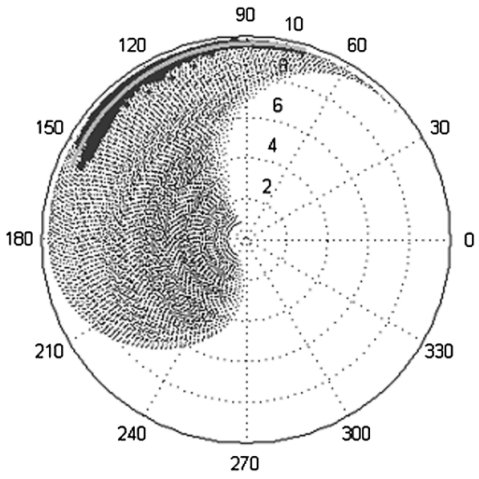

Figure 2: Example of the index fingertip location across all 20 trials for 1 subject. Lighter dots represent the extent of the potential workspace in which the fingertip could move. From [9]

joint is at r=0, with $\theta = 90$ in the neutral position. Lighter dots represent the extent of the potential workspace into which the fingertip could move, while darker dots represent the actual fingertip locations. The solid line represents the spiral fit to data (mean error = 0.1 cm). The trajectory chosen in this project to approximate the

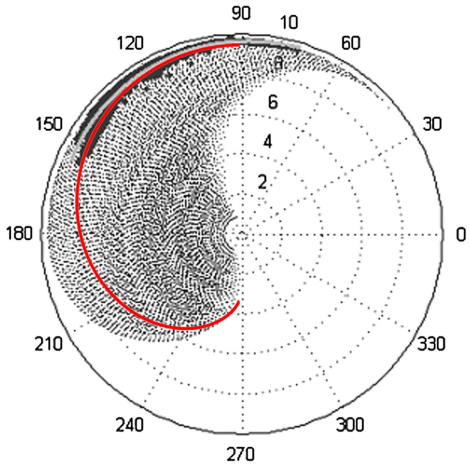

Figure 3: Overlapping of the cardioid function with the trial from the previous graph.

finger trajectory in grasping is given by the half of a cardioid function [4]. This function overlaps almost perfectly with the solid line in Figure 3, drawn as a red line in Figure 2.

Solidifying the thesis is the finding of Kamper et al.[9], who identified consistent patterns in finger trajectories during grasping movements in humans. Therefore the authors suggest that these trajectories may be generated by a stereotyped neural

[4]In geometry, the cardioid is a curve, specifically an epicycloid with one and only one cusp. It is shown in Figure 4

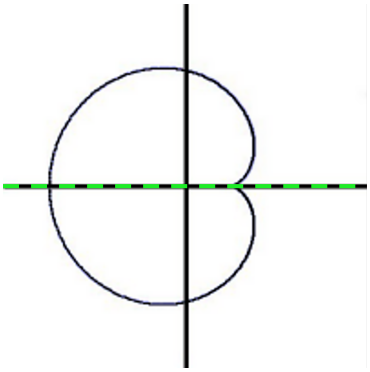

Figure 4: A cardioid, an epiclycloid with one and only one cusp

control strategy, supporting that movements in the grasping phase can be planned using stored motor planes.

In support of the same thesis, in the article *Hand synergies during reaching and grasping*, Mason et al. (2001)[11] found that during monkey grasping movements, a small number of hand synergies account for most of the variance in hand posture. Thus, the authors reflect on how these hand synergies may be the result of neural control mechanisms that simplify the control of hand movements.

However, as Santello et al. (2013) note in the almost eponymous article *Neural bases of hand synergies*[12], sensory feedback also plays a crucial role in fine-tuning grasping movements. The brain receives information about the size, shape, weight, and texture of the objects through sensory receptors placed in the skin and muscles, which regulate ongoing motor commands to ensure that the hand and fingers move and position themselves appropriately to grasp the object.

Considering these results, grasping movements might be generated using a combination of stored motor plans and sensory feedback, although stored motor planes can produce approximate finger trajectories since they are then fine-tuned based on sensory information. Thus, trajectories can simulate those movements with a good approximation with respect to what brains do in grasping phases.

2.2 Inverse Kinematics

From a mechanical point of view, a manipulator is a kinematic chain of rigid bodies (the links) connected through prismatic or revolute joints. In open-chain structures, one extreme is constrained to a base, while an end-effector is typically attached to the other end; the composition of the elementary motions of consequent links results in the movement of the structure. Descriptions of the end-effector position, orientation, or other related task coordinates are needed for kinematics purposes: these quantities are a function of the joint variables and are expressed in the most convenient reference frame. Direct kinematics handles this matter. Inverse kinematics is instead a problem of a more complicated nature, as it aims to determine the joint variables that correspond to given operational space coordinates. These configurations must be feasible for the robot and thus belong to its workspace. The solution to this problem allows transforming the motion specifications assigned to the end-effector in the workspace into the corresponding joint space motions that allow the execution of the desired movements. As previously mentioned, the inverse kinematics problem is much more complex than the direct one for the following reasons:

- The closed-form solution is not always possible since the equations to solve are generally non-linear;
- multiple or even infinite solutions may exist (redundant case);
- there might be no admissible solutions (the end effector position and orientation must belong to the manipulator workspace) [13].

When a manipulator has N degrees of freedom, which is larger than the number of variables needed to describe a given task (M), it is said to be redundant for that specific assignment. Redundancy is a concept relative to the task assigned to the manipulator [13]. Nevertheless, it is possible to take advantage of these extra degrees of freedom to achieve more robot motions, avoid obstacles or kinematic singularities, respect joint ranges, or minimize energy consumption or needed motion torques [14]. In the case of redundancy, the inverse kinematic problem admits infinite solutions, so there are several methods of solving it by optimizing an objective function, including Jacobian-based methods. Those are based on choosing one of the infinite solutions, especially the one that minimizes the task error norm [5]

$$||\dot{r} - J(q)\dot{q}||^2 \tag{1}$$

Null-space method, instead, has an additional term in the solution that belongs to the null space of the jacobian so that it does not affect the execution of the task trajectory

[5] If the task is feasible, no task error occurs, and minimization of joint velocities is obtained [15].

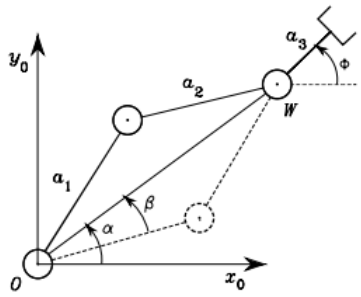

Figure 5

(self-motion). This allows to possibly satisfy further criteria without influencing the desired trajectory. In the case of the Jacobian-based method, it means finding a solution in the form $\dot{q} = J^{\#}\dot{r}$, which in this case is not immediate because the matrix J is not square (redundancy case: MxN), consequently, the inverse cannot be applied, but the concept of pseudo-inverse comes into play. The pseudo-inverse joint velocity is the only one that minimizes the norm 1. However, issues may occur in the case of proximity to a singularity configuration, where excessively large $\dot{q}$ are required even for small $\dot{r}$. These joint velocities are not feasible and would be dangerous for the robot itself (actuators) [15]. The result of this potential problem is the *Jacobian Damped Least Squared (DLS)*, which induces a more robust behavior when near a singularity. In this case, the objective function undergoes a modification in that the parameter μ (damping factor) is added, yielding a solution with the following expression:

$$\dot{q} = J^T(JJ^T + \mu^2 I_M)^{-1}\dot{r}, \tag{2}$$

where μ can be constant or modified online(i.e., taking in consideration the distance from a singularity)

- $\mu = 0$ when one is far from the singularities (returning to the pseudo inverse case);
- $\mu > 0$ when near or right at the point of singularity, allowing to decrease in the joint velocity (*damped*) while executing the end-effector velocity with an error [16].

In this work, each finger of the virtual hand is, kinematically, a planar robot with three rotational joints. Only three out of five fingers can be actuated independently, yielding nine DOFs in total. Six more degrees of freedom are specific to the hand, which is free to move in space and represents the mobile base. Each finger follows a target on a path via a CoppeliaSim function that implements inverse kinematics, as developed in the chapter 2.1 and 3.3.3.3.

2.3 Haptic Feedbacks

The word haptics refers to the capability to sense a natural or synthetic mechanical environment through touch [17]. Making the user experience as immersive as possible is a focal goal of Virtual and Augmented Reality, which attracts efforts from a wide variety of fields, starting from the medical domain, of which literature is unanimous in assessing the usefulness of virtual reality, up to the video-game industry [18]. Although necessary, a simple first-person view rendered by a head-mounted display (HMD) is insufficient to achieve such immersion. In this scenario, haptic feedback can help by enriching the user experience, including the perception of texture, shape, weight, softness, and temperature. A virtual scene is thus rendered realistic through sensory stimuli that imitate real sensations as closely as possible. The wide definition of haptic spans from simple devices used daily to more sophisticated devices capable of returning a collection of types of feedback that can be hugely different. Haptic feedback can be divided into two macro-categories, which differ in how different devices are perceived by the user, rather than by a difference in the technology used.

2.3.1 Kinesthetic Feedbacks

Kinesthetic feedback refers to the sense of position and motion of the user state (kinesthesia) mediated by a variety of receptors, called mechanoreceptors[6], located in the skin, joints, skeletal muscles, and tendons. They simulate contact with an object by imitating the sensations of pressure and weight an object can take on. This type of feedback does not act on the user's sense of touch, merely stimulating nerves located on the skin, but allows the stimulus to spread to the muscle, requiring a shift to the movement imposed by the user [20]. Force feedback ideally returns resistance to the user to the movement exerted on the haptic device, being detected by mechanoreceptors in muscles and tendons. Most of these devices send force feedback signals through electric servomotors that limit the user's movement. By simulating the reaction force that a stationary object would impose, these devices return feedback that could be easily associated with contact with entities existing only in the virtual world. Commercially, devices capable of returning force feedback are characteristic of the video game industry, especially in *grounded* controllers such as steering wheels or cloches.

Force feedback is returned by different types of technologies: the more traditional ones were initially developed to overcome issues from aircraft control equipment that no longer supported mechanical feedback as it was electrically controlled. These systems

[6]Mechanoreceptor is a receptor sensitive to mechanical forms of stimuli. Examples of mechanoreceptors are *the receptors in the ear that translate sound waves into nerve impulses, the touch receptors in the skin, and the receptors in the joints and muscles* as described by American Psychological Association in their Dictionary of Psychology [19].

involve high-voltage servomotors coupled with high-resolution torque sensors (load cells) and low backlash gear to create a closed-loop system [20]. Other systems on the market as electromagnetic feedback interacts with the user through magnetic fields. In the case of wearable dispositive, a real force-feedback is only possible when anchored externally or concerning two separate limbs[7]. The figure 6 depicts different grounding of devices delivering force-feedback.

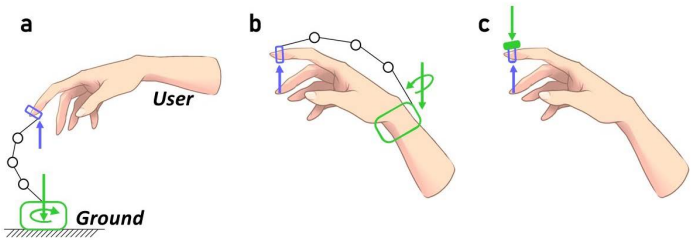

Figure 6: Schematics of different groundings of force feedback haptic devices. (a) world-grounded haptic device, (b) body-grounded kinesthetic device, (c) finger-worn cutaneous device. From [17].

2.3.2 Cutaneous and tactile feedback

Many haptic systems rely on skin surface-based interactions to deliver haptic feedback. These are feedbacks that simulate the user's sense of touch, and they refer to the spatial distribution of pressure in the region of touch and are sensed by skin mechanoreceptors[21]. Although cutaneous feedback can be in principle provided for the whole body, it is mostly given through fingertips, as these are usually employed for grasping and manipulation, and are rich in mechanoreceptors.
In literature, cutaneous and tactile feedback devices fall into three different categories: skin devices, active deformable surfaces, and mid-air haptics [18]. In contrast to active surfaces and mid-air haptics, cutaneous devices, currently the most widespread category of haptic devices, provide vibration, temperature, and texture feedback through direct contact with non-deformable surfaces. The most popular haptic wearable devices, including TouchDIVER developed by Weart, make extensive use of this mode of signal transmission[18].
For a complete description of how Weart works, refer to section 3.2. For the sake of conciseness, the following paragraph shows the specific tactile feedback that the TouchDIVER relies on.

Moving Platform

Given the highest density of skin mechanoreceptors which detect local surface orientation and skin stretch [17], finger-worn devices focus the stimulation perceived by the user

[7]NB., not to be confused with pressure actuators such as those found in fingertip wearables [17].

on the fingerpads. In fact, in the case of the device used, force feedback can not limit the movement of the hand in space (or of the fingers to the hand) but only provide a sensation of contact through small metal plates. Despite the mentioned above distinction in real force feedback, stimulating the *mechanoreceptor* through force and torque without a properly grounded device, it has been shown that, to some extent, it is possible to compensate for the lack of kinesthesia with the modulated cutaneous force technique, without significant performance degradation [22].
Most haptic thimbles base their feedback on moving platforms, such as TouchDIVER, or shearing belt to resemble a force feedback [17]. Usually with 3 DoF, moving platforms are typically placed in the distal phalanx of the finger, placed on the volar side, compressing it between a base, fixed in the dorsal side. In contrast to the ones returning kinesthetic haptic, these devices are relatively cheap and are shrinking in dimensions and weight over time [17].

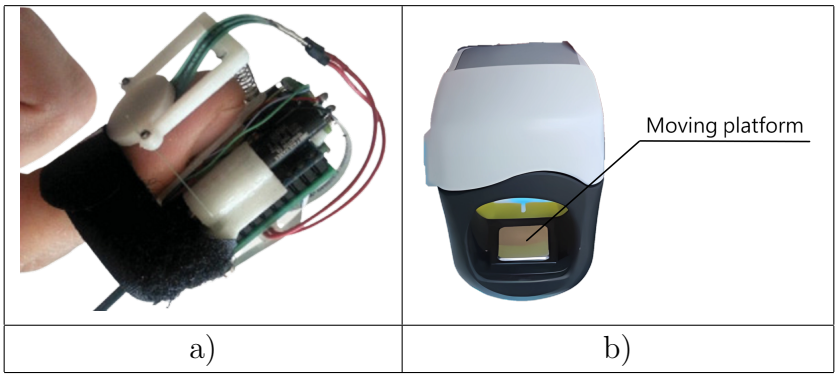

Figure 7: a) A 3 DoF finger-worn device [23], b) the TouchDiver, highlighting the moving platform.

Vibrotactile feedback

Vibrotactile stimuli are ubiquitous in the haptic domain due to the cheap and small vibrotactile actuators which allow them to be placed on the finger without compromising the user's mobility. In *Presenting Surface Features Using a Haptic Ring: A Psychophysical Study on Relocating Vibrotactile Feedback* [24], authors concluded that vibrotactile feedback returned by a device placed on the proximal phalanx manages to give the user the ability to distinguish between different surface textures. Generally, with different eccentric rotating masses per finger, these actuators module the frequency of the vibration proportional to the interaction force to give the sensation of tapping a stiff object in virtual environment [25].

Thermal feedbacks

The ability of humans to sense temperatures through touch is a critical attribute of the perception of the external environment. Thermal perception is affected by other

sensory perceptions including visual and olfactory stimuli [26]. In [27], it is shown that thermal cues are effective in enhancing the overall immersion of the user and are able to have a higher impact even than visual stimuli. Most state-of-the-art thermal displays use Peltier devices, which rely on electrical current to generate or transfer heat [28]. The great advantage of using solid-state thermistors that exploit this effect is to avoid the use of fluids and moving parts inside wearable devices that could not accommodate refrigeration-cycle equipment.

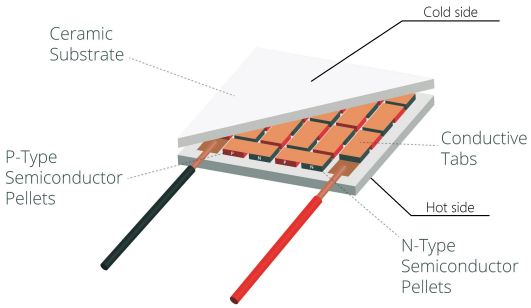

Figure 8: Schematics of a Peltier cell.

A basic schematic of the operation of a Peltier cell is shown in Figure 8, proving the compactness of the device.

2.4 Framework

In software applications, the complexity of available features and user interfaces is grown over time. Therefore, it was necessary to standardize the developing process of software applications for optimization purposes. This standardization process resides in the concept of *frameworks*. In computer science, particularly software development, a framework represents a standard for building and developing applications. As a standard, the structure of a framework imposes a precise methodology on the programmer in software development.

It is a logical supporting architecture, usually implemented as a specific *design pattern*[8], on which software can be modeled to facilitate its development by the programmer; thus represents an abstract architecture that allows the software to be modified or expanded sectorally, making it multifunctional and, in addition, several programmers can work concurrently on it. It is possible to use proprietary frameworks designed from scratch or leverage pre-existing ones, extending or combining several. The environment is universal, and versatile allowing better usability, even for large platforms. Frameworks may include support programs, compiler files, code libraries, toolsets, and application programming interfaces (APIs) that gather all the different components to enable the development of a project or system.

[8]Design pattern is a general design solution to a recurring problem in a dedicated context.

3 Tools and Methods

The present chapter describes in detail the implied hardware and software. Tracking (wrist and fingers) and feedback (visual and haptic) are achieved by the Oculus Rift S and TouchDIVER. These will be the matter of the first part of this chapter. An in-depth description of the environment configuration and framework implementation follows. Particular attention is given to what concerns the robotic hand.

3.1 Oculus

Oculus Rift S is one of the main tools used in this project, and it not only allows immersion in the virtual scene but is also remarkably accurate in motion tracking for all its components, an essential point for the integration with the TouchDIVER haptic device.

It consists of two controllers and a visor, with the following technical features:

- Display: Single Fast Switch LCD;
- Resolution: 1,280 x 1,440 per eye;
- PPI: 600;
- Refresh: 80Hz;
- Field of view: 115°;
- Interpupillary distance: managed via software;
- Connections: DisplayPort 1.2 and USB 3.0;
- Cable: 5 meters;
- Passthrough: low-latency Stereo Passthrough+ [29].

The Oculus Rift S has several improvements over its predecessors, such as a more realistic visual impact and accurate tracking. It has switched from an AMOLED-type display to an LCD, which might seem like a downgrade. On the contrary, due to the much higher resolution and refresh rate, it led to a reduction in the screen door effect (SDE) [9] and significantly improving the optical performance.

[9] The SDE is the occurrence of thin, dark liners or a mesh appearance caused by the gaps between pixels on a screen or a projected image [30].

Oculus Insight tracking system

Oculus Insight technology implements the 6-degrees-of-freedom head and hand tracking, exploiting the five front-facing cameras on the visor (two frontal, two on the side, and one on top), allowing it to detach itself from external sensors.

Oculus Insight is an innovative technology that leverages Computer Vision (CV) systems, simultaneous inertial localization, and mapping systems to calculate a precise position in real-time for both the visor - via SLAM -and the controllers- via constellation tracking. All these must be computed within limited time intervals (milliseconds) to translate the exact movements of a person in VR. Insight CV systems fuse multiple sensor inputs from the visor and controllers to increase the precision, accuracy, and response time of the position tracking of the system to achieve a smooth VR experience [31].

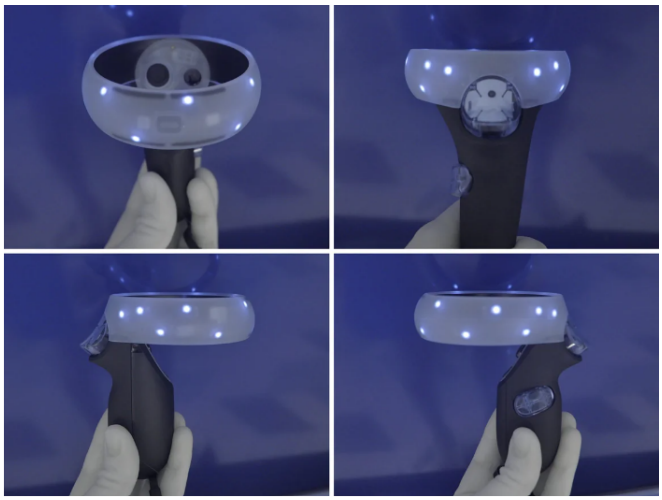

Figure 9: Oculus controller with visible infrared LEDs

Constellation tracking

The previous versions of the Oculus used the constellation tracking method for headset and controller tracking. As of the Rift S version, it is used only for controller tracking: this method takes advantage of the 22 infrared LEDs in the hardware ring- as shown in Figure 9- that are invisible to the human eye. The cameras have filters to display the IR lights that can identify the position of each of them. The software can also recognize which LED it is seeing because it knows exactly how all the LEDs that compose the constellation are positioned, in addition to the fact that each IR LED blinks at a specific frequency to identify itself. In this case, the reference sensors are the cameras on the headset, which allow the system to limit the controller position drift caused by integrating multiple IMUs[10]. Following a long series of observations,

[10]The Inertial Measurement Unit (IMU) is a device used to measure the motion of an object in three-dimensional space and is usually composed of three main sensors: an accelerometer, a gyroscope,

Oculus Insight can triangulate the 3D position of each point in the environment. Constellation advantages:

- Low cost to integrate;
- High-quality tracking;
- Works in most environments.

Constellation disadvantages:

- Each sensor has a wired connection to the PC;
- Large USB bandwidth causes issues with many motherboards;
- Sensors have a limited vertical field of view [32].

Using two different methodologies for tracking the visor and controllers presents some issues. In the previous series, all the hardware adopted the constellation tracking method, thus using an external reference camera that could detect the LEDs hidden both in the visor and on the controllers. As previously mentioned, however, in the Rift S an external camera is no longer used but directly those instantiated on the headset. It is more complicated to detect the LEDs since they are reduced compared to their predecessors (only the controller rings have them), in addition to the fact that they may be outside the camera field of view of the observer or occluded [33].

Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is a method mainly used in robotics to locate a robot within an environment: in this particular application, it is adopted for headset tracking.

As mentioned, in the Rift S version, cameras are on the visor; this is why the tracking system described is named *insight out* VR because the cameras on the headset are *in* the system and are pointed towards the outside world.

In robotics, especially for non-fixed-based robots, localization is crucial to estimate real-time robot configurations based on how it moves in the environment. Generally, landmarks are used, which can be either artificial or natural, meaning objects already in the surroundings, such as doors or windows. Since the position of these landmarks is known, the poses of the robot in the environment can be computed thanks to

and a magnetometer. The accelerometer and gyroscope measure linear acceleration (or velocity changes) and angular velocity (or orientation changes) along the three axes X, Y, and Z, respectively. On the other hand, the magnetometer measures the direction and strength of the Earth's Magnetic Field. Combining the data from these three sensors allows an IMU to provide a detailed picture of the motion in the space of an object and, integrating the returned data (accelerations and angular velocities) over a period, returns a position.

these landmarks, leaving some margin for error by introducing probability. Frequently, however, the external environment and the position of the landmarks are not known a priori. In these cases, SLAM fits perfectly, whereby the robot uses its sensors to build a map of the environment while locating itself. The idea is to estimate the positions of the landmarks, then compute the pose of the robot beginning from the fixed landmarks. In this case, these landmarks are the map of the environment; therefore, estimating their position and then reporting the robot's pose relative to these markers means measuring the robot's position with respect to the map. SLAM exploits the two types of Oculus sensors:

- the chambers on the viewer that return an ample visual spectrum;
- the IMUs.

Figure 10: Tracking of an enclosed environment. The light dots are corner with associated landmarks.

Concerning tracking the headset in the environment, the fundamental concept is the same: after landmarks are identified in the setting, the headset's position is estimated. Generally, the visor is located inside an enclosed environment like the room of a house, and landmarks are associated with corners, as in the reference Figure 10. Therefore, whenever the user moves, he builds up a point cloud for the mapping, while localization is obtained by finding the position of the visor with respect to these points [34].

Once the visor is tracked, the pose of the controllers will refer directly to the headset itself. 11

Figure 11: Constellation system on the controllers as perceived by the headset.

3.2 Weart TouchDIVER

Weart[11]'s TouchDIVER (on the market since 2021) is a wearable haptic glove that can render three different characteristics of the sense of touch to the fingers of the hand: force, texture, and temperature.

The possible applications range from entertainment, marketing, design, or in the medical field. The latter enables increased realism for medical training in a virtual environment.

The device is not a full glove (Figure 13); it is instead composed of three actuation points placed respectively in three thimbles - called *Caps*- and a Control Unit that attaches to the wrist through a bracelet. Different technologies are embedded in the thimbles to deliver different feedbacks type. Thanks to actuators inside the thimble, a metal plate in contact with the fingertip stretches the finger skin, resembling force feedback. This technology falls into the 'moving platform' category, as described in Section 2.3.2.

On the other hand, the same moving plate also acts as a heat exchanger, sharing one side with the Peltier cell [36] thus becoming hot and cold to allow the user to feel the temperature of a virtual object, ranging from a maximum of 42°C down to a minimum of 18°C [37]. The general mechanic of a Peltier cell is described in Section 2.3.2. Vibration in the thimbles, generated by eccentric rotating mass (ERM), are regulated to render the feeling of texture, belonging in the vibrotactile feedbacks.

[11]Weart is an innovative startup that develops wearable and portable products for digitalizing the sense of touch. It was born by the end of 2018 from the idea of the three co-founders: Giovanni Spagnoletti, Guido Gioioso, and Domenico Prattichizzo [35], Full Professor at the University of Siena. It results from the collaboration between e-Novia, the University of Siena, and the Italian Institute of Technology, and the head office is in Milan, Italy [3].

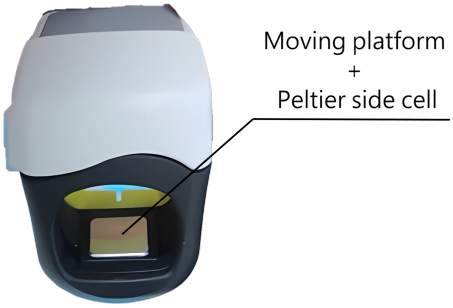

Figure 12: A close-up view of the metal plate, working both as a moving platform and as a heat exchanger.

As recounted by Fabio Pizzato[12] in the Laval Virtual Festival[13] interview [38], ninety percent of grasping movements occur with three fingers only, so it is these that manipulate objects and on which feedback is transmitted. The other two (the ring and the little finger) can follow the middle one in VR. Hence the choice of three fingers; is a compromise between complexity and rendering quality. Apart from these elements, the packaging includes two adapters to plug third parties controllers, a USB Dongle, and user manuals. It comes with a Universal Adapter hosted on the control Unit to plug in the most commercially available controllers such as HTC Vive Trackers, Oculus Controllers, etc. Moreover, there are silicone thimbles of different diameters to fit different-sized fingers: the material used allows their sanitation. The hardware compatibility is with Oculus, Pico, HTC, and Windows Mixed Reality.

Figure 13: A TouchDiver worn without the controller for the wrist tracking.

From the software point of view, its architecture is modular and multi-layered. One of its layers is the WEART Middleware, a background process responsible for communicating with the device (In Figure 14, the user interface panel).

Instead, a second layer includes the SDK that enables the integration of WEART TouchDIVER into client applications. The latter comprises:

- Low-level API accessible by any client application, showing simple functions to

[12]CoE: Chief of Entrepreneurs at e-Novia and Managing Director at Weart
[13]Europe's 1st Virtual Reality and Augmented Reality exhibitio

access tracking data and control haptic signals;

- Developer-friendly SDKs for Unity and Unreal Engine;
- Example projects;
- Ready-to-use demos;
- WEART Tactile Library of forty pre-existing textures[14].

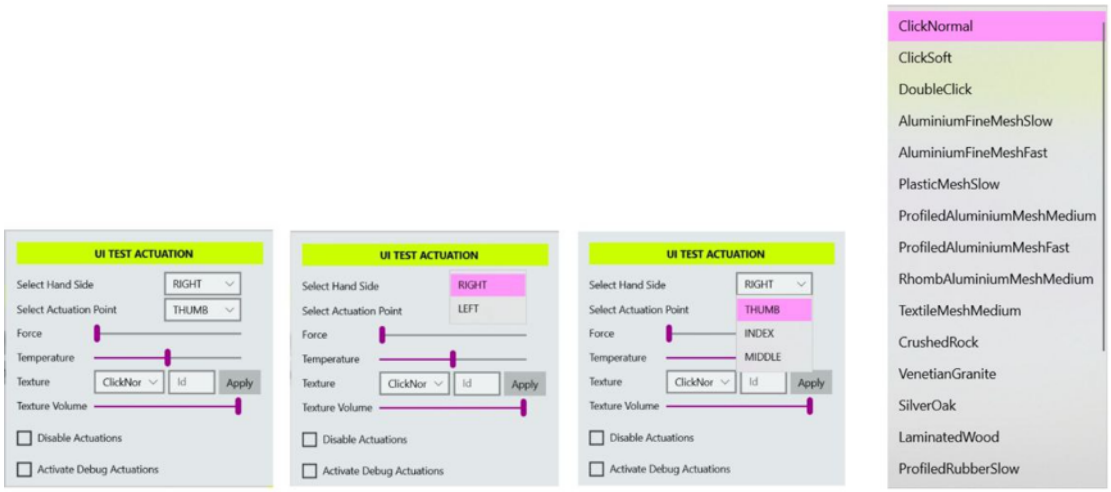

Figure 14: WEART Middleware and a focus of some of the pre-defined textures available

Considerations

When first interacting with a device, some time is needed to understand its behavior. For this purpose, a series of trials were necessary. When doubts arose, the communication with Weart's staff was fruitful. The most relevant results of these tests are listed below.

- Battery: the battery level of the haptic interface, as shown in the middleware, is a mean of the charging states of the three boards present in the Control Unit. For this reason, the percentage is approximately 96% at most, and some oscillations of its value can emerge. The device reveals its full capabilities when the charge is above 30%. Below this threshold, issues may arise; tracking and feedback related to the thumb are more likely to be negatively affected.
- Finger tracking: the equipment embeds an algorithm to track the three thimbles. The firmware executes the algorithm, which computes the closure of the fingers from data coming out of the IMU. A float value between zero and one reflects the closure, where zero represents a fully open digit. The TouchDIVER fails to track slow movements, most probably due to the limits of the implied sensors.

[14]It is possible to obtain new textures by tuning the existing ones.

- Feedback: Force and texture responses are delivered to the fingertips through the actuation of a metal plate in a single direction. The temperature can vary rapidly in the operating range of the device. The highest temperatures reached can be somewhat too intense for some users.

- Wearability: the total weight amounts to less than 200 grams. However, when the device gets coupled with an Oculus controller for wrist tracking, the weight is more than doubled. When limited in time, the experience is pleasing. Prolonged usage can be tiring due to the stress the fingers undergo and the weight the arm has to sustain. Adopting other tracking methods, such as optical markers, avoids the lattermost inconvenience.

3.3 Environment configuration and implementation

The initial phase of the work focused on setting up the work environment beginning from the installations. These latter concern the software and libraries needed for the medical robotics framework upon which the project arises. The mandatory ones are listed below:

1. Eigen (tested with *v3.3.8*);
2. CoppeliaSim (tested with *v4.2.0*);
3. Visual Studio 2019 (tested with Platform toolset *v142*).

In addition, given the use of Weart's TouchDIVER and Meta's Oculus, it was necessary to have:

1. Weart Middleware, Visual Studio UWP module;
2. Oculus.

Finally, it was necessary to define the environment variables as follows:

1. EIGEN;
2. VREPx64_coppelia.

(For Weart TouchDIVER device)

1. WINDOWS_WINMD_PATH;
2. PLATFORM_WINMD_PATH.

3.3.1 CoppeliaSim

CoppeliaSim is a robot simulator used in industry, education, and research, initially produced as part of Toshiba's research and development and is currently actively deployed and maintained by Coppelia Robotics AG.

The CoppeliaSim robot simulator, with an integrated development environment, is based on a distributed control architecture: each object/model can be controlled independently via an embedded script in C/C++, Python, Java, Lua, Matlab, or Octave, a plug-in, a ROS node, a remote API client or a custom solution. All of these features make CoppeliaSim highly versatile and ideal for multi-robot applications and it allows rapid algorithm development, factory automation simulations, rapid prototyping and verification, robotics education, remote monitoring, dual security control, as a digital twin, and much more.

CoppeliaSim uses a kinematics engine for forward and reverses kinematics calculations and several physics simulation libraries (MuJoCo, Bullet, ODE, Vortex, Newton Game

Dynamics) to perform rigid body simulation. Models and scenes are developed by assembling various objects, such as meshes, joints, sensors, point clouds, or OC trees, in a hierarchical structure. Additional features include motion planning, synthetic vision, image processing, collision detection, minimum distance calculation, customized graphical interfaces, and data visualization.

A premise is necessary regarding the use of this software in the project: CoppeliaSim allows the manipulation and management of objects that can be merged as one, as happens for the virtual robotic hand- which is detailed later- is composed of many individual links and joints. These, following physical rules, work together to form the hand.

The functionality of CoppeliaSim mentioned above is highly convenient, as it allows the designer to manipulate components of all shapes and sizes and combine them to make any shape required to complete a task. Moreover, it is necessary to know how different objects can work together so as not to run into structural and physical issues that could block the entire operation of the project.

The scene of the project requires objects to interact with each other: without passing through, generating a collision, and passing through without colliding [39].

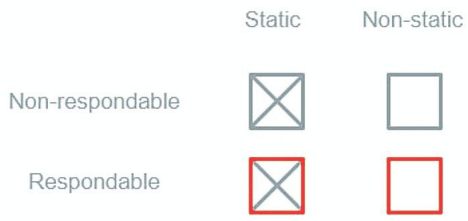

Figure 15: Categories of CoppeliaSim shapes

It is essential to premise that shapes, depending on their behavior during dynamic simulation, can be classified into four categories in CoppeliaSim as shown in Figure 15. During dynamic simulation, static shapes are not affected by any changes in position (i.e., their position relative to their parent object is fixed), whereas non-static ones are influenced directly by gravity or other constraints. Respondable shapes influence each other during dynamic collisions (i.e., they produce a mutual collision reaction: they bounce off each other). The figure 16 illustrates the static/non-static, respondable/non-respondable behaviors. To sum up, two objects collide only under certain circumstances, and specifically, they must comply with the following characteristics:

- they must be respondable;
- they must be subject to dynamics.

These two characteristics, however, require specifications. For the respondability of an object, it is necessary to check a group of flags that can lead it both to penetrate

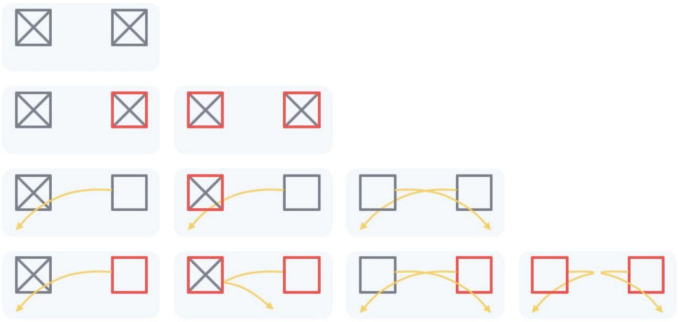

Figure 16: Behaviour of different shape types in collision scenario

with others or to collide. For example, considering all the components that make up a hand, depending on the check of the flags of the phalanges, they might clash one against the other. In conclusion, the correct configuration of flags means the right choice of object to -or not to- interpenetrate. The flags explained compose *Collision masks*: two respondable shapes will always produce a collision reaction unless their respective *Collision masks* do not overlap. Shape properties of respondability and dynamicity and their *Collision masks* can be modified in the *Shape dynamics properties* dialog (generic example shown in Figure 17).

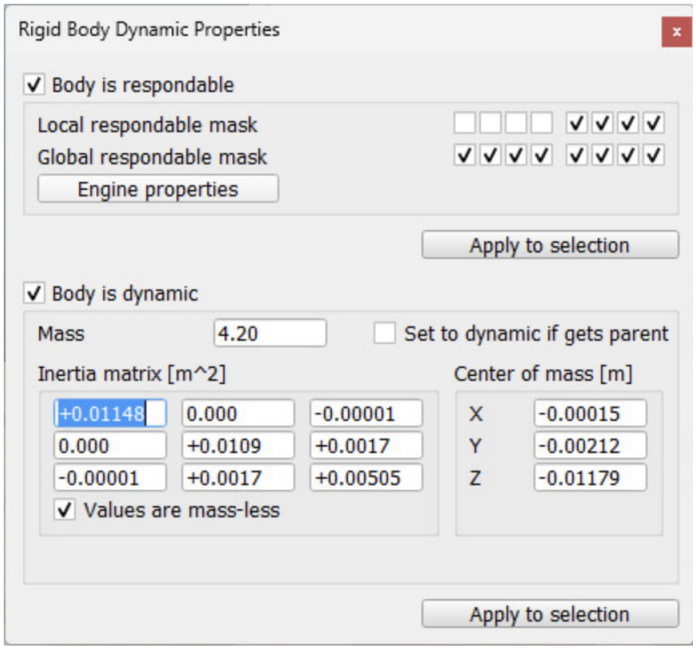

Figure 17: Dialog window box on body properties

As shown in the Figure 17, the combination of eight flags checkings (2^8 combinations) composes two types of masks:

- Local;

- Global.

Since -as already mentioned- CoppeliaSim organizes the components of a scene in a hierarchy with parent-child relationships between the objects, the local mask concerns parts belonging to the same hierarchical branch, while the global regards the entire scene.

Concerning the need for components to be subject to dynamics, it raises a question as this leads to objects being subject to gravity, inertia, friction, damping, and other physical quantities that derive from dynamics. In CoppeliaSim, only a limited number of components (shapes, joints, and force sensors) provide for a dynamic simulation, and it will depend on the scene structure as on object properties, whether the latter is dynamic itself. Those are easy recognizable during simulation since an icon appears next to the object in the scene hierarchy.

The kinematic engine [15] chosen is Bullet v2.78 [16] which is one of the most recommended for collision detection among the engines in CoppeliaSim. Initially, the activation of dynamics in the scene led to the dynamic objects falling and rolling, as these objects had unstable positions and orientations or uneven surfaces as support bases. To solve this problem at the script level (in *Lua*), it is controlled that the components affected by this complication are repositioned cyclically. Specifically, as the simulation starts, the wrist takes the position of the Oculus controller and its orientation modified as shown in picture 18, which is managed externally and set in motion by the human operator controlling the simulation. Regarding phantom components, they are positioned respecting

```
pos = sim.getObjectPosition(righthand,sim.handle_world)
sim.setObjectPosition(simBase,sim.handle_world, pos)
orient = sim.getObjectOrientation(righthand,sim.handle_parent)
orient[1] = orient[1] + (math.pi/2)
orient[2] = orient[2] + (math.pi)
sim.setObjectOrientation(simBase,sim.handle_parent, orient)
```

Figure 18: Set_position and *Set_orientation* functions

the initial disposition of the digital twin of the abdomen and therefore remain in their position despite the influence of gravity through a *Set_position* function until grasped by the hand (Figure 19).

[15] CoppeliaSim currently supports five physics engines: the Bullet physics library, the Open Dynamics Engine, the MuJoCo physics engine, the Vortex Studio engine, and the Newton Dynamics engine. Although they present only little differences, it's possible to switch at any time from one to another depending on the necessities in the simulation. Making simulations is an elaborate process, and can be obtained by emphasizing one aspect more than others, such as precision/realism, speed, or with support of various features:

[16] Bullet physics library is an open source physics engine featuring 3D collision detection, rigid body dynamics, and soft body dynamics; games and movies for visual effects use this engine. It is often considered a *Game physics* engine.

```
pos3 = {0.5461, 0.0620, 0.7908}
sim.setObjectPosition(left_kidney,sim.handle_world, pos3)
```

Figure 19: *Set Object Position* function

When this occurs, through a *Set parent* function

$$(sim.setObjectParent(attachedShape, connector, true)),$$

they become *Children* [17] of the hand as long as a force is detected. This force arises from the contact between the hand and the object (organ) in question; on the contrary, when it is lost - i.e., when the hand releases the phantom component- its initial position is restored. The same applies to the primitive forms (such as a cube), which are initially positioned on the table and, once released by the hand, fall again on the table.

Considerations

At the educational and robotic research level, CoppeliaSim is a benchmark that has established its primacy in this type of application over the years. However, CoppeliaSim's vastness of functionalities led to performance issues, and thus to ensure a successful outcome, it was necessary to manipulate these functionalities according to the purposes to achieve. Initially, CoppeliaSim's graphics and kinematics engines were not performing well enough for task completion, and to make them so, it was necessary to lighten the scene, for instance, by avoiding making palpable and graspable all the organs that are part of the phantom. Moreover, since the phantom was reconstructed through abdominal Computed tomography (CT), the organs are multifaceted and deformed and that makes them much more complex to render than *primitive shapes*[18].

Primitive shapes are the most highly recommended, but CoppeliaSim works well dynamically also with other generic objects as long as they are convex[19]. For this reason, most of the operations were initially tested and simulated using a cube[20]. Only at a later stage and mainly for demonstration purposes, some organs of the phantom were made graspable and palpable, specifically left and right kidney and a liver lesion since they are of a suitable size to be taken. However, that led to a stall in processing by the CoppeliaSim VR Interface application, which graphically failed to

[17] Objects in the scene hierarchy can be dragged and dropped onto another object, to create a parent-child relationship [40].

[18] Primitive shapes could be cuboids, cylinders, or spheres. In CoppeliaSim, a primitive shape (or primitive compound shape) is the most suitable for computing dynamic response to collisions because it performs fast and stable [41].

[19] Convex shapes are *optimized for dynamics collision response calculation (but primitive shapes are recommended)*[41]

[20] Cubes are convex and *primitive* shapes

render all the objects. For this reason, the simulation was interrupted without being able to test any functionality.

Therefore, to make simulations, only the right kidney was made graspable. Making the objects dynamically enabled means that they are subject to gravity, leading them inevitably to fall as the simulation starts. To overcome this issue, the positions of the mentioned objects and the hand - that had to be dynamic to allow the interactions- were forced by code. They are thus cyclically maintained at a given position in the absence of external perturbations.

The phantom objects are held in their initial position until the perception of a contact force is given by the hand's attempt to grasp them. The hand, instead, updates its current position continuously based on the position of the Oculus controller plus an offset[21]. When the kidney is taken, it becomes a child of the hand and follows its position until it is released: at that point, it returns to its initial position.

3.3.2 CoppeliaSim VRInterface

At the initial stage, the goal is to make CoppeliSim scene VR-ready, so that it is possible to switch from a 2D to a virtual view experiencing Virtual Reality even in a robotic simulation environment- such as CoppeliaSim- and extract information of interest. That is possible thanks to a specific interface. Before moving on to the installations, it is necessary to check that the computer is VR-ready. Next, once on the site of CoppeliaSim VR Toolbox [42], the following steps must be followed:

1. Click on the tags in the upper part of the page;
2. select v1.0;
3. install vrep_vr_installer.msi.

Up to this point, launch the bat file (*copyToVREP.bat*) in the CoppeliaVRInterface folder:

$$(C : Files(x86)VRinterface).$$

First, verify in the file of interest that the line of code below has the correct path.

$$SETvrepLocation = C : Files$$

Once done, run the file as an administrator since it is inside the program folder. The folder named above has all the pre-compiled libraries, a subfolder containing other CoppeliaSim scenes (on which it is possible to test the interface), and another subfolder

[21]The offset was chosen empirically, considering the average distance between the wrist-at the height where the Oculus controller is docked-and the hand. The hand was rotated 90 degrees in the pitch axis and 180 degrees in the yaw axis, given the position shown in Figure 1

with the scripts that allow making the scene virtual. The latter is the most significant since they should be added directly to the CoppeliaSim scene, as further detailed in 3.3.3. The CoppeliaSim VR interface has the functionalities listed below:

- reading all renderable geometry from CoppeliaSim
- reading dynamically generated geometry

Furthermore, it synchronizes all poses in real-time and sends positions of controllers and headset to CoppeliaSim, including all controller button-press/button-touch as string signals.

3.3.3 CoppeliaSim Scene

In this paragraph, is developed the scene design to work on and within which to perform the simulations. As mentioned in the introduction, the project aims to create an interaction between physical and Virtual Reality, controlling a robotic hand within CoppeliaSim through the TouchDIVER device, more specifically, a robotic hand that faithfully reproduces the movements of a human operator's hand in physical reality. The purpose of the human operator is to handle, manipulate and palpate human organs located within the digital phantom.

A scene represents the 3D virtual environment, consisting of cameras, lights, views, scripts, and viewable objects. It is made VR by the addition of the *HTC VIVE* object[22]. The latter is composed of:

- Headset
- RightController
- LeftController

and replicates the actual headset and controllers real movement in the virtual one. But most importantly, the scene of this project includes the phantom -placed on a table-, and a robotic hand [23]. In addition, primitive shapes were imported to facilitate simulations, since CoppeliaSim graphics engine works more efficiently with those.

3.3.3.1 Robotic hand

As mentioned in the grasping chapter 3.3.3.4, when first approaching the problem, it was convenient to use Coppeliasim's grippers. The latter present characteristics that differ widely from those of the human hand. Nevertheless, they were helpful in understanding the general behavior of the system. In particular, the grippers allowed

[22]This object comes from the subfolder mentioned in 3.3.2

[23]In chapter 3.3.3.1, an in-depth on how it was built and modeled to have a shape and freedom of movement similar to a human hand

mapping the closure value returned by the TouchDIVER to the closing of a virtual object without introducing excessive complexity. *Jaco hand* and *Barret hand* served this purpose (Figure 20).

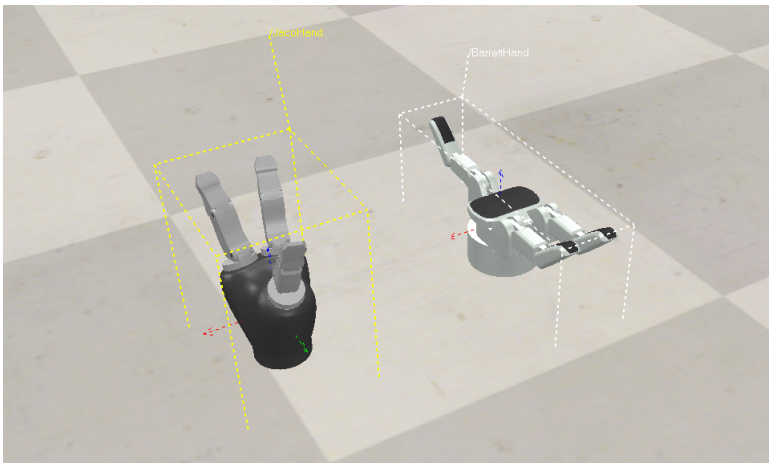

Figure 20: The Jaco hand (on the left) and the Barret hand

Both present three fingers and therefore were easy to control using the values returned by the three thimbles. Each finger includes two links and two rotational joints. After some trials, the movement of these simplified hands was consistent with the evolution of the physical hand. Consistency is not sufficient for a satisfactory VR experience. Indeed, the virtual objects should be as similar as possible to the real ones to improve the realism of the scene.

Hand design For this reason, a suitable hand model had to be found. After consulting different sources, a candidate model was identified[43]. It was a 3D model of a left robotic hand with five fingers, each made of three links (Figure 21) Some

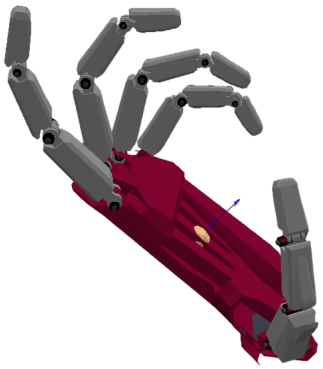

Figure 21: The 3D model of the robotic hand

modifications were necessary to obtain the desired result: reversing the hand using a third part software, and adding joints- to be placed appropriately- between every link, directly on CoppeliaSim. Three rotational joints were placed on each finger: one

between the palm and the first link (proximal phalanx) and the other two between the first and second link (intermediate phalanx) and the second and third link (distal phalanx), respectively. A hierarchy had to be defined on the hand to establish dependencies between consecutive elements of the structure. 22

Moreover, the *Right Controller*[24] of the Oculus had to be set as parent of the hand, to properly move it in space according to the position of the controller. Coppeliasim offers the opportunity to choose how to control the joints, as can be appreciated in Figure 23.

In this work, all joints are controlled in position. Position control is usually appropriate when the robot has to follow a prescribed trajectory, like in the present case [44]

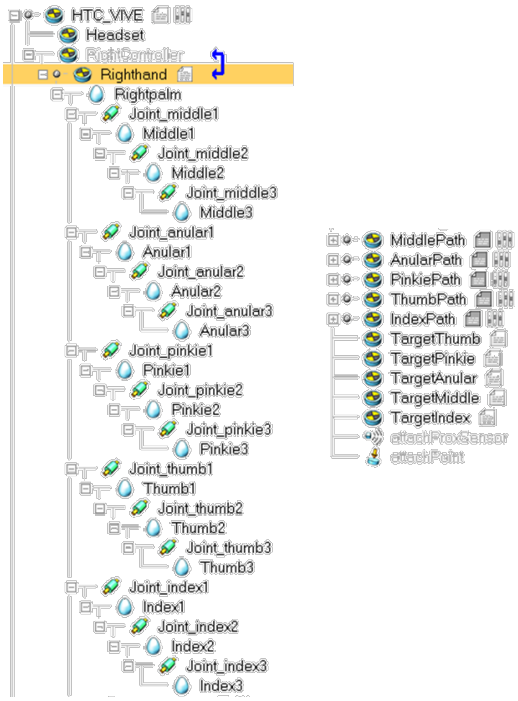

Figure 22

All joints had to be dynamically enabled, given the dynamic nature of the simulation [45]. After positioning the joints, the range for each one had to be specified to avoid non-physiological movements. For index, middle, ring, and little fingers, their ranges are:

- joint one 70°;
- joints two and three 90°.

As regards the thumb:

[24]In the scene the VR elements are represented by Headset, RightController and LeftController objects3.3.3

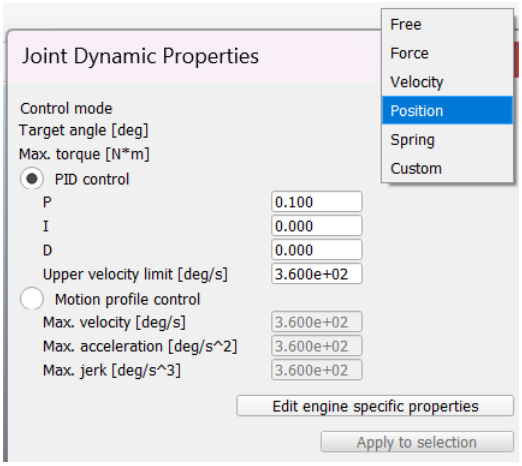

Figure 23

- joint one 30°;
- joint two 70°;
- joint three 90°.

The hand, as described above, presented a non-smooth surface that led to complications during runtime, since -as already detailed in the chapter 3.3.1- it performs better with convex shapes. For instance, calculating distances between complex shapes is computationally more demanding. Thankfully, CoppeliaSim allows morphing an object into its convex hull. After this process, the model looks smoother as shown in figure 24.

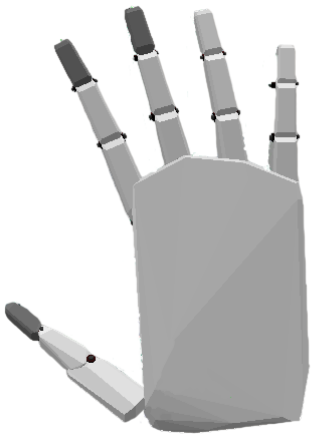

Figure 24: The convex 3d model, modified final version.

Hand code On CoppeliaSim, each scene object can have an associated child script. It can be written in two programming languages, Python or Lua, and they can be both threaded and non-threaded [46]. For consistency with the scenes already applied

to the framework, each implemented child script is in the Lua programming language.
Non-Threaded code:

- Access to a script is via callback functions not running in a thread. Thus, when recalling those functions, they should perform their task and signals its end. When this doesn't occur, CoppeliaSim stops;
- A non-threaded script is executed in the same thread as CoppeliaSim (for Lua scripts).

Threaded code:

- It allows the execution of a task in a loop without interrupting the executive flow of the program;
- Coroutines manage threads that switch to others at regular intervals.

In an early version, the *Right Hand* object contained a non-threaded script, then converted to a threaded one because it became necessary for a continuous loop recognition of the eventually touched object. In fact, with the *sensing* function, which belongs to a non-threaded child script, when there are no objects in touch, the variable associated with the object id [25] touched, will not be updated to a null value, equivalent to the meaning of *zero objects in contact*. Instead, the id of the last touched object persisted, even when there was no contact, not to interrupt feedback rendering. Since the recognition task was in a sensing function, was executed only when sensing occurred instead of continuously. Therefore, it was necessary to make the code threaded.

The threaded script of *Right Hand* is developed from four predefined functions:

- The *init* function, where the assignments of the objects useful for task realization occur -such as the ids of the hand and the elements it will have to interact with-, the creation of the *IK environment*, and where the coroutine is created and instantiated;
- The *actuation* function, which is responsible for carrying out the task of hand movement and grasping;
- The *coroutineMain* function represents the core function of the script, within which resides the thread which cyclically acquires information regarding thimbles to render force and texture feedback. The information refers to force vectors, depending on the contact between finger and object in the scene, and the identifiers of those objects touched, then being sent to the framework to realize the rendering of the appropriate feedback;
- The *clean_up* function, launched only at the end of the execution of the scene.

[25] In CoppeliaSim, each element has a unique identification number represented by an integer

3.3.3.2 Path design

As explained in chapter 2.1, the grasping movement of the fingers follows a path described by a semi-cardioid curve. The implementation starts from the basic function of the cardioid parameterized according to the *time of flight*[26], namely:

$$\begin{cases} x = (1 - cos(s))cos(s) \\ y = sin(s)(1 - cos(s)) \end{cases} , s = TimeOfFlight. \tag{3}$$

To obtain the desired curve, the axes were then inverted:

$$\begin{cases} x = sin(s)(1 - cos(s)) \\ y = (1 - cos(s))cos(s) \end{cases} \tag{4}$$

and was subsequently rotated by 180 degrees with respect to the x-axis:

$$\begin{cases} x = sin(s)(1 - cos(s)) \\ y = -(1 - cos(s))cos(s) \end{cases} \tag{5}$$

Finally, for the shape to be akin to the closing of the fingers, the point of origin has been changed, replacing the parameter s by $(\pi - s\pi)$

$$\begin{cases} x = sin(\pi - s\pi)(1 - cos(\pi - s\pi)) \\ y = -(1 - cos(\pi - s\pi))cos(\pi - s\pi) \end{cases} \tag{6}$$

The steps described in the equations 3, 4, and 5 are respectively depicted in Figure 25.

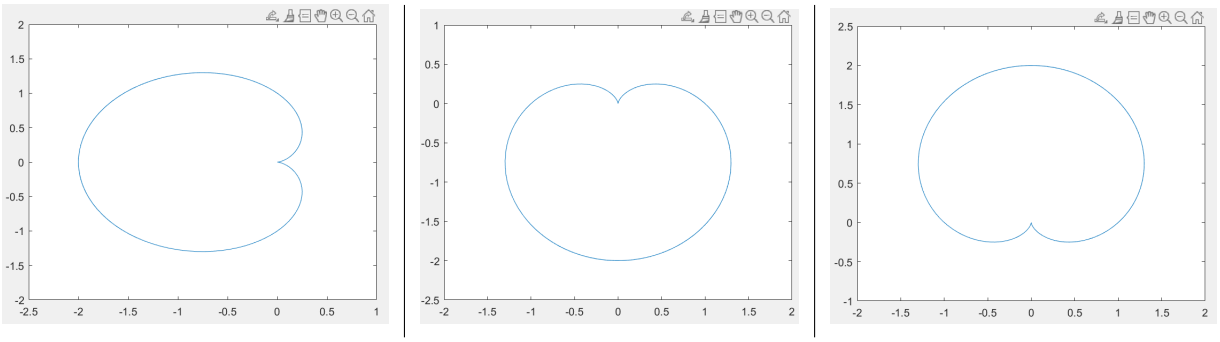

Figure 25

At a later stage, the path has been implemented in CoppeliaSim. The simulator has a *path object*, namely a pseudo object that represents a succession of points via dummies with orientation in space [47] called *control points*, allowed to be manipulated.

Therefore, it was necessary to determine the control points that would create the semi-cardioid, which can be derived from the mathematical equation using a

[26]It is a value between 0 and 1 indicating the percentage of finger closure, as better detailed in 2.1

Matlab script. This was possible thanks to a *for* loop that, starting from equation 5, selects only the control points of the quadrants with the positive x-axis. (Figure 26) Accordingly, the coordinates of the control points previously obtained were inserted within the path object, generating the semi-cardioids.

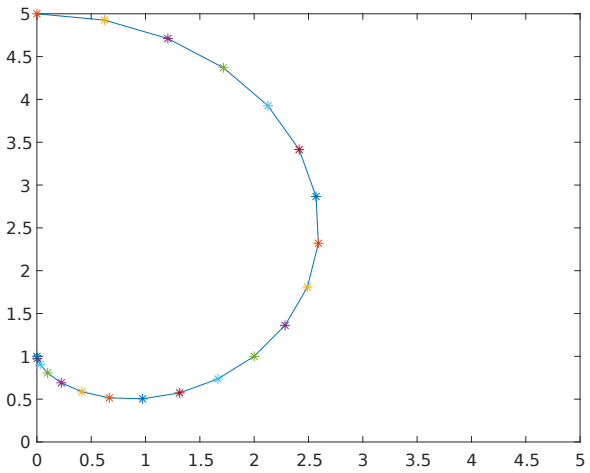

Figure 26

Each finger has its path to follow, with the necessary readjustments: for example, the thumb requires fewer control points since it is limited to making the end part more curved. For the other fingers, it was just a matter of scaling the path in size since the pinkie makes a smaller path than the middle finger due to a simple difference in link lengths. (Figure 27).

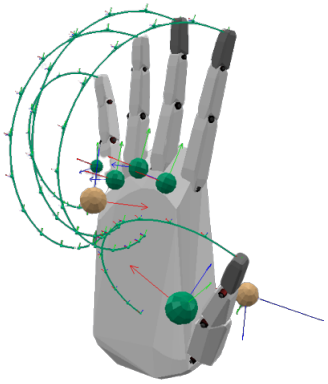

Figure 27: In green, for each finger, is shown the path of the relative fingertip

To utilize the inverse kinematics function (as explained in chapters 2.2 and 3.3.3.3), five dummy targets-one for each finger-must follow the paths described above.

Therefore, a Lua script was implemented for each finger target, starting from the reference in the Coppeliasim manual [47] but adapted according to *time-of-flight*. Notice that, in the example reported below, the position along the path depends on a

value called *trigger thumb* that represents the *time of flight* of - in this example case- the thumb [27]. Hence, from the reference code of CoppeliaSim:

```
posAlongPath=posAlongPath+velocity*(t-previousSimulationTime)
posAlongPath=posAlongPath % totalLength
```

follows:[28]

```
posAlongPath = trigger_thumb * totalLength
```

[27]The code is analogous for the other finger paths.

[28]the modulo operator % makes the variable *posAlongPath* cyclic.[48] In this specific case, the modulo operator has been changed with the multiplication one since the cyclicity was not useful. In this way, the movement is controlled with the *trigger thumb*

3.3.3.3 Inverse kinematics implementation

After selecting the paths for the fingertips, the issue was deciding how to solve the inverse kinematics problem. Luckily enough, CoppeliaSim offers a flexible inverse kinematics plug-in. Thanks to this, a collection of C++ functions (*Coppelia Kinematics Routines*) is accessible. Using such tools gives the possibility of setting up complex kinematics tasks in a distinct *IK environment*, allowing for independence from other aspects of the simulation model (such as the dynamics). The calculations are Jacobian based and support several resolution methods, namely *Jacobian pseudo-inverse*, *damped least squares pseudo-inverse (DLS)*, and *null-space projections*.

The first step in setting up the task was defining the *IK elements*, representing kinematic chains consisting of at least one joint. To characterize these elements properly, a *base object*, a *tip dummy*, and a *target dummy* that the *tip dummy* must follow are needed [49]. Secondly, it must be established the type of constraint- which can be in position, orientation or pose- that links the tip and the target. In this project, it was instantiated one *IK element* for each finger with a shared base object, i.e. the *RightPalm*. (Figure 28) The tip dummies were made coincident with the extremity of

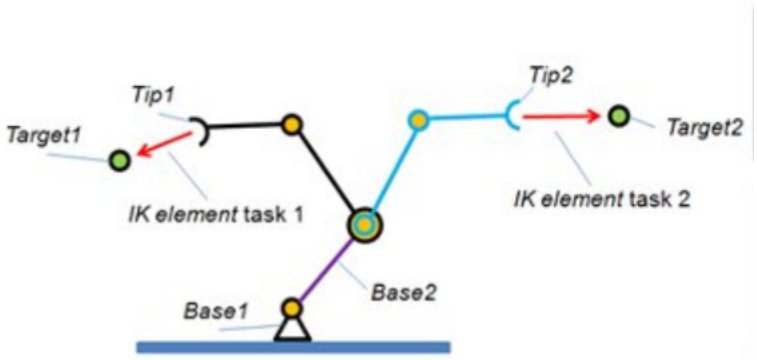

Figure 28

the fingers, while the target dummies were placed on the appropriate paths, as shown in Figure 29. Position-only constraint was set for all the targets.

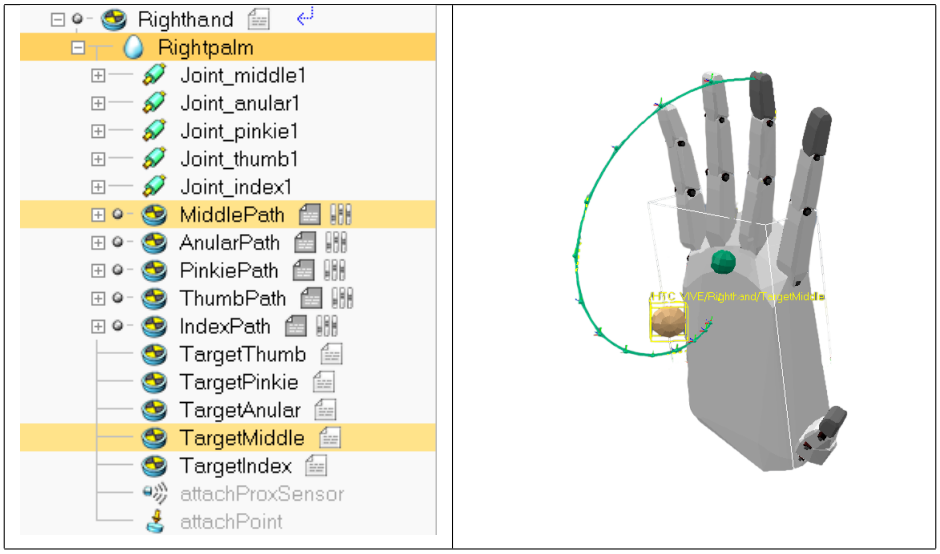

Figure 29

Subsequently, the five elements were organized into two *IK groups*.[29] The first *IK group*, employed the *undamped pseudo-inverse* method. Whilst, the second group served instead as an alternative in case of failure, and implemented the DLS method. Further details on the number of iterations and the damping factor can be appreciated in the code below[30].

In general:

Lua synopsis | simIK.setGroupCalculation(int environmentHandle,int ikGroupHandle,int method,float damping,float maxIterations) [50]

specifically:

```
simIK.setIkGroupCalculation(ikEnv, ikGroup1_undamped, simIK.method_pseudo_inverse, 0, 6)
simIK.setIkGroupCalculation(ikEnv, ikGroup1_damped, simIK.method_damped_least_squares, 0.1, 99)
```

This function is called with two different resolution methods, as mentioned above. The damping factor was introduced to reduce joint velocities. 2.2 This parameter takes values between 0 (minimum value) and 1 (maximum value). To prevent velocities from being slowed down excessively during simulation, a value of 0.1 was chosen. In fact, damping factors higher than 0.5 led to delays between the physical hand and the virtual one. This effect was even more evident near the target, resulting in a unrealistic simulation.

In general terms, it is not trivial to determine the optimal number of iterations needed to solve an inverse kinematics problem required by the function

[29] An IK group allows setting the overall solving properties, such as the solver type and the number of iterations.

[30] *setIkGroupCalculation* sets calculation properties for a *IK group*.

setIkGroupCalculation. This parameter depends on the complexity of the robot's model and on the desired accuracy, and must be consistent with the computational power of the used machine. For this reason, a series of trials are commonly executed before selecting the optimal number of iterations. The parameter corresponding to the most convincing simulation is eventually adopted. In this work, the number of iterations was specified for both methods. As shown in the code above, this parameter varies widely between pseudo-inverse and DLS even differing in the order of magnitude. This difference allows attempting to solve the problem using the undamped method without risking wasting too much simulation time in finding a feasible solution. If failure is returned after these few iterations, the solver switches to DLS. This approach offers increased robustness, stability, and a faster convergence rate and is therefore favored to perform several iterations.

3.3.3.4 Grasping

One of the main goals of this project is to grasp objects in the scene and receive feedback on TouchDIVER thimbles. The first issue that arose in this part was making graspable the components of the CoppeliaSim scene, without passing through them. As explained in chapter 3.3.1, the boxes of the respondable masks had to be ticked to avoid permeation between objects. Two examples are shown in Figure 30, reporting respectively the respondable masks of the hand palm and the cuboid.

Figure 30: Possibile combinations of body respondability

The several grippers already present in CoppeliaSim, established the base for the choice of grasping modality for the robotic hand since they are capable of correctly taking objects. In the same way as the latter work, the hand makes a *fake grasping*, which works as a magnetic attachment.

For two objects to interact, CoppeliaSim requires both to be respondable and dynamic. That led to an other issue with grasping, namely the need to enable the dynamic property of the objects, which consequently tended to fall because of the force of gravity. One proposed solution is to give them a fixed position and thus force them not to yield to the dynamic forces to which they are subjected 3.3.1; however, other possible solutions to be explored, are proposed in the chapter on future developments 5.

Another limitation is in the rendering of the scene: as reported in the CoppeliaSim manual and empirically confirmed through tests and simulations, CoppeliaSim works better with convex objects. Accordingly, in some tests, the components inside the

phantom were made convex: nevertheless, it is not always possible to transform concave objects, as in the case of the fat enveloping the phantom. Indeed, it is hollow with all the organs inside, so making it convex meant *filling it*, losing both its realism and practical usefulness as a *container object*.

Moreover- surprisingly- making the objects convex led to a reduction in the Frame Per Second (FPS) of the scene, with a significant slowdown in the simulation phase. The frame rate is lowered even adding dynamically enabled and respondable objects. As a result, only primitive shapes and just one component of the phantom have been made graspable (with the dynamics enabled). Otherwise, it would almost be impossible to do tests because of the excessive delay.

CoppeliaSim allows the simulation of the fake grasping using a force sensor called *attachPoint*, namely an object that is functional only if the scene is dynamically enabled (3.3.1), to combine with a proximity sensor. Both are in the center of the robotic hand palm. If the proximity sensor detects the closeness to an object, it sticks to the *attachPoint* as a magnet.

Nevertheless, there is no perception of magnetic attachment in the simulation phase; instead, it gives the feeling of a realistic grip (thanks also to the force feedback the haptic device provides).

What specifically happens when grasping is performed is listed below.

- CoppeliaSim receives the closure values of each thimble (thumb, index, and middle finger) from the framework 2.1, via a *getFloatSignal* function; [31].
- When a graspable object makes contact with the hand, and the latter is in the semi-close position (i.e., its closure value is ≥ 0.5), it occurs the magnetic attachment and the touched object becomes child of the *attachPoint*. Conversely, when the hand is open (closure value < 0.5), the object disengages and returns to be a child of its initial parent again;
- Finally, it is checked that the object to be picked up by the hand is not the hand itself.

After some tests with primitive shapes, the right kidney in the phantom was made graspable. As required by the simulator, it has been made respondable and dynamically enabled, but for the issues mentioned above, its position in space had to be fixed so that it could *resist* the effect of gravity. Subsequently, it is performed a *force check* through the *getContactInfo* function, to decide whether the object should be grasped:

- null force: object stays in its position;

[31] On the side of the framework there is a *setFloatSignal* in which the cyclic update of the closure values for each thimble takes place

- non-zero force: grasping the kidney with attached force feedback and change of position.

Once you *drop* the kidney, it returns to its original position.

3.3.3.5 Feedback implementation from CoppeliaSim viewpoint

As already anticipated in chapter 3.3.3.1, the coroutine thread contributes to the rendering of feedback. In the *CoroutineMain* function, defined in the righthand script, is implemented the communication with the framework to accomplish the force and texture feedback rendering.

First of all, in the loop, it's cyclically performed a *getContactInfo* [51] function for the thumb, index, and middle thimbles. This function provides:

- The IDs of the two objects contacting;
- The normal vector at the contact point;
- A force vector, which is generated by the contact.

This function also provides coordinates of the contact point but represents an irrelevant parameter for the aim purpose.

Once the parameter values are stored, signals are sent to the framework, enclosed in a float data type via the *setFloatSignal* function.

As already mentioned in chapter 2.3 and 3.2, the Weart haptic device used for this project is built to render three different feedback: force, temperature, and texture. The feedback realization depends on events[32] captured on the CoppeliaSim scene and handled in the framework. This paragraph will describe only how the events are caught, to be handled by the framework in chapter 3.4. The information captured is useful only for the force and texture rendering because the temperature rendering is completely handled in the framework implementation.

Force feedback: For each finger, it's executed a check of the force vector values.

- When it's zero for each coordinate, is sent a zero value to the framework, which corresponds to an idle state of feedback render. Thus, there is no rendering action.
- Otherwise, the scalar product is computed between the force vector and the normal vector, stored in *float data*, and then sent to the framework. The value corresponds to the intensity of the elastic force due to finger pressure relative to the normal surface of the object in contact.

[32] the events of interest consist of the collision of the robotic hand's thimbles with the objects dynamically enabled or at least respondable.

Texture feedback: In the *CoroutineMain* function, defined in the *righthand* script, is also implemented communication with the framework to accomplish the texture rendering. It is checked for each finger if an object is touched at that time.

- When no object is in touch, the *setFloatSignal* function will send a zero signal, which corresponds to an idle state of feedback render. Also here, there isn't a feedback rendering action;

- Otherwise, it sends a value corresponding to the id of the object touched. It's recalled that it will be rendered a different feedback according to the specific object touched, thus the identification value is essential for the render realization in the framework implementation.

Since communication must occur between CoppeliaSim and the framework, object ids must match on both sides for a correct interpretation. From the framework viewpoint, the identifiers are fixed, but every time the scene is launched, the relative id on CoppeliaSim may change arbitrarily. Therefore, a fictitious id is assigned to the particular object with texture to render. In this way, the identification value will always be correctly recognized from the framework viewpoint.

In this case, the table and fat ids, the only objects in the scene that will have texture renderings, have been assigned specific values, so out of the ambiguity of interpretation.

The signal sent is a float converted from an integer since the communication through *setFloatSignal* involves a constraint on the data type, thus necessitating a trivial type cast.

3.3.4 Configure CoppeliaSim for multi-threading communication

In this project, CoppeliaSim must be able to communicate with other software since it must exchange data with the haptic device, the VR device, the entire framework application, and the CoppeliaSim VR interface. By default, CoppeliaSim has only one communication channel. To make it interact with other external applications it is necessary to edit its text file, called *remoteApiConnections.txt*, and configure its communication socket ports to enable it to work on several threads in parallel. Therefore, it became necessary to create a new *remoteApiConnections.txt*, in which four different communication ports were configured. This configuration file was placed in CoppeliaSim's system folders, replacing the previous one for CoppeliaSim's multi-threaded communication.

The setting of the ports turned out to be of considerable importance in the communication between devices so that the proper set of the ports guarantees the subsequent operation of the whole project. An improper configuration generates conflicts between the devices and the software used. These conflicts can lead to a loss of part of the information during the exchange of data and therefore cause certain functionalities to fail.

3.3.5 Inter-device communication

This project exploits a pre-existing framework capable of interacting with different devices, such as manipulators, controllers, or haptic devices. Specifically, the framework allows these devices to communicate with the virtual environment and the CoppeliaSim simulator, accomplishing specific tasks described in it. The framework in question comprises four layers, following the standard terminologies, representing the abstraction levels separating the operator using the various devices and the implemented functionalities. The name definition of each layer follows the nomenclature rule of the standard defined before.

The used framework is structured as follows:

The **Proxy Layer** is the highest. It represents the layer that shares all the classes that interface with the physical devices connected to the framework. Thus, there exists a class for each connected device that uses its libraries to communicate. In the image 31, there are various example devices: *RobotProxy* interfaces with the physical Kuka manipulator; *JoystickProxy* is a class used to interface with the Nintendo Switch controllers; *GeomagicProxy* uses the Geomagic libraries to be able to read and send data. The *Proxy* component of interest is *WeartProxy*, which interfaces with the haptic device TouchDIVER via Low-Level API SDK, thus getting its thimbles positions and closure values and setting the feedback accordingly. The *VREPProxy* class has been exploited to interact with the CoppeliaSim virtual environment. This class was already implemented, guaranteeing the correct communication socket ports between the framework and CoppeliaSim. One of the default ports was the 19997.

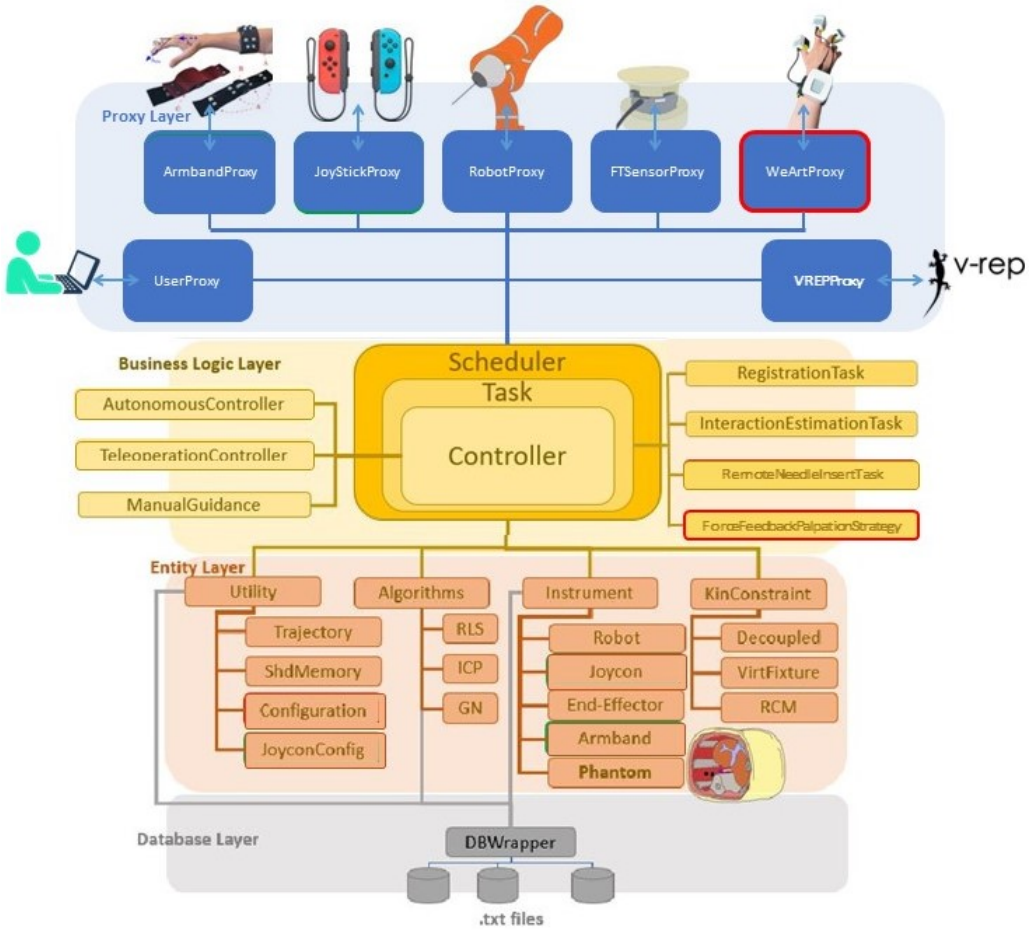

Figure 31: Schematics of the framework

However, CoppeliaSim VR interface also communicated through port 19997 creating a communication conflict 3.3.4, The problem was solved by opening port 19994 to separate the communication channels.

- 19997: communication port between CoppeliaSim and CoppeliaSim VR interface.
- 19994: one of the communication port between CoppeliaSim and framework.

Only some predefined methods were exploited to transmit and receive information with CoppeliaSim, namely the *setFloatSignal* and *getFloatSignal* functions. Through these, it is possible to set and get the exchanged values by passing them through the connection port of the assigned server. The definitions of these functions follow.

Under the *Proxy* level, there is the **Business Logic layer** (BLL). This level is responsible for the description and implementation of high-level tasks. The classes at this layer interface with the higher (*Proxy*) and the lower (*Entity*) level. At this layer, the *Force Feedback Palpation Strategy* is implemented as deepened afterward in 3.4.1.1.

At the **Entity Layer**, each class represents an atomic component called up in the calculations of higher-level tasks. Each object exists as an abstract element that serves

the higher task classes to be able to process data. There is no difference between the physical and simulated devices at this layer. For example, differential kinematics is implemented in the robot class regardless of whether, in the *Proxy* layer, there is a physical robot or not. Differential kinematics is only needed to perform the computation by exchanging data with *Proxy* while, in reality, the robot in the *Entity* layer is entirely abstract.

The lowest level is the **Database Layer**. It interfaces with all configuration files to read and write data during task execution. [52]

3.4 Framework implementation

The project aims to move a virtual hand in the CoppeliaSim environment and return the feedback upon contact with objects in the scene. Accomplishing the project means working on two fronts: CoppeliaSim on one hand, and the framework on the other. Via CoppeliaSim means making objects in the reference scene and associating them to related scripts, while, on the framework side, adding classes and modifying pre-existing ones.

3.4.1 Overview

As previously described, the framework already consisted of a stable structure. The aim was to enrich it, implementing new tasks and new features. It was necessary to work exclusively on the two highest levels: on the *Proxy layer* - improving the interaction and exploiting the full potential of the haptic device -, and on the *Business Logic layer* by implementing the goal task, namely the *Force Feedback Palpation Strategy*.

3.4.1.1 Business Logic Layer (BLL)

At the *Business Logic* level was implemented a class named *Force Feedback Palpation Strategy*. For brevity, from now it will be referred to as *Palpation Strategy*. This class was not developed ex novo: it already exists a class that implements a task similar to the purposes of the project, namely the *Force Feedback Needle Insertion Strategy* (Figure 2.4). In particular, also it deals with Weart's TouchDIVER, which was employed as a reference model so that it was already guaranteed the proper communication between the haptic device and the framework.

Inside the *Palpation Strategy* class are captured the signals from the CoppeliaSim events and handled as follows:

- The transmitted signals from the CoppeliaSim scene are captured. As already detailed in chapter 3.3.3.5, the signals are due to the hand interactions, namely, to contact between the fingertips of the virtual hand and the scene objects[33];

  The captured signals are:

  - Force signals for each of the three fingers to render force feedback;
  - Identification signals for the specific object in contact to render the correct combination of temperature and texture feedback.

- Force and identification signals are stored and transmitted to the *Control Context* class, as further detailed below.

[33]It's mandatory for these objects to be dynamically enabled or at least respondable.

The *Palpation Strategy* also takes real-time positions of the three fingertips from the haptic device, and they are sent to CoppeliaSim to render them graphically, positioning the robotic hand accordingly. There is a *getter* function to catch these signals in the *hand script*, as described in 3.3.3.4.

The *Control Context* class handles the defined tasks with all the devices. The changes in this class contribute to achieving the goal task by catching the signals sent from the *Strategy* class. The latter, caused by the CoppeliaSim events and captured by the *Strategy*, are set into the abstract object's attributes related to the haptic device. The instantiated elements are object of the *Entity* layer class and are related to the *HapticInterface* and *WeartDevice* classes, which the last one extends hierarchically the first one and both reside in the *Entity* layer. The classes just mentioned were already in the framework. The advantage of using a framework lies in this kind of benefit: the workflow was focused on the implementation of the high-level task, not caring about how the entity device was developed, as it has already been carried out.

On the other hand, the *Task Context* class handles the task chosen by the user in the framework menu application. When launching the application, a menu that allows the user to select which one to start appears (Figure 32). Modifying this pre-existing class made it possible to initiate the *Palpation* strategy with force feedback via the menu interface.

```
1. Registration
2. Interactive Surgical Planning
3. Needle-Tissue interaction parameter identification
4. Friction Estimation Task
5. Force Feedback Palpation Task

0. Quit
```

Figure 32: Framework menu

3.4.1.2 Proxy Layer

A focal point of the work consists of updating the *WeartProxy* class. This class deals with running the haptic device thread, allowing it to get and set values of its interest in runtime. To realize this task, the main changes are as follows:

- The objects associated with the fingers are instantiated and have independent properties. These properties are related to force feedback, textures, temperatures, and their distance in time-of-flight.

- Inside the loop thread, the render effects on the device it is updated by taking the new attribute values of the related finger object. Those values has been updated before in the *Control Context*;

- Considering the identification value of touched object, a specific texture intensity value, the velocity associated with the texture, the temperature value, and the elastic force feedback is assigned.

3.4.2 Feedback transmission

A detailed description of how the information passes through the framework follows, supported by snippets. The information is used for the rendering of the feedback on the haptic device and also for the hand gesture in the CoppeliaSim scene.

The description will start from the data generation, following step by step how it is transmitted through the framework layers until it arrives at the receiver that executes the task.

Regarding the feedback task, the information is generated by CoppeliaSim and is transmitted to the framework to compute the proper feedback to render on the TouchDIVER. Instead, for the hand gesture task, the information is generated by the TouchDIVER and is acquired by the framework to compute the proper pose to render in the CoppeliaSim scene.

The analysis regarding how the information for the feedback render passes through the framework will be divided into the following steps:

- Force feedback analysis at the business logic layer;
- Texture feedback analysis at the business logic layer;
- Analysis of how all the types of feedback (force, texture, and temperature) at the Proxy layer are handled.

3.4.2.1 BLL - Force feedback

In this section, there is a description of how the information is captured and handled at the Business Logic Layer, delving into the *Palpation Strategy* and the *Control Context* classes. Then the analysis moves to the *Proxy Layer* up to the TouchDIVER.

Palpation Strategy

Inside the *TaskStrategy* header file is defined a pointer to a *VREPProxy* object and it is possible to access it by including that file inside the *ForceFeedbackPalpationStrategy* header file. To the pointer inside the *Palpation Strategy*, it is called its *getFloatSignal* method to catch data sent by CoppeliaSim, specifically by the *setFloatSignal* function, the complementary function of the *getter*, REcalled? in the *righthand* script.

Inside the *ForceFeedbackPalpationStrategy* header file, the variables related to the forces and the ID values for each thimble are defined. Each of these variables contains the values sent by the *setFloatSignal* in the text it righthand script of CoppeliaSim.

```
// Palpation force feedback getter
this->vrep->getFloatSignal("force_thumb", this->force_thumb, simx_opmode_streaming, this->simPort);
this->vrep->getFloatSignal("force_index", this->force_index, simx_opmode_streaming, this->simPort);
this->vrep->getFloatSignal("force_middle", this->force_middle, simx_opmode_streaming, this->simPort);
```

Once the values are acquired, a three-dimensional vector of type *VectorXf* is created using the Eigen library. The first element of the vector is related to the thumb, the second to the index, and the third to the middle finger. After its construction, this vector is used to store the three variables of the force relative to the three thimbles. The choice of such a container for this data is for the necessity of complying with the allowed inputs for the *setForceFeedback* function.

```
//Set Force Feedback
Eigen::Vector3f f_feedback;
f_feedback << force_thumb, force_index, force_middle;
this->haptics["TouchDIVER"]->setForceFeedback(f_feedback);
```

This function, member of the *HapticInterface* class, accepts as input a single variable of type *Vector3f*. The *HapticInterface* [34] is a pre-existing class belonging to the *Entity* level. The *setForceFeedback* function is defined in the *HapticInterface* header as follows:

```
inline void setForceFeedback(const Eigen::VectorXf& f) { this->fbForce = f; }
```

Inside the *Palpation Strategy* header file is defined the *haptics* map variable. It represents a list of all possible haptic interfaces to work with, and it is defined as follows:

```
std::map <std::string, HapticInterface* > haptics;
```

Control Context

The value sent from the *Palpation Strategy* through the *setForceFeedback* function, is captured in the *Control Context* class using the *getForceFeedback* function. This value is then saved in the*masterForce* support variable and given as input to the *setHapticForce* function.

```
Eigen::VectorXf masterForce = haptics[hapticName_i]->getForceFeedback();
hapticProxies[hapticName_i]->setHapticForce(masterForce);
```

[34]It is the reference class for the definition of the *WeartDevice*. Specifically, the *WeartDevice* class hierarchically extends the *HapticInterface* class but it was not necessary for the transmission of force values. It will be described in depth during the texture transmission analysis.

Similarly to what has been done for the definition of *haptics* - the *HapticInterface* map variable declared in the *Palpation Strategy* - an *HapticProxies* map variable is defined. This object contains a list of all possible haptic devices to work with and is declared in the *Control Context* class as follows:

```
std::map < std::string, HapticProxy* > hapticProxies;
```

The *setHapticForce* function is defined in the *HapticProxy* header and belongs to the *Proxy* layer. This function performs an assignment to the *hapticState* attribute of the *HapticProxy* object. An instance of the *HapticProxy* class is the abstract representation of the physical haptic device.

```
inline void setHapticForce(const Eigen::VectorXf& f) { this->hapticState.force = f; }
```

The *hapticState* attribute is an object of the *HapticState* data structure and it is declared in the *HapticProxy* header as follows:

```
struct HapticState {
    Eigen::VectorXf force;      // force vector of the kinestetic feedback
    float vibroIntensity;       // intensity of the vibrotactile feedback
    std::array<TextureType, 3> textureType;
};
```

3.4.2.2 BLL - Texture feedback

For the transmission of information about the texture, there are similarities with the force information transmission.

Palpation Strategy

As before, it is applied the *getFloatSignal* function to catch data sent by CoppeliaSim right hand's script.

```
// Texture feedback getter
this->vrep->getFloatSignal("objectsInContact_thumb", this->objectsInContact_thumb, simx_opmode_streaming, this->simPort);
this->vrep->getFloatSignal("objectsInContact_index", this->objectsInContact_index, simx_opmode_streaming, this->simPort);
this->vrep->getFloatSignal("objectsInContact_middle", this->objectsInContact_middle, simx_opmode_streaming, this->simPort);
```

It is constructed a *TextureType* array, a data type defined in *WeArtCommon* class of the Weart's SDK. The *TextureType* class contains a list of all available texture types that can be rendered.

Then is executed a check on the ID values to be then stored in the array mentioned before. A snippet reference for this check only for one thimble follows:

```
//Set Texture Feedback
std::array<TextureType, 3> textureType_array;
TextureType tmp;
switch (static_cast<int>(objectsInContact_thumb)) {
    case 300: //table_id
        tmp = TextureType::SilverOak;
        break;
    case 301: //fat_id
        tmp = TextureType::Leather;
        break;
    default: //all other cases
        tmp = TextureType::ClickNormal;
        break;
}
textureType_array[0] = tmp;
```

Once the array is filled, is passed as input for the *setTextureType* function, belonging to *WeartDevice*. It is necessary to do a cast of data type since the function belongs to the WeartDevice class, not the *HapticInterface* class. Remember that *WeartDevice* extends *HapticInterface* hierarchically.

```
dynamic_cast<WeartDevice*>(this->haptics["TouchDIVER"])->setTextureType(textureType_array);
```

This function has been defined ad-hoc for this project with the aim of transmitting texture information to the proper haptic device object, as happens for the force rendering purpose. It will assign the value taken as input into a *WeartDevice* object's attribute.

```
inline void setTextureType(const std::array<TextureType, 3> tt) { this->textureType = tt; }
```

Control Context

There is an analog procedure for the force information transmission, to communicate the texture information to the Proxy Layer.

```
std::array<TextureType, 3> textureType_array = dynamic_cast<WeartDevice*>(haptics[hapticName_i])->getTextureType();
hapticProxies[hapticName_i]->setTextureType(textureType_array);
```

Notice that the *setTextureType* function here used belongs to the *HapticProxy* class and not to *WeartDevice* as previously explained, even if they have the same name definition.

```
inline void setTextureType(const std::array<TextureType, 3> tt) { this->hapticState.textureType = tt; }
```

3.4.2.3 Proxy Layer - Force, texture and temperature feedback

The *WeartProxy* class extends the *HapticProxy* class hierarchically. In the *WeartProxy* header file is defined a pointer to a *WeArtClient* object, a class defined in the Weart's SDK that deals with the direct communication with the device. Therefore, a *WeArtClient* object is constructed.

```
this->client = new WeArtClient("127.0.0.1", WeArtConstants::DEFAULT_TCP_PORT); //IP ADDRESS and PORT of Middleware PC
```

The *client* object is given as input to the *WeArtHapticObject* constructor, which is a class defined in Weart's SDK and represents the thimble reference for the force feedback rendering.

```
this->haptic_thumb = new WeArtHapticObject(this->client);
this->haptic_index = new WeArtHapticObject(this->client);
this->haptic_middle = new WeArtHapticObject(this->client);
```

Next, the values are assigned to the attributes of the objects just constructed. The attributes are related to the thimble type, such as whether it is the thumb, the index, or the middle thimble, and which hand is considered, such as the right or left hand.

```
this->haptic_thumb->handSideFlag = HandSide::Right;
this->haptic_index->handSideFlag = HandSide::Right;
this->haptic_middle->handSideFlag = HandSide::Right;

this->haptic_thumb->actuationPointFlag = ActuationPoint::Thumb;
this->haptic_index->actuationPointFlag = ActuationPoint::Index;
this->haptic_middle->actuationPointFlag = ActuationPoint::Middle;
```

Furthermore, the objects that handle the specific information related to temperature, force, and texture are created. The ones that handle the force information are *WeArtForce* objects, *WeArtTemperature* objects for the temperature information, and WeArtTexture for the texture information. Also, these three classes as defined in Weart's SDK.

```
// Define feeling properties to create an effect

WeArtTemperature temperature_thumb = WeArtTemperature();
WeArtForce force_thumb = WeArtForce();
WeArtTexture texture_thumb = WeArtTexture();

WeArtTemperature temperature_index = WeArtTemperature();
WeArtForce force_index = WeArtForce();
WeArtTexture texture_index = WeArtTexture();

WeArtTemperature temperature_middle = WeArtTemperature();
WeArtForce force_middle = WeArtForce();
WeArtTexture texture_middle = WeArtTexture();
```

Once these objects are initialized with the proper default values, the objects that handle the real-time update process of these values are constructed. These objects belong to the *TouchEffect* class and it is defined in the Weart SDK. Those objects are the input for the *AddEffect* of the *WeArtHapticObject* function which is in charge of taking the *TouchEffect* object and sending it to the device, to render the proper feedback.

```
// Create default texture behavior
texture_thumb.textureVelocity(0.0f, 0.0f, 0.0f);
texture_thumb.textureType(TextureType::ClickNormal);
texture_index.textureVelocity(0.0f, 0.0f, 0.0f);
texture_index.textureType(TextureType::ClickNormal);
texture_middle.textureVelocity(0.0f, 0.0f, 0.0f);
texture_middle.textureType(TextureType::ClickNormal);

// Instance a new effect with feeling properties and add effect to thimbles

touchEffect_thumb = new TouchEffect(temperature_thumb, force_thumb, texture_thumb);
touchEffect_index = new TouchEffect(temperature_index, force_index, texture_index);
touchEffect_middle = new TouchEffect(temperature_middle, force_middle, texture_middle);

this->haptic_thumb->AddEffect(touchEffect_thumb);
this->haptic_index->AddEffect(touchEffect_index);
this->haptic_middle->AddEffect(touchEffect_middle);
```

After the construction and the initialization of all necessary objects, those are exploited in the loop thread previously mentioned. In the thread, it is executed a cyclic acquisition.

For better clarity, it is recalled that the *Control Context* class was carried out as an assignment to the *HapticProxy*'s *hapticState* attribute through the *setHapticForce* function. In the loop, it is acquired the last value transmitted by CoppeliaSim, thus the last value stored in that attribute.

Focusing on the force, it is also recalled that the *hapticState*'s force attribute is a *VectorXf* data type of dimension three. It follows the snippet for the first element of

that vector, which is related to the thimble of the thumb. It is pointed out that the order starts from zero, thus the $n-th$ element is recalled in the $n-th$ position minus one. The second and the third element are respectively related to the index and the middle finger and the acquisition procedure is depicted in the snippet below.

```
// Update the force values
float f_thumb = this->hapticState.force(0);
f_thumb = (f_thumb < MAX_FLOAT_FORCE_VAL) ? ((f_thumb > -MAX_FLOAT_FORCE_VAL) ? f_thumb : (-MAX_FLOAT_FORCE_VAL)) : MAX_FLOAT_FORCE_VAL;
force_thumb.active = true;
```

As previously anticipated in the last snippet, the force, texture, and temperature objects need to be enabled in order to interact with them and update their attributes. The enabling procedure consists essentially in switching the default value of the objects' *active* boolean parameters, thus from a false to a true value.

```
texture_thumb.active = true;
texture_index.active = true;
texture_middle.active = true;

temperature_thumb.active = true;
temperature_index.active = true;
temperature_middle.active = true;
```

Regarding texture, it is recalled that the *hapticState*'s *textureType* attribute has also been updated in the *Control Context* class. The assignment is made through the *setTextureType* function of the *HapticProxy*. The values are stored in a vector and singly extracted, assigning each element to the corresponding support variables. In order, textures for the thumb, index, and middle thimbles are assigned.

```
TextureType x1 = this->hapticState.textureType[0];
TextureType x2 = this->hapticState.textureType[1];
TextureType x3 = this->hapticState.textureType[2];
```

Subsequently, a check for each thimble is made. The snippet that follows refers only to the thumb, but the approach is equivalent to each thimble. Ensuring that exists a force applied on the thimble, is performed a check on the texture value of the object touched in the scene. The selection criteria are described in chapter 4.

```
if (f_thumb != 0) {
    switch (x1) {
        case TextureType::SilverOak://table
            texture_thumb.volume(WeArtConstants::defaultVolumeTexture);
            texture_thumb.textureVelocity(1.0f, 1.0f, 1.0f);
            temperature_thumb.value(0.3f);
            force_thumb.value(0.0f);
            break;
        case TextureType::Leather://fat
            texture_thumb.volume(WeArtConstants::defaultVolumeTexture);
            texture_thumb.textureVelocity(1.0f, 1.0f, 1.0f);
            temperature_thumb.value(0.7f);
            force_thumb.value(0.0f);
            break;
        default: //all other cases
            texture_thumb.volume(0.0f);
            texture_thumb.textureVelocity(0.0f, 0.0f, 0.0f);
            temperature_thumb.value(WeArtConstants::defaultTemperature);
            force_thumb.value(f_thumb);
            break;
    }
}
else {
    force_thumb.value(f_thumb);
    texture_thumb.textureVelocity(0.0f, 0.0f, 0.0f);
}
```

Once the updates of the attributes are made, the objects related to force, texture, and temperature become the input for the update procedure of the feedback effects on the thimbles.

```
// Set the final touch effect on the device
this->touchEffect_thumb->Set(temperature_thumb, force_thumb, texture_thumb);
this->touchEffect_index->Set(temperature_index, force_index, texture_index);
this->touchEffect_middle->Set(temperature_middle, force_middle, texture_middle);

// Update touch effects
this->haptic_thumb->UpdateEffects();
this->haptic_index->UpdateEffects();
this->haptic_middle->UpdateEffects();
```

At the interruption of the thread, the general reset of all the values occurs.

```
// Reset feedback data

force_thumb.active = false;
force_index.active = false;
force_middle.active = false;

texture_thumb.active = false;
texture_index.active = false;
texture_middle.active = false;

temperature_thumb.active = false;
temperature_index.active = false;
temperature_middle.active = false;

this->touchEffect_thumb->Set(temperature_thumb, force_thumb, texture_thumb);
this->touchEffect_index->Set(temperature_index, force_index, texture_index);
this->touchEffect_middle->Set(temperature_middle, force_middle, texture_middle);

this->haptic_thumb->UpdateEffects();
this->haptic_index->UpdateEffects();
this->haptic_middle->UpdateEffects();
```

3.4.3 Time-Of-Flight transmission

Concerning the time-of-flight, the information flow is reversed, compared with the feedback information flow just described: starting with the input of the thimble position, it is described how this is captured at the Proxy layer and then passed down to the lower layer to be finally transmitted to the running CoppeliaSim scene.

3.4.3.1 Proxy Layer - WeartProxy

Outside the loop thread, the thimble objects - one for each thimble - are created to track its movement and are *WeArtThimbleTrackingObject* data type, defined in Weart's SDK.

```
// Create tracking object to track movements of thimbles
this->tracking_thumb = new WeArtThimbleTrackingObject(HandSide::Right, ActuationPoint::Thumb);
this->tracking_index = new WeArtThimbleTrackingObject(HandSide::Right, ActuationPoint::Index);
this->tracking_middle = new WeArtThimbleTrackingObject(HandSide::Right, ActuationPoint::Middle);
```

These objects are given as input parameters to the *AddThimbleTracking* function of *WeArtClient*, so that is possible to associate them with the client.

```
this->client->AddThimbleTracking(tracking_thumb);
this->client->AddThimbleTracking(tracking_index);
this->client->AddThimbleTracking(tracking_middle);
```

In the loop thread, is called the *GetClosure* function of *WeArtThimbleTrackingObject*, which returns the closure of the associated thimble. These values are assigned to the closure variables defined in the *WeartProxy* header file.

```
//Update closure value
closure_thumb = this->tracking_thumb->GetClosure();
closure_index = this->tracking_index->GetClosure();
closure_middle = this->tracking_middle->GetClosure();
```

3.4.3.2 BLL - Palpation Strategy

The closure values acquired through the *GetClosure* function, are used as the output of the *getter* function defined ad-hoc in the *WeartProxy* header for the purpose of the project. It allows to get the closure values outside the *WeartProxy* class. Thus, it could be possible to send them to CoppeliaSim to render graphically their positions in the virtual space of the scene.

```
float GetThumb() const { return closure_thumb; }
float GetIndex() const { return closure_index; }
float GetMiddle() const { return closure_middle; }
```

Lastly, the updated closure values of the thimbles, are sent to CoppeliaSim giving them the input of the *setFloatSignal* function. This function is the complementary one of the already mentioned *getFloatSignal* function, which allows getting data from CoppeliaSim, thus intuitively allowing to send them.

```
// Send position signal on CoppeliaSim/V-Rep
this->vrep->setFloatSignal("closure_thumb", this->weart->GetThumb(), simx_opmode_oneshot, this->simPort);
this->vrep->setFloatSignal("closure_index", this->weart->GetIndex(), simx_opmode_oneshot, this->simPort);
this->vrep->setFloatSignal("closure_middle", this->weart->GetMiddle(), simx_opmode_oneshot, this->simPort);
```

4 Results

The project aims to create an interaction between virtual and physical reality by connecting TouchDIVER and Oculus in the CoppeliaSim environment 1. As can be appreciated throughout the entire development of the report, it was necessary to proceed step by step to achieve this result. The first approach was to create graphically a robotic hand, by inserting new joints into a pre-existing model 3.3.3.1, making it able to move. Then it was necessary to make the hand able to close: the joints are controlled in position, giving the fingers a path to follow to simulate closing 3.3.3.2. At this stage, the hand could close, allowing a first grasping approach, making the robotic hand interact with a *primitive shape* 18, specifically a cube. Once achieved good results with this first interaction, the focus shifted to the digital twin of the abdomen, succeeding, by way of demonstration, in making graspable one of the organs within it and allowing to feel feedback on it and on the fat that surrounds the phantom.

The haptic device used is the TouchDIVER of Weart which has three types of feedback returned on as many thimbles, namely:

- Force (through a pressure applied by the moving platform 7);
- Texture (through a combination of vibrations);
- Heat:
  - Between the value 0 and 0.49 returns a feeling of cold;
  - When the value is 0.5, it is the default ambient temperature;
  - Between 0.51 and 1.0 returns a feeling of warmth.

Finally, for everything to work consistently and smoothly, were refined the objects' position, orientation, and physical and dynamic characteristics 3.3.1 (namely convexity, respondable masks, dynamics enablement, etc.). The assignment of vibration and temperature feedback follows:

- The table with a texture type *Silver oak*[35] and a cold temperature with 0.3 as the associated value;
- The fat around the phantom with *Texture leather* and a warm temperature with an associated value of 0.7.

The choice of associating a texture with the table and the phantom fat depends on their extent, which makes it appreciable. The temperature was assigned to the same for demonstration purposes only. All other objects respondable and dynamically

[35]Reference is made here to the predefined textures provided by Weart 14

enabled have a default texture called *Texture Click Normal*- which does not return any vibration - and a predefined temperature.

Whereas the force feedback - returning the pressure on the thimbles - indicates that contact occurred between the robotic hand and an object. In the scene on which the project focuses, the only objects causing this feedback are the right kidney and the cube 3.3.3.4. Indeed, for a computational issue, these characteristics have only been applied to a limited number of items otherwise, the scene becomes too heavy, and VR rendering becomes too slow to process.

4.1 Possible Improvements

The joints of the robotic hand are controlled in position to guarantee that the path followed is precise and corresponds to the semi-cardioid path that comes closest to the movement of a real finger 2.1. One might consider controlling the joints in force to free the hand from the constraint of following the path, leaving the finger free to move and thus increasing the possible movements. It would also avoid setting the position of the robotic hand on the Oculus controller since it would no longer be exclusively subject to the force of gravity, causing it to fall, but also to its own force counteracting it. That would make the rendering in VR more precise and responsive, and moreover, it would be more akin to what happens in reality.

Another possible change worthy of attention could be making the texture and, consequently, the vibratory feedback dependent on the speed of the hand touching the object rather than constant. Indeed, to date, a textured object returns feedback that remains unchanged over the entire duration of the interaction, regardless of the speed of the robotic hand touching it.

The same applies to making the force feedback not constant, rendering the pressure with which you grip an object. In fact, as with the texture, the force feedback remains constant regardless of the force with which the user grasps the object.

Another possible development arises from the new version of CoppeliaSim recently released. In the latter, Coppelia Robotics introduced deformable objects. This new feature undoubtedly opens up several possibilities, such as making the phantom and its internal organs malleable and then acting on their new deformation property, typical of human tissue.

Lastly, according to today's instrumentation, the TouchDIVER does not allow for wrist tracking, which instead is done through the Oculus controller. The latter is attached to the haptic device by means of a magnetic attachment and a velcro strap: this solution is objectively uncomfortable, heavy, and limiting for the user's movements. If, on the other hand, the haptic device itself could track in the wrist space, the user would be largely freer to move.

5 Conclusions and future works

In conclusion, we believe that our project introduces novelties: to date, no link between touchDIVER and CoppeliaSim can be found in the literature. Thanks to this integration, this project can be the basis for interesting future developments in research and the medical robotics field, with an emphasis on simulations.

Indeed, young physicians or medical students could exploit the integration mentioned above in conjunction with virtual reality (as already planned in our project) for study reasons - which would be more realistic and immersive - as to interface with training and testing already during their studies or from the first years of specialization. In fact, in the *Daily Biomed Magazine*, in an article written the 6th of March 2022, it can be read that: *In Virtual reality projects, the surgeon into a virtual safe environment within which the surgeon himself can interact and control his every movement- can learn and refine the operating technique in total safety.* [53]

Thus, a student or novice surgeon can confront real situations, but in VR, allowing them to refine their techniques without having a first impact directly with a physical patient. Of the same opinion is author Maria Tsarouva of iTechArt, who wrote last year: *(...), a recent University of Michigan study found that 30% of surgeons weren't able to operate independently after completing their residencies. However, today's surgeons in training aren't limited to training on cadavers, as the medical students of the past were. Today, med students often participate in VR simulations to help them understand what it feels like to operate on a patient. VR surgical training gives students a hands-on, immersive experience to develop their skill sets in the operating room on complex cases, without assuming any risk of causing injury to their patients. And it provides a superior learning experience to traditional hands-on training: A 2019 study from the University of California, Los Angeles, found that medical students trained on VR simulations were able to complete an orthopedic procedure 20 percent faster than traditionally trained students, and they completed 38 percent more steps correctly.* [54]

Moreover, on the other hand, an expert surgeon can plan a complicated operation by performing various tests before working on the actual surgery to find the best and least invasive strategy to deal with the surgery. This testing phase can then lead to a more precise calculation of the risk of surgery since for *risk assessment (RA), (...), and professional training, VR technology offers major advantages*[55].

Furthermore, new, infrequent, or even surgeries too risky to be considered realistic may become feasible, as virtual reality can lead to solutions not yet pursued because of its property of repeatability.

Thinking then of more distant but uniquely important developments, one could think of using our project for diagnosis: clearly, not as a replacement for the doctor's clinical experience, but as a supporting tool. For example, virtual reality can be used to visualize medical images in a more interactive and detailed manner, enabling doctors

to diagnose more accurately and quickly. Haptic technology, instead, can help doctors detect abnormalities or pathologies that might be difficult to detect otherwise. Lastly, CoppeliaSim is a simulation software that offers a wide variety of scenarios that can be tailored to specific case studies.

Along the same lines, it could allow doctors to make remote visits: for example, by combining the interaction we introduced with CT, ultrasound, and a detailed digital twin of a patient, a physician who is even miles away could analyze his symptoms and study his anatomy to make precise diagnoses. That would allow them to quickly access a consultation with a specialist without travelling or relocate, and this would reduce the cost of treatment and allow those who cannot travel due to medical conditions, to access the best treatment for their disease.

Today, the technological development of fields such as artificial intelligence and robotics, together with the concrete development of sophisticated and highly professional equipment, is leading to changes and solutions in multiple fields, that were unthinkable before.

These improvements are also being reflected in the field of medical research and innovation, making available to healthcare professionals, software and tools that make diagnoses and operations faster, operations that in some cases were not feasible before, while also taking into account a focus on patients, who now have high-precision but very low-invasive treatments available.

This project could be the basis for future research and implementations that could enrich the current panorama of tools and solutions available to health workers and students in the sector. The latter will thus have the opportunity to be trained in the best way, becoming increasingly prepared at the meeting with the patient, planning and searching for the best strategy for the case they are examining.

References

[1] Greg Ruthenbeck and Karen Reynolds. Virtual reality for medical training: The state-of-the-art. *Journal of Simulation*, 07 2014.

[2] Wael Othman, Zhi-Han Lai, Carlos Abril, Juan Barajas-Gamboa, Ricard Corcelles, Matthew Kroh, and Mohammad Qasaimeh. Tactile sensing for minimally invasive surgery: Conventional methods and potential emerging tactile technologies. *Frontiers in Robotics and AI*, 8, 01 2022.

[3] Weart, 2023. About - Weart.

[4] Meta Quest, 2023. Oculus RIFT S.

[5] Wikipedia, December 2022. CoppeliaSIM.

[6] Hoshi M. Oebisu N. Takada N. Ban Y. Nakamura H Shimatani, A. An analysis of tumor-related skin temperature differences in malignant soft-tissue tumors. *International journal of clinical oncology*, 27(1):234–243, 2022.

[7] Mohammad Motaharifar, Alireza Norouzzadeh, Parisa Abdi, Arash Iranfar, F. Lotfi, Behzad Moshiri, Alireza Lashay, Seyed-Farzad Mohammadi, and Hamid Taghirad. Applications of haptic technology, virtual reality, and artificial intelligence in medical training during the covid-19 pandemic. *Frontiers in Robotics and AI*, 8:612949, 08 2021.

[8] David Escobar-Castillejos, Julieta Noguez, Luis Neri, Alejandra Magana, and Bedrich Benes. A review of simulators with haptic devices for medical training. *Journal of Medical Systems*, 40, 02 2016.

[9] D. G. Kamper, E. G. Cruz, and M. P. Siegel. *Journal of Neurophysiology*, 90(6):3702–3710, 2003. Stereotypical Fingertip Trajectories During Grasp.

[10] John I. Calle-Sigïencia, Gabriel A. Encalada-Seminario, and Rodrigo A. Pinto-León. Design and kinematic analysis of a biphalange articulated finger mechanism for a biomechatronic hand prosthesis. In *2018 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*, pages 1–7, 2018.

[11] C. R. Mason, J. E. Gomez, and T. J. Ebner. *Journal of Neurophysiology*, 86(6):2896–2910, 2001. Hand Synergies During Reach-to-Grasp.

[12] Marco Santello, Gabriel Baud-Bovy, and Henrik Jörntell. Neural bases of hand synergies. *Frontiers in Computational Neuroscience*, 7, 2013.

[13] Giuseppe Oriolo Bruno Siciliano, Lorenzo Sciavicco Luigi Villani. *Robotics: Modelling, Planning and Control*. Springer Nature, 2010.

[14] PRISMA Lab. Redundant Manipulators.

[15] Alessandro De Luca, March 2023. Robots with kinematic redundancy. Part 1: Fundamentals.

[16] Alessandro De Luca, December 2022. Inverse differential kinematics Statics and force transformations.

[17] Adilzhan Adilkhanov, Matteo Rubagotti, and Zhanat Kappassov. Haptic devices: Wearability-based taxonomy and literature review. *IEEE Access*, 10:91923–91947, 2022.

[18] Carlos Bermejo and Pan Hui. A survey on haptic technologies for mobile augmented reality, 2017.

[19] American Psychological Association. Apa dictionary of psychology, mechanoreceptor, 2023.

[20] Irisdynamics, 2022. What is force feedback.

[21] Nima Enayati, Elena De Momi, and Giancarlo Ferrigno. Haptics in robot-assisted surgery: Challenges and benefits. *IEEE Reviews in Biomedical Engineering*, 9:1–1, 03 2016.

[22] Claudio Pacchierotti, Stephen Sinclair, Massimiliano Solazzi, Antonio Frisoli, Vincent Hayward, and Domenico Prattichizzo. Wearable haptic systems for the fingertip and the hand: Taxonomy, review, and perspectives. *IEEE Transactions on Haptics*, 10(4):580–600, 2017.

[23] Claudio Pacchierotti, Asad Tirmizi, and Domenico Prattichizzo. Improving transparency in teleoperation by means of cutaneous tactile force feedback. *ACM Transactions on Applied Perception*, 11, 04 2014.

[24] Chiara Gaudeni, Leonardo Meli, Lynette A. Jones, and Domenico Prattichizzo. Presenting surface features using a haptic ring: A psychophysical study on relocating vibrotactile feedback. *IEEE Transactions on Haptics*, 12(4):428–437, 2019.

[25] Andualem Tadesse Maereg, Atulya Nagar, David Reid, and Emanuele L. Secco. *Frontiers in Robotics and AI*, 4, 2017. Wearable Vibrotactile Haptic Device for Stiffness Discrimination during Virtual Interactions.

[26] Kaushik Parida, Hyunwoo Bark, and Pooi See Lee. *Advanced Functional Materials*, 31(39):2007952, 2021. Emerging Thermal Technology Enabled Augmented Reality.

[27] Roshan Lalintha Peiris, Wei Peng, Zikun Chen, Liwei Chan, and Kouta Minamizawa. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, page 5452–5456, New York, NY, USA, 2017. Association for Computing Machinery. ThermoVR: Exploring Integrated Thermal Haptic Feedback with Head Mounted Displays.

[28] Hsin-Ni Ho and Lynette Jones. Development and evaluation of a thermal display for material identification and discrimination. *TAP*, 4, 07 2007.

[29] Matteo Lorenzetti, September 2019. Oculus Rift S - recensione.

[30] Gavin Wright, September 2022. What is screen door effect?

[31] Oskar Linde Joel Hesch, Anna Kozminski, August 2019. Powered by AI: Oculus Insight.

[32] David Heaney, May 2019. Tracking systems 'Constellation' (Original Oculus Rift).

[33] Andrew Melim, September 2019. Increasing Fidelity with Constellation-Tracked Controllers.

[34] Ananth Ranganathan, June 2022. The Oculus Insight positional tracking system.

[35] Weart, April 2023. Weart — Linkedin.

[36] WEART s.r.l. Haptic Ring.

[37] Weart s.r.l., February 2023. A Haptic Glove for Virtual Reality.

[38] Fabio Pizzato, June 2022. Weart : chaud, froid, sensations tactiles en VR ! [Laval Virtual 2022].

[39] CoppeliaSim User Manual, March 2023. Designing dynamic simulations.

[40] CoppeliaSim User Manual, March 2023. User Interface.

[41] CoppeliaSim User Manual, March 2023. Shapes.

[42] Boris Bogaerts, 2019. CoppeliaSim VR Toolbox.

[43] TurboSquid, 2023. Professional Free 3D Robot Hand Models.

[44] Nicolas Lauzier, May 2016. Robot Force Control: An Introduction.

[45] CoppeliaSim User Manual, March 2023. Joint modes.

[46] CoppeliaSim User Manual, March 2023. Threaded and non Threaded Code.

[47] CoppeliaSim User Manual, March 2023. Paths.

[48] Coppelia Robotics Forum, May 2021. Dummy's problems of following an open path.

[49] CoppeliaSim User Manual, March 2023. Solving Inverse/Forward Kinematics Tasks.

[50] CoppeliaSim User Manual, March 2023. IK plugin API reference.

[51] CoppeliaSim User Manual, March 2023. GetContactInfo function.

[52] Marilena Vendittelli Marco Ferro, Claudio Gaz. A framework for sensorless identification of needle-tissue interaction forces in robot-assisted biopsies, June 2020. ICRA 2020 Workshop on Shared Autonomy: Learning and Control (SALC), Virtual conference.

[53] Close-Up Engineering, March 2022. CUENews-biomed.

[54] Maria Tsarouva, January 2022. iTechArt.

[55] Patrick Puschmann, Tina Horlitz, Volker Wittstock, and Astrid Schütz. Risk analysis (assessment) using virtual reality technology - effects of subjective experience: An experimental study. *Procedia CIRP*, 50:490–495, 2016. 26th CIRP Design Conference.

[56] CoppeliaSim User Manual, March 2023. Shape dynamics properties.

[57] Timothy R. Coles, Nigel W. John, Derek Gould, and Darwin G. Caldwell. Integrating haptics with augmented reality in a femoral palpation and needle insertion training simulation. *IEEE Transactions on Haptics*, 4(3):199–209, 2011.