TRAJECTORIES PROGRAMMING FOR INDUSTRIAL ROBOTS THROUGH AUGMENTED REALITY DEVICES

Author:
Matteo Tessarotto

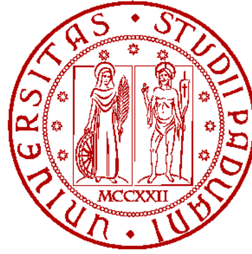
Supervisor:
Dr. Stefano Ghidoni

Company's supervisor:
Ing. Roberto Polesel

Master's Degree in Computer Engineering

Academic Year 2016/2017

Padova, 09 October 2017



UNIVERSITY OF PADOVA
DEPARTMENT OF INFORMATION ENGINEERING
MASTER'S DEGREE IN COMPUTER ENGINEERING

TRAJECTORIES PROGRAMMING FOR INDUSTRIAL ROBOTS THROUGH AUGMENTED REALITY DEVICES

SUPERVISOR: Stefano Ghidoni

COMPANY'S SUPERVISOR: Roberto Polesel

AUTHOR: Matteo Tessarotto

Padova, 09 October 2017

Abstract

The purpose of this work is to describe the realization and use of a software for the recording and execution of trajectories for industrial robots, in particular for its use in some real applications for the company Euclid Labs. After a brief introduction of the discussed topics, the detailed studies of the used devices will be illustrated underlining their precision and use. A description of the software realization and its many features will follow. Finally some real applications for which the project has been used are shown, highlighting the improvements over previously used solutions and possible future improvements.

Abstract (Italian)

Questo elaborato ha l'obiettivo di descrivere la realizzazione e l'utilizzo di un programma per la registrazione ed esecuzione di traiettorie per robot industriali, in particolare per il suo utilizzo in alcune applicazioni reali per l'azienda Euclid Labs. Dopo una breve introduzione degli argomenti trattati, si illustrano gli approfonditi studi effettuati sui device utilizzati, sulla loro precisione e possibilità di utilizzo. Si passa quindi alla descrizione della realizzazione del programma e delle sue molteplici funzionalità. Vengono infine illustrate alcune applicazioni reali per cui il progetto è stato utilizzato, mostrandone i miglioramenti rispetto alle soluzioni utilizzate precedentemente e i possibili miglioramenti futuri.

Contents

1 Introduction **3**
1.1 Programming industrial robots 3
1.1.1 On-line programming 4
1.1.2 Off-line programming 5
1.2 Virtual Reality and Augmented Reality in robotics 6
1.3 Tracking solutions in modern Virtual Reality systems 10
1.3.1 Oculus Rift Constellation 10
1.3.2 Microsoft/Acer Inside-Out tracking 11
1.3.3 Valve Lighthouse tracking system 12

2 Tracking System: working details and testing **15**
2.1 The system in-depth 15
2.1.1 Base-Stations 16
2.1.2 Tracked Objects 17
2.1.3 System calibration 19
2.2 Testing system performance 22
2.2.1 Residual noise test 22
2.2.2 Precision test 24
2.2.3 Accuracy test 25
2.2.4 Accuracy confirmation, tracked volume and occlusion 27
2.3 Future system developments 35

3 Trajectories programming, developed software **37**
3.1 First test program 37
3.2 Full off-line programming package development 43
3.2.1 Recording trajectories 44
3.2.2 Real-time trajectories projection 45
3.2.3 Showing and modifying trajectories 48
3.2.4 Sending trajectories to the robot 50

3.2.5 Diagnostic tools 50

4 Real-life applications **53**
4.1 Jeans Water Brush (prototype) 53
4.2 Sofas gluing 58

5 Conclusions **59**

List of Figures

1.1 GUI for paths definition realized in [6]. 4
1.2 Steps of off-line programming 5
1.3 VPL EyePhone and DataGlove (a) and VPL DataSuit (b). 7
1.4 Robot programming system created in [15]. 8
1.5 Interactive editing of the projected trajectory realized in [21]. . . . 9
1.6 The 3 main competitor in the VR market today. 10
1.7 The array of infrared leds on the Oculus Rift DK2. 11
1.8 The Microsoft/Acer headset with its controllers. 12
1.9 Prototype of the Valve HMD and tracking system. 13
1.10 The HTC Vive with its controller and base-stations. 14

2.1 Internal components of a base-station [1]. 16
2.2 The sudden changes in the reported position are the laser tracking corrections over the *dead reckoning* method[2]. 18
2.3 First step of the calibration process. 19
2.4 Second step of the calibration process. 20
2.5 Third step of the calibration process. 20
2.6 Fourth and last step of the calibration process. 21
2.7 Residual noise with both lighthouses (a) and with only one (b)[3]. . 23
2.8 The retail box of the HTC Vive. 24
2.9 The sheet of paper used for accuracy measures. 25
2.10 Distribution of errors in the measures with the lighthouse system. 27
2.11 The Kreon Baces arm used for the tests. 28
2.12 Tracked volume is greater than expected, all positions are still being tracked. 31
2.13 Error message showed by the system when base-stations do not have free line of sight between them. 32
2.14 The Vive Tracker used for several accessories. 33
2.15 The new Valve's Knuckles Controllers compared to the Vive Controllers. 34

2.16 Base-stations update with SteamVR Tracking 2.0. 35

3.1 First test of inverse kinematic using a HTC Vive Controller and a Fanuc R-1000iA/80F Max robotic arm.[4] 38
3.2 The virtual world in the developed program. 43
3.3 The frame teaching form. 44
3.4 The object definition frame. 46
3.5 The projection window[5]. 47
3.6 Visualization of a recorded trajectory. 48
3.7 The filter in action, with a 30 mm distance threshold (a) and with only 1 mm distance threshold (b). 49
3.8 The computer send a robot program to the robot controller which will move the robot accordingly. 50
3.9 Pop-up menu of the program and Arm status tool. 51
3.10 TeachPendant tool. 51

4.1 On-line robot programming in the first version of Water Brush [6]. . 53
4.2 Tracking device for off-line programming in the Water Brush system, old system (a) and new system (b). 54
4.3 The software realized for Tonello[7]. 56
4.4 The robot in the Water Brush system executing a trajectory. . . . 56
4.5 A virtual tour of the robotic cell realized for Tonello. 57
4.6 Manual gluing of a cushion [8]. 58

List of Tables

2.1 Residual noise of the Valve “Lighthouse” system with HTC Vive. 22
2.2 Subsequent measurements of the position of the controller. 25
2.3 Accuracy of Valve’s “Lighthouse” system on all three axes, all values in mm. 26
2.4 The precision and accuracy of the Kreon Baces arms from [41]. . . 27
2.5 Variances in 10 measures of the same point with Baces arm and HTC vive controller. 30
2.6 Errors of the lighthouse system compared to the Baces arm. . . . 30

1 Introduction

Industrial robots have long been used in a variety of manufacturing processes, where human intervention is dangerous or task repetitiveness leads to an increase in productivity and efficiency through its automation.
When robots are used for big manufacturing processes, repeating the same task for long periods of time, their programming can take months of work from specialized operators, while the actual cycle time is in the order of hours. Small to medium sized enterprises (SMEs) and big enterprises operating in different fields, where products change quickly, normally lack specialized operators and cannot benefit from robotic automation due to this programming time overhead.
The number of areas where automation is integrated rose vertiginously in recent years and is projected to replace human operators in nearly every field.
This trend leads to a demand for easier and safer ways to program robots.

1.1 Programming industrial robots

The number of programming methods for industrial robots increased through the years, with new approaches studied and researched to improve the simplicity or efficiency of the programming over older methods. A comprehensive review of research progress on this field in the last decade is discussed in [1].

The two main categories of robotics programming methods are:

- On-line programming (including lead-through and walk-through);
- Off-line programming (OLP)[2];

Both with their strengths and weaknesses, those categories received a lot of attention from researchers around the world thanks to the incredible growth of automation in recent years.

1.1.1 On-line programming

The first category is the simplest from a programming stand-point and has a lower initial cost. It usually consists in a manual approach where the operator moves the end-effector to the desired position and orientation at each stage of the robot task. Relevant robot configurations are recorded by the robot controller; a robot program is then generated to command robot movement through the recorded end-effector poses.
There are many drawbacks in using this approach. Guiding the robot arm with its elevated degree-of-freedoms (DOFs), maintaining the desired position and orientation while avoiding any collision, is a very difficult and time-consuming task, in particular for articulated workpieces. In addition, a lot of testing is required for the generated program before it meets precision and safety requirements; it also lacks re-usability and flexibility, the process has to be repeated for every new workpiece even if similar. Robots cannot be used for production during the teaching period and operators are exposed to a hostile environment while recording end-effector positions.
Despite all these drawbacks, on-line programming is, in most cases, the only solution for smaller enterprises and many different approaches have been elaborated. Those approaches can be separated in two main categories: "operator assisted on-line programming" and "sensor guided on-line programming". Examples in the first category used a force/moment direction sensor for moving the robot arm more easily [3], or introduced two teaching support devices on the measuring unit tip, to measure position and direction vectors of the dummy tool. This information was then used to generate the robot program [4]. In the second category instead, "higher level" sensors were used. For example in [5] a visual servoing approach was used, enabling the robot arm to follow a path marked directly on the workpiece with a standard marker pen. In [6], instead, users interact directly with an image of the workpiece defining the path to follow (Figure 1.1).

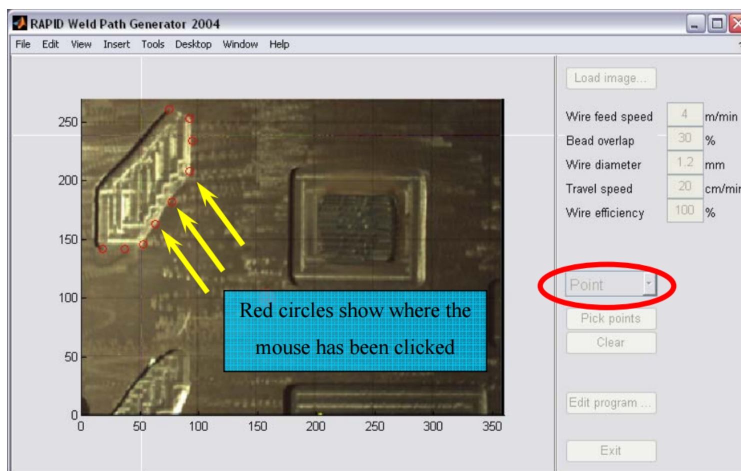


Figure 1.1: GUI for paths definition realized in [6].

Even with all the research done in this field, many of the outcomes are not

commercially available, mostly because of their specific approaches that cannot be generalized. With new cost-effective sensors, the "sensor guided on-line programming" approach can produce solutions for many of SMEs demands, possibly gaining a big market share.

1.1.2 Off-line programming

Off-line Programming (OLP) uses 3D models of the workpiece, robot and robot cell to generate and simulate robot programs. OLP methods have the following key steps:

- *Generation of 3D CAD model*, with a 3D scanner from the real workpiece or created directly in a CAD program
- *Tag creation*, extracting the robot position tags from the 3D CAD model with a specific Tool Center Point (TCP)
- *Trajectory planning*, selecting from the multiple configurations that the reverse kinematic returns for the points in the programmed trajectory
- *Process planning*, for cooperation of multiple robots in complex manufacturing process to minimize cycle time
- *Post-processing*, to add external I/O for additional hardware and to convert the program in the specific robot language
- *Simulation*, allowing for verification of the generated program without the use of an actual physical robot
- *Calibration*, for solving eventual difference from the 3D models and the actual geometry of the pieces in the robot cell.

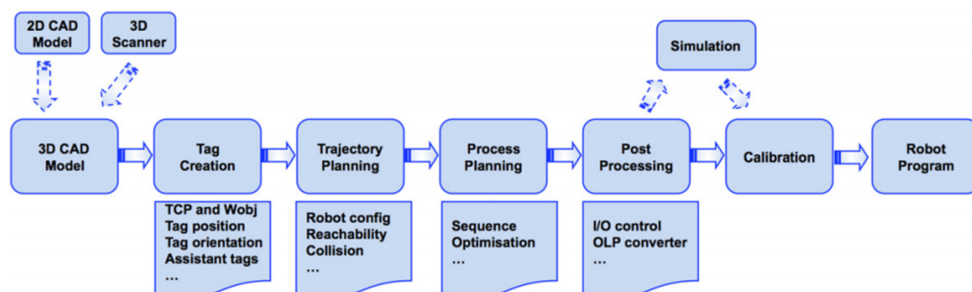


Figure 1.2: Steps of off-line programming

Many OLP packages exist; almost every robot manufacturer has its own OLP software, like KUKA-Sim and Camrobot from KUKA [7], MotoSim from Motoman [8], Roboguide from Fanuc [9] and RobotStudio from ABB [10]. There are

also some generic OLP packages like Delmia [11], RobCAD [12], Robomaster [13] that provide interfaces for many robots from different manufacturers.
Methods that use an OLP approach offer many advantages over on-line programming methods; realizing new programs for a robot without blocking its production, reducing dramatically the robot down-time, is one of the most important. Programs realized with OLP methods are also more flexible and some of them can be reused for different workpieces with little changes. This approach usually incorporates also a simulation of the movements, allowing to check for collisions or other errors without using the real robot.
The biggest problem of OLP is its cost, both for specialized operators that know how to use an OLP package and for the OLP package itself. For SMEs this cost and the programming overhead, required to customize the software for a specific application, is way too high to be profitable.

1.2 Virtual Reality and Augmented Reality in robotics

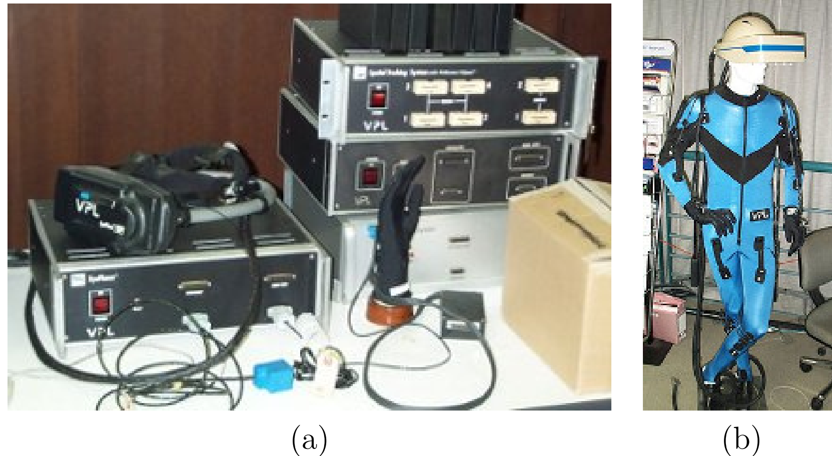
Virtual reality (VR) is an high-end human-computer interface that allows user interaction with simulated environments in real time and through multiple sensorial channels [14]. Interaction between users and simulated environment is mediated by input/output devices that allow to track and show (usually in a HMD[1]) the user's movements inside a virtual world. The VR engine responds to user's movements by moving objects and the view in the virtual environment accordingly, immersing the user in the virtual world.

Virtual Reality systems exist long before anyone expects. The term "virtual reality" was coined by Jaron Lanier whom founded VPL Research in 1984. The company was one of the first ever to sell virtual reality products, including a programming language to develop application for it. Some of the innovative products developed by VPL Research include:

- The EyePhone, one of the first head-mounted display unit similar to those we see today. It used two LCD screens to provide a slightly different image to the eyes and give the illusion of 3D. Figure 1.3 (a).
- The DataGlove, a device which uses a glove as a form of input. Virtual reality gloves are one the most popular symbols of virtual reality as a whole. Figure 1.3 (a).
- The DataSuit, a full-body outfit with sensors for measuring the movements of arms, legs, and trunk. Figure 1.3 (b).
- Body Electric, a visual programming language used to control and program all other components as part of the whole VPL virtual reality experience.

[1] Head Mounted Display

- Isaac, a real-time 3D visual rendering engine, which is controlled and works with Body Electric to create the virtual environment.
- The AudioSphere, a unit which uses stereo to create the illusion of 3D sounds.



(a) (b)

Figure 1.3: VPL EyePhone and DataGlove (a) and VPL DataSuit (b).

This ecosystem for virtual reality was light-years ahead of anything seen before, image quality and movements tracking were good enough for that time. The system cost of approximately 1 million of today dollars was the limiting factor in its diffusion, it was relegated to only a couple of universities causing the company's bankruptcy in 1990 and the sale of all its patents to Sun in 1999 [24] [25]. The high cost was the main cause for the failed attempts of many others commercial projects in the virtual reality market through the years.

VR is used in many different applications, from architectural modeling and manufacturing plant layout, to training in servicing equipment and, in recent years, for entertainment and gaming. One of the major applications for VR in the last decade is robotic and in particular CAD design, teleoperation and robot programming [15]. Researchers tried to simplify the use of OLP through Virtual Reality. One of the early examples of this approach is [16], where programming is done on a virtual robot in a virtual environment, with programmer interaction being mediated by VR I/O devices (sensing glove, tracker, HMD, etc.). The programmer feels immersed in the application where he can move and look at the scene from any angle specifying a trajectory with just an hand gesture.

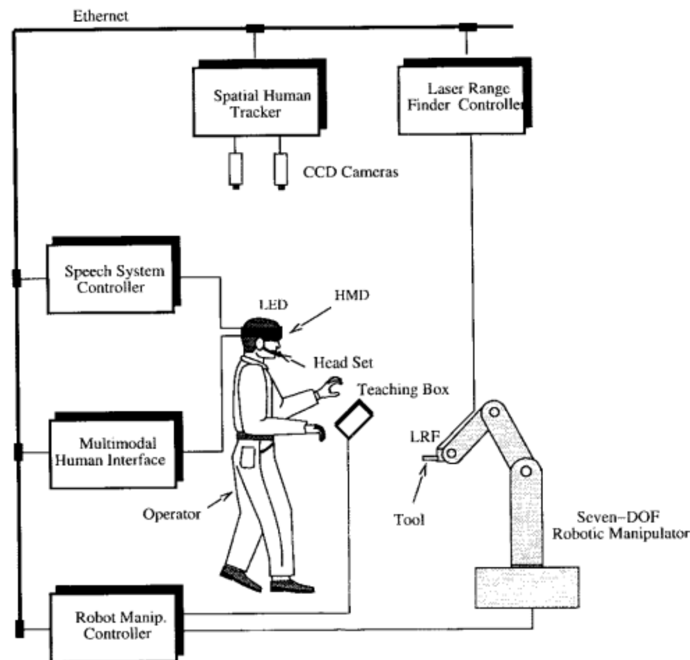


Figure 1.4: Robot programming system created in [15].

Usually the VR approach for off-line programming of industrial robots consists in a complete modeling of the robot cell in the virtual world. Inside this virtual world users can interact with the robot to visualize, or even program, robot trajectories. In this way the main advantages of off-line programming remain: the real robot can continue its work without interruption and programmed trajectories can be tested in the virtual world with good precision if models are accurate. It also reduces the need for specialized operators thanks to a better usability compared to a standard off-line programming package, when the software is fully functional at least. The total cost for this solution is higher compared to a traditional OLP package, due to the added complexity of the VR engine and of needed additional hardware.

A different approach derived from VR is Augmented Reality (AR). Like VR, AR has been used in many different applications, some examples are maintenance [17], manual assembly [18] and computer-assisted surgery [19].
The main difference between AR and VR is that AR does not need a model of the entire environment as it supplements real world instead of replacing it. Additional informations are shown on a display overlapping an image of the real world [20] or projected directly on real world objects [21].

Using this approach there is no need to create a model for the whole robot cell, preserving advantages of both OLP and VR approach while lowering workload for both expert and novice operators [22]. Costs and time needed for modeling the entire robot cell are also removed from the process. The downsides of this

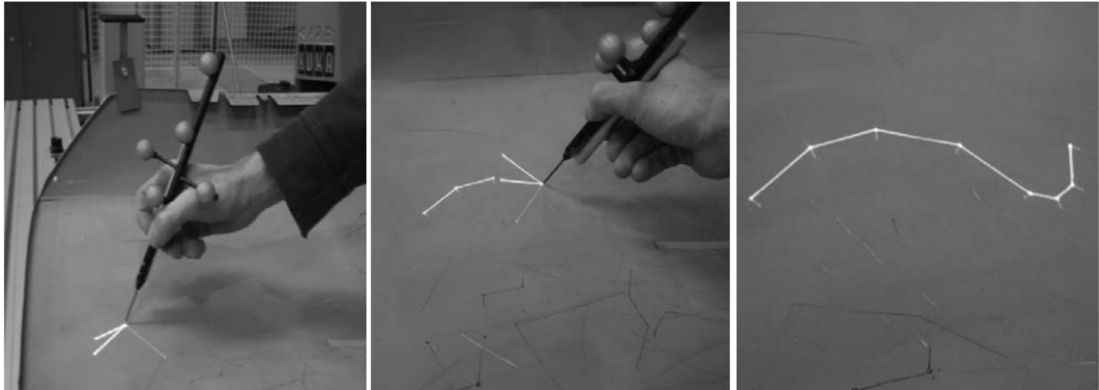


Figure 1.5: Interactive editing of the projected trajectory realized in [21].

approach include additional complexity of the software and an higher cost of the hardware for tracking user's movements in space.

Both VR and AR approaches for robot programming have the same problem, tracking the user's movements in space. This problem has been addressed in several ways, one of the most used solution is to use one or more cameras and a specific marker placed on the tracked object. Using standard vision algorithms is then possible to track that marker, and consequently the object, in space.
With this approach, tracking has a relatively low precision and cannot track object orientation, only its position. There are solutions that use multiple markers or "fiducials" to track also the object orientation ([23] for example uses this approach). These tracking solutions have a high cost considering both hardware and software that need to be used and realized from scratch. Systems realized in this manner are also usually limited to the specific application for which they were made. An economic and ready solution for tracking user's movements could change drastically this type of approaches, with lower costs and easier realizations.

Some "off-the-shelf" solutions for the aforementioned tracking problem are now available thanks to modern virtual reality systems. These systems, realized for video games and entertainment, have been recently introduced in the market and could revolutionize the off-line robot programming thanks to their low price compared to equal "home-made" solutions.
Thanks to these systems many robotic applications can be improved in combination with a software for robot programming. Some of these applications will be discussed in the chapters 3 and 4 of this work, with an off-line programming software for the recording and execution of trajectory for a painting robot and the integration of one of these systems for a spraying robots that bleach jeans with water.

1.3 Tracking solutions in modern Virtual Reality systems

Three different approaches used to solve the tracking problem in modern virtual reality systems will be analyzed in this section. Keep in mind that for this work, which uses the AR approach, only the tracking technologies of these systems are important, for this reason the following analysis will focus only on this aspect ignoring the HMDs which are not used.

The modern era of affordable Virtual Reality systems started with the Kickstarter campaign of Oculus, presenting the Rift DK1 [26] in 2012. The campaign raised $2,437,429 from a goal of just $250,000 and offered an HMD with a 7” 1280×800 pixel display and used a Hillcrest Labs 6DoF “Invensense MPU-6000” head tracker (3 axis accelerometer, 3 axis gyroscope) at 250Hz for a price as low as 300$. This solution did not provide a true tracking of the head position, only of its orientation. A real tracking solution was added in a later version of the Rift. After the Rift DK1 more systems arrived on the market: systems like HTC Vive, Sony PlayStationVR, Microsoft/Acer Headset in the near future and many others. These systems use different approaches for solving the tracking problem. Sony solution is similar to the one used in the Oculus Rift, but how do they work?

Figure 1.6: The 3 main competitor in the VR market today.

1.3.1 Oculus Rift Constellation

Oculus Rift’s first real tracking solution was integrated in the second version of the development kit (DK2), released in July 2014. It used an array of infrared leds (Fig. 1.7) and a standard web-cam with an infrared filter, combined with the data from an inertial measurement unit (IMU) to perform the tracking.
Each led on the HMD broadcasts a specific flashing frequency with various levels of brightness. Tracking cameras can identify the source of each light identifying the exact blinking pattern [27]. Knowing where each led is, allows the system to track the head position and not only its orientation.
With the following version of the headset, Oculus added some leds into the rear of the headset to have a 360° tracking. In the final version of the system released in April 2016, even though the camera has an higher resolution, the same technology

Figure 1.7: The array of infrared leds on the Oculus Rift DK2.

is used.
After the introduction of the HTC vive with its tracked controllers and "room-scale" experience, Oculus had to close the gap and introduced, later in 2016, the Oculus Touch: two controllers tracked with the same technology as the HMD.
To track both HMD and controllers in a large enough area, Oculus needed to add more cameras, at least 3 or 4. Even with the increased number of cameras, this tracking solution is not perfect; it has various problems of occlusion and loss of tracking, and increasing the distance from the cameras its precision deteriorates rapidly, limiting the size of the tracked area to a couple of meters.
This technology is proprietary so its details are not known (even though, with some reverse engineering, most of its functioning is now known [28]) and there are not official values for the tracking precision. Oculus also does not provide an API to use this technology in projects outside the video games industry.

1.3.2 Microsoft/Acer Inside-Out tracking

In May 2017, during the Microsoft Build 2017 conference, Microsoft presented a new "low-cost" VR headset in collaboration with Acer that will be available later this year. Other similar products from Dell, Lenovo, ASUS, and HP that use the same technology will also be sold.
The tracking solution of this product is somewhat different from the one in the Oculus Rift. Controllers, like the Oculus Touch, have several infrared leds that are tracked using a camera in a similar way to the Oculus solution. The main difference is in the camera position: instead of having one or more cameras placed

Figure 1.8: The Microsoft/Acer headset with its controllers.

in the environment (with all problems of correct positioning and cables management), it uses two cameras placed directly in the HMD front. The HMD itself uses these cameras to track controllers position and to find its own position in the real world using object recognition algorithms on the surroundings without needing any external sensors (for this reason it is called "inside-out" tracking).
This solution trade-offs the simplicity of the system deployment for a lack of controllers tracking when they are out of the line of sight and a lower precision compared to other solutions, at least for now.

1.3.3 Valve Lighthouse tracking system

Valve, one of the biggest company in the video games industry, showed a prototype of its work in virtual reality during 2014. At that time the prototype used several markers placed around the room and a camera on the headset itself to track its movement in the space (Fig. 1.9), in a similar fashion to what Microsoft's Inside-Out tracking promise to do without markers.
During the Mobile World Congress keynote on March 2015 HTC unveiled its device created in collaboration with Valve, the HTC Vive, which arrived on the market a year later in March 2016. This device used an approach completely different from the others, instead of having leds on the HMD and controllers, they have receivers: 32 on the headset and 19 on each controller. Those receivers collect data from laser beams emitted from 2 base-stations called "lighthouses".

Base-stations emit two laser beams each, one that swipes horizontally and one vertically, with a specific time interval from one to another. Every single receiver counts the time passed from the swipe start to the moment the laser beam reaches it. Knowing that time-frame for both the vertical and horizontal beams, with some trigonometric calculation, it is possible to triangulate its position in space. More details on the exact functioning of this system will be discussed in the next chapter.

Figure 1.9: Prototype of the Valve HMD and tracking system.

Another great aspect of this system is its openness. On August 2016 Valve announced that they will provide royalty-free licensing to third parties interested in using Valve's "Lighthouse" tracking technology to create new tracked objects, like VR controllers and other peripherals [29].
They provided all technology details and several tools to help the realization of new peripherals including:

- Software toolkit to assist with optimal sensor placement
- Calibration tools for prototyping and manufacturing
- Schematics and layouts for all electronic components
- Mechanical designs for the reference tracked object and accessories
- Datasheets for the sensor ASICs

They also supply directly all the necessary hardware to create new peripherals that are perfectly compatible with the "Lighthouse" system [30].
In addition they offered a completely open API, called OpenVR [31], that allows SteamVR (the closed source program that makes the "Lighthouse" system works) integration in every possible project in C++, C# and in Unity game engine.
Thanks to this openness the technology has been rapidly adopted in all major

Figure 1.10: The HTC Vive with its controller and base-stations.

games engines and many other projects not strictly connected to the video games world. Many different companies have also announced new products compatible with the "Lighthouse" system including a new VR system by LG [32].

2 Tracking System: working details and testing

In the previous chapter different methods for industrial robots programming and their relative problems were discussed. VR and AR approaches of off-line programming, have a major problem in the precise tracking of user's movements. Thanks to modern Virtual Reality systems, there is now a way to solve this issue with standard and relatively economic products, without the need to design and make one from scratch. From the three different technologies analyzed previously, the only one that provided at the same time a good precision and an API for its use in external commercial projects is the Valve's "Lighthouse" system, used currently only in the HTC Vive.
In this chapter more details on the functioning of Valve's "Lighthouse" tracking system will be described. Multiple tests confirming limits and precision of this technology will follow.
Most details in the following sections have been explained directly by Alan Yates, the system creator [33], and in an extraordinary in-depth review done by the blogger Doc Ok [34] [35].

2.1 The system in-depth

The system created by Valve for the HMD and controllers movements tracking is based on a mix of "time-structured" laser beams, precise timing and IMUs [1].
As opposed to the more "standard" solution, used on Oculus Rift and PlayStationVR, where tracked objects emit light and some external sensors recognize that light, in Valve's "Lighthouse" system the external sensors emit light and tracked objects receive it.
Emitted light can be called "time-structured" because the laser beams, emitted from the so called "base-stations", are synchronized and follow a very precise pattern.

[1] Inertial Measurement Unit

2.1.1 Base-Stations

Base-stations are the reference points for any tracked device. They contain some stationary leds and two revolving laser emitters that spin at 3,600 rpm, 60 rotations per second. One laser projects an horizontal line of light (in the base-station's coordinate system) that sweeps through the tracking volume in front of the base-station from bottom to top (when looking at the base-station's front); the other laser projects a vertical line of light that sweeps through the tracking volume from left to right (also when looking at the base-station's front). The stationary leds flash after every half a revolution (that is 8.333 ms long) to synchronize the two base-stations and tracked objects (Figure 2.1). As there can be only one laser sweeping the tracking volume at any time (for now), the two lasers inside one base-station, and the four lasers of two linked base-stations (A and B), are interleaved. First the base-station (A) performs the vertical sweep followed by the horizontal one, then the base-station (B) performs its two own sweeps. This process takes 33.333 ms to conclude before starting again from the beginning. Every tracked object is hit by a laser beam every 8.333 ms.
The way a base-station works allows a relatively low-cost construction and no need of synchronization with anything but the second base-station. This means they need no cables except for the power cable and that they can be placed anywhere with the only limit that the two base-stations need to see each other (problem solvable with a synchronization cable between them).

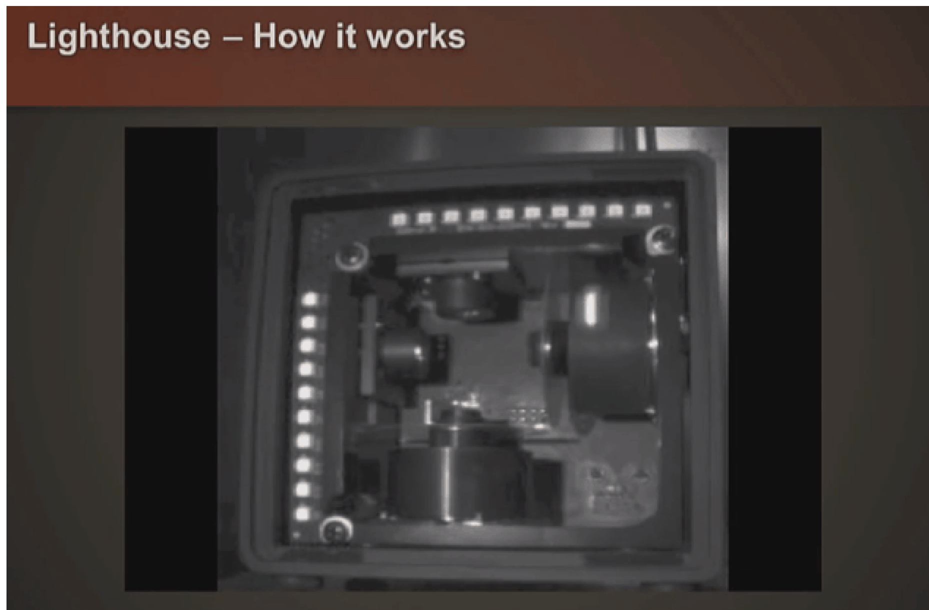


Figure 2.1: Internal components of a base-station [2].

[2]Video at `https://www.youtube.com/watch?v=5yuUZV7cgd8`

2.1.2 Tracked Objects

Having seen how base-stations operate, it can be concluded that all the work for the tracking is done inside every tracked object. Every object that has to be tracked needs to have an IMU and multiple IR receivers. Every receiver is composed by a photo-diode and some integrated circuits. There is also an ASIC[3] that performs all the calculations for the pose estimation. In the HTC Vive this role is played by a Lattice Semiconductor ICE40HX8K-CB132 High-Performance FPGA [36].
Every IR receiver perceives the flash of stationary LEDs inside the base-station as a starting point and simply counts the time passed before being hit by the laser swipe to determinate its own position ([37] explains visually this concept).

Given the time offset between a sync pulse and the laser sweep pulse in microseconds, the angle at which the sensor is located in respect to the base-station is calculated as

$$angle = \frac{(\text{<time offset>} - \text{<central time offset=4000>}) * \pi}{\text{<cycle period=8333>}}$$

Calculating in this way the vertical angle ($angle_{vertical}$) and the horizontal angle ($angle_{horizontal}$) from the two sweeps of the base-station, normals of the two planes identified by those angles can be found as:

$$\begin{cases} V_{horizontal} = (0, cos(angle_{horizontal}), sin(angle_{horizontal})) \\ V_{vertical} = (cos(angle_{vertical}), 0, -sin(angle_{vertical})) \end{cases}$$

Finally the line that intersects those two planes, linking the base-station to the IR receiver, can be described by the vector

$$U = V_{horizontal} \times V_{vertical}$$

Vector U represents the computed 3D line in the local reference frame of the base-station. It can be converted to a global reference frame simply multiplying it by the base-station rotation matrix M:

$$U_{global} = M * U_{local}$$

The IR receiver is now localized in a global reference frame, but more information is needed to determine its position in the third dimension (we only know the line from the base-station to the IR receiver, but not its exact position along this line). Doing the same calculations with timings computed from laser sweeps of the second base-station, a second line in the same global reference frame is found. The intersection point of these two lines is the exact point in space of the IR receiver [38]. Knowing the position of different receivers in the global reference frame and their position on the tracked object itself, the pose (position

[3] Application-Specific Integrated Circuit

+ rotation) of the whole object in the 3D world can be found.

With this method the position of every IR receiver, that sees both base-stations, can be computed every 33.333 ms or 30 times per second. This frequency is much lower than the 90 Hz required for a fluid experience in VR. In fact this method is only used to acquire the object pose for the first time, where at least 5 receivers that see both base-stations are needed.
After the first acquisition of the pose, tracking is done primarily using the IMU inside the object (an Invensense MPU-6500 6-axis Gyroscope and Accelerometer Combo for the HTC Vive [36]). IMUs allow for a tracking called *dead reckoning* [39]; accelerometers measure linear motion along the x, y, and z axes (axial acceleration), while the gyroscopes measure rotation (angular velocity) around these axes. This type of tracking is good enough only for a very small amount of time because the noise and bias drift introduced by gyroscopes result in fast-growing errors in the spatial position. In that small time window *dead reckoning* allows for a very high frequency tracking.
In the “Lighthouse” system, drift errors that perturb the position calculated with the dead reckoning method, are corrected by the base-station laser tracking (Figure 2.2). For these corrections the system needs only one receiver that sees both base-stations or at least three receivers that see only one base-station. Using this method, with the HTC Vive, the system provides new poses at a rate of 225 Hz for the headset and 250 Hz for the controllers; talking directly to the “sensor fusion code” the frequency is even higher, with 1,006 Hz for the headset and 366 Hz for the controllers. The “sensor fusion code” is the heart of the system, it merges data from the IMUs with data from the laser system, using a filter similar to the Kalman filter [40]. This filter merges data in a manner such that the combination of the two noise sources is less noisy than either source alone.

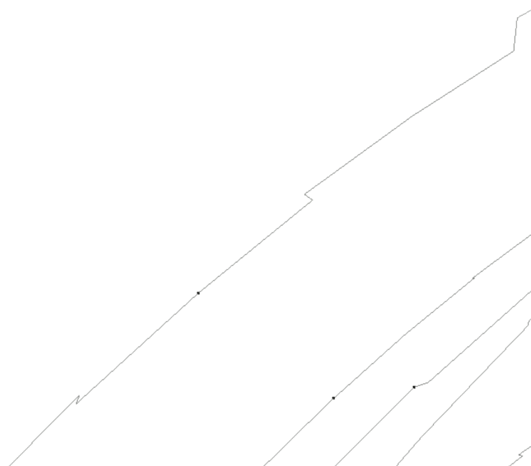


Figure 2.2: The sudden changes in the reported position are the laser tracking corrections over the *dead reckoning* method[4].

[4]Video at https://www.youtube.com/watch?v=A75uKqA67FI

2.1.3 System calibration

A tracking system calibration is usually one of the most cumbersome steps for users, it is normally quite long and requires someone who knows how the system works.
Since the “Lighthouse” system needs to be integrated into mass market products, where deployment of the system is done by the final user, all steps, including calibration, must be easy.
To accomplish this task, Valve made an easy to follow calibration program that can be used for all products that use SteamVR. This program consists in 4 steps, some of them are needed to make the virtual world safe when the user has the HMD on the head.

1. First of all, both controllers and the HMD need to be inside the tracked area. They need to see both base-stations to perform the first acquisition with the method described in the previous section.

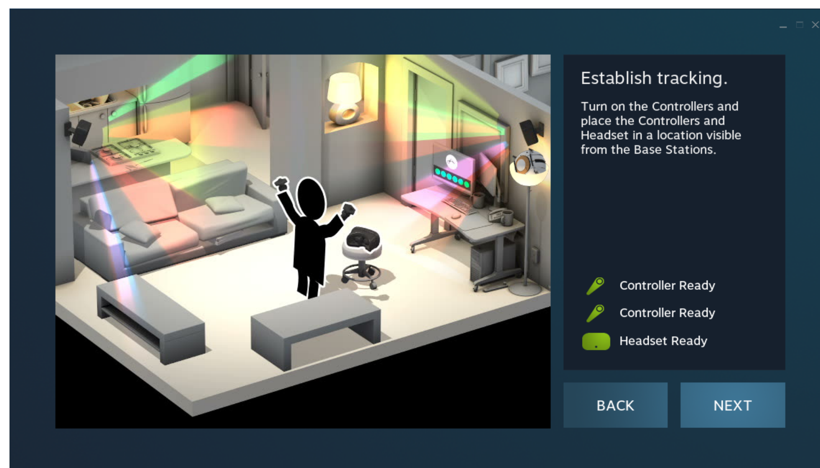


Figure 2.3: First step of the calibration process.

2. In the second step the user needs to point a controller in the direction of the computer monitor. This step is only used to orient the virtual world in this direction, in this way the user can maintain some sort of orientation even inside the virtual world.

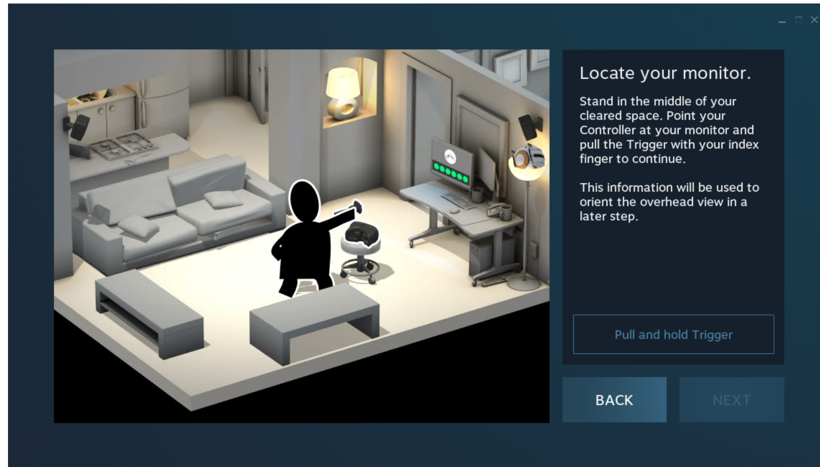


Figure 2.4: Second step of the calibration process.

3. In the third step the user is asked to place both controllers on the floor in the middle of the tracked space. In this way the floor plane can be found reading the controllers position in both base-station's local frames; with those positions a global frame can be computed establishing the Y axis that represent the UP direction. This reading uses both controllers to mitigate eventual oscillations mediating on their poses.

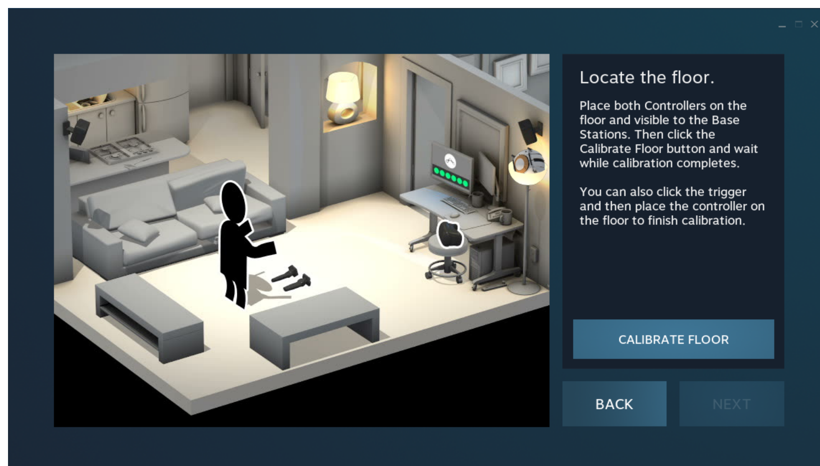


Figure 2.5: Third step of the calibration process.

4. At this point the tracking is already working correctly and controllers, HMD and base-stations positions are showed. This step is only needed to trace a virtual bound of the user's obstacle-free space; when controllers/HMD come near to this traced bound a virtual wall will be shown in the virtual world to warn the user that the obstacle-free space is ending. At this point the program tries to find the largest rectangle that fits inside this virtual bound. This rectangle is called "play-area" (green rectangle in Figure 2.6)

and it is used to orient the global frame. The X axis is aligned with the longest side of this rectangle, while the Z axis with the shortest (the Y axis was already aligned straight up in the previous step). The Z axis is pointed on the opposite side of the user monitor (found in the second step) by default, but its orientation can be changed by the user, and the X one will be oriented consequently.

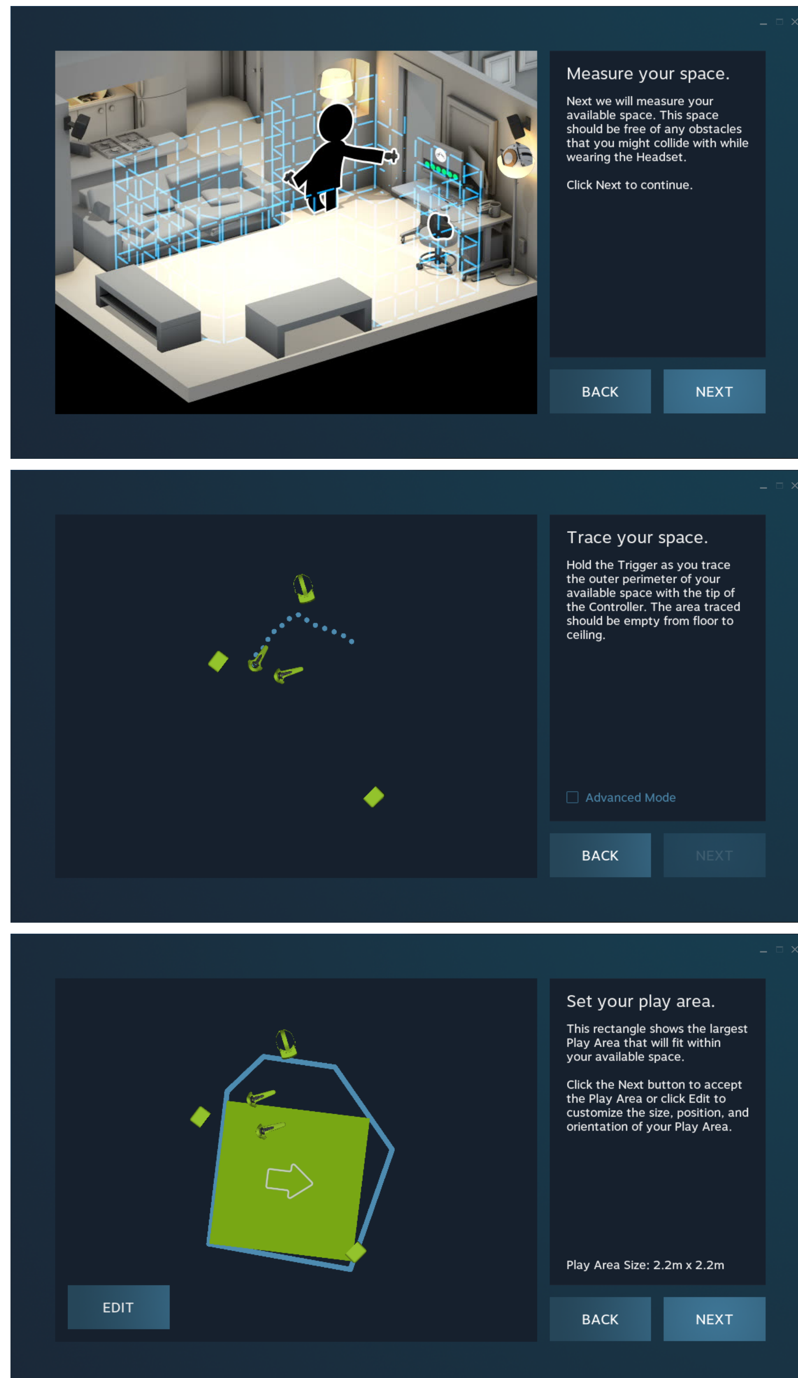


Figure 2.6: Fourth and last step of the calibration process.

2.2 Testing system performance

Knowing the system working details, it is now essential to test it and see what it is capable of. The tests performed were aimed to find the system limits, in particular to find its residual noise (or tracking jitter), precision and accuracy. The results of these tests were also compared to the results obtained in [34].
Other tests have also been performed to see different aspects of the system, like the volume of the tracking area and eventual occlusion problems.

2.2.1 Residual noise test

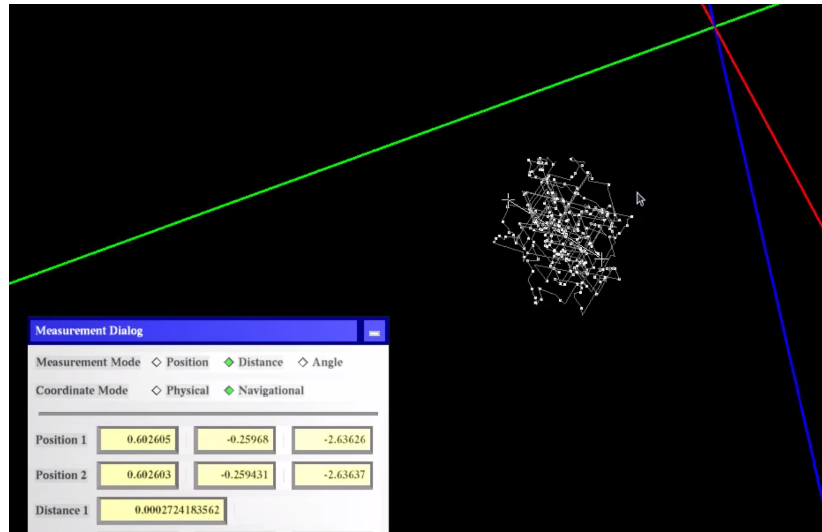
The residual noise of a tracking system is the amount of jitter in the measured position when the real object is actually fixed in space. To do this test a controller was placed on a chair in the middle of the tracked volume and its reported position checked for one second. Reported residual noise is the maximum difference between positions in this time span, calculated separately for the three axes.
The first test was done with both lighthouses in sight of the controller, then two other tests were performed occluding one of the two lighthouses to see eventual differences. All reported values are computed as mean of 5 different measures, each one second long. Results are reported in Table 2.1.

all values are in mm	X error (on 5 measures)	Y error (on 5 measures)	Z error (on 5 measures)
Test 1 (two lighthouses)	0.4325	0.4325	0.4450
Test 2 (one lighthouse)	0.9180	0.7520	0.6420
Test 3 (one lighthouse)	0.6040	0.4660	0.5240

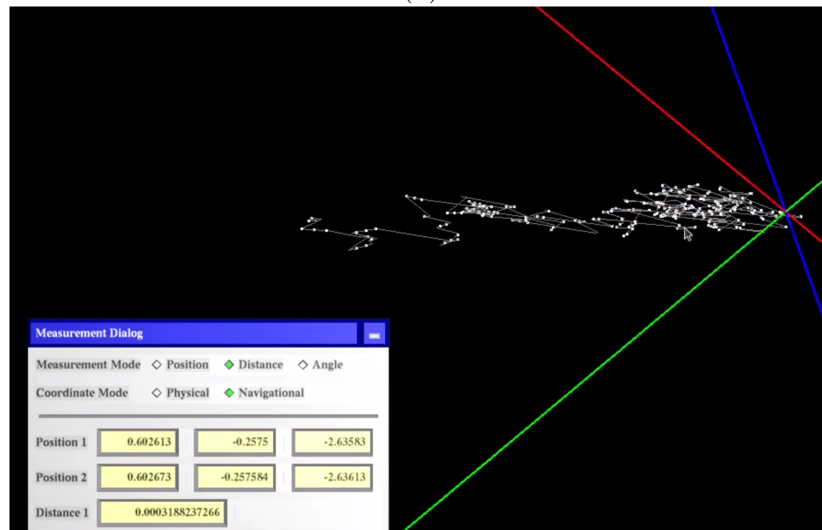
Table 2.1: Residual noise of the Valve “Lighthouse” system with HTC Vive.

With both lighthouses in sight of the controller, the residual noise is about 0.4mm on all three axes appearing perfectly isotropic. Occluding one lighthouse an increase in the error on all three axes can be seen, with a maximum of 0.9mm. As opposed to what was expected, the error does not seem highly anisotropic.
The axes in Table 2.1 represent the global reference frame, and not the local frame of the base-stations. Transforming reported positions from the global frame to the local one, the values follow the expected results; occluding one lighthouse leads to an higher error in the Z axis, the one pointing to the other lighthouse. In fact, as seen in the previous chapter, with only one lighthouse the system can only find the line from the object to the base-station, and only with the second lighthouse it can interpolate the exact point along this line. Without the second lighthouse, the third dimension is calculated using only IMUs information and the position of different IR receivers on the object (closer to the base-stations the

time between hits of different receivers is greater than when the object is further away). The lower precision of those methods leads to an increased residual noise on that axis.



(a)



(b)

Figure 2.7: Residual noise with both lighthouses (a) and with only one (b)[5].

Residual noise values obtained in [34] are very similar, with a figure of 0.3mm for the two lighthouses situation and an increase to 2.1mm in the axis pointing to the occluded lighthouse. Results for the headset are very similar in values and distribution.

[5]Video at https://www.youtube.com/watch?v=Uzv2H3PDPDg

2.2.2 Precision test

The precision of a tracking system is the difference between multiple subsequent measurements of the same point in space. To do this test, the structure that maintains still controllers and base-stations in the HTC Vive box (Figure 2.8) was used. The controller was removed and repositioned for every measurement on the structure (that was fixed on a table with Scotch tape) and the reported position checked.

Figure 2.8: The retail box of the HTC Vive.

The reported controller position was taken for 10 times after removing and repositioning it in the structure. Reported values were really close, with an RMS[6] distance from the mean position for every axis lower than 2 mm; for the Z axis this value was even lower than the other two with 0.85 mm.

In [34] measurements for precision were executed saving the position of every inch of a 36" long ruler for two times, and calculating the RMS distance of those two sets of measures. The values obtained are comparable with those measured here, with a slightly lower maximum distance of 1.5 mm.

[6] Root Mean Square

all values are in mm	X position	Y position	Z position
1	-1592.12	760.62	474.88
2	-1594.24	764.35	476.74
3	-1595.05	765.26	477.17
4	-1594.99	765.89	477.31
5	-1595.39	766.3	477.56
6	-1595.98	767.14	477.38
7	-1594.04	767.31	477.73
8	-1593.02	767.37	477.78
9	-1595.46	766.96	477.84
10	-1595.27	766.81	477.85
Mean	-1594.556	765.801	477.224
Mean Distance	**0.9608**	**1.4346**	**0.5764**
RMS	**1.1463**	**1.9593**	**0.8492**

Table 2.2: Subsequent measurements of the position of the controller.

2.2.3 Accuracy test

The accuracy of a tracking system is how close the measurement of a point is to the actual position of that point in space. To do this test, a sheet of paper with black and white squares of known size was used (Figure 2.9). The sheet was taped to a table and the position of every color change was measured. The table was aligned with the global frame axes, and measures were done for all three axes. The measured distance of every color change was then compared to its real length.

Figure 2.9: The sheet of paper used for accuracy measures.

Color change	Measure	X	Y	Z	Error X	Error Y	Error Z
white/white	1	176.14	-839.63	-0.89			
white/white	2	126.39	-789.98	-50.7	**0.75**	**0.65**	**0.81**
white/black	3	75.98	-741.72	-100.75	**1.41**	**-0.74**	**1.05**
black/white	4	24.61	-690.33	-150.62	**1.37**	**1.39**	**-0.13**
white/black	5	-24.26	-640.5	-201.39	**-0.13**	**0.83**	**1.77**
black/white	6	-75.27	-588.91	-250.34	**1.01**	**1.59**	**-1.05**
white/black	7	-124.46	-540.97	-301.6	**0.19**	**-1.06**	**2.26**
black/white	8	-173.81	-489.93	-351.04	**-0.65**	**1.04**	**-0.56**
white/black	9	-222.95	-441.05	-401.49	**0.14**	**-0.12**	**1.45**
black/white	10	-273.38	-389.65	-450.51	**0.43**	**1.4**	**-0.98**
RMS					**0.782**	**1.014**	**1.203**
Mean black distance	**50**	**50.54**	**51.355**	**50.633**	**0.54**	**1.355**	**0.632**
Mean white distance	**49**	**49.472**	**48.912**	**49.418**	**0.472**	**-0.088**	**0.418**
Total distance	**448**	**449.52**	**449.98**	**449.62**	**1.52**	**1.98**	**1.62**

Table 2.3: Accuracy of Valve's "Lighthouse" system on all three axes, all values in mm.

In Table 2.3 results are summarized. Only the tested axis values are reported, the other two axes of every measure remain constant with an error comparable to the values in the precision test.
In the Error columns, differences between the measured length of a square and its real length are reported (this length is 49 mm for white squares and 50 mm for black squares). The measured length is computed as the difference between two consecutive measures taken at every color change. The RMS of these errors are reported, with values close to 1 mm for all three axes. Those values are slightly better than the results obtained in [34], where the accuracy calculated on a ruler had an RMS of 1.9 mm.
In Table 2.3 the mean distance for white and black squares and the total distance, measured as sum of all the segments, are also reported. These measures have a maximum error of 1.98 mm out of 50 cm for the sum of all segments in the Y axis.

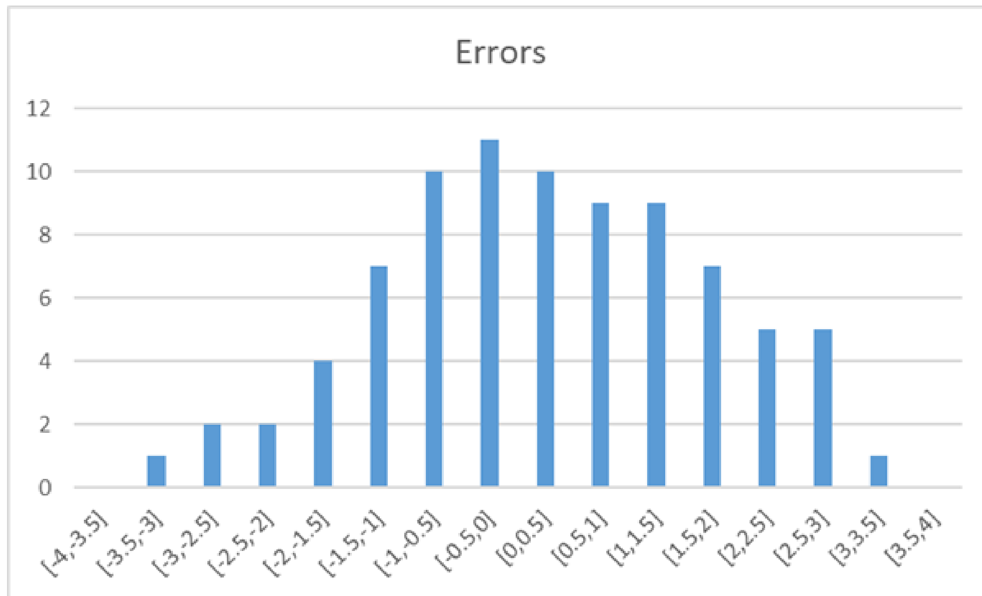


Figure 2.10: Distribution of errors in the measures with the lighthouse system.

Analyzing all the errors gathered during all the tests performed, a normal distribution with 0 mean should be found. In Figure 2.10 all errors are plotted and, even thought with a slight tendency to positive errors instead of negative ones, the errors clearly follow a normal distribution.

2.2.4 Accuracy confirmation, tracked volume and occlusion

Confirmation of accuracy against Kreon Baces

To confirm the accuracy values obtained in the previous chapter, the "Lighthouse" system was tested against a Kreon Baces [41] (Figure 2.11), a 6 axes measuring arm. This arm has a very high accuracy with values in the order of tens of millimeters and can be assumed as ideal compared to the "Lighthouse" system under test.

Baces 6 axes	Probing performance Point repeatability	Probing performance Volumetric accuracy
M - 2.60 m	0.028 mm	0.044 mm
G - 3.20 m	0.045 mm	0.064 mm
XG - 4.20 m	0.072 mm	0.097 mm
XL - 4.60 m	0.094 mm	0.120 mm

Table 2.4: The precision and accuracy of the Kreon Baces arms from [41].

The smaller model, with the highest precision and accuracy, was used. To compare measures between the two systems, in a first try, an HTC Vive controller

Figure 2.11: The Kreon Baces arm used for the tests.

has been attached to the probe of the arm and several measures were taken at different points in space. Ideally the euclidean distance from the arm probe tip and the controller should be constant and any measured difference of this distance should be the error of the "Lighthouse" system, assuming the arm measures as perfect.
Sadly this method cannot work, the "Lighthouse" system and the Baces arm have different reference systems and the euclidean distance between them does not have the desired meaning. To compare measures between the two systems, one must find how the two reference systems relate to each other.
Attaching the controller to the arm probe with an offset from its tip complicates the relationship between the two reference systems. In fact two reference systems should be related with a rigid transformation, but the measures obtained with this controller position cannot be used to find that transformation; for every arm probe tip position, the controller can be in a whole sphere of positions around it. This negates the possibility to find a rigid transformation between the measured points.
To simplify calculations and remove every possible origin of external errors, like unwanted drift in the controller position and human error during measurements, the acquisition method was changed. Controller and arm were used separately to

measure the same physical points, every point was measured 10 times with each device to mitigate manual errors. Those measures were then used to find a rigid transformation between the two reference systems.

Finding the rigid transformation between two reference systems requires to solve, for R and t, the equation:

$$B = R * A + t$$

Where R and t are the transformations applied to measured points in the reference system A to align them with the respective points in the reference frame B, as best as possible.
Ideally, if both systems were perfect, measures taken with the controller in the reference frame A and those taken with the arm in the reference frame B, should have a constant rigid transformation. Once found the ideals R and t, they can be used to find the error of the "Lighthouse" system (assuming the measures of the arm have no error, assumption fairly easy to make observing the values in Table 2.4). This error is the difference from a point measured with the controller, transformed from its reference frame to the other, and the same physical point taken with the arm directly in the second reference frame.
To find the values of R and t, first the centroids of the measured points are computed as:

$$\text{centroid}_A = \frac{1}{N} \sum_{i=1}^{N} P_A^i$$

$$\text{centroid}_B = \frac{1}{N} \sum_{i=1}^{N} P_B^i$$

From them matrix R is found using the SVD[7] function [42]:

$$H = \sum_{i=1}^{N} (P_A^i - \text{centroid}_A)(P_B^i - \text{centroid}_B)$$

$$[U, S, V] = \text{SVD}(H)$$

$$R = VU^T$$

There is a special case when finding the rotation matrix. Sometimes the SVD will return a "reflection" matrix, which is numerically correct but is actually nonsense in real life. This is addressed by checking the determinant of R (from SVD above) and seeing if it is negative (-1). If it is, then the 3rd column of V is multiplied by -1.[43]
At this point t can be found simply as

$$t = -R \times \text{centroid}_A + \text{centroid}_B$$

This method has been implemented in MATLAB.

[7]Singular Value Decomposition

Values for R and t found for 5 different physical points are:

$$R = \begin{pmatrix} 0.9924 & 0.0726 & -0.0995 \\ -0.0704 & 0.9972 & 0.0254 \\ 0.1011 & -0.0182 & 0.9947 \end{pmatrix}$$

$$t = \begin{pmatrix} 37.7308 & 85.2182 & -127.2298 \end{pmatrix}$$

variances of the measures are reported in Table 2.5 and errors of the "Lighthouse" system are reported in Table 2.6.

All values are in mm	Device	Variance		
		X	Y	Z
Point 1	Baces	0.231	0.321	0.403
	Vive	1.114	0.642	0.434
Point 2	Baces	0.106	0.081	0.474
	Vive	0.038	0.020	0.012
Point 3	Baces	0.119	0.055	0.388
	Vive	0.901	0.523	0.071
Point 4	Baces	0.325	1.033	1.381
	Vive	0.045	0.080	0.047
Point 5	Baces	0.270	0.492	0.660
	Vive	0.544	0.414	0.176

Table 2.5: Variances in 10 measures of the same point with Baces arm and HTC vive controller.

All values are in mm	X Error	Y Error	Z Error
Point 1	1.588	1.139	0.691
Point 2	2.459	1.607	0.625
Point 3	2.644	0.624	1.258
Point 4	0.677	1.109	0.175
Point 5	3.838	1.014	1.361

Table 2.6: Errors of the lighthouse system compared to the Baces arm.

As it can be seen, error values are quite consistent with those found for accuracy in the previous section, with only a couple of error greater than expected with nearly 4mm. These higher values can be motivated by human error; even though measures were repeated 10 times and only their mean was used to reduce this type of error, having both systems used by hand can introduce some small errors in the measures. Variances of these measures are small, with only a couple reaching 1 mm (Table 2.5), but even these small errors can lead to a not perfect

computation of the rigid transformation between the two reference systems and to a greater error in the difference computation used to estimate the “Lighthouse” system error.

Volume of tracked space

An important aspect of a tracking system for many applications, is the tracked area volume and its working with occlusion.
For the first aspect, official values specify an area of 3.5x3.5m as an upper limit, but in our tests tracking worked without problems even in a greater area. The main factor for this size limit, is the syncing leds power. Their blink needs to be seen by all tracked objects; when the area is too large the leds do not have enough power to illuminate all the volume. A limit, in the current version of the system, seems to be near 7x7m, where the system works just fine but sometimes the tracking is lost. This limiting factor will be removed with the next version of the system thanks to a different approach (more details on this will be explained in the next section).

Figure 2.12: Tracked volume is greater than expected, all positions are still being tracked.

The tracked area (orange box in Figure 2.12) is the area between the base-stations, where objects see both of them and are tracked with the highest precision. In reality objects will be tracked in places that are out of this area, even though with a lower precision caused by the use of only one base-station (as discussed

before). In Figure 2.12 the controllers are still being tracked by the system in all the positions, even if they are out of the tracked area.

Occlusion of the base-stations

The “Lighthouse” system, as already discussed, works with laser beams and, like every optical system, can have occlusion problems. In particular the system will continue to work with only one lighthouse, even though with reduced accuracy, but when both are occluded the tracking will be lost. For a couple of seconds the system will continue to compute object position and orientation using only the internal IMU, but in those moments there is no drift correction provided by the base-stations so the values cannot be trusted if a high precision is required.
This problem can limit the system use in applications where the line of sight is not guaranteed like, for example, where there are boxes or highly convex objects and the controllers need to be tracked inside those objects. A solution to this can be found thanks to the “dumb” functioning of the base-stations: the only caveat on their positioning is the need to have free line of sight between them (or, if not possible, to use a sync cable to connect them).

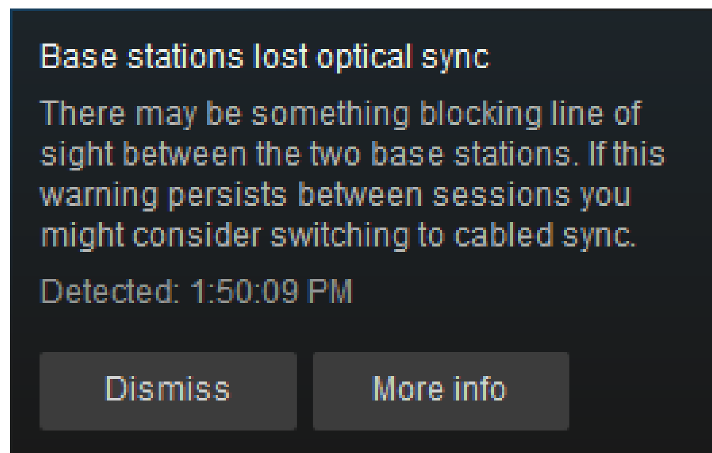


Figure 2.13: Error message showed by the system when base-stations do not have free line of sight between them.

Placing one of the base-stations on the room roof, for example, will solve every problem for vertical boxes or highly convex objects, but not for horizontal ones. The use of more than two base-stations could solve all problems of this type with a careful positioning; sadly only two base-stations can be used in the actual version of the system but this limit will be removed with its next version.

Different types of tracked objects

In all previous tests the HTC Vive's controllers were used, but this technology can be used on everything. To facilitate the tracking of every type of objects, Valve (in partnership with HTC) presented the Vive Tracker [44]. This little device can be attached to anything and it will track its movements in space. It has a standard screw on the bottom with six pins that can be assigned to mirror the standard buttons of a Vive Controller.

The Vive Tracker is already used by several companies in gaming and other industries to track various objects like gloves (Figure 2.14 (a)), pistols (Figure 2.14 (b)), baseball bats (Figure 2.14 (c)), mock fire hoses (Figure 2.14 (d)) enabling the creation of games, safe training environments and various applications in many different fields.

(a) (b) (c) (d)

Figure 2.14: The Vive Tracker used for several accessories.

A new type of controller is also being developed by Valve, the Knuckles Controllers (Figure 2.15). These controllers allow users to use their hands freely thanks to a strap that grab on the palm of the hand; they are more “worn” than “held” like the old Vive Controllers. They offer a full tracking not only of the hand position, but, thanks to several capacitive sensors, also of the aperture of every single finger [8].

Figure 2.15: The new Valve’s Knuckles Controllers compared to the Vive Controllers.

[8]Video at `https://www.youtube.com/watch?v=kovhtH3r9o0`

2.3 Future system developments

On June 2017, Valve announced a Technology Update for the SteamVR Tracking [45]. The current IR receivers used on the HTC Vive and the Vive Tracker, use the TS3633 ASIC from Triad Semiconductor, which has 11 components per sensor and produces a single "envelope" pulse per laser or sync blinker hit. To use the new technology developers have to use the new TS4231 ASIC which uses 5 components per sensor, so it is cheaper to place at each sensor [46]. More importantly, it provides a burst of data per laser or sync hit. Using that data allows information to be transmitted on the laser itself, which can be used to learn about the source of that laser.
This new capability to encode information in the laser is significant for two reasons:

- It allows support for more than two base-stations, and thus larger tracking volumes without the need for a line of sight between different base-stations.
- It allows a base-station to function without including a synchronization blinker, which is the source of most of the interference between base-stations and the principal limiting factor of the tracking volume (and is also a significant driver of base-station cost). This technology is called sync-on-beam.

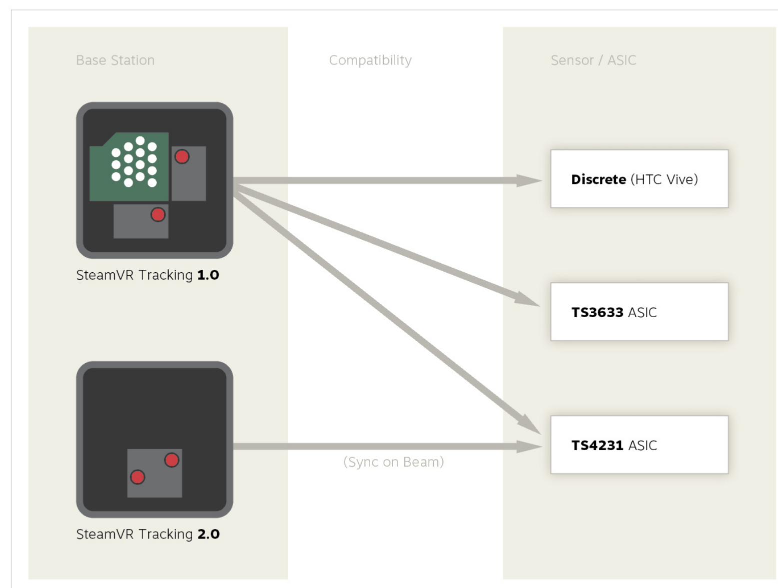


Figure 2.16: Base-stations update with SteamVR Tracking 2.0.

This technology change will solve practically all problems the "Lighthouse" system has currently. The tracked area can be larger than before thanks to the lack of a synchronization blink, occlusion problems can be mitigated adding more base-stations where needed and line of sight between different base-stations is no longer needed.

3 Trajectories programming, developed software

With the results obtained in the previous chapter we now have a good idea of the system precision and its limits; with this in mind we can decide what are the applications where this system can be used. For example trajectories for arcs welding, one of the most requested and difficult applications in robotics, requires a higher precision and accuracy than the two millimeter figure this system has. For many others commercial applications the “Lighthouse” system precision is more than enough.
Developed software is written in C# using Visual Studio 2015 Community Edition IDE [47].
The OpenVR API is developed in C++ and the C# wrapper present in [31] is constructed automatically without any human test. To familiarize and test the C# wrapper of the OpenVR API I started creating an empty world with a single robotic arm using Euclid Labs APIs. Only after thorough testing of the API in this test program, I’ve started developing a full off-line programming package for trajectories recording that uses an augmented reality approach.

3.1 First test program

Integrating the “Lighthouse” tracking technology into the virtual world, experimenting with its possibilities, adding tracked 3D models in the environment and using inverse kinematic to move the robotic arm with the controller allowed to test all main functions of the API.
To add 3D models of controllers, HMD and base-stations, I have created a configuration file (.ini) where all details, including the paths to the actual 3D model of every object (provided by Valve inside SteamVR folders) are specified.

```
[ViveControllerRight]
Ambient=Gray
Bitmap=onepointfive_texture.png
Body=viveController.stl
```

```
CollisionCheckEnabled=1
Diffuse=Gray
Height=150.000
Length=50.000
ModelName=
ModelType=-1
Shape=0
Specular=Gray
Visible=0
Width=50.000
Wireframe=0
ABCType=ZYX
```

These details are then used to initialize a *3DMeshObject* that will be visualized inside the virtual world for every device. Updating the virtual world position of these objects with the positional data provided by the OpenVR API results in real-time tracked 3D models inside the virtual world.
Using the tracking data to also move the robotic arm is quite easy thanks to Euclid Labs APIs. The robot is defined in a similar way to the 3D models discussed previously, by an ini file that describes all its details, which are used to initialize a *RobotManipulatorStandard* object. This object allows to move the robot in various ways, including axis by axis, linearly and by setting directly the tool position, automatically resolving all the inverse kinematic required. With this method just by setting the tool position to the current controller position, provided by the OpenVR API, the robotic arm will move alongside the controller.
In this first program I have also inserted the tracked area, saved in the current calibration of SteamVR, as a green parallelepiped to visualize where the user can move (Figure 3.1).

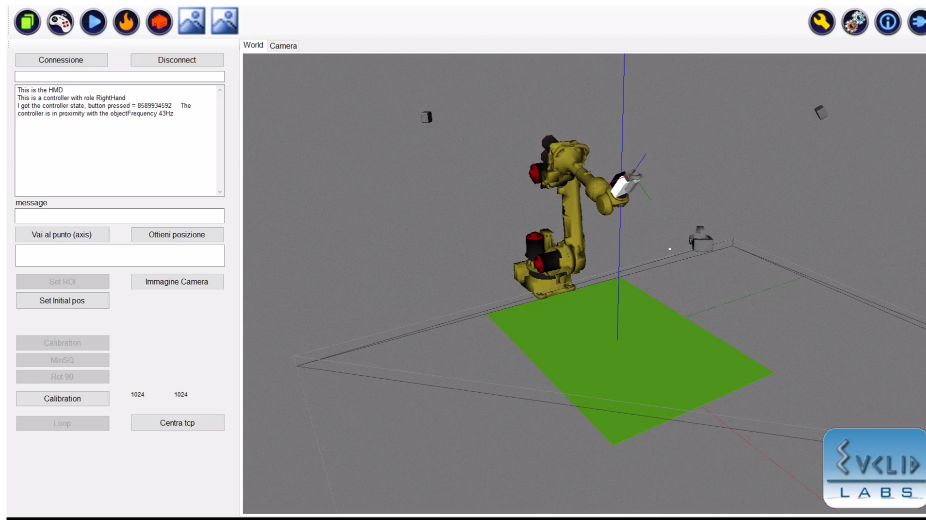


Figure 3.1: First test of inverse kinematic using a HTC Vive Controller and a Fanuc R-1000iA/80F Max robotic arm.[1]

Several problems with the C# wrapper has been found thanks to this test:

- Matrix representation is unusable and with swapped axes. To obtain poses is sufficient to call the *GetDeviceToAbsoluteTrackingPose* function:

```
//Define an array to contain the poses of all devices
TrackedDevicePose_t[] pose = new TrackedDevicePose_t[OpenVR
    .k_unMaxTrackedDeviceCount];
//Get the current pose of all devices (the second
    parameters is the number of seconds from now to predict
    poses for)
OpenVR.System.GetDeviceToAbsoluteTrackingPose(
    ETrackingUniverseOrigin.TrackingUniverseStanding, 0,
    pose);
```

where the *TrackedDevicePose_t* is a struct defined as:

```
public struct TrackedDevicePose_t
{
public HmdMatrix34_t mDeviceToAbsoluteTracking;
public HmdVector3_t vVelocity;
public HmdVector3_t vAngularVelocity;
public ETrackingResult eTrackingResult;
[MarshalAs(UnmanagedType.I1)]
public bool bPoseIsValid;
[MarshalAs(UnmanagedType.I1)]
public bool bDeviceIsConnected;
}
```

The variables *bPoseIsValid* and *eTrackingResult* allow to check if the device is inside the tracked area and if it is tracking correctly or using only the internal IMU because it cannot see a base-station, respectively. The actual pose is contained in the *mDeviceToAbsoluteTracking* variable, it is already in the global frame of the system, but the used variable type *HmdMatrix34_t* allows only an item by item access and the used global frame has the axes swapped. The Z axis points backward and the Y axis points upward compared to the global frame used in the virtual world. To correct this incongruence and to use a standard variable type, I've wrote the following method:

```
private HomogeneousMatrix getHomogeneousMatrix(
    HmdMatrix34_t matrix)
{
HomogeneousMatrix homogMatrix = new HomogeneousMatrix();
homogMatrix.Rotation = new RotationMatrix(matrix.m0, matrix
    .m1, matrix.m2, matrix.m4, matrix.m5, matrix.m6, matrix.
    m8, matrix.m9, matrix.m10);
```

[1] Video at `https://www.youtube.com/watch?v=21GKyRVQNgI`.

```
//Converto from mm to m to respect convention
homogMatrix.Position = new Vector3D(matrix.m3 * 1000,
   matrix.m7 * 1000, matrix.m11 * 1000);
//Matrix to transform from vive coordinate (Y up, -Z in the
     direction of 'our' Y) to the world3D coordinate
HomogeneousMatrix correctionMatrix = new HomogeneousMatrix(
   new RotationMatrix(1, 0, 0, 0, 0, -1, 0, 1, 0));

return correctionMatrix * homogMatrix;
}
```

- The function to check for pressed buttons does not exist.
  In the C++ API buttons pressed on the controllers can be checked by calling the *GetControllerState* function:

```
//It is the index of the device you want the state
uint device;
VRControllerState_t controllerState = new
   VRControllerState_t();

OpenVR.System.GetControllerState(device, ref
   controllerState, (uint)System.Runtime.InteropServices.
   Marshal.SizeOf(controllerState));
```

  where the *VRControllerState_t* is a structure defined as:

```
public struct VRControllerState_t
{
    public uint unPacketNum;
    public ulong ulButtonPressed;
    public ulong ulButtonTouched;
    public VRControllerAxis_t rAxis0; //VRControllerAxis_t
        [5]
    public VRControllerAxis_t rAxis1;
    public VRControllerAxis_t rAxis2;
    public VRControllerAxis_t rAxis3;
    public VRControllerAxis_t rAxis4;
}
```

  rAxis0 contains the (x,y) position where the touch-pad is touched, *rAxis1* contains the trigger button state expressed as a percentage, the others are not used for now: they will be used for the new Knuckles Controllers that will track all 5 fingers. The *ulButtonPressed* and *ulButtonTouched* variables instead are a mask that identifies pressed and touched buttons respectively that follows this *ButtonMask*:

```
enum EVRButtonId
{
    k_EButton_System            = 0,
```

```
        k_EButton_ApplicationMenu   = 1,
        k_EButton_Grip              = 2,
        k_EButton_DPad_Left         = 3,
        k_EButton_DPad_Up           = 4,
        k_EButton_DPad_Right        = 5,
        k_EButton_DPad_Down         = 6,
        k_EButton_A                 = 7,

        k_EButton_ProximitySensor   = 31,

        k_EButton_Axis0             = 32,
        k_EButton_Axis1             = 33,
        k_EButton_Axis2             = 34,
        k_EButton_Axis3             = 35,
        k_EButton_Axis4             = 36,

        // aliases for well known controllers
        k_EButton_SteamVR_Touchpad  = k_EButton_Axis0,
        k_EButton_SteamVR_Trigger   = k_EButton_Axis1,

        k_EButton_Dashboard_Back    = k_EButton_Grip,

        k_EButton_Max               = 64
    };
```

in the original C++ library this mask can be used calling the function *ButtonMaskFromId*

```
    uint64_t ButtonMaskFromId( EVRButtonId id ) { return 1ull
        << id; }
```

by passing the Id of the wanted button and comparing them bit by bit:

```
    if((controllerState.ulButtonPressed & ButtonMaskFromId(
        EVRButton.k_EButton_Grip)) != 0)
        //The grip button of the controller is pressed
```

Sadly this function is not present in the C# wrapper and the value 1*ull* is not valid in this language. To solve the problem I've created a dedicated class:

```
    public class ButtonMask
    {
        public const ulong System = (1ul << (int)EVRButtonId.
            k_EButton_System); // reserved
        public const ulong ApplicationMenu = (1ul << (int)
            EVRButtonId.k_EButton_ApplicationMenu);
        public const ulong Grip = (1ul << (int)EVRButtonId.
            k_EButton_Grip);
        public const ulong Axis0 = (1ul << (int)EVRButtonId.
            k_EButton_Axis0);
```

```
        public const ulong Axis1 = (1ul << (int)EVRButtonId.
            k_EButton_Axis1);
        public const ulong Axis2 = (1ul << (int)EVRButtonId.
            k_EButton_Axis2);
        public const ulong Axis3 = (1ul << (int)EVRButtonId.
            k_EButton_Axis3);
        public const ulong Axis4 = (1ul << (int)EVRButtonId.
            k_EButton_Axis4);
        public const ulong Touchpad = (1ul << (int)EVRButtonId.
            k_EButton_SteamVR_Touchpad);
        public const ulong Trigger = (1ul << (int)EVRButtonId.
            k_EButton_SteamVR_Trigger);
    }
```

Using this class the buttons can be checked in a similar way to the C++ original implementation

```
    if ((controllerState.ulButtonPressed & ButtonMask.Grip) !=
       0)
        //The grip button of the controller is pressed
```

- Controller vibration does not work.
  In the C++ library there is a function to make the controller vibrate

```
    /** Trigger a single haptic pulse on a controller.*/
    virtual void TriggerHapticPulse( vr::TrackedDeviceIndex_t
       unControllerDeviceIndex, uint32_t unAxisId, unsigned
       short usDurationMicroSec ) = 0;
```

This function will trigger an haptic pulse on the controller for the selected number of microseconds. The C# version instead does only a small, inaudible vibration when used with the same parameters.

```
    public void TriggerHapticPulse(uint unControllerDeviceIndex
       ,uint unAxisId,char usDurationMicroSec)
```

After several try and research on the issue I've developed a custom version of the function that works in a similar way to the original C++ version even with the wrong wrapped function, repeating its calling several times:

```
    // Trigger an haptic pulse in the selected device
    public void HapticPulse(uint device, ushort
       durationMicroSec = 500, float strength = 0.5f,
       EVRButtonId buttonId = EVRButtonId.
       k_EButton_SteamVR_Touchpad)
    {
        var system = OpenVR.System;
        if (system != null)
        {
```

```
                var axisId = (uint)buttonId - (uint)EVRButtonId.
                    k_EButton_Axis0;
                for (float i = 0; i < durationMicroSec; i++)
                {
                    OpenVR.System.TriggerHapticPulse(device, axisId
                        , Convert.ToUInt16(3999 * strength));
                }
            }
        }
```

3.2 Full off-line programming package development

With the experience gained creating the test program, I started programming a full software for trajectories recording based on the Eleven 11 program developed by Euclid Labs [48], which used the Kreon Baces arm utilized in previous precision tests.

The program is designed as a full off-line package for industrial robots programming. It uses the "Lighthouse" system for tracking user's movements and an AR approach to facilitate the robot programming for its use even with inexperienced users. It can be used from the initial recording to the execution on the real robot, including the possibility to fully check the program on a virtual robot and, if needed, applying some modifications.

Figure 3.2: The virtual world in the developed program.

3.2.1 Recording trajectories

Trajectories recording needs a common reference frame between the recording device and the robotic arm that will perform the trajectory. To do this I have implemented a frame definition form (Figure 3.3), where the user can define a frame (representing the table in the virtual environment) touching three points: the origin, the X axis and the Y axis. At every point the tracking device position is saved. When the "Compute Frame" button is pressed, these three positions are used to compute a new *HomogeneousMatrix* that represents the new frame. Applying this change with the second button will save the *HomogeneousMatrix* in the program parameters. Every successive tracking device position will be pre-multiplied by the inverse of this matrix to refer to the defined frame.
Having a constant distance between the robot and this frame, the recording device coordinates, changed in this way, can be used on the real world robot to move it at the correct point.

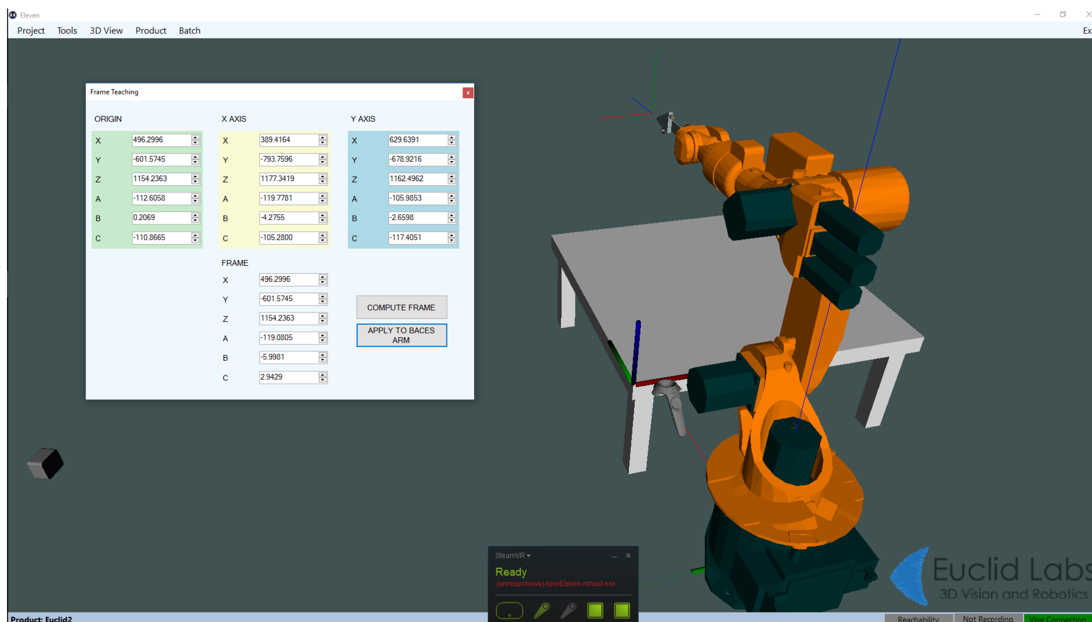


Figure 3.3: The frame teaching form.

The software has been organized with a distinct separation between the recording device and the rest of the program, in a way that makes it is easy to use the HTC Vive controller, the Kreon Baces or something else as an input device, with only little changes to the code. In fact the recording device is represented with an interface that call system events for every meaningful event like a button press or the arrival of an updated pose.

In this way the integration of the HTC Vive is pretty trivial thanks to the OpenVR API. There is only the need to initialize the API with the call of *OpenVR.Init*:

```
CVRSystem init = null;

//Steam VR init
var error = EVRInitError.None;

init = OpenVR.Init(ref error);
```

if no error occurred during the init call, the devices are already working and their poses and buttons can be requested calling *GetDeviceToAbsoluteTrackingPose* and *GetControllerState* respectively.

With just those two calls, and the relative management code, all needed informations are available and the trajectory followed by the device can be recorded.

Trajectory recording can be started with the press of a button. During the recording every controller updated position is saved with the current trigger state and the punctual speed and acceleration in a *BacesTrajectoryElement* object and added to a *List* where all points are saved.

```
public class BacesTrajectoryElement
{
    public HomogeneousMatrix Position { get; set; } =
       HomogeneousMatrix.Identity;
    public Vector3D Speed { get; set; } = Vector3D.Zero;
    public Vector3D Acceleration { get; set; } = Vector3D.Zero;
    public long TimeMillis { get; set; } = 0;
    public bool GripperActive { get; set; } = false;
    [XmlIgnore]
    public bool Selected { get; set; } = false;
    [XmlIgnore]
    public bool Reachable { get { return RobotPosition != null; } }
    [XmlIgnore]
    public elRobotPosition RobotPosition { get; set; }
    public TrajectoryElementType TrajectoryElementType { get; set; }
        = TrajectoryElementType.LinearAbs;
}
```

Speed and acceleration will be checked against the limits of the used robot arm to confirm the trajectory repeatability.
The list with all points of the trajectory is also saved to disk, serialized in a XML file to preserve it after the program closure. More than one trajectory can be saved for a single product, allowing for complex work for a single workpiece.

3.2.2 Real-time trajectories projection

As said in the introduction chapter, one way to facilitate users in the use of off-line programming applications, is the use of augmented reality. In this case the use of a wireless and light device as the HTC Vive's controller already simplifies the users' task compared to a trajectory defined only through mouse and keyboard

in a virtual world, or using directly the robot arm in an on-line programming environment.
This solution though, lacks some visual feedback for the user; in particular for applications where the robot arm does not follow only a path, but also uses a tool. A robot arm with a paint sprayer is an example of this. For this application the tool orientation and its distance from the painted object are important as much as its position in space.
To solve this problem I have first added a shape definition form, were the user can define an object that represents the real-life workpiece in the virtual world. To define it, in a similar way to the frame definition form, the user needs to touch some points on the object with the recording device (Figure 3.4).

Parallelepiped are the only definable shape in the developed program, but this method can be easily extended to different shapes.

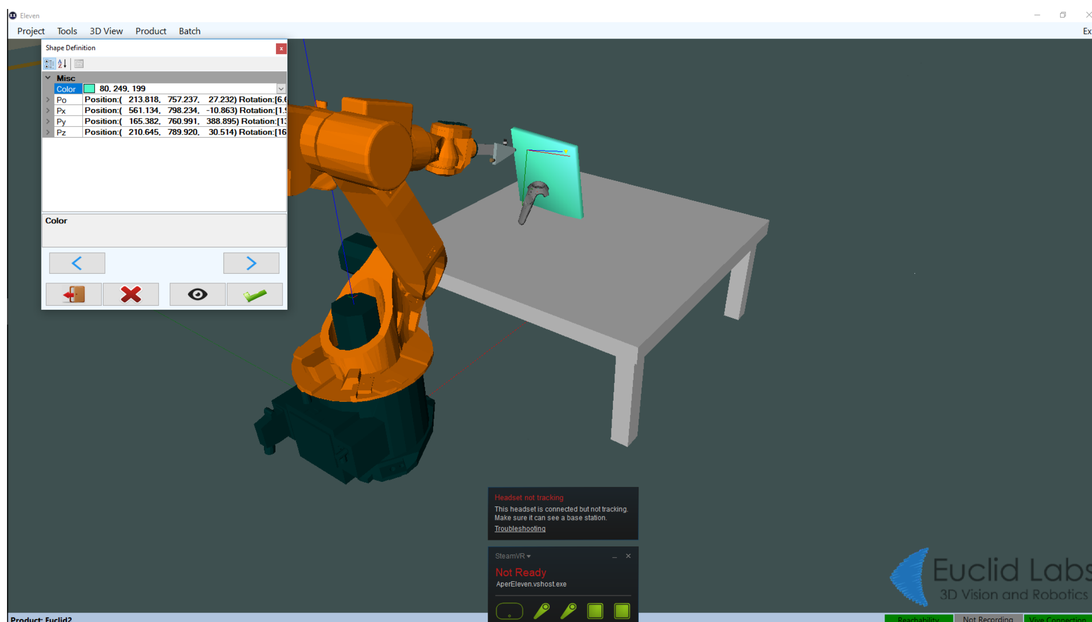


Figure 3.4: The object definition frame.

To define a parallelepiped the position of four edges are needed, three for the frontal side (po, px, py) and one for the third dimension (pz). For every point the tracking device position is saved and the four remaining edges are computed from them as:

```
//Compute the remaining edges
Vector3D pn = px + (py − po);
Vector3D pxb = px + (pz − po);
Vector3D pyb = py + (pz − po);
Vector3D pnb = pn + (pz − po);
```

With these informations the virtual shape is constructed as a list of 3D triangles

that represent the object in a *3DMeshObject*. Finally the frontal plane is saved as a normal/distance couple and a bitmap is created with a suitable size.
The "projection plane" represented by a bitmap for every defined shape, can be seen with the "Projection Window" tool (Figure 3.5). Thanks to the plane normal/distance couple saved for every shape, the intersection points between them and the recording device "projected ray" can be computed easily.
Those points are used in the "projection planes" to show, both in the virtual and real world, where the tool is pointing and how wide the spray will be based on its distance from the object. "Projections planes" will also show the "sprayed" trajectory saving the path into the previously generated bitmap.

These planes can be projected with a standard projector and easily aligned with the real object. In this way the user will see in real time on the real world, where and how much the workpiece will be sprayed by the robot arm and can change his actions accordingly; without the tedious work of checking for errors after the recording or, even more wearisome, only during a test with the real robot.

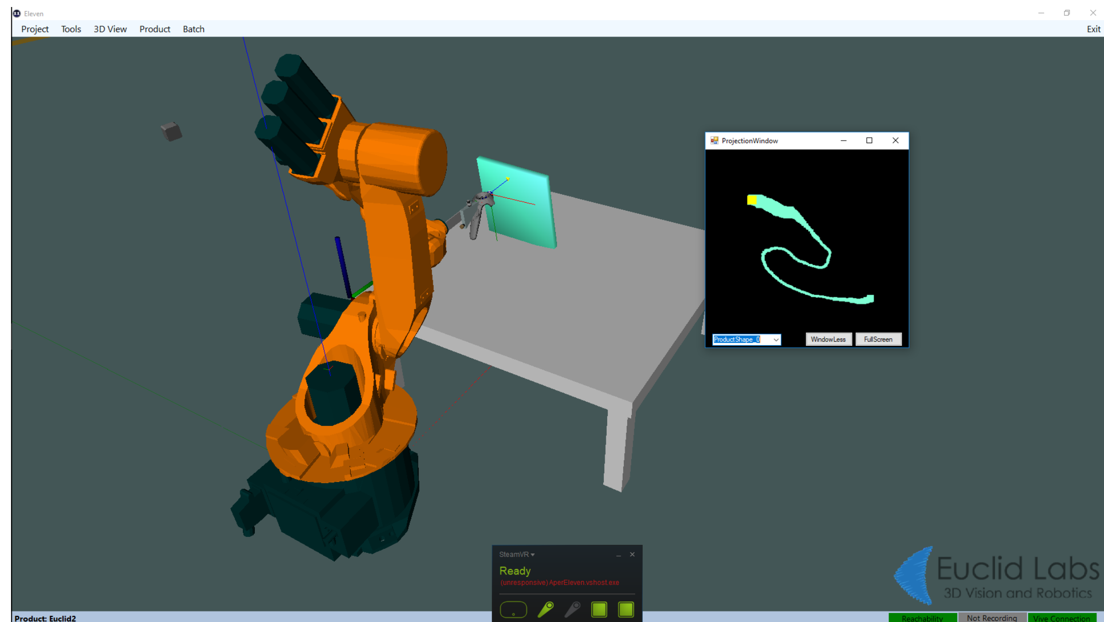


Figure 3.5: The projection window[2].

[2]Video at `https://www.youtube.com/watch?v=yarayDETz2A`

3.2.3 Showing and modifying trajectories

After the recording of a trajectory, its path will be visualized in the virtual world, with reachable points in white, spraying and reachable points in green, unreachable points in black and points with speed or acceleration higher than robot thresholds in red (Figure 3.6). The trajectory can also be executed by the virtual robot to check for eventual problems.

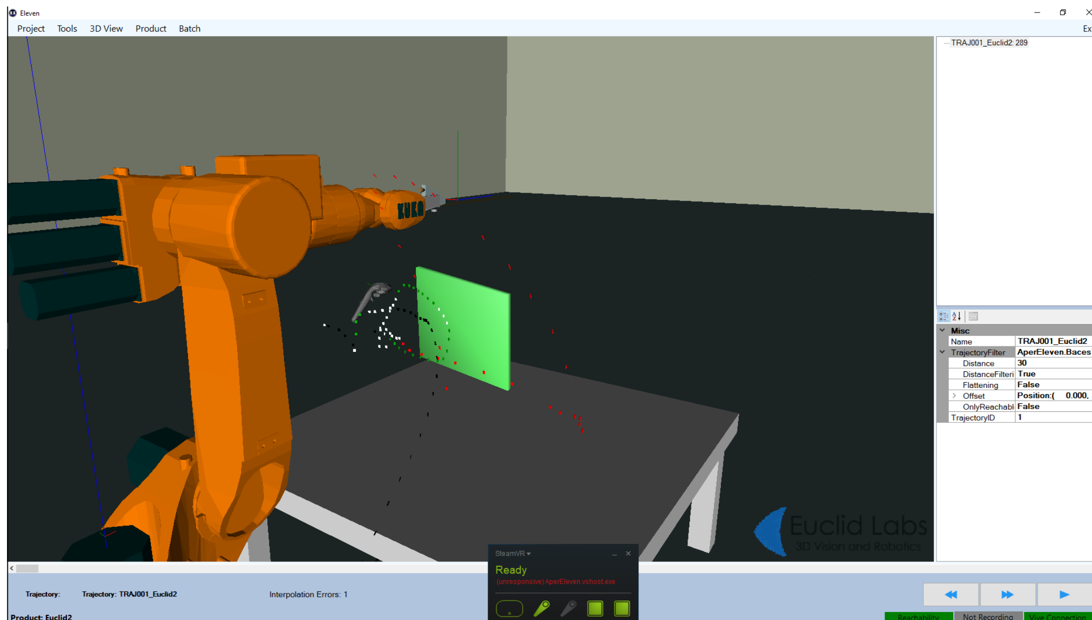


Figure 3.6: Visualization of a recorded trajectory.

Once the trajectory is displayed in the program, a number of operations can be done to it:

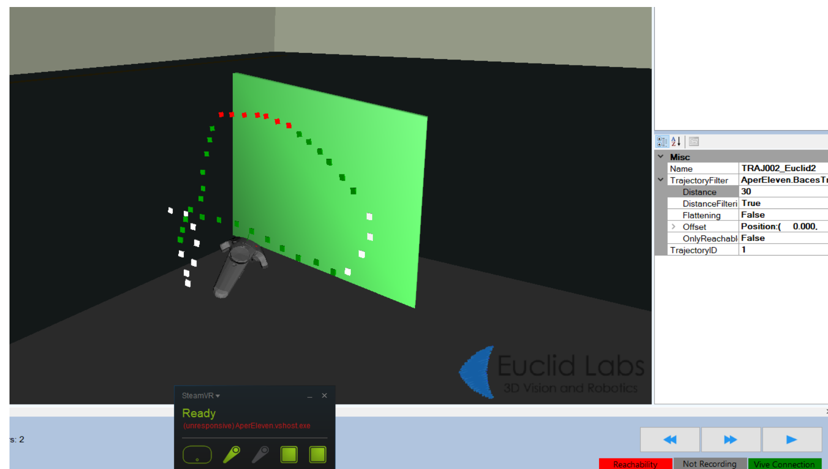
- Distance filtering; it filters out points from the trajectory that are closer than the selected threshold (30 mm by default). In this way only some of the saved points remain in the trajectory and the robot can follow them more easily as opposed to all points. Thanks to the high update frequency of the OpenVR tracking, the number of saved points is pretty high. If the robot arm would try to follow all points, its motion would be wobbling and could damage the arm itself. Removing some points smooths out the movements while maintaining the precision of the recorded trajectory.

- Flattening; it changes every point of the trajectory with the mean of more points around it, smoothing the trajectory itself. Used together with the distance filter will have the most impact and should provide the best results for the real robot.

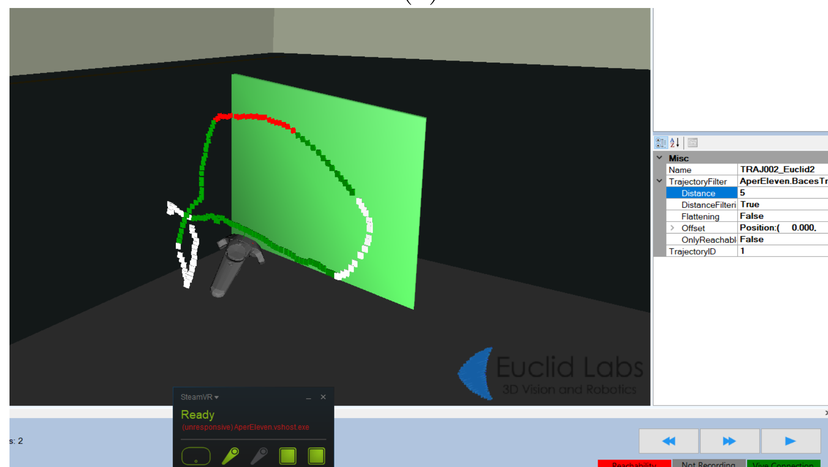
- Offset; will move the saved trajectory by a specified offset. This operation allows to use the same trajectory, or pieces of it, for different workpieces

or to correct a problem in the initial position of the trajectory (maybe the robot or the frame has been moved from the last frame definition).

- Only reachable; allows to specify if the trajectory should contain all the points or if the points not reachable by the robot should be excluded. Some points could be unreachable because there was some obstacles added in the virtual world, if in the real world this object is not present the points can be used without problems for the real robot. If instead the unreachable point are really unreachable in both the virtual and real world, they can be excluded from the trajectory and the program will generate automatically a new patch that connects the last reachable point with the next one, eluding obstacles (if they are present in the virtual world).



(a)



(b)

Figure 3.7: The filter in action, with a 30 mm distance threshold (a) and with only 1 mm distance threshold (b).

Different trajectories can also be merged together in a single one or separated in more than one. If two different trajectories are merged and the final and starting points don't match, a path that connect them will be automatically generated.

3.2.4 Sending trajectories to the robot

Every robot manufacturer has a specific way to program their robots, some have advanced functions that allow for specific movements (like B-spline), but the majority support an axis by axis programming with a couple of position/time or position/speed.

For this reason every OLP program needs a section dedicated to translate the saved information to a robot program for the specific connected robot with its language.

The program presented here is based on a precise execution of the saved trajectories and the robot program needs to respect this precision.
Transforming a trajectory in a series of position/time couples is an easy task and the results are guarantied by the robot itself. If this type of programming is not supported by the robot, a conversion to position/speed couples is needed, a much difficult task. If this second method is also not supported a precise execution cannot be guarantied.

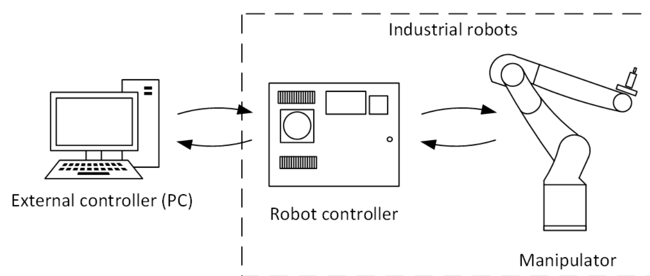


Figure 3.8: The computer send a robot program to the robot controller which will move the robot accordingly.

Once the robot program is created, it has to be send to the robot controller. This step is usually done using the UDP protocol, between the computer where the program is running and the robot controller, to send all necessary data. The robot controller will then move the robot accordingly following the robot program instructions.

3.2.5 Diagnostic tools

Some tools to monitor robot parameters have also been integrated. The Arm status (Figure 3.9) shows the current pose of the recording device with the pressed

buttons and its update frequency; the TeachPendant tool (Figure 3.10) instead shows the robot arm complete status and allows to change it. Most used functions are reachable through a pop-up menu usable directly with the controller (Figure 3.9).

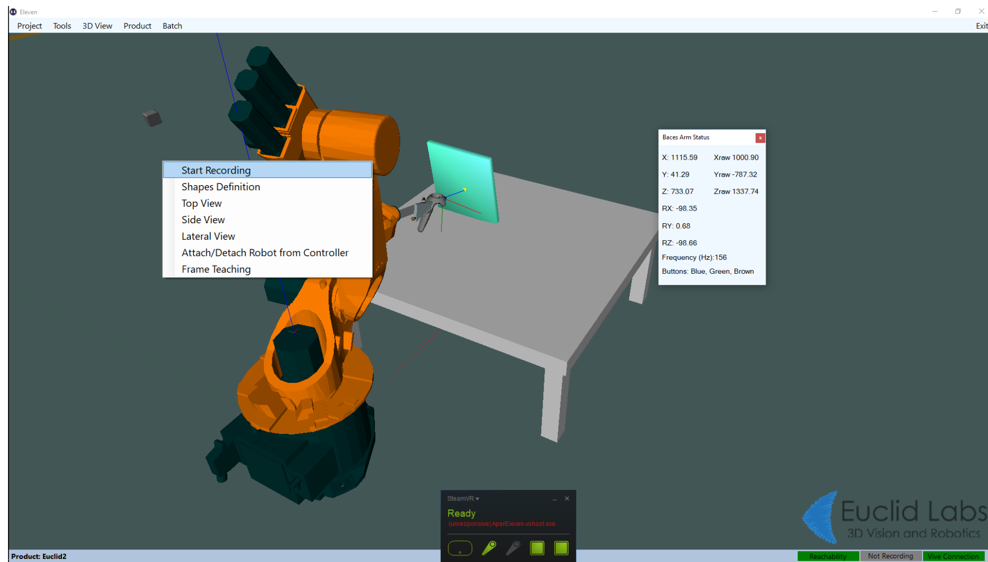


Figure 3.9: Pop-up menu of the program and Arm status tool.

(a) (b) (c)

Figure 3.10: TeachPendant tool.

4 Real-life applications

4.1 Jeans Water Brush (prototype)

The experience and technology developed in this work have also been used in a new version of the Water Brush system for bleaching jeans by Tonello: an italian company that has been a reference point for the most important laundry and dyeing companies and fashion brands all over the world for the last 35 years.
The Water Brush system[1] presented in 2015 introduced a revolutionary change to the world of denim garment treatment, nominated Launch of the Year at the Future Materials Awards. It runs on water only without permanganate and manual brushing, both very irritating and harmful for the eyes, the skin and the respiratory system; they also require considerable labour (whereas Water Brush needs just one operator) [49].

This first version of the Water Brush system used an on-line approach for the robot programming, an operator guided the robot arm through the desired path (Figure 4.1).
As discussed in the first chapter, this approach is easier and has a lower initial cost, but it also has strong drawbacks: the robot arm cannot work while trajectories are recorded and every trajectory is limited to a single workpiece (in this case a single type of jeans brush). These problems strongly limit the system performance, mainly when many new trajectories need to be recorded.
To remove these limits the system needed to be changed with an off-line programming approach. In this way big companies can command several robot cells using only a central "recording hub". In this "hub" operators can record and save different trajectories without interfering with the working robots. When a cell need to change the robot program it can select from all trajectories previously recorded in the "hub" reducing robot down-times to a minimum.

[1] The Water Brush was a prototype presented at the ITMA 2015 Exhibition, but subsequently was never industrially manufactured nor marketed. Therefore, the Water Brush machinery is not available on the market.

[2] Video at https://www.youtube.com/watch?v=UzFYGZu90ek

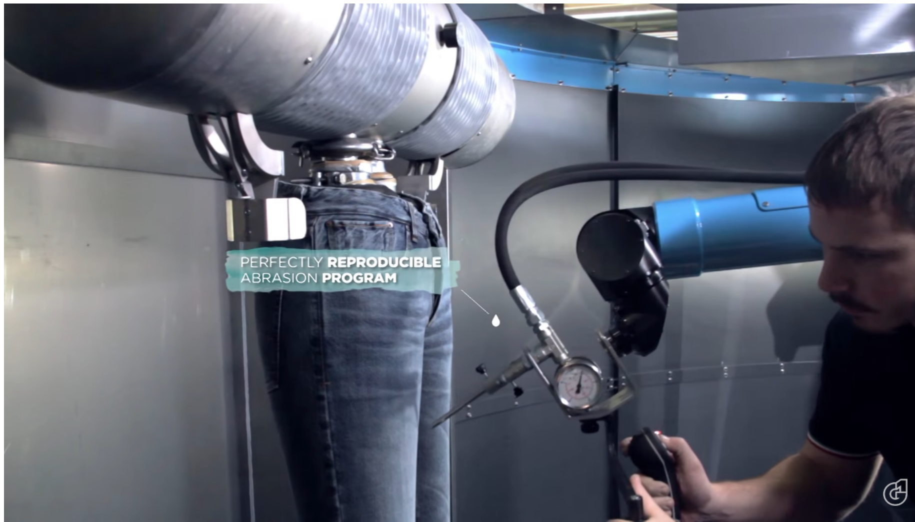
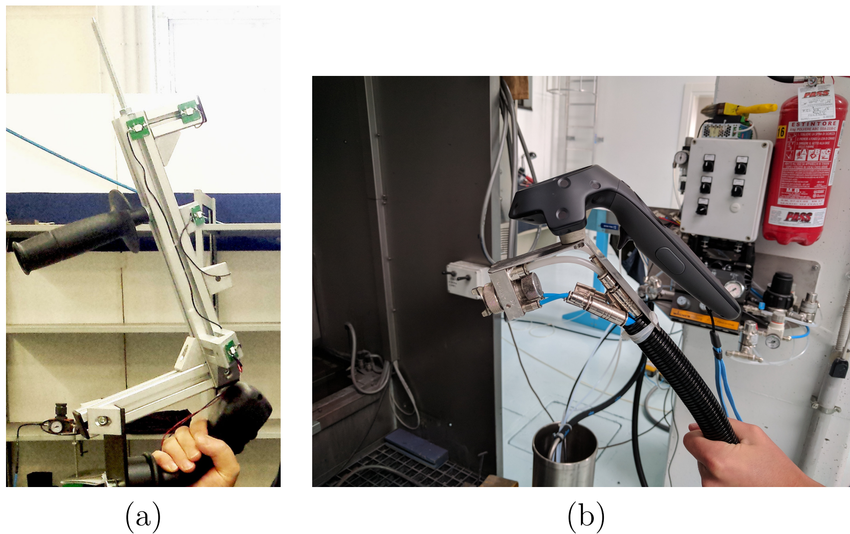


Figure 4.1: On-line robot programming in the first version of Water Brush [2].

Euclid Labs started working to this change in late 2015, designing a whole new system to track and record trajectories and developing the relative software that uses these information to program the robot arm [50]. The tracking system consisted in 2 cameras, fixed at the top of a mannequin, recording the area. An operator would then execute the trajectory with a special tool equipped with several leds (Figure 4.2 (a)).



(a) (b)

Figure 4.2: Tracking device for off-line programming in the Water Brush system, old system (a) and new system (b).

Knowing the leds positions on this device, the system would analyze the recorded video and reconstruct the performed trajectory showing it in the virtual world. This type of tracking system, constructed from scratch, has a really high cost and many engineers worked on it for several months before it could be used. It also

has a small tracking area due to cameras low FoV[3] and a cumbersome calibration process.

All these problems have already been solved in the software developed in this work, thanks to the use of the Valve's "Lighthouse" system. Its use seemed perfect for this application: a really high precision is not required, some millimeters of error does not compromise the final result, and a bigger tracking area can improve operators' quality of life. It also lowers the system total cost compared to the "home-made" solution used previously, while simplifying the calibration process and reducing size and weight of the tracking device (Figure 4.2).
Integration of the "Lighthouse" system inside the previously developed software required less than a month of work, where a good amount of time was used to remove and re-factor all code needed for the old cameras system that is not used anymore. The final result is a better system overall, easier to use for operators and cleaner from a programming standpoint.
The controller has been attached to a water spray pump activated with the trigger of the controller (Figure 4.2 (b)), in this way the operator can see physically if the trajectory he's performing is actually correct by checking the jeans used during the recording. This is the same concept of the projected trajectory used in the software developed for this thesis.
Being this a software that use an off-line programming approach, the saved trajectories can be visualized inside the virtual world and edited with several operation like roto-traslation, mirroring or stretching. With these functions trajectories can be adapted for their use even for different workpieces, without the need to record new ones every time. Another important function is the filter that will smooth the raw data to improve robot movements removing noise introduced by the user.
The software can be used in two different modes: *recorder* or *supervisor*. These modes are needed when the system is used with several robot cells. A single work-station uses the recorder mode to record all needed trajectories with the "Lighthouse" system; trajectories are then saved in a shared folder. Every robot cell has a computer where the software is in supervisor mode; in this mode the operator can create a "recipe", using one or more trajectories from the shared folder. Trajectories used in a "recipe" can be combined to perform complex works to a single workpiece. Those trajectories are saved in local memory; editing them will only alter the local version and not the original one in the shared folder. Once the "recipe" is completed it can be sent to the connected robot controller using its specific transfer protocol, the controller will then move the robot which will perform the requested trajectories.
Using this organization every robot cell is independent from others and from the recording of new trajectories, increasing significantly the productivity.

For smaller companies which use only one cell, the software can also be used in

[3]Field of View

[4]Video at `https://www.youtube.com/watch?v=vBmTDpuS-5o`

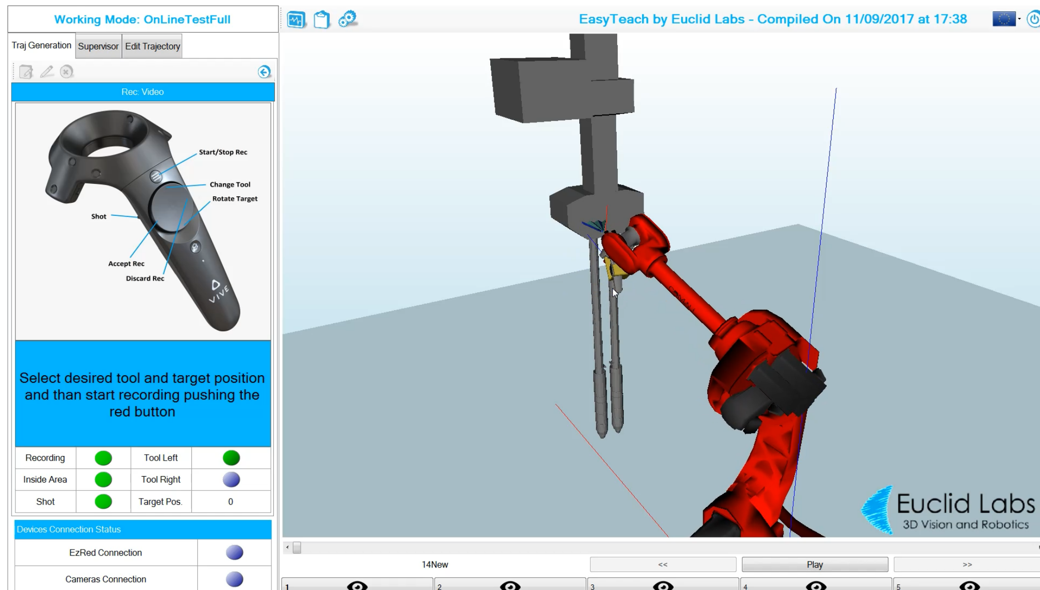


Figure 4.3: The software realized for Tonello[4].

Figure 4.4: The robot in the Water Brush system executing a trajectory.

"FullMode" where functions from both *recorder* and *supervisor* modes are enabled; in this way the same computer can be used to record trajectories, create "recipes" and send robot program to the robot controller.

To align the tracking system coordinate with the robot coordinates, the rectangle fixed at the top of the mannequin has been used. Acquiring the position of its vertices in both coordinate system a rigid transformation can be found. Every trajectory position will then be multiplied by this transformation before creating the robot program.

For this application the timing of the trajectories is as important as the spatial precision, spraying too long in a position will change the results. To control this aspect the robot program created by the software is a simple text file with a list of positions. Every position is represented by a line with the values of every joint of the robot. Every line will be executed by the robot exactly every 4 ms, in this way the trajectory can be followed precisely with an error lower than 4 ms which guarantees consistent results between different executions.

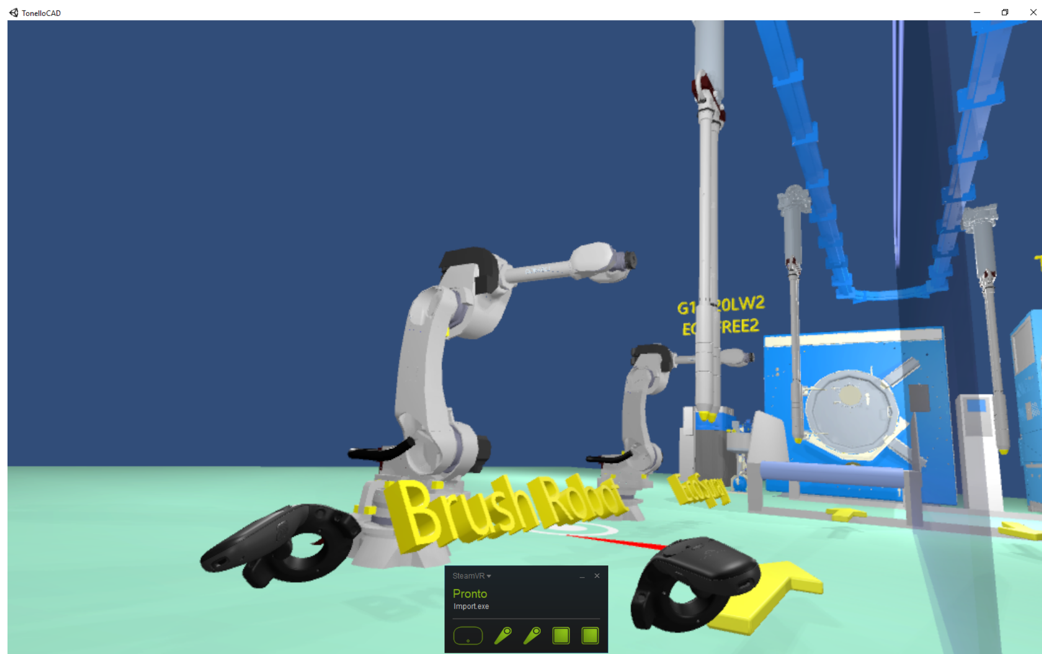


Figure 4.5: A virtual tour of the robotic cell realized for Tonello.

Another application developed for Tonello, in particular for the official presentation of the Water Brush system, is a "virtual tour" of the robotic cell (Figure 4.5) using the HTC Vive headset. I've programmed this "tour" using Unity games engine and the SteamVR plug-in. Following a basic tutorial [51], I've learned how to use SteamVR inside Unity and how to program some actions in the virtual world like grabbing objects with the Vive Controllers and teleporting around. I then added several 3D models, provided by Tonello, representing a complete robot cell for multiple processing. in this way users can explore and see all components from any position even while the real cell is working and cannot be visited.

4.2 Sofas gluing

Sofa manufacturers often use glue to fasten fabrics to the base structures. Although it is an easy task, it can be problematic for operators because of toxic fumes from the glue and repetitiveness. Euclid Labs solved this problem using a Kreon Baces arm to record the path and developed a program which will create

the robot program for the arm that will put the glue along it.

Figure 4.6: Manual gluing of a cushion [5].

While this solution solves all problems for operators security, the Kreon Baces cost is really high and moving the arm with its high DoF can be problematic for some strange path without a trench to follow.
Using the “Lighthouse” tracking system can solve both problems with a lower cost and easier movements thanks to the wireless device, its precision is also good enough for this task.

Sofas can have strange shapes and some points, where the glue has to be placed, could be hidden without direct line of sight from both base-stations. For this reason the current generation of the “Lighthouse” system that can use only two base-stations cannot be used. The next generation, that will be available in commercial products in early 2018, should be perfect even for this type of applications thanks to the possibility of using more base-stations and cover every blind spot.

[5] Video at `https://www.youtube.com/watch?v=BeEN7XzI2jQ`

5 Conclusions

In this work robot programming was analyzed thoroughly, with a particular focus on off-line programming through Augmented Reality. This method, as well as VR, is considered to be the future of industrial robot programming thanks to its easier approach compared to a standard OLP package; allowing for its use even by novice operators, which do not have previous knowledge on off-line programming, and reducing programming time without robot down-times.
An important aspect in the Augmented Reality approach is users' movement tracking, an evergreen problem in this field. Tracking solutions of modern Virtual Reality systems have been analyzed; they solve this problem in various ways, offering cheap and off-the-shelf products which can be used for this purpose.
Valve's "Lighthouse" tracking system, used in the HTC Vive, has been chosen for its openness. An exhaustive analysis of its working followed, with various tests performed to determinate its precision, accuracy and limits. Results of these tests have been used to determine on which applications this system can be used.
Finally a full OLP package, that uses HTC Vive's controller as an input device, has been developed. The software can record and visualize, in both virtual space and on real objects through projection, trajectories performed by an operator; it is then possible to generate a robot program from them and send it to robot arms that will follow recorded movements precisely.

Results obtained during tests performed in this work, showed a figure of 2 mm as error for precision and accuracy of the "Lighthouse" system, this value is low enough for many applications. Introduction of this system in these applications can greatly decrease the entry barrier of off-line programming and improve operators quality of life during its use compared to a traditional OLP approach.
A cost of roughly 600€ for the entire tracking system (in the case of HTC Vive, more products that use the same technology will be available in the future), is a fraction of the price of more advanced (and precise) systems or of a solution created from scratch; this is another great factor of the system created in this work, that will help decrease the total cost of OLP packages that use an AR or VR approach, which is probably the biggest drawback on their diffusion in SMEs.

Adapting the realized system for a real-life application like the Water Brush system thanks to Euclid Labs, was a great opportunity to compare a problem studied and analyzed only theoretically with requests and problems of a real company.
In the future this system could be used for many more applications, beginning with sofa gluing like described in the previous chapter. Introduction of the new "Lighthouse" technology[45], arriving later this year or at the beginning of next year with commercial products, will also create new possibilities for its use in robotic applications that have not yet been conceived.

Bibliography

[1] Z. Pan, J. Polden, N. Larkin, S. V. Duin, and J. Norrish, "Recent progress on programming methods for industrial robots," *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 2, pp. 87 – 94, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0736584511001001

[2] M. H. Ang, L. Wei, and L. S. Yong, "An industrial application of control of dynamic behavior of robots-a walk-through programmed welding robot," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 3, 2000, pp. 2352–2357 vol.3.

[3] M. H. Choi and W. W. Lee, "A force/moment sensor for intuitive robot teaching application," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 4, 2001, pp. 4011–4016 vol.4.

[4] S. Sugita, T. Itaya, and Y. Takeuchi, "Development of robot teaching support devices to automate deburring and finishing works in casting," *The International Journal of Advanced Manufacturing Technology*, vol. 23, no. 3, pp. 183–189, Feb 2004. [Online]. Available: http://dx.doi.org/10.1007/s00170-003-1602-5

[5] H. Zhang, H. Chen, N. Xi, G. Zhang, and J. He, "On-line path generation for robotic deburring of cast aluminum wheels," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2006, pp. 2400–2405.

[6] A. Nicholson and U. of Wollongong. Faculty of Engineering, "Rapid adaptive programming using image data," 2005, typescript. [Online]. Available: http://ro.uow.edu.au/theses/422

[7] “Kukasim by KUKA,” https://www.kuka.com/en-us/products/robotics-systems/software/simulation-planning-optimization/kuka_sim.

[8] “Motosim by Motoman,” https://www.motoman.com/products/software/default.

[9] “Roboguide by Fanuc,” http://robot.fanucamerica.com/products/vision-software/roboguide-simulation-software.aspx.

[10] “Robotstudio by ABB,” http://new.abb.com/products/robotics/robotstudio.

[11] “DELMIA Robotics OLP 2,” https://org-www.3ds.com/products-services/delmia/products/v5/portfolio/domain/Factory_Definition_Simulation/product/OLP/.

[12] “Robcad OLP,” https://www.plm.automation.siemens.com/en/products/tecnomatix/robotics_automation/robcad/robcad_olp.shtml.

[13] “Robotmaster v6 - Robot programming software,” http://www.robotmaster.com/en/products.

[14] G. C. Burdea and P. Coiffet, *Virtual Reality Technology*, 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[15] G. C. Burdea, “Invited review: the synergy between virtual reality and robotics,” *IEEE Transactions on Robotics and Automation*, vol. 15, no. 3, pp. 400–410, Jun 1999.

[16] S. W. Flaig T., Neugebauer J.-G., “Transputer-based virtual reality workstation as implemented for the example of industrial robot control,” 1993. [Online]. Available: http://publica.fraunhofer.de/documents/PX-36977.html

[17] U. Neumann and A. Majoros, “Cognitive, performance, and systems issues for augmented reality applications in manufacturing and maintenance,” in *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium (Cat. No.98CB36180)*, 1998, pp. 4–11.

[18] J. Molineros and R. Sharma, “Computer vision for guiding manual assembly,” in *Proceedings of the 2001 IEEE International Symposium on Assembly and Task Planning (ISATP2001). Assembly and Disassembly in the Twenty-first Century. (Cat. No.01TH8560)*, 2001, pp. 362–368.

[19] R. T. Azuma, “A survey of augmented reality,” *Presence: Teleoperators and Virtual Environments*, vol. 6, no. 4, pp. 355–385, 1997. [Online]. Available: http://dx.doi.org/10.1162/pres.1997.6.4.355

[20] J. Chong, S. Ong, A. Nee, and K. Youcef-Youmi, “Robot programming using augmented reality: An interactive method for planning collision-free paths,” *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 3, pp. 689 – 701, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0736584508000665

[21] M. F. Zaeh and W. Vogl, “Interactive laser-projection for programming industrial robots,” in *2006 IEEE/ACM International Symposium on Mixed and Augmented Reality*, Oct 2006, pp. 125–128.

[22] S. Stadler, K. Kain, M. Giuliani, N. Mirnig, G. Stollnberger, and M. Tscheligi, “Augmented reality for industrial robot programmers: Workload analysis for task-based, augmented reality-supported robot control,” in *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Aug 2016, pp. 179–184.

[23] T. Pettersen, J. Pretlove, C. Skourup, T. Engedal, and T. Lkstad, “Augmented reality for programming industrial robots,” in *Proceedings of the 2Nd IEEE/ACM International Symposium on Mixed and Augmented Reality*, ser. ISMAR ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 319–. [Online]. Available: http://dl.acm.org/citation.cfm?id=946248.946832

[24] “Vpl Research,” https://www.vrs.org.uk/virtual-reality-profiles/vpl-research.html.

[25] “Interview with futurist Kevin Kelly,” https://www.youtube.com/watch?v=mYatVwjtZCs.

[26] “Oculusrift kickstarter campain,” https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game.

[27] “Reverse Engineering the Oculus Rift DK2 Provides Brilliant Insight into Inner Workings,” https://www.roadtovr.com/reverse-engineering-oculus-rift-dk2-positional-tracking-camera-linux-sdk/.

[28] “Hacking the Oculus Rift DK2, part I,” http://doc-ok.org/?p=1095.

[29] “SteamVR® Tracking,” https://partner.steamgames.com/vrtracking.

[30] “SteamVR Tracking HDK,” http://www.triadsemi.com/steamvr-tracking/.

[31] “OpenVR SDK,” https://github.com/ValveSoftware/openvr.

[32] “LG’s SteamVR Headset,” https://uploadvr.com/gdc-2017-hands-lgs-steamvr-headset/.

[33] “Alan Yates interview,” https://www.youtube.com/watch?v=xrsUMEbLtOs.

[34] "Lighthouse tracking examined," http://doc-ok.org/?p=1478.

[35] "Lighthouse tracking examined, Reddit discussion," https://www.reddit.com/r/Vive/comments/4kz1ye/lighthouse_tracking_examined/.

[36] "iFixit HTC Vive Teardown," https://www.ifixit.com/Teardown/HTC+Vive+Teardown/62213.

[37] "HTC Vive Lighthouse Chaperone tracking system Explained," https://www.youtube.com/watch?v=J54dotTt7k0.

[38] "Vive position DIY sensor," https://github.com/ashtuchkin/vive-diy-position-sensor/wiki/Position-calculation-in-detail.

[39] F. Ayazi, "Multi-dof inertial mems: From gaming to dead reckoning," in *2011 16th International Solid-State Sensors, Actuators and Microsystems Conference*, June 2011, pp. 2805–2808.

[40] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME – Journal of Basic Engineering*, no. 82 (Series D), pp. 35–45, 1960. [Online]. Available: http://www.cs.unc.edu/~{}welch/kalman/media/pdf/Kalman1960.pdf

[41] "Kreon3d Baces measuring arm," http://www.kreon3d.com/scanning-arms-portable-cmm/baces-measuring-arm-portable-cmm/.

[42] G. H. Golub and C. Reinsch, *Singular Value Decomposition and Least Squares Solutions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1971, pp. 134–151. [Online]. Available: https://doi.org/10.1007/978-3-642-86940-2_10

[43] "Finding optimal rotation and translation between corresponding 3D points," http://nghiaho.com/?page_id=671.

[44] "Vive-Tracker," https://www.vive.com/eu/vive-tracker/.

[45] "SteamVR™ Tracking 2.0," https://steamcommunity.com/games/steamvrtracking/announcements/detail/1264796421606498053.

[46] "TS4231 SteamVR™ Tracking Light to Digital Converter with DATA Output," http://www.triadsemi.com/product/ts4231/.

[47] "Microsoft Visual Studio IDE," https://www.visualstudio.com/.

[48] "ELEVEN 11 How to quick and easily program a Robot," http://www.euclidlabs.it/robot-offline-programming/eleven11/.

[49] "Tonello presents Water Brush: jeans for humans," http://blog.tonello.com/2015/11/16/tonello-presents-water-brush-jeans-for-humans/.

[50] "Euclid Labs' Easy Teach," http://www.euclidlabs.it/robot-offline-programming/easy-teach/.

[51] "HTC Vive tutorial in Unity," https://www.raywenderlich.com/149239/htc-vive-tutorial-unity.