FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

TIFEE: An Input Event-Handling Framework With Touchless Device Support

Rui Miguel Vaz Teixeira

Master in Informatics and Computing Engineering

Supervisor: Rui Pedro Amaral Rodrigues, PhD (Assistant Professor)

17^{th} February, 2012

TIFEE: An Input Event-Handling Framework With Touchless Device Support

Rui Miguel Vaz Teixeira

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa, PhD (Associate Professor, DEI/FEUP)
External Examiner: Maximino Esteves Correia Bessa, PhD (Assistant Professor, UTAD)
Supervisor: Rui Pedro Amaral Rodrigues, PhD (Assistant Professor, DEI/FEUP)

17^{th} February, 2012

Abstract

The Human-Computer Interaction research field is ever evolving, producing new interaction methods, always seeking efficiency, innovation and intuitiveness. Among these innovations we have the concern with reducing the distance between natural human behavior and human-computer interaction. Interfaces which address this concern are called natural user interfaces. One particular field of these interfaces, touchless interaction, has been the object of new advancements that provide a new spot on the market for this concept. Devices, such as, depth cameras, that enable this type of interaction have been flooding the market at a fast pace, with developers and hackers trying to harness its capabilities. However, developing an application based on touchless interaction may be a cumbersome process, requiring a considerably good understanding on the subject, including frameworks and how to properly exploit their capabilities. As the interactive process does not require contact with any device it makes sense to incorporate gesture recognition, which can also be difficult.

Therefore, currently, there exists the lack of a tool that enables a rapid and easy application development using touchless interaction, while at the same time offering adequate performance and gesture recognition. This is the solution proposed and developed in the course of this thesis, a framework designed with a small set of goals as guidelines, which include the ability to provide the developer with a way of building these applications and/or adapting preexisting ones. Additionally, having a small learning curve and being possible to easily extend will possibly fill the gap described, allowing the developers to keep their focus on the applications themselves.

As a proof-of-concept two different applications were developed, using the provided solution as the backbone in the touchless interfacing process. These use cases are described and tested, as to show how easy it can be to work with, and extend, the framework.

Resumo

A área de Interação Pessoa-Computador está em constante expansão e evolução, assitindo-se frequentemente à introdução de novos métodos de interação, cada vez mais eficazes, inovadores e intuitivos. Nesta área existe a preocupação constante em melhorar a forma como o utilizador interage com um dado sistema, aproximando essa forma de interagir do comportamento humano. Interfaces que respeitam esta preocupação são coumummente chamadas de interfaces naturais. Um tipo particular destas interfaces, a interação sem-toque, tem sido alvo de significativos avanços tecnológicos, que providenciam uma nova posição no mercado para este tipo de interação. Dispositivos, como por exemplo câmaras que são capazes de perceber a profundidade de uma dada cena, penetraram o mercado de consumo rapidamente, dando a hipótese a programadores de tentar explorar as capacidades desses mesmos dispositivos. Contudo, construir aplicações que tenham por base este tipo de interação (sem-toque) pode ser um processo trabalhoso e de considerável dificuldade, sendo necessário que o programador conheça bem as ferramentas de desenvolvimento e até como funciona o próprio dispositivo em uso. Sendo este tipo de interação caraterizado pelo uso do corpo (por parte do utilizador), em particular das mãos, é apenas natural a inclusão de gestos.

Assim, é possível verificar que atualmente não existe uma ferramenta que permita ao programador desenvolver uma aplicação, que tenha por base esta forma de interação, de forma simples, rápida e eficaz, que mantenha ao mesmo tempo níveis de performance aceitáveis (tendo em conta o contexto de interação). Resumidamente, esta é a solução proposta para o problema identificado: uma *framework* que tenha por base um pequeno conjunto de objetivos, que incluem uma forma simples de construir novas aplicações interativas sem-toque e ao mesmo tempo permitir adaptar aplicações pre-existentes para esta forma de interagir. A possibilidade de extender a ferramenta, adicionando novas funcionalidades é também uma funcionalidade a incluir, pois deve permitir manter a mesma atualizada de forma simples, permitindo assim que o foco seja no desenvolvimento da aplicação e não no suporte de interação.

Como prova de conceito foram ainda desenvolvidas duas aplicações distintas, tendo como base a ferramenta desenvolvida. Estas aplicações, em conjunto com a ferramenta, são descritas e testadas, por forma a mostrar quão fácil é trabalhar com a solução proposta e ainda adicionar à mesma novas funcionalidades.

Acknowledgements

A simple, heart-felt thank you:

- to my family and friends, for the support, as well as to my colleagues;
- to my supervisor, for his guidance and support.

Rui Teixeira

Contents

CONTENTS

List of Figures

List of Tables

Abbreviations

2D	Two dimensions
3D	Three dimensions
AI	Artificial Intelligence
API	Application Programming Interface
BSD	Berkeley Software Distribution
CLNUI	Code Laboratories Natural User Interface
CRT	Cathode Ray Tube
FAAST	Flexible Action and Articulated Skeleton Toolkit
GNU	GNU's Not Unix
GUI	Graphical User Interface
HCI	Human-Computer Interaction
IR	Infra-Red
LED	Light-Emitting Diode
LGPL	Lesser General Public License
MIT	Massachusetts Institute of Technology
MS	Microsoft
NITE	Natural Interaction for End-User
NUI	Natural User Interface
OpenCV	Open source Computer Vision
OpenNI	Open Natural Interaction
OS	Operating System
OSC	Open Sound Control
R&D	Research and Development
RGB	Red, Green, Blue
SDK	System Development Kit
TIFEE	Touchless Interaction for Enriched Experiences
TV	Television
UDP	User Datagram Protocol
UI	User Interface
XML	Extensible Markup Language

Chapter 1

Introduction

Throughout the years, since the first computer was invented, until today, not only the computer itself but also the way we interact with computers has been constantly evolving. This evolution is carried through the hands and minds of those who dedicate their time researching this human-computer interaction (HCI) area, bringing exciting new interfaces and interactions that are more natural and intuitive for the common user.

1.1 Human-Computer Interaction and Natural User Interfaces

One can define HCI as a research field, that comprises different subjects, concerning the design, evaluation and implementation of interfaces and their context, seeking the efficient use of computer systems.[MS91, HBC+96, HH07]

Regarding the design of efficient interfaces there is the concept of Natural User Interfaces. This NUI model refers to a type of interface which, ideally, is invisible to the user, implying that the user perceives the interface as natural. This means that the interface does not need to perceive the natural human behaviour in order to be natural. As Daniel Wigdor and Dennis Wixon state, in "Brave NUI World": "*A NUI is not a natural user interface, but rather an interface that makes your user act and feel like a natural. An easy way of remembering this is to change the way you say "natural user interface"—it's not a natural user interface, but rather a natural user interface.*". So, NUI is a form of providing HCI, and there is no particular method for achieving NUI, given that it incorporates the need to assess "the domain of use and the requirments of context". [WW11]

As it will be detailed, mostly in the second chapter (State of the Art), there currently are several devices that can provide an interaction context that can fit the NUI concept.

1.1.1 A Focus on Touchless Interaction

After the recent boom of touch-based interaction devices - tablets and smartphones being two common examples - we are starting to see a new wave of interaction technology, seeking to free the user from contact with any device - touchless interaction. Although there are different interpretations on what touchless interaction is, throughout this document it shall be considered as an interaction between at least one user and a certain system that does not require the user to manipulate any kind of device. Voice or speech recognition fits this description but the focus will be given to gesture-based interaction, in which the user interacts with a given system using a body member, such as a hand, or even his whole body.

So, this project will focus on touchless human-computer interaction and particularly with the support of hand gestures.

This form of interaction is not a novelty, or recent, but it has recently gained momentum, in great part due to the evolution and availability of more powerful hardware. This hardware evolution occurred both at the level of input devices, processing units and memory/storage capabilities. This fosters new methods in the area. Considering the previous definition of touchless interaction and its intrinsic relation to the field of computer vision, it is also acceptable to state that the borders of computer vision have been expanded, with new methods/algorithms emerging and the access to the technology itself continuously increasing.

As any form of interaction between at least one human and a machine, touchless interaction can be applied in many different contexts. Considering the touchless gesture-based interaction there are certain cases in which it can be of particularly practical use, for example, in scenarios when touching any kind of device in order to interact with the system is not acceptable. Sterile environments, like a surgery room, are a great example [JOS+11]. Entertainment-oriented applications are also a great target for this form of interaction [WKSE11].

In spite of all of the possible applications in those fields, interactive multimedia installations are to be considered the primary use, or possible target, of the work presented here (not meaning that it cannot be applied in other cases). So, it is important to define this concept of interactive multimedia installations considering, once again, the framing of this thesis. When referring to these installations, given the context, one must note, at least, two distinct situations in which they are applicable, or useful: information stands and interactive multimedia art installations.

Public information stands are used to provide information to the public, usually relative to the surroundings of the stand itself. These stands can be located in several distinct places, such as a museum or a place of historical significance, a monument or an important town square, a governmental institution, a school, or a private corporation branch stand,

Figure 1.1: Two examples of public information stands.

a bank. A simple art installation can also employ touchless interaction so as to provide the viewer with a more natural form of interaction, like Myron Krueger's "Videoplace" [KGH85]. These use cases were taken as the most probable situations, or places, in which a touchless interaction process might best fit. Although, none shall be seen as restrictive in terms of when and where touchless interaction can be used.

1.2 Problem Definition

As we walk towards a more and more ubiquitous computing environment, more natural forms of interaction become more of a need than a simple technological novelty. As shown, touchless interaction is an example. However, developing applications using this form of interaction might not be trivial. Also, there are numerous devices currently available, offering distinct forms of interaction. Some inteded to provide a more natural experience too, like the Microsoft Kinect (provides touchless interaction) and the Wii Remote (a controller that can detect gestures).

Despite all the advances in the field of human-computer interaction, developing new software using these types of interaction is still considerably laborious, having the developers to dedicate some time integrating technology, namely several software frameworks that demand a considerable amount of time to understand.

So, one of the open issues today is to harness the power of these devices, in a proper context and in a simple and practical way. The focus of this Master Thesis, however, will be given to touchless interaction, as described previously. Still, the solution provided shall be sufficiently generic so as it can provide, without much effort, integration of other forms of interaction.

The understanding of a need for a simpler, yet powerful, way[1] to rapidly develop

[1] A simpler, easier way to develop when compared to developing the same intended application, with the same features, from scratch, that is, not taking advantage of the solution presented in this document.

applications based on this type of interaction, comes from the concept of HCI itself. Since HCI covers such a wide area of scientific research and development, there are bound to be people without the technical knowledge needed to pursue their goals in what regards to touchless interaction. At the same time, enabling a faster application development based on touchless interaction allows for rapid idea prototyping, which can be useful in a commercial context too.

As it will be shown throughout the document, there is also a lack of a standard API that supports gesture recognition and is capable of interacting with preexistent applications. So, a middleware solution that can fulfill these needs is missing.

1.3 Motivation

Since touchless interaction is not a fully developed interaction process and it has only recently gained a share on the mass market there seems to be many possible ways of contributing to this interesting topic [GJM11, Rom10]. Building a new tool that can help others in their efforts to develop new applications and prototypes is usually a good contribution. Specially when considering the time usually spent studying documentation. Such a tool would allow its users to focus on what really matters.

1.4 Goals

Given the context and problem at hand, the objective of this thesis is to provide a system for supporting touchless interaction with the following goals in mind:

- The ability to function as a standalone application, gathering data, processing it and feeding it to a separate, preexisting, application;
- The ability to serve as a library, or framework, allowing other developers to build their applications on top of it;
- Being expansible, so that it can be updated, by adding extra capabilities, as needed;
- Being commercial, that is, complying with the possibility of being included, used in a product or service that can be sold;
- Recognizing hand gestures and allowing for adding new gestures;
- Allow for a rapid and easy application development (minimizing the need for the developer to study further documentation);
- Given the solution's main purpose (to be used for interactive multimedia installations), building a well responsive (low latency) framework should also be considered a goal.

This arrangement of goals outlines what the project solution must be, mainly in terms of its functionalities and how they should be provided to the ones wanting to take advantage of the solution. In spite of these objectives narrowing what should be done and how it should be done, there are important decisions regarding, mostly, the architecture and the implementation that need to be made. This is explained, in detail, in the respective chapters, 3 - Proposed Architecture - and 4 - Implementation.

In short, a functional middleware solution is expected to be built, that can cover all the requirements presented.

Although not listed as a requirement, using Microsoft Kinect, or a similar device, was mandatory, mostly because it allowed to gather some important input data, together with the proper software, such as the user's hand point[2] and the skeleton[3], with the benefit of capturing these data in a three-dimensional environment. These topics - recognizing a hand, retrieving a user's skeleton - represent a distinct area of scientific research, that goes beyond this thesis' purposes.

In spite of not being properly a goal, it is expected that the solution provided is useful for a single user, that is, any application built on top of the solution should work for a single user, since it is considered that multiple users are out of this MSc thesis' scope. The same goes for multiple hand gestures.

So, having outlined the goals, there is only the need to present the solution name: TIFEE. It stands for Touchless Interaction for Enriched Experiences.

1.5 Framing

Concerning the development of this thesis, it involves two different entities: Ubiwhere (the proponent) - a business organization of technological base, focused on R&D in the areas of ubiquitous/pervasive/invisible computing and environments [Ubi] - and FEUP - Engineering Faculty of the University of Porto.

1.6 Structure

This document is structured as described next:

In the current chapter (Chapter 1 - Introduction) the theme at hand is introduced as well as the specified problem, given the context. The expected results are also presented, as a list of goals.

[2] A hand point is a three-dimensional point that represents the user's tracked hand, as he/she is interacting with the system.

[3] A user's skeleton gathers information about each of the user's joints, as perceived by the hardware sensor - MS Kinect.

Chapter 2 - State of the Art - encloses a study on the subject of Human-Computer Interaction, with a focus on touchless interaction, providing scientific and technologica. A particular emphasis on gesture recognition is also supplied.

Chapter 3 - Proposed Architecture - provides a close view of the proposed solution's architecture, offering an explanation on how the solution's architecture was designed taking into account the aforementioned requirements. In turn, chapter 4 - Implementation - covers the details of implementing the architecture.

The 5th Chapter - Tests and Results - includes the details of the testing process, along with the respective test results, serving as a proof of concept for the proposed solution, described in the previous chapters.

In the 6th Chapter - Discussion - the test results are discussed, from an exempt point of view. An overview of the solution is presented, describing possible improvements.

Finally, the 7th Chapter - Conclusions - holds a final wrap-up, contemplating what was described in the preceding chapters, as well as some deductions, or inferences, considering the solution, as an answer to the identified problem. This final chapter also holds some considerations on future work and how [this future work] it influenced the design and implementation processes.

Chapter 2

State of the Art

Throughout the years, since humans started to interact with computer machines on a regular basis, many means of conducting this interaction have been developed and introduced into the market, each with the main goal of providing a simple, intuitive yet effective way for humans to interact with computers. Several alternative HCI methods were also created but did not get to the wide public market. However, as described below, these systems are constantly being developed and updated.

Looking at the context described in the first chapter (the information stands), currently, there are a lot of devices that provide different takes on HCI, These myriad of devices brings about an issue: the necessity of a tool that can provide an abstraction for all of the data made available by these devices.

2.1 Historical Context

Thanks to the vertiginous evolution of technology in the past two decades (90s and 00s) [Wika, Wikb] one can state that computing technologies have become available everywhere, thus introducing the concept of ubiquitous computing - a post-desktop HCI model, in which computers are envisioned to be embedded in a natural environment, in a non-intrusive manner, providing easy access to information, everywhere and communications through wireless networks [LY02, Wei93].

It is common, today, when thinking of computing, or a computer, to immediately imagine a keyboard and a mouse. These two of devices were not the first devices used to interact with a computer, although they have been around for some time now [Bel]. A pointing device, nowadays known as the trackball - see figure 2.1 - was described around the year of 1952, by Tom Cranston, Fred Longstaff and Kenyon Taylor [Var94]. A few years later, in 1965, Douglas Engelbart introduced, amongst other inventions, the first

mouse prototype [Bar97], similar to the mouse we know today. Together, the keyboard and mouse devices still constitute the primary method of human-computer interaction, mostly due to their versatility, albeit, sometimes, this pair of peripherals can present itself as a barrier to initiate or even allow the progress of interaction (depending mostly on user experience/knowledge).

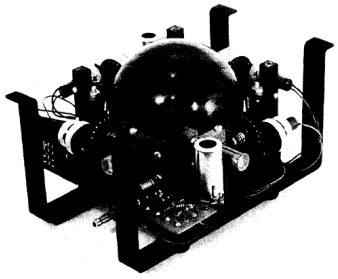

Figure 2.1: The DATAR Trackball, the first invention of this type, a pointing device [Var94].

Older than the computer mouse, the keyboard has become one essential productive tool, after the decline of the punch cards and paper tapes [Wike]. Not several years ago it would be difficult for so many to imagine using a computer without a keyboard. As an indirect descendant of the typewriter [Wiki] the computer keyboard inherited this key stroking method, serving as the standard input method for practically any computer. Even the QWERTY keyboard pattern was inherited from a 1874 typewriter [Wikm]. As every other tool, or peripheral, the keyboard has evolved, being in the spotlight as one of the most used computer peripherals; new keys, representing new functionalities were added, virtual keyboards were created (so people with reduced mobility capacities could use a computer), there were adaptations to fit new gadgets (such as cellphones) and there have been several different approaches to the regular keyboard, varying in accordance with the pretended usage (gaming, for example). Even though, the keyboard is still not acceptable in some particular interaction contexts, as public displays.

We have also witnessed the birth of other peripheral devices to HCI, focusing a specific set of tasks, such as graphic tablets [Wikd] (intended for recording drawing and handwriting) and gamepads (used to play games). Remote controllers (in the likes of TV remote controllers) and other devices (e.g. smartphones), that can be paired with a computer through some sort of wireless connection (IR, Bluetooth, WiFi) are also used to accomplish some tasks in a computer.

Nowadays, we look at the mouse and keyboard and tend to see a productive tool, as other means of interaction are emerging, being made publicly available, in the technological market. These other forms of interaction have been gathering the interest, and a fair share, of the technological market, for several reasons, being the ease of use and the constant lowering of the prices of such devices, some of those reasons. A touchscreen is an example of this form of interaction. Multi-touch has now been on the spotlight for a

considerable amount of time, spreading mostly through smartphones and tablets but being equally useful in public installations - malls, shops and fairs are a few examples.

Figure 2.2: The Apple iPad, a multi-touch enabled tablet computer [App].

2.2 Common Input Devices for Interactive Setups

Next, a brief review of some devices is presented, considering what was mentioned until now, describing in detail technology related to touchless interaction, while covering superficially other HCI related devices. Some of the devices listed below gathered much interest of the hacking community, for its particular capabilities, being the Wii Remote an example [BW07]. Beyond the interest of this hacking community, this gadgets have been the focus of interest of many HCI studies and researchers. So, this list should be regarded as a list of devices that are currently eligible as tools for an interactive multimedia installation, as described in chapter 1 - Introduction and, therefore, could be used with the solution.

2.2.1 Keyboard and Mouse

This pair of devices are still considerably common in today's public displays, in spite of the fact that they are not very intuitive to work with, in such interaction context. In fact, despite being part of the HCI process for a very long time, the keyboard and/or mouse can pose as an interaction barrier since they usually require a considerable amount of time to get used to. They are also not very practical, again, considering the interaction context.

2.2.2 Touch

Albeit the term "touch" is, nowadays, strongly connected to multi-touch devices there are a several other applications not multi-touch based. This type of devices is usually referred as touchscreens, as it's a more comprehensive/broadening term since it comprises screens or surfaces that are not, necessarily, multi-touch enabled.

Despite that this type of technology is usually referred through smartphones and tablets, touchscreen technology first appeared around 1965, invented by E. A. Johnson

[Joh69]. Virtually, any surface can support a panel that provides touch sensing capabilities. Touch sensing functionality is provided through several distinct techniques, being resistive and capacitive two of the most used [Wikl]. Resistive touchscreens are usually composed of two conductive layers separated by an air gap. When these two layers are pressed, they touch each other and, through electrical current, the contact point can be determined [Wikg]. Capacitive touchscreens are also typically composed of two layers, being one an insulator and the other a conductor. Capacitive touchscreens work based on the conductivity of the human body; when the user presses the screen, the electrostatic field is altered, allowing to detect the contact point [Wikc].

In public displays this type of devices can be quite useful, since they enable a more natural interaction (when compared to the use of a keyboard, for example). Users can quickly and easily navigate through panels of information and select what they want to. There is no need to handle a computer mouse, or a trackball, to point to where the user wants to click.

- **DiamondTouch**: DiamondTouch was presented to the public in 2001, emerging from the work of researchers at the Mitsubishi Electric Research Laboratories [Cir]. This piece of technology is the outcome of research on a touch surface that aimed to achieve certain characteristics, such as multi-point detection and multi-user, associating each touch point with the respective user. It was the first multi-touch surface offering multi-user capabilities, functioning through the use of antennas - placed on the surface - and receivers - coupled with each user, meeting the requirement to identify each contact point, i.e., associate each point with the respective user [DL01].

Figure 2.3: DiamondTouch - multi-touch surface with a projector on top, allowing to place any kind of objects on the surface [Cir].

- **Microsoft Surface 2.0**: The second version of Microsoft's Surface tabletop makes use of its PixelSense technology to detect points of contact. Although it does not use cameras, PixelSense's touch point detection is based on the detection of IR light emitted from the surface itself through a set of sensors [Micd].

Figure 2.4: Microsoft Surface 2.0 - The LCD layer contains sensors that can sense points of contact through IR reflection [Sam].

Other approaches to touch surfaces are worth mentioning, particularly Edigma's products, such as Displax Skin[Edi] that enables multi-touch capabilities on any surface, making it specially useful for public displays.

2.2.3 Motion Controllers

In this document, the term motion controller is refers to a range of devices that enable interaction between the user and the system based on a controller that is able to sense its movement and orientation.

2.2.3.1 Wiimote (Wii-Remote)

Nintendo's Wiimote (figure 2.5, on the left) was a game changer [Wiko] from the moment it was announced, back in 2005, considering the market for which it was being targeted, the videogame market, since it was the first game controller whose functionality was based on sensing, or detecting movement, its producer intended so, from the first moment it was just an idea. Such functionality is possible through the use of an accelerometer (three degrees of freedom: forward-backward, left-right, and up-down) and an IR sensor, that combined with a plastic bar containing several IR emitters allows for such things as detecting if the controller is pointing left, right, up, down and the distance from the bar to the controller [Nina]. Later on, Nintendo developed a boosted version of the Wii Remote, called the Wii MotionPlus, featuring a set of renewed sensors, allowing for a more precise tracking of the device [Wikn, Inv08].

Wiimote has been used on several projects, "Don't touch me" is a simple example, where this device is used to navigate in a map and mark areas of interest using geometric shapes and associating voice annotations. It is also multi-user enabled, since it can support several Wii Remotes [BMDA10].

2.2.3.2 Playstation Move

Sony's PlayStation Move (figure 2.5, on the right) was launched in Europe around September, 2010. Despite not being the first PlayStation accessory offering motion sensing capabilities - the Sixaxis and DualShock 3 gamepads had this capability - Move is a complex controller, considering its features and compared to prior PlayStation controllers.

PlayStation Move works as a combination of the PlayStation Eye camera and the Move controller itself [Son]. The controller is composed by some buttons, a rubber ball (containing an RGB LED inside) and some electronic components [IFib], a two axis plus a one axis gyroscope, an accelerometer and an electronic compass [AKM]. The RGB LED provides a bright light, variable in color. This allows the tracking of the device, through the PlayStation Eye camera.

Figure 2.5: Motion Controllers - Nintendo's Wiimote[Ninb] on the left and Sony's PlayStation Move[Son] on the right.

These devices are commonly used to provide a gesture-based interaction context, since the data that their output is based on position and movement. Considering a generic solution and the listed goals (section 1.5 - Goals), the usage of this type of devices could be easily integrated, providing, as an example, the position of the controller in order to recognize a gesture.

2.2.4 Computer Vision

The *webcam* is referred here as the common device used in computer vision, although other similar devices are used, such as IR cameras (sensible only to IR light, reflected naturally by the human body). Comparing to all the technologies listed until now, the only technology that truly enables touchless interaction is computer vision, since it does not require manipulating any kind of device. A *webcam* is no more than a camera that provides a colorful frame, or image, to a computer.

The first *webcam* was invented around 1991, in Cambridge [Fra95] and since then this device has been constantly evolving, offering better frames at a lower price per device. The availability of these *webcams* has also increased over the years, being nowadays a common element on every laptop and smartphone.

Considering that the technology is largely available, new interaction methods have emerged, taking advantage of its use. However, when working with a common *webcam*, one has to deal with some issues. A low resolution frame can pose as a problem, making it difficult to perceive, or detect, certain features in that image - identifying someone through facial recognition algorithms, for example.If the image sensor quality of the *webcam* being used is low, lighting conditions can drastically affect the obtained image. Low lighting conditions (excessive or insufficient light) can also preclude the proper recognition of certain features. Considering the photographed scene, even the quantity of moving elements captured - in case of processing a sequence of images - can represent a major problem, since it can be hard to define what belongs in the main scene, that is what should be processed, and what belongs in the background scene, what should not be processed.

Several solutions, or partial solutions, have been proposed to solve some of these problems. These solutions are image processing algorithms, that can enhance image quality, by adjusting the brightness and/or contrast of a given picture. Other solutions are focused on particular tasks, such as the operation to detect the contours of an object. This is a fundamental operation nowadays, in computer vision [DG01] and was first theorized in 1980 by D. Marr and E. Hildreth in their paper "Theory of edge detection". One popular algorithm to accomplish edge detection was described by J. Canny, in 1986. This algorithm became known as the Canny edge detection algorithm - a sample of the edges detected by this algorithm can be seen in the figure 2.6.

These and similar techniques serve as a basis for several applications, from motion tracking software to facial recognition, including applications in the food industry [Bro04] and gaze tracking. Some of these applications are interaction oriented, providing a panoply of novelty human-computer interaction methods. Myron Krueger began developing this particular subject in the late 1960s in his Metaplay interactive installation, where the user, or visitor, would be filmed by a camera, connected to a computer in another room, allowing an artist to draw pictures over the video feed and feed them back to a projector located in the same room as the visitor [Kru77].

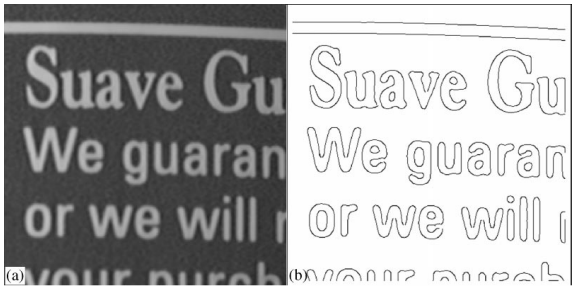

Figure 2.6: Canny Edge Detection - On the left, the original image. On the right the same image after being applied the Canny Edge Detection algorithm [DG01].

After developing another project, called Physic Space, Krueger went on to develop

Videoplace, probably, his best known installation. Critter, a part of the Videoplace project, demonstrated an early version of what can be called an interface controlled by gestures; a virtual creature - Critter - would respond to the actions of the user, perceived by the system through his/her silhouette, that in turn was obtained through a camera. Critter would flee from the user if the latter moved too quickly in its direction but, if the user held out his hand, the creature would land on it performing a series of movements attached to the user's body [KGH85].

Since this technology has become so widely available, it fostered developers' creativity, allowing them to come up with several different uses for it. Computer vision has become crucial in much of today's touch-interaction based installations since it is not expensive and allows the use of other techniques, such as detecting fiducial markers[1] placed on top of the surface [JGAK07], as reproduced in 2.7.

- **reacTable** is an example of a multi-touch surface based on computer vision, using rear projection, an IR camera and IR LEDs to provide illumination on the surface [KB07]. This technique is known as diffuse illumination [Groa]. Since it is based on computer vision, markers placed above the reacTable can be detected;

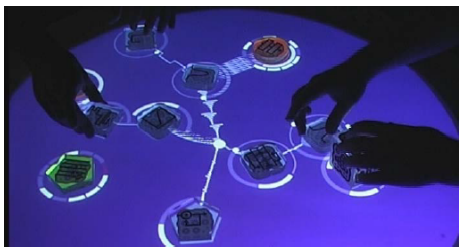

Figure 2.7: reacTable - A reacTable multi-touch surface with fiducial markers on top, recognized by the reacTIVision framework [JGAK07].

- **Multi-Touch Interaction Wall**, is another touch surface using computer vision to accomplish multi-touch sensing capabilities. It is based on the FTIR (Frustrated Total Internal Reflection) principle that also uses IR LEDs, rear projection and an IR camera, but the IR light stays trapped in the surface, being released (frustrated) when touch occurs [Han06, Grob];

[1] Fiducial markers are printed images, or elements, recognized by a computer vision system, so as to represent a certain object, virtually. Usually information such as the distance (from the camera) and the rotation of the marker are perceived.

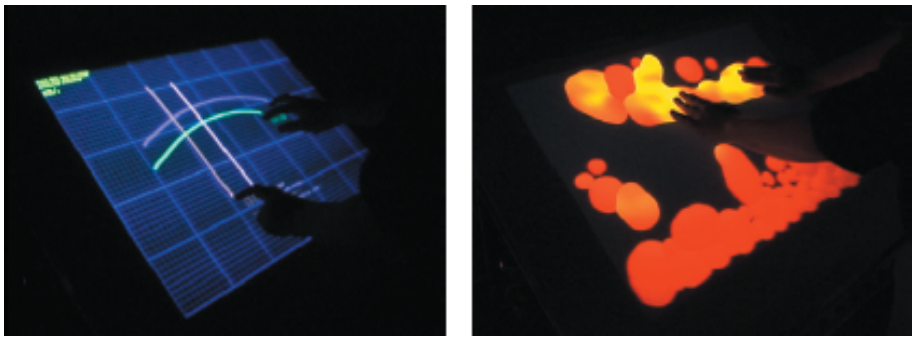

Figure 2.8: The Multi-touch Interaction Wall - Two usage samples [Han06].

Using IR cameras, one can easily overcome some of the obstacles that are inherent to the traditional *webcams*, such as the projection being visible to the camera and poor lighting conditions, as perceived through the example projects mentioned above. The article "User Interface by Virtual Shadow Projection" describes a project that also uses an IR camera, to achieve touchless interaction between the user and the system (see figure 2.9). The IR camera is aimed at the user shadow - an IR shadow - providing the computer with a live feed of frames containing, virtually, only the user silhouette. Each frame is processed in the computer, feeding, in turn, a finger tracking algorithm [XIHS06]. The output of this algorithm allows the user to control the system.

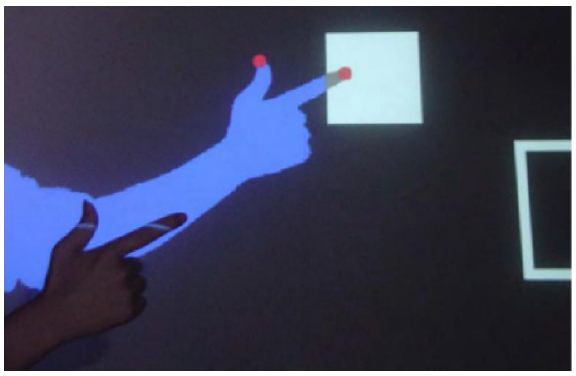

Figure 2.9: Virtual Shadow Projection - A hand is captured by the camera. The silhouette is projected on the surface and the index and thumb fingers are marked with a red dot on each respective fingertip [XIHS06].

Even though solutions based on IR cameras can prove to be a better solution in some cases, there are solutions based exclusively on the usual digital cameras, or *webcams*. Some of these solutions are listed next:

- **Microsoft Touchless SDK** is an open source project maintained by Microsoft. It works by tracking colors , so its usage can be somewhat limited [Micg].
- **Camspace** is very similar to the Touchless SDK but it is more commercial oriented. It also works by tracking colors [Cam].

It is not very hard to pinpoint the flaws of these two solutions, based on what was mentioned so far about computer vision. Using one of these solutions would imply a somewhat controlled interaction context, or environment, as well as the usage of easily trackable objects, that is, objects that can not be easily mistaken with other parts or elements in the background, that are not meant to be tracked.

In these techniques the use of bright colored objects is common, since tracking a hand, for example, can be difficult. Even considering using certain workarounds so the user does not have to hold any object, such as using fiducial markers [KB07] on each fingertip or colored markers, true touchless interaction is not achieved.

However there are other methods, or techniques, that in fact try to achieve true touchless interaction, besides the IR shadow method mentioned before. As demonstrated in the article "Real-time Hand Tracking and Finger Tracking for Interaction", it is possible to use two inexpensible *webcams* to accomplish stereoscopic vision[Mal03]. In this particular example, there is the need to detect both thumb and index fingers to perform a simple pinch gesture[2], as demonstrated in figure 2.10.

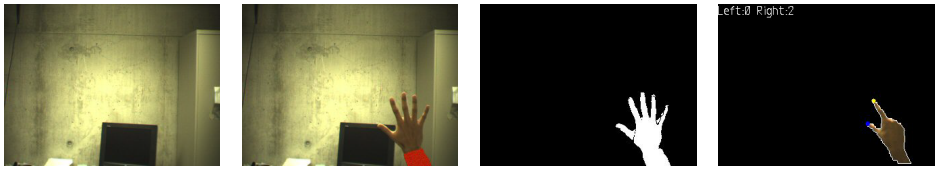

Figure 2.10: "Real-time Hand Tracking and Finger Tracking for Interaction" - From left to right, on the first image it's possible to observe the background scene, subtracted to the second image to later provide a mask of the hand - the third image. The fourth and final image represent the recognition of the index and thumb fingers [KEA03].

Hand and finger tracking are a considerable part of what touchless interaction currently is. Another approach to the pinch gesture consists of detecting touching thumb and index fingers, through image segmentation [Wil06]. Here no particular finger tracking occurs, but the (practical) result can be considered the same, although this gesture is different from the one described in the previous method, concerning the way that each one is recognized.

So, there is a lot of information one can withdraw, from a scene, through computer vision alone. This information, take for example the finger point data, could be fed to the solution, as input data. Another example is the TUIO protocol [KBBC05], that provides a specification for applications built for table-top tangible user interfaces, like the reacTable (figure 2.7). Through this protocol, information retrieved by the camera is sent via OSC[3]

[2] A pinch gesture is a gesture performed with the index and thumb fingers, by moving those fingers apart or together.

[3] OSC is a communication protocol based on UDP, it provides message formatting and encapsulation, among other features

to the applications. In order to expand the solution's capabilities, TUIO support could also be included.

2.2.5 Depth Cameras

Depth cameras can be described as regular cameras that provide information about the distance, from the sensor to each pixel of the provided image. This is commonly called a *depth map*.

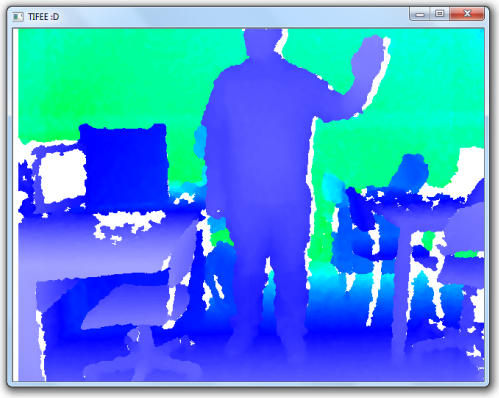

Figure 2.11: Depth map provided by the MS Kinect device. It is possible to see the color gradient used to mark each pixel based on the distance to the sensor.

There are not many commercially available solutions and relatively to those that are indeed available on the mass market, the information about how do they function is not provided. Still, some will be presented next.

- **MS Kinect** is the best known depth camera, considering its commercial success [Bri11]. It is composed of an RGB camera, an IR camera, an IR emitter and a four microphone array. Through the IR emitter, a pattern composed by dots is emitted on the environment, the IR camera captures these dots and, through image processing the depth map is built calculating the distortion of the pattern. This technique can be considered the core of this device, since it allows to build the depth map, that in turn, enables the software capability of skeletal tracking. Kinect also features an accelerometer, a LED and a motor, to adjust its vertical position. [IFia, Wikf]

Figure 2.12: MS Kinect [Micc].

- **ASUS Xtion Pro** is available in two different versions, the Xtion Pro (figure 2.13, on the left), which features the IR camera and IR emitter only, and Xtion Pro Live (figure 2.13, on the right), that, besides the IR camera and emitter, boasts an RGB camera [ASU].

Figure 2.13: The ASUS Xtion Pro (left) and Pro Live (right) [ASU].

Most of the projects developed using MS Kinect's capabilities lack any documentation, besides the common video showing off the application. These videos often portrait very creative uses, taking advantage of Kinect's capabilities. One of these usages consists in using the depth sensing capability to perceive when a finger, or multiple fingers, touch a certain surface, easily providing a function very similar to multi-touch [Wil10].

There are other similar devices that provide similar capabilities through different approaches, however. In the paper "Identifying Hand Configurations with Low Resolution Depth Sensor Data", Sameer Shariff and Ashish Kulkarni describe how they used a Swiss Ranger 4000 to help detaching a hand from a given image [SK09]. The Swiss Ranger is a time-of-flight camera. This type of device functions based on calculating how much time a signal (a wave) takes to travel from the camera to a certain object, thus providing depth data [Ima, Wikj, Wikk].

Using computer vision in order to process a depth map can also bring interesting input data to the solution, as it is easier to separate distinct objects or areas in the scene (e.g. separating the user from the background).

2.2.6 Synergizing

There are other forms of interacting with a computer, that were not described so far. Some of them do not seem well suited for performing HCI in a public context or are simply not that well developed. Yet, some are worth mentioning.

An example of this synergy between different interaction techniques can be seen in the paper "Sensor Synaesthesia: Touch in Motion, and Motion in Touch" [HS11], where a smartphone's sensors are used together with its touchscreen to enhance gestures functions, as an example, a picture deletion method is explained, where the user selects a picture, on the screen, by pressing on it and then shakes the phone to delete it (see figure 2.14).

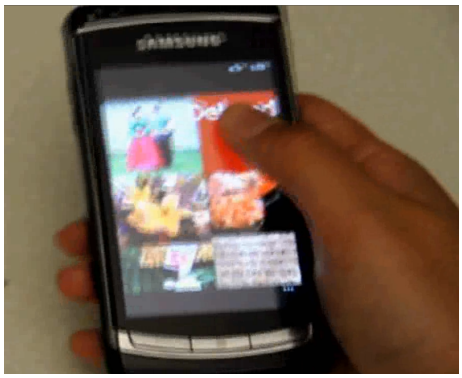

Figure 2.14: “Sensor Synaesthesia” - The user selects a picture and shakes the phone in order to delete it [HS11].

Also, there is the possibility to have a smartphone, equipped with accelerometers, working in a similar manner to a motion controller. As lot of today’s smartphones are also equipped with accelerometers and different connectivity capabilities, data from these devices can easily be transmitted across the network, from the smartphone to a computer.

2.3 Touchless Interaction, a focus on Gesture-based Interaction

Touchless interaction can encompass distinct practices of human-computer interaction that require no touch between the user and the machine. These practices include voice commands, speech recognition, gesture recognition, hand and finger tracking, gaze tracking, facial tracking, body tracking and, some might consider, brain-computer interfacing. Nevertheless, the gist of this MSc thesis is on gesture recognition and, to a smaller extent, body tracking.

Gesture recognition is actually a very generic term too, since it can be associated with so many interaction contexts, from touch to motion controllers and, of course, touchless too. One should note that in what touchless gesture recognition concerns, there are different types of gestures to be recognized: hand gestures, or hand poses, such as the poses that a certain person performs in order to communicate via a sign language, and gestures that implicate more movement of the user’s hand(s). It is important to differentiate between poses and gestures in this context since in the research field they are usually referred to as just gestures.

There have been several approaches to this issue, i.e, on how to effectively recognize a gesture, using a motion controller, tracking a finger or a hand or using touch. Gesture recognition can be hard coded as in [Mal03, Wil06] or it can be extensible, as intended, allowing to easily add new gestures.

There are a number of ways currently used to train the system to recognize gestures. One of those methods, is template matching, that works, as the name describes, by taking

samples of the features, of the gesture to be recognized, and matching them again previously recorded samples. There are other approaches, such as statistical classification and structural matching. Statistical classification-based approaches involve identifying a new gesture through the probability of it belonging to a set of gestures previously recorded [Wikh] - Bayesian Networks, Hidden Markov Models, Artificial Neural Networks and Support Vector Machines are well-known examples used in such approach. The structural matching technique works by matching sub-patterns of the major pattern (gesture). [Woo00]

To gesture-based interaction not only concerns how the gestures should be represented, in a computer system, but also which gestures are more suited for interaction. This research topic has been growing in interest lately, due to the success of devices such as the Wii Remote, PlayStation Move and MS Kinect. Still, there is no standard or common practice in touchless interaction, i.e., there is no set of gestures that are considered universal, emerging the need to study which gestures should be mandatory [GJM11]. There are even, as expected, very distinct ways of representing a gesture, such as the hand position, the fingertips position and orientation or the quantity of movement captured in the scene. This is important when dealing with different methods of recognizing a gesture, since at performance time, there is the need to properly capture all these features so that the recognition process is the least possibly affected by the lack of those same features [WH99].

2.3.1 Brief Review of Template Matching Gesture Recognition

The concept of template matching applied to gesture recognition can be easily understood, still, its implementation presents a few challenging issues. In this subsection an overview of this gesture recognition technique will be provided, referring some solutions in particular. There are some advantages to this approach at gesture recognition, such as the ease of development, since it does not require advanced knowledge in computer science fields, like Artificial Intelligence or even algebra and advanced calculus. It can easily be integrated in a UI prototype, given the ease of development, so it is in the grasp of people not directly related with software development.

The template matching technique gets its name from the tasks it performs in order to recognize a gesture. A template matching gesture recognizer attempts to recognize a gesture by comparing a candidate gesture to a series of gesture templates, scoring each one based on how similar they are and indicating the best match. So, in order to apply template matching, a gesture must be perceived as a path, or vector, of points, either two or three dimensional.

Four different template matching mechanisms were studied ($1, Protractor, $3 and Protractor3D) each having its distinctive characteristics but, at the same time, some others

in common. Since all of these matching procedures work with the same type of data - paths of points (see figure 2.15) - and two of them are extensions from the other two they all have in common some steps.

Figure 2.15: The “star” gesture, as a path of points. The raw gesture data (as performed) on the left, after resampling (N=32) on the middle and after resampling (N=64) on the right [WWL07].

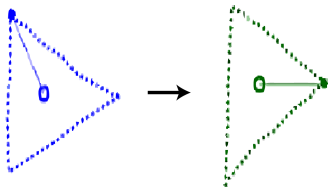

Figure 2.16: A “triangle” gesture, after rotation, on the right [WWL07].

It is easy to understand that a k-nearest-neighbor approach[Li10, Dud76] is the heart of template matching, considering that the comparison between two different gestures (candidate and template) consists essentially in comparing how far apart each equivalent point, from each gesture, is. It is also easy to see that some problems arise from this approach, namely the high probability of a candidate gesture having a different number of points than the template gestures and/or being rotated relatively to these template gestures. A brief description on how these issues are addressed, in each procedure, follows.

2.3.1.1 $1

This procedure has become a reference in what template matching gesture recognition concerns. In spite of recognizing only two dimensional gestures, it served as a foundation for the other two procedures presented below, as well as others, such as a multi-stroke (or multi-path) gesture recognizer, $N[AW10]. $1 has also been implemented in many programming languages, such as Java, C++, Objective-C, Python and ActionScript [WWL].

$1 was developed with a list of eight goals in mind[WWL07], including rotation invariance [4] and position invariance [5], being easily implemented with few lines of code, allow adding template gestures (with only one gesture example), coping with the difference of points in a gesture sample, or candidate, and a gesture template, and being fast enough for interaction purposes.

From this set of goals, a four-step procedure was created:

[4] Rotation invariance is a gesture recognition property that refers to the capability of recognizing the same gesture independent of its rotation in the two or three dimensional space.

[5] Position invariance refers to the capability of recognizing the same gesture independent of where it was performed in the two or three dimensional space.

1. **Resampling the point path**: as stated before, one of the issues with template matching is comparing two gestures whose number of points are different. Resampling a path consists of transforming a given gesture path with K points into one with a predefined size of N points. Wobbrock et. al state that a N of 64 points is perfectly adequate, as well as any $32 \leq N \leq 256$. Resampling a path is accomplished through interpolation, starting at the first point and iterating through the path. When the distance from a certain point to the next one exceeds an increment value ($I = \frac{pathLength}{N-1}$) a new point is created, via interpolation. Figure 2.15 provides an example of this step.

2. **Rotating the gesture**: this step is not only crucial, given that rotation invariance is a necessity, but it is also the step that usually poses the most difficulties in template matching. $1 addresses this issue by first finding the gesture's "indicative angle", the angle formed between the centroid of the gesture and its first point. The gesture is then rotated so that this angle coincides with the XX axis. Figure 2.16 provides an example of this step.

3. **Scale and translate**: one could state that $1 is also size invariant, since it recognizes a given gesture despite its size (in terms of area). Take for example a "triangle" gesture, it can be performed in several different ways, each having a different area, but it is still a "triangle". In order to attain this, $1 scales the gesture into a square, based on the gesture's bounding box[6]. As mentioned before, $1 is also position invariant, that is why the gesture is also translated so its centroid coincides with the origin of the coordinate system (0,0).

4. **Recognize the gesture**: now that the gesture is normalized (resampled, rotated, scaled and translated) it is time to compare it against all the stored gesture templates, in order to find out which is more similar. This is accomplished by finding the average distance between corresponding points (summation of the euclidean distances between corresponding points, divided by the number of points, N) and then a score (ranging from 0 to 1) is calculated.

$1 has some limitations however. Take for instance the third step, when the gesture is scaled. Unidimensional gestures can be heavily distorted due to the scale to square step, making it impossible for $1 to recognize those type of gestures, as is. This scaling step also strips this procedure of the possibility of distinguishing from ovals and circumferences, as well as squares from rectangles.

[6] A box that fully comprises the gesture

2.3.1.2 Protractor

Protractor is very similar to $1 [Li10], as it is based on the same principles. However there are several key differences on how Protractor performs its template matching technique:

1. **Resampling the point path**: this step is performed in the same way that $1's is, except that it normally uses a N valuing 16 points. Unlike $1, that treats a gesture as a path of points, Protractor stores a gesture in a single vector, where the first element is the X coordinate of the first point, the second element is the Y coordinate of the first point and so on:

$$g = [x_0, y_0, x_1, y_1, ...]$$

2. **Translating and Rotating the Gesture**: in this step the gesture is translated, through its centroid, to the origin of the coordinate system. Unlike $1 the gesture is not scaled, to avoid distorting it. The rotation procedure is also different from $1: the gesture is rotated (by its indicative angle, like $1) to the closest of one of eight base orientations (see figure 2.17). The vector is also normalized, by having each element divided by the vector magnitude, that is why Protractor can still recognize a gesture despite of its size without the need of scaling it;

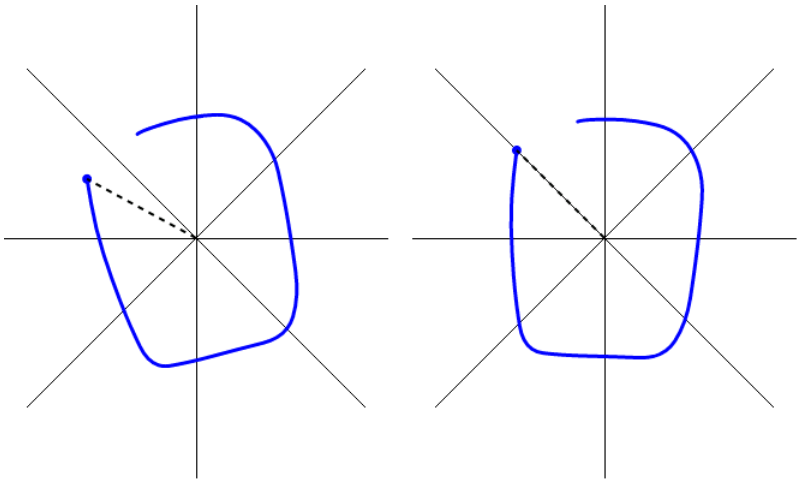

Figure 2.17: An example of Protractor's gesture base rotation [Li10].

3. **Matching**: matching between the candidate gestures and the set of templates occurs by calculating the cosine distance [7] between the candidate gesture and each of the templates stored (unlike $1 that employs a euclidean distance comparison).

$$g = [x_0^g, y_0^g, x_1^g, y_1^g, ...]$$

$$t = [x_0^t, y_0^t, x_1^t, y_1^t, ...]$$

[7] The cosine distance between two vectors determines, more or less, if those two vectors are pointing in the same direction.

A sum of the cosine distances between each pair of equivalent points is calculated. This cosine distance sum serves as a measure to compare the gestures with the templates, the gesture-pair template with the least distance value is marked as the best recognition possible.

2.3.1.3 Protractor3D

Protractor3D extends the Protractor gesture recognizer[KR11] in order to provide three-dimensional gesture recognition. It was developed to be used with 3D accelerometer data (like $3 [KR10]). Protractor3D, when compared to the gesture recognizers presented so far, employs more complex mathematical concepts. In short, it is composed by the following steps:

1. **Resampling**: the same resampling process used in $1 is applied (with an N valuing 32);

2. **Scaling and Translating**: once again translating occurs just like in the previous cases, where the gesture is translated so its centroid coincides with the origin of the coordinate system (0,0,0). Also equivalent to $1 is the scaling step, where the gesture is scaled to fit a cube of fixed size (S=100);

3. **Rotating**: in this step the rotation needed to best align the candidate and template gestures is determined through the use of quaternions (please refer to the original work for details);

4. **Matching**: again, the scoring of each template-candidate pair is based on the euclidean distance (like $1).

2.3.2 Brief Review of Statistical Classification Gesture Recognition

Statistical classification gesture recognition is usually associated with good recognition results. On the other side they require a considerable implementation and training effort [KR10, KR11]. There are several methods of employing statistical classification gesture recognition, such as Artificial Neural Networks, Bayesian Networks and Support Vector Machines. However, in this brief review, a focus on Hidden Markov Models is provided.

An HMM is essentially a collection of states and transition probabilities between them, as well as outputs (which are visible) that are probable to occur at a given state. Each state transition must verify the Markov property [8] and the model's states are hidden. An example of an HMM can be seen in figure 2.18.

[8] According to the Markov property a transition between any given pair of states must only depend on the current state (transitions are "memoryless").

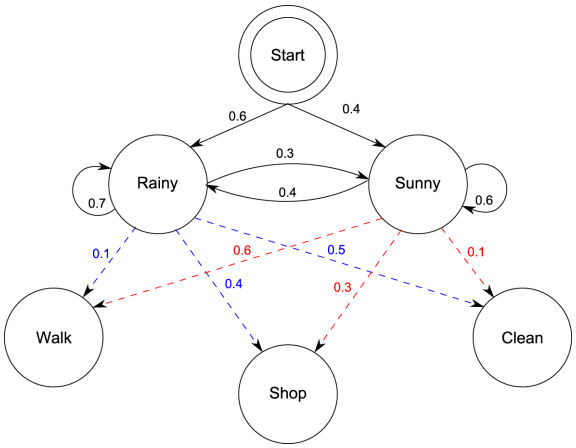

Figure 2.18: An example of an HMM. “Walk”, “Shop” and “Clean” are the hidden states [Wikk].

There are three problems associated with HMMs: **[WH99, Rab89]**

- Evaluation - $P = (O|\lambda)$: which translates to the probability of occurring a certain sequence of observable variables, O, given the model, λ. This is normally solved through the forward-backward algorithm;

- Most Likely State Sequence (decoding problem) - $maxP(S|O,\lambda)$: determining the most probable state sequence, S, given an observation, O, and a model, λ, which is essentially “uncovering” the hidden states. This is can be determined by applying the Viterbi algorithm;

- Training the HMM: optimizing the model parameters to produce a certain output. This is done, usually, through the Baum-Welch algorithm.

So, a given gesture is translated into a certain HMM and that model is trained. When a candidate gesture is performed, there is a given set of visible observations (outputs) that need to be matched with the corresponding model. [YX94]

These observations that occur upon the gesture performance are usually called features, or gesture features, since they are used to represent a gesture. With a computer vision-based approach, these features can be extracted from the image frames of the gesture performance. In [RKE98] some of these features are taken from a given frame and its adjacent frame, building a new frame based on subtracting each corresponding pixel value in these two frames. After applying a simple piecewise image noise reduction technique, a frame that encodes the intensity of motion is obtained. These new frames are used for feature extraction, producing features such as the center of gravity, the average of the intensity values. At the end of this extraction process, a total of seven features are extracted. These features are used to train the HMM as well as to recognize a certain candidate gesture, whose performance has to undergo the same process.

2.3.3 Frameworks and Similar Projects

Regarding touchless interaction, and considering this MSc thesis' focus on gesture recognition and the usage of MS Kinect, there are some frameworks, libraries and projects that deserve taking a close look at.

2.3.3.1 OpenNI & NITE

The bulk of the projects being developed with MS Kinect, and all of them that employ ASUS Xtion Pro, take advantage of the OpenNI framework. OpenNI, as in Open Natural Interaction, is a "multi-language, cross-platform framework that defines APIs for writing applications utilizing Natural Interaction" [Pri11].

OpenNI organization, established in November 2010 and responsible for the framework, is composed of a few corporations: PrimeSense (that develops the framework and produces the IR sensors found in MS Kinect and ASUS Xtion Pro), Willow Garage (supports the development of ROS, PCL and OpenCV libraries) [Wilb], Side-Kick (develops motion-controlled games), ASUS (produces hardware, such as the Xtion Pro) and appside (an online marketplace for gesture-controlled applications). This non-profit organization promotes natural interaction [Pria] and has the purpose of accelerating the introduction of NI applications in the market.

The OpenNI framework, along with the proper drivers, operates as an interface between the hardware (MS Kinect, ASUS Xtion Pro) and the middleware logic, enabling developers to easily access the devices' raw sensor data (such as a depth map or an RGB image) at the same time that the middleware components can produce more meaningful information from that same data. This organizational structure can be seen in figure 2.19.

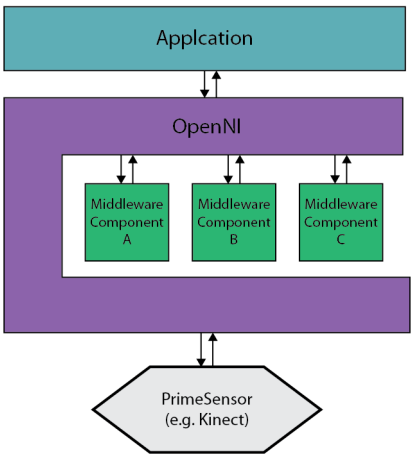

Figure 2.19: OpenNI interfacing between the hardware and the application.

OpenNI is divided into two series of components, or modules, the hardware modules and the middleware modules. The middleware modules provide some interesting functionalities to those developing in touchless interaction, like:

These middleware facilities, which interfacing with the hardware is provided by OpenNI, are offered by NITE, also produced by PrimeSense. NITE "includes both computer vision algorithms that enable identifying users and tracking their movements, as well as framework API for implementing Natural-Interaction UI controls that are based on user gestures" [Prib].

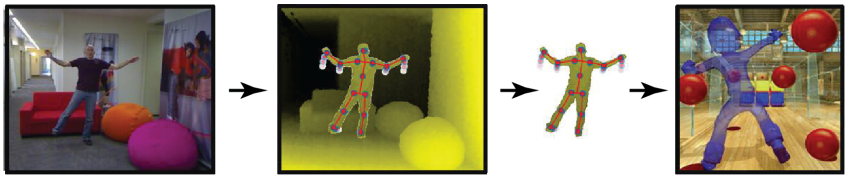

Figure 2.20: An example of OpenNI and NITE at use [Prib].

So NITE is essentially a middleware implementation that provides the following functionalities:

- Body analysis, provides body related information (skeletal tracking[9]);
- Hand point [10] analysis, processes raw sensory data, returning the location of a hand point as a result;
- Gesture detection, allowing to detect a few gestures,

and is divided into two layers:

- Algorithms: this is the lower layer, performs the groundwork for the layer above and any application drawing on NITE's features (mentioned above);
- Controls: based exclusively on hand point analysis, offers facilities to control an application based on that analysis as well as gesture recognition[11] and a small set of touchless UI elements[12].

[9] In skeletal tracking a set of joints, corresponding to specific regions of the user's body, is assigned to that user. For this to happen it is usually necessary that the user performs a certain pose, called the calibration pose.

[10] The term hand point refers to a point that is assigned to a hand, after it performs a certain gesture, known as the focus gesture. Unless there occurs occlusion of that hand for a considerable period of time, the hand point shall remain attached to the hand until the user, or the hand, can't be detected by the device or the application terminates.

[11] The gesture recognition provided by NITE is based only a small set of gestures: waving hand, swipe (up, down, left, right), push, circle.

[12] NITE's touchless UI elements are, in essence, one and two dimensional sliders, represented as a rectangle each. In a one dimensional slider the hand point position, relative to the slider itself, is calculated, producing a single value, whereas in a two dimensional slider that position produces two values (hand position relative to the slider's XY axis).

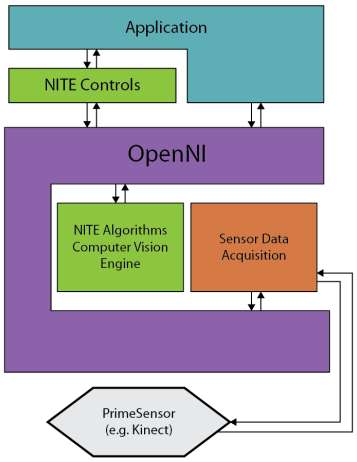

Figure 2.21: The NITE middleware and its interfacing with OpenNI and the application.

2.3.3.2 Microsoft Kinect SDK

In spite of being the corporation that owns and distributes MS Kinect, Microsoft was one of the last (if not the last) to produce an SDK for Kinect. Besides being the official Kinect SDK, there is one advantage to this framework, it is the only SDK that allows direct access to Kinect's array of microphones.

Furthermore, this SDK offers skeletal tracking as well as access to the depth map and RGB camera. streams. Other features, like gesture and posture (skeleton poses) recognition can be added through Kinect Toolbox, a set of tools built for developing with MS Kinect SDK.[Kin] Kinect Toolbox's gesture recognition works only on two dimensions, but allows for adding new gesture templates to the gesture database.

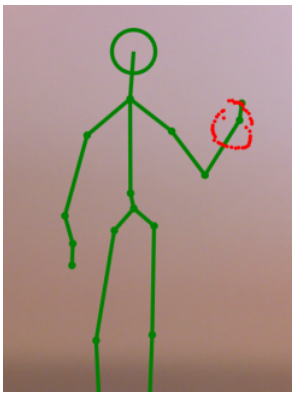

Figure 2.22: An example of the Kinect SDK at work, showing a "stick figure" [Kin].

2.3.3.3 CLNUI

CLNUI was one of the first Kinect platforms released, after about two months of the MS Kinect launch (November 4th, 2010). Since that first version, only one update was issued,

two days later. However, it boasted some interesting features, including access to the depth map and RGB camera streams, accelerometer data, motor and LED control.[Lab]

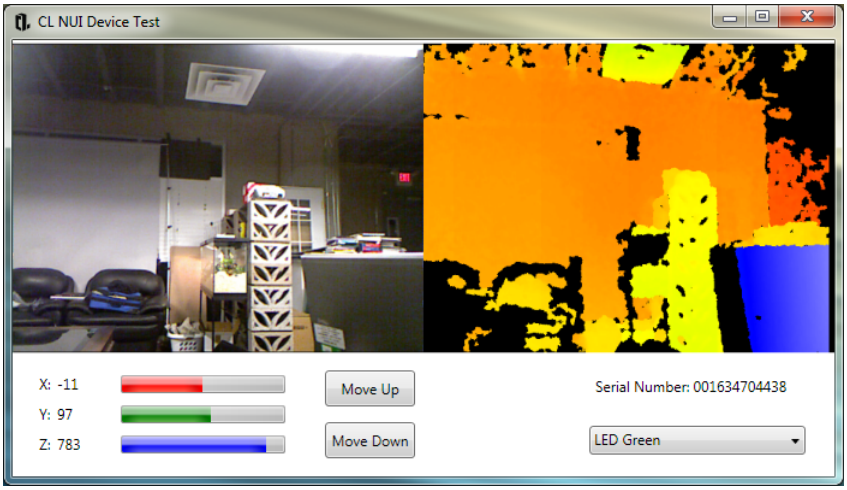

Figure 2.23: The CLNUI test application [Lab].

2.3.3.4 OpenKinect

OpenKinect, as the name suggests, is an open source project, with the goal of being cross-platform, that is, running on MS Windows, Linux and Mac OS. It allows access to the depth map and RGB camera streams, as well as motor and LED control. It boasts a series of wrappers, making it possible to develop in several distinct programming languages, such as C++, Java, C#, Python and ActionScript [Ope].

2.3.3.5 Comparing and Choosing

Before starting the development of this MSc Thesis, there was the need to choose which frameworks, or libraries, to use, when working with MS Kinect. The following table establishes a comparison between them.[Pria, Ope, Lab, Mica][13]

OpenNI and NITE were chosen among the other frameworks since they provide skeletal and hand tracking and allow the development of commercial applications.

2.3.3.6 Other projects

Several projects that harness Kinect's capabilities make use of a toolkit named FAAST [SLR+], that detects certain user poses, through the skeletal tracking capability and allows simulating OS events when those poses are detected. Other tool, enabling developers too easily take advantage of Kinect enabled skeletal tracking is the OSCeleton, that tracks users' body joints and sends this data through the network, using the OSC protocol [Sen].

[13] Although MS Kinect SDK does not provide hand point tracking per se, i.e. like NITE provides a hand point stream, it is possible to track a skeleton joint corresponding to the user's hands.

	OpenNI + NITE	OpenKinect	CLNUI	MS Kinect SDK
OS	Windows, Linux, MacOS	Windows, Linux, MacOS	Windows 7	Windows 7
Programming Languages	C, C++ (Java)	C++ (Python, Actionscript, C#, Java, Javascript, Lisp)	C# (Java)	C++, C#
Depth Map Access	YES	YES	YES	YES
RGB Camera Access	YES	YES	YES	YES
Microphone Array Access	NO	NO	NO	YES
LED Control	NO	YES	YES	NO
Motor Control	NO	YES	YES	NO
Accelerometer Access	NO	YES	YES	NO
Skeletal Tracking	YES	NO	NO	YES
Hand Tracking	YES	NO	NO	NO
Software License	GNU LGPL	GNU LGPL	undefined	Noncommercial
Device Compatibility	MS Kinect, ASUS Wavi Xtion	MS Kinect	MS Kinect	MS Kinect

Table 2.1: Framework Comparison

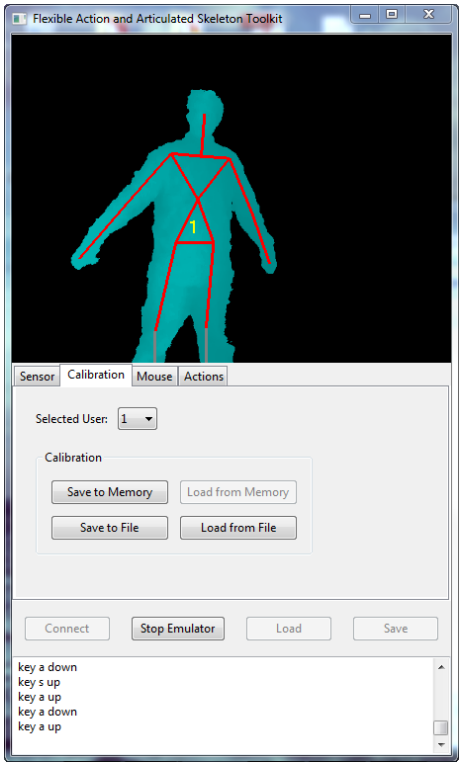

Figure 2.24: A *snapshot* of the FAAST toolkit. The user silhouette is clearly visible, as well as the skeletal tracking feature.

As a result of having quite a few libraries, SDKs or toolkits to work with, many developers built countless small applications, publishing videos and screenshots on the web showing off the applications capabilities. Numerous portals dedicated to gather these Kinect hacks began to emerge, but a problem persisted, most of these applications remained properly undocumented, being, most of the times, an online video the only reference to them.

The OpenNI organization put together a web portal - OpenNI Arena - to serve as a showcase where developers can post their applications and receive feedback from users. One of those applications is KineticSpace, that enables the recording and previous recognition of gestures, using the Dynamic Time Warping technique. KineticSpace also allows for sending OSC messages across the network when a certain event occurs [Kin12].

Chapter 3

Proposed Architecture

After consideration upon the goals of the final solution, the first step in its development was, necessarily, to define the software architecture to be implemented. Seeing that the usual interaction process, intended for an application based on the implemented solution, could be split into, at most, three main stages - input, processing, output - the proposed architecture, and the data flow, had to be, surely, strongly influenced by these three stages, as portrayed in figure 3.1. Additionally, there was also the need to meet other requirements, that could also influence the software architecture. So, the proposed architecture should allow for the solution to be expandable.

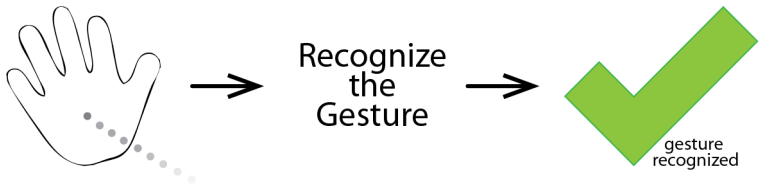

Figure 3.1: An illustration of an interaction process' three stages.

3.1 Three Stages

As stated earlier, three distinct stages were immediately identified, bearing in mind the hardware used as well as the interaction process itself and all the processing needed in between. So, resulting from a first study approach on the subject, was the following top-level view of the system's architecture:

Figure 3.2: The Three Stages, connected.

The three main stages represent each, the three distinct types of components in the architecture: input, middleware and output components. Besides representing the architecture, these three stages also refer, from a top-level perspective, to the flow of information, running a project based on this proposed solution.

Data flows from the first stage to the next one, being possible to skip one stage (given that there are only three). This means that input data can be directly fed to components from the output stage, skipping the middleware phase. Although, the input stage is mandatory, that is, for the application to work properly, there always has to be, data retrieved from this input phase. These data flow along these three distinct stages forms what shall be called from now on, a pipeline.

Although only one hardware device is used specifically for input data gathering, it is possible to request, from this device, along with the proper software, distinct types of data. These data is processed to some degree, still in the first stage - the input stage - providing more refined information, or data, to the next phases - either the middleware or the output stages.

3.1.1 Input

As explained earlier, the Input stage is the first stage and an essential one, being the application in execution in solo mode or just using the solution as a framework.

As the name suggests and as it was mentioned earlier, during this first stage, the input data is gathered from the outside world, using proper hardware. In this phase, all input data to be used, or input data requirements, should be defined so it will be available during the entire execution of the pipeline.

Besides retrieving data from the hardware, in this stage the raw data - since it is still not processed at all in the running pipeline - can and must be processed according to what is needed for the next stages, either the middleware stage or the output one. When these data processing is finished, the new already processed data can be fed to the next stages, producing more data depending on its destination stage.

Looking at a concrete example such as a two-dimensional gesture recognition mechanism, in the input stage a three-dimensional hand point would be captured. This would be the raw data, needing processing in order to be converted to a two-dimensional hand

point, since the gesture recognition mechanism only supports two-dimensional gestures. This can be seen in figure 3.3.

Figure 3.3: The Input stage.

3.1.2 Middleware

The middleware stage represents the core functionality of the solution since all the further data processing, that enables gesture recognition, has to happen at this point. Although it might contain the core of the functionalities when it comes to gesture recognition, for example, this stage is probably the only one that can be skipped, in a practical situation, having data flowing directly from the input stage to the output one. Imagine a simple example, where one would only need hand point data passed to another application via a network message. The hand point position could be transferred from the input stage (where it is made available) directly to the output stage (see figure 3.4).

Figure 3.4: Skipping the Middleware stage.

Continuing with the gesture recognition example, the middleware stage would receive the hand point data from the previous stage, already converted into a two-dimensional space. Using these data the gesture recognizer would attempt to recognize a gesture and transmit the results to the next stage, as portrayed in figure 3.5.

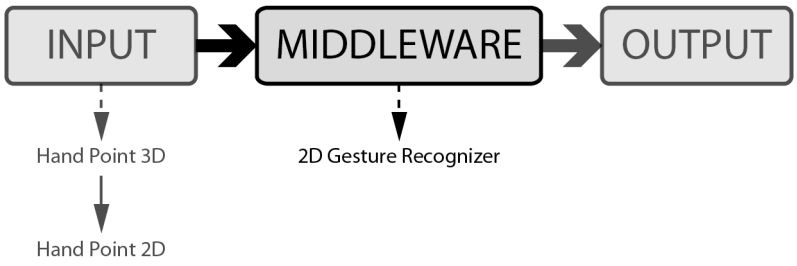

Figure 3.5: The Middleware stage.

3.1.3 Output

The final stage is the output one. The purpose of this stage is to present, somehow, all the work that was done in the previous stages. It is the stage where data comes and does not get transferred again within the pipeline, but only out.

Taking a look at the same practical example of the gesture recognizer, a simple way of presenting the results could be just printing to the console the results of the gesture recognition attempt.

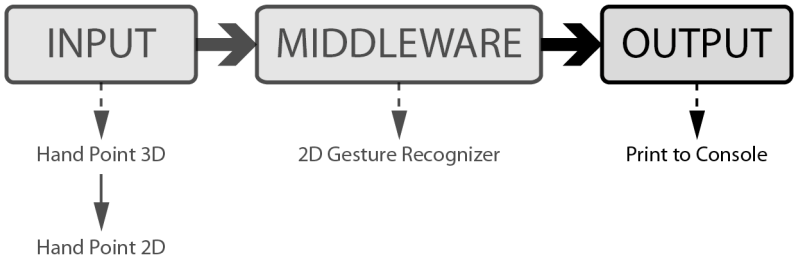

Figure 3.6: The Output stage.

3.2 Filter Graph Approach

Filter graphs can be considered of common use in multimedia [1] processing, allowing to decompose complex tasks into smaller, simpler ones. These tasks usually take the form of filters. Each filter typically performs only one task. Patching together a series of filters forms a directed (filter) graph, that represents the data flow, since the data flows from one filter to the next, usually undergoing some data processing, resulting in new data to be processed by the next filter. Commonly, the last filter is responsible for the render process, that is, display, somehow, the work that was done by all the previous filters. Filters are, fundamentally, data conversion entities, transforming input in output data. One should

[1] GStreamer and Microsoft's DirectShow, that have this filter graph approach as its core, are widespread tools used in media processing.

see the tasks that a given filter performs, essentially as data conversion, since that those tasks are not the focus in this kind of approach.

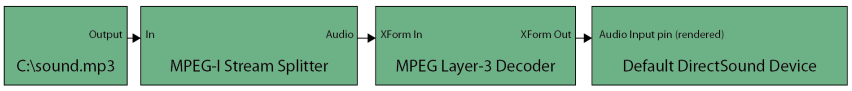

Figure 3.7: A DirectShow filter graph.

This approach fits not only the three stages architecture described in the previous section but also some of the requirements listed earlier in this document. So, in order to implement the three stages filter-based architecture, two different implementations of the filter graph approach were studied: GStreamer and DirectShow. OpenNI and NITE are also discussed, in spite of not being based, architecturally, on filters, the approach is still considerably similar. A brief presentation on each one follows.

3.2.1 GStreamer

According to the GStreamer's official webpage, "GStreamer is a library for constructing graphs of media-handling components." [GSt] Being a graph-based library (a filter is a graph node), GStreamer can be very versatile, allowing for the developer to build an arbitrary pipeline by connecting together each filter in order to accomplish a given task. [TBW$^+$] In GStreamer, a filter is known as an element, so a pipeline is a collection of connected elements. A filter, or element, is provided by a plugin. A more comprehensive term for an element collection, in GStreamer, is the term "bin". A pipeline is a particular type of bin, in which all its elements can be executed.

Each element (a filter) can be connected to other elements through its pads. A pad is essentially a port through which data flows. A pad has an associated data type, restricting which type of data can flow through it. Only two pads that share the same data type can be connected. Pads can differ in functionality, since there are virtually two types of pads in GStreamer, the sink pads - through which an element receives input data - and source pads - through which an element transmits data to other elements. It is easy to think of it as the output pad of one element - lets call it element A - being the data source of the next element (B), and the input pad of element B as the pad where element A "sinks" its data.

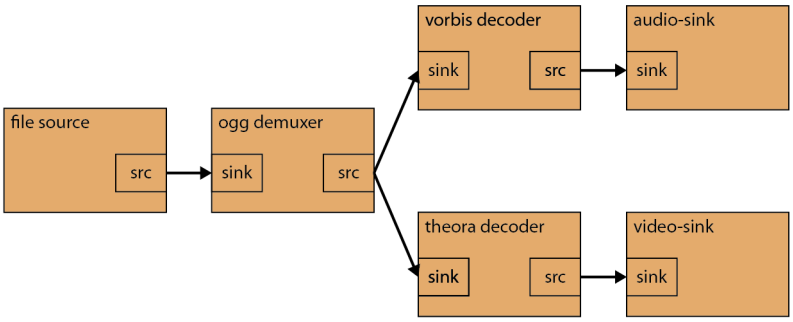

Figure 3.8: A GStreamer pipeline, composed of six filters.

As easily perceived in figure 3.8, building an application based on GStreamer consists essentially of connecting (and developing if necessary) the right filters. The media player application example is probably the most practical example to present. A *file-source* filter reads data from a video file (Ogg format) and transmits it to a *ogg-demuxer* filter, which in turn separates the sound and video data and transmits both to distinct filters (note that the *ogg-demuxer* has a sink pad and two source pads). The *vorbis-decoder* receives the sound data and has to guarantee, somehow, that this sound data is played, by transmitting the data to the *audio-sink* (lets think of it as a sound card). An analogous process happens with the video data, this time regarding the *theora-decoder* and *video-sink* filters. All these connections between filters are represented by a proper "link" object, that links pads.

It is possible to divide GStreamer, virtually, into two distinct parts. The first part, the core, consists of a series of core functionalities, such as the graph-based structure, the pipeline architecture, the API for developing applications or plugins, data passing and others. The second part comprises a collection of plugins divided intro three different sections - the good, the bad and the ugly - according to the quality of each plugin as well as its proneness to cause distribution problems [2].

Since GStreamer is, essentially, a framework for building other applications on top of it, it is likely that a need for pipeline-application communication arises. In this framework, the messaging system works as displayed in figure 3.9.

[2]Since GStreamer is a cross-platform framework, some of the plugins included might not work properly in one, or more, of the operating systems to which GStreamer is available (Linux/Unix, MacOS X, Windows).

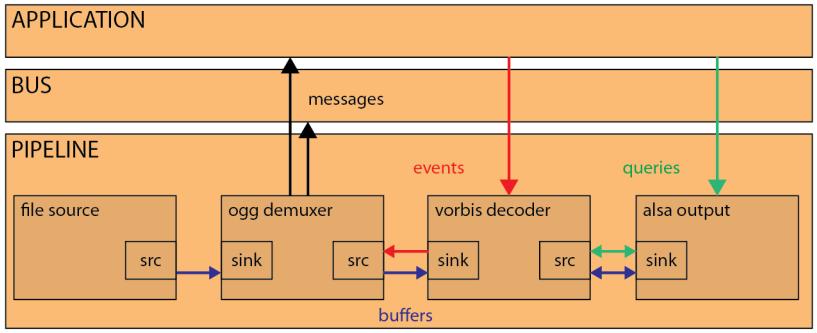

Figure 3.9: A GStreamer pipeline, with its communication flows represented.

Buffers are objects that contain streaming data and always travel from source to sink pads. Events can be sent between elements/filters or from the application to an element. Queries are used by the application, or by other elements, to query a certain element about something about its task. Messages are posted by elements on the message bus, and are commonly used to post error information.

3.2.2 DirectShow

DirectShow is a multimedia framework, part of the DirectX SDK. It relies, whenever possible, on DirectDraw - to render data to the graphics card(s) - and on DirectSound - to render data to the sound card(s). Microsoft built DirectShow in order to facilitate the development of applications that need to process different types of media. Some of the issues that DirectShow addresses include providing a solution that is able to process different media data formats (sound or video) and interface different sources of data (file system, hardware, network) [Micb].

To overcome these challenges (and others) DirectShow implements a modular architecture, based on filters, each, like described above, having its purpose. As in the GStreamer approach, the filter is the application's building block. Concerning the connections between filters, DirectShow provides an interface for the effect, called "pin". A pin holds information about the media types that can traverse it, the data flow direction, which filter owns it and which filter it is connected to.

In DirectShow's architecture filters are divided into three distinct sections:

- Source filters, responsible for gathering input data (from a file, a network stream, for example);
- Transform filters, that apply transformations on the data received from source filters (overlaying text, changing colors, for example);
- Rendering filters, that render data (using DirectDraw to draw an image onscreen or DirectSound to transmit sound to the sound card, for example).

A collection of filters is called a filter graph, and filters have states, being possible for a filter to be running, stopped or paused. To contain filters, or a filter graph, DirectShow provides a high-level component, called Filter Graph Manager. As the name suggests, this component manages the filters, serving this purpose by providing means of adding, removing and connecting the filters. The filter graph manager also coordinates filters' state changes, handles synchronization as well as event notification (DirectShow's equivalent of GStreamer's messages). This process is shown in the figure 3.10.

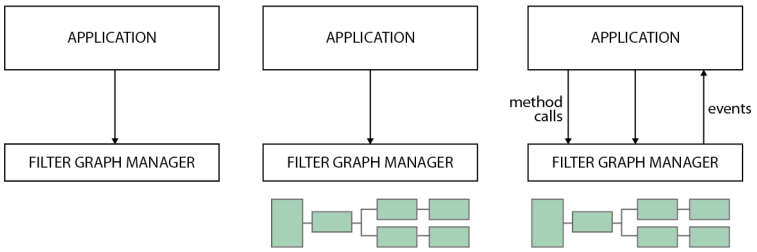

Figure 3.10: An example of an application built on top of DirectShow. It uses the Filter Graph Manager to build a filter graph and to control it.

An application taking advantage of DirectShow, must work on top of it, using the filter graph manager as an interface between the said application and the filters.

3.2.3 OpenNI and NITE

OpenNI and NITE have been mentioned and briefly introduced before, in the second chapter, however, and since these tools represent a very important part of the solution, due to their functionalities and capabilities, this section will aim at describe the influence they had in the conception of the proposed architecture.

OpenNI is based on production nodes. Essentially, a production node is an architectural element, or component, that produces some kind of data. A production node is an essential element when it comes to the interface between OpenNI and the middleware, since it does not only produce data but it also makes that data available for any other node or middleware. In order to overcome any differences in the hardware devices that can be used along with the OpenNI framework, it incorporates the concept of capabilities. A capability is an extra feature provided by a certain device, such as the ability to flip the RGB image or the depth map captured (known as the mirror capability). So, production nodes are elements that perform a specific interfacing task, enabling the middleware to access the data it needs to produce more "meaningful" results.

Concerning NITE, it is mandatory to explain two concepts, NITE's session management and its (point) controls. To take advantage of NITE's capabilities, namely the point

controls, there is the need for the user to be in session. There are three different possible session states: not in session, in session and quick refocus. The transitions between different session states is explained next and can be seen in figure 3.11.

For the user to start a session he/she needs to perform a focus gesture [3] and once he does, the session is started ("in session") and the hand that performed the focus gesture starts being tracked, having the corresponding hand point made available. When the hand point is lost (due to occlusion, as mentioned earlier) the session state changes to "quick refocus", and it only leaves that state if a quick refocus gesture [4] is performed (then the state changes back to "in session" and the user's hand that performed the gesture continues being tracked) or, after a pre-determined period of time, the session is lost and so is the hand point (session changes to "not in session").

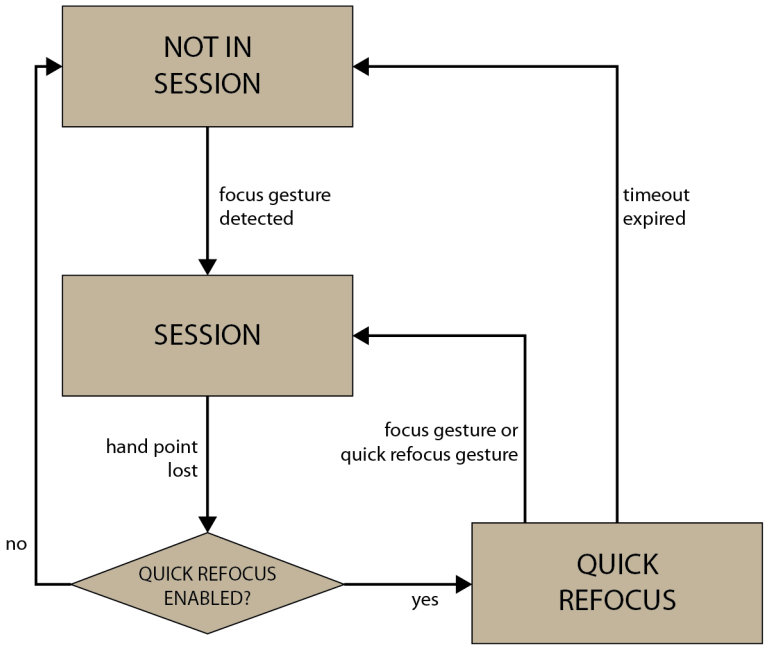

Figure 3.11: NITE's session flow.

NITE's controls, in turn, receive a stream of these hand points and attempt to translate these data into more meaningful information. One of these controls might be a gesture detector, such as the push detector. The push detector receives a stream of hand points (one hand point per frame, for example) and calculates the speed with which the points in the stream move along the Z axis. When this movement is performed in such a manner that it is practically perpendicular to the XY plane, a push gesture is recognized.

[3] A focus gesture is a hand gesture that is recognized by NITE. A few focus gestures are included in the NITE framework, such as the "waving hand" gesture.

[4] The quick refocus gesture is a gesture that should be performed when the hand point is lost, for some reason. This gesture is performed in order to avoid completely losing the current NITE session.

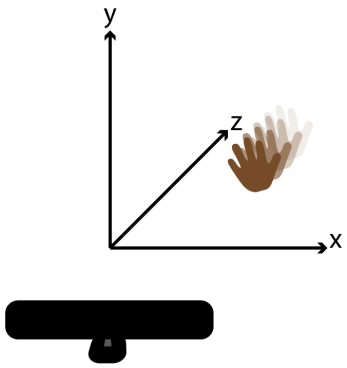

Figure 3.12: A “Push” gesture being performed.

As OpenNI has its modules, NITE has controls. One can think of controls as small modules that perform a specific task, receiving certain input data (usually a hand point stream) and then, after interpreting that input data, return more meaningful data, just like the push detector example mentioned before. Controls can be connected to each other, forming a data flow between them. When connected together, the controls form what is called a NITE Tree (see figure 3.13), which can be viewed as a graph, with each node being a control.

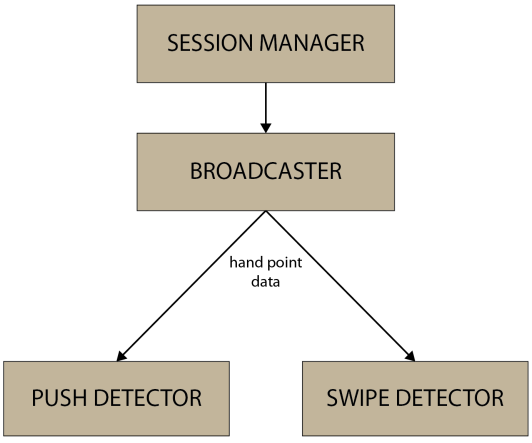

Figure 3.13: A NITE tree.

3.3 TIFEE Approach

Figure 3.14: TIFEE's approach, based on three stages.

Considering the filter approach (namely the DirectShow and GStreamer approaches) along with the set of requirements previously presented, the architecture for the proposed solution was designed heavily based on the filter graph approach. Although NITE's controls usually transmit only hand point data between them, this approach has also drawn some ideas from those controls as well, since these controls are organized in a way much similar to the one that filters (in the filter graph approach) are.

As it is possible to assert from image 3.14, each stage of the proposed architecture is composed of modules. That is, one can look at the three main stages as three distinct modules and, at a closer look, it is possible to check that each stage is constituted of other modules, each "smaller" module specific to the respective stage. This architecture design was achieved after some architectural design iterations, seeking to ensure that the most goals, as possible, were incorporated in the design. In terms of the architecture, each of these bigger modules would be represented by the abstract notion of a stage and each of the smaller ones would be a filter. So, a filter would be placed in a stage and, besides that, all filters would belong to a pipeline. As in the examples mentioned above, a filter would have input data that it would process, producing output data. The data would start flowing from a single filter in the input stage and then it would be constantly transformed, as it passed by each filter, until it reached an output stage filter. This is the normal data flow in the pipeline.

One of the goals of the proposed architecture is modularity. Designing an architecture with that goal in mind demanded that each task, referring to a normal execution of the pipeline, had to be decomposed in tasks as small as possible. These tasks would then be encapsulated in small modules. This would help fulfill architectural modularity, along with the attempt at building each small module as independent from the others as possible. Achieving modularity this way would allow to satisfy the expansibility goal, since a filter could be easily swapped by an equivalent one. So, in order to expand the framework's capabilities, one would only have to develop (at least) a new filter and integrate it into the pipeline.

The filter approach, just like in GStreamer and DirectShow, also helps fulfill the goal of enabling a fast and simple application development environment, using the framework.

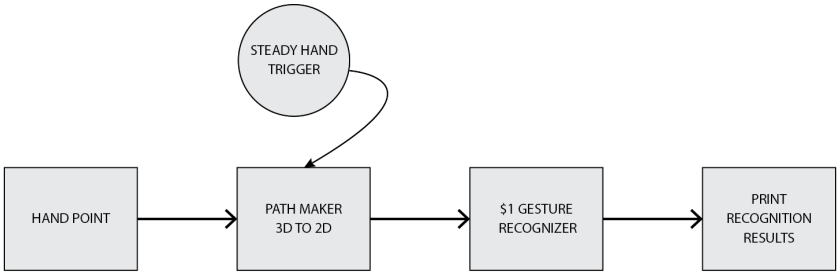

Figure 3.15: A "TIFEE approach" pipeline, with the filters connected.

The developer just needs to work with the pipeline and the filters in order to build an application, adding the filters as needed and connecting them. Much like a filter graph manager (DirectShow) or a NITE tree, the pipeline is the object that holds and manages the filters, hence managing its execution too. This execution management happens continuously, meaning that each filter is updated constantly. However, sometimes, there might be the need to alter, or temper with, the normal execution flow of the pipeline, when a certain event occurs. To cope with this need, a new element was designed, the trigger. A trigger is an element which sole purpose is to force filters to execute, so the pipeline execution can be molded, to a certain extent, if so is necessary.

A pipeline, considering this approach, that contains four filters and a trigger, can be seen in figure 3.15.

In the likes of GStreamer, a messaging system is also provided, so a message pool is contained in the pipeline object. Filters can post messages to the pipeline, in order to enable the application to have access to some type of specific information.

Looking at the architecture from a top-level perspective again it is possible to see that two other goals were taken into account upon the design process. In order to have the solution function as both a stand-alone application and a framework, it was necessary to provide an interface for developers (an API) and an interface for other applications. The framework interface is explained in detail in the next chapter. As for the application interface, it was incorporated in the output section, by providing facilities to allow the framework side to communicate with the said application.

In short, the architecture is composed of three elements:

- The Filter: the basic production unit, recives and transmits data;
- The Trigger: allows forcing filters' execution at a given moment;
- The Pipeline: acts as a container for the filters and triggers and manages their execution.

These elements can be seen in the following figure (3.16):

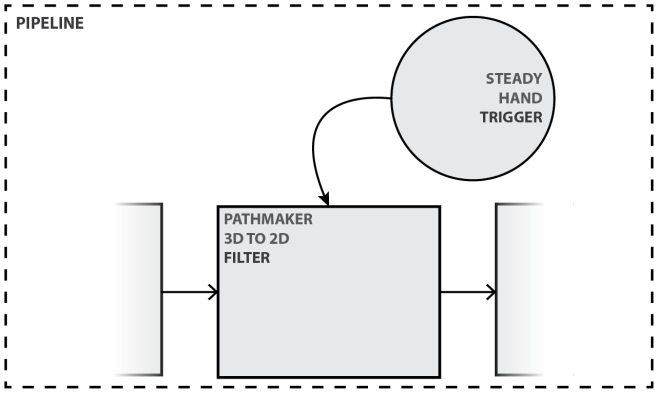

Figure 3.16: A segment of the pipeline presented in figure 3.15.

Chapter 4

Implementation

In the previous chapter it was described the solution's architecture, in this chapter a close look at the implementation process will be undertaken, seeking to explain how the proposed architecture was, not only designed, but implemented. A detailed view of which components make up the architecture will also be provided.

4.1 Architecture

Summing up what was presented and explained so far, on the proposed architecture, we have:

- Filters, that are the elements that implement a certain specific task and transmit data from one to another;
- Triggers, that are sometimes used to control filter's execution, therefore controlling, in a certain way, the pipeline execution too;
- the Pipeline, that holds filters and triggers, and ultimately controls the execution of those filters and the application.

4.1.1 Filters

So, the filter is the fundamental element of the architecture and based on what was stated in the previous chapter, there are some particular needs to take in account concerning a filter. Essentially, a filter has to:

- receive data;
- send data;

- perform its task when so is intended.

Although these just seem three simple requirements, they imply other particularities. First, and to better understand these particularities, there is the need to explain how the pipeline, and more importantly, its execution, were implemented.

The pipeline gathers, or contains, all the filters and triggers and is responsible for their ordered execution. Like in the GStreamer example (section 3.2.1), the filters in a pipeline have a proper execution order, each executing its task in the proper moment, so that, together, they accomplish a bigger, more meaningful task (in the example presented in section 3.2.1 - GStreamer that bigger task is playing back a recorded video, with sound).

4.1.1.1 Updating Filters

So, there was the need to determine how each filter would be updated, assuring the proper flow in the pipeline. There were at least two different implementation possibilities, on one hand a filter, upon finishing its task could notify the following filters, on the other, the pipeline could be responsible for forcing each filter to update, orderly. Practically speaking, both approaches are very similar, and equally valid.

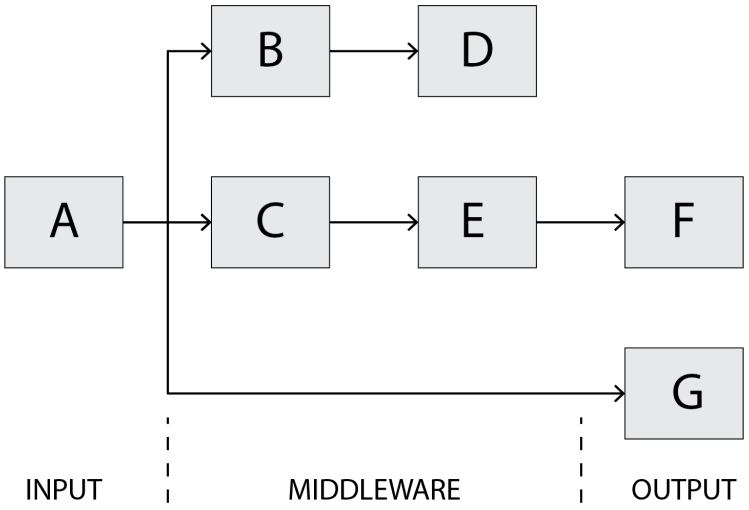

Figure 4.1: A pipeline composed of seven filters.

Taking the above pipeline (figure 4.1) as an example, according to the first approach, the execution order would be A, B, C, G, D, E, F, whereas according to the second approach we would have A, B, C, D, E, F, G. The second approach was implemented due to the resulting execution being in accordance with the top-level architecture diagram (as it is possible to see in figure 4.1), that is, the input filters are updated first, then the middleware ones and later the output filters.

4.1.1.2 Data Transfer between Filters - Data Types

Going back to the filter object, it is clear now that it has to have a way to allow the pipeline to update it and also a form of sending and receiving data. As it needs to send and receive data, it must know where to send the data, so a list of other filters, called the output filters, must be included in each particular filter. As filters can be connected to one another, and as the data gets transferred between them, there is the need to check if the data that is to be transmitted is between each given pair of connected filters is compatible. This happens when the pipeline is being built, that is, when the filters in the pipeline are being connected. Given this, each filter has an input data type - type of data that it expects to receive - and an output data type - type of data that it outputs, as a result of performing its task. Consequently, if a given filter has a certain output data type and another filter has a distinct input data type, they can't be connected. As an example, if filter A has the output data type X, and filter B has the input data type Y, you can't connect A to B in such a way that A would be feeding data to B.

4.1.1.3 Filter's State and its Impact on Update

Although it might not be clear at first, there is also the need to keep a record, in each filter, of the filters that provide input data, called the input filters. This happens because a filter has a state, that is, it can be active or inactive. This can lead to some trouble when executing a pipeline. Take for example a filter in the middleware stage, in the middle of the pipeline, just like in portrayed in figure 4.2, below.

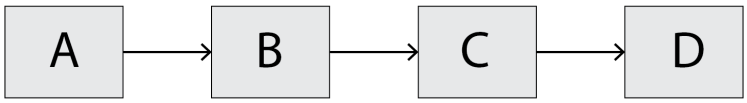

Figure 4.2: A pipeline containing four filters.

If, for some reason, filter B has its state changed to inactive, filters C and D wouldn't be able to carry out their tasks, compromising the whole pipeline, possibly resulting in an application crash, since filters C and D do not have any data to process as to produce theirs. By keeping record of the filters we can avoid this situation at some cost, however. If each filter knows which filters provide it with input data, then filter C can check whether filter B is active, hence producing output data, or not. If not, filter C chooses not to update and as a consequence it does not produce any output data either. As easily perceived, this solution creates a chain reaction, effectively rendering most of the pipeline execution useless, but it avoids any application crash. Looking back at the previous example (figure 4.1), it is also possible to see that with this solution, if filter E, for example, would stop, filters A, B, C, D and G would still update.

4.1.1.4 Working out extra dependencies - Molding the Pipeline flow as needed, through filters

Still, about the filters, there is another detail worth mentioning. The pipeline is responsible for making sure that all filters are updated orderly but, for whatever reason, there might be the need to assure some different update order. To address this, it is possible to assign to any given filter a series of other filters, of which that given filter is dependent, that is, it can't update unless the others have expressed so. This is, essentially, a form of forcing the pipeline to update the filters in an arbitrary order.

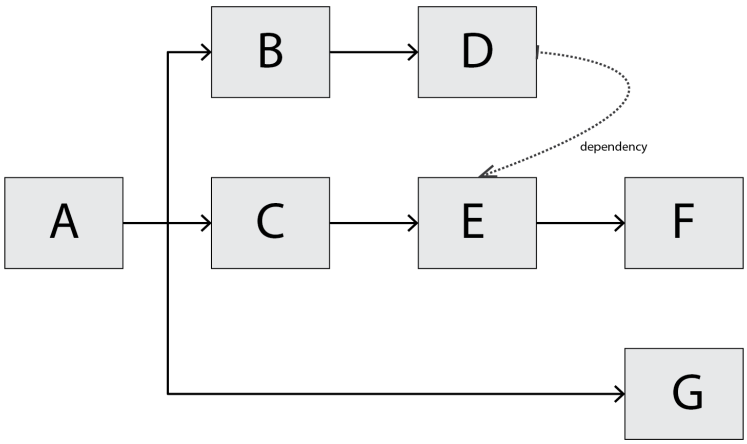

Figure 4.3: A pipeline containing seven filters. Filter D is registered as dependent of E.

Take figure 4.3, above, as an example. Filter D is listed as being dependent of filter E. According to the pipeline update process previously described, D would be updated first, but this won't happen unless filter E notifies D that it can update.

As mentioned before, filters can post messages to the pipeline. This is useful when working in the framework mode, building an application on top of it, since filters can communicate with the application, notifying it of any event if needed.

4.1.1.5 Transferring Data between Filters

There are a few details left to explain about the filters, namely how data is transferred between them. Three different approaches were considered to address this matter. At first, an approach based on transmitting messages was considered. Two distinct ways of implementing this messaging system, between filters, were thought of. An analogous system to the pipeline-filter messaging system, explained earlier, would be possible, in which filters would post messages (containing the data to transmit) to a common message pool and upon the update, each filter would check if there were any messages directed to it. This would cause some overhead, since every piece of data to be transmitted would

have to be posted in the common message (or data) pool and every filter would have to constantly check if there was new data for it to take in.

Another way of implementing a messaging system would be to provide, in each filter, a message listener, along with means of sending messages. However, this could be considerably problematic, since, in order to guarantee that no message could be lost, all filters would have to be listening for new incoming messages constantly. To solve this problem, each filter could run on a separate thread, but this would pose serious synchronization problems, rendering the thread's advantages (such as possibly better performance) useless. A simpler solution was needed.

A callback based approach was also considered, in which each filter, upon registration, as an output filter, to other filters had to provide a unique callback that would allow that filter to transfer data to it. This approach was practical, however it could bring some unwanted complexity to later development with the solution, since callback mechanisms aren't always easy to understand (probably setting the solution a bit back on the enabling easy development goal).

Finally, the implemented approach, that drew heavily from the callback based one. In order to avoid having to define a callback when building a new filter, a method specifically for having a filter receive data was included, allowing any type of data to be transmitted. This way, when a filter needs to transmit data to a certain filter, it only has to invoke that method on that filter.

4.1.2 Triggers

Triggers were created to fulfill a special need of controlling the pipeline's flow, to some degree. Fundamentally, a trigger is useful to notify filters that a certain event occurred and as a consequence, there is something that shooed be performed by that filter. When the trigger detects the event, it is said that the trigger "fired".

Although it was stated earlier that triggers force filters to update, what really happens, is a little bit different. So, there might be a difference between what task should a filter perform when triggered by a certain event and the task that the filter should perform regularly, that is, when it is updated. To deal with this possibility, the filter object provides a way of allowing a trigger to notify it.

This behavior does not properly fit any of the three architectural stages, mainly because a trigger is not really a productive element, like a filter is, that is, a trigger does not transmit data across the pipeline.

A trigger is much like a system interrupt, in a way that it provides a notification of asynchronous events. The "trigger" concept emerged from looking at the (touchless) interaction process itself. Imagining that a certain user enters the scene and wants to interact with a system that recognizes hand gestures, a system that could be built with

this solution. The interaction process is solely based on gestures, so a problem arises, how is it possible to know when the user stopped performing a gesture, so an attempt at recognizing it can be performed? That is when a trigger is useful. Assuming that the event that makes a certain trigger fire could be detecting when the tracked hand point stopped, that trigger could then notify a gesture recognition filter that the user stopped performing the gesture so it could be recognized.

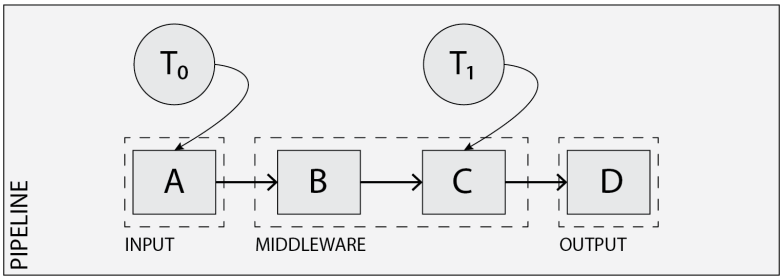

Figure 4.4: A pipeline composed of four filters and two triggers.

To better understand the trigger integration in the architecture, a practical example is presented in section 4.3 - A Practical Example.

4.1.3 Pipeline

The pipeline object has a little more to it than holding all filters and triggers and managing its execution. As the one object that is mandatory for the application to run properly, the pipeline has to hold and manage some essential OpenNI and NITE objects, besides holding all the messages in a message pool, as explained earlier.

These objects are an OpenNI context and a NITE session manager. An OpenNI context is the main OpenNI object, it keeps track of the production nodes in use, so, in a way, it is similar to this pipeline. OpenNI's production nodes are updated through the context object, much like filters are updated through the pipeline object.

NITE's session manager has a similar functionality compared to OpenNI's context object, it holds NITE's control objects. It also controls the session, as described previously. So, the context object is needed for the purpose of accessing some of the Kinect's raw data, like the depth map, as well as for using NITE's session manager, given that there are some OpenNI production nodes that are required in order to have the session manager function properly.

Referring specifically to the OpenNI context object, since it is essential to have the solution working, its update process can occur through one of four distinct routines:

- *WaitAnyUpdateAll*: that waits for any production node to have new data and when so happens, updates all of them;

- *WaitOneUpdateAll*: that waits for a specific production node to have new data and then updates all;

- *WaitAndUpdateAll*: waits for all nodes to have new data and updates them all;

- *WaitNoneUpdateAll*: waits for no node to have new data and updates all of them.

Along with updating the triggers and filters, the pipeline also has to update the context and session objects. This is done in a separate thread (as shown in figure 4.5), to further increase the easiness of developing with this framework (one of the goals). If the pipeline tasks were not to be run parallely, then they would have to be in the application main cycle, or loop. This means that it would be the responsibility of the developer to assure that the pipeline would be properly updated.

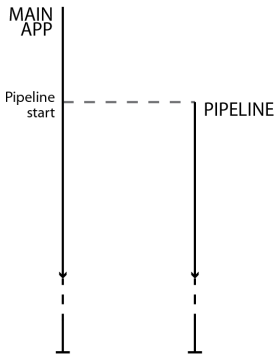

Figure 4.5: The pipeline execution is run parallely to the main application cycle.

4.2 The Three Stages and the Architectural Elements

Previously, each architectural element was described according to its generic functionality, but in this section a more detailed overview of specific filters and triggers is provided. These are elements that extend the generic filter or trigger, providing functionalities that allow any developer to build an application from ground up, solely based on this solution, enabling at the same time, that same developer, to build his/her own filters and triggers as needed. This addresses the first two goals: having the solution work as a standalone application and enabling it to serve as a framework.

Table 4.1 presents a list of filters created in the course of this solution development process. Filters that embed OpenNI production nodes are marked with "OpenNI", whereas filters or triggers that embed NITE controls are marked with "NITE".

Input	Middleware	Output	Trigger
DepthNode[OpenNI]	*Recognizer*	*PrintRecognitionResults*	*PushTrigger*[NITE]
HandsNode[OpenNI]	*Dollar2D*	*OSCSencerNode*	*SteadyHandTrigger*[NITE]
GestureNode[OpenNI]	*Protractor3D*	*HandPointOSC*	*OSCReceiverTrigger*
UserNode[OpenNI]	*SkeletonPoseRecognizer*	*RecognitionResultsOSC*	
PointControlNode[NITE]		*KeyboardEvents*	
RealWorldToProjective		*DepthMap*	
PathMaker		*OpenCVNode*	
PathMaker3Dto2D		*RecognitionResultsToKey*	
SkeletonNode			
SkeletonJoint			
JointToPoint			

Table 4.1: List of Filters and Triggers incorporated in the solution.

4.2.1 Input

It really is not possible to state which of the three stages is most important, but then again, none of the remaining two stages will produce without input data. Since these data is captured from the outside, physical world through an hardware device, MS Kinect, using OpenNI's production nodes, there was the need to incorporate these nodes into the architecture.

The following OpenNI production nodes were encapsulated in filters:

- Depth Generator was encapsulated into a *DephNode* filter, since it is the OpenNI node that provides access to the depth map. Although the depth map is only used for debug purposes, the Depth Generator offers the capability for converting point coordinates between the projective and the real world coordinate system[1]. Encapsulation assured access to these facilities;

- Hands Generator production (encapsulated into the *HandsNode* filter). In spite of being of no special use in the solution, this node is needed by NITE's Point Control (used for starting the NITE session through a "focus gesture");

- Gesture Generator (encapsulated into *GestureNode* filter) the was also included for the same reason as the Hands Generator node. Although it is of no particular use, the gesture detection functionality[2] was made available in the respective filter;

[1]The coordinates mapped in OpenNI do not match the real word coordinates, so, there might be a need to convert them.

[2]It allows to detect if a certain gesture was performed (from a small set of gestures incorporated into OpenNI).

- User Generator node (encapsulated into *UserNode*) offers some critical features, namely the skeleton tracking capability (including the calibration) and detecting if a user has entered or left the scene.

These last three filters (*HandsNode*, *GestureNode* and *UserNode*) may not be required for a certain framework usage but still they must be included in the pipeline. It was opted to not have them embedded into the pipeline to provide, to the developer, more control and transparency over the pipeline and because it suits the architecture.

Regarding NITE, a single control was encapsulated into a filter:

- A Point control (encapsulated into the *PointControlNode* filter). This point control listens directly to NITE's session manager for the hand points[3] being tracked at the moment. The respective filter makes this point data available throughout the pipeline (to whatever filters are connected to it).

In order to enable gesture recognition, a path of points is needed, so besides these filters, two other auxiliary filters were created, to be used still in the input stage:

- A *PathMaker* filter. Receiving a hand point on each update, it builds a point path that it can transmit to other filters;
- A *PathMaker3Dto2D* filter. Just like the normal *PathMaker*, but since the hand point is a 3D point, it builds a path of 2D points exclusively, ignoring the depth data (Z value), enabling a two dimensional gesture recognizer (like $1) to work with these point paths.

Since there are two coordinate systems supported by OpenNI - real world and projective coordinates - conversion between them might be necessary:

- The *RealWorldToProjective* filter, receives a 3D point as input data and converts that point from real world to projective coordinates. The point data is usually provided in real world coordinates and there might be a need to convert from real world coordinates to projective, so it is possible to have hand point data properly mapped to a certain screen area.

Finally, there are three filters related to the skeletal tracking functionality:

- The *SkeletonNode* filter. The *UserNode* provides access to (calibrated) users' skeletal information[4], so there is the need to separate these skeletons and gather just one. This is the sole purpose of this filter;

[3] Although the focus is on single hand gestures, some support for multiple hand detection is provided, as part of the expansibility goal.

[4] As in the likes of the *PointControlNode* filter, the *UserNode* provides more information than needed, since it could be built in order to provide skeletal information of just one user. However, providing skeletal information of more than one user could help in later development, considering again the expansibility goal.

- The *SkeletonJoint* filter. This filter takes as input a user skeleton and outputs a single joint from that skeleton;
- and the *JointToPoint* filter, that converts information from a specific skeleton joint to a 3D point.

4.2.2 Middleware

Although this stage does not comprise many filters, the ones that it does are responsible for the gesture recognition capabilities, a core function of the solution. There is a generic recognizer filter that provides an interface for other filters, that implement any recognition facilities:

- The *Recognizer* filter adds on the regular filter object, by enabling it to function in two different modes, the recognizer mode - when recognition is needed - and the trainer mode - when it is intended to add new templates. A recognizer filter must also incorporate a recognition mechanism. Three different recognizer filters were included in the solution:
  - *Dollar2D* encapsulates the $1 gesture recognition mechanism, it receives a path of two-dimensional points and outputs the results of the gesture recognition;
  - *Protractor3D* encapsulates the Protractor3D gesture recognizer, that takes in a path of three-dimensional points as input data and outputs the gestures recognition results;functionality
  - *SkeletonPoseRecognizer* encapsulates a skeleton pose recognizer, receiving a user skeleton as input and transmitting the recognition results as output.

4.2.3 Output

The output stage, as the last stage in the chain, is the one that interfaces with the application. Its filters have the sole purpose of transmitting information about the data processing performed by the filters in the prior two stages.

In order to provide communication with pre-existing applications, an OSC filter was included in this stage. The OSC protocol provides simplified message formatting and transmission over UDP[MATC]. There are several implementations available, so even adapting a pre-existing application should be a simple task.

The output filters included are:

- The *PrintRecognitionResults* filter, that receives information from recognition mechanisms (*Dollar2D*, *Protractor3D* and *SkeletonPoseRecognizer*) as prints it out to the console;

- The *OSCSenderNode* is filter that provides an interface to other filters, in a similar fashion to the Recognizer middleware filter. An *OSCSenderNode* based filter only has to prepare its own OSC message as intended (using facilities provided by *OSCSenderNode*), all the related network work is performed by the *OSCSenderNode*. Two different filters were created, based on this one:
  - *HandPointOSC* which purpose is to transmit a tracked hand point through the network;
  - *RecognitionResultsOSC*, that transmits the results of a recognition procedure.
- The *KeyboardEvents* filter, that is responsible for simulating a certain key press;
- The *RecognitionResultsToKey*. Receives a information about a recognition process as input and "converts" it into a given keyboard keycode, as output;
- The *OpenCVNode* filter, used for visualization purposes, such as displaying images that it takes as input data;
- The *DepthMap* filter, that receives a depth metadata object[5] and outputs an OpenCV compatible image, for displaying purposes.

4.2.4 Triggers

Triggers do not fit any of the stages, but are useful for controlling filters that do. A few of NITE's controls provide gesture recognition by interpreting stream point data. Some of these controls might be useful to use as triggers, detecting certain specific gestures as events and modeling the pipeline flow as intended.

Two of these controls were encapsulated into triggers:

- Push Detector (encapsulated into the *PushTrigger*). A trigger that fires when a push gesture is detected;
- Steady Detector (encapsulated into the *SteadyHandTrigger*). This NITE control works much like the push detector, but with a different purpose, it is used to detect when the tracked hand is stopped (or barely moving);

An extra trigger was created, to show how information gathered from outside the pipeline can influence its flow:

- The *OSCReceiverTrigger* fires when a specific message is received via OSC.

[5] A depth metadata object includes not only a depth map but other useful information, such as its resolution.

4.3 Base Classes

The base classes, responsible for most of the work, are encapsulated in the above-mentioned filters. Some of the classes represent object types that are very common throughout all the implementation, such as an object that holds a gesture.

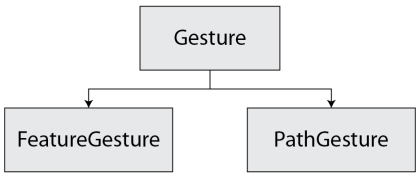

Figure 4.6: Gesture classes.

In order to store the gestures properly, a parent *gesture* class was created, supporting two other classes:

- *PathGesture* class, that stores gestures as paths of points. This is a template class, meaning that it supports either two-dimensional (*PathGesture<Point2D>*) or three-dimensional (*PathGesture<Point3D>*) data;
- *FeatureGesture* class, aimed at representing gestures to be recognized by statistical classification techniques[6].

Likewise, the *GestureRecognizer* class is also a parent class for two other classes:

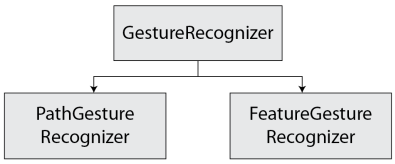

Figure 4.7: Gesture Recognizer classes.

- The *PathGestureRecognizer* class, that holds a template matching gesture recognition mechanism as well a training one. This class has two inheritant classes, which implement the recognition mechanisms:
  - *DollarRecognizer* implements the $1 gesture recognition mechanism;
  - *Protractor3D* implements the Protractor3D gesture recognition mechanism.
- The *FeatureGestureRecognizer* class, serving as a placeholder for a gesture recognition mechanism based on statistical models, since none were implemented.

[6] Again, although this feature was not implemented, some support was included, taking into account the expansibility goal

The training mechanisms are contained in the *GestureTrainer* class that, like *GestureRecognizer*, also has two inheritant classes:

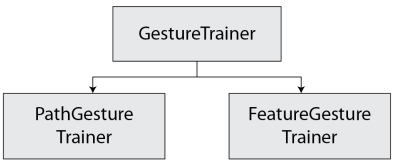

Figure 4.8: Gesture Trainer classes.

- *PathGestureTrainer* which implements gesture training facilities compatible with template matching gesture recognition, that is, it loads from and stores to files path gestures (two or three-dimensional);
- *FeatureGestureTrainer* is also a placeholder, in the likes of *FeatureGestureRecognizer*.

There are other classes that make up the solution, along with the ones presented, such as the Joint and the Skeleton classes, used to represent skeletal information. The classes *Pose* and *PoseDelta*, that represent a distance between joints, are used to, as the name suggests, represent skeleton poses. The *PoseRecognizer* and *PoseTrainer* classes work in a similar fashion to its gesture counterparts, recording poses as distances between specific skeleton joints and attempting to recognize them by comparing a candidate pose with the recorded poses.

As it is possible to see, from the brief descriptions of each base class, concerning its design, a similar approach to the architecture was taken, allowing to have sufficiently generic classes that can be easily extended in the future, once again, in accordance to the expansibility goal.

Regarding the this concept of base classes and the architecture, and considering specifically the expansibilty goal, the architectural elements (filters and triggers) were designed, as described, to be easily extendable. A trivial example of an extended Filter follows:

```
1  class GenericFilter : public Filter {
2  private:
3      DATA_TYPE myData; // to store the data to send to other filters
4  public:
5      GenericFilter(string name="GenericFilter") : Filter(STAGE,INPUT_DATA_TYPE,
           OUTPUT_DATA_TYPE,name) {
6          myData = DATA_TYPE();
7      }
8
9      void update() override {
10         if (!shouldGo()) // checks if the filter should update
11             return;
```

```
12
13          // filter logic goes here
14
15          spread(&myData); // spreads the results over the output filters
16      }
17
18      void trigger() override {
19          // what the filter should perform when triggered
20      }
21
22      void receive(void* data) override {
23          myData = *((DATA_TYPE*)data);
24      }
25
26  };
```

As for an example of an extended Trigger:

```
1   class GenericTrigger : public Trigger {
2   public:
3       GenericTrigger(string name="SteadyHandTrigger") : Trigger(name) {
4
5       }
6
7       void update() override {
8           if (!isActive())
9               return;
10
11          // logic that verifies if a certain event occurred
12
13          if (event)
14              fire();
15      }
16  };
```

4.4 A Practical Example

Looking at a practical example may help consolidate all these concepts.

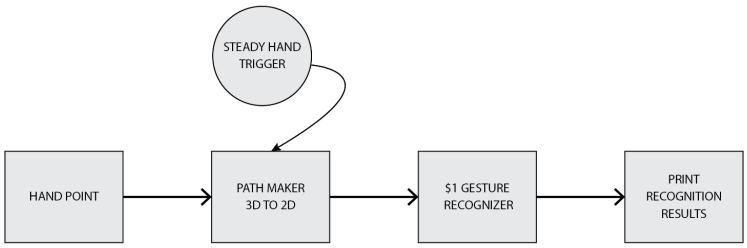

Figure 4.9: A pipeline for gesture recognition.

Considering the pipeline pictured above (figure 4.9), it is constituted of four filters and a trigger:

- A Hand Point filter (*PointControlNode*) that gathers point data from the NITE session manager, and outputs a 3D point, although it does not take any input;
- A Path Maker 3D to 2D filter (*PathMaker3Dto2D*), which receives a 3D point, converts it to a two-dimensional point and adds it to an internal set of points (a *PathGesture<Point2D>*);
- A $1 Gesture Recognizer filter (*DollarRecognizer*), that takes as input a *PathGesture<Point2D>* and attempts to recognize it, providing the recognition results as output (a *RecognitionResults* object, that contains the gesture name that is most similar to the candidate and the respective score);
- A Print Recognition Results filter (*PrintRecognitionResults*), that receives a *RecognitionResults* object and prints its contents to the console;
- Finally, a Steady Hand trigger (*SteadyHandTrigger*) that, upon detecting a steady hand triggers all the filters connected to it (in this case, just the *PathMaker3Dto2D* filter).

The first step to create the pictured pipeline would be to declare the filters and the trigger, and then connect them:

Filter Name	**Input Filter(s)**	**Output Filter(s)**
Hand Point	(none)	Path Maker 3D to 2D
Path Maker 3D to 2D	Hand Point	$1 Gesture Recognizer
$1 Gesture Recognizer	Path Maker 3D to 2D	Print Recognition Results
Print Recognition Results	$1 Gesture Recognizer	(none)

Trigger Name	**Followers**
Steady Hand Trigger	Path Maker 3D to 2D

Table 4.2: Connections between filters and a trigger's followers.

After connecting them and registering them in the pipeline, the pipeline would be started. So, imagine now that a user enters the scene, and performs a session start gesture in order to interact with the system. As a consequence, the hand that performed the gesture starts being tracked and its corresponding hand point is gathered by the Hand Point filter, from the NITE session manager.

That 3D point is transmitted to the *PathMaker3Dto2D* when the pipeline updates the *HandPoint* filter, by having the *HandPoint* filter spreading that data to all its followers. As the *PathMaker3Dto2D* is updated (after receiving the point data), it converts those points into 2D points and adds them to the *PathGesture<Point2D>* object it contains. At a certain instant, the user stops his/her hand. This event causes the *SteadyHandTrigger* to go through its list of followers (in this case it is only the *PathMaker3Dto2D* filter) and trigger them.

As the *PathMaker3Dto2D* filter is triggered, it spreads its *PathGesture<Point2D>* object through all its output filters (the $1 *GestureRecognizer* filter, in this case). The $1 *GestureRecognizer*, upon receiving a *PathGesture<Point2D>* stores it and when it is updated, it attempts to recognize the gesture, producing a RecognitionResults object as a consequence and transmitting it to all its filters registered as output (the *PrintRecognitionResults* filter). As the *PrintRecognitionResults* filter is updated it prints out the object that received from the $1 *GestureRecognizer* filter.

4.5 The TIFEE Stack

The TIFEE solution is organized in a stack, as presented:

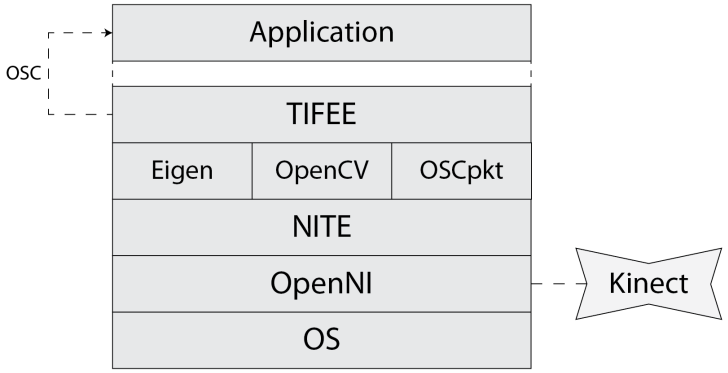

Figure 4.10: The “TIFEE Stack”.

OpenNI is at the base of the solution, providing the interface, along with NITE (but at a different level), between the solution and the hardware. In order to access the hardware raw data so that it is available for both NITE and the solution (TIFEE), some OpenNI’s production nodes had to be embedded in the solution. The same goes for some of the NITE’s controls.

In the TIFEE layer resides the implementation of some of the solution's features, such as gesture recognition. In order to support some of these features, a few libraries were selected and incorporated into the framework:

- Eigen, used in the Protractor3D implementation, to work with matrices (namely eigenvectors and eigenvalues). It provides a simple API, it is cross-platform and it does not have any dependencies, other than the C++ standard library. Eigen's software license (LGPL) allows for it to be included in proprietary, closed-source software [Eig];

- OSCpkt, a minimalistic OSC library that, like Eigen, does not have any dependencies except the C++ standard library. It is also cross-platform and its software license (zlib) also allows for it to be included in proprietary, closed-source software [OSCb];

- OpenCV, a well-known computer vision library, that provides support for working with images. It is used to display the depth map, mostly for debug purposes. Like Eigen and OSCpkt it is cross-platform and its software license (BSD) also allows for it to be included in proprietary, closed-source software [Wila].

Referring to the matrix math library used (Eigen), other libraries were considered, however they lacked some features that Eigen includes. Armadillo [Arm] does not offer eigenvector and eigenvalue support without including other libraries, the same goes for IT++[IT+]. Newmat [Dav] could also be used, however it is not associated with a particular software license and Eigen provides better documentation.

Other OSC libraries were considered too, OSCpack [OSCa] is also a C++ OSC library however it does not implement OSC pattern matching and it is heavier than OSCpkt.

Also, both OSCpkt and Eigen libraries, are based exclusively on header files, which makes it simpler to have them included in the solution (when comparing to the usage of dynamic libraries, for example).

OpenCV was chosen since it is a well-known image processing library, resulting in the high availabilty of proper documentation and examples. It also boasts good performance as well as image processing facilities (implementation of some image processing algorithms, such as the Canny edge detection mentioned in section 2.2.4 - Computer Vision). Such facilities can be useful in later development. Take for example the image frames provided by MS Kinect: using image processing techniques might help extract information from those image frames, provided by the device. That information can be useful for the interactive process. Providing the developer with this opportunity helps fulfill the expansibility goal.

CImg [CIm] is another image processing library that could be used, however, despite having the advantage of being composed of only a header file, it does not include any particular image processing feature.

Chapter 5

Tests and Results

The solution went through a series of tests, that are documented, together with their results and corresponding analysis. The testing process occurred, naturally, throughout the development of the solution, testing each procedure as it was implemented. These tests were focused mainly on the features provided, such as the gesture recognition (testing both algorithms that were implemented: Protractor3D and $1), the OSC related filters as well as the architecture design (testing connections between filters and triggers, for example). Each of the OpenNI and NITE encapsulated filters were tested too, seeking to assure that the respective functionalities were maintained.

Every single filter, and the corresponding functionalities, were tweaked in order to offer an acceptable performance, that is, the set of filters provided were built accordingly to what seemed the best way to fit their respective purpose.

During this testing stage a small, simple application was built with two goals in mind: serve as an example of an application built using TIFEE as a framework and to function as a gesture trainer (recording gestures to files) and as a testbed for the implemented gesture recognizer mechanisms.

Figure 5.1: A snapshot of the testbed application.

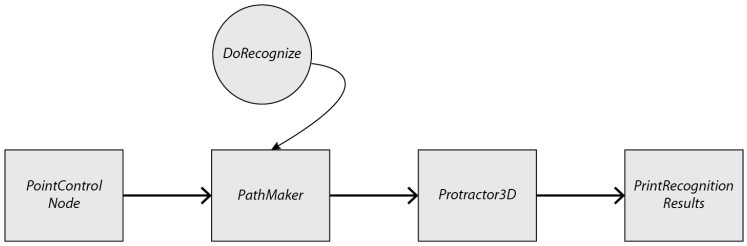

Figure 5.2: The pipeline for the testbed application.

The following aspects of the solution were considered for testing, due to their close relation to the goals (chapter 1 - Introduction):

- **Performance**: considering the interaction process, performance is vital, since building a well-responsive application, using the framework, is considered a goal. It is crucial that all the processing (and time consumption) related to the solution can be of reduced weight in the overall set of the application and the framework processing.

- **Versatility/Flexibility**: having built an application with the framework as a basis, it should be easy for a developer to alter the contents of the pipeline, that is, adding a new filter or trigger, or removing another should be as easy as possible.

- **Expansibility**: seeing that expansibility is one of the goals, the possibility of creating a new filter, or trigger, offering a new capability to the system should not pose as a difficult task to any developer. As an example: if a developer wishes to add a new template gesture recognition mechanism to the solution, the two tasks that he/she would need to perform would be to implement that mechanism and encapsulate it into a filter.

Two other aspects were also considered, however due to requiring a considerable amount of time to test as well as a set of test subjects (developers) these aspects were left out of the batch of tests conducted for this work:

- **Applicability**: taking in account the intended use for the solution - building applications based on touchless interaction - it is only natural to ask at what extent the solution is applicable, that is, when and how it can be useful.

- **Usability**: since the proposed solution is built as a framework, intended for developers to use, the question of how much usable is the API provided is posed. This question, in turn, can lead to other interrogations: Does developing based on the solution requires deep knowledge on the subject - touchless interaction and gesture recognition? Is it necessary for the developer to have a deep understanding of how each filter, trigger and the pipeline work in order to use them?

These two characteristics would require, in a testing process, a carefully selected set of developers to test, since they would have to have time available to be properly introduced to the theme as well as the solution, in order to later try and develop an application on the solution, providing feedback about the experience after. Unfortunately, this was not possible to achieve within the work's timeframe.

The three characteristics that were chosen to evaluate the proposed solution were performance, versatility and expansibility. Versatility is related to usability, as described before. How versatility was achieved can be shown through a proof of concept. However, the same is not applicable for usability.

Even though it was explained throughout the document how expansibility was sought (Chapter 3 - Proposed Architecture and Section 4.3 - Base Classes), it is not an easy characteristic to prove, or measure, being dependent of tests with developers and hence subjective opinions.

Performance is a characteristic that can be considerably easy to measure. By measuring how much time each filter, in a pipeline, consumes on average, it is possible to infer the overall performance of any given pipeline, thus evaluating the solution implementation.

5.1 Testing Conditions

To properly test the solution, that is, test the performance, versatility and expansibilty characteristics, a testing setup was developed. This arrangement includes a couple of applications and time logging mechanisms. The system (an ASUS K53SV) in which the tests were carried out is composed of the following components:

- Intel Core i7-2630QM;
- GeForce GT540M;
- 6GB of DDR3 RAM;
- running Windows 7 64bit Home Premium.

5.1.1 Example Applications

The two applications developed were built taking into account the three chosen aspects to test. Each of the applications has a different set of filters and triggers, forming different pipelines with distinct execution flows. To test the performance a clocking mechanism was built in the pipeline execution management, with the purpose of measuring how much time, on average, each filter needed to perform its task as well as how much time was

spent on updating OpenNI's production nodes (to gather new data from the hardware device) and, as a consequence, from NITE's session manager too (this is called the pipeline overhead).

5.1.1.1 Paint

Paint is a simple example that uses the solution to provide an application that can transmit hand point data to another preexisting application, via OSC. That other application receives the hand point data and interprets it as the position of a cursor, painting a canvas in the corresponding location.

Figure 5.3: The pipeline for the Paint example application.

Figure 5.4: A snapshot of the Paint application.

5.1.1.2 True/False

This example application was built to showcase the gesture recognition capabilities. Using the solution as a framework, an application that could send the gesture recognition results via OSC was developed. On the other side (the preexisting application), a question is displayed onscreen and the user must answer either true or false, by performing "checkmark" or "delete" gestures, respectively. A score is kept accordingly to the user's responses.

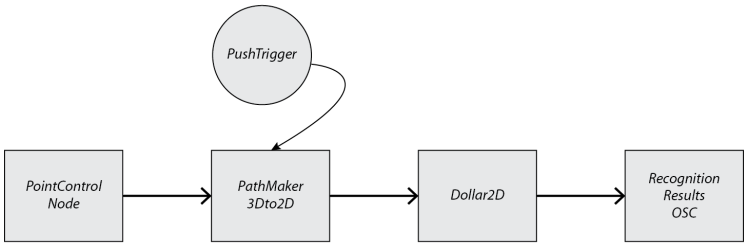

Figure 5.5: The pipeline for the True/False example application.

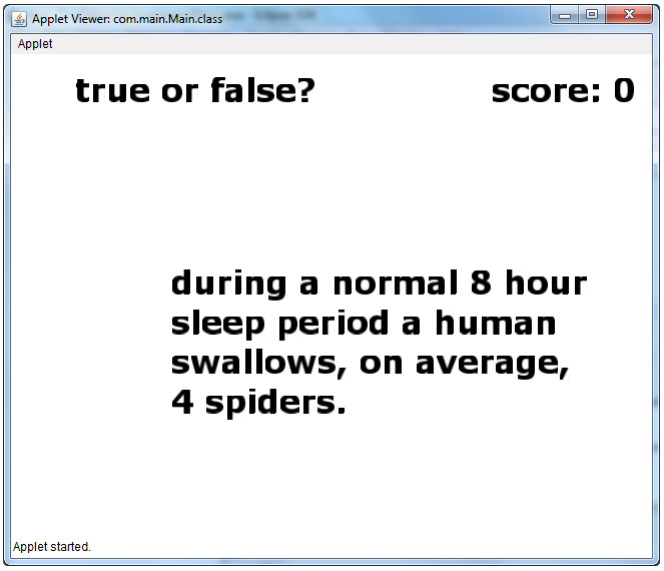

Figure 5.6: A snapshot of the True/False application.

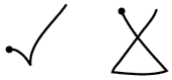

Figure 5.7: The "checkmark" (left) and "delete" (right) gestures.

5.2 Evaluation Assessment

Through the development of the two example applications it was sought to show how flexible and expansible the framework really is. The performance tests were carried out in these two applications too.

5.2.1 Performance

This test was executed through a proper mechanism, built into the pipeline, for the effect: since the pipeline is responsible for updating all filters, by selecting a specific filter, one could force that filter to update any given number of times (iterations). By clocking the

time difference between when the updating process started and when it ended and dividing it by the number of times the filter updated, an average update time is obtained:

$$UpdateTime = \frac{updateend - updatestart}{iterations} \text{ (milliseconds)}$$

Still, it had to be assured that the filters, upon that force repeated updating, had already received data, so as to test a normal filter update routine.

To provide an overall view of each filter clocking results, the filters of both applications were grouped together in the following table, along with each application's pipeline overhead.

	1000000
DepthNode	0.001597
UserNode	0.000769
HandsNode	0.00103
GestureNode	0.000026
PointControlNode	0.002572
RealWorldToProjective $^{\triangle}$	0.003136
HPOSC	0.052004
DepthMap	3.57782
OpenCVNode	0.006214
PathMaker2D †	0.001194
Dollar2D †	0.001226
RecognitionResultsOSC †	0.086737
ConsolePrint †	0.000047
Pipeline Overhead	0.025773

Table 5.1: Clocking results.

A †marks the True/False application pipeline's filters and a $^{\triangle}$ marks the Paint application pipeline's filters. Unmarked filters are common. All the measurements are taken in milliseconds.

It is important to mention that, upon the clocking process for the True/False application, the gesture trainer (holds the gesture templates) for the *Dollar2D* filter was holding 13 templates.

The Pipeline clocking results represent the time that it took, on average, to update the OpenNI context object and the NITE session manager.

5.2.2 Extensibility and Versatility

To showcase the extensibility and versatility properties two example applications were developed, from scratch. The corresponding pipelines were designed, as pictured in 5.3 and 5.5, according to the needs of the those two applications; in the case of the Paint

application there was the need to have the hand point data transmitted over OSC, in turn the True/False application needed to receive the name of the recognized gesture, over OSC too.

5.2.2.1 Paint

When designing the pipeline, at first, the corresponding pipeline for this use case seemed to only need two filters: one to provide the hand point (*PointControlNode*) data and another to transmit that data via OSC (*HandPointOSC*). So, initially, this pipeline would be translated into the code displayed in the .

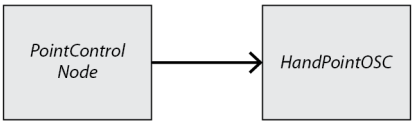

Figure 5.8: The first pipeline for the Paint example application.

However, due to the fact that the hand point coordinates provided by the *HandPointOSC* filter did not allow a correct mapping of the hand point in the painting application, there was a need to convert those coordinates. So, in order to have the hand point data properly mapped by the painting application a filter that would convert hand point data from Kinect's (real world) coordinates to projective coordinates had to be created. This filter was the *RealWorldToProjective* filter, mentioned in the subsection 4.2.1 - Input. The OpenNI "user" production node provides methods for converting between these different coordinate systems. However, this production node was already encapsulated in a filter - the *UserNode* - that provides as output a user's skeletal information. To overcome this problem the *RealWorldToProjective* incorporates a pointer to the *UserNode* filter so it can perform this conversion when needed. Coding this filter, in the framework, we would have the code available in the D.2.

After having built the needed filter, adding it to the described pipeline would be just like adding any other pre-existent filter, as displayed in D.3.

5.2.2.2 True/False

The corresponding pipeline of this application example is similar to the "testbed" application pipeline, but since it does not have any interface elements (the "testbed" example has buttons) it requires a trigger to detect when to attempt gesture recognition. So, translating the framework side of the example to would result in the code displayed in E.1.

Since there is the need to know when to perform the gesture recognition - an asynchronous event, when compared to the normal pipeline flow - a new trigger was built for the effect, the *PushTrigger* (referenced in section 4.2.4 - Triggers). This trigger has the

purpose of detecting a push gesture and, as a consequence, forcing the gesture recognition process to happen on the *Dollar2D* filter. Adding the trigger to the framework produced the code displayed in E.2.

All that is left to do now is declare the trigger and register the *PathMaker* filter to it, so, when a push gesture is detected, the trigger notifies that filter, making it transmit the path (gesture) that has been performed so far to the gesture recognizer. This is considerably easy to achieve, as displayed in E.3.

Chapter 6

Discussion

Looking back at the design and development process it is not hard to see that the architectural design is a considerably important part of the solution as its implementation supports the whole solution. These two processes were undertaken with the utmost precaution, as to avoid any shortcomings. This was achieved, since there are no compromising design or implementation failures. However, there are some design and implementation aspects that could be improved. These are presented next.

6.1 Architecture

The filter class is possibly the most important element of the architecture, since it provides most of the solution's functionalities. It was designed to best suit the solution's needs, yet there might be some challenges when developing with filters.

6.1.1 Input and Output Data Types

When looking at the filter element, a possible drawback that probably draws the most attention is the fact that only one input type and one output type are allowed. This can pose as a problem if the developer wishes to send, as an example, a pair of values. Although, this issue can be easily circumvented, since the data passing between filters, as implemented, allows to pass any kind of data. Though the developer can encapsulate those values in a structure, class or tuple and transmit them across filters, it requires that the received object is properly parsed in order to collect the values.

6.1.2 Tracking Back Received Data

Also, upon receiving data, a filter has no way of knowing where that data came from, that is, which filter has sent it the data. Again, if there is a need to, any filter is flexible enough so that it is possible to include, upon data transmission, an identifier of the filter that is transmitting that same data. But then again, on the receiving end there would be the need for the developer to parse that information. Still, during the development of the solution there was not a situation in which tracking back the received data was needed.

6.1.3 Automatic Data Type Conversion Between Filters

In the previous chapter (chapter 5 - Tests and Results) it has been used in the "Paint" example the development process of a filter that converts point data and even the "True/-False" example features a filter (*PathMaker3Dto2D*) that converts 3D point data to 2D point data. In both these cases the developer needs to explicitly introduce those filters in the pipeline and connect them properly. It would be better, concerning the "Rapid and Easy Application Development" goal, if the pipeline figured out if these conversions were needed and applied them if so.

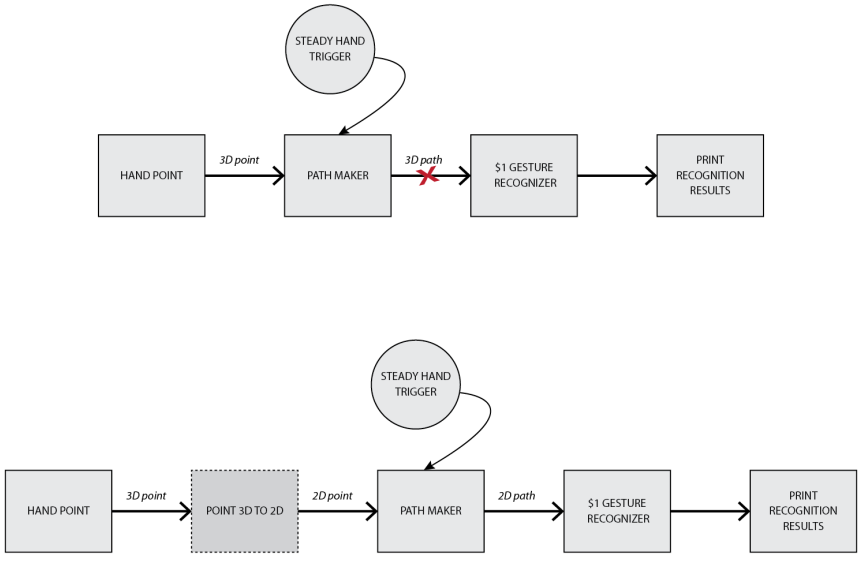

Figure 6.1: An example where automatic conversion between types, using filters, would be useful.

This can be achieved by having the pipeline, just before run-time, iterating through all the filters and checking each of those filter's connections. If any given connection between a pair of filters does not guarantee data type compatibility the pipeline would search for a filter that ensured the needed data type conversion (this would mean that the pipeline had to have a list of filters whose only purpose was converting between data types).

6.1.4 Managing Connections at Run-Time

Regarding the connection management between filters, that was implemented, as described, in the filters themselves, there was also a possibility of having the pipeline control these connections. Currently, it is possible manage the connections (add or remove) with the pipeline still running, this can lead, in some cases, to having most of the pipeline not updatable (much like the case described in 4.1.1.3 - Filter's State and its Impact on Update). So, when using the solution, the developer has the responsibility to ensure that this does not happen, which basically translates into not removing connections at run-time, unless that is what is really intended.

6.1.5 Data Passing Between the Pipeline and its Filters

The filters and the pipeline are distinct entities that do not share the same memory space, i.e., filters' private members are not accessible to the pipeline and vice versa (unless there is a certain method for that specific purpose). In a given situation there might be the need in which a filter needs access to a pipeline's private member, such as the case when the filter needs to post a message to the message pool.

The filter class allows posting messages to the message pool, however it is mandatory that a pointer to the pipeline is passed to the filter at construction time, so that the filter can post messages. This could be overcome if all the pipeline design process (creating, adding and removing filters) was done through the pipeline itself, in a similar manner to DirectShow's Filter Graph Manager (see 3.2.2 - DirectShow).

6.2 Gesture Recognition

Even though the gesture recognition goal was considered as achieved, more could have been provided in the solution. Other approaches to gesture recognition were studied, however they were not implemented. As referenced in section 4.3 - Base Classes, the proposed architecture is prepared to incorporate statistical classification gesture recognizer mechanisms, such as methods based on Hidden Markov Models. Due to the difficulty in implementing this type of gesture recognizers, its training requiring a considerable effort and due to the fact that choosing features to represent the gestures properly is a considerably complex task, it was opted to left them out of the solution.

Nevertheless, looking at the template matching techniques an improvement related to training could be incorporated, to improve the gesture recognition capacity: by adding, to the template gesture database (a gesture trainer), the candidate gesture upon successful recognition. Another mechanism could also be incorporated, the $N gesture recognizer that would allow to recognize gestures with multiple strokes (performed either with more than one hand or with just a hand).

The Protractor recognizer could have also been implemented, since it outperforms the $1 gesture recognizer [Li10]. However this is hardly a critical flaw, since, as shown, the $1 recognizer's performance is satisfactory as it does not cripple the pipeline execution at all.

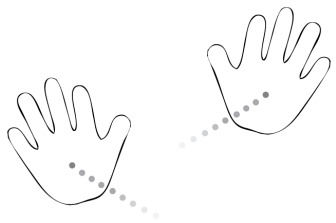

Figure 6.2: A multiple-stroke gesture being performed.

6.3 Other

6.3.1 Cross-Platform

Another drawback is related to the lack of cross-platform compatibility. To overcome this issue a solution is already provided in the framework, the possibility of communicating via the OSC protocol, however this increases the (communication) overhead, degrading the overall performance of the solution. Still, porting the solution to other operating systems is a relatively simple task, since all the libraries used (see section 4.5 - The TIFEE Stack) are cross-platform, so the only OS-specific code is related to the threads and keyboard-press simulation.

6.3.2 Better OpenNI and NITE Integration

During the implementation process it was sought to take the most out of OpenNI's and NITE's capabilities, incorporating them in the solution.

Although the focus of the solution is on a single user, support for multiple users was considered and included, to some extent. The *PointControlNode* filter and the *UserNode* filter are examples. The first already supports the detection of two hand points (could be used with a multiple stroke gesture recognizer, like $N). The latter supports multiple user detection and multiple user skeletal tracking too, but in the solution's current state none of those features are used.

6.3.3 Better Pipeline Building Facilities

Regardless of fulfilling the goal of having the solution functioning as a standalone application, building a pipeline still requires the developer to code, i.e., the pipeline design is hard-coded. This could be avoided by providing a facility to load (and save) a pipeline from an XML file or even having an application, with a proper graphical user interface, that would allow any user to build a pipeline simply by dragging boxes (filters and triggers) on a canvas and connecting them, much like Microsoft's GraphEdit, a visual tool for building DirectShow filter graphs [Mice].

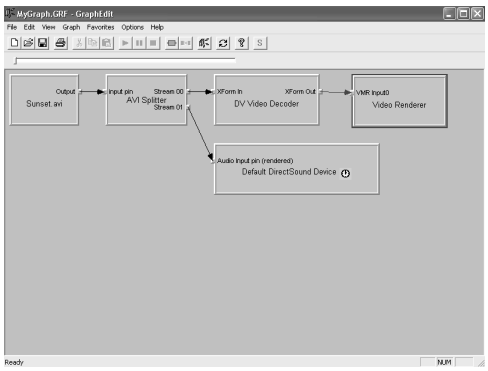

Figure 6.3: DirectShow's GraphEdit. [Micf]

6.3.4 Touchless UI and GUI

As touchless interaction is rapidly seeping into the market every developer makes an effort to leave his/her mark on the touchless interaction scene. This is great as we witness the enormous possibilities of this form of interaction, nevertheless we notice a huge lack of standards, in what concerns UI elements and its GUI counterparts [Nie10]. This can pose as a serious challenge to touchless interaction being established as a widespread form of interaction.

The NITE middleware provides a couple of UI elements, the 1D and 2D sliders that are, essentially, specific areas on the screen that, when the hand point is hovering them, produce a certain value relating the hand point to its own position onscreen.

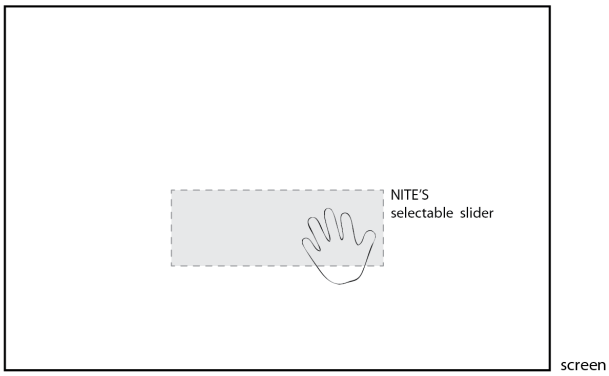

Figure 6.4: A NITE selectable slider.

Although NITE provides these elements, they are far from being enough to build a user interface. There is a need to properly study which UI and GUI elements are most adequate in this touchless context.

In order to provide a simple way to build a set of UI and GUI elements using the solution, thus providing a touchless UI and GUI, a facility for overlaying OpenCV images was built, as displayed in 6.5. However, the performance was not acceptable and it lacked a better design, in order to handle with the GUI elements (e.g. determining if the cursor was hovering a certain button).

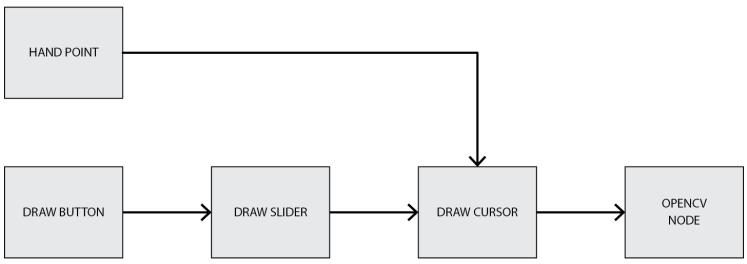

Figure 6.5: An example of a pipeline designed to produce a GUI.

Chapter 7

Conclusions and Future Work

Throughout the document the context in which the described work was developed was presented: a problem was clearly identified and a solution proposed. That same solution, designed with a small, concise and meaningful set of goals in mind, was tested. In this chapter the test results are presented and discussed. The solution, as a whole, is also criticized, from an exempt point of view.

7.1 Conclusions

As explained in the first chapter (Introduction), the set of goals that were devised served as a guideline for both the design and the development processes. As most of the goals were mentioned throughout the document, substantiating design and implementation decisions, this section will cover all of the goals, summing up how they were achieved.

Achieving the goal of having the **solution functioning as a standalone application** meant that it had to include facilities to communicate with preexisting applications. At the architecture level, this would be included in the output section, since whatever it would be to output it had to be the product of previous processing in the pipeline. Inspired by FAAST (referenced in 2.2.3.6 - Other Projects), a facility for simulating key presses was included. Still, generating key presses could, in many cases, be insufficient. So, a facility that would enable the solution to pass information directly to the other application was needed. The OSC protocol was chosen for two motives. It is a well-established protocol with several implementations available and its message formatting and parsing mechanisms are simple to use and understand, requiring little effort from the developer (related with the goal 6.1.6 - Enabling Rapid and Easy Application Development). Anyway, a filter specifically for the purpose of sending OSC messages was created. In order to send

OSC messages through the framework side the developer only has to create a filter based on this provided filter (see Appendix A for details).

Much related to the previous goal, is the goal of having the **solution working as a framework**, or library, given that the facilities provided to the developer for creating a standalone application are equally offered in the "framework mode". The key aspect for having the solution functioning as a framework is in the way that the pipeline execution is controlled: by having the pipeline execution management running in a separate thread, full control of the application's main cycle is given to the developer. This way the developer only needs to assure that the pipeline is started before the main cycle.

The "testbed" application example (section 5.1.1 - Gesture Trainer and Recognizer) serves as proof of concept for this case in particular, where the application main cycle is responsible for managing the GUI.

By providing a sufficiently generic architecture, either on the base classes level (section 4.3 - Base Classes) or on the architecture itself (Chapter 3 - Proposed Architecture), the **expansibility goal** is achieved. It is possible to look at the test applications that were built (sections 5.1.1 - Paint and 5.1.2 - True/False) as proofs of concept of the framework's expansibility, namely in what the architecture concerns.

Creating a new filter or trigger by extending the respective classes (see Appendices A and B) is easy, having the developer to:

1. Declare the class;
2. Write the constructor;
3. Override three methods in case of a filter (*update()*, *trigger()* and *receive(void* data)*), or one in case of a trigger (*update()*).

Having a **fully marketable solution** underwent ensuring that no element of the TIFEE Stack (presented in Chapter 4 - Implementation) could undermine the possibility of producing a commercial application based on the framework. As it was explained throughout the document all these elements have software licenses that enable any developer to produce commercial applications with them.

As mentioned in the first chapter (Introduction), **recognizing hand gestures** was one of the goals. Through the implementation, and encapsulation into filters, of two different gesture recognition algorithms (see section 2.2.1 - Brief Review of Template Matching Gesture Recognition) this goal was achieved. However, when built into a pipeline an issue may pose some barriers to the interaction process. Looking back at the example presented in section 4.1.2 - Triggers, where the *SteadyHandTrigger* functionality is first mentioned and explained, it is easy to understand why this steady hand detection is important. Even though, it lacks a good performance when it comes to detecting a (nearly) stopped hand, since it does not always happen when it should, that is, false positives are common as well

as stopped hands that go undetected. An attempt at solving the false positives problem, by ignoring any steady detection for a short period of time after a first stopped hand was detected, did not improve the performance as would be desirable.

Vindicating the goal of **enabling rapid and easy application development** has much to do with justifying how the expansibility goal was reached. The generic filter and trigger examples provided help in understanding that developing new filters and triggers is a simple task. Interpreting the tests presented in the previous chapter (Chapter 5 - Tests and Results) it is also easy to see that creating a pipeline is a simple task too, since its essentially a task of declaring which filters, and triggers to use, initializing them and adding them to the pipeline.

Providing a good **performance (low latency)** to any application based on the solution was also one of the goals. By interpreting the clocking results presented in the previous chapter one can notice that the time each filter takes to update is considerably small. The filter that consumes the most time to update is the *DepthMap* filter, used only for debug purposes. So, even considering the relatively long update time of the *DepthMap* filter as an issue it is not an essential filter for the interactive process to occur. Other than that, the second most time consuming process is the OpenNI context updating, together with the NITE session manager update (the latter depends of the first). These two updates are mandatory (in order to update the data provided by the hardware device), so they're not really influenced by how the pipeline manages its own execution. However, it is possible to update the context object in different ways (see section 4.1.3 - Pipeline).

Initially the pipeline updating process employed the *WaitAndUpdateAll* method to update the context object. This resulted, on average, on a 16-17 ms clocking time for both the updates (OpenNI context and NITE's session manager), but changing it to the *WaitNoneUpdateAll* method showed a difference of -5 ms (11-12 ms on average) for both updates, together, without any visible effect on the interactive process.

7.2 Future Work

As shown, the goals that defined the solution have all been met. Still, a lot more could have been done.

Touchless interaction is, at the moment, a research area that is evolving at a fast pace, mostly due to the massification of range camera devices such as the MS Kinect. Fundamentally, the goal of this MSc thesis was to harness and to enable others to exploit the possibilities of this recently galvanized form of interaction, by filling a void in the set of tools available to do so.

During the development of this thesis several ideas were considered as suitable features to incorporate in the framework. Though, not all were included or implemented.

Still, as one of the goals, a sufficiently flexible architecture was designed and implemented so it could be easily updated and extended in the future, to better suit any need that might come up.

Incorporating an XML parser (as referenced in section 6.3.3 - Better Pipeline Building Facilities) would allow to further improve the accomplishment of the "easy and fast development" goal. As for the lack of a touchless UI and its GUI counterpart, redesigning the method explained in 6.3.4 - Touchless UI and GUI, would offer a simple way to build interfaces properly, given the context.

7.3 Concluding Remarks

Looking back at the development process of the solution there are some main ideas to retain.

Building a framework demands, from its developer, a critical point of view throughout the whole process, so as to guarantee the delivery of a solution that can please most developers, in terms of what can be expected from a framework: ease of development, expansibility and provide control of the flow. This can be sought, in great part, through the solution's architectural design. So, starting with a good, solid design might just be crucial. In this particular case, different, well-established, architectures were studied, in order to gain some insight on how to better build and design the TIFEE architecture.

As the touchless interaction concept is still considerably unexplored, starting with a clean slate on this subject was an advantage but it also posed some remarkable challenges, as to define how the solution should provide this form of interaction. That is, considering that there are hardly any standards in this context, the process of developing a framework to support it has some pitfalls to it.

As demonstrated, all of the goals were achieved, with a relatively small number of shortcomings attached, that do not compromise the overall usefulness of the solution.

Concluding, the solution might be used as is, in the development on new applications, specifically new approaches on the usual public information stands, providing, in a considerably easy way, a touchless interaction context. The architecture developed and the corresponding implementation, employed in the solution, also enable future developers to extend the solution so as to provide other capabilities and support new devices.

References

[AKM] AKM. AK8973. http://pdf1.alldatasheet.com/datasheet-pdf/view/219477/AKM/AK8973.html [Last accessed: 17 January 2012].

[App] Apple. Apple - iPad 2. http://www.apple.com/ipad/features/ [Last accessed: 17 January 2012].

[Arm] Armadillo. Armadillo - C++ linear algebra library. http://arma.sourceforge.net/ [Last accessed: 17 January 2012].

[ASU] ASUS. WAVI Xtion. http://event.asus.com/wavi/product/WAVI.aspx [Last accessed: 17 January 2012].

[AW10] Lisa Anthony and J.O. Wobbrock. A lightweight multistroke recognizer for user interface prototypes. In Proceedings of Graphics Interface 2010, pages 245–252. Canadian Information Processing Society, 2010.

[Bar97] S.B. Barnes. Douglas Carl Engelbart: developing the underlying concepts for contemporary computing. Annals of the History of Computing, IEEE, 19(3):16–26, 1997.

[Bel] Mary Bells. History of the Computer Keyboard. http://inventors.about.com/od/computerperipherals/a/computer_keyboa.htm [Last accessed: 17 January 2012].

[BMDA10] Andrea Bellucci, Alessio Malizia, Paloma Diaz, and Ignacio Aedo. Don't touch me: multi-user annotations on a map in large display environments. In Proceedings of the International Conference on Advanced Visual Interfaces, pages 391–392. ACM, 2010.

[Bri11] Brian Heater. Microsoft Kinect shatters hyper-specific Guinness world record. http://www.engadget.com/2011/04/27/microsoft-kinect-shatters-hyper-specific-guinness-world-record/ [Last accessed: 17 January 2012], 2011.

[Bro04] T Brosnan. Improving quality inspection of food products by computer vision - a review. Journal of Food Engineering, 61(1):3–16, January 2004.

[BW07] Jamin Brophy-Warren. Magic Wand: How Hackers Make Use Of Their Wii-motes. http://online.wsj.com/article_email/SB117772630151685703-lMyQjAxMDE3NzI3ODcyMjg2Wj.html [Last accessed: 17 January 2012], 2007.

[Cam] Camspace. Camspace. http://www.camspace.com/ [Last accessed: 17 January 2012].

[CIm] CImg. CImg - C++ Template Image Processing Toolkit. http://cimg.sourceforge.net/index.shtml [Last accessed: 17 January 2012].

[Cir] Circle Twelve. DiamondTouch History. http://www.circletwelve.com/products/history.html [Last accessed: 17 January 2012].

[Dav] Robert Davies. Newmat - C++ Matrix Library. http://www.robertnz.net/nm_intro.htm [Last accessed: 17 January 2012].

[DG01] Lijun Ding and Ardeshir Goshtasby. On the Canny edge detector. Pattern Recognition, 34(3):721–725, 2001.

[DL01] Paul Dietz and Darren Leigh. DiamondTouch: a multi-user touch technology. In Proceedings of the 14th annual ACM symposium on User interface software and technology, volume 3, pages 219–226. ACM, 2001.

[Dud76] Sahibsingh a. Dudani. The Distance-Weighted k-Nearest-Neighbor Rule. IEEE Transactions on Systems, Man, and Cybernetics, SMC-6(4):325–327, April 1976.

[Edi] Edigma. Edigma's Displax Skin. http://www.edigma.com/en/products.html#/en/products/displax-skin.html [Last accessed: 17 January 2012].

[Eig] Eigen. Eigen. http://eigen.tuxfamily.org/ [Last accessed: 17 January 2012].

[Fra95] Quentin Stafford Fraser. The Trojan Room Coffee Pot. http://www.cl.cam.ac.uk/coffee/qsf/coffee.html [Last accessed: 17 January 2012], 1995.

[GJM11] S.A. Grandhi, Gina Joue, and Irene Mittelberg. Understanding naturalness and intuitiveness in gesture production: insights for touchless gestural interfaces. In Proceedings of the 2011 annual conference on Human factors in computing systems, pages 821–824. ACM, 2011.

[Groa] NUI Group. Diffused Illumination. http://wiki.nuigroup.com/Diffused_Illumination [Last accessed: 17 January 2012].

[Grob] NUI Group. Frustrated Total Internal Reflection. http://wiki.nuigroup.com/FTIR [Last accessed: 17 January 2012].

[GSt] GStreamer. GStreamer: open source multimedia framework. http://gstreamer.freedesktop.org/ [Last accessed: 17 January 2012].

[Han06] J.Y. Han. Multi-touch interaction wall. In ACM SIGGRAPH 2006 Emerging technologies, page 25. ACM, 2006.

[HBC+96] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong, and Verplank. ACM SIGCHI Curricula for Human-Computer Interaction. http://old.sigchi.org/cdg/cdg2.html [Last accessed: 17 January 2012], 1996.

[HH07] V. Hinze-Hoare. Review and Analysis of Human Computer Interaction (HCI) Principles. 2007.

[HS11] K. Hinckley and H. Song. Sensor synaesthesia: touch in motion, and motion in touch. In Proceedings of the 2011 annual conference on Human factors in computing systems, pages 801–810. ACM, 2011.

[IFia] IFixit. iFixit: MS Kinect Teardown. http://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066/3 [Last accessed: 17 January 2012].

[IFib] IFixit. iFixit PlayStation Move Teardown. http://www.ifixit.com/Teardown/PlayStation-Move-Teardown/3594/2 [Last accessed: 17 January 2012].

[Ima] Mesa Imaging. Mesa Imaging: Swiss Ranger 4000 Overview. http://www.mesa-imaging.ch/prodview4k.php [Last accessed: 17 January 2012].

[Inv08] InvenSense. INVENSENSE? IDG-600 MOTION SENSING SOLUTION SHOWCASED IN NINTENDO'S NEW Wii MotionPlus ACCESSORY. http://invensense.com/mems/gyro/documents/articles/071508.html [Last accessed: 17 January 2012], 2008.

[IT+] IT++. IT++ - C++ library of mathematical, signal processing and communication routines. http://sourceforge.net/apps/wordpress/itpp/ [Last accessed: 17 January 2012].

[JGAK07] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. In Proceedings of the 1st international conference on Tangible and embedded interaction, pages 139–146. ACM, 2007.

[Joh69] E. A. Johnson. Touch Displays, 1969.

[JOS+11] Rose Johnson, K. O'Hara, Abigail Sellen, Claire Cousins, and Antonio Criminisi. Exploring the potential for touchless interaction in image-guided interventional radiology. In Proceedings of the 2011 annual conference on Human factors in computing systems, pages 3323–3332. ACM, 2011.

[KB07] Martin Kaltenbrunner and Ross Bencina. reacTIVision : A Computer-Vision Framework for Table- Based Tangible Interaction. In Proceedings of the 1st international conference on Tangible and embedded interaction, pages 69–74. ACM, 2007.

[KBBC05] Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. TUIO: A Protocol for Table-Top Tangible User Interfaces. In Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation, 2005.

[KEA03] C. Keskin, A. Erkan, and L. Akarun. Real time hand tracking and 3D gesture recognition for interactive interfaces using HMM. ICANN/ICONIPP, pages 26–29, 2003.

[KGH85] M.W. Krueger, T. Gionfriddo, and K. Hinrichsen. Videoplace - An Artificial Reality. In Proceedings of the SIGCHI conference on Human factors in computing systems, volume 16, pages 35–40. ACM, 1985.

[Kin] Kinect Toolbox. Kinect Toolbox. `http://kinecttoolbox.codeplex.com/` [Last accessed: 17 January 2012].

[Kin12] Kineticspace. Kineticspace User Manual, 2012.

[KR10] Sven Kratz and Michael Rohs. A $3 gesture recognizer: simple gesture recognition for devices equipped with 3D acceleration sensors. In Proceedings of the 15th international conference on Intelligent user interfaces, pages 341–344. ACM, 2010.

[KR11] Sven Kratz and Michael Rohs. Protractor3D: a closed-form solution to rotation-invariant 3D gestures. In Proceedings of the 15th international conference on Intelligent user interfaces, pages 371–374. ACM, 2011.

[Kru77] Myron W. Krueger. Responsive environments. Proceedings of the June 13-16, 1977, national computer conference on - AFIPS '77, page 423, 1977.

[Lab] Code Laboratories. Code Laboratories NUI. `http://codelaboratories.com/kb/nui` [Last accessed: 17 January 2012].

[Li10] Yang Li. Protractor: a fast and accurate gesture recognizer. In Proceedings of the 28th international conference on Human factors in computing systems, pages 2169–2172. ACM, 2010.

[LY02] Kalle Lyytinen and Youngjin Yoo. Issues and Challenges in Ubiquitous Computing. Communications of the ACM - Special issue on computer augmented environments: back to the real world, 45(12):62–65, 2002.

[Mal03] S. Malik. Real-time hand tracking and finger tracking for interaction. Technical report, Department of Computer Science - University of Toronto, 2003.

[MATC] Center For New Music and UC Berkeley Audio Technology (CNMAT). Introduction to OSC. `http://opensoundcontrol.org/introduction-osc` [Last accessed: 17 January 2012].

[Mica] Microsoft. Kinect Windows SDK. `http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/` [Last accessed: 17 January 2012].

[Micb] Microsoft. Microsoft Developer Network: DirectShow System Overview. http://msdn.microsoft.com/en-us/library/ms783354(VS.85).aspx [Last accessed: 17 January 2012].

[Micc] Microsoft. Microsoft Kinect SDK For Developers. http://www.microsoft.com/en-us/kinectforwindows/ [Last accessed: 17 January 2012].

[Micd] Microsoft. Microsoft Surface, The Power of PixelSense. http://www.microsoft.com/surface/en/us/pixelsense.aspx [Last accessed: 17 January 2012].

[Mice] Microsoft. Overview of GraphEdit. http://msdn.microsoft.com/en-us/library/ms787460.aspx [Last accessed: 17 January 2012].

[Micf] Microsoft. Programming DirectShow Applications (sample chapter from "Programming Microsoft DirectShow for Digital Video and Television". http://www.microsoft.com/mspress/books/sampchap/6381.aspx [Last accessed: 17 January 2012].

[Micg] Microsoft. Touchless SDK. http://touchless.codeplex.com/ [Last accessed: 17 January 2012].

[MS91] G. Marchionini and J. Sibert. An agenda for human-computer interaction: science and engineering serving human needs. ACM SIGCHI Bulletin, 23(4):17–32, 1991.

[Nie10] Jakob Nielsen. Kinect Gestural UI: First Impressions. http://www.useit.com/alertbox/kinect-gesture-ux.html [Last accessed: 17 January 2012], 2010.

[Nina] Nintendo. Nintendo Wii: The Gyro Sensor: A New Sense Of Control. http://iwataasks.nintendo.com/interviews/#/wii/wiimotionplus/0/0 [Last accessed: 17 January 2012].

[Ninb] Nintendo. Wii - What is Wii? http://www.nintendo.com/wii/what-is-wii/#/controls [Last accessed: 17 January 2012].

[Ope] OpenKinect. OpenKinect. http://openkinect.org/wiki/Main_Page [Last accessed: 17 January 2012].

[OSCa] OSCpack. OSCpack - A simple C++ Open Sound Control (OSC) packet manipulation library. http://code.google.com/p/oscpack/ [Last accessed: 17 January 2012].

[OSCb] OSCpkt. Ultra minimalistic OSC library. http://gruntthepeon.free.fr/oscpkt/ [Last accessed: 17 January 2012].

[Pria] PrimeSense. OpenNI. http://www.openni.org/ [Last accessed: 17 January 2012].

[Prib] PrimeSense. PrimeSense NITE Controls User Guide - March 2011.

[Pri11] PrimeSense. OpenNI User Guide, 2011.

[Rab89] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE, 77(2):257–286, 1989.

[RKE98] Gerhard Rigoll, Andreas Kosmala, and Stefan Eickeler. High performance real-time gesture recognition using hidden markov models. In Martin Wachsmuth, Ipke and Fröhlich, editor, Gesture and sign language in human-computer interaction, pages 69–80. Springer, 1998.

[Rom10] David Roman. Interact naturally. Communications of the ACM, 53(6):12, June 2010.

[Sam] Samsung. Samsung SUR40 for Microsoft Surface. `www.samsunglfd.com/solution/sur40.do` [Last accessed: 17 January 2012].

[Sen] Sensebloom. OSCeleton. `https://github.com/Sensebloom/OSCeleton` [Last accessed: 17 January 2012].

[SK09] Sameer Shariff and Ashish Kulkarni. Identifying Hand Configurations with Low Resolution Depth Sensor Data. Technical report, 2009.

[SLR+] Evan A. Suma, Belinda Lange, Skip Rizzo, David Krum, and Mark Bolas. FLEXIBLE ACTION AND ARTICULATED SKELETON TOOLKIT (FAAST). `http://projects.ict.usc.edu/mxr/faast/` [Last accessed: 17 January 2012].

[Son] Sony. PlayStation Move Product Info. `http://us.playstation.com/ps3/playstation-move/product-information/` [Last accessed: 17 January 2012].

[TBW+] Wim Taymans, Steve Baker, Andy Wingo, Ronald S. Bultje, and Stefan Kost. GStreamer Application Development Manual (0.10.35.1).

[Ubi] Ubiwhere. Ubiwhere. `http://www.ubiwhere.com` [Last accessed: 17 January 2012].

[Var94] J. Vardalas. From DATAR to the FP-6000: technological change in a Canadian industrial context. IEEE Annals of the History of Computing, 16(2):20–30, 1994.

[Wei93] Mark Weiser. Ubiquitous computing. Communications of the ACM - Special issue on computer augmented environments: back to the real world, (804):71–72, 1993.

[WH99] Ying Wu and Thomas Huang. Vision-based gesture recognition: A review. In Gesture-Based Communication in Human-Computer Interaction, pages 103–115. Springer Berlin / Heidelberg, 1999.

[Wika] Wikipedia. Wikipedia: 1990s in Science and Technology. `http://en.wikipedia.org/wiki/1990s_in_science_and_technology` [Last accessed: 17 January 2012].

[Wikb] Wikipedia. Wikipedia: 2000s in Science and Technology. http://en.wikipedia.org/wiki/2000s_in_science_and_technology [Last accessed: 17 January 2012].

[Wikc] Wikipedia. Wikipedia: Capactitive Sensing. http://en.wikipedia.org/wiki/Capacitive_sensing [Last accessed: 17 January 2012].

[Wikd] Wikipedia. Wikipedia: Graphics Tablets. http://en.wikipedia.org/wiki/Graphics_tablet [Last accessed: 17 January 2012].

[Wike] Wikipedia. Wikipedia: Keyboard (computing). http://en.wikipedia.org/wiki/Keyboard_(computing) [Last accessed: 17 January 2012].

[Wikf] Wikipedia. Wikipedia: Kinect. http://en.wikipedia.org/wiki/Kinect [Last accessed: 17 January 2012].

[Wikg] Wikipedia. Wikipedia: Resistive Touchscreen. http://en.wikipedia.org/wiki/Resistive_touchscreen [Last accessed: 17 January 2012].

[Wikh] Wikipedia. Wikipedia: Statistical classification. http://en.wikipedia.org/wiki/Statistical_classification [Last accessed: 17 January 2012].

[Wiki] Wikipedia. Wikipedia: Teleprinter. http://en.wikipedia.org/wiki/Teletype [Last accessed: 17 January 2012].

[Wikj] Wikipedia. Wikipedia: Time-of-flight. http://en.wikipedia.org/wiki/Time-of-flight [Last accessed: 17 January 2012].

[Wikk] Wikipedia. Wikipedia: Time-of-flight camera. http://en.wikipedia.org/wiki/Time-of-flight_camera [Last accessed: 17 January 2012].

[Wikl] Wikipedia. Wikipedia: Touchscreen. http://en.wikipedia.org/wiki/Touchscreen [Last accessed: 17 January 2012].

[Wikm] Wikipedia. Wikipedia: Typewriter. http://en.wikipedia.org/wiki/Typewriter [Last accessed: 17 January 2012].

[Wikn] Wikipedia. Wikipedia: Wii MotionPlus. http://en.wikipedia.org/wiki/Wii_MotionPlus [Last accessed: 17 January 2012].

[Wiko] Wikipedia. Wikipedia: Wiimote. http://en.wikipedia.org/wiki/Wiimote [Last accessed: 17 January 2012].

[Wila] WillowGarage. OpenCV. http://opencv.willowgarage.com/ [Last accessed: 17 January 2012].

[Wilb] WillowGarage. WillowGarage Overview. http://www.willowgarage.com/pages/software/overview/ [Last accessed: 17 January 2012].

[Wil06] A.D. Wilson. Robust computer vision-based detection of pinching for one and two-handed gesture input. In Proceedings of the 19th annual ACM symposium on User interface software and technology, pages 255–258. ACM, 2006.

[Wil10] Andrew D. Wilson. Using a depth camera as a touch sensor. ACM International Conference on Interactive Tabletops and Surfaces - ITS '10, page 69, 2010.

[WKSE11] Juan Pablo Wachs, Mathias Kölsch, Helman Stern, and Yael Edan. Vision-based hand-gesture applications. Communications of the ACM, 54(2):60, February 2011.

[Woo00] Daniel Wood. Methods for Multi-touch Gesture Recognition. Technical report, University of Cape Town, 2000.

[WW11] Daniel Wigdor and Dennis Wixon. Brave NUI world. Morgan Kaufmann, 2011.

[WWL] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. $1 Unistroke Recognizer in Javascript. `http://depts.washington.edu/aimgroup/proj/dollar/` [Last accessed: 17 January 2012].

[WWL07] J.O. Wobbrock, A.D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In Proceedings of the 20th annual ACM symposium on User interface software and technology, pages 159–168. ACM, 2007.

[XIHS06] H. Xu, D. Iwai, S. Hiura, and K. Sato. User interface by virtual shadow projection. In SICE-ICASE, 2006. International Joint Conference, pages 4814–4817. IEEE, 2006.

[YX94] Tie Yang and Yangsheng Xu. Hidden Markov Model for Gesture Recognition. Technical Report May, Carnegie Mellon University, 1994.

Appendix A

Filter Class

```
class Filter {
protected:
        string name;
        Stage stage;
        bool active;
        DataType intype;
        vector<Filter*> inputs;
        DataType outtype;
        vector<Filter*> outputs;

        map<Filter*,bool> dependencies;

        Pipeline* pipe;

        void placeMessage(Message m);
        void placeMessage(string m, void* c=NULL);

public:
        Filter(Stage Stage, DataType inType,
DataType outType, string Name);
        Filter(Stage Stage, DataType inType,
DataType outType, Pipeline* p, string Name);

        virtual void update() = 0;
        virtual void trigger() = 0;
        virtual void receive(void* data) = 0;

        void spread(void* data);

        bool isActive();
        void activate();
        void deactivate();

        void addDependency(Filter *f);
```

```
        void popDependency();
        bool isDependableOf(Filter* f);
        void go(Filter* f);
        void goOutput();
        bool shouldGo();

        DataType getInType();
        void addInput(Filter* f);
        void popInput();
        bool hasInInput(Filter* f);

        DataType getOutType();
        void addOutput(Filter* f);
        void popOutput();
        bool hasInOutput(Filter* f);

        string getName();
};
```

Appendix B

Trigger Class

```
class Trigger {
private:
        string name;
        bool active;
        vector<Filter*> followers;

public:

        Trigger(std::string s);

        string getName();

        bool isActive();
        void activate();
        void deactivate();

        void fire();

        void addFollower(Filter* f);

        virtual void update() = 0;
};
```

Appendix C

Pipeline Class

```
class Pipeline {
protected:

        string name;
        bool active;
        vector<Filter*> filters;
        vector<Trigger*> triggers;

        MessagePool pool;
        bool inSession;

        Context context;
        XnStatus status;

        XnVSessionManager session;
        string focus;

        static UINT threadRun(LPVOID param);

public:
        Pipeline(std::string n="Pipeline",
string f="Wave");

        xn::Context* getContext();
        XnVSessionManager* getSessionManager();

        void addFilter(Filter* f);
        bool filterOnPipe(Filter* f);

        bool link(Filter* first, Filter* second,
bool dependant=true);
        bool addLink(Filter* first, Filter* second);

        void addTrigger(Trigger* t);
```

```
        bool triggerOnPipe(Trigger* t);

        void place(Message m);
        void place(std::string s, std::string m,
void* d=NULL);
        bool hasNewMessage();
        Message getLatest();

        void run();
        void stop();
        void init();
        void step();
        void updateTriggers();
        void updateFilters();

        bool isInSession();
        void startSession();
        void endSession();

        static void XN_CALLBACK_TYPE SessionStart(
const XnPoint3D& pFocus, void* UserCxt);
        static void XN_CALLBACK_TYPE
SessionProgress(const XnChar* strFocus,
const XnPoint3D& ptFocusPoint,
XnFloat fProgress, void* UserCxt);
        static void XN_CALLBACK_TYPE
SessionEnd(void* UserCxt);
};
```

Appendix D

Paint Code Example

D.1 Paint Pipeline - First Version

This is the paint pipeline example in its first stage. Before adding the RealWorldToProjective filter.

```
1  Pipeline pipe;
2
3  UserNode* User;
4  HandsNode* Hands;
5  GestureNode* Gesture;
6  DepthNode* Depth;
7  PointControlNode* HandPoint;
8  HandPointOSC* HPOSC;
9
10 DepthMap* Map;
11 OpenCVNode* View;
12
13 Depth = new DepthNode(pipe);
14 pipe.addFilter(Depth);
15
16 User = new UserNode(pipe);
17 pipe.addFilter(User);
18
19 Hands = new HandsNode(pipe);
20 pipe.addFilter(Hands);
21
22 Gesture = new GestureNode(pipe);
23 pipe.addFilter(Gesture);
24
25 HandPoint = new PointControlNode(pipe);
26 pipe.addFilter(HandPoint);
27
28 HPOSC = new HandPointOSC("127.0.0.1",12000,"HPOSC");
29 pipe.addFilter(HPOSC);
30
31 Map = new DepthMap(width,height);
32 pipe.addFilter(Map);
33
34 View = new OpenCVNode(width,height);
35 pipe.addFilter(View);
36
37 Depth->addOutput(Map);
38 Map->addOutput(View);
```

```
39 HandPoint->addOutput(HPOSC);
40
41 pipe.run();
42
43 while (running) {
44         while (pipe.hasNewMessage()) {
45                 Message m = pipe.getLatest();
46                 cout << "message: " << m.getMessage() << endl;
47         }
48 }
49
50 pipe.stop();
```

D.2 RealWorldToProjective Filter

The code corresponding to the RealWorldToProjective filter, tha converts points between the two projections (RealWorld and Projective) supported by OpenNI.

```
1  class RealWorldToProjective : public Filter {
2  private:
3  DepthNode* depthnode;
4
5  // received point, real world coordinates (input)
6  Point3D received;
7
8  // converted point (output)
9  Point3D converted;
10
11 public:
12 RealWorldToProjective(DepthNode* node, Pipeline* p, string name="
       RealWorldToProjective") : Filter(INPUT_STAGE,POINT3D,POINT3D,p,name) {
13         received = Point3D();
14         converted = Point3D();
15         depthnode = node;
16 }
17 void update() override {
18         if (!shouldGo())
19                 return;
20
21         XnPoint3D toConvert = XnPoint3D();
22         toConvert.X = received.x;
23         toConvert.Y = received.y;
24         toConvert.Z = received.z;
25         XnPoint3D xnconverted = depthnode->convertRealWorldToProjective(
               toConvert);
26         converted = Point3D(xnconverted.X,xnconverted.Y,xnconverted.Z);
27
28         spread(&converted);
29 }
30 void trigger() override {
31         update();
32 }
33 void receive(void* data) override {
34         received = *((Point3D*)data);
35 }
36 };
```

D.3 Paint Pipeline - Second Version

```
1 RealWorldToProjective* PointConversion;
2
3 PointConversion = new RealWorldToProjective(Depth,&pipe);
4 pipe.addFilter(PointConversion);
5
6 // instead of: HandPoint->addOutput(HPOSC);
7 HandPoint->addOutput(PointConversion);
8
9 PointConversion->addOutput(HPOSC);
```

Appendix E

True/False Code Example

E.1 True/False Pipeline

```
1  Pipeline pipe;
2
3  UserNode* User;
4  HandsNode* Hands;
5  GestureNode* Gesture;
6  DepthNode* Depth;
7
8  PointControlNode* HandPoint;
9  PathMaker3Dto2D* PathMaker;
10 Dollar2D* GestureRecognizer;
11 RecognitionResulstsOSC* RROSC;
12 PrintRecognitionResults* PrintRR;
13
14 DepthMap* Map;
15 OpenCVNode* View;
16
17 Depth = new DepthNode(pipe);
18 pipe.addFilter(Depth);
19
20 User = new UserNode(pipe);
21 pipe.addFilter(User);
22
23 Hands = new HandsNode(pipe);
24 pipe.addFilter(Hands);
25
26 Gesture = new GestureNode(pipe);
27 pipe.addFilter(Gesture);
28
29 HandPoint = new PointControlNode(pipe);
30 pipe.addFilter(HandPoint);
31
32 PathMaker = new PathMaker3Dto2D();
33 pipe.addFilter(PathMaker);
34
35 GestureRecognizer = new Dollar2D(pipe);
36 pipe.addFilter(GestureRecognizer);
37
38 RROSC = new RecognitionResulstsOSC("127.0.0.1",12000);
39 pipe.addFilter(RROSC);
40
```

```
41 PrintRR = new PrintRecognitionResults();
42 pipe.addFilter(PrintRR);
43
44 Push = new PushTrigger(pipe);
45 pipe.addTrigger(Push);
46
47 Map = new DepthMap(width,height);
48 pipe.addFilter(Map);
49
50 View = new OpenCVNode(width,height);
51 pipe.addFilter(View);
52
53 Depth->addOutput(Map);
54 Map->addOutput(View);
55 HandPoint->addOutput(PathMaker);
56 PathMaker->addOutput(GestureRecognizer);
57 GestureRecognizer->addOutput(RROSC);
58 GestureRecognizer->addOutput(PrintRR);
59
60 pipe.run();
61
62 while (running) {
63         while (pipe.hasNewMessage()) {
64                 Message m = pipe.getLatest();
65                 cout << "message: " << m.getMessage() << endl;
66         }
67 }
68
69 pipe.stop();
```

E.2 Push Trigger

The code for the needed *PushTrigger* follows.

```
1  class PushTrigger : public Trigger, public XnVPushDetector {
2  private:
3  XnCallbackHandle detectedCB, stabilizedCB;
4  XnVSessionManager* sessionManager;
5
6  public:
7  static void XN_CALLBACK_TYPE PushDetected(XnFloat fVelocity, XnFloat fAngle,
       void *UserCxt) {
8          (*(PushTrigger*)UserCxt).fire();
9  }
10 static void XN_CALLBACK_TYPE PushStabilized(XnFloat fVelocity, void *UserCxt)
       {}
11
12 PushTrigger(Pipeline &p, string name="SteadyHandTrigger") : Trigger(name),
       XnVPushDetector(name.c_str()) {
13         sessionManager = p.getSessionManager();
14         sessionManager->AddListener(this);
15
16         detectedCB = this->RegisterPush(this,PushDetected);
17         stabilizedCB = this->RegisterStabilized(this,PushStabilized);
18         p.addTrigger(this);
19 }
```

```
20
21 void update() override {
22         if (!isActive())
23                 return;
24 }
25 };
```

E.3 Adding the PushTrigger

```
1 PushTrigger* Push;
2 Push = new PushTrigger(pipe);
3 Push->addFollower(PathMaker);
```