UNIVERSITÀ DEGLI STUDI DI TRENTO
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

Final thesis

Using Kinect to emulate an Interactive Whiteboard

Uso del Kinect per emulare le funzionalità di una Lavagna Interattiva Multimediale

Relatore / 1st Reader:
Prof. Marco Ronchetti

Laureando / Graduant:
Mattia Avancini

Anno Accademico 2011 - 2012

“Stay Hungry. Stay Foolish."

— Steve Jobs (Stanford Commencement Address, 2005) —

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ACRONYMS AND ABBREVATIONS

ADC Analog-to-digital Converter

AEC Automatic Echo Cancellation

API Application Programming Interface

BSD Berkeley Software Distribution

CMOS Complementary Metal Oxide Semiconductor

DC Direct Current

DST Dispersive Signal Technology

E3 Electronic Entertainment Expo

FAAST Flexible Action and Articulated Skeleton Toolkit

FBK Fondazione Bruno Kessler

FPS Frame per second

FTIR Frustrated Total Internal Reflection

GNU GNU is Not Unix

GPL General Public License

HCI Human Computer Interaction

IDE Integrated Development Environment

IDE Integrated Development Environment

IR Infrared

IT Information Technology

IWB Interactive Whiteboard

LED Light Emitting Diode

LGPL Lesser General Public License

MIT Massachusetts Institute of Technology

NI Natural Interaction

NITE Natural Interaction Technology for End-user

NUI Natural User Interface

OpenNI Open Natural Interaction

PC Personal Computer

PCM Pulse Code Modulation

QVGA Quarter Video Graphics Array

RAM Random Access Memory

RGB Red, Green and Blue

ROS Robot Operating System

SDK Software Development Kit

SOA Service Oriented Architecture

UML Unified Modeling Language

USB Universal Serial Bus

VGA Video Graphics Array

VRPN Virtual Reality Peripheral Network

WiiLD Wiimote Lavagna Digitale

WiiLDOS Wiimote Lavagna Digitale Operating System

XML eXtensible Markup Language

GLOSSARY

background removal The background removal is a visual feedback reported to the user, where the user body is cut otu form the surrounding environment., p. 50.

click gesture A click gesture is a gesture where the user needs to push hand towards in the direction of the sensor and immediately pull his or her hand back., p. 35.

depth camera A depth camera is a particular type of sensor which is able to measure the depth for each of the captured pixels at a knowing framerate., p. 12.

focus gesture A focus gesture is a gesture that starts a tracking session during which the user's hand is tracked and modeled as an hand point., p. 35.

hack It stands for a workaround, an inelegant but effective solution to a particular problem: in this case use *Kinect* in a different way from usual, that is not connected to Xbox 360. You can find some examples at http://www.kinecthacks.com/, p. 1.

Kinect A motion sensing input device developed and sold by Microsoft to control and interact with the Xbox 360 without the need of a physical controller., p. x.

limbs The straight lines drawn by the SimpleOpenNI library that connects two adjacent joints, it is a graphic model representation of the 3D model of the skeleton., p. 50.

point control A point control is an object which receives the stream with current active hand point, analyze it and tries to recognize some meaningful behaviour, in practice it tries to recognize if a specific gesture is performed., p. 37.

production chain A production chain is the interact between various production nodes to produce high-level data consumed by the application., p. 31.

production node A production node has the key role of *producing* and *generating meaningful 3D data* for *Natural Interaction (NI)* based applications, which can be a complex task., p. 29.

push gesture A push gesture is a gesture where the user needs to push hand towards in the direction of the sensor and immediately pull his or her hand back., p. 38.

range imaging camera A camera that produce a 2D image showing the distance to points in a scene from a specific point., p. 7.

reference pattern The reference pattern is stored in the memory of the sensor and it is defined capturing a specific plane at a known distance from the sensor.[12]. This is used to calculate the distortion of the observed pattern and finally compute the 3D structure of the detected environment., p. xvi.

speckle pattern Kinect uses a dot pattern based on speckles to calculate the depth of the objects. The pattern is obtained projecting an infra-red light on objects and measuring the distortion with the respect to a *reference pattern*., p. 9.

State Pattern The State Pattern is a behavioral design pattern. This pattern is used to represent the state of an object, in particular when an object is a a function of its state and it must change its behavior at run-time depending on that state [9]., p. 73.

steady detector A steady detector is a gesture that starts a tracking session during which the user's hand is tracked and modeled as an hand point., p. 38.

structured light Structured-light is a specially designed light pattern used to determine the depth of object in the space using only a single image of the reflected light., p. 9.

swipe gesture A swipe gesture is a gesture where the user perform a short movement with the hand in a specific direction, followed by the hand resting., p. 38.

wave gesture A wave gesture is a gesture where the user changing the hand direction suddenly from left to right and viceversa within a timeout., p. 35.

1 INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

It is widely known the fact that *traditional Interactive Whiteboard (IWB)* are very *expensive*. At University of Trento there are different teams which work to possible new solutions to this problem examining new technologies and new paradigms of interaction such as the projects we will see in 2.3.3 and 2.3.4. These projects are based on the introduction of *gaming devices* as a concrete tool to support *teaching and learning activities*. They can be considered the forerunners in this type of approach to the problem. They constitute the starting point of this thesis and an important inspiration on the approach we needed to elaborate a concrete solution to this specific problem.

In November 2010 *Microsoft* has released what they define "the birth of the next-generation of home entertainment"[26]. Microsoft didn't now at the moment of the launch the fact that *Kinect* is not only a "common" game device, it is more than that. It is a new technology and a *new paradigm of interaction*, it renders legacy external peripherals currently used today, such as remote controls, keypads or a mouse *obsolete*. It originates a lot of interest in all the *Information Technology (IT)* world, indeed on the Web were appeared rapidly a lot of "*hack*s" and university projects that highlight the potentiality of a technology of this kind. Many of these projects were supported by the frameworks provided by the creator and precursor of this new paradigm of interaction, an Israeli company called "*PrimeSense*" which was the first to publish stable frameworks to interact and develop *Kinect*-based application. Microsoft itself has noted this phenomena and released a non-commercial *Kinect Software Development Kit (SDK) for Windows* on June 16, 2011.

The *Kinect* can be defined a *Natural Interaction (NI)* device, which operates in the context of *Natural User Interface (NUI)*, a new emerging metaphysical paradigm in *Human Computer Interaction (HCI)*. *NI* refers to a concept whereby the *interaction* with devices is based on *human senses* [17]. The main goal of the *NI* paradigm is to transform the legacy interfaces and devices used during the last years of technology evolution in a more intuitive and naturally interaction without the need to wear particular devices or learn complicated instructions to use them. The *Kinect* can perfectly play this role in the process of transformation.

The supervisor of this thesis Professor Ronchetti, who hold the role of scientific coordinator of the various projects designed to find possible *low-cost solutions to emulate an IWB*, he proposed me to study the possibility this new technology and its possible applications in this specific field of research.

1.2 OBJECTIVES

The aim of this thesis is to analyse this new technology and all the problems directly connected to its introduction, then understand if the capabilities and features offered can be compatible with the requirements needed to implement an *emulation* of an *IWB* and lastly develop a possible solution for the problem using the *Kinect*.

1.3 STRUCTURE AND CONTENTS

In this work we will propose a *low-cost solution to emulate an IWB* based on the introduction of the *Kinect* as a concrete tool to support *teaching and learning activities*. The solution proposed in this thesis take advantage from the technology offered by this kind of device, in particular we will see how the *depth camera* of the *Kinect* can be used to perform a *skeleton recognition* mapping the mouse position. We project the computer desktop on a surface and the mouse is positioned exactly where the user puts his or her index finger, controlling in this way the computer involved in the tracking. In Chapter 2 we will present the device, its technical characteristics and specification and its different operating modes. Inside the Chapter 2 we will give a brief look at the *Natural Interaction (NI)* concept, the new paradigm of interaction which *Kinect* belongs. In Chapter 2 we will also see a brief recap regarding the *IWB*, how they works and what are the main technology aspects involved and we will see other two projects developed at University of Trento to propose low-cost solution to emulate an *IWB* introducing other gaming device to support teaching activities.

In Chapter 3 we will analyse the main *Kinect* frameworks proposed by the market which constitute the infrastructure of this thesis. We will see the main functionalities and capabilities offered by *Open Natural Interaction (OpenNI)* and *Natural Interaction Technology for End-user (NITE)*. In addition we will make a brief parenthesis on the *Software Development Kit (SDK)* offered by Microsoft, this solution isn't considered in this thesis because when we started to develop our prototype wasn't already released. We will also propose other possible frameworks you can find on online.

The Chapter 4 is the key point of this work because here we will present our developed prototype relative to the emulation of an *IWB*. The solution we proposed provides an emulation of an *IWB* where we *control the mouse position* mapping it with the user's *hand joint* exploiting the *skeleton tracking* capabilities offered by the device and the *OpenNI/NITE* frameworks, in particular we are able to put exactly the mouse cursor on the top of the index finger of the user. We will start showing the *Unified Modeling Language (UML)* diagrams constitute the foundation of our proposed solution from an external and internal point of view. We will present the SimpleOpeNI library, a simple wrapper, a simple *OpenNI* and *NITE wrapper* for *Processing*. Our prototype was built on top of this library to have a simple and abstract programming tool which can take advantage on all the functionalities and capabilities provided by *OpenNI* and *NITE*. We will drive the user in the process to understand how develop a *Kinect*-based application providing some examples written in Processing we used as reference during our development process. In the last two sections of the Chapter 4 we will analyse the development details of the proposed solution examining the development process we have adopted phase-by-phase and the implementation details contained in this project providing some code examples and solutions to the problem we have encountered.

The last Chapter 5 illustrates the various results and all the data collected during the test phase we performed to understand the quality of the solution provided and how the people react when they have to deal with a technology of this kind.

2 BACKGROUND

2.1 KINECT

2.1.1 Introduction

Kinect[1] is a *motion sensing* input device developed and sold by Microsoft to control and interact with the *Xbox 360* without the need of a physical controller. *Kinect* originally known as "*Project Natal*"[26] enters in the same market sector occupied by *Wii Remote*[2] sold by *Nintendo* and *PlayStation Move*[3] with *PlayStation Ey*[4] motion controllers produced by *Sony* to interact with home consoles *Wii* and *PlayStation 3*, respectively. *Shane Kim*, corporate vice president for strategy and business development at Microsoft's game division defines *Kinect* in the interview reported in [26] as "the birth of the next-generation of home entertainment".

Figure 1: Motion Gaming Controller Comparison. In this picture you can see the *Kinect*, the Wii Remote and the PlayStation Move

1 http://www.xbox.com/en-US/kinect

2 Wii Remote is the primary controller of Wii, a console produced by Nintendo. It allows gesture recognition and pointing through the use of accelerometer and optical sensor technology.

3 It is an handheld motion controller PlayStation 3 produced by Sony. It interacts with the PlayStation Eye camera to track the position and uses inertial sensors to detect its motion in the space.

4 A digital camera, which is based on technology that uses computer vision and gesture recognition to process images taken by the camera. Like the *Kinect* is equipped with a built-in microphone array.

Kinect is not only a new innovative game controller but starting from the launch date on November 4, 2010 it has created a lot of interest in all the *IT* world, indeed on the Web were appeared rapidly a lot of *"hack*s" and university projects that highlight the potentiality of a technology of this kind. Microsoft itself has noted this phenomena and has released a non-commercial *Kinect Software Development Kit (SDK) for Windows* on June 16, 2011. Some rumours on the Web as reported in [31] talk about a possible integration of the *Kinect* on Windows 8: leaked documents out of Redmond show some *Kinect*-like features inside Windows 8. We will discover the fact that control the computer using *Kinect* is a real possibility in the future.

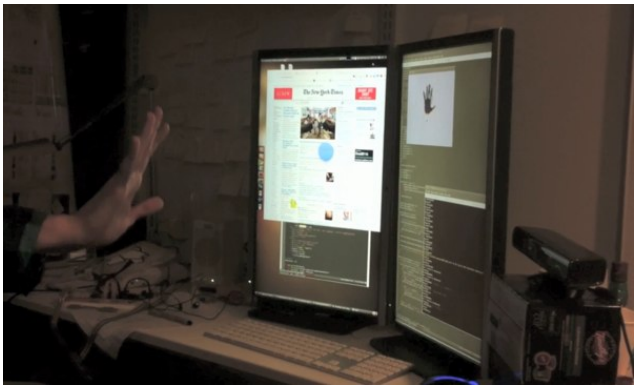

Figure 2: DepthJS. A famous "*Kinect* hack" developed by a group from *MIT* Media Lab by the name of Fluid Interfaces Group. DepthJS is framework that allows any web page to interact with Microsoft *Kinect* via Javascript.

In this chapter we will present the Microsoft *Kinect* starting from a brief *history* relative to its development and launch. The next step is to going into more *technical details* seeing the hardware components and some specifications. The chapter ends with a broad explanation on the different *operating modes* of the *Kinect*.

2.1.2 History

In this section we will report a brief history relative to *Kinect* development and launch. *Kinect* was first announced on June 1, 2009 during the *Electronic Entertainment Expo (E3)* 2009, identified with the previously mentioned code name "*Project Natal*". On June 13, 2010 during the E3 2010, Microsoft announced that the device would officially be called *Kinect* and the launch date scheduled for North America will be November 4, 2010. The Europe launch date was postponed by one week. Having sold *8 million* units in its *first 60 days* on the market, *Kinect* has claimed the Guinness World Record of being the "fastest selling consumer electronics device"[14].

Kinect hardware is based on the technology developed by a company called *3DV System* acquired by Microsoft in June 2009 [34], specifically on the product *ZCam*, a real-time range imaging camera. In [15] Microsoft announces the fact that another Israeli company *PrimeSense* supplies the *3D sensing technology* used in *Project Natal*. PrimeSense is also working with *ASUS* to create a new device called *WAVI Xtion*, a pc-compatible device similar to *Kinect*. The launch of this *Kinect*-like device will be scheduled in the second quarter of 2011 [5].

Figure 3: WAVI Xtion. A motion sensing control device produced by ASUS.

From a software point of view the *Kinect* software technology is developed internally by *Rare*, a subsidiary of Microsoft Game Studios owned by Microsoft. In December 2010, as reported in [16], PrimeSense released their own *open source drivers*, along with *motion tracking middleware* called "*Natural Interaction Technology for End-user (NITE)*". PrimeSense create with the collaboration of two two other companies a not-for-profit organisation focused on *certifying* and *improving interoperability* of *Natural Interaction (NI)* devices called "*Open Natural Interaction (OpenNI)*". *OpenNI* is not only an organization but also a *cross-platform framework* that provides the interface for both physical devices and middleware components as described in [17].

Figure 4: OpenNI logo

Figure 5: PrimeSense logo

2.1.3 Kinect Hardware and Specification

As you can see in Figure 6 *Kinect* is not a *single piece device* but it's composed by a lot of different components and technologies to offer to the user a new entertainment experience that involves different senses.

Figure 6: Kinect disassembled. This Figure shows the result of a disassembling procedure performed on the device by a specialized website.[5]

The *Kinect* is designed to be positioned below the video display and is mainly formed by an *horizontal bar of sensors* connected to a small base with a *motorized pivot*. The motorized base enables to change the vertical range of acquisition up or down of ±28° as reported in [23]. The Figure shows the three main sensor of the *Kinect* or better the "Kinect's eyes": *two cameras* and an *Infrared (IR) projector*. The first component on the left is the *IR* projector as illustrated in the Figure , the central components is a *Color Complementary Metal Oxide Semiconductor (CMOS)*, a simple *Red, Green and Blue (RGB) camera* with a resolution of 640×480 32-bit colour at 30 frames/sec and lastly on the right we have the *IR CMOS* or the *IR Receiver* with a resolution of 320×240 16-bit depth at 30 frames/sec. The device is also equipped with an array of microphones that permits the *Kinect* to receive and recognize *vocal commands*. The multi-array microphone enables *acoustic source localization* and *ambient noise suppression*. The four microphones are disposed in a line, three of them in the left side and another in the right, all of them placed below of the device.

Below we report the specification as descripted in [23]: the sensor has a viewing angle field of 57° horizontally and 43° vertically, while the mechanical drive in the base, as previously mentioned is capable of moving the sensor to 27° either up or down.

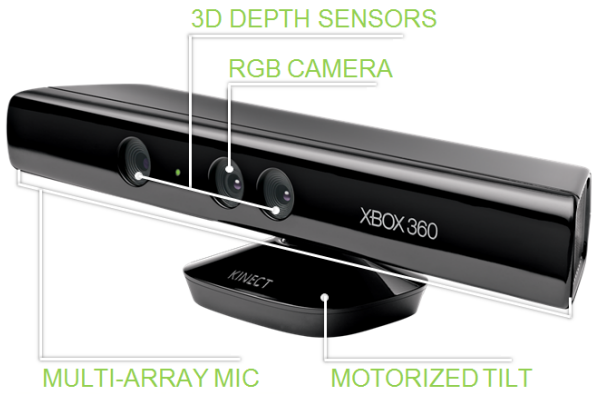

Figure 7: Kinect sensors. This picture highlight all the sensors that are part of the Kinect, like the 3D depth camera (*IR* projector and *IR* camera), the *RGB* camera, the multi-array microphone and motorized tilt.

Table 1: Kinect Sensor Array Specifications[23]

Sensor item	Specification range
Viewing angle	43° vertical by 57° horizontal field of view
Mechanized tilt range (vertical)	±28°
Frame rate (depth and color stream)	30 frames per second (*FPS*)
Resolution, depth stream	*QVGA* (320×240)
Resolution, color stream	*VGA* (640×480)
Audio format	16-kHz, 16-bit mono pulse code modulation (*PCM*)
Audio input characteristics	A four-microphone array with 24-bit analog-to-digital converter (*ADC*) and Kinect-resident signal processing such as acoustic echo cancellation and noise suppression

If we want to give a definition on what *Kinect* is exactly, we can say it is an *infrared* structured light *3D scanner*. The PrimeSense diagram illustrated in Figure 8 explain how their reference platform works. The *IR* transmitter sends out infrared *structured light* carrying encoded information in the form of varying *light patterns*, a *speckle pattern*. When its *IR* sensor detects the returned signal, it looks for deformations in the encoded pattern and the correlation against a *reference pattern* at a known depth. From these changes it does calculations for determining depth information and various characteristics of the objects and people that are present in the *Kinect* detection field [29][12].

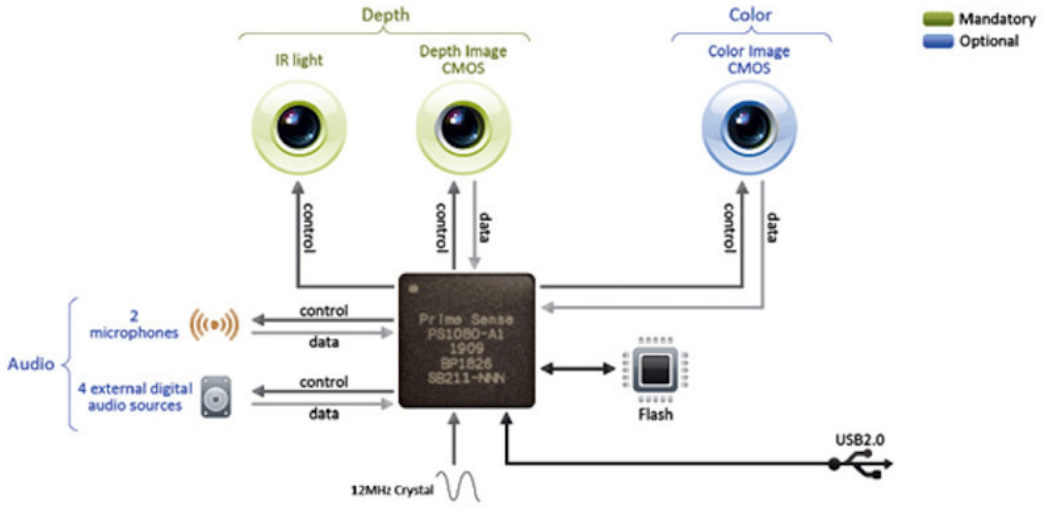

Figure 8: PrimeSense functioning diagram. This diagram illustrates how the PrimSense platform works, in particular the interaction between the different sensors composing the device. As we previously said *Kinect* includes this kind of technology.

If we look at Figure 9 we better understand how the *Kinect* perform depth detection. In the picture you can see the invisible pattern composed by the set of speckle projected onto the environment by the *IR* projector and detected by the *IR* receiver which builds the correspondent 3D model of the scene analyzed as illustrated in Figure 11.

Another important issue to take care examining the hardware are the power requirements. The *Kinect* requires 12 volts *Direct Current (DC)* to operate; it consumes 12 Watts of power during operation. This is far greater than the 5 volts and 2.5 Watts designed into the *Universal Serial Bus (USB)* port in most home computers. This means the *Kinect* has an additional transformer type *wall-plug power supply* as showed in Figure 10.

Figure 9: Kinect infrared light pattern. In this Figure is represented the light pattern projected by the *Kinect* in the environment to perform depth detection.

Figure 10: Kinect power supply. As you can see the *Kinect* power supply has a specific *USB* adapter (female) where to plug the *Kinect* and an additional male *USB* plug that you have to insert in the PC.

2.1.4 Operating modes

In this section we will try to explain broadly how *Kinect* recognise and detect *skeleton information*, *gestures* and *vocal commands* using the different components is equipped. We don't want to go into much technical details on the functioning of each technology involved because this isn't the target of this thesis. We already mentioned in the last section some technical details regarding the functioning of the device, like the fact that is based on an *IR structured light* technology. This kind of technology uses specially designed light patterns projected onto objects to collect 3D data about objects and people present in the environment. Hence, we can claim the fact *Kinect* is to all intents and purposes a *depth camera*. To better understand how the device can detect objects and people, we make a comparison with *normal cameras*. They build and image transforming the light bounces on objects in the colours that you see. A depth camera uses *IR* light to build the same image but in this case each the colour of the pixels identifies how far is this part of the image from the camera itself as illustrated in Figure11. We can define the image built from the camera a *depth image*, the information contained are more accurate and precise with respect to a conventional color image, hence more sophisticated data analysis can be conducted offering the possibility to *track the body* of individual people.

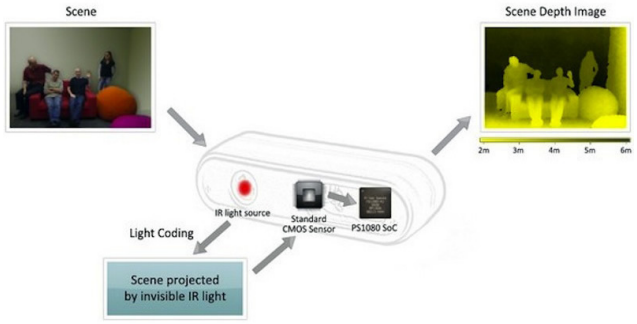

Figure 11: Kinect depth detection.. This picture indicates how a scene depth image can be created observing a room with some people inside.

Speaking about *body tracking*, *Kinect* is not limited on simple body detection but enables using some dedicated frameworks to locate *individual joints* and different body part. This is one of the most exciting capability the *Kinect* offers to developers to create new interactive applications. In Table 2 is defined the *playable range* where the device is able to perform a good *skeletal tracking*. In Figure 12 is illustrated an example of skeletal tracking using *OpenNI*. We will see through this thesis how the skeleton tacking can be used to change the way the user interact with the computer.

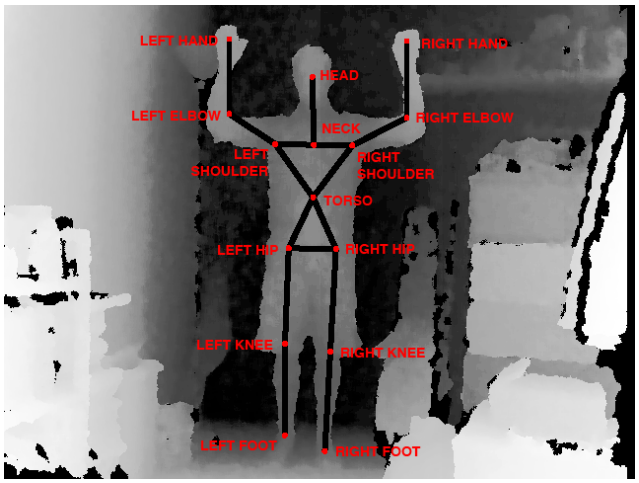

Figure 12: Kinect skeletal tracking.. This figure shows a possible example of skeleton tracking using *OpenNI*.

Table 2: Playable Ranges for the Kinect

Sensor item	Playable range
Color and depth stream	4 to 11.5 feet (1.2 to 3.5 meters)
Skeletal tracking	4 to 11.5 feet (1.2 to 3.5 meters)

Kinect is a complete device because it offers also *audio features*, like *acoustic source localization*, *ambient noise suppression* and using Microsoft *Kinect SDK* a simple form of *vocal command recognition*. The *four-microphone array* is positioned bottom the bar and simulate the behaviour of the ears. Having different acquisition points for the sound is possible to understand the direction of the sound source analysing the different instants when the signal arrive at each microphone in the array. Another positive improvement obtained using a microphone arrays with respect to a single microphone is the fact that it can support more effective *noise reduction* and *Automatic Echo Cancellation (AEC) algorithms*. In this thesis we focus our attention on body detection, hence the audio part is not analysed in details but we made some simple audio tests in laboratory to comprehend the potentiality of vocal commands detection when we examined the Microsoft *Kinect SDK*.

Microsoft added the *motor pivot* to the *Kinect* in order to guarantee a good flexibility and adaptivity in the different rooms and environments. The motor and gears can *tilt* the device up and down with a range of 28°. Like in the case of audio features we not explore this functionality in this thesis because the library we chose to perform skeletral tracking doesn't provide the moving features for the device.

2.2 NATURAL INTERACTION

2.2.1 Introduction

NI refers to a concept whereby the *interaction* with devices is based on *human senses* [17]. These new paradigms render external peripherals currently used today, such as remote controls, keypads or a mouse obsolete. The *Kinect* can be defined a *Natural Interaction (NI)* device, which operates in the context of *Natural User Interface (NUI)*, a new emerging metaphysical paradigm in *Human Computer Interaction (HCI)*. In 2006 *Christian Moore* founded a research community with the aim of learning and sharing new *HCI* methods and concepts defined as *NUI* technologies. The *NI* paradigm can be identified as particular instance of *NUI* because this field of research covers a lot of other different paradigms, pattern and technologies. As descripted in [6], *NUI* is the next step in the evolution of the interfaces following the objective of create more organic interfaces which are based on more *traditional human interaction paradigms* such as touch, vision, speech and most importantly creativity. The main goal of the *NI* paradigm is to transform the legacy interfaces and devices used during the last years of technology evolution in a more intuitive and naturally interaction without the need to wear particular devices or learn complicated instructions to use them.

Figure 13: *HCI* evolution chart. This figure represents the evolution of the *HCI* paradigms in the years.

As we previously mention *NI* is more focused on a set of particular type of interaction and *NUI* approaches:

BODY MOTION TRACKING the analysis and interpretation of 3D data models enables full body motion tracking and skeleton recognition locating to locate individual joints and the different body parts;

HAND GESTURES RECOGNITION where pre-defined hand gestures models are recognized and used to enable a new way to control devices;

SPEECH AND COMMAND RECOGNITION vocal commands trigger action execution on the devices and can be integrated with the other paradigms of interaction augmenting the set of possible commands that can be recognized and executed.

In this chapter we will understand and examine all the aspects related to this new paradigm of *HCI* through the analysis of the different types of interaction involved and all the problems follows. We will try to define what a gesture can represent and how we can recognize it.

2.2.2 Gesture

This dissertation [30] regarding *NI* defines gestures citing Turk as an expressive, meaningful body motion with the intent to convey information or interact with the surroundings environment. In [11] is defined a taxonomy to recognize the various possible type of gestures: gesticulation, manipulations, sign language gestures, deictic and semaphores. These last two forms of gesture taxonomy are particularly meaningful from our domain of application because they well describe the type of interaction the user can have with an *IWB* based on *Kinect* technology. *Gesticulation* is as one of the most natural forms of gesturing and it defines the kind of gestures that accompany conversational speech. This kind of gestures does not require the user to perform any poses or to learn any gestures like it happens for semaphores where pre-defined and recorded gestures are recognized. *Manipulative gestures* are another form of interaction we have to consider because it is the most used actually to interact with computers, in fact it involves the use of the mouse or stylus to move a cursor or manipulate a window. In a 3-dimensional environment manipulative gestures are applied to physical objects which represent digital objects on the screen. *Deitic gestures* are pointing actions that refer to objects in the context of the application domain like can be a user pointing on a large screen in order to identify, locate or maybe move a specific object. The *semaphore* pattern defines stylized static and dynamic gestures like can be assuming a particular pose or waving a hand. We will discover in the next chapters why this kind of gesture paradigm is particularly important to interact with the *Kinect*.

2.2.3 Gesture recognition

Perform *gesture recognition* is not an easy task because there is a problem of disambiguation or rather is not so easy build effective heuristics or machine learning algorithms that are able to understand what is a specific gesture from everything else. Gesture analysis should isolate *invariants* in gesture characteristics to start

build a possible *pattern* to recognize a particular gesture. In some cases psycholinguistic studies are necessary to understand the context in which gestures are performed. The gestures selected to interact and gain the control of the system should *not too similar* otherwise they require extremely precise motion, difficult to be replicated with a total loss of the intuitive and natural components that a gesture can have. If we consider instead the other side of the same medal there's a clear necessity to identify some patterns and invariant to perfectly identify a gesture, hence we have the need to give back to the user a precise *feedback* to guide him or her in the process of disambiguation. As [30] suggests the fact that gestures must be *simple* and *intuitive* motions, easy to replicate not increasing the cognitive load of the subject. Give a real-time feedback of the user on the screen establish a *visual control loop* to create the right feedback enabling the previously mentioned disambiguation process. We will see in details in 4.6.1 a possible instance of this type of *feedback and visual loop*.

2.2.4 Body Tracking

The role of *body motion* as described in [30] is forgotten because the devices and sensors that a user has to wear reduce heavily the pleasure of moving into space. *Skeletal tracking* enables a more intuitive and naturally interaction without the need to wear particular devices or learn complicated instructions to use them. Think about all the possibilities of application in the medical field for example to help people during rehabilitation or like in our case in the educational sector such as described in [10] where *Kinect* as a *NI* interface which provides *full body tracking* is defined as an interactive technology that can *enhance teaching and learning* increasing the student participation and engagement in classroom activities creating proper pedagogical opportunities for students to interact with content knowledge. This feature offered by the *Kinect* can provide natural and intuitive interaction patterns and as suggested in pedagogical strategies it can encourage student participation eliminating every interaction barrier a complex device can introduce.

2.2.5 Speech Recognition

As argued [30] the audio component of the interaction can be an important source of information. The *speech recognition* process can have troubles in a noisy environments, such as can happen in a crowded room during a university lesson, and often requires microphones carried close to the speaker's mouth. In order to deal with not optimal situations, two main approaches can be implemented: the first is the use of an *array of microphones* where audio devices contain a set of dif-

ferent microphones and a digital signal processor that allows to perform *beam formation* understanding where is a specific audio source, eliminating the other ones with the help of algortims of *acoustic noise suppression* and *echo cancellation*. As described in 2.1.3 its functioning is based on the relative latencies between signals taken from different microphones. The second proposed solution consists in the use of a *directional microphone* oriented towards the user. Audio analysis becomes a powerful tool specially when it accompanies computer vision like happen for the *Kinect*: the combination of both can enhance a lot the quantity of command can be recognized or better can make the detection more reliable improving the disambiguation process described in 2.2.3.

2.3 INTERACTIVE WHITEBOARDS

2.3.1 Introduction

An *Interactive Whiteboard (IWB)*, is a very powerful tool designed and thought to provide quality education to the students. It is made up of a *large interactive board surface* that usually connects to a *computer* and a *projector*. The projector transfers the computer's desktop onto the display's surface and through the use of either special pens or even fingertips the users control the computer. Typically *IWB*s are used in classrooms, in the business world or anywhere people need to make presentations or conduct a lesson mounted to a wall or floor stand[32].

Figure 14: *Interactive Whiteboard (IWB)*. In this figure is presented a possible example of a conventional *IWB*.

In this chapter we will describe and report the *different technologies* are the bases of the various solution proposed. Once we will have a clear overview we will

make a focus on *two interesting research project* created at the University of Trento to develop low-cost whiteboards. These projects are the source of inspiration for this thesis and optimal tools of comparison.

2.3.2 Technology

*IWB*s may based on different type of several types of sensing technology to track the interaction with the screen surface:

RESISTIVE Resistive whiteboards are composed of two flexible sheets of resistive material separated by a small air gap. This touch-sensitive membrane is used to detect where a student or teacher touches the board. When contact is made to the surface of the touchscreen, the two sheets are pressed together, registering the precise location of the touch that is used as input to the *IWB* software. This technology allows one to use a finger, a special stylus, or any other pointing device on the surface of the board[32][28].

ELECTROMAGNETIC (PASSIVE OR ACTIVE) These interactive boards feature an array of wires embedded in the whiteboard surface interacts with a stylus tip determining the pen coordinates with respect to the board surface. The stylus can be passive, it not contains batteries, it simply alters electrical signals produced by the board or active, the pen contains an electromagnetic coil formed by tiny magnetic fibers that implements the sensing technology. In other words, the magnetic stylus activates the sensors in the board that react and send a message back to the computer. For example, when the stylus is pressed on the board, the board can signal a mouse click to the computer[32][28].

CAPACITIVE The operating principle is the same as the electromagnetic type, the capacitive type works with an array of wires embedded or behind the board. In this particular case however the wires interact directly with the fingers touching the screen determining the different coordinates when the finger tips the board[32].

OPTICAL The whiteboard based on optical technologies can be categorized considering the different type of light involved or how the light interact with the board to detect the finger or pointer position:

INFRARED LIGHT This kind of optical board is based on attaching *IR* scanning devices to an existing ordinary whiteboard or flat surface. It uses an *IR* light and a triangulation algorithm to determine the location of a stylus on the board. These scanning devices are light and portable and can be used with different types and sizes of ordinary whiteboards[32].

LASER LIGHT The operating principle in this case is very similar to *IR* scanning devices but in this case the *IR* lasers are directly mounted in the top corners of the board that detect the marker movement. They require the use of special passive reflective pens and not emitter like in other cases, each of them has a unique encoded reflective collar used to reflect the laser beam back to the source in a way that stylus position can be easily triangulated[32][28].

FTIR *Frustrated Total Internal Reflection (FTIR)* is a simple technique enables to build robust boards at a minimum of engineering effort and expense, in fact, this technology is based on the fact that pressing with a finger on a surface where *IR* light bounces causes a disruption in the internal reflection detected by cameras and transformed in coordinates [7][32].

CAMERA PEN AND DOT PATTERN This technology is licensed and proprietary better known as "*Anoto*" technology. The *IWB*s based on this kind of approach have a wireless active stylus which contains an *IR* camera that reads the microscopic dot pattern embedded in the writing surface. The pen is able to recognize the exact point where is on the board uploading the detected coordinates via wireless to a pc. The accuracy is high since the coordinates are usually fixed at about 600 dots per inch[32][2].

WII REMOTE Now we will describe the most inspirational technology related to this thesis. At University of Trento, we will discover in the next sections, there are two main research projects studying and developing solution based on this particular technology to build a low-cost *IWB*[24]. This proposed new way to emulate an Interactive Whiteboard consists in a Wii Remote connected to a computer through its Bluetooth connection capabilities. A specific designed open-source software installed on the computer transform in possible coordinates of the mouse on the screen the *IR* light detected by the Wii Remote which receives a beam projected from an *IR*-Pen, a special pen containing a momentary switch and *IR Light Emitting Diode (LED)* battery-powered, pointed to any surface usually where is projected the computer desktop, like can be a desk, a wall, an existent whiteboard and so on. The Wii Remote has a very accurate *IR* light tracking camera. Once calibrated, the Wii Remote detects a mouse click at the screen location of the stylus. This type of solution was first adapted for use as an *IWB* by *Johnny Chung Lee*[13][32].

DST *Dispersive Signal Technology (DST)* introduced in 2002 by *3M*[1], it determines a touch by measuring the mechanical energy (bending waves) within a substrate created by a finger or stylus touching the surface of the glass, which

is detected by corner-mounted sensors. Piezoelectric sensors positioned in the corners on the backside of the glass convert this smeared mechanical impulse into an electrical signal. The distance from each sensor determines a different smearing in the signal enabling the possibility to individuate the exact source position of the signal. Using advanced digital signal processing and proprietary algorithms, we are able to apply dispersion correction algorithms which analyze the signals and report an accurate touch[32].

ULTRASONIC We can have two type of ultrasonic whiteboard: the first using an active pen emitter where ultrasonic sound is emitted by the stylus whenever it is pressed against the board's surface. Microphones located at different positions along the board pick up the sound and triangulate the position[27]. Sometimes this approach can be combined using in addition an *IR* light. The second one using a totally different approach using a passive stylus and two ultrasonic transmitters in two corners and two receivers in the other two corners. The ultrasonic waves are transmitted by the surface of the whiteboard. Very small marks in the whiteboard borders generate reflecting waves for each ultrasonic transmitter at different and recognizable distances. When the user touch the board with a finger or a pen it causes a suppression of these point waves, this fact is detected and the receivers communicate what happen to the controller.

2.3.3 WiiLD

In this section we examine one of the technologies relative to the *IWB* implementation mentioned before which can be a forerunner in the introduction of *gaming devices* as a concrete tool to *support teaching and learning activities*.

Wiimote Lavagna Digitale (WiiLD) is a low cost technological tool with huge potentialities. *Johnny Chung Lee*[6] demonstrated in 2007 how is possible to emulate an *IWB* using a *Wii Remote*, the controller of the famous Nintendo's home console called Wii [13][24].

The fundamental elements composing the *WiiLD* control system are:

- a PC with an active Bluetooth connection;
- a projector;
- a WiiMote, the Wii's primary controller;
- an *IR* pen, it is a special pen with a momentary switch, power source and an *IR* emitter;

6 Johnny Chung Lee is a visionary *HCI* researcher currently working at Google. He became very famous in 2007 for his *Wii Remote head tracking project* [13]. He was named one of the world's top 35 innovators under 35 in 2008.

- a surface that can be turned into an *IWB*;
- a tracking software.

It is possible to use a sensor bar and a second WiiMote, which has the role of pointing device.

Operating mode

As described in [24] the *WiiLD* operating mode is very simple, in practice we obtain *computer's desktop projection* onto the surface where users control the *computer* using the *IR pen*. The stylus positioned in the proximity of the surface emits the *IR* light, imperceptible to the human eye but detectable using the *Wii Remote* controller as receiver of the *IR* light. The latter transmits the detected signal to the computer using the Bluetooth connection as illustrated in Figure 15. The software running on the *Personal Computer (PC)* receive the transmitted data by the *IR* receiver and perform some computation, for example positioning the mouse cursor in the detected position with respect to the screen. A preliminary phase of calibration enables to map the physical projection space with the logic position on the desktop (the visible screen portion). In this way it is possible to interact with the *PC* using and moving the optical *IR* pen into the projection space, obtaining a new example of *IWB*. The commercial version of *IWB* are sensible devices creating the same mapping operation between a recorded position on the board and a virtual position on the desktop of the computer.

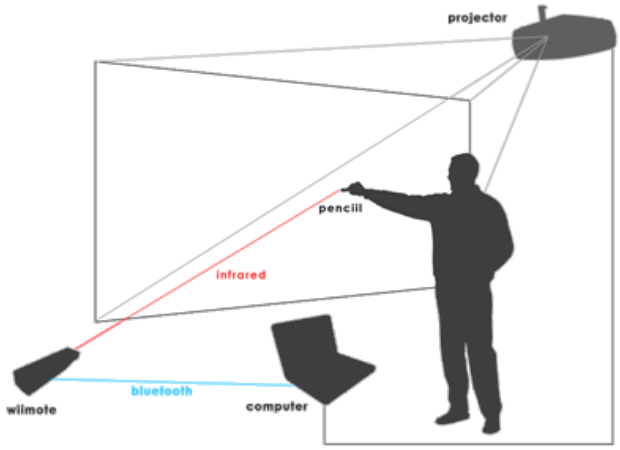

Figure 15: *WiiLD* operating mode schema. In this picture you can see the logical operating mode of the *IWB* solution proposed using the Wii Remote technology.

The technologies used inside commercial *IWB*s are completely different and are based on various models as we argued in the previous section describing the *IWB*'s technologies. The pointing device used in this case can be a specific

designed pointer, a finger or simply a passive device. The *software* associated with the device is a *key component* because it has a fundamental role, in fact it guarantees the *position detection* and the relative mapping, but it can also add additional resources to teachers offering annotation and writing capabilities on the desktop or a system for preparation and filing of the lessons or whatever other specific tools to support the different educational activities.

This project is already a reality in the Italian school system, especially in the province of Trento because a lot of school are searching some possible solution to adopt as many as possible low-cost *IWB*s. At the University of Trento exists an apposite project, the "*Wii4dida*"[7] project, designed to introduce the experimentation and the real utilization of the *WiiLD*.

2.3.4 WiiLDOS

WiiLDOS[8] is a *free operating system* created and designed to *enhance* and facilitate the *utilization* of the *Wii Remote agile whiteboard*. This is not a simple software but it is a complete free GNU Linux distribution live or installable. It is conceived to be used in combination with *IWB*s or Tablet Pcs for educational purpose and activities. *WiiLDOS* is a new operating system based on Ubuntu[9] distribution, it offers a modular, complete and simple approach, it is free and open-source and available for all the users and teachers want to change and update the way they conduct their lessons using as base an *IWB* including an entire suits of programs conceived to this particular purpose. This distribution is designed to provide a *set of software dedicated to teaching activities* and offer a better support to tools like the *WiiLD* or *Kinect* because it proposes a emphsimplified interface and *intuitive commands* and actions that can enhance the integration with technologies that doesn't have in the *precision* and *complexity* their point of strength[18].

History

WiiLDOS is a Linux distribution derived from Lubuntu designed and developed starting from the third quarter of 2010 by *Pietro Pilolli*, a software developer and researcher within the *Service Oriented Architecture (SOA)* Research Unit at *Fondazione Bruno Kessler (FBK)*[10] in Trento, Italy. The first version was released in December 2010 based on Lubuntu 10.04, the latest stable is the version 1.04 published in June 2011, and the latest alpha is the version 1.07 released in November 2011.

7 http://sites.google.com/site/wii4dida/
8 http://sites.google.com/site/wiilavagnadigitale/wiildos
9 http://www.ubuntu.com/
10 http://www.fbk.eu/

Figure 16: *WiiLDOS* GNU/Linux Distribtuion. This figure show the *WiiLDOS* install screen.

Usability

The interface proposed by *WiiLDOS* showed in Figure 17 is subdivided in *tabs* containing *large icons* to facilitate the usage as a "whiteboard". Performing a *single click* on the top bar is possible to choose a tab and inside it select a particular program as we previously highlight the fact that this operative system is focused on an utilization by devices with a *simple usability* and *poor precision*. If you want to insert some text you can use the *Florence virtual keyboard*[11], *Dasher*[12] or *Eviacam*[13] using your head. You can utilize the panel for the handwriting recognition called *Cellwriter*[14][18].

Portability

WiiLDOS is free software, hence it can be copied or redistributed without restriction to the students. Each student can install *WiiLDOS* on a *USB* flash drive, and thanks to the persistent modality it is possible to *personalize* the distribution adding additional software and programs using the Ubuntu packet manager called Synaptic[15] and *save data and modification* inside the flash drive. In this way the student can use the same software utilized at school and saving on the same *USB* key the work and production made at home or at school. This specific features offered *WiiLDOS* is compatible with the portability provided by the WiiLD tool allowing an easy transportation and utilization everywhere[18].

11 http://florence.sourceforge.net
12 http://www.inference.phy.cam.ac.uk/dasher/
13 http://eviacam.sourceforge.net/eviacam.php
14 http://risujin.org/cellwriter/
15 https://help.ubuntu.com/community/SynapticHowto

Figure 17: *WiiLDOS* Interface. In this pictures is illustrated the *WiiLDOS* interfaces showing tabs and programs, in addition you can see the Florence virtual keyboard.

Lightness

As reported in [18] *WiiLDOS* propose an implementation and development oriented to save as mush as possible computational resources in order to be used on old computers, in fact the minimal system requirements requested are:

- Pentium 2 or equivalent;
- 128 MB of *Random Access Memory (RAM)*;
- 2 GB of disk space;
- a graphic card and a monitor with resolution at least 800×600.

Of course, the developer suggests to have a more reactive system and better response times these specific characteristics:

- Pentium 3 or equivalent;
- 512 MB of *RAM*;
- 4 GB of disk space;
- a graphic card and a monitor with resolution at least 1024×768.

We will discover in the next sections the minimal and suggested system requirements needed to run *Kinect* based application are unfortunately much more than this.

Licensing

This modified operating system derived from Lubuntu is *freely usable, copyable and redistributable* for a non commercial utilization. For a different purpose of utilization a write authorization from the authors is needed containing a double license for commercial use.

3 KINECT FRAMEWORKS

3.1 OPENNI

3.1.1 Introduction

Open Natural Interaction (OpenNI)[1] is an industry-led, not-for-profit organization founded in November 2010 by:

PRIMESENSE An Israeli company founded in 2005, it is and industry leader leader in *NI* and 3D depth sensing solutions. It provides the 3D sensing technology used in the Microsoft *Kinect*.[2]

WILLOW GARAGE *Willow Garage* is a robotics research lab. It develops hardware and open source software for personal robotics solutions.[3]

SIDE-KICK This company is a leading game developer and publisher specializing in motion games that use human gestures as the game controllers.[4]

ASUS *ASUS* joins the *OpenNI* organization as an industry member providing hardware for purchase to promote *NI* applications. ASUS produce a *Kinect*-like device called *WAVI Xtion*.[5]

APPSIDE Founded in 2011, AppSide is the first end-to-end content marketplace for motion-controlled entertainment devices.[6]

The main aim of the organization is to certify and promote the *compatibility* and *interoperability* of *NI* devices, applications and middleware accelerating the introduction of *NI* applications into the marketplace. As a first step on the way to achieve the goal, the organization has made available a *multi-language, cross-platform open source framework*. This framework defines a set of interfaces for writing applications utilizing *Natural Interaction (NI)*. The main purpose of the framework as reported in [17] is to offer a *standard Application Programming Interface (API)* that enables communication with the both hardware and software.

1 http://www.openni.org
2 http://www.primesense.com/
3 http://www.willowgarage.com/
4 http://www.sidekick.co.il/
5 http://www.asus.com/
6 http://www.app-side.com/

OpenNI's *API* provide a set of *API* to be implemented by the various sensors and another set of *API* for the other side that is implemented by the middleware components: the software components that analyze all the data collected from the scene and elaborate it. For example, in this specific case *NITE* frameworks receive the data collected by the *Kinect* and elaborating it performing *skeletral tracking* of the user 3D model. *OpenNI* can be represented as you can see in Figure 18 using a *three-layered view*. The first layer on top represents the software that use the *API*s offered by the framework. The middle layer represents *OpenNI* itself, providing communication interfaces for both the sensors and the middleware components. The bottom layer shows the hardware devices that captures and acquire data observing the environment.

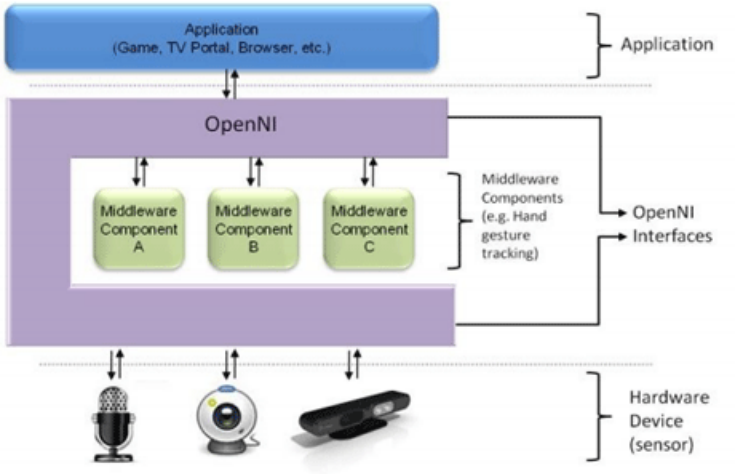

Figure 18: *OpenNI* Architecture. The OpenNI Framework is an abstract layer that provides the interface for both physical devices and middleware components.

OpenNI is designed and built according with the philosophy "*write once, deploy everywhere*". The level of abstraction offered by this kind of architecture enables applications to be written and ported with no additional effort to operate on top of different middleware modules and sensor types. Using *OpenNI* is possible to write applications which use informations regardless of which sensor device has produced them, and vice versa offers the possibility to built new sensors that power any *OpenNI* compliant application.

3.1.2 Modules

One strength point of the framework that provides good flexibility and adaptivity is the *modular architecture* design, in fact the *API* enables the registration of multiple components in the framework. These components are referred to as *modules*, they can produce data or consume and process data generated by other

components. Hence, you can combine the modules in different way creating new interaction patterns. The current supported sensor modules are:

- 3D sensor;
- RGB camera;
- *IR* camera;
- Audio device (a microphone or an array of microphones).

The current supported middleware modules are:

FULL BODY ANALYSIS MIDDLEWARE A software component that processes data generated by the different sensors and creates body model structures like data describing joints, orientation, center of mass, and so on.

HAND POINT ANALYSIS MIDDLEWARE A software module that processes sensory data and elaborates the location of a hand point.

GESTURE DETECTION MIDDLEWARE A middleware component that identifies predefined gestures (for example, a waving hand or a push) and throws specific events to alert the application.

SCENE ANALYZER MIDDLEWARE A structured component that analyzes the environment in order to produce information as:

- the separation between the figures in the foreground of the scene and the background;
- the coordinates of the floor plane;
- the individual identification of figures in the scene.

3.1.3 Production nodes

OpenNI defines a *production node* as described in [17]. It has the key role of *producing* and *generating meaningful 3D data* for *NI*-based applications, which can be a complex task. Typically, this begins by having a sensor device that produces is a *depth map*, a simple form of raw data, where each pixel is represented by its distance from the sensor. Specific dedicated middleware software elaborates this raw data, and produces a higher-level output, which can be understood and used by the *NI* applications. Each *production node* implements the logic of the functionality that relates to the generation of the specific data type. The *API* related to the *production node*s only defines the interface provided for the applications.

In case when for example the application wants to track the motion of a human figure in a 3D scene as illustrated in Figure 12. This requires a *user generator*

that provides body data for the application. This specific *production node* obtains its data from another *production node*, a *depth generator*. A depth generator is implemented by a sensor, which takes raw sensory data from a depth sensor (for example, a stream of X frames per second) and outputs a depth map. In principal, each *production node* is a standalone unit that generates a specific type of data, and can provide it to other *production node*, or directly to the application itself. Usually, *production node*s always use determined production nodes to analyse lower level data types, using it to produce higher level data for the application.

We can have different production nodes types as we mentioned above: sensor-related and middleware-related production nodes. Below are reported some of the currently sensor-related production nodes supported in *OpenNI*:

DEVICE A node that represents a physical device (for example, a depth sensor, or an RGB camera). The main role of this node is to enable the possibility to configure the device.

DEPTH GENERATOR This node should be implemented by any 3D sensor that wishes to be certified as *OpenNI* compliant and has the capability to generate a depth-map.

IMAGE GENERATOR This kind of generator identifies a node which produce coloured image-maps. This node should be implemented by any color sensor that wishes to be certified as *OpenNI* compliant.

IR GENERATOR Any *IR* sensor that wishes to be certified as *OpenNI* compliant should implement this production node.

AUDIO GENERATOR A node that generates an audio stream. This node should be implemented by any audio device that wishes to be certified as *OpenNI* compliant.

The middleware-related production nodes which can be registered within the framework are:

GESTURES ALERT GENERATOR Generates callbacks to the application when specific gestures are identified.

SCENE ANALYZER Analyses a scene, including the separation of the foreground from the background, identification of figures in the scene, and detection of the floor plane. This type of node produces as output a labelled depth map, in which each pixel holds a label that states whether it represents a figure, or it is part of the background.

HAND POINT GENERATOR It supports hand detection and tracking. This node generates provide alerts when a hand point (meaning, a palm) is detected, and when a hand point currently being tracked, changes its location.

USER GENERATOR This is one of the most important generator because it elaborates a representation of a full or partial body in the 3D scene.

For recording purposes, the following production node types are supported: recorder, player and codec.

3.1.4 Production chains

In previous section we described the capability of *OpenNI* to defines production nodes that interact to produce high-level data consumed by the application. This kind of interaction can be defines as *production chain*. In the example we made talking about skeletal tracking, the user generator is reliant on the depth generator in order to produce the required body data. Hence, *OpenNI* creates the possibility of build different *production chain*s composing various implementation of the same production node offered by different vendors on the market, like depicted in Figure 19.

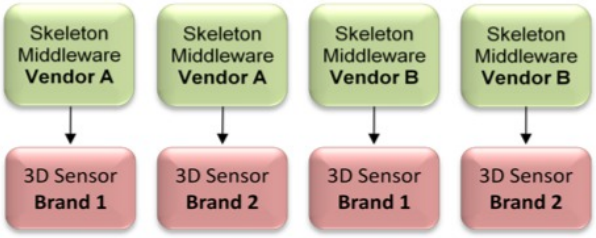

Figure 19: *OpenNI* Production Chains. This illustration displays four optional production chains.

3.1.5 Capabilities

OpenNI supports the flexibility of the registration of multiple middleware components and devices. Non-mandatory extensions are defined by the OpenNI *API*. These optional extensions to the *API* are called *capabilities* and reveal additional functionality, enabling providers to decide individually if a production node can supports a specific capability. A specific set of capabilities is released, with the option of adding further capabilities in the future. The application can specify the capabilities that should be supported by the modules composing the chain as criteria when enumeration of the various production chain is requested. Only modules that support the requested capability are returned by the enumeration.

The currently supported capabilities are:

ALTERNATIVE VIEW This capability enables any type of map generator (depth, image, *IR*) to transform its data to appear as if the sensor is placed in another location.

CROPPING Cropping can be very useful capability for performance boosting. It enables a map generator (depth, image, *IR*) to output a selected area of the frame as opposed to the entire frame. When cropping is enabled, the size of the generated map is reduced to fit a lower resolution (less pixels). For example, if the map generator is working in *Video Graphics Array (VGA)* resolution (640×480) and the application chooses to crop at 300×200, the next pixel row will begin after 300 pixels.

FRAME SYNC It allows frame synchronization of two different sensor to permit their frames arrive at the same time.

MIRROR Mirroring is one of the most used capability, also implement by *Kinect*. Mirroring is useful if the sensor is placed in front of the user, as the image captured by the sensor is mirrored, so the right hand appears as the left hand of the mirrored figure.

POSE DETECTION To start user skeleton calibration the user is posed in a specific position.

SKELETON This capability is the most important for this thesis because is the core of the entire application developed. It enables a user generator to output the skeletal data of the user. This data includes the location of the skeletal joints, the ability to track skeleton positions and the user calibration capabilities.

USER POSITION It allows to detect the user position enabling a depth generator to optimize the output depth map that is generated for a specific area of the scene.

ERROR STATE A capability used for error status reporting, meaning that on a practical level, the node may not function properly.

LOCK AWARE It enables a production node to be locked outside the context boundary.

3.1.6 Licensing

OpenNI is written and distributed under the *Lesser General Public License (LGPL)* which means that its source code is *freely-distributed* and available to the general public. You can redistribute it and/or modify it under the terms of the *LGPL* license as published by the Free Software Foundation, either version 3 of the License, or any later version.

Modules and applications

OpenNI provides a simple *licensing mechanism* that can be used by modules and applications to ensure that modules are only used by authorized applications. It also provides a global registry for license keys, which are loaded whenever a context is initialized. This means that a module is accessible only if the license is provided by the application using the module. A license is composed of a vendor name and a license key.

3.2 NITE

3.2.1 Introduction

Natural Interaction Technology for End-user (NITE) is the middleware that *translates the world in 3D* represented using depth maps into *meaningful data*. The *Kinect* sensor observes the scene, while *NITE* acts as the *detection engine* that is able to tell apart the user body from the surroundings environment capturing eventual interaction. The framework offers different features like *user identification* and *movement detection*, as well as a set of framework *API*s for implementing *NI* controls that are based on *user gestures*. *NITE* works over *OpenNI* providing software implementations for all the *OpenNI* module referred as *"NITE algorithms"*, this computer vision middleware layer is explained in details in the next subsections [21][20].

The framework stack is composed by different layers like depicted in Figure 20:

- starting form the bottom, we have the *OpenNI* infrastructure with all its interfaces on the both side with sensor's driver and with the middleware software. *OpenNI* supports the registration of different modules, *NITE* implements *gesture* and *hand generator*, *scene analyzer* and *user generator* performing *skeletal recogniton*;
- the second layer is composed by the previously mentioned *NITE Algorithms*, which is a set of different computer vision algorithms which are able to process the depth data transforming it in meaningful informations and models that the *NITE* controls can understand;
- the next level is represented by the *NITE* controls, an application framework that provides gesture identification and UI controls to enable application writers to design new application following the flow according with informations, event and action generated by the *NITE* Algorithms.
- the layer on top of the depth data processing stack is composed by the *NI*-based application. This application can receive callbacks from *NITE* controls

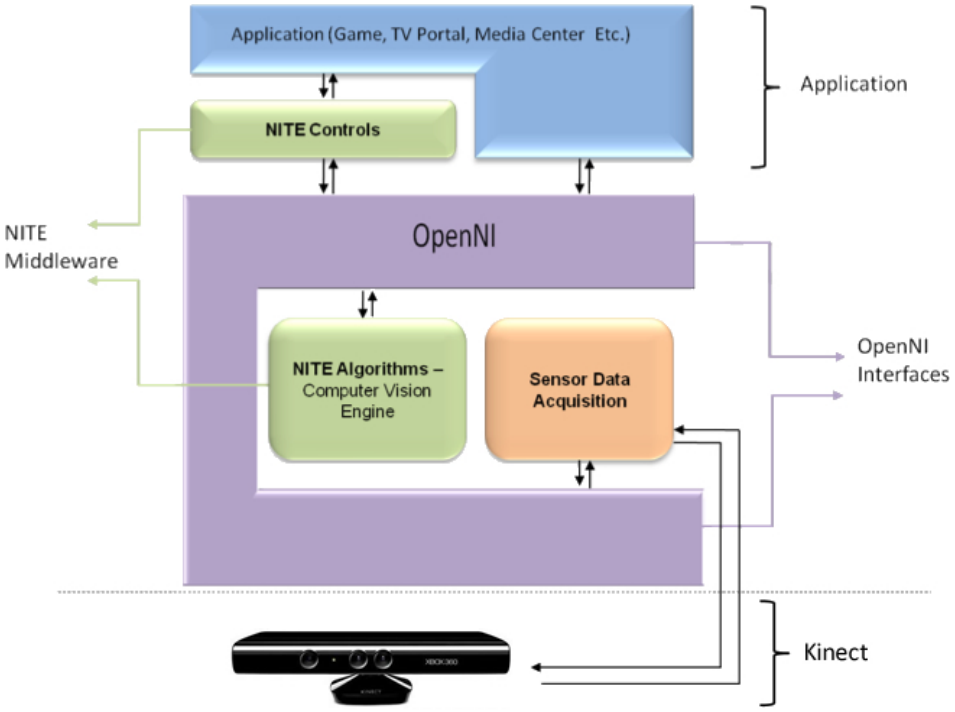

Figure 20: Nite Layered View. A simply overview of the depth data acquiring and elaboration process.

or interact directly with *OpenNI* to have access to the data produced by the sensor.

3.2.2 NITE Algorithms

NITE Algorithms represented the first layer of data process elaboration composing the *NITE* framework stack. This middleware layer use different computer vision algorithms to process the stream of raw data coming from the device driver through *OpenNI* architecture. These algorithms are able to segment the scene separating the users from the surroundings environment and objects, detect hand points and track specific pre-defined gestures and lastly perform a full body tracking to examining the current user pose highlighting the different joints forming the human body[19][20].

Gesture Control Algorithms

As described in these notes [19] you can gain the control using *specific predefined gestures* inside the field of view of the *Kinect*. To start you need to perform a special gesture called *focus gesture*. A *focus gesture* can be a *wave gesture* or a *click*

gesture. In order to perform a *wave gesture* is needed to change the hand direction suddenly from left to right and viceversa within a timeout, however, in order to perform a *click gesture* is needed to push the hand towards in the direction of the sensor and immediately pull his or her hand back. Following the suggestions reported in [19] you can gain the control until you release the control according to the application flow leaving a pre-defined space or when the hand point or the user figure is lost. Different type of gestures can be performed as we have previously mentioned:

- click;
- wave;
- swipe left,right, up and down;
- raise a candidate hand;
- hand candidate moved.

User Segmentation Algorithms

The set of algorithm dedicated to *user segmentation* help the developer to identify and track the user in the scene. The notes [19] indicates that a user can be marked and identified with a persistent ID producing a label map giving the user ID for each pixel composing his figure. The skeleton tracking algorithm start from this label map to generate a body skeleton. If user is lost his ID is freed and a new ID can be assigned to user in a further detection.

Skeleton Tracking Algorithms

Finally we examine one of the most interesting feature offered by the *NITE* framework the *skeleton tracking*. The guide reported in [19] indicates as basic assumption to create an ideal environment where the tracking can be performed in the best way a scene without occlusion by objects or other users and an ideal distance of the body from the device around 2.5 meters. A *calibration process* is used to adjust the skeleton model to the user's body proportions and starts the pose tracking. The user should staying in the pose indicated in Figure 21 facing the *Kinect* for a few seconds, this specific pose is defined in the documentation the "*Psi*" pose.

The body tracking produce as output *positions* and *orientations* of the *skeleton joints*. In [19] is highlighted the fact that exits a clear difference between positions and orientations. Summarizing this concept the tracking model used by the *NITE*'s *API* only consider the joint position without take care of the joint angles because using an angle-driven computation can introduce too much noise with

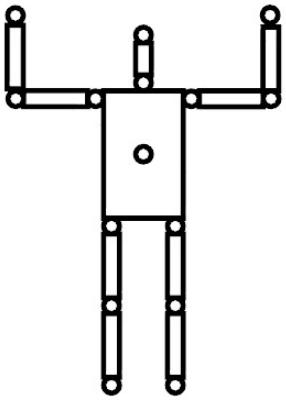

Figure 21: *NITE* Calibration Pose. The figure represented indicates a stylized body pose the user has to assume to start the skeleton calibration procedure.

respect to using the joint position due to the fact we can have a huge variation in the length of the different body segments. The framework in this sense can have difficulties to detect a user "*twist*": the feature extractor in some cases tries to give its best guess. Hand and foot joints orientation is derived from elbow and knee orientation.

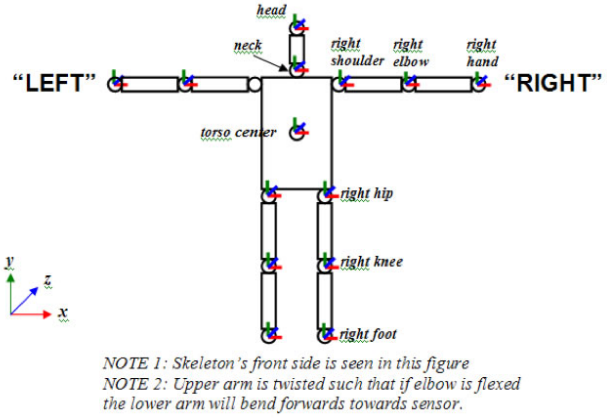

Figure 22: *NITE* Joint Definitions. This figure illustrates the coordinate system and joint definitions used by *NITE* in *mirrored mode*.

The Figure 22 illustrates the coordinate system and skeleton representation for a user facing the sensor. As you can probably imagine the definition of "*left*" and "*right*" side is very important. *NITE* has already predicted this possibility allowing a *mirrored mode* where you see on the screen a *mirror-like reflection* of yourself, hence if you move your right part of the body the right side of the skeleton model on the screen moves as well.

The notes in [19] reports some knowing issues we noted during our test and research phase, we want to mention them because we want to explain clearly the system behaviour because sometimes it seems unpredictable and not so precise as we expected. For example, if you take the arm close to your body the tracking is not stable and both arms can be mixed up or the fact that leg tracking is quite unstable at the moment, the head must be visible, sometimes you need to assume a "*simple*" pose facing the device to resume a stuck in the tracking session or the fact that sometimes fast motion detection may cause a tracking failure. For a complete list see [19].

3.2.3 NITE Control Paradigms

These layer of the framework is a set of *API* that enable developers to implement the flow of the application according to hand points generated by *NITE Algorithms*. *NITE Controls* framework identifies *specific gestures*, and provides a set of *UI controls* that are based on these identified gestures.

Session Management

As already described when we talk about gesture control algorithms in 3.2.2 the frameworks needs to define a time during which the tracking of the hand is enabled, hence a *session* is defined as a state in which the user is in control of the system using his or her hand. The user to start gaining the control needs to perform a pre-defined gesture labelled as *focus gesture*. When this specific gesture is recognized starting form this point the hand of the user is tracked until it leaves the detection field of the *Kinect*. For a more detailed explanations and session state description consult [21].

Point Controls

point control are a set of objects that receive a stream of the current active hand points, analyse it and accordingly to the hand points' behaviour, in the case a *meaningful behaviour* is recognized they perform some action. Any *point control* has some events in common with the others, for example when a point is created, moved, destroyed and so on. Now we will examine each point control trying to highlight what is the meaningful behaviour recognized using this particular *point control* in a way that we can shows all the possible gestures can be detected.

PUSH DETECTOR This point control is designed to recognize a *push gesture* performed by the currently tracked hand point. A *push gesture* is detected when a certain velocity is reached in an angle close to the *Z-axis* for a certain period of

time, in a few word is a push in the direction of the *Kinect* and back again. This gesture is typically used for selecting and picking items [21][20].

SWIPE DETECTOR This point control attempts to recognize hand point motion as a *swipe gesture*, either up, down, left or right. A swipe motion is a *short movement* in a specific direction, followed by the hand resting. For example, during a presentation, swipes to the left and right can be used to move from one slide to the next one [21][20].

STEADY DETECTOR The *steady detector* try to recognise when a hand point has been still standing for some interval of time. *Steady* means that the user doesn't move the hand in any direction, the hand is steady in front of the body. The detection takes into account that a person's hand may still slightly move when held steady. The *steady detector* uses the variance of the hand point within a specific time frame. Steady is mostly useful for the detection of a resting hand between gestures, to clearly signal that one gesture has ended and another one can potentially start [21][20].

WAVE DETECTOR This point control attempts to recognize hand point movement as a *wave gesture* like indicated in Figure 23. A wave is a number of *suddenly direction changes* from left to right and viceversa within a timeout. By default, four direction changes are needed to identify a wave [21][20].

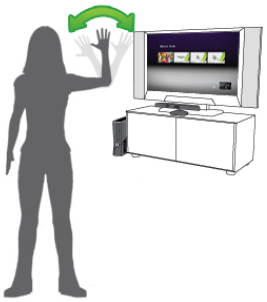

Figure 23: NITE Wave gesture. This Figure illustrates a possible wave gesture.

CIRCLE DETECTOR The circle detector attempts to identify hand point movement as a *circular motion*. This point control needs a *full circle* clockwise or anticlockwise as represented in Figure 24 in order to detect the movement as a circle [21][20].

SELECTABLE SLIDER 1D This point control tries to detect hand point movements along any one of the 3 sliding axes: x (left-right), y (up-down) or z (close-far). This

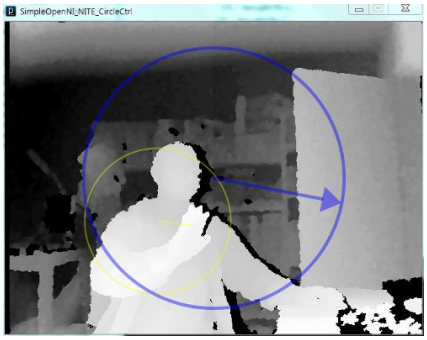

Figure 24: NITE Circle gesture. This Figure illustrates a possible circle gesture performed testing the examples supplied with SimpleOpenNI library, a *OpenNI* and *NITE* wrapper for Processing.

kind of action is defined a *slider*. This point control provides a value between 0 and 1 for each frame indicating where the hand point is relative to the preset ends of the slider. In addition, it is possible to define selectable areas dividing the sliding zone equally into a number of areas, where each area defines a single selectable item in the slider. As you can imagine this solution can be used to implement a sort of menus, with each item being a single menu option. [21][20].

SELECTABLE SLIDER 2D This point control tries to recognize an hand point movement on a predefined X-Y plane. The control provides two values between 0 and 1 for each single frame indicating where the hand point is comprised in both X and Y axes relative to the ends of the slider. This specific point control also enables dividing a virtual plane into equal areas. Each area defines a single selectable item in the virtual plane. The user can select a particular item performing a *push gesture*. It allows the recognition to specific events related to item selection such as when the hand is over a different item, when that item is selected [21][20].

3.2.4 Licensing

NITE middleware framework is proprietary. It is a licensed under a *free license* with a specific *limitation* to the use only with a *PrimeSense chip*. Using NITE with any other 3D sensor breaks the licensing agreement. The *Kinect* contains PrimeSense PS1080-A2 image sensor processor, hence you can use the framework inside commercial software.

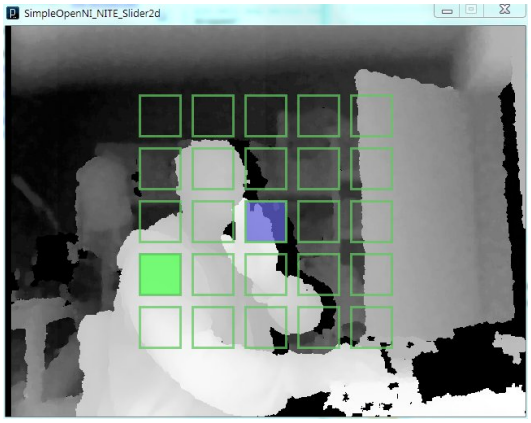

Figure 25: NITE Selectable Slider 2D. This Figure illustrates an instance of the 2D selectable slider implemented in SimpleOpenNI library, a *OpenNI* and *NITE* wrapper for Processing.

3.3 KINECT FOR WINDOWS SDK BETA

3.3.1 Introduction

Kinect for Windows Software Development Kit (SDK) Beta[7] is the programming toolkit provide by Microsoft Research to application developers as a starter kit enabling an easy access to the capabilities offered by the Microsoft *Kinect* device to develop creative and rich experiences applications running on the Windows 7 operating system. The Kinect for Windows *SDK* Beta includes drivers for using the sensor with a Windows-based PC, rich APIs and device interfaces including technical documentation and source code samples. This framework is designed and conceived for developers who build applications with C++, C#, or Visual Basic by using Microsoft Visual Studio 2010[8].

3.3.2 Features

In this section we will present all the features and capabilities provided by the Kinect for Windows *Software Development Kit (SDK)* Beta even if we chose to base our development on the *OpenNI-NITE* framework. We tested the library to have a global overview on all the possible *Kinect*-driven development systems.

RAW SENSOR STREAMS It enables receive and access raw data streams generated from the depth sensor, color camera, and four-element microphone array.

7 http://kinectforwindows.org/
8 http://msdn.microsoft.com/en-us/vstudio

SKELETAL TRACKING It offers the capability as depicted in Figure 26 to track the skeleton 3D model of one or two people within the *Kinect* field of detection allows the creation of gesture-driven applications. If you want to better understand how this framework performs the tracking, the development team of the framework published this [25].

ADVANCED AUDIO CAPABILITIES The frameworks also include some audio processing capabilities providing sophisticated acoustic noise suppression and echo cancellation, beam formation to identify the current sound source. It supports the integration with the Windows speech recognition *API*.

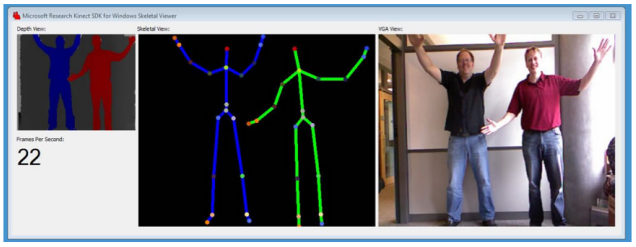

Figure 26: Skeletal Viewer Walkthrough using Kinect for Windows *Software Development Kit (SDK)* Beta.. This figure shows ones of the source code example provided with the Kinect for Windows *Software Development Kit (SDK)* Beta relative to skeleton tracking capabilities offered by the *Kinect*.

We tested in the research laboratory the framework capabilities and all the functionalities offered even though they aren't defined in our domain of application to better understand what a device like *Kinect* can offer opening new possible scenarios for the future but we decided to not introduce and use in our proposed implementation the Kinect for Windows *Software Development Kit (SDK)* Beta for various motivations. We will better analyse and comprehend the main differences between Kinect for Windows *Software Development Kit (SDK)* Beta and *OpenNI-NITE* framework in 3.5.

3.3.3 Licensing

Kinect for Windows *Software Development Kit (SDK)* Beta is designed for non-commercial purposes only; a commercial version will be available in early 2012 as reported in the website of the framework.

3.4 OTHERS

3.4.1 Introduction

In this section we will make a brief recap on the others most known and famous frameworks used to interact with the *Kinect*.

3.4.2 OpenKinect - Libfreenect

OpenKinect[9] is an *open source library* developed by an open community of over 2000 members with the main goal of creating the best possible suite of applications for the *Kinect*. Their primary focus is the development of the *libfreenec* software which includes all the code necessary to start develop application with the *Kinect* sensor. The library includes a set of *cross-platform* APIs that works on Windows, Linux, and OS X with a large set of bindings for the most common programming languages and platform like C, C++, Java, Python and so on. *Libfreenect* is the core library for accessing the Microsoft *Kinect* and enables acquiring data from *RGB* and depth camera, motors, and led. OpenKinect operates at low-level and derives from the initial, reverse-engineered *Kinect* driver with respect to OpenNI/NITE which is developed by PrimeSense, the creator of the depth sensing technology at the base of the *Kinect* functioning. It doesn't offer the possibility to have specific 3D data models for skeletal tracking or gesture recognition, hence we decide to not consider it in our work. We can define OpenKinect more a driver than a middleware framework.

Licensing

The code contribution to OpenKinect are available under an Apache 2.0 or optional *GNU is Not Unix (GNU) General Public License (GPL)* v2 license.

3.4.3 FAAST

Flexible Action and Articulated Skeleton Toolkit (FAAST)[10] is a middleware framework to facilitate the integration of full-body control with games and Virtual Reality applications developed by the "*MxR*" team at the University of Southern California Institute for Creative Technologies starting from December 2010. The framework relies upon *OpenNI* described in section 3.1 and *NITE* described in section 3.2 to track body motion using the Microsoft *Kinect* as you illustrated in Figure27. The toolkit can also *emulate keyboard input* triggered by gestures or body

9 http://openkinect.org
10 http://projects.ict.usc.edu/mxr/faast/

postures allowing the user to add custom body-based control mechanisms to the existing video games or applications. *FAAST* also enables *mouse control* from a front position with respect to *Kinect*. It also includes a custom *Virtual Reality Peripheral Network (VRPN)* server to *stream* and diffuse the user's skeleton over a network, allowing clients and applications to read the skeletal joints as trackers. The preliminary version of *FAAST* is currently available for Windows only. This is one of the big motivation why we decide to not use this proposed solution.

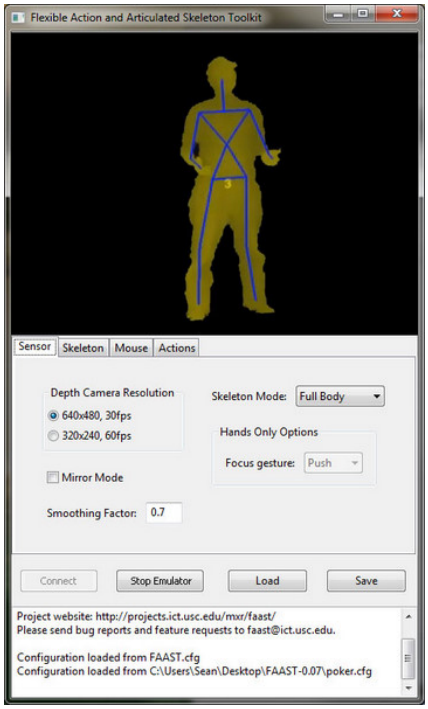

Figure 27: *FAAST*. In this figure is presented the *FAAST* software interface as it appear to the user.

Licensing

FAAST is free to use and distribute for research and non-commercial purposes. If you want to use it for commercial purposes you need to contact the development team.

3.4.4 ROS OpenNI

Robot Operating System (ROS) Kinect[11] stack wraps the *OpenNI* drivers to integrate the Microsoft *Kinect* with *ROS*, as well as higher level libraries like skeleton

11 http://www.ros.org/wiki/kinect

and gesture tracking. *Robot Operating System (ROS)* is a huge set of libraries and tools provided to create *robot applications*. This driver provides the full capabilities of the sensor, including in-sensor registration for *RGB* and depth camera, and full audio support. This driver supports Linux, OS X, and Windows. This framework is at the basis of the famous "*MIT* Kinect Demos"[12], you can see an example in 28, it has good performance and very powerful but we didn't want to be limited to work only in the *ROS* environment, in addition it's very huge, quite dispersive and focused on different purpose with respect to our objectives of build a more lightweight and simple solution.

Figure 28: MIT Kinect Demos - Minority Report Interface. A famous demo created extending *ROS* by some *MIT* students.

Licensing

All the *ROS* core code is licensed under *Berkeley Software Distribution (BSD)* and also the *ROS OpenNI* is under *BSD* license.

3.4.5 As3Kinect

The as3Kinect[13] project is a particular framework that acts like a socket server; it reads or receive data from the *Kinect* and sends that data over the network to an *ActionScript client*. This application is known as the *as3kinect server*, or sometimes as the *as3kinect wrapper*. To use it, you need a client application. This client needs to know which protocol the server uses when sending the data. To ac-

12 http://www.ros.org/wiki/mit-ros-pkg/KinectDemos
13 www.as3kinect.org/

complish that, the as3Kinect project includes an *ActionScript 3* library known as the "*as3kinect client library*". Flash developers can use this *API* to communicate with the server/device. This library is a wrapper to other libraries, these libraries generates the data processed by the framework, the frameworks supported are *OpenKinect*, *OpenNI* and *Codelaboratories NUI* (not developed anymore).

3.5 COMPARISONS

3.5.1 Introduction

In this section we will make a *detailed comparison* between the *OpenNI/NITE* framework and the *Kinect for Windows Software Development Kit (SDK) Beta*, they are the most famous and used framework to develop *Kinect*-based applications. This comparison has the main objectives to highlight the main differences between the libraries, analyse all the pros and cons, and lastly to explain clearly, strengthen and better motivate our decision to use the *OpenNI/NITE* framework as the basis of our developed solution.

The comparisons structure, model and development is based on [4] and on [8], we report some informations listed in these articles adding some extra note we collected during our test phases.

3.5.2 OpenNI/NITE Framework

We start examining and listing the positive features offered by the *OpenNI/NITE* framework:

- the license allow a commercial use;
- it provides hand tracking and hand gesture recognition;
- it can automatically synchronize depth image frames with the color image frame;
- it detects full body skeleton:
  - the framework generates a tracking set of 20 joints;
  - it also take in considerations orientations for the joints;
  - it seems to consume less resources than Microsoft *Kinect SDK*'s body tracker because it doesn't perform a predictive analysis, this analysis can increase the tracking accuracy but requests a lot of computational resources[25].

- it supports other devices like the PrimeSense sensor and the ASUS WAVI Xtion device.
- it supports multiple sensors;
- it offers a specific architecture where more device, middleware software and applications can be registered and composed in different ways;
- it is cross-platform running on Windows (including Vista and XP), Linux and Mac OS X;
- it comes with code for full support in Unity 3D game engine;
- it support for record and playback the audio to and from the storage;
- the library has events to detect particular situations like a new user enters the detection field, leaves the field and so on.

This framework has also some cons and aspects that can be improved:

- it does not offer advanced support for audio;
- it does not have any kind of support for using the motor;
- the body tracking features needs some improvements:
  - the orientations is not supported by all the joints especially for the head, hands, feet, clavicles;
  - a calibration pose is needed to start tracking (but it can be saved and loaded for reuse)
  - occluded joints are not estimated sometimes;
- it requires three different separate installers and a *NITE* license string (but the process can be automated building a unique installer)
- *SDK* does not have a specific callbacks for events as new video or new depth frame is available.

3.5.3 Kinect for Windows SDK Beta

Now we will analyse and listing the positive features offered by the *Kinect for Windows SDK Beta* library:

- it provides full body tracking:

  - it does not need a calibration pose because the incoming data from the depth stream is compared with known images of humans quickly determine human shaped objects[25];
  - the framework generates a tracking set of 20 joints;
  - it seems to deal better with occluded joints.
- it supports multiple sensors;
- it offers advanced audio capabilities:
  - acoustic noise suppression;
  - echo cancellation;
  - beam forming;
  - it can be integrated with Windows speech recognition *API*.
- it allows access to the motor/tilt;
- a single installer is needed (but if you want to use all the features and capabilities the number of installer needed increase)
- the framework not provide callbacks for events for like a new video or new depth frame is available;

This framework has also some cons and aspects that can be improved:

- it is licensed for non-commercial use only;
- it tracks only full body, no hand tracking supported;
- it does not offer frame synchronization of two different sensor of the color&depth image streams to one at the moment;
- the body tracking provided needs some enhancements:
  - it only calculates positions for the joints and not orientations;
  - it only tracks the full body, you cannot track only the upper-body or single hands;
  - it is more prone to false positive with respect to *OpenNI*[25];
  - it seems to consume more computation resources than *OpenNI/NITE* due to the fact it performs a predictive analysis of the tracking joints specially when the sensor loses user's tracking[25].
- no gesture recognition system is present;
- it only support a single device: the *Kinect*;

- it runs only on Win7 (x86 and x64);
- no support for Unity 3D game engine;
- no support to stream the raw *IR* video data;
- the library does not trigger events when new user enters in the detection field or when the user is lost and so on.

Now we have a clear overview of the main differences and features offered by the different development system. We are ready to explain the main motivations why we decide to base our developed solution on the *OpenNI/NITE* framework:

1. when we started our development process the Windows *Software Development Kit (SDK)* wasn't already released;
2. this framework doesn't provide at the moment gesture recognitions capabilities;
3. *OpenNI/NITE* framework seems to be more lightweight with respect to the Microsoft proposed solution;
4. our application need the support of special events to enable the configuration management as new user enters in the detection field or when the user is lost and so on;
5. lastly, the most important because we decided to maintain a platform independence guaranteed from the fact that *OpenNI* and *NITE* provide a cross-platform solution.

4 DEVELOPED SYSTEM

4.1 PROPOSED SOLUTION

This chapter is the key and crucial point of this work because here we will present our developed prototype relative to the emulation of an *IWB* based on the utilization of Microsoft *Kinect*. Our proposed solution provide an *IWB* where we *control the mouse position* mapping it with the user's *hand joint* exploiting the *skeleton tracking* capabilities offered by the device and the *OpenNI/NITE* frameworks, in particular we are able to put exactly the mouse cursor on the top of the index finger of the user. We will start showing the *UML* diagrams constitute the foundation of our proposed solution from an external and internal points of view. We will present the SimpleOpeNI library, a simple *OpenNI* and *NITE wrapper* for *Processing*. Our prototype was built on top of this library to have a simple and abstract programming tool which can take advantage on all the functionalities and capabilities provided by *OpenNI* and *NITE*. We will drive the user in the process to understand how develop a *Kinect*-based application providing some examples written in *Processing* we used as reference during our development process. In the last two sections we will analyse the development details of the proposed solution examining the development process we adopted phase-by-phase and the implementation details contained in this project providing some code examples and solutions to the problem we have encountered.

4.2 UML

In this section we will introduce the *UML* diagrams at the base of our developed solution. We will approach the proposed solution from different perspective: the *external view* considering the system as a black box and the *internal view*, in this case we will examine the system from the inside, understanding what are the different components and how they interact.

4.2.1 External view

The external view doesn't explain the details regarding the internal functioning mechanisms of the system. The system is represented as a black box. It simply

helps to understand what the system offer to the user and tries to explain the system behaviour perceived from the outside.

Use Case

The first diagram we will see is the *Use Case Diagram*. This diagram is designed to highlight all the *features and functionalities* offered by the *system* to the *user*. In our specific case, like depicted in Figure 29, the system provides:

- the system enables the user to perform a left or right click: we are currently *still use* the *mouse* to perform this activity because we decided to put the device behind the user back in order to map the mouse position using the skeleton tracking feature provided by the *OpenNI/NITE* framework. This specific choice has created us some problems because our aim was to eliminate completely the use of any kind of device but in this situations we maintained the possibility to click with the mouse or another similar device because the user isn't able to perform any kind of gesture having the *Kinect* behind his back;
- the user in order to move the mouse mapped on his hand or better on his index finger must perform a *calibration procedure* or load an existing one. The calibration phase is subdivided in three different stages, we will discover in the next sections how they works:
  - skeleton calibration;
  - the point calibration;
  - the finger calibration.
- different *visual feedbacks* must be reported to the user such as a *background removal* completed with skeleton *limbs* and all the events relative to the user-tracking procedure such as a new user is detected, the user is lost, the calibration is completed successfully and so on.

Activity Diagram - Calibration

The next diagram we will examine regards the different *stages* composing the *calibration procedure* (see details in 4.6.2). As you can see from the diagram illustrated in Figure 30 when a new user is detected he or her can decide to perform a new calibration or load an existing one (like described in 4.6.2), if the user chooses to perform a new calibration the skeleton calibration is the first step the user needs to cope with to go through the calibration process. When the *skeleton calibration* (see details in 4.2.2 and in 4.3.3) is completed successfully, the user is tracked and now he can perform the *point calibration* (see details in 4.6.2) stage

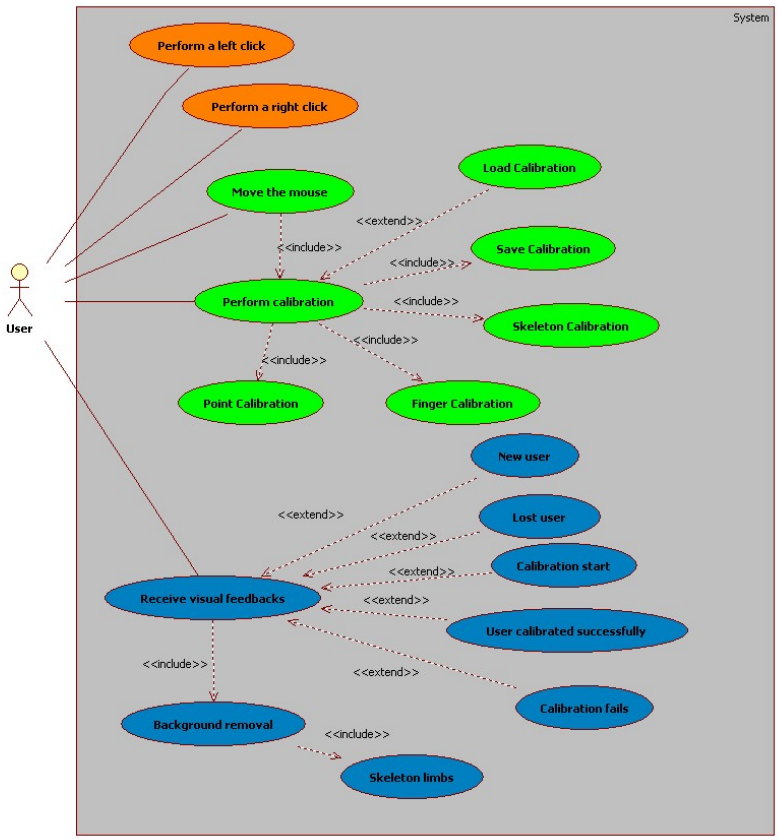

Figure 29: Use Case Diagram. In this picture you can see the Use Case Diagram of the system.

which allows to to map the screen space projected onto the wall and the view field of the device fixing nine points on the screen. The last stage regards the *finger calibration* step (see details in 4.6.2). This stage enables to fix the right offset of the top of the index finger where we must put the mouse cursor with respect to the hand joint provided by the tracking framework. The framework doesn't perform directly finger detection at the moment.

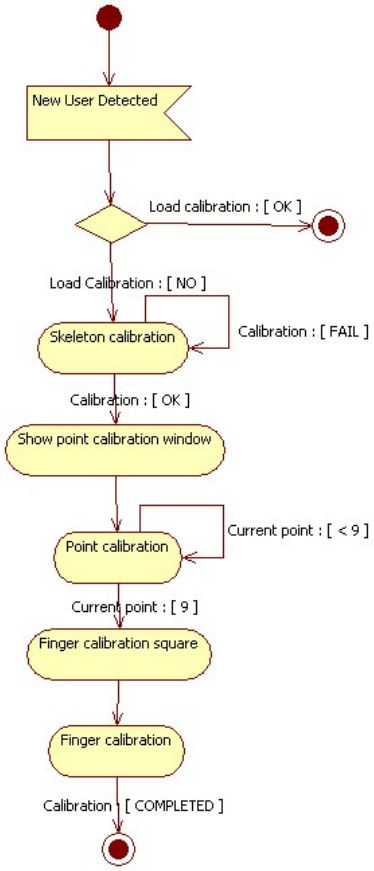

Figure 30: Activity Diagram - Calibration. In this picture you can see the Activity Diagram relative to the calibration procedure.

4.2.2 Internal view

The internal view describes the structures and components, the internal processes and activities, relationships and interactions between the different components inside the system.

Package Diagram

We will start understanding the different units composing the system inspecting the *Package Diagram* of the system presented in Figure 31. This diagram is useful to clarify the *hierarchy* and the *relationship* between the different part form the application. In our specific case we have the *simpleopenni* package which is the core of the application and the interface with the tracking frameworks. This package interacts with the *calibration* package to control the calibration process through the different stages and with the *ui* package dedicated to provide visual feedbacks to the user.

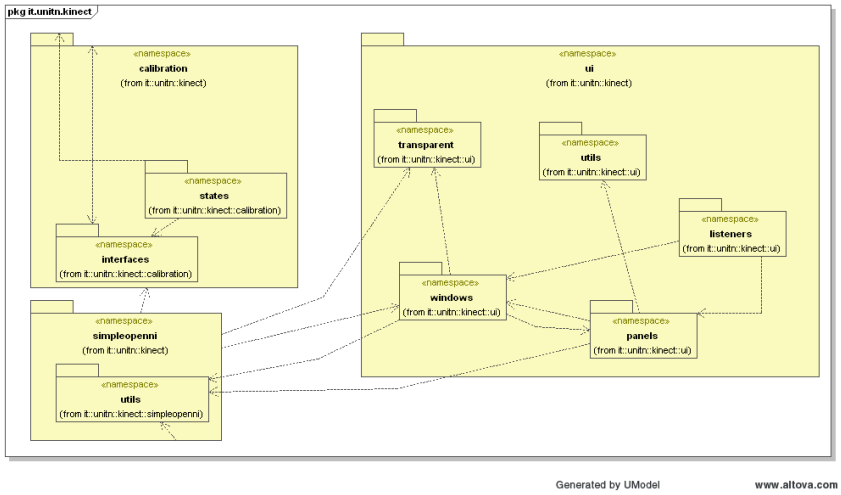

Figure 31: Package Diagram. In this picture you can see the Package Diagram relative to the entire application.

Class Diagram - Calibration

The *calibration* package is subdivided in the *context*, the interface with the external world useful to retrieve the current state and move to the next, and the different implementations of the various states of the calibration procedure like illustrated in the diagram in Figure 32.

Sequence Diagram - Skeleton Calibration

This *Sequence Diagram* is a preview on what we will see in a more detailed explanation in section 4.3.3. In this diagram we will try to explain what are the *interaction* between the *application* and the *OpenNI* framework during a *skeleton calibration*: when a news user is detected an event is triggered in the application, the application in turn request to the framework to control if the user has assumed the calibration pose showed in Figure 21, when happen *OpenNI* triggers the next

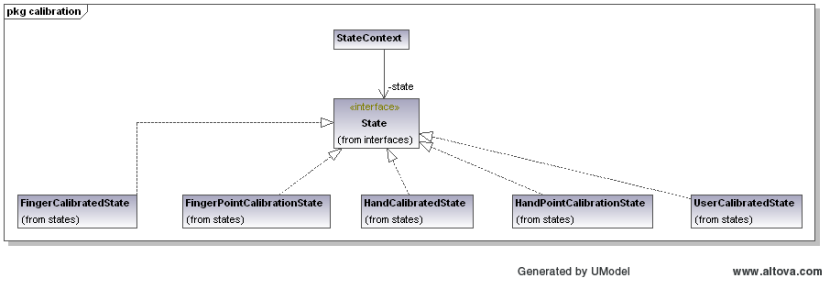

Figure 32: Class Diagram - Calibration. In this picture you can see the Class Diagram relative to the calibration process.

callback in turn and application request to really calibrate the user, finally *OpenNI* responds with the outcome of the calibration. The application is now able to track the user skeleton.

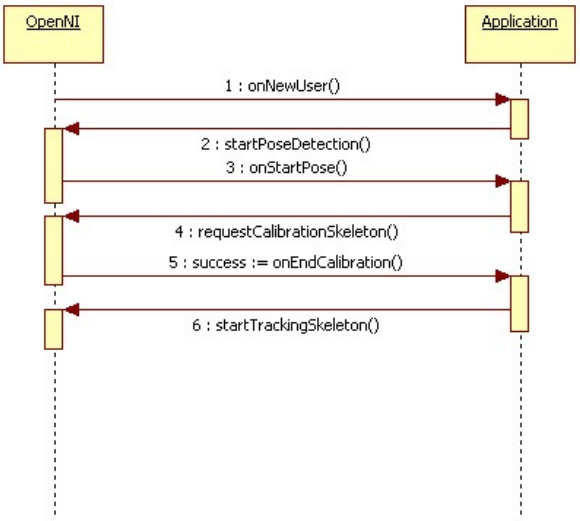

Figure 33: Sequence Diagram - Skeleton Calibration. In this picture you can see the Sequence Diagram relative to the skeleton calibration procedure.

4.3 SIMPLEOPENNI

SimpleOpenNI represents the *core library* of this project. It is a simple *OpenNI* and *NITE wrapper* for *Processing* language. It's meant to enable a more simple access at some functionalities offered by the *OpenNI/NITE* framework such as skeleton and hand tracking, gesture recognition, scene analysis and so on. In this section we will help the reader to comprehend why we decided to adopt this library in

the project introducing the Processing language and its particular *philosophy*. We will discover the fact that this philosophy fits very well the approach we chose to use to develop the *IWB* emulation. We will also analyse some examples written in the same language that will illustrate all the main functionalities offered by the library highlighting some pieces of code and the simplicity the library enables to obtain in a few line of code our first working example.

4.3.1 Processing

Processing[1] is an *open source programming language* and also an *Integrated Development Environment (IDE)* designed and built for students, artists, designers, researchers and programmers who want to create *easily* images, animations, and interactions. Initially developed to serve as a software for the electronic arts and visual design communities with the purpose of *teaching* the basics of computer programming in a visual context, and to serve as the foundation for electronic *sketchbooks*, Processing also has evolved into a tool for generating finished professional work. It was conceived and built in 2001 by *Casey Reas* and *Benjamin Fry* both members of the group called at the "Aesthetics and Computation" Group *MIT* Media Lab[2][33]. Processing provides a new way to learn programming through creating interactive graphics and its philosophy is based on a particular technique called *sketching*, where the approach to the programming language is simplified making it more accessible: Processing has the capacity to offer an *immediate visual feedback* and adding more lines of code the program is created one piece at time keeping the visual feedback offered to the developer or designer. For example on [22] the creators of the language cite a possible approach for develop a simple program: "The idea is to write a single line of code, and have a circle show up on the screen. Add a few more lines of code, and the circle follows the mouse. Another line of code, and the circle changes color when the mouse is pressed". Another important characteristic offered by Processing and very useful for the development part of this thesis is the *extreme flexibility* provided, in fact exists more than a hundred extension libraries of the language covering different and various domains and the main core library used in this project *SimpleOpenNI* is one of them.

The language is derived from the *Java* programming language. The Processing environment is entirely written in Java. Programs written in Processing are also translated to Java and then run as Java programs. This particular feature has enabled us to include the Processing code inside a more complex Java project allowing us to exploit the potentiality of both the languages. We need to make a large distinction between Processing and Java, Processing provides powerful

1 http://processing.org/
2 http://www.media.mit.edu/

graphics library and a simplified programming style that doesn't require users to understand more advanced concepts like classes, objects, or animation and double-buffering (while still making them accessible for advanced users) making programs shorter and easier to read.

We have created a special appendix A where we will indicate all the operations needed to *install* Processing, the SimpleOpenNI library and try to run the example we will explain below and reported in a complete version in B. At this point we need to make a clarification, the *various examples* reported below can be *copied* inside the Processing environment and *tested*. The emulation of the *IWB* we propose can be launched without installing Processing and SimpleOpenNI, *all the libraries* needed for functioning are already *included*, you need only to previously install the *Kinect* driver and the correct version of *OpenNI* and *NITE*.

In the next sections we will start to introduce some example of source code written in Processing and based on the wrapper used in our project to interact with *OpenNI* and *NITE*. We start with some Processing example to help the reader to have a clear comprehension on what means write a *Kinect*-based application. As we have previously mentioned our project is based on a *normal Java project* (created using Eclipse *IDE*) that *includes* the *Processing code* needed to interact with the SimpleOpenNI library, hence at the end of the chapter we will show how is possible to establish this particular integration. In the source code introduction we will take as reference the examples provided with the SimpleOpenNI wrapper and others reported in this *beautiful book* [3] wrote by Greg Borenstein and released in October 2011 as an early release with the first two chapters, the final release will be estimated for January 2012. We need to make a *clarification* at this point, the code written for the proposed *IWB* emulation you will see in this thesis is not copied by the example described in the book because the code development relative to this project is ended in the first days of September and the book was released in October. Probably with the help of the book we could even wrote a better code or conduct a more faster development. In this thesis we *don't provide* an *hand gesture specific example* because we not really use it inside our emulation of the *IWB* even though we tested it during our development process and we reported some details regarding the hand tracking in 3.2.3. If you are interested you can find one in the examples provided within the SimpleOpenNI library or in the book [3].

4.3.2 First Program

In this section we are going deeper into technical details and implementations starting to see our *first Processing program* based on the *SimpleOpenNI* library to better understand how design software application that can interact and take advantage of the *Kinect* features and capabilities. This simple program read the

images from the *depth camera* and the *RGB camera* displaying them side-by-side on the screen. We take as first model of *Kinect*-based application a modified version of the "*Depth Image Test*" you can find in the SimpleOpenNI library examples. In this section we refer of some informations reported in the *Chapter 2* of the book we have mentioned before [3].

Now we will examine the code reported in B.1 line by line to have a clear comprehension since if it is your first time working with this library it is really important for you to understand this example. On line 1 we start importing the library:

```
import SimpleOpenNI.*;
```

The second step is to declare the library object, the variable will be instantiate during the *setup* process:

```
SimpleOpenNI  context;
```

This object is the *most important* one and the core of the source code because it enables to *access all the data* generated by the *Kinect* as depth and color images, skeleton models and hand points. In this point we have simply *declared* it and not instantiate it, hence in the setup function below we need to be sure to do it. The first function we need to implement is the *setup* function. As described above, firstly, we need to instantiate the SimpleOpenNI instance that we declared at the top of the sketch:

```
void setup()
{
  context = new SimpleOpenNI(this);
```

In the example reported in [3] they start defining the size of the application but in our case we derive it summing the depth map dimension and the dimension of the *RGB* images like in it happen on line 18. We proceed enabling the *mirror mode*, as we have already told the mirror mode enables to project the images on the screen like if you are in front of a mirror.

```
// mirror is by default enabled
  context.setMirror(true);
```

The next two instructions are very important because they allow to specify we're going to enable both the depth image and the *RGB* image streams generated from the Kinect. This phase where the developer specifies the kind of data he want to have access is very important because there is a clear indication on what we need and the library is able in this way to do just enough work to provide to the developer the exact information he needs optimizing the computation.

```
// enable depthMap generation
 context.enableDepth(640,480,30);

 // enable ir generation
 context.enableRGB(640,480,30);
```

Here we need to do a parenthesis on the *Kinect* provided resolution and frame rate. In the specification reported in 2.1.3 we said the *Kinect* as a depth resolution of *Quarter Video Graphics Array (QVGA)* 320×240 at 30 frames per second but it can support also *VGA* 640×480 at 30 frames per second, hence we use the last one for both *RGB* and depth streams. By default both streams are setted to this resolution but to be more clear in this case we specify it. Other resolutions for the *RGB* stream are supported but at the moment we decide to use this one. The configuration in this case is forced from code but is possible to load it from an *OpenNI eXtensible Markup Language (XML)* configuration file like this one:

Listing 4.1: SamplesConfig.xml

```
<OpenNI>
        <Licenses>
                <License vendor="vendor" key="0KOIk2JeIBYClPWVnMoRKn5cdY4
                    ="/>
        </Licenses>
        <Log writeToConsole="true" writeToFile="false">
                <!-- 0 - Verbose, 1 - Info, 2 - Warning, 3 - Error (
                    default) -->
                <LogLevel value="3"/>
                <Masks>
                        <Mask name="ALL" on="false"/>
                </Masks>
                <Dumps>
                </Dumps>
        </Log>
        <ProductionNodes>
                <!-- Normal Image -- -->
                <Node type="Image" name="Image1">

                        <Configuration>
                                <MapOutputMode xRes="640" yRes="480" FPS=
                                    "30"/>
                                <Mirror on="true"/>
                        </Configuration>
                </Node>

                <!-- HighRes Image -->
                <!--
                <Node type="Image" name="Image1">
                        <Configuration>
                                <MapOutputMode xRes="1280" yRes="1024"
                                    FPS="15"/>
```

```
29                              <Mirror on="true"/>
30                      </Configuration>
31              </Node>
32              -->
33
34              <Node type="Depth" name="Depth1">
35                      <Configuration>
36                              <MapOutputMode xRes="640" yRes="480" FPS=
                                    "30"/>
37                              <Mirror on="true"/>
38                      </Configuration>
39              </Node>
40              <!--
41              <Node type="Audio" name="Audio1">
42              </Node>
43              -->
44      </ProductionNodes>
45 </OpenNI>
```

In the configuration file are defined the vendor license as illustrated in 3.1.6 and the production nodes, for a detailed description see section 3.1.3, in this particular case a “normal" RGB image node and depth node. To load the file however you need a different instantiation of the library:

```
context = new SimpleOpenNI(this,"SamplesConfig.xml");
```

Closing this parenthesis on the resolution and frame rate adopted we proceed in the code analysis defining the appearance of our application indicating the background color and the size of the window derived as we have already anticipate form the depth and *RGB* images dimensions.

```
background(200,0,0);
size(context.depthWidth() + context.rgbWidth() + 10, context.rgbHeight());
```

Now we are ready to examine the “dynamic" part of our application or better the *draw* loop where we access to the streams and we draw them on the screen inside the application window. This tells the library to get updated data from the device so that we can process it. The update rate also depends from the frame rate generated by the *Kinect* (30 frame per second) and the update frequency of the application, in this specific case the application is simple, hence the update rate of the application is probably more faster with respect to the *Kinect*. This is an aspect you have to deal when you are building complex application.

```
void draw()
{
  // update the cam
  context.update();

  // draw depthImageMap
```

```
  image(context.depthImage(),0,0);

  // draw RGBImageMap
  image(context.rgbImage(),context.depthWidth() + 10,0);
}
```

This is the key point of the example because here the depth and color images are displayed using the *depthImage()* and the *rgbImage()* which returns the freshest images available and passed to the Processing's built-in image() function along with the position where to put the images inside the application window. As result of running our first example of *Kinect*-based application we obtain:

Figure 34: First program with SimpleOpenNI. This figure represents a screen capture obtained running our first program, as you can see there is the depth map image side-by-side the color image obtained activating the *RGB* camera.

4.3.3 Skeletal Tracking

Now we are able to create and run our first *Kinect*-based program, we are ready to see something more interesting and strictly related to the code developed in this thesis: a *skeletal tracking example*. In this case we will base our code explanation on the example provided with the library called "*User*", you can find all the code in the Listing B.2. Citing another time the book [3] all the *Chapter 4* is dedicated to skeleton tracking, in this section we will refer to some advices and annotations reported in the book.

In the last example we simply print the depth data obtained form the *Kinect*. In this example we use a different approach, *OpenNI* has the ability to process the depth image for us in order to detect people and produce with the analysis performed by the *NITE* framework a 3D model of a person's skeleton or better it provides directly a *skeleton* model. *OpenNI* once has detected the user, it will tell us the position of each of the user's visible *joints*: head, neck, shoulders, elbows, hands, torso, hips, knees and feet, like depicted in Figure 12. We are

building an *IWB* and the skeleton tracking provided is exactly what we need to pursue our goal. If we want to be more specific our goal is to *map the mouse position* on the *top of the user's index finger*. We will see in the next chapters how we had pursued our goal but at the moment we start understanding how the *skeletal tracking* works. First of all, we will have to learn to use the functions that SimpleOpenNI provides for *accessing the joint data* and during this process we will need to understand how the *skeleton calibration* works because it is the basis to start the tracking process. SimpleOpenNI also provides its own callbacks that let us detect some *events* during the *user-tracking process*: when a new user is discovered, when tracking first begins, when a user is lost and so on. Learning how to work with these callbacks is key to taking full advantage of *OpenNI*'s tracking facilities. The last phase once we have learnt how to access skeleton data and specially to an individual joint we will use all the joints to draw a a basic *stickman* that follows the pose of the user. SimpleOpenNI provides special functions to draw the straight lines that connects two adjacent joints called "*limbs*". Analysing this example we skip all the instructions examined in the last proposed example and we will focus our attention to the skeleton related instructions and functions. Now we are ready to start, firstly we will see the instruction to enable the user skeleton tracking:

```
// enable skeleton generation for all joints
  context.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
```

The enabling function takes an argument *SimpleOpenNI.SKEL_PROFILE_ALL*, this is a constant that tells SimpleOpenNI that we want to *track all the joints* in the user's skeleton. In this case we not examine the code line by line from top to bottom but we skip some lines and we return back later because we have to introduce the *particular events* triggered by *OpenNI* and detected by the library relative to the user-tracking process. In addition we need to make a small deviation explaining all the *phases* of the *skeleton calibration procedure*. When we talk about user-tracking events we refer for example to the event triggered when a new user is detected:

```
void onNewUser(int userId)
{
  println("onNewUser - userId: " + userId);
  println("  start pose detection");

  context.startPoseDetection("Psi",userId);
}
```

This event enables a *pose detection procedure*, as we have argued in 3.2.2 a calibration process is used to adjust the skeleton model to the user's body proportions and starts the pose tracking. The user must assume what in the technical literature and in PrimeSense's own documentation it is called the "*Psi*" pose: you have to stand in front of the device with your feet together and your arms raised above your shoulders on the sides of your head, like illustrated in Figure 21. In

Figure 35 are presented the calibration flow composed by different stages where you have to go through to enable the skeleton tracking. In the flow in Figure

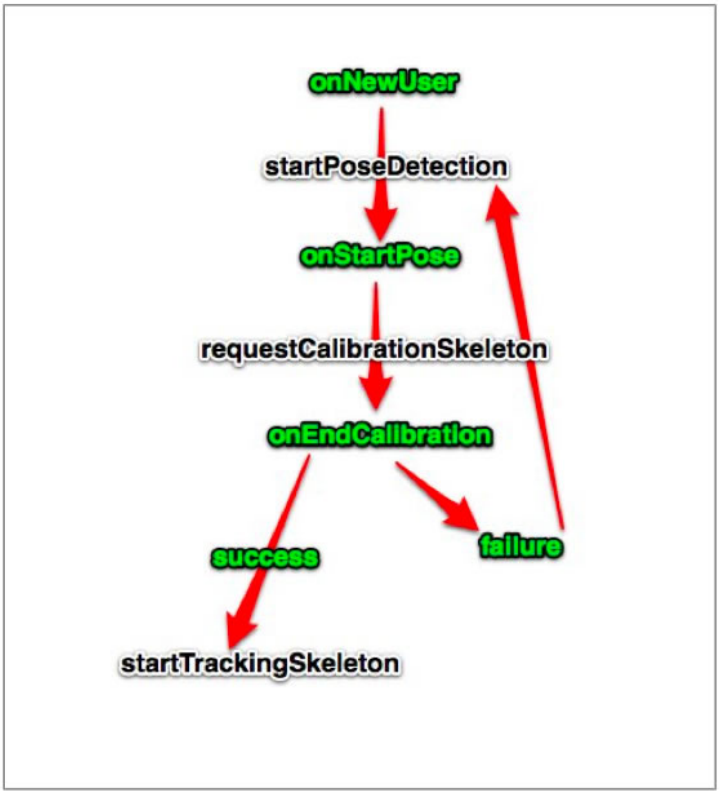

Figure 35: NITE Calibration flow. This figure represents the different step needed to perform the skeleton calibration phase.

35 you can find the different actions and callbacks needed to complete the process. The calibration phase is defined in [3] a "back-and-forth process between our application and *OpenNI*", in fact, at each step, *OpenNI* will trigger one of the defined callback functions, for every callback correspond a specific action to go through the various step of the process. When *OpenNI* detects the first user it calls *onNewUser()* function. Within this function, we can start the tracking process by calling *startPoseDetection()*. This function tells to *OpenNI* to control if the user has assumed the calibration pose showed in Figure 21. As soon as the user does assume the pose, *OpenNI* will trigger the *onStartPose()* function.

```
void onStartPose(String pose,int userId)
{
  println("onStartPose - userId: " + userId + ", pose: " + pose);
  println(" stop pose detection");

  context.stopPoseDetection(userId);
  context.requestCalibrationSkeleton(userId, true);
```

```
}
```

Then, in order to proceed through the calibration process, we need to call *requestCalibrationSkeleton()*. That function will call the *real* skeleton detection process. When this process is completed *OpenNI* will report back by calling the *onEndCalibration()* function.

```
void onEndCalibration(int userId, boolean successfull)
{
  println("onEndCalibration - userId: " + userId + ", successfull: " + successfull);

  if (successfull)
  {
    println("  User calibrated !!!");
    context.startTrackingSkeleton(userId);
  }
  else
  {
    println("  Failed to calibrate user !!!");
    println("  Start pose detection");
    context.startPoseDetection("Psi",userId);
  }
}
```

The calibration process cannot be always successful, in fact, the status of the calibration will be reported back by *OpenNI*. If it succeeded, we can call the last function *startTrackingSkeleton()* to begin accessing the joint data from the user. In the case the calibration process fails, we can start from the beginning by recalling *startPoseDetection()*. In order to improve the chances of a successful calibration we added to the project an additional on-screen visual feedback such as described in section 4.6.1.

Closing the parenthesis relative to the different calibration stages we can continue the analysis of the code example. Once the calibration phase is completed and the user skeleton is tracked you can start drawing it.

```
// draw the skeleton if it's available
if(context.isTrackingSkeleton(1))
    drawSkeleton(1);
```

In the code reported below you can see the drawing of some *limbs*, for each segment is defined the start joint and the end joint we have to connect with the segment. The joint labels in Figure 12 correspond directly to the constants by which SimpleOpenNI refers to the joints.

```
// draw the skeleton with the selected joints
void drawSkeleton(int userId)
{
```

```
// to get the 3d joint data
/*
PVector jointPos = new PVector();
context.getJointPositionSkeleton(userId,SimpleOpenNI.SKEL_NECK,jointPos);
println(jointPos);
*/

context.drawLimb(userId, SimpleOpenNI.SKEL_HEAD, SimpleOpenNI.SKEL_NECK);

context.drawLimb(userId, SimpleOpenNI.SKEL_NECK, SimpleOpenNI.SKEL_
LEFT_SHOULDER);
context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER, SimpleOpenNI.SKEL_
LEFT_ELBOW);
context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW, SimpleOpenNI.SKEL_
LEFT_HAND);
```

If you want to see the outcome you can obtain running this example see the Figure 36 below.

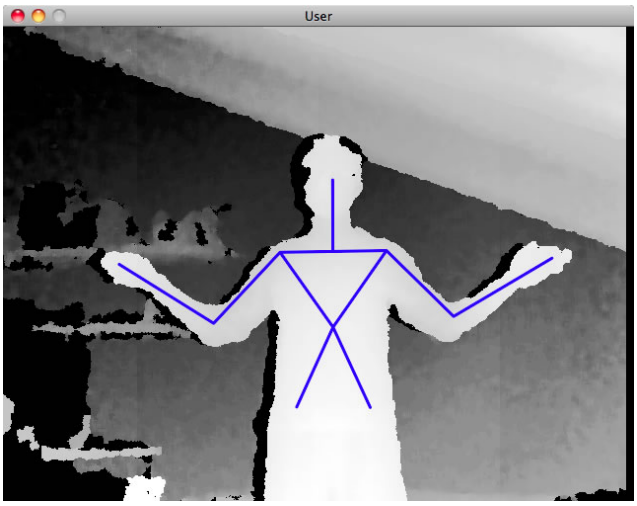

Figure 36: Skeleton Tracking example. This figure represents a screen capture obtained running the skeleton tracking example, as you can see there is the depth map image of the body with the various limb drawn on it.

4.3.4 Eclipse

Once we have understand how to track the user body we will introduce how to *integrate* the *Processing code* inside a *normal Java project* like we did in this project to exploit the potentiality of Java and Processing together. In the code listing B.3 we provide the main class of our developed software to help you understand how

to move the code. We take as reference the guide provided on the Processing website[3].

The first operation you need to perform is to include in your previously created class the core Processing library:

```
import processing.core.*;
```

Then you have to extend your class with the PApplet class is the parent Processing class that allows us to have access to all the Processing goodies.

```
public class Main extends PApplet {
```

Next you have to specify the main Processing functions you need to run a program: *setup()* and emphdraw() (specify "public" for these functions). In our code we don't specify the *draw* method because this is the main class used only to launch the program and all the logic of the program is included in a separated one. If you want to run your code as a Java Application (rather than applet) and turn on Processing's present mode: You must add a main() function to your code.

```
public static void main(String args[]) {
logger.debug("--- Enter - Paramenters = args: " + args.toString());

PApplet.main(new String[] { "--present", "it.unitn.kinect.Main" });

logger.debug("--- Exit");
}
```

4.4 SYSTEM ARCHITECTURE

In this section will guide the reader to understand the system architecture of the proposed solution. As you can see from Figure 37 and as we have previously mentioned different times our developed system is based on the *OpenNI/NITE* framework. The first bottom level of the system architecture is composed by the device and its driver. Then we have *OpenNI* which provide a set of *API* to be implemented by the various sensor and another set of *API* for the other side that is implemented by the middleware components, in fact, *OpenNI* wraps the *NITE* middleware framework. Then we have SimpleOpenNI, an abstraction layer for the underlying structure. On top of the stack we have our application based on SimpleOpeNI, which is based in turn to *OpenNI/NITE* framework, which in turn interacts with the device.

3 http://processing.org/learning/eclipse/

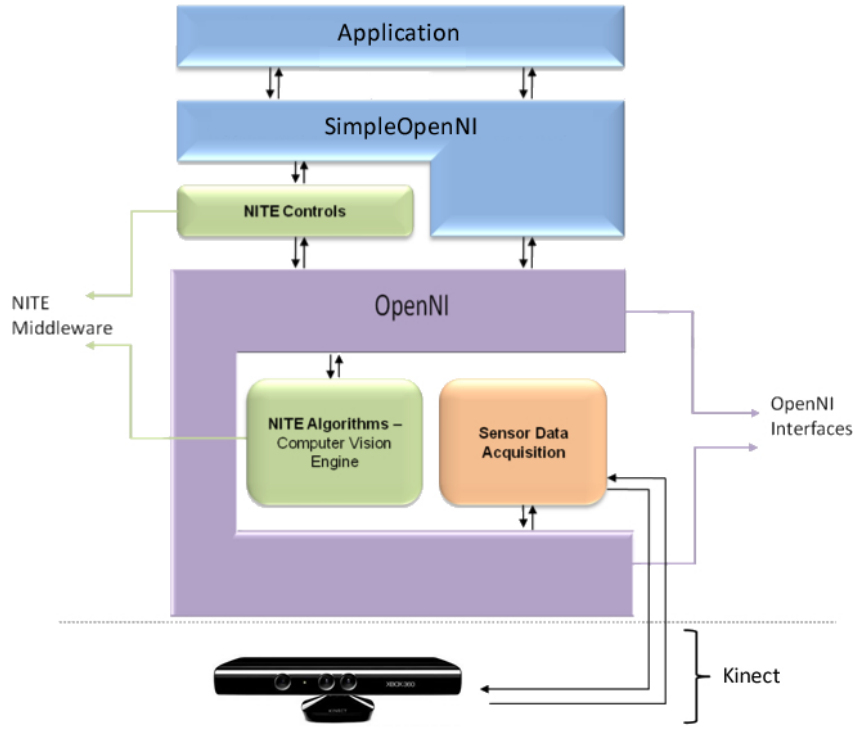

Figure 37: System Architecture. In this picture you can see the system architecture.

4.5 DEVELOPMENT PROCESS

In this section we will drive the reader through an overview of the main phases of our *development process*. Before we start to see phase by phase the evolution of our process we need to make some considerations about our *methodology adopted*, the approach used to solve some of the problem encountered during our development process. All this aspects are strictly connected with the state of the art of the technology and all the libraries involved in the project. Talking about the state of the art when we started examine what this particular technology can offer we encountered different problems due to its recent release (December 2010), such as:

1. the first and most important aspect is related to the fact that this technology is *new*, it introduces a new paradigm of interaction with the computer and all the software introducing a new set of considerations and problems we don't evenly know;

2. it seems the current *Kinect firmware* used is *limited* in various aspects like resolution and frame rate for example;

3. all the *libraries* examined are in *continuous evolution* due to their recent release, at the moment there aren't available solid libraries developed in the years;

4. the *tracking algorithms* provided by the different frameworks are often *unstable* and they don't offer the performance or the functionalities we wish;

5. the majority of the *Kinect*-based application, library and framework you find online are in most of the cases "*hack*s" and not stable libraries that can be usable for a project like this one, this aspect can create some confusion sometimes;

6. connected to the fact the libraries used to interact and process the data generated by the *Kinect* are in rapid evolution, the relative *documentation* is in the same state, creating some problem for the comprehension of the different code functionalities;

7. if we consider the *literature and academic productions* connected to this specific topic and all the related aspects we are in the same situation for the documentation, you can find only a few good reference;

8. probably we was among the *first in the world* to use the *Kinect* in a project like this one;

9. probably we was among the *first in the world* to decide to use the *Kinect* positioned *behind the user* and not in front of it comprehending all the problems related, in fact the device is designed and meant to be used with the user in front of it. The user must assume a pose in a way that all the body part can be visible and distinguishable by the *Kinect* without obstructions in between or to attach to the body. Surely some of the instabilities detected in the tracking algorithm are strictly connected to this fact.

The set of problems listed above that we have encountered in the first phase of the project during the research phase drove us to change our approach and methodology from the traditional approach to a more "*handmade*" process. We decide to adopt the philosophy proposed by the creators of the Processing language. This philosophy is based on a particular technique called *sketching*, where the approach to the programming language is simplified making it more accessible: Processing has the capacity to offer an immediate visual feedback and adding more lines of code the program is created one piece at time keeping the visual feedback offered to the developer or designer. Hence, we built our application on

step at a time in a cycle manner, adding a single feature for each step and testing it immediately to see possible problem we didn't consider because we aren't addicted to this kind of technology or simply the fact that the user wasn't able to perform some actions because we aren't using a precise device to move the mouse or the fact that the *tracking* can be *noisy* sometimes with the necessity to introduce an intermediate *approximation* level to smooth the detected points. At this point we can start examining all the phases we go through in our development process:

1. during the research phase the first step was to *test* a lot of libraries to see what in practice the *Kinect* can do and started trying to become more familiar on what means build a *Kinect*-based application;

2. the next phase consisted to *isolate the most used libraries* and the libraries offered the features we needed to emulate an *IWB*, as you can see in 3 there aren't a lot and offering the features we needed the choice was quite easy because when we start there was only one complete framework, the *OpenNI/NITE* framework (see respectively 3.1 and 3.2), developed by the company conceived the technology is at the base of the *Kinect* functioning;

3. once we have chose the libraries at the base of our developed solution we *installed* them on the different operating systems, we *ran* all the demos trying to understand what the library can did and how to use it for our particular purpose, or rather emulate an *IWB* taking as inspiration the *WiiLD* project described in 2.3.3; during this step of the development process we discovered the *SimpleOpenNI* library (see 4.3), a wrapper wrote in Processing that enables us to create an abstraction layer on top of both *OpenNI/NITE*;

4. the next phase regarded running *our first application* coded in Processing and based on the new amazing library we discovered, like the one examined in 4.3.2. We tried all the example proposed within the library and we started modified some of them to gain the mouse control using our hands from a frontal approach with respect to the device;

5. once we have gained the control of the mouse using the right hand and *gesture recognition* to enable the tracking session, we decided to create something more interesting giving a feedback on what the *Kinect* see as its detection field performing a *background removal*;

6. then we decide to introduce a *notification system* intercepting the events triggered during the user-tracking process such as a new user is discovered, lost and so on, providing another feedback to the user. We took as inspiration of our notification model (see 4.6.4) the famous software "Growl"[4].

4 http://growl.info/

During the implementation of this solution we decide to *move* our development outside the Processing *IDE*, because in our opinion is not the proper environment where build complex application. We integrated the code inside an out-and-out Java project (Processing language is derived from Java and it is completely integrable inside normal Java code) in the Eclipse *IDE* in a way that we can use a more suitable development environment and in addition having all the “power" provided by Java;

7. once we moved the development inside a Java project we tried the first attempt to create an *IWB*: we *moved* the *Kinect behind the user's shoulder*, but we made a mistake because in the process to map the mouse position exactly where is the user's hand we used the hand tracking provided by the gesture recognition algorithm of the *NITE* framework. We added a fixed offset to the mouse position with respect to the hand center position because the *NITE* algorithms don't perform finger detection and we want the mouse under the top of our index finger. We discovered that this approach works but its not precise at all (the fixed offset not help at all). During this phase we implemented our first form of *calibration* establishing a mapping between the virtual space of the *Kinect*'s detection field and the space delimited by the computer desktop projected on the wall. We used two point on the diagonal to enable the mapping but this a too much simple solution because if the *Kinect* is not positioned perfectly parallel to the wall surface the mapping can be wrong;

8. the next step is a crucial step in the development process because it represents the turning point on the right direction to achieve our goal: we decided to introduce the *skeletal tracking* and map the mouse position on the hand joint (also in this case the framework not perform finger detection provided by the *NITE* framework; at the same time we augmented the calibration point to nine subdividing the space in four quadrant. These choices allowed a better mapping of the mouse position but it wasn't perfect as we want, we need to use a custom offset;

9. our proposed solution relative to enabling a customized offset was to create an *additional calibration* (see 4.6.2 for more details) specific for the *finger* using as a model the calibration used for old joystick and gamepads;

10. during the optimization phase we discovered the fact that pointing on very precise points and writing can be difficult due too a *noisy tracking* in some cases with huge continuous oscillations of the mouse position, hence we introduced some *smoothing* and *approximation* on the points. This solution unfortunately has a bad counter-indication: it introduces some delay during the tracking;

11. in the last and final step we introduced the possibility to *save* all the calibration points and reload them when the user is lost or detected for the first time.

4.6 IMPLEMENTATION DETAILS

In this section we will highlight some *implementation details* of our proposed solution to illustrate some *new features* we added to the software or to explain in details how we solved some known *problems* explained previously. We will present some pieces of code and images showing the different solutions proposed.

4.6.1 Background Removal

As we have already mentioned in 2.2.3 the *visual feedback* returned to the user is vitally important during the process of *disambiguation* helping him or her assuming the correct pose to be detected enabling a the right behaviour of the software. In our case the user's position is very important with respect to *Kinect* because it enables a correct skeleton tracking from which depends a right mouse position on the screen mapped exactly on top of the index finger of the user. The visual feedback returned to the user allows also a good reference during the calibration phase to fix the correct virtual points (the nine point of calibration on the screen) with respect to the screen position from which we compute the mouse position with respect of the hand position in the virtual detection field of the device. Our application displays on the right bottom corner of the screen a small window where we separate out the part of the image that contains people from the background, practically we display what the *Kinect* detects only showing the user body cut out from all the rest part of the environments the body reside. We add also to the figure displayed the various *limbs*, the different segments which connects two adjacent joints (like described in 4.3.3 and big point positioned on the hand joint the user chose as the reference to map the mouse position.

The code we use to perform the *background removal* is illustrated in B.4. The code listed below shows how to retrieve the different streams needed to perform the *background removal*: firstly we retrieve the *RGB* image coming from the *RGB* camera, then we need to load the user's depth map.

```
// retrieve the current image from the RGB camera
rgbImage = kinect.rgbImage();

// load the pixel composing the colored image coming from the RGB camera
rgbImage.loadPixels();
```

```
// find out what are the pixels belonging to each single user
userMap = kinect.getUsersPixels(SimpleOpenNI.USERS_ALL);
```

Once we have this informations a loop cycling on the pixels of the background image creates a new image with the user figure depicted in the pixel labelled as belonging to the user inside the depth map. The last phase is to load the new created image in the window designed to show this visual feedback to the user.

```
// update the new image pixels
userImage.updatePixels();

// load the new user image in the feedback window in position 0,0
image(userImage, 0, 0);
```

The result of this particular operations is the one proposed in Figure 38.

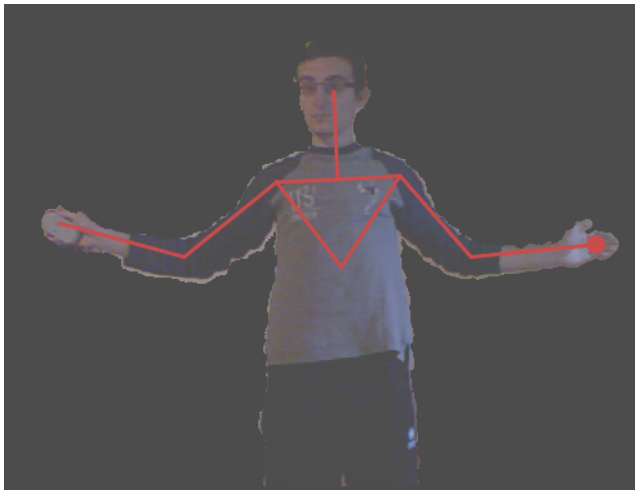

Figure 38: Background Removal. In this picture you can see a *background removal* example. In the figure are reported also the various *limbs*.

4.6.2 Calibration

The crucial point of the *IWB* emulation is the *calibration phase*. A good calibration enables a good tracking, if the calibration phase is not perfect it can influence in negative the tracking phase. This particular phase is subdivided in three different moments:

1. the calibration of the *user's skeleton* provided by the *OpenNI/NITE* framework;
2. the *point calibration*, during this procedure we need to map the screen space projected onto the wall and the view field of the device fixing nine point

showed on the screen, we will discover in 4.6.2 how the various calibration points will be fixed;

3. the last phase: the *calibration of the finger*, in the sense we need to compute the offset between the top your index finger, where we want the mouse cursor is positioned, and the center of the the user's hand select to be the reference for the mapping. We need to perform this specific operation because at the moment the framework *doesn't provide any sort of finger detection*, hence in order to position the mouse correctly with respect to the hand joint provided by the detection framework we need it to apply the right offset.

Skeleton calibration

This is the operation at the base of the *skeleton tracking*. We will not go into details on this particular phase because we have already described in details in the previous section 4.3.3, we simply make a small recap to explain clearly what the user must do to complete each phase of the calibration stage the software request. The *skeleton calibration* comprehends different stages, the user needs to assume a particular pose illustrated in Figure 39, once the starting pose is detected, the framework performs the real skeleton detection process returning back the outcome of this operation.

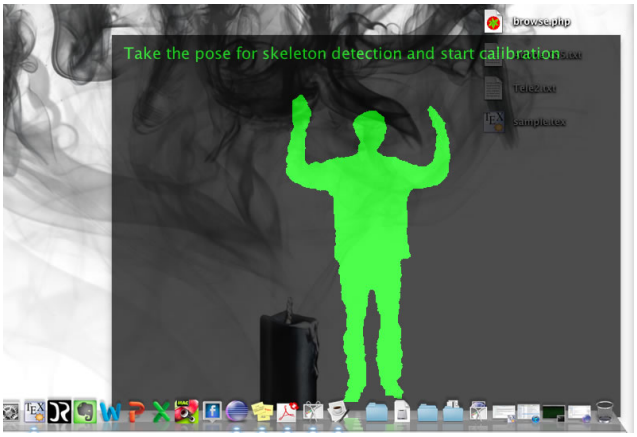

Figure 39: Start Pose Visual Feedback. In this picture you can see the start pose visual feedback is reported to the user indicating the correct pose he/her needs to assume.

If the skeleton calibration is performed successfully by the user, the software show on the visual feedback window the user body with a *background removal* (explained in previous section 4.6.1) with the addition of the skeleton *limbs* and big red point on the hand defined to be the reference for the mouse mapping as illustrated in Figure 38. Ended this phase the user must continue through the calibration process with the point calibration, see next section 4.6.2.

Point calibration

The *point calibration* phase is a crucial stage under the calibration process during which the user must fix nine points positioning the center of the hand on each single point and clicking with the left button on the mouse to fix the position. Through this procedure we are able to map the *screen space* projected onto the wall and the *view field* of the device. When you fix a point the next one will be highlighted to help the user understand which is the next one. The user start fixing the top left point then he or her moves horizontally on the same line from left to right, once he finish the first line, he starts again on the left with the next line from top to bottom.

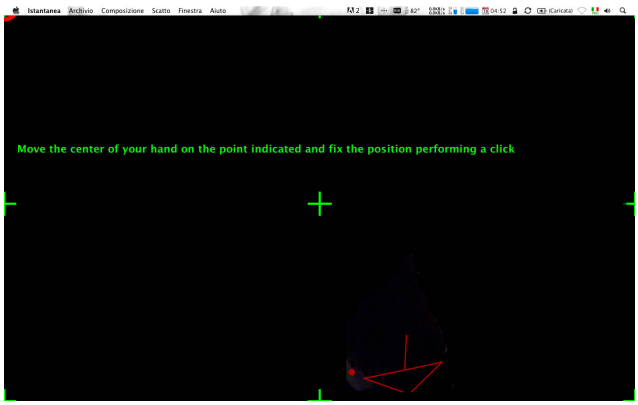

Figure 40: Point Calibration Window. In this picture you can see the point calibration window which enables to fix nine points positioning the center of the hand on each single point and clicking with the left button on the mouse to fix the position. Through this procedure we are able to map the screen space projected onto the wall and the view field of the device.

We adopted a specific Java pattern to control all the state transitions during the calibration process from a point to another or when a specific phase is completed implementing a "*State Pattern*". The *State Pattern* is a behavioral design pattern. This pattern is used to represent the state of an object, in particular when an object is a a function of its state and it must change its behavior at run-time depending on that state [9]. In the code listing B.5 you can find the implementation of "*context class*" which represents the only one interface with the pattern where you can retrieve the current state and move to the next state. We have represented the different "*states*" composing the state machine as derived classes of an abstract *State* base class implementing inside each instance of the various derived class the state-specific behaviour. This pattern does not specify where the state transitions will be defined. In our case we chose to implement the state transitions inside each single state to keep the context class more simple as possible. In the code

example below is represented the instantiation of the main class control the state transitions during the calibration phase, practically the context class:

```
calibrationState = new StateContext(new UserCalibratedState(), 9);
```

All the data collected during the point calibration phase is *saved* and used to *compute at runtime* the current mouse position. We chose to fix nine points to subdivide the screen in four quadrant to increment the accuracy of the mouse position in some particular cases when the device is not positioned correctly and perfectly parallel to the wall surface. We *pre-compute* some common parameters and constants to speed up the runtime calculation. All the operation performed are linear calculation.

Finger calibration

As we have previously mentioned the framework doesn't perform finger detection at the moment, so we add this feature using as a model the calibration used for old joystick and gamepads: we designed a square where the user have to put the palm of the hand at the bottom of the square like depicted in 41. There's a line firstly going form left to right and then from top to bottom, when the line pass under the top of the index finger the user must perform a left click to fix the right offset where we must put the mouse cursor.

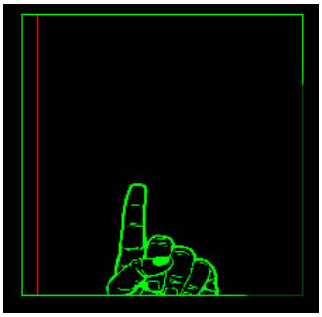

Figure 41: Finger Calibration. In this picture you can see the finger calibration part which enables to compute the offset between the top your index finger, where we want the mouse cursor is positioned, and the center of the the user's hand select to be the reference for the mapping.

Saving and Loading

Another important feature we add is the possibility to *load* a previously saved calibration. When a new user is recognized the window reported in 42 is proposed to the user enabling the possibility to perform a new calibration or load an existing one.

Figure 42: Start Calibration Window. In this picture you can see the start calibration window which enables to load an existent already saved calibration.

Talking about *saving* the calibration parameters, SimpleOpenNI already provide the possibility to do that:

```
kinect.saveCalibrationDataSkeleton(userId, "./calibration/calibration.skel")
```

We also add the possibility to save all the other informations collected during the calibration phase, or rather all the mapping point retrieved during the point calibration stage and the offset where we put the mouse cursor with respect to the hand joint detected by the framework.

4.6.3 Smoothing data

During our development process we have discovered the fact that the *skeleton tracking is quite noisy*, hence, we needed to apply some *approximations* and *smoothing* procedure to enable pointing on specific icons or writing with Ardesia possible without too much oscillations of the mouse position. SimpleOpenNI provides a specific function to define smoothing on the skeleton data:

```
// set the smoothing factor for the skeleton
kinect.setSmoothingSkeleton((float) 0.2);
```

We added another additional approximation taking the *mean* of the last 7 points. The introduction of all this forms of approximation and smoothing has a bad counter-indication, it generates a sort of lag in the tracking. We tried to find trade-off between this two side of the same coin, maybe in future a more accurate study can be conducted on this specific topic.

4.6.4 Notification System

In Figure 43 you can see an instance of the possible *notification feedbacks* displayed in the top right corner of the screen to inform the user about the event detected during the user-tracking process like can be a new user, the current user is lost, the skeleton calibration is completed successfully and so on. We decide to add this features because as explained in the example 4.3.3, the framework is able to trigger some callbacks when specific predefined events happen.

Figure 43: Notification System. In this picture you can see the notification system used to notify the detected events during the user-tracking process.

4.6.5 Internationalization

The proposed solution was *internationalized* enabling the possibility to choose between two languages of utilization: *English* and *Italian*. We made use of the well known Java technique based on "*ResourceBundles*"[5] to add this particular feature. This proposed solution helps to maintain a single source code base for all language versions, in future it will be possible to add other languages in case of necessity. We designed a predefined set of string stored in different files containing the various translations.

5 http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/

5 VALIDATION

In this chapter we will present a detailed description of the *results* obtained and data collected during our test phase, but firstly we will mention all the *conferences and events* we have participated showing our demo of the solution developed. During each test session we proposed to the various a testers a *questionnaire* to fill asking some general questions about the *Kinect* and the *Interactive Whiteboard (IWB)*. Then, we reported all the outcomes collected tracking if the various users were able to complete the calibration and perform some predefined actions or activities.

5.1 RESEARCHERS' NIGHT 2011

The "*Researchers' Night*" is a Europe-wide event bringing together the public at large and researchers once a year on the fourth Friday of September. In 2011 it took place on 23 September in over 800 venues of 320 European cities in 32 countries. It took place also in our city (Trento) and during the Researchers' Night "normal people" have the unique opportunity to discover research facilities that are usually not open to public (laboratories, research centres, museum collections), to use the most recent technologies and instruments with the guidance of scientists, to participate in experiments, competitions and quizzes, to watch demonstrations and simulations, to exchange ideas and to party with the researchers. For one night, everyone can be a scientist. We took part with a stand where we allowed "*normal people*" to try our emulation of the *IWB*. We collected 13 tests, we asked to the the users at the end of the test to fill a questionnaire, we also added to this questionnaire informations about the *progress of the test*. All the results obtained are presented in last section 5.4 of this chapter. The results obtained depends more from the fact that the user was able to perform a good calibration and from the fact the user was able to assume a correct pose with the shoulder parallel to the device, staying in a position not too much far from the wall and keeping the arms not too close to the body. All these expedient enables a good skeleton tracking and enables to perform the different activities in most of the cases without difficulties.

Figure 44: "Researchers' Night" in Trento

Figure 45: Our stand at "Researchers' Night" in Trento

5.2 UNA RETE DI LAVAGNE

We participated also to a conference "*Una Rete di Lavagne*" on 1 October, 2011. This conference was dedicated to discuss about the introduction at school of the *IWB*. At this conference has taken part over a hundred teachers. Here we introduced this project and showed a simple demo of the emulated *IWB*. During this conference we collected various opinion and suggestions to improve our project and for future works from the teachers, other researchers and experts in this specific field.

5.3 PRESENTATION DURING OPEN DAY

A demo of the software has been presented to future university students during the *Open Day* at the university. We hope the university will have more students the next year.

5.4 RESULTS AND STATISTICS

In this section we will analyse all the results and statistics relative to the data we collected during the tests performed by the users. As we have previously mentioned all the informations reported in this section come from the test performed during the "Researchers' Night" in Trento. We have decided to conduct our tests during this event because we want to see the impact on "normal people" of a technology like this one. We have chosen to not focus our tests only on students and teachers or people directly connected to the world of education but to extend our domain of interest to all the other people, in fact, at the test took part developers, journalists, video makers but also students and teachers. In the test we haven't

Figure 46: Una Rete di Lavagne

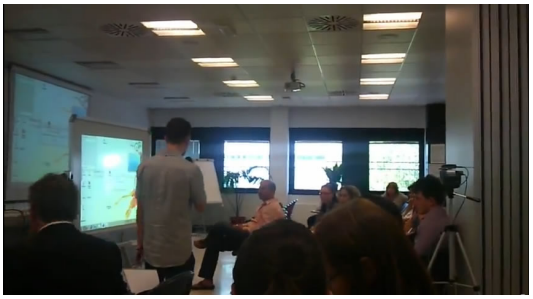

Figure 47: Demo during the conference "Una Rete di Lavagne"

involved children because the board was mounted too high and because the tracking skeleton features at the moment don't work well with small bodies. We asked to the user to fill a questionnaire collecting "background" informations, then we added the outcomes about the progress of the test such as the completion of the calibration phase or the fact that the users was able to perform some predefined actions like open an application or write some on the board using Ardesia[1], a free digital sketchpad.

5.4.1 Personal informations

In Figure 48 we will present the data collected during the test phase relative to the personal informations belonging to the people have taken part to the tests. As you can see the majority of the people have participated was men and all of them was "*technology addicted*" in the sense that they use daily technology devices or computers. Naturally, this aspect can helped them to easy understand the operating mode of the device and use it in a better way. This good relationship with computers and technology devices in general is also explained by the fact

1 http://code.google.com/p/ardesia/

the majority of them have an age comprised between 20 and 30 years, but it is not always the case.

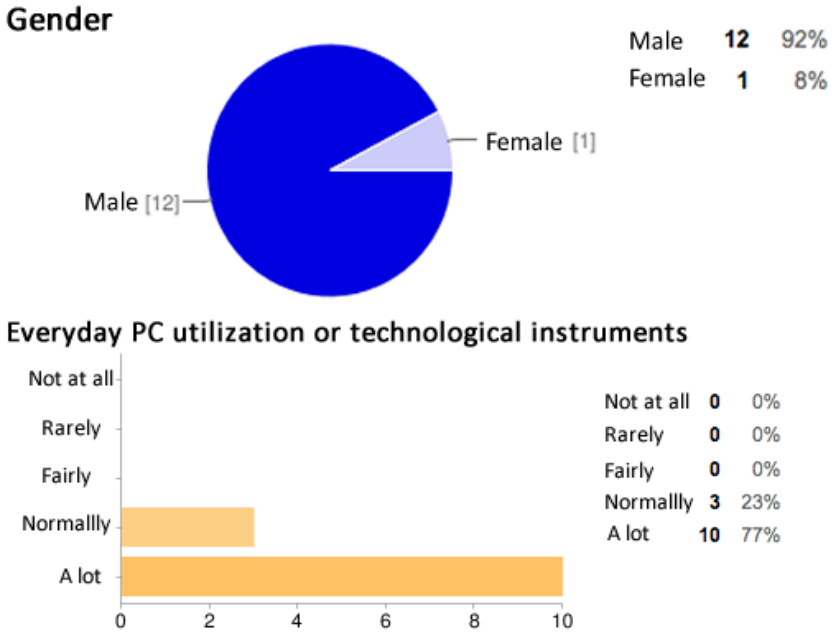

Figure 48: Personal informations results. In this picture we report the data collected during the test phase relative to the personal informations of the users participate our test phase.

5.4.2 Kinect

Once we have understand the "*type*" of people have taken part to our testing stage in terms of personal informations and the relationship they have with technology in general, we asked them some questions regarding the *Kinect*: if they knew the *Kinect*, if they have ever used one or even owned one. The results collected are illustrated in the Figures 49, 50, 51.

In Figure 49 we report how many users knew the *Kinect*, the 70% already known it, but only the 40% have already used it, as you can see in Figure 50. The fact that most of the people involved in the test already known it is explained from the popularity of the *Kinect*, having sold 8 million units in its first 60 days on the market, *Kinect* has claimed the Guinness World Record of being the "fastest selling consumer electronics device"[14]. Microsoft also spent a lot of money for commercial especially in Italy, so probably most of the people have seen the spot on TV but in practice not many of them have ever used it, in fact only the 40% have interact with it at least once as depicted in Figure 50. The percentage of the people own it is very low, just above 20%, this value can be derived from the

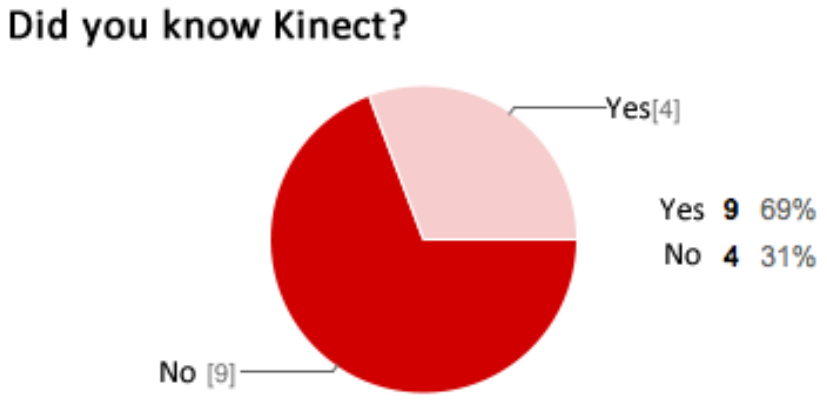

Figure 49: Did you know Kinect?. In this figure we represent how many user already known the *Kinect*.

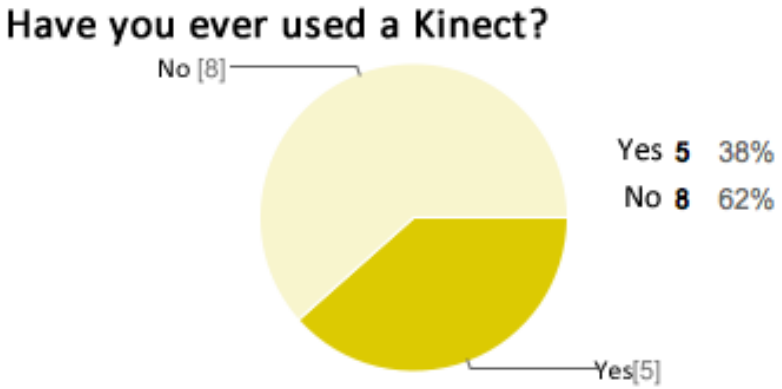

Figure 50: Have you ever used the Kinect?. In this figure we represent how many user have already used the *Kinect* during their life.

fact that in Europe the device diffusion is quite slow with respect of the U.S. and surely the financial crisis has a huge clout on the consumers, the *Kinect* cannot be defined an essential good. All this numbers influence a lot the results of the test because the fact that most of them already knows the *Kinect* could help, but the fact that more than half hasn't interacted in the past with the device explains why during the first attempt of the calibration a half of the user miss it, but we need to say this is not the only one motivation, we will discover in the next sections why.

We asked also the motivation why they bought it but in only one case the user has directly bought it, in the other cases it is a gift for Christmas.

5.4.3 Interactive Whiteboard

In the questionnaire we proposed to our testers we have also included some questions about the level of knowledge relative to the *IWB*'s world. The data col-

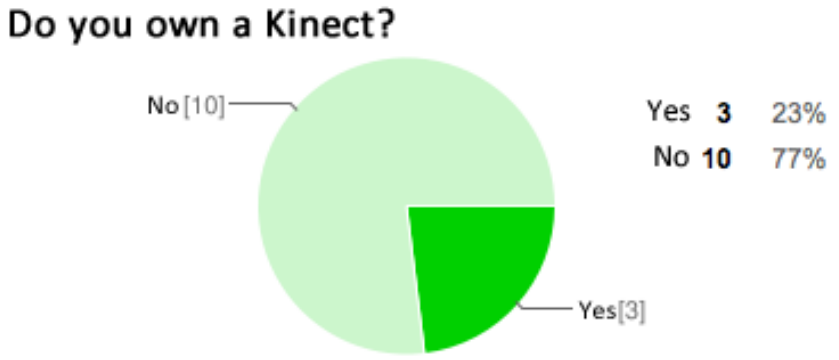

Figure 51: Do you own a Kinect?. In this figure we represent how many user already owned a *Kinect*.

lected is quite similar to the information regard the *Kinect* as you can understand from Figure 52 and 53, but in this last case more users have already interact with a traditional *IWB*, probably because a large set of our testers are students of the university or senior high school and probably because they use it during lessons. This aspect was positive because enables to the users a fast comprehension of the different modalities and interaction methods proposed by an *IWB* in general and in particular from our proposed solution.

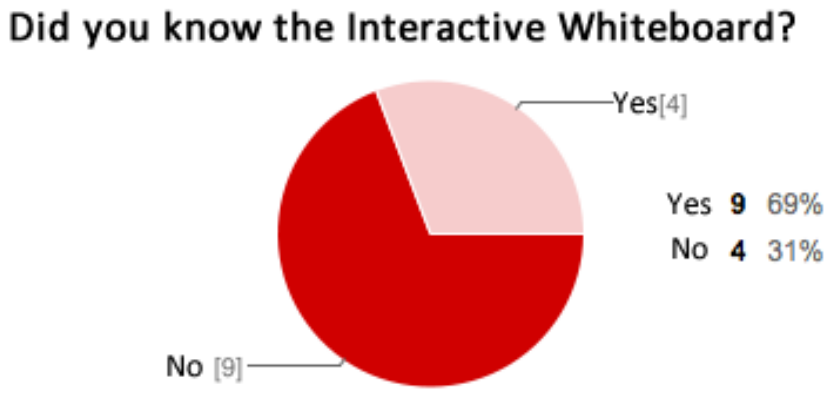

Figure 52: Did you know the Interactive Whiteboard?. In this figure we represent how many user already known the *IWB*.

We asked also if they think the *IWB* is a useful tool for teaching activities and improving the quality of lessons, the different opinions are reported in Figure 54. Only a few of them are pretty sure it can helps to improve the traditional education methodology but the majority of the users involved in the test consider the *IWB* a good tool to increase the level of participation during the lessons at school or university.

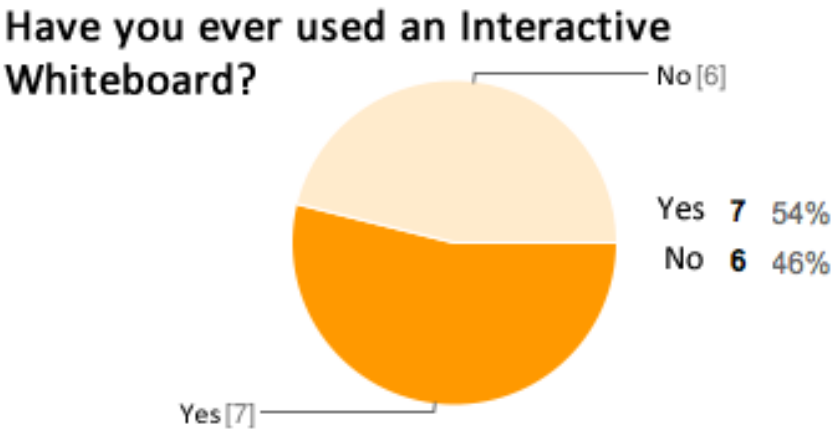

Figure 53: Have you ever used an Interactive Whiteboard?. In this figure we represent how many user have already used the an *IWB* during their life.

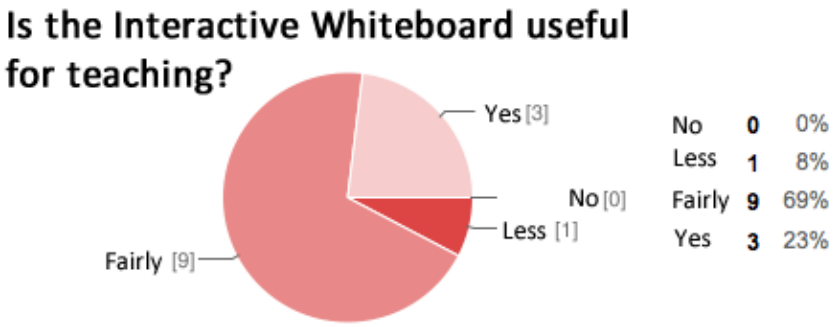

Figure 54: In your personal opinion can be the Interactive Whiteboard useful for teaching?. In this figure we represent how many user consider the *IWB* a useful tool to support teaching activities.

5.4.4 Calibration phase

In the previous sections we have seen all the “background" informations relative to the users we collected through the questionnaire proposed during the tests. The data presented until this point can be defined as “outline" in the sense it supports the key informations presented in this section. Here we will report the results relative to the completion of the calibration stage subdivided for each attempt. As you can see from Figure 55 the users took at *maximum three tentative to complete the calibration phase* and the majority have completed it in two attempts. During the first attempt we noted the fact that most of the user had troubles with the mouse click to fix the calibration point because click with a mouse in the air can be tricky and they missed some points. Hence, we didn't label this problem as malfunctioning of the tracking system but we simply said the user aren't able to complete the calibration phase due to the fact they have missed a point because click on a mouse in the air is not precise as we wish. In the future we will need to change the device used for clicking and fixing the calibration point. We used a mouse because the university didn't provide us other specific and suitable devices.

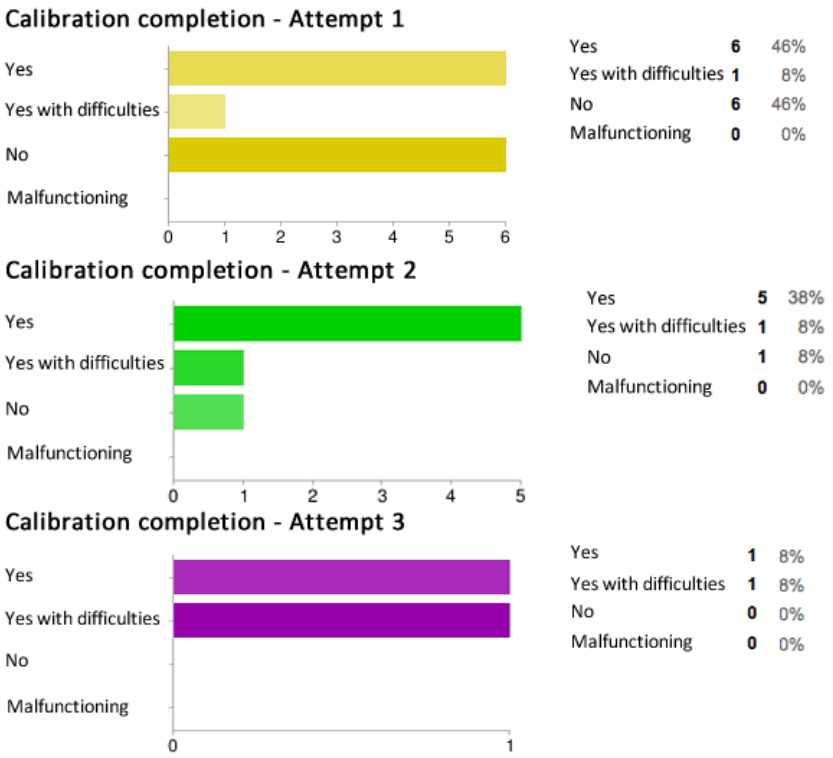

Figure 55: Calibration phase. In this figure we represent the number of attempt the various users take to complete the calibration.

5.4.5 Utilization phase

During every test session we requested to the various users to perform some *predefined actions* or activities to see if they was able to do that, the result obtained are proposed in Figure 56. The majority of the users was able to perform without problems the action we suggested, some of them encountered some difficulties only during writing with *Ardesia*. The outcome of the calibration phase has influenced a lot the possibility to perform the predefined actions: if the user has fixed the points in the right way during the calibration the mapping with the mouse was very precise, hence open a folder or click on a link was a very simple operation. We want to cite a specific case to help you understand what means perform a good calibration, for example the videomaker, who has taken part to our tests, has performed a very precise calibration, this fact has enabled him to obtain a precise mapping and during the writing phase he draw without difficulties a beautiful picture using *Ardesia*.

5.4.6 Final opinion of the user

We are at the end of the validation data collected during the test phase. The last two questions we have asked to the user concerns what they think about the usability of the device and if it can be suitable to support and improve teaching activities. As you can see from Figure 57 and 58 the user have a *positive opinion* in both the cases. We hope this can be somehow derived form the fact they had a positive experience with our software to emulate an *IWB*.

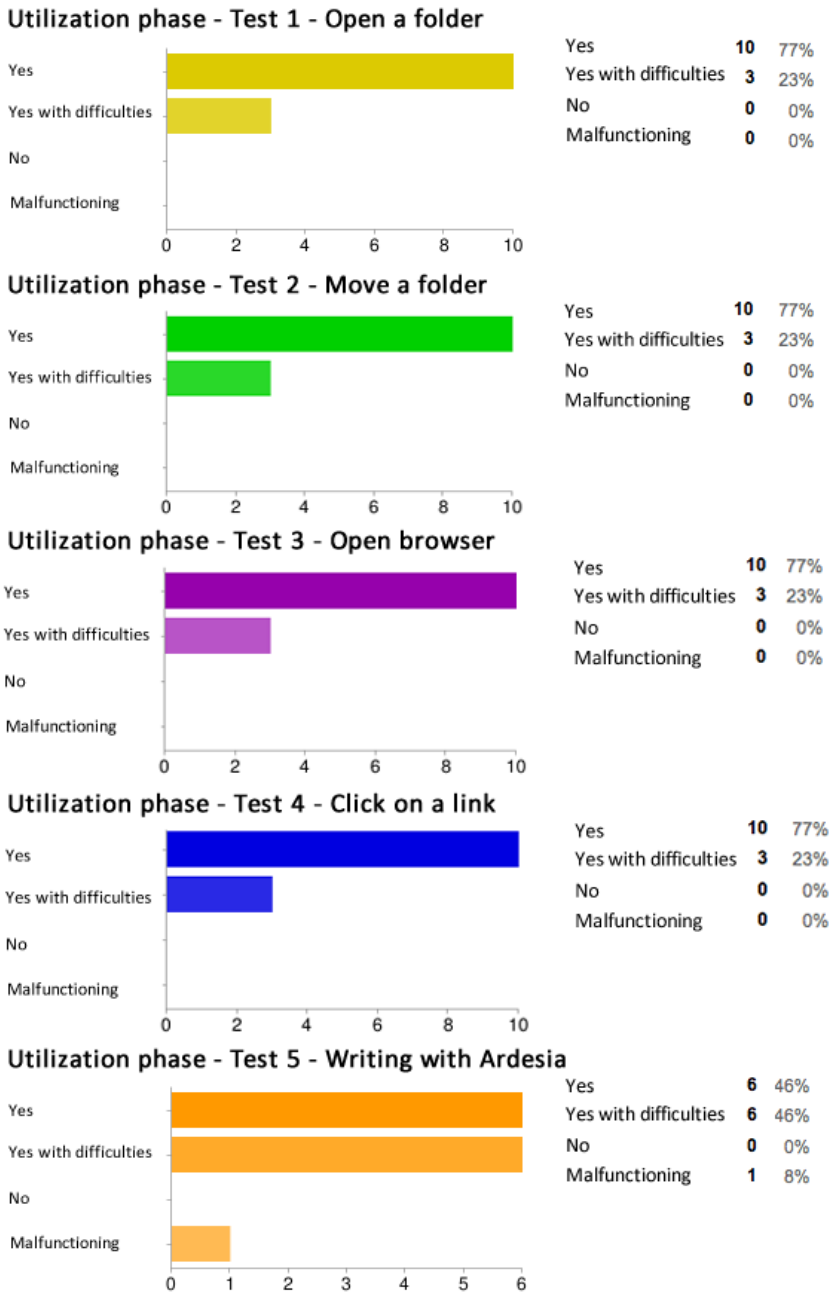

Figure 56: Utilization phase. In this figure we present the results obtained during the test phase, in this case we asked to the various users to perform some predefined actions or activities to see if they was able to do that.

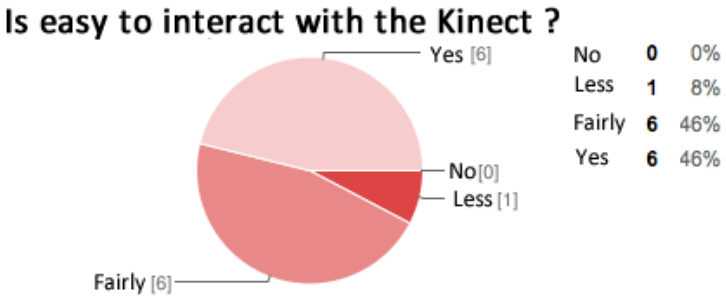

Figure 57: Is easy to interact with the *Kinect*?. In figure it is illustrated what the user think about the usability of the *Kinect*.

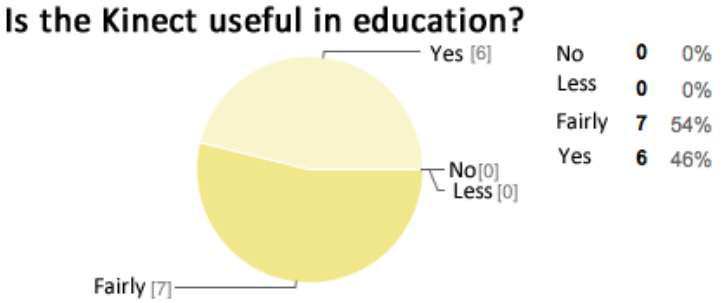

Figure 58: Is the *Kinect* useful in education?. In this picture you can see what the user think about a possible application of the *Kinect* in the education field to support teaching activities.

6 CONCLUSIONS AND FUTURE WORKS

The aim of this thesis was to understand the possibility to introduce the *Kinect* as a concrete tool to support teaching activities and built a low-cost emulation of an *IWB*. The obtained *results* are *satisfactory* and they enable us to say we were able to achieve the objectives we fixed in advance before starting to develop this project. Maybe we needed to accept some compromises due to the fact this is a completely new technology and paradigm of interaction and we had to cope with some problems initially we didn't expected. The fact this is a fresh technology and all the world directly connected is also fresh itself could initially create some problems and it can be a little bit destabilized, but as you can see from this thesis the possibility to build concrete tools is already a reality. Certainly this technology needs to be improved a lot before becoming a stable tool in the educational world but it is in rapid development, hence early we will see the first *Kinect*-based applications appearing on the market, proposing low-cost solution in different fields and probably also in this one. During the progress of this thesis we provided to the reader an overview of this new technology, the basis to start interacting with this kind of devices based on *Natural Interaction (NI)* paradigms and create your first *Kinect*-based application. This thesis can be defined as an input point to this amazing world and surely not the last word.

6.1 FUTURE WORKS

This thesis is the first step in the complex process to create *Kinect*-based application and produce low-cost solution to emulate the traditional expensive *IWB*s present on the actual market. Certainly in the future the hardware and the frameworks designed to interact with it will be subjected to huge improvements. Hence, we can leave to our successors a list of possible directions they can take to continue exploring this amazing field of research. Let's go down into details defining some possible activities and tasks for the future:

1. upgrade the SimpleOpenNI wrapper, installing new version of *OpenNI* and *NITE* because new versions will be released rapidly;
2. develop a similar solution basing the development the Microsoft *SDK*, in order to have a clear tool of comparison on the performance of the tracking system;

3. another solution can be put the *Kinect* in a different position, maybe on top of the board or retry a frontal approach adding a gesture control for different type of activities;
4. try remove completely the use of any kind of device;
5. design different “modality" of utilization, for example define a “writing mode" with some specific parameters to enable a faster recognition removing some smoothing;
6. work on the parameters of smoothing and approximation to reduce the delay introduced at the minimum;
7. add a custom configuration of the points used during the configuration enabling the fact a point can be corrected in the case is missed or fixed badly.

A INSTALLATION

In this section we will list all the installation procedures needed to install the *OpenNI* and *NITE* framework used as reference framework for this thesis. We will indicate what you need to install the *SimpleOpenNI* library and *Processing* to start try the first examples based on this library. The application we developed to emulate an *IWB* doesn't require to install the *Processing* environment and *SimpleOpenNI*, it's only requested the installation of *OpenNI* and *NITE*, all the other libraries are already included in the project. We have already packaged the application for the different operative systems.

Before start describing all the installation procedures we will report all the different versions of the software we have used during our implementation:

- *OpenNI v1.1.041*;
- *Nite v1.3.1.5*;
- *avin2-SensorKinect-28738dc*;
- *SimpleOpenNI Version 0.20*;

The versions reported above are not the last one, hence online you will find the most recent, using the various versions listed above we are sure the software works, with the new versions we didn't already tested. We know the fact that the developer of the SimpleOpenNI library is working on a porting to better support the new versions of *OpenNI* and *NITE*.

A.1 INSTALLING ON WINDOWS

A.1.1 Installing Kinect driver on Windows

The first step is to install the *Kinect* driver:

1. download the driver from `https://github.com/avin2/SensorKinect`;
2. inside the *Bin* folder you will find all the binaries for the different operating systems;

3. install the binary for Windows, we suggested to use the 32 bit version at the moment.

A.1.2 Installing OpenNI on Windows

In order to install *OpenNI* on Windows you need:

1. download the latest stable or unstable *OpenNI* binaries from *OpenNI* website at `http://www.openni.org/Downloads.aspx`;
2. install the binary for Windows, we suggested to use the 32 bit version at the moment.

A.1.3 Installing NITE on Windows

In order to install *NITE* on Windows you need:

1. download the latest stable or unstable *NITE* binaries from *OpenNI* website at `http://www.openni.org/Downloads/OpenNIModules.aspx`, you need to select the option in the combo with the wording *OpenNI Compliant Middleware Binaries*;
2. install the binary for Windows, we suggested to use the 32 bit version at the moment;
3. some versions of the *NITE* installer ask you for a key: insert *oKOIk2JeIBYClPWVnMoRKn5cdY4=*.

A.1.4 Trying out the *OpenNI* and *NITE* samples on Windows

If you want to run the *OpenNI* samples navigate to `C:\ProgramFiles\OpenNI\Samples\Bin\Release` and try out the demo applications. The *NITE* demos are stored in `C:\ProgramFiles\PrimeSense\NITE\Samples\Bin\Release`.

A.1.5 Installing Processing on Windows

As we have previously mentioned this operation is not necessary if you want to run the proposed emulation but we report these instructions in the case you want to test some of the *SimpleOpeNI* examples or create your customized *Processing* code.

In order to install *Processing* on Windows you need:

1. download the latest version of the binaries from *Processing* website at *http://processing.org/download/*;
2. install the binary for Windows, we suggested to take the version without Java.

A.1.6 Installing SimpleOpenNI on Windows

As we have previously mentioned this operation is not necessary if you want to run the proposed emulation but we report these instructions in the case you want to test some of the *SimpleOpeNI* examples or create your customized *Processing* code based on the library. On the *SimpleOpenNI* website at *http://code.google.com/p/simple-openni/wiki/Installation* are indicated the basic instruction to install the library on Windows.

The main operation you need to perform once you have dowloaded the library:

- copy the *SimpleOpenNI.zip* into `C:\Users\'yourusername'\Documents\Processing\libraries`;
- if you don't have this folder then create it.

A.1.7 Run the *IWB* emulation

If you wan to run our proposed emulation of an *IWB*, you must simply run the *bat script* inside our packaged application.

A.2 INSTALLING ON OS X

Before start installing the *OpenNI/NITE* framework we need to satisfy some prerequisites.

Macports is required to be installed. Please check `http://www.macports.org/` for its own dependencies and installation procedure. It requires *XCode* and would need to be installed on your Mac:

- Xcode
- MacPorts

With macports installed and ports list updated install *libtool* and *libusb-devel*:

1. *sudo port install git-core*
2. *sudo port install libtool*

3. *sudo port install libusb-devel +universal*
4. *mkdir Kinect*

Installing Kinect driver on OS X

The first step is to install the *Kinect* driver:

1. *mkdir Kinect/SensorKinect*
2. *cd SensorKinect*
3. download the driver from `https://github.com/avin2/SensorKinect`
4. inside the *Bin* folder you will find all the binaries for the different operating systems
5. *tar xzvf avin2-SensorKinect-28738dc.tar.gz*
6. *cd avin2-SensorKinect-28738dc/Bin/*
7. *tar SensorKinect-Bin-MacOSX-v5.0.1.32.tar.bz2*
8. *cd SensorKinect-Bin-MacOSX-v5.0.1.32*
9. *sudo ./install.sh*

A.2.1 Installing OpenNI on OS X

Now, we can start installing *OpenNI*:

1. *mkdir Kinect/OpenNI*
2. *cd OpenNI*
3. download the latest stable or unstable *OpenNI* binaries for Mac OS X from *OpenNI* website at `http://www.openni.org/Downloads.aspx`;
4. *tar xvf OpenNI-Bin-MacOSX-v1.1.0.41.tar.bz2*
5. *cd OpenNI-Bin-MacOSX-v1.1.0.41*
6. *sudo ./install.sh*

This will install the OpenNI libraries at */usr/lib*.

A.2.2 Installing NITE on OS X

In order to install *NITE* on OS X you need:

1. *mkdir Kinect/NITE*
2. *cd NITE*
3. download the latest stable or unstable *NITE* binaries from *OpenNI* website at `http://www.openni.org/Downloads/OpenNIModules.aspx`, you need to select the option in the combo with the wording *OpenNI Compliant Middleware Binaries*;
4. *tar xvf NITE-Bin-MacOSX-v1.3.1.5.tar.bz2*
5. *cd Nite-1.3.1.5*
6. *sudo ./install.sh*

During installation, provide the following (free) PrimeSense key if requested: *oKOIk2JeIBYClPWVnMoRKn5cdY4=*.

A.2.3 Trying out the samples on OS X

If you want to run the *OpenNI* samples navigate to `OpenNI/Samples/Bin/Release` and try out the demo applications. The *NITE* demos are stored in `NITE/Samples/Bin/Release`.

A.2.4 Installing Processing on OS X

As we have previously mentioned this operation is not necessary if you want to run the proposed emulation but we report these instructions in the case you want to test some of the *SimpleOpeNI* examples or create your customized *Processing* code.

In order to install *Processing* on OS X you need:

1. download the latest version of the binaries from *Processing* website at `http://processing.org/download/`;
2. install the binary for OS X, we suggested to take the version without Java.

A.2.5 Installing SimpleOpenNI on OS X

As we have previously mentioned this operation is not necessary if you want to run the proposed emulation but we report these instructions in the case you

want to test some of the *SimpleOpeNI* examples or create your customized *Processing* code based on the library. On the *SimpleOpenNI* website at `http://code.google.com/p/simple-openni/wiki/Installation` are indicated the basic instruction to install the library on OS X.

The main operation you need to perform once you have dowloaded the library:

- copy the *SimpleOpenNI.zip* into */Users/'your username'/Documents/Processing/libraries*;
- if you don't have this folder then create it.

A.2.6 Run the *IWB* emulation

If you wan to run our proposed emulation of an *IWB*, you must simply run the *sh script* (*sh run.sh*) inside our packaged application.

A.3 INSTALLING ON LINUX

We tested the installation on both 32 and 64 bit Linux and without no serious problems but we suggest to use a 32 bit version. We would recommend *Ubuntu 10.10* or greater. We had done the installation on *Ubuntu 11.04*.

Before start installing the *OpenNI/NITE* framework we need to satisfy some prerequisites.

We suggest to update your Linux distribution before to start with the well-known command:

```
sudo apt-get update
```

You need to install these files which are necessary for proper installation of drivers:

1. *sudo apt-get install mono-complete*
2. *sudo apt-get install libusb-1.0-0-dev*
3. *sudo apt-get install freeglut3-dev*
4. *mkdir Kinect*

A.3.1 Installing Kinect driver on Linux

1. *mkdir Kinect/SensorKinect*
2. *cd SensorKinect*

3. download the driver from `https://github.com/avin2/SensorKinect`
4. inside the *Bin* folder you will find all the binaries for the different operating systems
5. *tar Sensor-Bin-Linux32-v5.0.1.32.tar.bz2*
6. *cd Sensor-Bin-Linux32-v5.0.1.32*
7. *cd Platform/Linux-x86/CreateRedist/*
8. *sudo ./RedistMaker*
9. *cd ../Redist*
10. *sudo ./install.sh*

A.3.2 Installing OpenNI on Linux

Now, we can start installing *OpenNI*:

1. *mkdir Kinect/OpenNI*
2. *cd OpenNI*
3. download the latest stable or unstable *OpenNI* binaries for Mac OS X from *OpenNI* website at `http://www.openni.org/Downloads.aspx`;
4. *tar xvf OpenNI-Bin-Linux32-v1.1.0.41.tar.bz2*
5. *cd OpenNI-Bin-Linux32-v1.1.0.41*
6. *sudo ./install.sh*

A.3.3 Installing NITE on Linux

In order to install *NITE* on OS X you need:

1. *mkdir Kinect/NITE*
2. *cd NITE*
3. download the latest stable or unstable *NITE* binaries from *OpenNI* website at `http://www.openni.org/Downloads/OpenNIModules.aspx`, you need to select the option in the combo with the wording *OpenNI Compliant Middleware Binaries*;

4. *tar xvf NITE-Bin-Linux32-v1.3.1.5.tar.bz2*
5. *cd Nite-1.3.1.5*
6. *sudo ./install.sh*

Use this license if asked during the installation: *0KOIk2JeIBYClPWVnMoRKn5cdY4=*.

A.3.4 Trying out the samples on OS X

If you want to run the *OpenNI* samples navigate to `OpenNI/Samples/Bin/Release` and try out the demo applications. The *NITE* demos are stored in `NITE/Samples/Bin/Release`.

Testing if everything is working fine Connect you kinect and run the samples from the OpenNI folder cd /kinect/OpenNI/Samples/Bin/Release/ (assuming the kinect folder is in home folder, otherwise go to the respective folder) ./NiViewer You should get the depth map and image stream on your window.

A.3.5 Installing Processing on Linux

As we have previously mentioned this operation is not necessary if you want to run the proposed emulation but we report these instructions in the case you want to test some of the *SimpleOpeNI* examples or create your customized *Processing* code.

In order to install *Processing* on Linux you need:

1. download the latest version of the binaries from *Processing* website at `http://processing.org/download/`;
2. install the binary for Linux, we suggested to take the version without Java.

A.3.6 Installing SimpleOpenNI on Linux

As we have previously mentioned this operation is not necessary if you want to run the proposed emulation but we report these instructions in the case you want to test some of the *SimpleOpeNI* examples or create your customized *Processing* code based on the library. On the *SimpleOpenNI* website at `http://code.google.com/p/simple-openni/wiki/Installation` are indicated the basic instruction to install the library on Linux.

The main operation you need to perform once you have dowloaded the library:

- copy the *SimpleOpenNI.zip* into */sketchbook/libraries*;

- if you don't have this folder then create it.

A.3.7 Run the *IWB* emulation

If you wan to run our proposed emulation of an *IWB*, you must simply run the *sh script* (*sh run.sh*) inside our packaged application.

A.4 INSTALLING MICROSOFT KINECT SDK

A.4.1 System requirements

In this section we report some useful informations regarding the requirements needed to run an application *Kinect*-based in the Windows environment, the software requirements are specific for the Kinect for Windows *Software Development Kit (SDK)* Beta but the hardware part can be taken in consideration as general requirements when you are developing something involving the *Kinect*.

Hardware:

- 32 bit (x86) or 64 bit (x64) processor;
- dual-core 2.66-GHz or faster processor;
- dedicated USB 2.0 bus;
- 2 GB RAM;
- a special USB/power cabling as we have previously described in the section 2.1.3 when we talk about the *Kinect* hardware.

Software:

- Microsoft Visual Studio 2010 Express or other Visual Studio 2010 edition;
- .NET Framework 4.0.

For C++ Skeletal Viewer Samples:

- Microsoft DirectX SDK - June 2010 or later version;
- Runtime for Microsoft DirectX 9.

For Speech Samples:

- Microsoft Speech Platform Runtime, version 10.2, select 32-bit if you are running 32-bit Windows. If you have 64-bit Windows, we suggest that you install both the 32-bit and 64-bit runtime;

- Microsoft Speech Platform - Software Development Kit, version 10.2, 32-bit or 64-bit according to your Windows installation;
- Kinect for Windows Runtime Language Pack, version 0.9.

A.4.2 Installation

To install the *SDK* Beta we reported the instruction taken from `http://kinectforwindows.org/download/`:

1. Make sure the *Kinect* device is not plugged in to the *USB* port on your computer.
2. If you installed any previous *Kinect* for Windows *SDK*, uninstall it first. Similarly, you must remove any other drivers for the *Kinect* device.
3. Close Visual Studio.
4. Note: You must close Visual Studio before installing the *SDK* and then restart it after installation to pick up the KINECTSDK_DIR environment variables that the *SDK* requires.
5. From the download location for the *SDK* Beta package, double-click the link for your Windows installation: 32-bit or 64-bit.
6. Once the *SDK* has completed installing successfully, ensure your *Kinect* device is plugged in to an external power source then plug the *Kinect* into the *PC*'s *USB* port.
7. The drivers will load automatically. Your *Kinect* should now be working correctly.
8. To run the samples, ensure that the DirectX and Speech prerequisites are installed. They are listed in the previous section.

B SOURCE CODE

B.1 FIRST PROGRAM

Listing B.1: DepthImage.pde

```
1  import SimpleOpenNI.*;
2
3  SimpleOpenNI  context;
4
5  void setup()
6  {
7    context = new SimpleOpenNI(this);
8
9    // mirror is by default enabled
10   context.setMirror(true);
11
12   // enable depthMap generation
13   context.enableDepth(640,480,30);
14
15   // enable RGB image generation
16   context.enableRGB(640,480,30);
17
18   size(context.depthWidth() + context.rgbWidth() + 10, context.rgbHeight
         ());
19 }
20
21 void draw()
22 {
23   // update the cam
24   context.update();
25
26   background(200,0,0);
27
28   // draw depthImageMap
29   image(context.depthImage(),0,0);
30
31   // draw RGBImageMap
32   image(context.rgbImage(),context.depthWidth() + 10,0);
33 }
```

B.2 SKELETAL TRACKING

Listing B.2: User.pde

```
1  import SimpleOpenNI.*;
2
3  SimpleOpenNI  context;
4
5  void setup()
6  {
7    context = new SimpleOpenNI(this);
8
9    // enable depthMap generation
10   context.enableDepth();
11
12   // enable skeleton generation for all joints
13   context.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
14
15   background(200,0,0);
16
17   stroke(0,0,255);
18   strokeWeight(3);
19   smooth();
20
21   size(context.depthWidth(), context.depthHeight());
22 }
23
24 void draw()
25 {
26   // update the cam
27   context.update();
28
29   // draw depthImageMap
30   image(context.depthImage(),0,0);
31
32   // draw the skeleton if it's available
33   if(context.isTrackingSkeleton(1))
34     drawSkeleton(1);
35 }
36
37 // draw the skeleton with the selected joints
38 void drawSkeleton(int userId)
39 {
40   // to get the 3d joint data
41   /*
42   PVector jointPos = new PVector();
43   context.getJointPositionSkeleton(userId,SimpleOpenNI.SKEL_NECK,jointPos
         );
44   println(jointPos);
45   */
46
47   context.drawLimb(userId, SimpleOpenNI.SKEL_HEAD, SimpleOpenNI.SKEL_NECK
         );
```

```
48
49    context.drawLimb(userId, SimpleOpenNI.SKEL_NECK, SimpleOpenNI.
          SKEL_LEFT_SHOULDER);
50    context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER, SimpleOpenNI.
          SKEL_LEFT_ELBOW);
51    context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW, SimpleOpenNI.
          SKEL_LEFT_HAND);
52
53    context.drawLimb(userId, SimpleOpenNI.SKEL_NECK, SimpleOpenNI.
          SKEL_RIGHT_SHOULDER);
54    context.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER, SimpleOpenNI
          .SKEL_RIGHT_ELBOW);
55    context.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW, SimpleOpenNI.
          SKEL_RIGHT_HAND);
56
57    context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER, SimpleOpenNI.
          SKEL_TORSO);
58    context.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER, SimpleOpenNI
          .SKEL_TORSO);
59
60    context.drawLimb(userId, SimpleOpenNI.SKEL_TORSO, SimpleOpenNI.
          SKEL_LEFT_HIP);
61    context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_HIP, SimpleOpenNI.
          SKEL_LEFT_KNEE);
62    context.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_KNEE, SimpleOpenNI.
          SKEL_LEFT_FOOT);
63
64    context.drawLimb(userId, SimpleOpenNI.SKEL_TORSO, SimpleOpenNI.
          SKEL_RIGHT_HIP);
65    context.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP, SimpleOpenNI.
          SKEL_RIGHT_KNEE);
66    context.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_KNEE, SimpleOpenNI.
          SKEL_RIGHT_FOOT);
67 }
68
69 // ------------------------------------------------------------------
70 // SimpleOpenNI events
71
72 void onNewUser(int userId)
73 {
74   println("onNewUser - userId: " + userId);
75   println("  start pose detection");
76
77   context.startPoseDetection("Psi",userId);
78 }
79
80 void onLostUser(int userId)
81 {
82   println("onLostUser - userId: " + userId);
83 }
```

```
84
85 void onStartCalibration(int userId)
86 {
87   println("onStartCalibration - userId: " + userId);
88 }
89
90 void onEndCalibration(int userId, boolean successfull)
91 {
92   println("onEndCalibration - userId: " + userId + ", successfull: " +
       successfull);
93
94   if (successfull)
95   {
96     println("  User calibrated !!!");
97     context.startTrackingSkeleton(userId);
98   }
99   else
100  {
101    println("  Failed to calibrate user !!!");
102    println("  Start pose detection");
103    context.startPoseDetection("Psi",userId);
104  }
105 }
106
107 void onStartPose(String pose,int userId)
108 {
109   println("onStartPose - userId: " + userId + ", pose: " + pose);
110   println(" stop pose detection");
111
112   context.stopPoseDetection(userId);
113   context.requestCalibrationSkeleton(userId, true);
114
115 }
116
117 void onEndPose(String pose,int userId)
118 {
119   println("onEndPose - userId: " + userId + ", pose: " + pose);
120 }
```

B.3 ECLIPSE INTEGRATION

Listing B.3: Eclipse Intergration Example

```
1 public class Main extends PApplet {
2
3   //
      ////////////////////////////////////////////////////////////////
4   // Fields
```

```
5   //
        ------------------------------------------------------------------------

6
7   /** The logger. */
8   private static Logger logger = Logger.getLogger(Main.class);
9
10  /** The config window. */
11  private UIConfigurationWindow configWindow;
12
13  /** The kinect. */
14  private PApplet kinect;
15
16  /** The open kinect. */
17  private PApplet openKinect;
18
19  //
        ------------------------------------------------------------------------

20  // Constructors
21  //
        ------------------------------------------------------------------------

22
23  //
        ------------------------------------------------------------------------

24  // Methods
25  //
        ------------------------------------------------------------------------

26
27  // constructor
28  /* (non-Javadoc)
29   * @see processing.core.PApplet#setup()
30   */
31  public void setup() {
32
33    size(50, 50);
34
35    logger.debug("--- Enter");
36
37    // openKinect = new Tilt();
38    // openKinect.init();
39
40    kinect = new KinectBackSkeleton();
41
42    configWindow = new UIConfigurationWindow(kinect);
43
44    logger.debug("--- Exit");
```

```
45   }
46
47   //
       ____________________________________________________________________

48   // Main method
49   //
       ____________________________________________________________________

50
51   /**
52    * The main method.
53    *
54    * @param args the arguments
55    */
56   public static void main(String args[]) {
57     logger.debug("--- Enter - Paramenters = args: " + args.toString());
58
59     PApplet.main(new String[] { "--present", "it.unitn.kinect.Main" });
60
61     logger.debug("--- Exit");
62   }
63
64   @Override
65   public void destroy() {
66     kinect.destroy();
67     super.destroy();
68   }
69 }
```

B.4 BACKGROUND REMOVAL

Listing B.4: Background removal function

```
1  /**
2    * Draw user.
3    */
4   void drawUser() {
5
6     // retrieve the current image from the RGB camera
7     rgbImage = kinect.rgbImage();
8
9     // load the pixel composing the colored image coming from the RGB
          camera
10    rgbImage.loadPixels();
11
12    // find out what are the pixels belonging to each single user
13    userMap = kinect.getUsersPixels(SimpleOpenNI.USERS_ALL);
14
15    // populate the new image pixel array
```

```
16      for (int y = 0; y < kinect.rgbHeight(); y += 1) {
17        for (int x = 0; x < kinect.rgbWidth(); x += 1) {
18          int i = x + y * kinect.rgbWidth();
19          if (userMap[i] != 0) {
20            // if is a pixel belonging to the user copy the
21            // correspondent pixel in the RGB image in the new user
22            // image
23            userImage.pixels[i] = rgbImage.pixels[i];
24          } else {
25            // else take a black pixel
26            userImage.pixels[i] = 0x00000000;
27          }
28        }
29      }
30
31      // update the pixel in the RGB image
32      rgbImage.updatePixels();
33
34      // update the new image pixels
35      userImage.updatePixels();
36
37      // load the new user image in the feedback window in position 0,0
38      image(userImage, 0, 0);
39
40    }
```

B.5 STATE CONTEXT

Listing B.5: State Context

```
1  public class StateContext {
2
3    /** The state. */
4    private State state;
5
6    /** The total point number. */
7    private int totalPointNumber;
8
9    // Creates a new Context with the specified state.
10   /**
11    * Instantiates a new state context.
12    *
13    * @param state the state
14    * @param totalPointNumber the total point number
15    */
16   public StateContext(State state, int totalPointNumber) {
17     this.state = state;
18     this.totalPointNumber = totalPointNumber;
19   }
20
```

```
21  // Returns the state.
22  /**
23   * Gets the state.
24   *
25   * @return the state
26   */
27  public State getState() {
28    return state;
29  }
30
31  // Sets the state.
32  /**
33   * Sets the state.
34   *
35   * @param state the new state
36   */
37  public void setState(State state) {
38    this.state = state;
39  }
40
41  /**
42   * The next() method performs performs a state transition to the next
       state.
43   * Using the State pattern, we delegate this behavior to our current
       state
44   * object.
45   */
46  public void next() {
47    state.next(this);
48  }
49
50  /**
51   * Gets the total point number.
52   *
53   * @return the total point number
54   */
55  public int getTotalPointNumber() {
56    return totalPointNumber;
57  }
58
59  /**
60   * Sets the total point number.
61   *
62   * @param totalPointNumber the new total point number
63   */
64  public void setTotalPointNumber(int totalPointNumber) {
65    this.totalPointNumber = totalPointNumber;
66  }
```

BIBLIOGRAPHY

[1] 3M (2008), *Dispersive Signal Touch Technology - Technology Profile*, URL http://multimedia.3m.com/mws/mediawebserver?mwsId=66666UuZjcFSLXTtmxTXoxfaEVuQEcuZgVs6EVs6E666666--&fn=DST%20Tech%20Profile.pdf, [Online; accessed 12-November-2011].

[2] Anoto Group AB (2011), «Anoto - THE TECHNOLOGY», URL http://www.anoto.com/the-technology-1.aspx, [Online; accessed 12-November-2011].

[3] Borenstein, G. (2011), *Making Things See*, O'Reilly Media / Make, URL http://shop.oreilly.com/product/0636920020684.do, [Online; accessed 12-November-2011].

[4] Brekelmans, J. (2011), «Microsoft Kinect SDK vs PrimeSense OpenNI», URL http://www.brekel.com/?page_id=671, [Online; accessed 29-October-2011].

[5] BusinessWire (2011), «PrimeSense Teams Up with ASUS to Bring Intuitive PC Entertainment to the Living Room with WAVI Xtion», URL http://www.businesswire.com/news/home/20110103005276/en/PrimeSense-Teams-ASUS-Bring-Intuitive-PC-Entertainment, [Online; accessed 03-October-2011].

[6] Group, N. U. I. (2011), «Natural User Interface», URL http://wiki.nuigroup.com/Natural_User_Interface, [Online; accessed 23-October-2011].

[7] Han, J. Y. (2005), «Low-cost multi-touch sensing through frustrated total internal reflection», in «Proceedings of the 18th annual ACM symposium on User interface software and technology», UIST '05, pp. 115–118, ACM, New York, NY, USA, URL http://doi.acm.org/10.1145/1095034.1095054, [Online; accessed 12-November-2011].

[8] Hinchman, W. (2011), «Kinect for Windows SDK beta vs. OpenNI», URL http://labs.vectorform.com/2011/06/windows-kinect-sdk-vs-openni-2/, [Online; accessed 29-October-2011].

[9] Holzner, S. (2011), «State Design Pattern», URL http://sourcemaking.com/design_patterns/state, [Online; accessed 12-November-2011].

[10] Hsu, H. J. (2011), «The Potential of Kinect as Interactive Educational Technology», in «2nd International Conference on Education and Management Technology», vol. 13, pp. 334–338, IACSIT Press, Singapore, URL http://www.ipedr.com/vol13/64-T10050.pdf, [Online; accessed 12-November-2011].

[11] Karam, M. and Schraefel, M. (2005), «A taxonomy of Gestures in Human Computer Interaction», Tech. rep., University of Southampton, URL http://eprints.ecs.soton.ac.uk/11149/1/GestureTaxonomyJuly21.pdf, [Online; accessed 13-October-2011].

[12] Khoshelham, K. (2011), «ACCURACY ANALYSIS OF KINECT DEPTH DATA», Tech. rep., ITC Faculty of Geo-information Science and Earth Observation, University of Twente., URL http://www.isprs.org/proceedings/XXXVIII/5-W12/Papers/ls2011_submission_40.pdf, [Online; accessed 13-October-2011].

[13] Lee, J. C. (2008), «Johnny Chung Lee - Projects - Wii», URL http://www.johnnylee.net/projects/wii/, [Online; accessed 12-November-2011].

[14] Microsoft (2011), «Kinect Sales Surpass Ten Million», URL http://www.xbox.com/en-us/press/archive/2011/0308-ten-million-kinects, [Online; accessed 03-October-2011].

[15] Microsoft News Center (2010), «PrimeSense Supplies 3-D-Sensing Technology to "Project Natal" for Xbox 360», URL http://www.microsoft.com/Presspass/press/2010/mar10/03-31PrimeSensePR.mspx?rss_fdn=Press%20Releases, [Online; accessed 03-October-2011].

[16] Mitchell, R. (2010), «PrimeSense releases open source drivers, middleware that work with Kinect», URL http://www.joystiq.com/2010/12/10/primesense-releases-open-source-drivers-middleware-for-kinect/, [Online; accessed 03-October-2011].

[17] OpenNI (2011), *OpenNI User Guide*, OpenNI, version 3 ed., URL http://www.openni.org/images/stories/pdf/OpenNI_UserGuide_v3.pdf, [Online; accessed 03-October-2011].

[18] Pilloli, P. (2011), «WiiLDOS - Wiki», URL http://wiildos.wikispaces.com/, [Online; accessed 05-November-2011].

[19] PrimeSense (2010), *Prime Sensor™NITE 1.3 Algorithms notes*, PrimeSense Inc., version 1.0 ed., [Online; accessed 16-October-2011].

[20] PrimeSense (2010), *Prime Sensor™NITE 1.3 Controls Programmer's Guide*, PrimeSense Inc., version 1.0 ed., [Online; accessed 16-October-2011].

[21] PrimeSense (2011), *PrimeSense™NITE Controls User Guide*, PrimeSense Inc., version 1.0 ed., [Online; accessed 16-October-2011].

[22] Reas, C. and Fry, B. (2010), *Getting Started with Processing*, O'Reilly Media, URL http://shop.oreilly.com/product/0636920000570.do, [Online; accessed 12-November-2011].

[23] Research, M. (2011), *Programming Guide - Getting Started with the Kinect for Windows SDK Beta from Microsoft Research*, Microsoft, beta 1 draft version 1.1 ed., URL http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/docs/ProgrammingGuide_KinectSDK.pdf, [Online; accessed 03-October-2011].

[24] Ronchetti, M., Bosetti, M., Ruffoni, M. and Pilolli, P. (2010), «Potenzialità di ingegnerizzazione della WiiLD», Tech. rep., University of Trento - Department of Information Engineering and Computer Science, URL http://sites.google.com/site/wii4dida/resources/Potenzialit%C3%A0diingegnerizzazionedellaWiiLD.pdf?attredirects=0&d=1, [Online; accessed 25-October-2011].

[25] Shotton, J., Fitzgibbon, A., Cook, M., Sharp, T., Finocchio, M., Moore, R., Kipman, A. and Blake, A. (2011), «Real-Time Human Pose Recognition in Parts from Single Depth Images», Microsoft Research, IEEE, URL http://research.microsoft.com/apps/pubs/default.aspx?id=145347.

[26] Takahashi, D. (2009), «Microsoft games exec details how Project Natal was born», URL http://venturebeat.com/2009/06/02/microsoft-games-executive-\describes-origins-of-project-natal-game-controls/, [Online; accessed 27-September-2011].

[27] Tech-FAQ (2011), «Interactive Whiteboard», URL http://www.tech-faq.com/interactive-whiteboard.html, [Online; accessed 09-November-2011].

[28] TechLearn (2011), «Interactive Whiteboards in Education», Tech. rep., Joint Information Systems Committee, URL http://www.jisc.ac.uk/uploaded_documents/Interactivewhiteboards.pdf, [Online; accessed 05-November-2011].

[29] Ten, S. (2010), «How Kinect depth sensor works – stereo triangulation?», URL http://mirror2image.wordpress.com/2010/11/30/how-kinect-works-stereo-triangulation/, [Online; accessed 03-October-2011].

[30] Valli, A. (2008), «Notes on Natural Interaction», Tech. rep., URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.7674&rep=rep1&type=pdf, [Online; accessed 23-October-2011].

[31] Warren, T. (2010), «Microsoft details early Windows 8 improvements to OEMs», URL http://www.neowin.net/news/microsoft-details-early-windows-8-improvements-to-oems, [Online; accessed 27-September-2011].

[32] Wikipedia (2011), «Interactive whiteboard — Wikipedia, The Free Encyclopedia», URL http://en.wikipedia.org/wiki/Interactive_whiteboard, [Online; accessed 05-November-2011].

[33] Wikipedia (2011), «Processing (programming language) — Wikipedia, The Free Encyclopedia», URL http://en.wikipedia.org/wiki/Processing_%28programming_language%29, [Online; accessed 01-November-2011].

[34] Wikipedia (2011), «ZCam — Wikipedia, The Free Encyclopedia», URL http://en.wikipedia.org/wiki/ZCam, [Online; accessed 03-October-2011].

RINGRAZIAMENTI

Qualcuno probabilmente resterà deluso ma sinceramente sono esausto, quindi sarò molto breve. Non farò nomi ne cognomi così eviterò di dimenticarmi qualcuno. Colgo l'occasione per ringraziare tutte le persone che mi hanno permesso di essere qui in questo momento a festeggiare questo importante traguardo nella mia vita di studente e nella mia vita come persona: ringrazio quindi i miei famigliari, la mia ragazza, i miei amici, i miei compagni di corso, il mio relatore, il mio datore di lavoro, insomma tutte le persone che hanno contribuito, mi hanno aiutato e mi hanno permesso di raggiungere questo importante traguardo come può essere una Laurea Magistrale. Grazie di cuore, davvero.