View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Electronic Thesis and Dissertation Archive - Università di Pisa

UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Informatica

Tesi di Laurea

A FRAMEWORK FOR TEMPORAL ANALYSIS OF SENSOR DATA IN GESTURE RECOGNITION

Relatore

Prof. Antonio Cisternino

Controrelatore

Prof. Giuseppe Prencipe

Candidato

Leonardo Festa

Anno Accademico 2012/2013

Alla mia famiglia

Contents

ABSTRACT **1**

1. INTRODUCTION **2**

2. STATE OF THE ART **4**

2.1 NATURAL USER INTERFACE (NUI) 4

2.1.1 Touch Based Devices *6*

2.1.2 Motion Based Devices *7*

2.1.3 Vision Based Devices *7*

2.2 GESTURE RECOGNITION 8

2.2.1 Gesture Recognition Techniques *9*

Template Matching Models 9

Statistical Classification 10

Neural Networks 10

Skeletal Model Based Approach 11

Syntax or Structure Matching 12

2.2.2 Touch Based Gestures *12*

2.2.3 Vision Based Gestures *14*

2.3 AVAILABLE FRAMEWORKS 15

2.3.1 Proton/Proton++ *16*

Proton 16

Proton++ 19

2.3.2 Deedle *20*

3 TOOLS **22**

3.1 THE .NET FRAMEWORK 22

3.1.1 F# *24*

3.2 LEAP MOTION CONTROLLER 25

3.2.1 Technical characteristics *26*

3.2.2 The Hand Model *26*

Controller 27

Frame 27

Hands 28

Fingers and Tools 30

Gestures 31

Listening Events to Leap Motion Controller 32

3.2.3 Strength and Weakness *33*

3.3 GESTIT 34

3.3.1 Petri Nets *35*

3.3.2 Gesture Description Model *37*

Ground Terms 37
Composition Operators 39
Handlers 40
3.3.3 The Library 40
The Fusion Sensor 40
Gesture Expression 41
Gesture Net 42
Shortcut Operators 42
3.3.4 Using the Library 43
3.3.5 An Example 43

4 DESIGN AND IMPLEMENTATION 47

4.1 ARCHITECTURE 49
4.2 DESIGN CHOICES 50
4.2.1 Gesture Analysis 53
4.3 DEVELOPMENT 56
4.3.1 Data Types 56
4.3.2 Buffer Types 58
The Accumulator Buffers 58
The Sequence Buffers 60
4.3.3 Event Types 66

5 A CASE STUDY: LEAP MOUSE 70

5.1 PROBLEM 70
5.2 DESIGN 72
5.3 IMPLEMENTATION 74
5.3.1 The Leap Event Wrapper 75
LeapActivity 76
LeapSensorEventArgs 77
LeapSensor 78
5.3.2 The Mouse Controller Simulator 80
5.3.3 The Graphical User Interface 81
5.3.4 The Event Logic Module 83
Gestures as Petri Nets 83
Realizing Customized Events 88

6 CONCLUSIONS 94

7 ACKNOWLEDGEMENTS 96

8 BIBLIOGRAPHY 98

Abstract

Natural User Interfaces entered our everyday life and changed the classical approach we had in the human-computer interaction. In the last decade devices as multi-touch screens and various kinds of motion sensors become part of the common glossary, and the interest from industry and the development of new devices have boosted the research in this branch.

The common approach in the application-programming interface (API) of this kind of devices is to provide the user interface developer with a limited set of high-level gestures and all the raw data gathered from the sensors. This approach lacks of flexibility in adding new custom gestures and the processing of raw data is not simple.

The aim of this thesis is to design and develop generic-purpose tools that give to the user the possibility to make statistical analysis on time series of data. The user can use classical high-level gestures and integrate them with customized gestures, in a more design-oriented structure, for an improved maintenance of the code.

1. Introduction

In the last decade, the traditional paradigm for human-computer interaction based on mouse and keyboard has been sided by new and emerging devices in order to create a more natural user experience. In the first place, multi-touch devices emerged followed by passive sensors, such as Microsoft Kinect, whose goal is to observe users making gestures. This last form of interaction is often referred as Natural User Interface (NUI). Gesture and gesture recognition have become common terms now and there is an increasing request of methodologies to describe and implement new gestures.

Natural User Interfaces are a broad range of devices which return various kinds of data with very different nature. In absence of well-defined models and abstractions, the approach taken by the various device SDKs is to provide a fixed set of high-level gestures signaled as atomic events, together with the possibility to directly access to raw data from the sensors.

This approach lacks of flexibility and in most cases it prevents the possibility to add customized gestures; moreover, the processing of raw data is not simple and comes without additional support, because gestures are sometimes complex and develop in a timespan. This kind of gestures requires a temporal analysis, that does not fit well the typical observer pattern used for classic event listening.

The aim of this thesis is to provide a generic-purpose framework that allows programmers to do efficient statistical and mathematical analysis on time series of data, thus allowing the introduction of new gestures recognized by a device with a design-oriented structure, which results in code that is easier to maintain.

2. State of the Art

In this chapter, we will start with a description of the diffusion of the Natural User Interfaces in everyday life, and how they changed our interaction with the computers with respect to the usual mouse and keyboard approach.

Following we will describe the gesture paradigm developed to interact with these systems, and in the end we will see what are the actual frameworks available to build up gestures and to make time analysis on collected data.

2.1 Natural User Interface (NUI)

The human-computer interaction is not exclusively bound anymore only to the use of mouse and keyboard.

With the advent of Wii Remote™ controller (2006) [1] the interaction with game consoles changed, as well as the iPhone® (2007) [2] demonstrates a successful usage for a multi-touch screen and the Microsoft Kinect™ (2010) [3] added the possibility to interact without using or wearing any kind of controller.

Nowadays these technologies surround us, widespread in the everyday life. In some cases we expect them to be present: For example, when you see a mobile phone screen, an ATM or a wall-screen in a museum, probably you expect it to be a multi-touch interactive device and not only a video display.

All this interfaces go under the name of Natural User Interfaces (NUI) [4]. This category includes many devices with heterogeneous nature, but with some common elements.

The first one is that Natural User Interfaces break the WIMP (Windows-Icon-Menu-Pointer) paradigm used in classical Graphical User Interfaces. A second common element is that they are mostly intuitive or easy to assimilate.

For some aspects the Natural User Interfaces have overturn the way to interact: the user does not feel forced to understand the metaphor that will make the system work; in most cases, instead, the user expects the system to understand the movement which intuitively represents the action.

The research in this area is not just the result of the last decade; the first prototypes of multi-touch devices were developed in the '80s [5]. An important part of the success of a Natural User Interface depends from the perception of how much natural it is for the user.

The precision of the devices has constantly grown together with the computational capabilities and the technological advance. This allows the recognition of more complex gestures. The improvement of quality and reactivity in gesture recognition is an important part of the recent success of NUIs.

It is hard to make an exhaustive list of the kinds of devices available on the market and it is not the goal of this thesis. In the next sections, we will make a short overview of the devices that are targeted by our framework and we will describe the technologies used in them.

2.1.1 Touch Based Devices

We call “touch devices” all of the touch-sensitive surfaces that are able to recognize one or more points of contact. The detection of multiple points of contact (so called multi-touch devices) opened the road to the possibility to implement even more complex functionalities than it was possible with older devices.

Under this category are classified the multi-touch screens, tables, tablets, smartphones and touch-pads. The technologies behind the touch devices are various and with different costs; they can be grouped in three basic categories:

Resistive systems;

Capacitive systems;

Surface acoustic wave systems.

Resistive systems consist of a glass panel, covered with a conductive metallic layer and a resistive layer. Spacers hold the two layers apart and a scratch-resistant layer is placed on top of them for protection. When the user touches the screen, the two layers make contact, changing the electrical field, and a process calculates the coordinates of the point of contact.

In capacitive systems, a layer that stores electrical charge is placed on the glass panel of the monitor. When the user touches the monitor with his fingers, some of the charges are transferred to the user, and the charge of the layer changes. On each corner of the monitor, there are circuits that measure the decrease of the charges and send this information to the system. The capacitive system is more efficient than the resistive system.

In surface acoustic wave systems, two transducers (one receiver and one transmitter) are placed along the x and y axes of the glass plate. Reflectors are placed on the glass: as its name suggests, they reflect the signal sent by

one transducer to the other. The receiving transducer is able to tell if the wave is been disturbed by a touch event at any instant, and can locate it accordingly.

2.1.2 Motion Based Devices

The name “motion-based devices” describes all of the devices that must be worn or moved to detect the movements of the user.

An example of this kind of devices are data-gloves, gloves that users has to wear to map finger movements. Similar devices use a combination of sensors like gyroscopes and accelerometers. Commercially widespread controllers of this type are the Wii Remote™ Controller from Nintendo or the PlayStation® Move from Sony™.

Accelerometer and gyroscope are widespread on smartphones, too. In hand-held devices they are mostly used for recognizing the orientation of the device or in the entertainment applications.

2.1.3 Vision Based Devices

All the devices that recognize movement without wearing or using any kind of visible sensor belongs to this category. The most common commercial devices of this type are the Microsoft Kinect™, Leap Motion Controller and PlayStation® Eye.

The Leap Motion Controller is a sensor that detects and tracks hands, fingers and finger-like tools. The controller uses infrared cameras and LEDs to observe a reverse pyramidal space from the origin point located in the device.

The effective range is from 25 millimeters to approximately 60 centimeters. In the next chapters, we will examine this device and its library more in depth.

PlayStation® Eye and Microsoft Kinect™ are full body 3D motion capture devices. They use a combination of RGB cameras, an infrared sensor for depth and a multi-array microphone; the devices allow the recognition full body 3D motion, facial recognition and have voice recognition capabilities. The Microsoft Kinect™ can be used through a software development kit, which has been officially released by the producer and is free for non-commercial uses. This library allows to access to the raw sensor streams, to a skeletal model tracking and to some advanced audio capabilities. According to current specification has a range of from 0.8 m to 4.0 m depth. It allows to access to two streams of 640x380 images at a 30fps (frame per second), in which one is a RGB video image, and the other is an 11-bit depth image which is used to compute positional data.

2.2 Gesture Recognition

Gesture recognition is a field of computer science whose goal is to successfully interpret human gestures via mathematical algorithms. Recognizing gestures is a complex task, which may involve many aspects, such as motion modelling and analysis, pattern recognition and machine learning.

With the diffusion of Natural User Interfaces, gesture recognition is receiving more and more attention in the recent research. For recognizing gestures, it is important to know the technical specification of the sensors and the context in which the gestures are recognized.

It is important to understand which gestures are comfortable and natural and fit better in the normal usage of the interface. This depends on the type of input device and is a fundamental part of the gesture analysis.

Gesture design plays an important role: for every device it must be clear which are the strengths and the weaknesses, to understand if that device is suitable in a specific application domain. A poor design of the gesturing system may lead to a lack of success and diffusion of the device, independently from the technical features.

One of the first steps consists in recognizing the typical patterns that would be natural to use and imagine what reaction the user would expect from the system.

2.2.1 Gesture Recognition Techniques

There are many different approaches used in the gesture recognition, depending from the nature of the device and the subject to recognize. We will make an overview of the most common ones.

Template Matching Models

This approach involves to find some small subsection of an image which representing the input and checking if it matches any template image. Template matching is probably the simplest method of recognizing postures, but in some cases it is also used to recognize gestures. [6]

The raw sensor data is used as input for a decision phase. It is implemented with built-in functions which evaluate whether it matches with the template. The functions measure the similarity between templates of values and the

input; for each template there is a similarity threshold value, below which the input is rejected as not belonging to any possible classes.

The similarity function is usually the Euclidean distance between the set of sensor values and a template.

Statistical Classification

Statistical classification represents each pattern as a feature in an appropriate n-dimensional space and assumes that patterns are generated by a probabilistic system.

An example from this category is the Hidden Markov Model (HMM). A Hidden Markov Model is a Markov Chain in which the states are not directly observable. It is a collection of finite states connected by transitions, much like Bayesian Networks. Each state has a transition probability and an output probability distribution.

This model need a phase of training, to set up the correct parameters, and in the long term this models learns the most likely way that a human will perform the gesture. This method can be used in the classification of the new gestures.

The typical application of HMM is in speech recognition systems [7], but they are used even in the two-dimension spatial/temporal gestures recognition [8].

Neural Networks

Neural networks have received much attention for their successes in pattern recognition. [9]

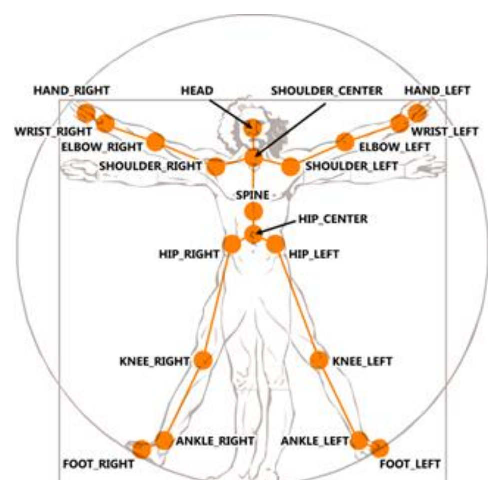
In this method, the representation is distributed over a graph as a series of interdependent weights instead of a conventional local data structure. One of the keys of the success of neural networks is that the decision process is robust even for noisy or incomplete input and it is tolerant to approximate matches, since even the same user will not reproduce the gestures exactly in the same way.

The drawbacks of a neural network are mostly in the training phase: thousands of examples are needed to train a network correctly and the training phase must be repeated whenever new gestures are added. Moreover, some care must be taken to avoid overlearning the examples and to ensure that bad examples do not prevent the convergence of the net.

The formal basis for constructing the neural network is not as well understood as the other kind of recognizers, so after the training the developer must understand, mostly by trial and error, if the net behavior models appropriately what the user wanted to represent.

Skeletal Model Based Approach

This approach is used mostly in three-dimensional modelling; working with a full 3D body model in the recognition would have a great cost in resources, in terms of processing and parameters. Instead the skeletal model represents a simplified version of the body by reducing it to segments and joints.



The analysis uses the position and the orientation of the segments and the relation between each of them with a body part.

Examples of this approach are the Microsoft Kinect™ skeletal model [10], represented in the previous picture, or the Leap Motion hand model.

Syntax or Structure Matching

The syntax or structure matching approach consists in composing complex gestures by starting from short patterns.

As in a language, the libraries offer a grammar with ground terms and operators. This approach relies on the compositionality of the gestures. The Proton library and the GestIT library are based on this methodology.

The user must define gestures by writing an expression using the syntax provided by the library. Complex gestures are built building from shorter and simpler ones, which can be recognized natively by the sensor as ground terms.

2.2.2 Touch Based Gestures

Multi-touch gestures today are common on devices such as touch-screens, tablets, smartphones, and even some notebooks. The current Start Menu in Windows 8 or 8.1 for desktop computers is an example of this tendency towards a more touch-focused interface.

Here is a list of typical touch gestures, which includes the ones classically supported by the main software development kits:

- *Tap:*
  The tap gesture consists in touching one point the screen. It can be recognized as single or multiple tap, depending on whether it happens in the same spot.

It is normally associated to the selection of a graphical item on the screen, just like a traditional click from the mouse.

- *Long press (or tap and hold):*
  This gesture consists in touching the screen in one point and holding the finger down for a certain amount of time without leaving the position. This functionality is typically associated to a pop-up menu or to auxiliary functions of the object which is targeted by the touch.
- *Pan:*
  Panning consists in touching the screen in one point and moving it in any direction without losing contact with the screen.
  This functionality is commonly associated to dragging.
- *Swipe:*
  This gesture consists in moving in a horizontal or vertical direction the fingers, starting or reaching the borders of the screen. Some devices recognize and discriminate if the gesture is made with multiple fingers, the common uses are to pop-up the application menu from the borders, to go back or forward in the navigation or to scroll the view.
- *Pinch in and out:*
  This gesture is performed by touching the screen with two fingers and moving closer or farther from each other in a linear direction. The most common use of this gesture is to zoom in and out on documents, photos or browsers. As a consequence the usual parameter of the event handlers of this gesture is a scale factor.
- *Rotate:*
  This gesture consists in rotating two touches. This gesture triggers when the user moves two fingers in opposite circular directions on the screen, and it is commonly associated to the rotation of the current view of the screen, or to a rotation of the elements which are displayed (like pictures or the current selection).

2.2.3 Vision Based Gestures

When working with vision-based devices, the first thing we lose respect to a touch device is the possibility to be certain of the event triggering. We will consider high-level gestures in the Leap Motion Controller and the Microsoft Kinect™, because of the availability of a free and official standard development kit.

The Leap Motion Controller recognizes some typical patterns in the hand movements. The current version of the development library allows you to access the raw data, but it also has the possibility to recognize four high level gestures, related to the finger movements:

- Circle gesture;
- Swipe gesture;
- Screen Tap gesture;
- Key Tap.

Since we will use the Leap Motion Controller for the thesis, in the section 3.3 we will describe in depth the access to the raw data and the Leap Motion hand model.

The Microsoft Kinect™ standard development kit toolkit officially offers two high-level gestures for recognizing movement with the hands:

- *Grab:*
  The gesture consists on moving a hand towards the device and closing it as you were grabbing an item.
- *Push:*
  It consists of moving a hand towards the camera from a firm position to indicate a selection

The new Kinect™ 2.0, available the end of November, has increased the hardware capabilities, improved the skeletal recognition and facial model,

improved the tracking robustness, but they did not introduce any new gestures, as stated in the producer's announcement [11].

For the other similar devices, like PlayStation® Eye and Wii Remote™ Controller the SDK are not publicly available.

2.3 Available Frameworks

In this section, we will describe some of the frameworks and libraries available on the market to describe or compose gestures in our domain or to analyze time series.

When a programmer has to design and implement a new customized gesture from scratch, the main problem is the manual handling of all the generated events at low level.

The main issues when programming new gestures with this approach are mostly two:

- The recognition code is probably split in various location of the source code
- Different gestures may have a common part

Imagining future extensions, adding a new gesture will cause a rewriting of parts of code in the handlers, so the complexity will grow and will cause conflicts, making the future maintainability problematic.

In the next sub-chapter, we will see a brief description of Proton++, a software developed by the Berkeley University of California that uses regular expression and nondeterministic finite state automata (NFA) to express

gestures. Then we will describe Deedle, a library working with structured data frames, ordered and unordered data to make some offline statistical analysis.

In a following chapter, under the tools section of this thesis, we will analyse GestIT, a library using a Petri Net model to represent the system, developed in the Università di Pisa.

2.3.1 Proton/Proton++

Proton++ [12] is a C++ framework developed in Berkeley University of California, which allows defining multi-touch gestures as regular expressions, applied to a stream of touch events, adding the possibility to include customizable attributes linked with the gesture.

Proton++ has been developed as an extension of Proton [13], the older framework developed from the same university; the main addition of this extension is the customizable attributes.

As the two framework are bound together, we will analyze Proton first, and then we will see the improvements added in Proton++.

The latest version of Proton is released under the BSD license, and available at [14].

Proton

Proton is a declarative multi-touch framework that allows developers to specify gestures as regular expressions of touch event symbols. Proton converts all the touch events, generated by a multi-touch hardware, in a stream of touch event symbols, touch IDs and custom touch attributes.

The touch symbols are three, representing an encoding of the touch action (down, move and up). When defining a new gesture, the developers have to

define the multi-touch gestures as a regular expression of these symbols. Proton will match this regular expression with the event stream.

The approach to gesture development can be synthesized as follows: the programmer writes the gesture as a regular expression, and adds the callback associated to the gesture completion. The framework builds a finite-state-automata that will manage all the event system.

Then the framework, with some static analysis on the regular expression, will point out all the conflicts caused by similar gestures, and the programmer will have to write the disambiguation logic behind, to solve the conflicts.

Following this procedure will therefore avoid the manual management of the conflicts and consequentially solves the necessity to track the state machine and the changes in the handlers.

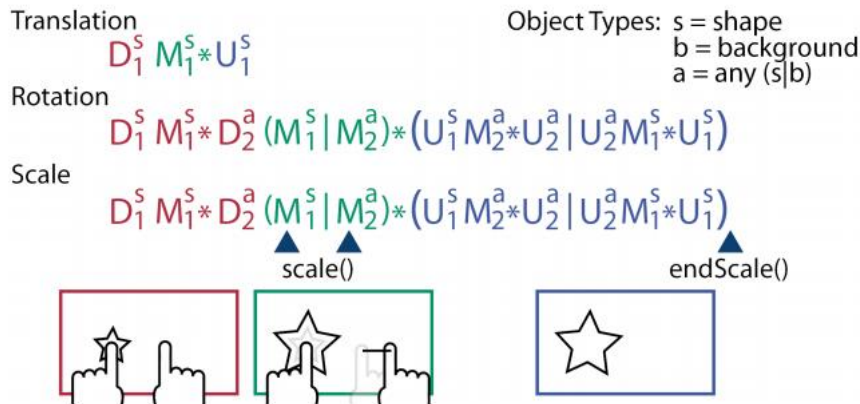
In Proton every event is represented as a symbol:

$$E_{id}^{type}$$

Where:

- $E \in \{D, M, U\}$ (representing respectively touch-down, touch-move, touch-up);
- $type$ represent the type of object influenced by the touch;
- id is the ID of the event, regroups events that belongs to the same touch.

In this image, for example, are reported the description of classical gestures, such as translation, rotation and scaling:



The rotation gesture starts with a first touch on the shape to select it, then a second touch on the shape or canvas (defined as any in the type); then both touches will perform the move event repeatedly. In the end, the touches can be released in any order.

The pseudo code to implement this gesture is the following:

$$gesture: \ D_1^S M_1^S * D_2^A (M_1^S | M_2^A) * (U_1^S M_2^A * U_2^A \ | \ U_2^S M_1^S * U_1^S \ )$$

$$gesture: addTrigger(rotate(\ ); 4) \ // \ M_1^S$$

$$gesture: addTrigger(rotate(\ ); 5) \ // \ M_2^A$$

$$// \ rotate() \ is \ the \ callabck \ that \ computes \ rotation$$

$$gesture: finalTrigger(endRotate(\ ))$$

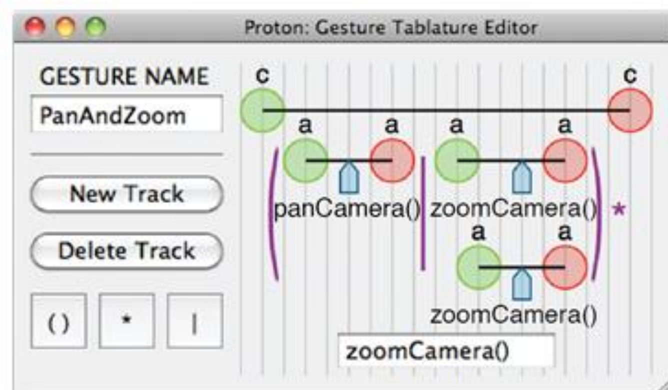
$$// \ endRotate() \ is \ the \ callback \ that \ perform \ rotation \ cleanup$$

$$gestureMatcher: add(gesture))$$

Another instrument the framework offers is the gesture tablature, which allows to define and organize multi-touch events using a graphical tool, with an intuitive definition. The notation is similar to guitar tablature notations.

The developer adds a horizontal line for every touch tracked, called touch track. Every track has a green element representing the starting of a touch event (touch-down) and a red element representing the end (touch-up). A black line between the green and the red element represents an arbitrary number of touch move events.

The temporal order of the events is given by their position in the tablature, with the time flowing from left to right. The other element in the tablature, represented as a blue pin linked to events are the triggers, representing the callback methods.



Proton++

Proton++ is the evolution of Proton, which adds the possibility to integrate the gesture definitions with customizable attributes, increasing the expressivity of the regular expressions without losing the advantage of static analysis.

Typical attributes that can be added are: the direction of the touch, computed from the last two positions, binding the movement only to specific axis directions, like northbound only, or adding a pinch attribute to reveal if the two touches are moving toward each other. Moreover, Proton++ adds the possibility to code a temporization of the touch symbols.

To obtain this custom attributes in Proton++ the user has to write an attribute generator, that has the duty to map the hardware discrete values to the attribute information.

The touch event has nearly the same grammar than before, but the type has now become a set of attribute values that have to be verified.

More formally:

$$E_{id}^{attributes}$$

Where:

- $E \in \{D, M, U\}$ (representing respectively touch-down, touch-move, touch-up);
- $attributes$ represents a sequence of attributes values, split by a semicolon (A_1: A_2: A_3 ...);
- id is the ID of the event, regroups events that belongs to the same touch.

For example, if we have to represent the move event of the item “s” in direction west is $M_1^{s:W}$

In Proton++, there is also the possibility to instantiate a time constraint. The syntax involves writing a timespan over the element we want, as a superscript over the event close in brackets. The time unit is fixed, and is equal to one thirtieth of second.

Using the previous example, if we want to indicate the previous move event in west direction, giving a timespan of half second, we need to write $(M_1^{s:W})^{1-15}$.

2.3.2 Deedle

Deedle is an F# library for manipulating data and time series and for scientific programming. It supports working with structured data frames, ordered and unordered data, and time series.

The main application of Deedle is working for exploratory programming using the interactive console, but it can be used in compiled code.

The Blue Mountain Capital Management LCC, a company working mostly on private capital investment and finance, develops the library, and releases freely under BSD license [15].

The background of the company makes the library more oriented towards working on offline analysis of big series of data about stock exchange market with the interactive console, but it also offers a range of operation that allow the users to develop analysis on time series.

The data series are organized with a relational approach: it works on tuples, seeing data as rows and attributes as columns, and, similarly to databases, offers queries for data manipulations.

Time is an essential type for performing the data manipulation and they offer some operation based on it, as:

- Sorting based on time;
- Zip time series entries together with some aggregating operators;
- Extracting or grouping data by a timespan;
- Lookup to find the nearest available data to a specified date;
- Sliding time windows to represent and select groups of data;
- Time sampling.

The main statistical operators are for calculating cardinality, sum maximum and minimum, mean, median and standard deviation.

All the operations can be applied to the whole dataset or to certain groups of data, defined by applying a levelling operator, that makes a hierarchical indexing, using constraints for splitting and aggregating values.

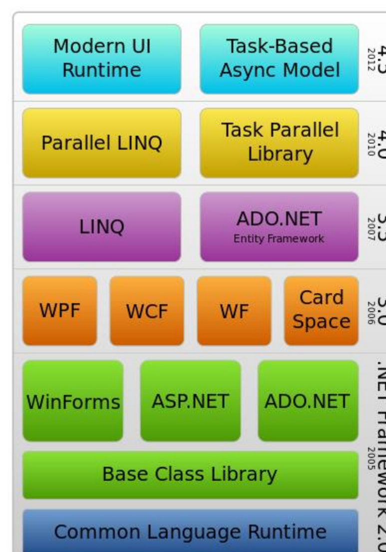
Further documentation on the library is available at [16].

3 Tools

In this chapter we will see which are the tools used in this thesis. In the first part, we will see the .Net Framework and the main features of the F# language in particular. Then we will examine the Leap Motion Controller and in the end the GestIT library and its typical usage.

3.1 The .Net Framework

The .Net Framework [17] has been developed by Microsoft in the late 90s, with the first official release in 2002, and the latest release is version 4.5.1. It is a technology that runs primarily on Microsoft Windows platforms and it supports building and running applications, with different programming languages.



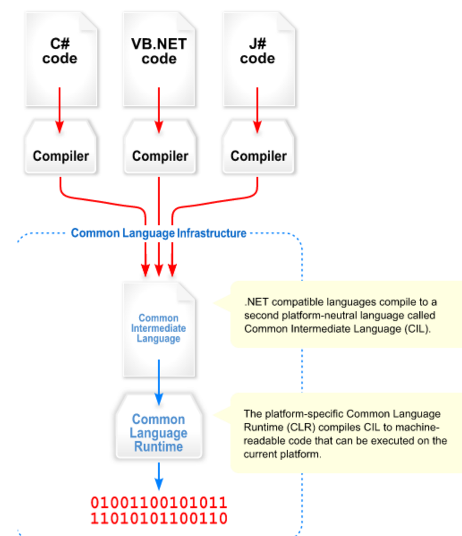
There are two main integrated development environment (IDE) used for developing with .Net Framework: Visual Studio [18], developed by Microsoft for Windows only, and MonoDevelop [19] from the Mono project, an open source project for making the .Net Framework cross-platform, available

both on Windows and on Unix like platforms like Linux and MacOS X. In this thesis, we used Visual Studio 2012.

The .Net Framework supports various types of language, the most famous are C++, C#, F#, Visual Basic .Net, ASP.Net and Jscript.Net.

The foundation of the .Net Framework is the Common Language Infrastructure (CLI), a specification originally proposed by Microsoft, now open and standardized by ISO [20] and ECMA [21].

The Common Language Runtime (CLR), an implementation of the CLI, is the core of the Microsoft .Net Framework.



The CLR is an application that provides a virtual machine that runs the Common Intermediate Language (CIL) and guarantees some important services, such as security, code execution, memory management, thread management, garbage collection and exception handling. This layer provides an abstraction layer over the operating system, and every .Net compliant language has a compiler from the language to the CIL.

The second important part of the .Net Framework is the Base Class Library: this library is the base of the functionalities available in all the languages supported by the framework. It is an object-oriented collection of reusable types, which implements the basic functionalities for developing applications. The class library includes all the common functionalities: from basic input-output operations, to graphic rendering, to file managing, to database interaction and so on.

Another important aspect of the .Net Framework is the Common Language Specification and the Common Type System.

The Common Language Specification is a set of rules which the languages targeting the Common Language Infrastructures must respect to interoperate with other CLS-compliant languages.

The Common Type System define how to declare, use and manage types in the Common Language Runtime. This enables cross-language integration, type safety and high-performance execution.

Following the two guidelines, while programming the application, guarantees language independence and the possibility to build a re-usable component. This feature enables the possibility to write components in different languages, hopefully the most suited and efficient for the task being coded, and make them interact easily.

In our case, this possibility has been relevant, as the Leap Motion Controller Library is written in C++; we developed the framework using F# and the graphical user interface in C#, with the help of the visual editor.

3.1.1 F#

F# [22] [23] is a programming language that can be considered a multi-paradigm language. It provides support for functional programming constructs in addition to the traditional object-oriented and imperative programming paradigms.

F# is strongly typed and it uses type inference to statically type any value in the program. It implements functional programming with eager evaluation. It has discriminated union types, lambda expressions and it supports partial application of functions and closures. Functions are a first-class values and they can be used in higher-order functions.

F# also supports classic imperative and object-oriented programming; this includes array and object types, inheritance, static and dynamic method dispatch, for and while loops and most of the usual OO patterns.

The main advantages of using F# are the possibility to express most concepts in a very concise way thanks to the functional language, and to be integrated with the features and the advantages offered by the .Net Framework and by the Common Language Infrastructure.

We choose this language for developing our thesis for various reasons:

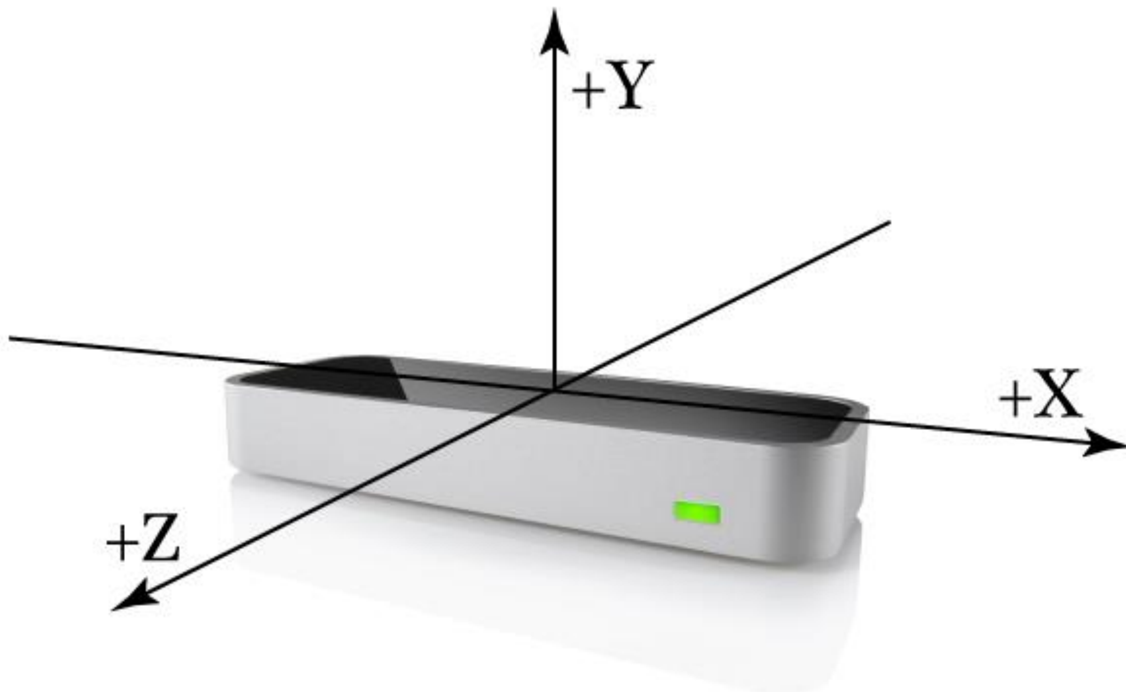
- We want to implement a library that gives the possibility to define time analysis on data series in a declarative way, by expressing properties on the input; functional languages are better suited for this purpose;
- With functional languages it is easy to work on collections of uniformly typed data like time data series;
- The functional paradigm provides a convenient abstraction due to his conciseness and the possibility to express functions as first-class values; the type constraints enforced by the interface ensure correctness even in general-purpose usage;
- It provides the best integration with the GestIT library, which is written in F# and was used as starting point to develop the thesis.

3.2 Leap Motion Controller

The Leap Motion controller [24] is a new infrared device that tracks the movement of fingers, hands and finger-like tools. It is a small USB peripheral that uses an infrared system with LEDs and cameras to detect with high precision the positions of these objects.

3.2.1 Technical characteristics

Using three infrared LEDs and two CCD cameras, the sensor observes a hemispherical area above the sensor with a radius of about 60 centimeters. The minimal distance from the sensor which can be measured is 1 centimeter. The producer states that the measurement error is about 0.01 millimeters.



The Leap employs a right-handed Cartesian coordinate system. The center is the center of the leap device. The x-axis and z-axis lie in the horizontal plane, with the x-axis that is parallel to the long edge of the device. The y-axis is vertical with only positive values increasing upwards.

3.2.2 The Hand Model

The Leap sensor tracks hands, fingers and tools in its field of view and updates their set of data, called frame. Each frame consists of lists of basic tracking data, such as hands, fingers, and tools, as well as recognized gestures and factors describing the overall motion in the scene.

When the sensor detects a hand a finger or a tool, it assigns a unique ID to it. This ID remains the same as long as the entity is visible in the device field of view. If the tracking of an entity is lost and then recovered, the sensor may assign a new ID, but has a system to track back if the entity is the same.

In the next sub chapters we will describe the API and the main classes [25] to use for interacting with the device.

Controller

The controller class is the main class for using the Leap Motion Controller.

An instance of the controller class has to be created for accessing the frames, for tracking data, and gathering all the configuration information. Calling the *frame()* method allows the user to poll the controller and get last frame data. The controller stores up to 60 frames.

Every controller has associated a *Config* class, that contains configuration parameters for the gesture recognizer. Other useful information that can be obtained from this class are: a list of active devices if there are more Leap Motion Controllers plugged in the system, a setter to enable high-level gestures recognition, and the possibility to add a listener to the controller, to receive the notification of new frames, following the typical observer pattern.

Frame

The device sends a stream of frames. The frame class is the main data structure and it contains the list of all the item and features recognized. It represents the entire environment recorded from the leap sensor in a selected instant.

This is the list of the main tracking data supplied by the library:

- *HandList*: all the hands detected in the frame;
- *ToolList*: all the tools detected;

- *PointableList*: the list of pointable objects (both tools and fingers) recognized;
- *FingerList*: the list of fingers detected in the current frame;
- *GestureList*: the list of gestures recognized or continuing in the current frame

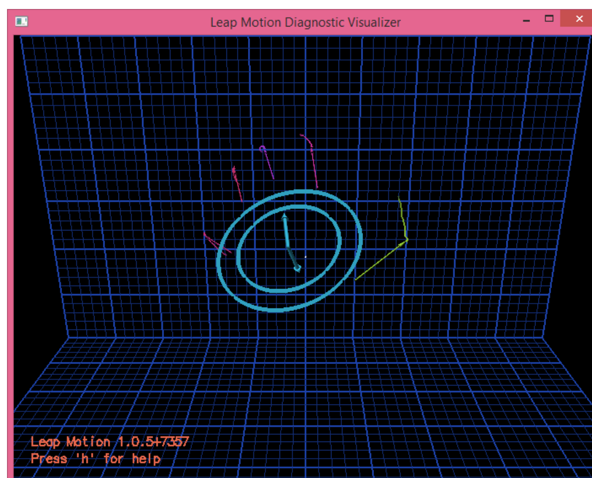
Other useful values regarding this class are the item id, the possibility to access to current frame-rate and of course the timestamp.

Hands

The hand model provides information about position, characteristics and movement of a detected hand and the list of fingers and tools associated.

The API provides as much information as possible. However, it is not possible to have all the attributes in every frame. For example when a hand is closed in a fist, the fingers are not visible to the Leap sensor so the fingers list will be empty. This is an important remark, as the programmer has to handle this case in their code.

The Leap does not discriminate between the left and right hand, and there is no explicit recognition of the thumbs respect to other fingers. The sensor can recognize multiple hands but the recommendation for an optimal tracking quality is to avoid the use of more than two hands together.



The Hand object provides several attributes reporting physical characteristics of detected hands. The most relevant are:

- *Palm Position:* indicates the center of the palm, measured in millimeters;
- *Palm Velocity:* The speed of the palm in millimeters per second;
- *Palm Normal:* A vector that is perpendicular to the plane formed by the palm of the hand, pointing downwards;
- *Direction:* A vector pointing from the center towards the fingers;
- *Sphere center:* The center of a sphere that fit to the curvature of the hand;
- *Sphere Radius:* The radius of the sphere that fit to the curvature of the hand. This radius changes with the shape of the hand.

The direction and the palm normal are unit direction vectors that describe the orientation of the hand respect to the Leap coordinate system, while the sphere center and the sphere radius describe the curvature of the hand.

The object also provides several attributes that represent the motion of a detected hand between frames. The Leap sensor analyzes the motion of the hand and its associated fingers and tools, and it reports translation, rotation, and scale factors. Translation represents moving the hand around the Leap field. The rotation is produced by turning, twisting, or tilting the hand. Moving the fingers or tools toward or away from each other will end in the scaling attribute.

These values are derived by comparing the characteristics of the hand in the current frame with those specified in the earlier frame.

The most relevant attributes are:

- *Rotation Axis*: A direction vector expressing the axis of rotation;
- *Rotation Angle*: The angle of rotation clockwise around the rotation axis (using the right-hand rule);
- *Rotation Matrix*: A transform matrix expressing the rotation;
- *Scale Factor*: A factor expressing expansion or contraction;
- *Translation*: A vector expressing the linear movement.

Fingers and Tools

The accessibility to fingers and tools is possible by accessing one of this three lists:

- *Pointables*: Both fingers and tools as Pointable objects;
- *Fingers*: Just the fingers;
- *Tools*: Just the tools.

You can also find an individual finger or tool using an ID value obtained in previous frames. The Pointable list is convenient if the cases where there is no need to discriminate between fingers or tools.

The Leap classifies finger-like objects according to shape, expecting that a tool is longer, thinner, and straighter than a finger.

The physical characteristics of pointable objects include:

- *Length*: the length of the visible portion of the object, from the first point out of the hand to the tip;
- *Width*: the average width of the visible portion of the object;
- *Direction*: a unit direction vector pointing in the same direction as the object, from base to tip;
- *Tip Position*: the position of the tip in millimeters from the Leap origin;
- *Tip Velocity*: the speed of the tip in millimeters per second.

Every Pointable is classified as a Finger or a Tool. To discriminate between a tool and a finger is possible to use the *IsTool()* property.

Gestures

The Gesture class represents the recognized high-level pattern movements by the driver.

The Leap Motion Controller watches the activity, in his field of view, to recognize the movement patterns which represent typical gestures.

When the Leap Motion software detects a gesture, it assigns an ID to it and adds a Gesture object to the frame gesture list. For continuous gestures that last many frames, the Leap motion software updates the gesture by adding a Gesture object with the same ID and updated information.

Peculiar properties of this class are the time duration, available in seconds or microseconds, the frame, hands and pointable items that are associated to the gesture. The properties that define the state of the gestures are the gesture state and the gestures type.

The gesture type is an enumeration which is used to discriminate the high-level gesture; four gestures can be recognized:

- *Circle*: a circular movement by a finger;
- *Swipe*: a straight line movement by the hand with fingers extended;
- *Screen Tap*: a forward tapping movement by a finger;
- *Key Tap*: a downward tapping movement by a finger.

The gesture state is another enumeration of three possible values:

- *Start*: the software has recognized enough to consider that the gesture is starting;
- *Update*: the gesture is still in progress, with updated parameters;

- *Stop*: the gestures has completed or stopped.

The continuous gestures have all the three states, associated to the same gesture id. In case of a discrete gesture, the software will recognize a single gesture item, which always appears with a stop state.

The circle and swipe gestures are continuous, while both the screen tap and key tap are discrete gestures. Every high-level gesture is a subclass of the gesture class, with the additional parameters describing the specific gesture.

It is important to remark that the built-in gestures are not active by default. In order to use them, the user must activate the recognition of each of them in the controller class, calling the *EnableGesture(type, boolean)* method with type which indicates the appropriate kind of gesture and a Boolean to choose whether to enable or disable its recognition.

Listening Events to Leap Motion Controller

The listener class is the main interface to receive events from the device.

It follows the classical Observer pattern [26]; a user has to subclass listener class, implement the methods and subscribe the listener to the controller, calling the *AddListener(Listener)* method.

The controller object will call these listener methods as callbacks when an event occurs, passing a reference to itself:

- *OnConnect*: this method is called when the Controller object connects to the Leap Motion software, or when the Listener object is added to a Controller that is already connected;
- *OnDisconnect*: called when the Controller object disconnects from the Leap Motion software;
- *OnExit*: called when this Listener object is removed from the Controller or the Controller instance is destroyed;

- *OnFocusGained*: called when the application becomes the foreground application;
- *OnFocusLost*: Called when the application loses the foreground focus;
- *OnFrame*: Called when a new frame of hand and finger tracking data is available;
- *OnInit*: Called once, when this Listener object is newly added to a Controller.

3.2.3 Strength and Weakness

To sum up the main strength and weakness of the Leap Motion controller, we can say that its strength are:

- *Precision*: The device has a resolution of about 0.01 millimeters, which should result in position samples with a very high precision;
- *Frame Rate*: Its frame rate is high and stable; it can exceed 100 frames per seconds in general use and it stays above 50 fps even under stress;
- *Multiple hands*: It recognizes two hands with great stability; the case of having more than two hands over the same controller is unpractical because of insufficient physical space.

The weakness or issues are:

- *Distinguishing two close fingers*: In most of the cases they will be shown as a single finger, so sometimes the programmer can have problems with disappearing fingers if they get too close;
- *Aliasing:* the sensor sometimes assigns a new id the same element, if it loses track of it for a fraction of second as it believes that there is a new one in the same area. This may lead to strange behaviours if the application code is excessively bound to IDs. Recent versions of the driver have reduced the impact of this problem;

- *Overlapping*: Due to his nature, the sensor has only a point of view, so in case of multiple hands, the possibility of overlapping the hands or one hand and a pointable item, can cause problems.

3.3 GestIT

GestIT [27] is a library for managing gestures in a compositional and declarative way.

The library is developed in the Università di Pisa, and it is currently available in JavaScript and F# for the .Net environment. It is open source and freely downloadable on GitHub at [28].

GestIT is abstract with respect the platform: it can be used for describing gestures recognized by very different kinds of devices, such as multi-touch interactions with touch screens or body gestures detected using Kinect or Leap sensor.

The library permits to define high-level gestures by decomposing them in small parts and to assign handlers to the composing parts. The user defines an expression, composed by simple gestures and operators and the library transforms it in an object that performs the desired behaviour.

The meta-model behind GestIT is the Petri Net, more precisely the non–autonomous coloured Petri Net.

In the next sub-chapters, we will see a formal definition of Petri Nets and their variants, the basic concepts used in the GestIT library and how it works.

3.3.1 Petri Nets

A Petri Net [29] (also known as a place/transition net or P/T net) is a mathematical representation for a distributed system. They took their name from Adam Petri, who defined formally for the first time in his PhD Degree [30] in 1962.

A Petri Net (PN) is composed by a graph and a signature: the graph is a bipartite direct graph, with 2 kinds of nodes called places and transitions, and a marking that specifies the number of tokens in each place.

Formally, a Petri Net is defined as a quintuple:

$$PN = \{P, T, F, W, M_0\}$$

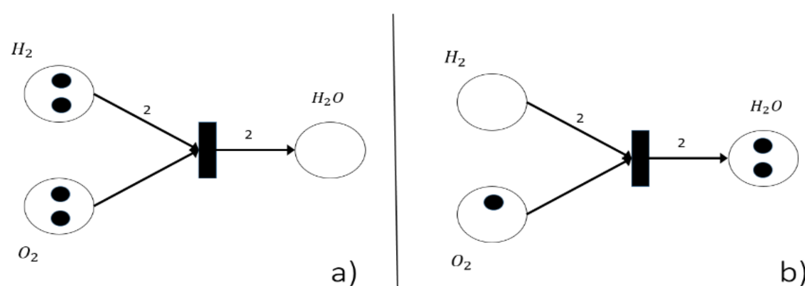
- *P* is a finite set of states called places;
- *T* is a finite set of states called transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ a set of arcs;
- $W: F \rightarrow N^{+}$ a weight function on the arcs;
- $M_o: P \rightarrow N$ the initial marking on the places;
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

There are various equivalent definitions, other authors describe Petri Net as $PN = \{N, M_0\}$, with *N*, defined as the first four elements in the quintuple, to underline the distinction from the net, that represents the structure of the system and the initial marking M_0 that is the part that represents the dynamicity.

So the behaviour of systems can be described in terms of states and their changes. To simulate the dynamicity of the system the marking changes according to the following transition rule, called firing rule:

- A transition is said to be enabled if each input place *p* of t is marked with at least $W(p, t)$ tokens, where *W* is the weight of the arc between *p* and *t*;
- An enabled transition may or may not fire (depending on whether or not the event take place);
- A firing of an enabled transition t removes $W(p, t)$ tokens from each of the input places *p* of *t*, and adds $W(t, q)$ tokens to each output place *q* from *t*, where $W(t, q)$ represents the weight of the arc from t to *q*.

As a graphical convention, the places are represented with circles with the tokens inside, and the transition are represented as tight rectangles. Every arc has a weight, if not stated the weight is one.



Here an example of a Petri Net before *(a)* and after *(b)* a transition, representing the well-known chemical reaction $2H_2 + 2O_2 = 2H_2O + \ O_2$

Petri Nets are often used to represent easily concurrent system or workflows [31], and are used in different contests with many variants from the original definition. For example, the earlier definition is the definition of autonomous Petri net.

The second rule of the firing rule states that an enabled transition may or may not fire. When describing a concurrent system, with this rule, we consider the system in a qualitative way, without giving any concept of timing or synchronization.

This concept is satisfactory when we are looking to a workflow system, where we are mostly interested in what happens after a transition, but does not fit

very well in our case. We want a reactive system and the non-autonomous Petri Nets are the answer, in which transition are triggered by external events, independent from the system.

Finally, colored Petri Nets, preserve all the properties of the Petri Nets and at the same time extends the formalism, adding the possibility to distinguish tokens. With the color, we have the advantage to add information on the tokens and make compact representations.

Although this is convenient, in our case it does not actually make the model more complex, because colored Petri Nets with a finite number of colors can be converted in ordinary Petri Nets [32].

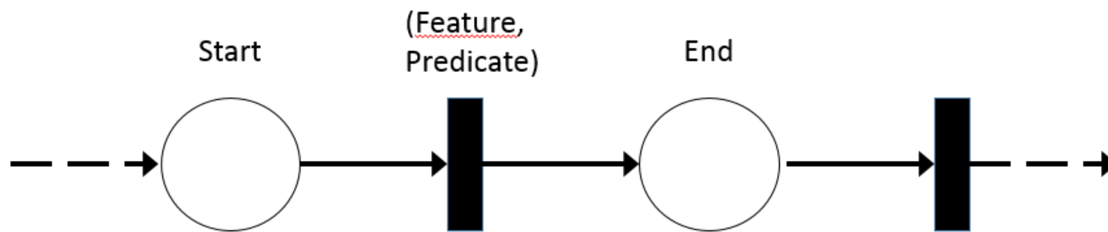
3.3.2 Gesture Description Model

In this section we will define the compositional gesture description model at the base of GestIT [33]. For composing a gesture, we need to understand the three main concepts of the library: ground terms, which are the basic building blocks, composition operators, useful to build complex gestures, and handlers.

Ground Terms

A ground term is the working unit of the system. A ground term has no temporal extension and it is composed by two parts: the *feature* and the *predicate*.

In this figure, we can see the equivalence of a ground term in a Petri Net. The dotted arcs are external to the ground term.



The feature represents the event tracked by the developers from the sensor. All the sensor events involved in the gesture recognition will be assigned as a feature. For instance, a mouse click, a touch event on a multi-touch device or a finger move fall into this category.

Optionally a predicate can be associated to the feature in a ground term. The predicate in an additional function that has to be verified, in order to allow the firing of the transition, after receiving the sensor notification.

As an general guideline a predicate should only verify additional properties just like a filter, and should never modify the state of the system with side effects, thus to avoid losing the compositionality of the system. As we will see in a moment, the handlers can perform side-effects without breaking the behaviour of the system.

The state of a gesture at a given time is represented by the current value of each feature. The state of a gesture recognition support over time can be represented by a sequence of states, considering a discrete time sampling.

We can define a feature f as a n-dimensional vector with data representing the state obtained from the device.

A gesture recognition support G_S can be seen as set of features, and a gesture recognition support state can be seen as the value of this features at a given time t_i. In the end the evolution of the system can be represented as a sequence of states.

$f \in \mathbb{R}^n$

$G_s = [f_1, f_2, .., f_n]$ $\quad G_s \in \mathbb{R}^k, f_i \in \mathbb{R}^{n_i}, \ \sum_{i=1}^{m} n_j = k$

$G_{S_i} = [f_1(t_i), f_2(t_i), .., f_n(t_i)]$ $\quad (t_i) \in \mathbb{R}$

$S = G_{S_i}, G_S, ..., G_{S_n}$ $\quad n \in \mathbb{N}$

The gesture building block in the end notifies the change of a feature value between time t_i and time t_{i+1}. This change can be optionally be associated to a condition, which in our case is a *predicate*.

Composition Operators

The composition operators allow the connection of ground terms or composed gestures in order to build up expressions, representing more complex gestures.

The supported operators are the following, and in brackets the shortcut symbol in the F# syntax:

- *Iterative (!*)*: it is a unary operator that expresses the repetition of a gesture recognition an unlimited number of times;
- *Sequence (|>>)*: a binary operator that represents that the two connected sub-gestures have to be performed in sequence, from left to right;
- *Parallel (|=|)*: this operator imply represents the requirement that both the two connected sub-gestures are recognized;
- *Choice (|^|)*: this operator express that the recognition of any of the two sub-gestures also leads to the recognition of the whole gesture;

- *Disabling (|>)*: this operator is used to let a sub-gesture to stop the recognition of another one, typically used to stop the iteration loops.

Handlers

A handler is a function that can be assigned to a ground term or an expression by using the shortcut symbol " | → " .

As we expect in a reactive system, it represents the routine that the library calls after an event (in our case a ground term) triggers and the Petri Net advances. The developer should use the handlers to perform most of the functionality and in particular any side-effect that modifies the state of his system.

3.3.3 The Library

In this section, we will see a short description of the features of the library with simple examples. We will see the Fusion Sensor, which is the interface between GestIT and the sources, the syntactical representation of the gesture with the operators.

The Fusion Sensor

The Fusion Sensor is the interface between GestIT networks and events sources. It makes possible to bind events to identifiers known as features. This can be done on multiple different event sources, so that GestIT networks can access input coming from different devices with a unified interfaces.

Additionally, the Fusion Sensor makes it possible to have a dynamic set of features that changes over time. This is important when implementing gestures which track objects that appear and disappear and allows easier use of timeout and feedback events.

This class uses a dictionary, and it stores all the linked events that the system has to be aware. The user has to create an instance of the Fusion Sensor type and add, using the Listen method, a series of entries to the dictionary.

The key can be any kind of item or value, as long the type supports comparison by equality. A typical appropriate usage is to make an enumeration of the current event possibilities, to make the code more readable.

Gesture Expression

The gesture expression type (*GestureExpr*) is the abstract type that represents the gesture, which will be transformed into a Gesture Net, on which the token transition will take place. Gesture expression is an abstract type, whose specializations are the ground term type or in the expression types, one for each composition operator.

The abstract type has the common methods for all the subclasses, that allow to manage the Petri net, leaving two abstract method that are peculiar for the type of item:

- *Children*: contains a sequence of gesture expressions, representing the sub-terms in the expression;
- *ToNet*: is the method that given a sensor, builds the Gesture Net.

The subclasses representing the possible expression are:

- *Ground Term;*
- *Iter;*
- *Parallel;*
- *Choice;*
- *Sequence.*

Gesture Net

The *GestureNet* type is the resulting type when we call the method *toNet* on a gesture expression. It represents the Petri net, and simulates the Petri net behaviour. It is an abstract type, which is specialized in two sub-types, *GroundTermNet* and *OperatorNet*.

As expected this two types represents the two possible cases, depending from the syntax tree of the grammar. The ground term represents the leaf of the syntax tree, and the operators are the internal nodes, that compositionally build the tree.

The *frontier* property states the places of the Petri nets, which currently have tokens active, or are reachable without any token consuming transition. The *completed* method is used to signal the completing of a net, while the *RemoveTokens, AddTokens and ClearTokens* are the methods that move effectively the token in the net.

In the building phase all this method work recursively, starting from the highest level of the net and forwarding the tokens in the subnets.

Shortcut Operators

To reduce the verbosity of the gesture expression, which would involve manually constructing the whole syntax tree, some operators have been introduced in the language.

Every operator has an infix alias, which helps to make the expression more readable and intuitive.

Here the list of the infix aliases and the corresponding operators:

- | ≫ for sequence operator;
- |^| for choice operator;
- | = for parallel operator;

- $!*$ for iterative operator;
- $|\rightarrow$ for assigning a handler to a ground term.

3.3.4 Using the Library

When he is using the library, the programmer does not need to worry about building a Petri Net manually and can focus on the expression which represents the device behaviour.

The typical step that a user must follows are:

- Create an instance of the fusion sensor;
- Register the list of features he needs to track;
- Define expression that fits with the behaviour he wants to implement;
- Write the handlers that his expression should trigger.

After these steps, the user call the method *ToGestureNet(s)* of the object which represents the expression, using the fusion sensor that he wants to link to the net as the *s* parameter.

3.3.5 An Example

This short piece of F# code tracks the clicks of a mouse:

As first step, we start defining an enumeration type that represents the mouse states we want to consider, that will be the features of our ground terms.

The second step is to create the fusion sensor and to use it to associate features and native.

We will pass the enumeration as features, and the classical built-in *MouseEventArgs*, the descriptor of a mouse event in the .Net Framework. Afterwards we will bind the mouse events with our mouse feature type.

```
type MouseFeatureTypes =
        | MouseDown = 0
        | MouseUp = 1
        | MouseMove = 2

let sensor = new FusionSensor<MouseFeatureTypes,MouseEventArgs>()

sensor.Listen(MouseFeatureTypes.MouseMove, app.MouseMove)
sensor.Listen(MouseFeatureTypes.MouseUp, app.MouseUp)
sensor.Listen(MouseFeatureTypes.MouseDown, app.MouseDown)
```

Then we will build the ground terms of our net. Before doing it we build two predicates that we will need and use: one to understand which button is clicked, the other one to check if there is no pressed button.

```
let pushbutton (t:MouseButtons) (e:MouseEventArgs) =
           if (e.Button.Equals(t))
                        then
                            true
                        else
                            false

let mouseup (e:MouseEventArgs) =
           if (e.Button.Equals(MouseButtons.None))
                    then
                        true
                    else
                        false
```

Now we can create the ground terms, which will represent the three different clicks we want to track:

```
let LeftButton   = new GroundTerm<_,_>
                                   (MouseFeatureTypes.MouseDown,
                                   (pushbutton MouseButtons.Left))

let MiddleButton = new GroundTerm<_,_>
                                   (MouseFeatureTypes.MouseDown,
                                   (pushbutton MouseButtons.Middle))

let RightButton  = new GroundTerm<_,_>
                                   (MouseFeatureTypes.MouseDown,
                                   (pushbutton MouseButtons.Right))

let MouseUp  = new GroundTerm<_,_>(MouseFeatureTypes.MouseUp,
mouseup)
```

To finish we need our handlers, which will be triggered when the net moves from the states.

```
let LeftClick_h (sender, f:MouseFeatureTypes, e:MouseEventArgs) =
                             System.Console.WriteLine("Left Click")

let MiddleClick_h (sender, f:MouseFeatureTypes, e:MouseEventArgs) =
                             System.Console.WriteLine("Middle Click")

let RightClick_h (sender, f:MouseFeatureTypes, e:MouseEventArgs) =
                             System.Console.WriteLine("Right Click")

let MouseUp_h (sender, f:MouseFeatureTypes, e:MouseEventArgs) =
                             System.Console.WriteLine("Mouse Up")
```

In the end we need the gesture expression, to build the Petri net and make the application run:

```
let events =  !*(
      (LeftButton |-> LeftClick_h) |^| (MiddleButton |->
MiddleClick_h) |^| ( RightButton |-> RightClick_h ) |^| (MouseUp |->
MouseUp_h)
)

events.ToGestureNet(sensor)|>ignore
Application.Run(app)
```

With this approach, we can reuse this code and use the already defined predicates to track something else. If we wanted to trigger an event when the user clicks left, middle and right button in this order, ignoring the rest of clicks, we can use the same code, build a new handler and change only the expression:

```
let events =
!*(( LeftButton |>>  MiddleButton |>>  RightButton) |-> ThreeClick_h)
```

4 Design and Implementation

The original idea of the thesis work was to develop the part of temporal analysis as an extension to GestIT by adding a new syntax for expressing the temporal properties of gestures and a system model history.

At that time, the Fusion Sensor was not yet part of GestIT and sensors were modeled differently within the library. The sensor would interface indirectly with native events and all the events would come from a unique source; when developing an application using GestIT the adapter between a sensor and the underlying device library was made by inheriting from a base sensor class.

Instances of these class were meant to receive events from devices and forward them to a GestIT sensor, which would fire the associated actions inside the Petri nets.

```
type MouseSensor () =
   inherit UserControl()
   let mutable down = false
   let sensorEvent = new Event<SensorEventArgs
                                <MouseFeatureTypes,MouseEventArgs>>()
   let debug = false

   override x.OnMouseDown(e) =
     if debug then
         printfn "MOUSE DOWN"
         sensorEvent.Trigger(new SensorEventArgs<_,_>
                               (MouseFeatureTypes.MouseDown, e))
   down <- true
...
   interface GestIT.ISensor<MouseFeatureTypes,MouseEventArgs> with
            [<CLIEvent>]
            member x.SensorEvents = sensorEvent.Publish
```

In this code snippet, we can see a part of implementation to obtain the mouse behaviour.

This was inconvenient because it required overriding every method and rewriting part of the logic to track the device. There were also problems with inheritance in case of multiple sensors.

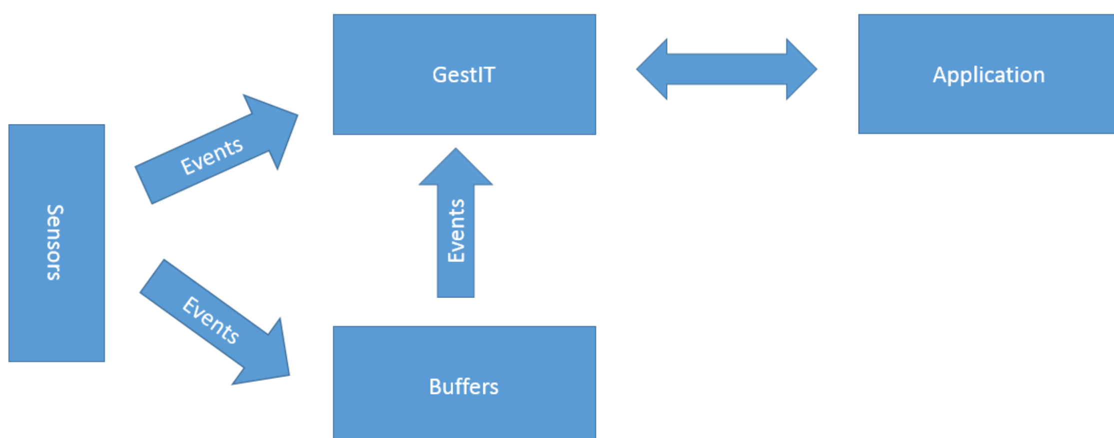
We started to define and analyse an internal mechanism that would work as a history component, to allow the possibility to proceed in the execution of the net and then to backtrack, by restoring the token configuration to a previous state, when needed.

We were looking for the definition in the language of a marker syntax, to express the elements and predicates that the history would need to track and satisfy.

This led to another problem because the grammar was already complex and adding more symbols would result in a drop of readability and practicality of use.

The introduction of the Fusion Sensor added the possibility to unify various types of native and customized event sources, and to detach the native event from the net triggering.

All this reasons led to the idea of making the framework independent from the GestIT library, even though for the development and the definition we



considered the cooperation between the two libraries as one of the primary requirements.

The library has thus been designed as a tool that allows to collect series of data from one or more sensors. It can compute various statistical measurements on these data series and it can be extended with customizable events that trigger when the collected data satisfies a defined predicate.

Another advantage of this approach is that is possible to use multiple instances at the same time for tracking the analysis. This divides the tracking in small independent units and reduces the complexity of their management.

4.1 Architecture

In this section, we will see the main aspects we took in consideration to build the library.

We were developing a general-purpose library, hence we could neither focus on a specific kind of sensors nor develop a list covering all the possible instantiations.

An important factor to consider is that the library should be real-time. It must calculate properties during the execution by elaborating the data received from sensors. This requirement prevents the use of analysis computations that are too deep and complex, otherwise we will lose the reactivity of the framework and the computations would introduce a delay between the input data from the sensor and the data produced by the analysis.

Following these ideas, and to integrate it with the GestIT library, we focused mostly on two aspects which we deemed desirable:

- We want that our framework generates events;
- We want to define events in a declarative way.

The GestIT philosophy was to define each ground term as a feature, representing a simple native event, and a predicate, which was carrying the conditional controls. In our paradigm, we wanted to reverse this approach: the framework will do the analysis and then fire a simple event, which has previously been associated to a complex predicate.

The final implementation of our framework works with three main types: the data types, buffer types and event types.

Buffers collect data acting as listeners of sensor events. The data types implement a common interface, which exposes the data values and guarantees us the possibility to implement a series of methods and properties.

The buffer has members that express in a declarative way properties of the time series.

Events are built using composition of buffer properties. We can define the triggering of an event as a message sent by an object to signal the occurrence of a certain condition. Since we are working in a functional language, we can represent events as a triggering function, which goes from the evaluations of our buffer data to a Boolean.

Events and data are collected in an event buffer, that will receive the input, forward it to the buffers, and trigger the appropriate events.

4.2 Design Choices

Before starting to illustrate the entire framework, we want to point out some choices we made during the development of the framework.

First of all, we started by defining the data structure. We realize that in our domain most of the cases are related to positional tracking of the gestures and checking thresholds on calculated values.

Vision-based devices, for example, tend to give a structured skeletal representation of the input, like the joints of the body.

Often the position is not the only information provided to the developer; it is possible to also find additional information about orientation, velocity, or acceleration. In this case, we can say that the sensor already implements a form of temporal analysis.

In this model of representation, for the purpose of gesture definition, the correlation between two or more tracked joints is usually more significant than the absolute position of each of them.

For this reasons, we decided to keep the data structure as simple as possible, and to offer the possibility to compose it. Every single buffer must cover consistently one aspect of the recorded data input, and then the programmer should define the correlation on them for the generation of the event.

We believe that allowing the user to define up to three values and a time dimension covers most of the cases of practical interest. We represented each of the values as a float, which in F# is a 64-bit floating point type.

In the first stages of the development, we evaluated the possibility to use units of measure [34].

F# supports static checking and inference on units of measure. It can be seen as a special type, added to the type of the value, that guarantees to avoid any measure inconsistency if used throughout the code. In our case, we opted to avoid this feature for two main reasons: first of all the other languages of the .Net Framework do not support it, and second we imagine that in most cases the user will operate mainly on data whose units of measure are uniform.

For data storage we chose to use a structure with a time threshold, which collects new data and automatically discards old data. In an event-driven environment, most of the properties are calculated starting from the last input value. With this approach, we can imagine to have a time window that moves with the evolution of the events. We decided to give a customizable threshold for each buffer, and to give the possibility to add a time span value on the properties, to evaluate them over a smaller time span.

We want to our code to work in a real-time event-driven environment. We need the properties to be evaluated and appropriate events to be triggered in a reactive way as a consequence of the reception of the input data. We had two main possibility for deciding on the implementation:

- Checking on every received input;
- Define a time interval for the control cycle.

We opted for the first option for various reasons.

First, we considered that we expect that the library will work mostly with events associated to one sensor or multiple instances of the same kind of sensor, so we probably want the control to occur whenever new data is received. Moreover, we desire to work with native and customized events together; in this cases, it is probably preferable to have the native sensor and the customized events working at the same frequency.

In the end, our framework makes it possible to filter the input and to track events that occur infrequently; in this cases polling can result in a waste of resources.

To reduce the computational cost of complex customized predicates we added the possibility to deactivate and activate events singularly, for a better reactivity of the framework.

4.2.1 Gesture Analysis

When studying a gesture over time, especially in touch-less devices, one of the aspects that is immediately apparent is that the lack of physical feedback and reference points causes uncertainty and loss of precision in the movement.

In this domain, we cannot expect that a plain mathematical definition is enough to define a gesture. You cannot pretend that the user performs the gesture in a perfect way, and definition which is too strict may lead to false negatives. A system to express the tolerance in the model is required in order to overcome this issue.

Another aspect which needs to be considered is the instrumental error of the sensor, which may lead to erroneous results. This part is more sensor-dependent, and, for this reason, we added the possibility to add a customized filter and the possibility to resample the input.

For the gesture recognition, we focused on developing properties and measures that can cover the following main simple cases:

- A threshold value triggering;
- A standing position;
- A straight movement;
- A curvilinear movement.

With the compositionality, provided by GestIT, is possible to obtain a definition for more complex gestures.

The threshold can have various uses: it can recognize a new entity entering in the sensor view, or define a limit for another gesture (for example do not recognize entities that are over a certain value). Using the average of a value over a short time span instead of a straight value comparison results in constraints which are more robust and avoids false positives caused by instrumental errors.

The standing position is a property typically used as a start of the recognition of the gesture, or for calibration phases. We decided to add an explicit property to cover this case, and to offer the possibility to pass a tolerance parameter, which will be checked on the time series.

For representing directional moments, we thought of different possibilities.

For a straight movement, we can rely on the simple linear regression on the time series to obtain an approximation; this is not sufficient, as we are not guaranteed that the input data point are on a straight line. To evaluate the approximation error, we apply the Euclidean distance from the original points to their linear approximation; if in the error average is under a certain tolerance threshold, a straight line is a good approximation of the movement.

For the curvilinear movement we used the Fast Fourier Transform to recognize periodic movement. The projection along each coordinate of a circular movement with a constant angular speed is a sinusoid whose period is the same at that of the movement. This can be recognized as a spike when the data associated with that coordinate is represented in the frequency domain. The FFT transforms the temporal data to the frequency domain, where this analysis is easier to perform.

For the second possibility we offered, we exploited the advantage of the functional language. The developer can pass two functions to the buffer, one that represents the approximation of the movement, and one to check the matching. The framework will apply the two functions to the input value and return a value which represents the percentage of correct matching.

The position is an important element but it is not the only one which matters when defining a gesture. Another important aspect is its evolution over time. We decided to express these two aspects separately and for this reason we defined properties that compute the typical statistical values, like average velocity, average acceleration, mean position, and standard deviation. The

full definition of the gesture is the composition of these two aspects, that can be expressed as a conjunction of constraints on these values.

The last aspect we want to point out about our construction is that we are working by receiving data from events and producing new events; this means that we can compose not only the properties, but also the buffers themselves. They can be structured in cascade to obtain more complex and interesting properties.

For example, we could have to track a three dimensional point and check if the average velocity of the point is constant for one second, but the developer SDK only offers positional values.

We do not have in our buffers an explicit property which represents this value, but we can build it using two buffers working in cascade. The first one gets the point position, and generates an event every time a new input is received.

A second buffer applies a listener to our first buffer, and records the average velocity value of the last $100ms$. The second buffer will be a simple one-dimension buffer, with one event that fires only if its value is stable for one second. With the composition of two simple properties, we derived easily a complex property.

The following code represents all the steps necessary to obtain this implementation in our framework.

```
let buffer1 = new Buffered3D<_>()
let eventbuffer1 = new EventBuffer<_,_,_>(buffer1)
let ev1 = new TEvent<_,_>(fun x -> true)
eventbuffer1.addEvent(ev1)

let buffer2 = new Buffered1D<_>()
let eventbuffer2 = new EventBuffer<_,_,_>(buffer2)
let ev2 = new TEvent<_,_>(
                fun x -> let valore = (x:Buffered1D<_>)
                valore.StationaryPosition(1000.0,10.0)
                )
eventbuffer2.addEvent(ev2)

ev1.Publish.Add(fun x ->
    let b = x:Buffered3D<_>
    eventbuffer2.AddItem ( new TD1D<_> (b.AverageVelocity(100.0))
    )
```

4.3 Development

In the first part, we will start from the data types, then we will see the buffers and the developed properties, and in the end, we will see the events definition and creation. For the complete code of the library we refer to [35].

4.3.1 Data Types

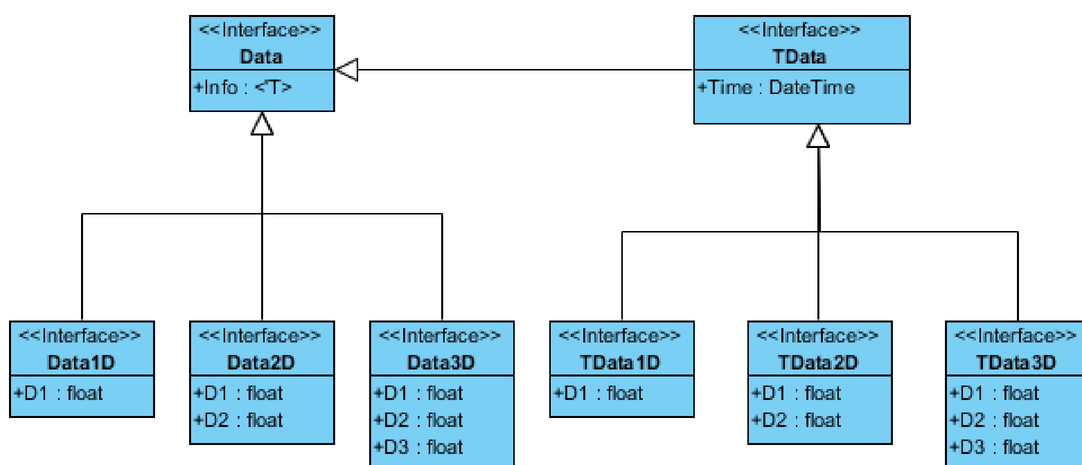
As a first step, we started to define a representation of the collected data. Our aim was to include an abstract interface that can fit in most cases, and give us a unified vision for the future realization of the statistical measurement.

We can assume that the typical application of our framework can be synthetized in two big areas:

- Tracking the changing of a certain value;
- Tracking the movement of a point in the space.

We opted to realize a hierarchy of interfaces that tracks one, two or three dimensions, with a possible fourth dimension representing the time.

This data may not be enough in some cases to distinguish between two instances of them, so we added as an additional field a customizable class, accessible as a field under the name of Info, with a generic type.



All the data types are available under the module *IData* of the framework. All the classes extends the marker interface *Data*, that contains only the info field, and there are three type of data with the time variable, (*TData1D*, *TData2D*, *TData3D*), and three without it (*Data1D*, *Data2D*, *Data3D*).

When instantiating the interface with a data type, the user has to declare the info field type that the type constraint will make uniform for all the data items of the same buffer.

4.3.2 Buffer Types

The buffer types are the types that will collect the data and, with their properties, offer the user some statistical measurement.

All the buffers extend the abstract type *BufferedData*, which implies a type constraint that will lock the buffer to one of the defined data types. The class extends *System.EventArgs* and has an *addItem* method, a function from the data type of the constraint to *unit*.

We build two parallel data hierarchies that fit in two different kinds of buffers. The type constraint will lock every buffer with the expected type to avoid erroneous usages. The buffer working with one of the Data items, without the time dimension, is more lightweight and works with accumulators; the other ones, that save the data for a defined time span which can be adjusted with a threshold value, work with the *TData* types.

The Accumulator Buffers

In some occasions, it is useful to follow a value evolution over a time and have some statistical measurement on it, while the relationship with time and the full series of the data is not interesting by itself for the analysis.

The typical examples are the cases where you want to check that a value does not go over a threshold, or you need to compute repetitions, average or standard deviation.

In these cases, we want to avoid the storage of the entire data set received; for this reason we implemented a more lightweight buffer, that works with accumulators.

We can see the accumulator as a component that offers two main interfaces, one for the managing of the data and another to access to the statistical measurements.

For the input of data, the accumulator offers two possibilities: adding another item, and resetting the cumulative counters.

In our implementation the two methods are exposed through the *Accumulator* interface: as expected from the names, the method *AddItem* is used for the insertion of the received data and to perform the update of the cumulative buffers, while the method *Restart* resets all the variables.

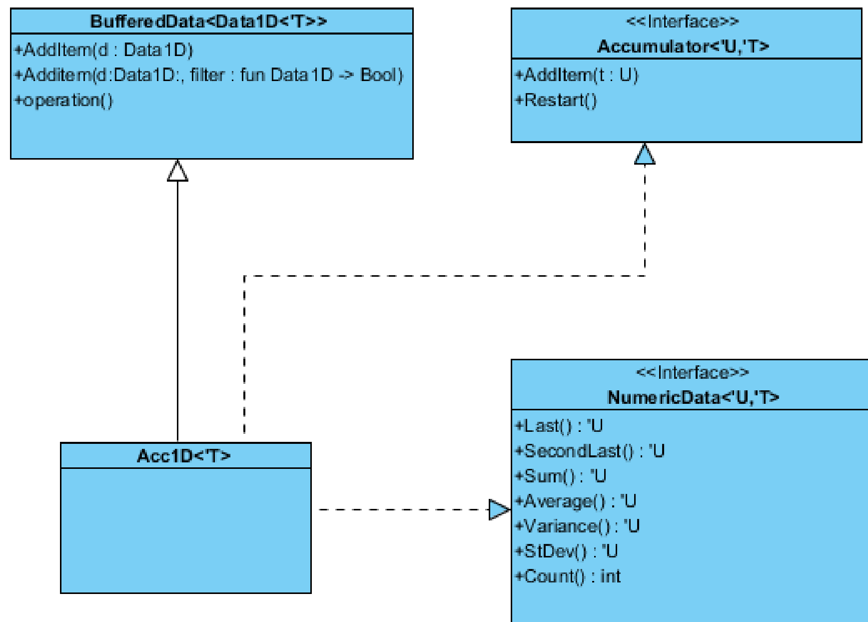
Whenever a new value is inserted, the accumulator will update the item counter, the average and the average of the squares of every dimension and the last item received.

Using the average and the average of the squares of the values it is possible to derive some other statistical values, like the sum, the average, the variance and the standard deviation.

In our implementation, the accumulator buffer exposes the *NumericData* interface, representing the following properties: *Last, SecondLast, Count, Sum, Average, Variance, StDev.*

The first two values represent the last and the second of last value received, while the third represents the number of item received. We save the last two values to allow the user to track an increment or decrement of a nearly static value, like for example the number of items tracked. The other four values represent the statistic measurement of the sum, the average, the variance and the standard deviation.

Every property will return an item of the same type of the inserted data, apart count that will return an integer.



The Sequence Buffers

The sequence buffers store the received data, and make it possible to perform a deeper analysis of the data.

Data are saved depending on the threshold assigned to the buffer. Using *TData* we have the time dimension included, and the buffer automatically discards the data after they go past the time threshold.

As in the previous case, we have three buffer types, whose main difference is the number of dimensions of the data.

Every buffer implements *BufferedData*, which exposes the method *AddItem* to insert new data. It also contains a list data structure (*itemlist*) that will hold all the inserted data.

Moreover, it includes some utility methods that allow to do the managing on the list, like counting the elements, resetting the buffer, defining the cut-off

threshold or getting the entire buffer as a list or as an array of TData, for further custom processing by the user.

Apart from these methods, we will focus on the description of the properties we realized to help the user with the analysis of this time series.

When you have to work on a time series of data, it is important to check the consistency of the data time series. It is important to check the regularity of the data input as first thing, assuming that afterwards we want to check more complex properties over time.

The user needs some tools to check or enforce some consistency properties of the data series has, like:

- the time series covers a certain time extension or duration;
- the signal is received continuously;
- the signal noise and the false positives have been removed.

From this perspective, we realized the following methods:

- *PeriodLength*: allows the user to check that the temporal distance between the first and the last item of the buffer;
- *Cardinality*: checks if a given integer Is bigger than the number of the items in the list;
- *IsContinuous*: has two floats, one representing a time window and another representing an interval. This property returns a Boolean which indicates if in the requested time window the distance between two consecutive data is not greater than the interval time.
- *Sample*: the aim of this method is to resample the data. It accepts a sampling function that will filter the input data and return a new buffer with only the sampled elements.

After this first step, we decided to implement computations to perform analysis of movement and position, under the assumption that we are using a Cartesian coordinate system.

In the next part of the chapter, we will describe the methods we implemented for each measure. The measures we covered are:

- *Distance*:

  We realized the method *TotalDistance* and *ComponentDistance*.

  These two methods allow calculating the point-to-point distance covered by the tracked sensor during a certain time window. The methods receives a float in input, representing the size of the time window, in milliseconds, back from the current moment. The first method allows the user to obtain the point-to-point Euclidean distance covered by the tracked sensor, the second gives the distance obtained accounting every dimension as independent.

  Another possibility offered is to use the method *DifferenceVector*, a function that given a timespan gives back a new buffer with the differential values between pairs of consecutive elements of time data series.
- *Velocity*:

  We made the methods *AverageVelocity* and *InstantVelocity.*

  The first methods is a function from float to float; it takes a value that represents the duration of the time window in milliseconds, backward from the current time. For calculating the average velocity we use the point-to-point distance, so it is important for the user to know that data affected by heavy noise might nullify the precision of the result.

  The *InstantVelocity* method computes the most recent instant velocity if in the last 100 milliseconds there are at least two samples to calculate the value.

  The velocity value is expressed as $units/s$ (where the units depends on the current measure used by the sensor).

- *Acceleration*:

  The methods *Acceleration* and *InstantAcceleration.*

  The average acceleration is calculated as the ratio between the velocity variation in an interval of time.

  $$\bar{a} = \frac{\Delta v}{\Delta t} = \frac{v_2 - v_1}{t_2 - t_1}$$

  The method *Acceleration* is implemented as a function from two floats to float. It allows the user to set the two points, which will represent two time windows: the former represents the duration of the time window that will be used to calculate the old velocity, and the latter to calculate the new one. We allow the user to set this parameter so that it can be tuned depending on the frame rate of the sensor.

  The second method, *InstantAcceleration*, similarly to the velocity case, will calculate the value using some default values to set the two time windows, specifically of 200 *ms* and 100 *ms*.

  The value is expressed in $units/s^2$ (where the units depends on the current measure used by the sensor)

- *Positioning:*

  The *AveragePosition* method allows calculating the average value of the data. The method accepts a parameter as usual representing the timespan window we want to analyze backward from current time.

  The method *StationaryPosition* is a function that given the timespan and a tolerance value, checks if the value has been stable within the given tolerance for the specified amount of time.

  The tolerance is expressed as an absolute value. We considered the possibility to have it as a relative value, but it found it inconvenient when tracking positions.

- *Directionality:*

  We developed various methods to retrieve information about the evolution of the movement. The possibility to express predicates using direction is surely one of the most interesting information achievable

from a data time series. Expressing a complex action as a sequence of directional movement gives a straightforward approach to express most of the usual gestures.

We realized a method that computes the simple linear regression on the data series using the time as the independent variable. The method uses the least square method with the QR factorization. The method name is *FittingToLine* and it has an optional parameter that defines the length in milliseconds of the time window on which the linear regression should be calculated. It returns two values that represent the slope and the offset of the straight line.

The linear regression alone may lead to wrong results, as it does not recognize the difference between a straight line and a curve, so we introduced some other methods to refine the result.

The method *IsStraightDirection* is a function from two floats to a Boolean that accepts a timespan representing the time window to consider and a tolerance value. The method returns true or false, depending if the Euclidean distance between the sensor values and the theoretical values (the ones obtained from the linear regression) are within the tolerance.

Another possibility is offered by the method *FollowingFunction*. This method requires the user to provide the timespan window to consider and two functions: a reference function and a checking function.

The former is a function from floats to our time data: it is used to calculate the theoretical trajectory that will be compared to the experimental values retrieved from the sensor, given the time as the independent variable.

The latter is the checking function: given two time data values it will return a Boolean value representing whether the relationship between

the theoretical values we calculated and the registered values in the buffer is satisfied.

This function will return a float number, which represents the percentage of measured samples satisfying the checking function.

- *Fourier Transform*

  The Fourier Transform has the ability to convert samples represented in the time domain to the frequency domain and vice-versa, and it has many applications in time series processing. One of the most common applications is the analysis of the spectral frequency energy in data series, assuming they have been sampled with evenly spaced time intervals.

  If the sensor we are tracking has a regular frame rate, we can assume that the samples are evenly spaced in time, hence we can apply the Fast Fourier Transform to our time data series.

  The Fourier Transform has many applications for the elaboration of digital signals, in particular it can be used both for frequency measurement and to filter signals.

  In our case, we realized two general methods to use it in our data series.

  The first, named *FFT* makes the Fast Fourier Transform and returns an array of floats representing the result.

  The second method, called *FFFilter* is a function taking a filter as an input and returning a buffer of the same type as the original one. This method performs the transformation, applies the filter on the values, and does the inverse transformation. The filter is defined as a function on Complex numbers that will be applied on the frequence values.

  To perform these transforms in our library we utilized the implementation available in the Math.Net Numerics [36] library. It provides optimized methods and algorithms for calculating the Fast

Fourier Transform. In our case uses the Bluestein's FFT algorithm (also called chirp z-transform algorithm).

4.3.3 Event Types

At this point, the library can be successfully used to study data series using the buffer and data types described so far.

The main reason for developing our library was to use it in cooperation with the GestIT library; therefore, since we are working in a reactive event-model environment, the possibility to define a customized event was the more natural way to interact with it.

We defined a custom generic event type, to integrate easily the two libraries and as well conform to the Microsoft .Net event model.

Events in .Net follow the delegate model, which is based on the observer design pattern [26]. Each event is a provider on which is possible to subscribe delegate handlers to receive notification from it. This results in a push-based notification model, in which messages flow from the provider to the subscribers, that will react with a response to the notification.

We want to realize a generic event that tracks a function based on properties of our buffers, and the usage of a functional language like the F# comes to the aid of our case.

```
type TEvent<'X,'V> (triggerfun : 'V -> bool, ?active: bool)=
    inherit Event<'X>()

    let mutable activity = match active with
                                  | None -> true
                                  | Some  h -> h

    member this.CheckFun(value:'V):bool =
        triggerfun value

    member this.IsActive():bool =
        activity

    member this.SetActive(v:bool) =
        activity <- v
```

As function are considered first-class values in F#, we had the possibility to use them as components of our structured event type. Because of the type constraints, the definition binds the user to express events as properties on data buffers. The result of the evaluation is a Boolean that indicates to the container if it should fire the event. Furthermore, by extending the Event class in the definition, we inherited the observer pattern methods that allow using our event as any CLI Event of the .Net language.

In our case, we wanted to have the possibility to have multiple events working on the same buffer so we decided to implement a container for collecting related events and buffers.

This container acts as the main component to interact with. The user will pass the data, which will be recorded in the data structures; the addition of a new sample will cause the checking of the function on the connected events.

All of our buffer types are extending the *System.EventArgs* class, that is the base type for all event data classes in the Common Language Runtime.

We made two types of containers, one having one buffer and multiple events, and a second one that gives the possibility to have more than one buffer linked as well with multiple events.

The first buffer is called *EventBuffer*. It has only one buffer that is passed at construction time, which will be the event argument returned when any of the connected events is triggered.

```
type EventBuffer<'T,'W,'U> when 'T :> BufferedData<'W> and 'W :>
Data<'U> (data:'T) =

    let eventlist = new List<TEvent<_,_>>()

    member this.addEvent(t:TEvent<_,_>) = eventlist.Add(t)

    member this.AddItem(d:'W,filter:'W -> bool) =
        data.AddItem(d,filter)
        if filter d then this.checkevents()

    member this.AddItem(d:'W) =
        data.AddItem(d)
        this.checkevents()

    member private this.checkevents() =
        eventlist
            |>Seq.filter(fun x-> (x.IsActive() && x.CheckFun(data)))
            |>Seq.iter(fun x-> x.Trigger(data))
```

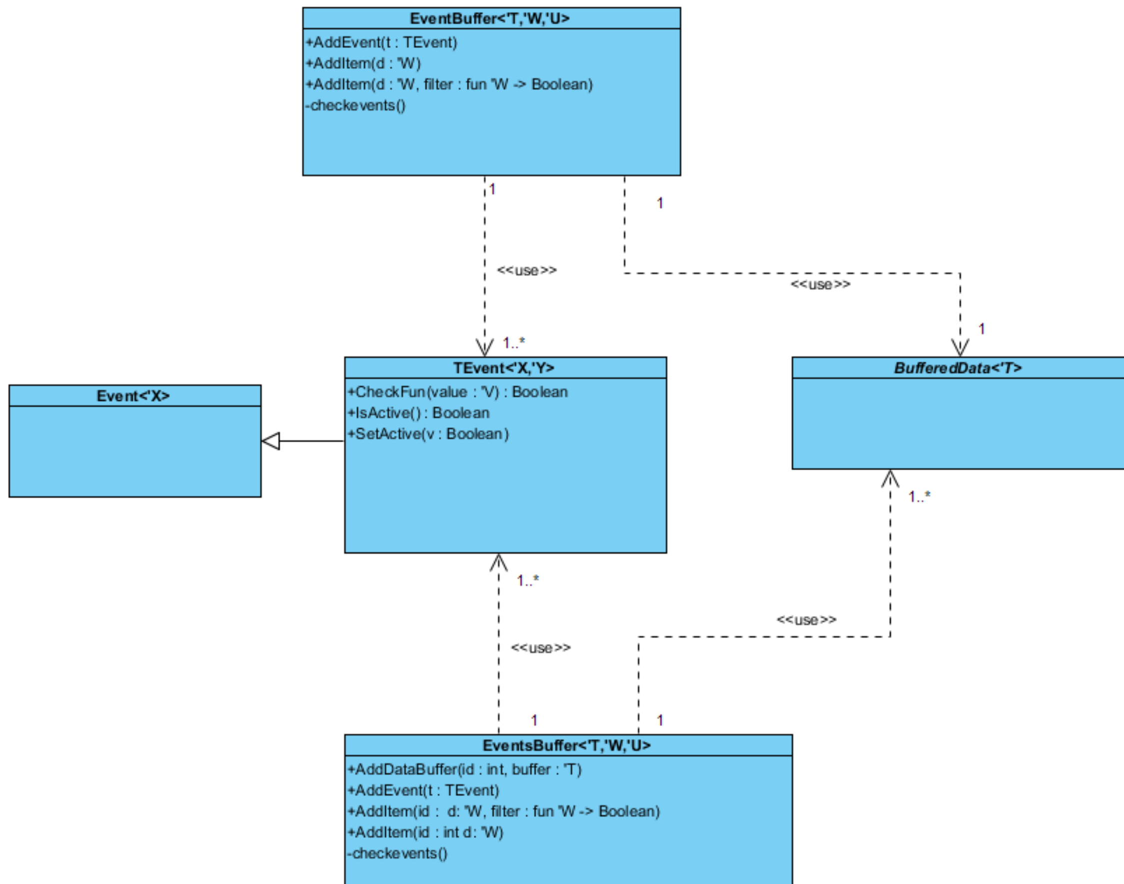
The generic *'T* type is constrained to types that implements *BufferedData*, and with the *'W* type is constrained to ensure that the right data type is passed to the container.

As shown in the code snippets, we have methods to extend the list of events and to add data, so that on every item added, we call the check function of the events with the current data, and in case of positive confirmation we trigger the event.

In the second case, the *EventsBuffer* class, the buffers have to be registered in a dictionary with an ID, represented by an integer. The user in this case has to bind the inserted data with the correct buffer, using the id he assigned and as well must use it in the triggering function of the events.

The only other difference between the two buffer classes is that if the developer is using the multiple data buffer, in the declaration of the object the user he has to pass as parameter a function that maps the dictionary to a subtype of the *System.EventArgs* class.

This function is needed, because in this case we do not have a value that we can directly use as the argument of the event. When the event will be fired, this function will construct the argument parameter associated with the event.



5 A Case Study: Leap Mouse

In order to test our library and to have a feedback from a real application, we decided to develop a mouse implementation with the Leap Sensor. We thought it was a good opportunity to use our library and to see how it integrates with the GestIT library.

The Leap Sensor has some peculiarities that make it an interesting case for testing our library.

First, the sensor has a good precision and responsiveness, and has a push method to send the tracked data. Second, it has some built-in high-level gestures together with the raw data, which we can try to integrate with our system. Third and most important, it has a hand model that guarantees the possibility to use our general-purpose structured system to track data with different natures, such as finger or hand position and properties like cardinality or directional vectors.

The full code of the application is available at [37].

5.1 Problem

We want to simulate all the behaviours of a mouse pointer with the Leap Motion sensor.

It is straightforward to imagine the binding of the Cartesian coordinate system of the Leap sensor with the desktop pixel coordinates, and using a finger as the pointer.

The main two problems need to be solved are:

- Calibration of the active area;
- Define the gestures that will represent the mouse click.

The first problem is a typical issue when you operate with any touch-less sensor. A user does not have a clear definition of the exact dimension of the active area.

Moreover, due to different height, habits, or just the positioning of the sensor respect to the user, giving a fixed dimension of the active space window may lead to an uncomfortable and awkward usage.

Therefore, in our application we want to have the possibility to activate a dynamical calibration procedure of the device, to set the active virtual desktop size depending on the user needs.

For the second part, we must decide how to link the mouse left and right clicks to the Leap Sensor environment.

As already stated when we described the sensor in chapter 3.2, it has a high-level gesture representing the click, which consists in pointing one finger and moving it forward from the user perspective. However, the click gesture is recognized as an atomic event, which starts and ends instantly.

In our case, since we want to realize the base functionality of a mouse, we prefer to have the components of a click, the mouse down and the mouse up event, represented separately. Otherwise, some of the fundamental actions that can be performed using a mouse, like for example drag and drop, cannot be realized, limiting the usefulness of the application. For the right click, we need to define a brand new gesture

In the next subchapters, we will describe the design and the implementation of the application.

5.2 Design

At first glance, the application we want to develop is mainly based on two functionalities, the calibration procedure and the active usage phase.

For the calibration phase, we want the user himself to define the active virtual window, which will associated to the current desktop dimensions.

The calibration procedure works as follows. The application will start by asking with a dialog window to point in the Leap space the point that the user sees as the top left corner of our desktop and to stay in that position for some seconds. After this, another popup will ask the user to point the bottom right corner of his virtual desktop.

Considering that the Leap will be positioned in front of the user in regular parallel position, the two points will represent two opposite vertices of a rectangle. Recording the x-axis and the y-axis of the two points, we can draw the virtual active desktop area. In the normal usage of the Leap Sensor, the z-axis should be nearly perpendicular to the screen plane, so we will ignore it in the procedure of definition of the space.

We could make the calibration work from every possible position, including the third dimension. In this case we would need another point in the space, to compute the relation with the screen. However, this approach would prevent the usage of the built-in top-level gestures and the measurements of the hand model would need additional processing.

The calibration phase will be separate from the usage phase, to make it possible to restart the procedure in case of an unsatisfactory calibration. For this reason, we will add a start and stop button for the mouse simulation usage.

During the usage of the Leap sensor as a mouse, a hiding button for our interface will be desirable to use the mouse over the desktop.

Starting from these considerations, we realized a minimal graphical user interface to interact with the user, and give the possibility to see current virtual desktop coordinates and change the state of the application.

The graphical user interface can be divided in three parts, starting from top to bottom:

1. The status indicator, represented as a led that indicates the connection status of the leap sensor.
   It uses the three color of the traffic light to represent the status: red means the device is disconnected, yellow means that it is busy (connected but inaccessible by the application), and green represents the connected and ready status.
2. The virtual desktop box, a rectangle reporting on the borders the actual maximum and minimum coordinates that the mouse simulator will consider in the Leap coordinate system, the x-axis and the y-axis.
3. The buttons pad, four buttons that offer the possibility for the user to interact with the application. The top two buttons allow activating and deactivating the calibration phase and the usage phase; the “hide” button minimizes the window, and the “exit” button terminates it.

After defining the procedures and the user interface, we need to define the gestures we will use to implement the left and right click on the Leap environment.

As stated before, the left click is a high-level gesture implemented and recognized by the Leap hand model, but we do not want the instant gesture to distinguish the mouse down and the mouse up event.

In our solution, we want the user to perceive it as if it were the same gesture, so we will re-implement it with our library, with the addition of a threshold. The library allows us to represent the mouse down event in a similar mode, and, with the addition of the threshold he should pass during the click, we could add as well add the possibility to realize a mouse up event, associated to the gesture of moving backwards.

For the implementation of the right click, we will use a second finger: we will associate the flicking of a second finger near the principal one to the right click event. More precisely, we will fire the mouse down event when the second finger appears near the first one, and we will fire the mouse up event when it disappears.

The flicking gesture fits well with the usage of the right click, as we consider that in most cases the right click is used to open option menus and normally with the complete click event. Making it an atomic event would still reduce its applications, so in this way we have a gesture which is quite natural to execute atomically with a flick, but still leaves the possibility to distinguish between the mouse down and mouse up event.

5.3 Implementation

In this chapter, we will illustrate the development of the application part by part. For the development, we can view the application composed by four

main components: the leap event wrapper, the mouse controller simulator, the graphical user interface, and the event-managing component.

We will start with the description of the three peripheral components and in the last sub-chapter we will see the event logic part, the core of our implementation, where we use our framework combined with the GestIT library.

5.3.1 The Leap Event Wrapper

When interfacing with the Leap API Leap hand model, as already described in the tools chapter, the user needs to instantiate a Controller object and to add a listener to it.

The listener class offers seven methods for the handling of events coming from the device, in which most are for the status of the device: connection or disconnection, gaining or losing focus, exiting and initialization. To access the frames coming from the device there is the method *OnFrame*.

The frame gives an exhaustive vision of the current state, giving the possibility to access various lists which represent the gestures, hands, fingers and pointable items, plus some global frame properties.

this case it is up to the developer to recognize by id the old or new fingers and to keep track of the correlation between new and old elements, so we decided to define a wrapper of the Leap event to have a convenient representation of the data.

We considered worthwhile to have a logic division of the element list, by dividing the element inside in three lists:

- *New elements*: the element that have an id that appears for the first time in the current frame;
- *Active elements*: the elements that are active in both the current and the frame before it;

- *Inactive elements*: the elements that were active in the preceding frames but that are not active anymore.

This new logic division is provided for each type of element, hands, fingers, pointables and gestures, and as well for each of this cases new built-in events will be triggered. The class was partially built by a previous thesis, and I expanded it to also cover the built-in gesture recognition.

There are three main types in the wrapper: the type *LeapActivity*, a back-up union type of the various type of element in the leap hand model, the type *LeapSensorEventArgs* that will be used as return type of the events, and the type *LeapSensor*, that will extend the listener of the LeapMotion API.

LeapActivity

The *LeapActivity* type is a union type of different integer lists. There is one type for each of the possible cases. This will be used to discriminate the type of the element in the processing of the *LeapSensorEventArgs*.

In this code snippet, we will report the part regarding the gestures; a similar implementation is made for each element type recognized. We will follow this assumption in the next code snippets.

```
type LeapActivity =
| ...
| NewGesture of int list
| ActiveGesture of int list
| InactiveGesture of int list
| ...
```

LeapSensorEventArgs

This class is the type that will be given as parameter in the events associated to a LeapSensor.

```
type LeapSensorEventArgs(f:Frame, ?activity) =
  inherit System.EventArgs()

  member this.Frame = f
  member this.Activity = activity

  ...

  member this.ActivityGestures
    with get() =
      match activity with
      | Some(NewGesture l) | Some(ActiveGesture l) ->
            f.Gestures()
            |> Seq.filter (fun h -> l |> List.exists(fun el -> h.Id = el))
            |> Seq.toList
      | _ -> []
```

The class will inherit *System.EventArgs* to be compliant as event argument type; it has a member frame that will contain the full frame that generated the event. Moreover, it has some read-only utility members that will contain the full list of active and new elements, divided by type: tools, fingers, hands, gestures in a cumulative list, and one member for each gesture separated.

LeapSensor

The *LeapSensor* class is the one that extends the Listener class provided by the Leap Motion API and the main class that the developer will use.

```
type LeapSensor() =
  inherit Listener()

  let controller = new Controller()

  ...

  let mutable activeGestures : int Set = Set.empty

  let newGestureEvt = new Event<LeapSensorEventArgs>()
  let activeGestureEvt = new Event<LeapSensorEventArgs>()
  let inactiveGestureEvt = new Event<LeapSensorEventArgs>()
```

As explained before, for each element recognized by the Leap Motion sensor, we want to have a set that collects all the active elements, and we want to have the classification in new items, active items and inactive items.

```
  member this.NewGesture = newGestureEvt.Publish
  member this.ActiveGesture = activeGestureEvt.Publish
  member this.InactiveGesture = inactiveGestureEvt.Publish
```

For each case, we made an event associated with it and we offered a member that exposes the events outside the class.

As we stated before the Leap Motion Sensor returns frames, so for the handling and processing of the data, we overrode the *OnFrame(controller)* method.

```
override this.OnFrame c =
  let frame = c.Frame()
  frameEvt.Trigger(new LeapSensorEventArgs(frame))

  ...

  let processGestures () =
    let ts = frame.Gestures() |> Seq.map (fun h -> h.Id) |> Set.ofSeq
    let nt = ts - activeGestures
    let at = ts |> Set.intersect activeGestures
    let dt = activeGestures - ts
    if not nt.IsEmpty then
           newGestureEvt.Trigger(
                 new LeapSensorEventArgs(frame,
                                   NewGesture (nt |> Set.toList)))
    if not at.IsEmpty then
           activeGestureEvt.Trigger(
                 new LeapSensorEventArgs(frame,
                                   ActiveGesture (at |> Set.toList)))
    if not dt.IsEmpty then
           inactiveGestureEvt.Trigger(
                 new LeapSensorEventArgs(frame,
                                   InactiveGesture (dt |> Set.toList)))
    activeGestures <- ts


  ...

  processGestures()


  ...
```

Starting from the controller, the method extracts the frame and processes it. It splits the elements of the hand model assigning each one to the corresponding item set, and raising an event for items whose classification changes.

The full code of the wrapper implementation is available under the project *LeapSensor*.

5.3.2 The Mouse Controller Simulator

To access mouse events and to simulate them in the .Net Framework, we had to import a native dynamic link library (dll) of the system.

We realized the module MouseInteroperator that imports the file user32.dll, and made an external call to the method mouse_event.

The extern modifier is used to declare a method that is implemented externally, and it is typically used to call unmanaged or driver code.

The mouse_event method accepts five sixty-four bit integers that descibe the event. The first one represents the event type, the second and the third ones are the x and y coordinate values, the fourth one represents additional attached data (for example the scroll dimension when using the middle scroll button) and fifth one other extra information.

We made some simplified methods to associate symbolic names to the events, leaving the native calls into this file.

```
open System.Runtime.InteropServices


[<DllImport("user32.dll", CharSet = CharSet.Auto,
                           CallingConvention = CallingConvention.StdCall)>]
extern void mouse_event(System.Int64, System.Int64,
                        System.Int64, System.Int64, System.Int64)

let MOUSEEVENTF_LEFTDOWN    = 0x02L
let MOUSEEVENTF_LEFTUP      = 0x04L
let MOUSEEVENTF_RIGHTDOWN   = 0x08L
let MOUSEEVENTF_RIGHTUP     = 0x10L
let MOUSEEVENTF_MIDDLEDOWN  = 0x20L
let MOUSEEVENTF_MIDDLEUP    = 0x40L
let MOUSEEVENTF_MOVE        = 0x01L

let MouseMove(xDelta:int64, yDelta:int64) =
    mouse_event(MOUSEEVENTF_MOVE,xDelta,yDelta,0L,0L)

let MouseLeftClickDown (xDelta:int64, yDelta:int64) =
    mouse_event(MOUSEEVENTF_LEFTDOWN, xDelta, yDelta, 0L, 0L)

let MouseLeftClickUp (xDelta:int64, yDelta:int64) =
    mouse_event(MOUSEEVENTF_LEFTUP, xDelta, yDelta, 0L, 0L)
...
```

5.3.3 The Graphical User Interface

The Graphical User Interface (GUI) has been developed in C# using the Visual Editor available in the Microsoft Visual Studio.

We opted for writing the GUI in the C# language because the visual editor designer does not work with F#, so we took advantage of the language interoperability provided by the Common Intermediate Language in the .Net framework.

The GUI can be divided in two parts, the main form and the dialog window that will be used during the calibration phase.

The main form is the one represented in the previous picture, and it is divided in three parts, from top to bottom: the connection status, the window status, the button pad.

The connection status represents the current situation of the Leap Motion sensor: green stands for connected, yellow for busy and red for disconnected.

The window status is a rectangle with some numbers on the sides, which epresent the current virtual desktop defined with the Leap Motion sensor.

After the calibration procedure each side reports respectively the top, left, right and bottom margin of the virtual desktop area.

The button panel has four buttons: one for starting the calibration, one for the mouse simulation, one for the hiding of the GUI and one for exiting.

The behaviour of this form is defined by the methods associated to the four events representing the mouse click on each of the buttons.

```
public event EventHandler CalibrationClickEvt;
public event EventHandler StartStopClickEvt;
public event EventHandler HideClickEvt;
public event EventHandler ExitClickEvt;

private void bCalibrate_Click(object sender, EventArgs e)
     {
          CalibrationClickEvt.Invoke(sender, e);
     }

...
```

For the interaction with the rest of the interface, there are three public accessible methods:

- *setDesktopMargin*, that receives 4 integer and sets this values as the values of the virtual desktop area of the Leap sensor;
- *switchColor*, that will change the color of the status button;
- *changeStartStopButton*, that renames the label of the button when the mouse simulation is launched.

Another part of the GUI is represented from the dialog box that will appear when the calibration procedure is executed.

It is a simple dialog box with customizable text and button label to guide the user and explain him the steps that he must follow for a correct calibration of the sensor.

The dialog has two public methods *setText* and *setButtonText* and an event *PopupAnnullaEvt*, to be raised if the procedure is aborted.

5.3.4 The Event Logic Module

In this section, we will describe the event logic module, the core of the application. The application has been developed following the model view controller (MVC) pattern, to separate logic, presentation and data.

In the description we will focus first on the GestIT Petri net representation of the two procedures, the calibration and the usage phase, then we will see how we implemented the predicates we needed to realize the customized events, and finally we will see how to compose the parts to work together.

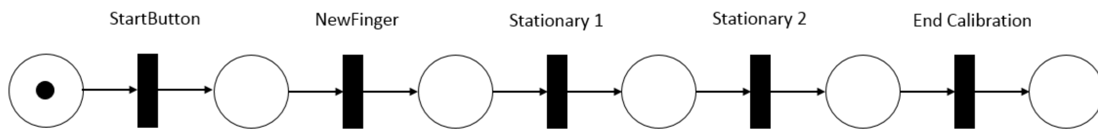
Gestures as Petri Nets

The two procedures, calibration and usage, are independent, we realized two different Petri Nets.

The calibration procedure follows these steps:

1. the user starts the procedure clicking the corresponding button in the GUI;
2. the Leap Motion sensor finds one finger;
3. a dialog window appears, asking the user to point the top-left corner of his virtual desktop window;
4. the user moves the finger in the requested position and wait for further instruction;
5. after the user’s finger mantains a stable position for four seconds, the position is registered;
6. the dialog box asks to reach the bottom-right corner of his virtual desktop;
7. after four seconds of a stable position the second position is registered;
8. a confirmation message appears, the calibration phase ended correctly and the new virtual desktop measurement are registered.

The current behaviour can be represented as a sequence in the Petri net model, defined with the following picture.



The *StartButton* transition is associated with the click event on the GUI to begin the procedure, *NewFinger* represents the appearing of a finger on the Leap sensor active space, the two *Stationary* events will trigger when the user manages to maintain a stable position for the calibration. Finally, the *End Calibration* transition concludes the procedure, with the closing of the dialog box used during the calibration and the registration of the values as new virtual desktop coordinates.

The following code snippet is a builder function that, given the controller, returns the Petri net for the calibration procedure.

```
let calibrazionebuilder(cont:LMController) =

    let newfinger    = new GroundTerm<_,_>(LeapFeatureTypes.NewFinger,
                                fun _ -> not (cont.AlreadyCalibrating()))
    let stable1      = new GroundTerm<_,_>(LeapFeatureTypes.Stabile)
    let stable2      = new GroundTerm<_,_>(LeapFeatureTypes.Stabile2)
    let calibrationend  = new GroundTerm<_,_>(LeapFeatureTypes.Calibrato)

    let calibrating =
      ((newfinger |-> setcalibratingfinger_h cont) |>>
      (stable1 |-> standingTL_h  cont ) |>>
      (stable2 |-> standingLR_h  cont ) |>>
      (calibrationend |-> nomod_h  cont))

    calibrating
```

We use four ground terms, the start button is not associated to a ground term, since it will cause the invocation of the whole net.

Every ground term is associated to an event; all events have been created with our library, apart the New Finger that is a native event.

For the first event, new finger, we associate a predicate to check that the net is not already active, to avoid starting concurrent istance of the procedure.

Each event has a handler associated to perform the requested operations.

```
let setcalibratingfinger_h (controller:LMController)
(sender:_,f:LeapFeatureTypes,e:System.EventArgs) =
       let ee = e:?> LeapSensorEventArgs
       controller.SetCalibratingFinger(ee.ActivityFingers.Head.Id)
       controller.OpenPopupCalibration1()

let standingTL_h (contr:LMController)
                    (sender:_, f:LeapFeatureTypes, e:System.EventArgs) =
       let ee = e:?>Buffered2D<FingerInfo>
       let element = ee.GetListBuffer().[( ee.GetListBuffer().Length - 1 )]

       ee.Clear()
       contr.setmouseTopLeft(element.D1,element.D2)
       contr.OpenPopupCalibration2()
       |>ignore

let nomod_h (controller:LMController)
(sender:_,f:LeapFeatureTypes,e:System.EventArgs) =
       controller.Modify(false)
       controller.ClosePopupCalibration3()
       controller.SetDesktopCoordinates()
```

Respecting the Model View Controller model, each handler does not perform the operation directly, but delegates the operation to the controller object.

Each handler is defined as a function that takes the controller and a triple sender, type and argument as input.

The first one sets the current calibrating finger and opens the first popup. The *standing_TL* handler, where *TL* stands for top left, passes the two values to the controller and clears the buffer, to avoid having a false positive in the next step. Symmetrically there is a handler for the bottom right corner.

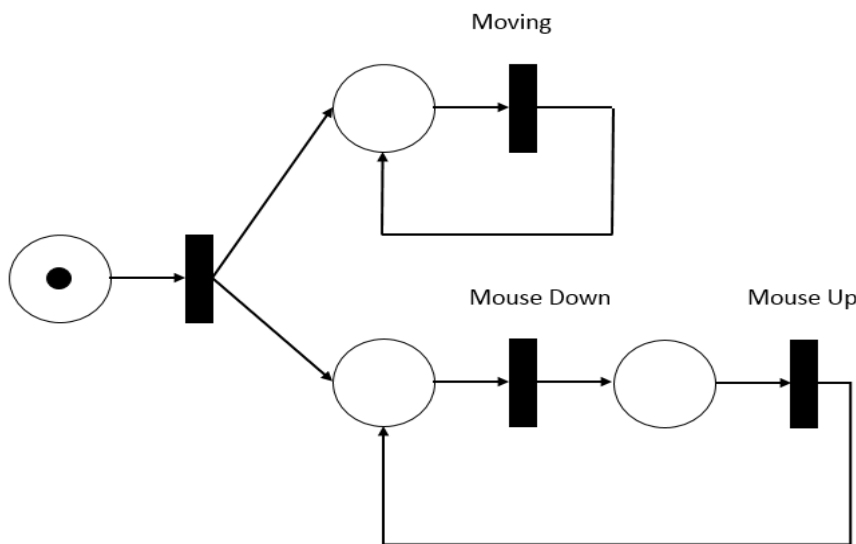
The last handler closes the popup, sets the new coordinates and signals to the controller that the procedure ended.

The second net is the one that performs the live mouse simulation, capturing the movements and the clicks.

By modeling the mouse behavior, we realize we need a never-ending procedure, which can execute the three performed operation repeatedly and concurrently.

In this case, we need to use the iterative operator for each single operation associated, movement, left click and right click, and bind them together with the choice operator.

The equivalent Petri net schema representing this behaviour, as described in [27], is quite complex and lacks of readability. In the following picture, we can see a simplified but significant version of the actual net.



The mouse down and mouse up subnet should be considered as replicate for each kind of click. The net should provide a cancellation procedure to stop the net that is not referenced in the model.

The corresponding code realized to implement the net is the following.

```
let eventbuilder(cont:LMController) =

    let moving     = new GroundTerm<_,_>(LeapFeatureTypes.Moving)
    let rclickDown = new GroundTerm<_,_>(LeapFeatureTypes.RClickDown)
    let rclickUp   = new GroundTerm<_,_>(LeapFeatureTypes.RClickUp)
    let lclickDown = new GroundTerm<_,_>(LeapFeatureTypes.LClickDown)
    let lclickUp   = new GroundTerm<_,_>(LeapFeatureTypes.LClickUp)

    let movement = !*(moving |-> moving_h cont)
    let rightclicks =  !* ((rclickDown |-> rightclickdown_h cont) |>>
                          (rclickUp |-> rightclickup_h cont) )
    let leftclicks  =  !* ((lclickDown |-> leftclickdown_h cont)  |>>
                          (lclickUp |-> leftclickup_h cont) )

    let events = (movement |^| leftclicks |^| rightclicks)

    events
```

The builder creates the ground term, associates the mouse up and down events to the handlers, and connects them together with the choice operator.

As in the previous case, the handlers calls the corresponding method on the controller to perform the correct action.

In the end, for the handling these two nets, and for simplifying the job of the controller we realized a net handler object.

The *NetHandler* class gets the two nets and the sensor and exposes to the user two members for each net, to start and stop the service.

```
type NetHandler(calibration:GestureExpr<LeapFeatureTypes,_>,
                movement:GestureExpr<LeapFeatureTypes,_>,
                sensor:FusionSensor<LeapFeatureTypes,_>) =

     let mutable calibrationnet = None
     let mutable movementnet = None

     member this.StartCalibration() =
            calibrationnet <- calibration.ToGestureNet(sensor) |> Some

     member this.StartMovement() =
            movementnet <- movement.ToGestureNet(sensor) |> Some

     member this.StopCalibration() =
            match calibrationnet with
                         | Some t -> (t :> System.IDisposable).Dispose()
                         | None   -> ()
            |> ignore

     member this.StopMovement() =
            match movementnet with
                         | Some t -> (t :> System.IDisposable).Dispose()
                         | None   -> ()
            |> ignore
```

Realizing Customized Events

In this section, we will describe how we realized the customized *TEvents* and the binding with the sensor.

The first step was to actually implement the data type interfaces, and realize the additional class for the auxiliary informations. We called this class FingerInfo, as we will need to save the finger ID for the calibration phase.

For the calibration phase and the mouse movement, we need to track the two axes, *x* and *y*, needed to define the active virtual window of the Leap sensor. For this purpose we used a sequence buffer, of type Buffered2D with two dimension, for the x and y-axis, plus the time variable.

```
let movementbuffer = new Buffered2D<_>()
let evbuffer = new EventBuffer<_,_,_>(movementbuffer)
```

Then we need to pass to the *EventBuffer* the *x* and *y* coordinates of the fingers; we need to bind it with a function that we will pass to the Leap Sensor as a listener.

```
fun t ->
      let fingerlist = t.ActivityFingers
      if (List.length fingerlist = 1) then
            new Td2d(
             float fingerlist.Head.StabilizedTipPosition.x,
             float fingerlist.Head.StabilizedTipPosition.y,
             new FingerInfo(fingerlist.Head.Id))

            |> evbuffer.AddItem
```

With this function we register the data received from the active fingers of the sensor only if there is only one finger visible, and we choose to register the fingertip coordinates.

From this data, we can build the events we are interested into: we need two events that tracks when the user is pointing the edges of the screen in the calibration phase, one that ends the calibration, and one event that tracks generally the movement of the finger.

```
let stationary(timespan,toll) =
      fun buffer ->  let bb = (buffer:Buffered2D<_>)
                      ( bb.PeriodLength() > timespan &&
                      bb.StationaryPosition(timespan,toll) )

let StationaryEvent  = new TEvent<_,_> (stationary (4000.0,40.0))
let StationaryEvent2 = new TEvent<_,_> (stationary (4000.0,40.0))
let StopModifica     = new TEvent<_,_> ((fun x -> true)
let MovingEvent      = new TEvent<_,_> ((fun x -> true)

    evbuffer.addEvent(StationaryEvent)
    evbuffer.addEvent(MovingEvent)
    evbuffer.addEvent(StationaryEvent2)
    evbuffer.addEvent(StopModifica)
```

To achieve this, we defined a function, called *stationary* that receives two floats representing a time window and a tolerance and using our library properties verifies that the user is in a stationary position and that we are registering data from more than the time window.

We realized two events, even if they will trigger in the same case, because of the limitations coming from the GestIT implementation: in the current version we cannot assign two different feature types to the same event. The movement event triggers every time there is one finger and new data, and, for this reason, we pass a function that always evaluates to true as parameter.

For the left click gesture we realized a sequence buffer with three dimensions, to follow the movement of the fingers along the three axes of the Leap sensor.

The linking procedure is similar and as expected we created two events, *LeftUp* and *LeftDown.*

We associated the left mouse down gesture with the moving of the finger toward the screen, nearly perpendicular in the z-axis, and that it passes a threshold.

To achieve this behaviour we used the following constraint:

- The movement in the z axis, acquired by the component distance predicate, is over 50 points;
- Through the interpolation, we checked that the gradient of the straight is in the right direction;
- We checked that the movement was over the threshold of zero in the z-axis;
- Using the difference vector, we checked that the movement is mostly parallel to the z-axis, to discriminate to some casual diagonal movement;
- We checked the movement tracked by the buffer was continuous, to avoid strange misbehaviours.

The following functional code corresponds to the previous constraints:

```
let leftdown(timespan) = fun buffer ->

  let bb = (buffer:Buffered3D<_>)
  if (bb.Count() <2)
  then
   false
  else
   let dati = (bb.cutBuffer(timespan))
   let x = bb.DifferenceVector(timespan)
   let last = bb.GetListBuffer()
              |> fun x -> (x.Item ( x.Length - 1))
   let x1,y1,z1 = x.AveragePosition(timespan)
   let _,grad = dati.FittingToLine(timespan)
   let _,_,zdist = bb.ComponentDistance(timespan)
   let zunder0 =  List.forall ( fun x -> (x:>TData3D<_>).D3 < 0.0)
                                                    (dati.GetListBuffer())
   if (zdist>50.0 && grad.[2] < -300.0 && zunder0 &&
       x1+y1<Math.Abs(z1-10.0) &&
       bb.PeriodLength()> timespan &&
       bb.IsContinuous(timespan,100.0))
       then
            true
       else
            false
```

For the *LeftUp* event, we realized a simpler event, where mostly we check that the user goes back from the threshold that coincides with the positive values on the z-axis.

To realize the right click gesture, we developed similarly two events, representing the right button up and right button down.

As we stated before, we want to realize the right click by representing it as a flick of a second finger. In this case then we need to track just the number of fingers present in the Leap Motion sensor view.

We do not need deep temporal analysis, and to store all the data received. In this case we opted for the more lightweight accumulator buffer, with one dimension representing the number of fingers.

We can resume the flicking movement in this way: the button down event when the fingers from one become two, the button up when from two active fingers, it goes back to one.

Using the last two saved data of the accumulator buffer, we can easily realize two predicates that express this concept, as the following, and associate to the events:

```
let rightdown() =
      fun x -> let bb = (x:Acc1D<_>):>NumericData<Data1D<_>,_>
               if (bb.Count >2)
                 then if (int bb.Last.D1  = 2 &&
                          int bb.SecondLast.D1 = 1)
                        then
                            true
                        else
                            false
               else false

let rightup() =
      fun x -> let bb = (x:Acc1D<_>):>NumericData<Data1D<_>,_>
               if (bb.Count >2)
                 then
                  if (int bb.Last.D1  = 1 && int bb.SecondLast.D1 = 2)
                     then
                        true
                     else
                        false
               else false

let RightDown         = new TEvent<_,_> (rightdown())
let RightUp           = new TEvent<_,_> (rightup())
```

Finally, we need to associate the events to the feature types previously assigned to the GestIT nets, to link the controller with the nets and the sensor and to make the GUI visible.

```
[<EntryPoint>]
let main argv =
    let sensor = new FusionSensor<LeapFeatureTypes,System.EventArgs>()
    let app = new TrayApplication()

    let gui = new Form1()
    let popup = new PopupDialog()
    let controller = new LMController(gui,popup)
    let calibrazione = calibrazionebuilder(controller)
    let eventi = eventbuilder(controller)


    let leap = new LeapSensor()
    leap.Controller.SetPolicyFlags(
                    Leap.Controller.PolicyFlag.POLICYBACKGROUNDFRAMES)

    ...

    evbuffer.addEvent(StationaryEvent)
    ...
    rightevbuffer.addEvent(RightUp)

    ...

    let upcasting =  Event.map (fun y -> y :> System.EventArgs)

    sensor.Listen( LeapFeatureTypes.Stabile,
                           upcasting StationaryEvent.Publish)
    ...
    sensor.Listen( LeapFeatureTypes.NewFinger ,
                           upcasting leap.NewFinger)

    leap.Connect() |> ignore


    let netshandler = new NetHandler(calibrazione,eventi,sensor)
    controller.setNets(netshandler)

    Application.Run(gui)
```

6 Conclusions

We started the development of this thesis by defining a part of a language to add the possibility of a history modeling into the GestIT library.

The idea of programming the gesture modeling in the new Natural User Interface devices, and in general in the event-based model, with a declarative and compositional approach is an important choice which will drive the future developments.

This approach removes the need for explicit management of concurrency, and it increases the code maintainability and reusability, by making it possible to define of different gestures separately and to compose them with smaller and simpler parts.

In our framework, we tried to continue this philosophy, by defining the time analysis as a conjunction of descriptive properties.

We think our approach implements a good encapsulation of the specific capabilities of a sensor, yet it retains the possibility to reuse and to cooperate easily with the original SDK. The main purpose of this feature is to help the programmer with preserving already-existing code and to customize it for a specific device.

Because of the wide and diverse range of target deices, which differ bothfor technical reasons and for the specific use they were originally designed for, we wanted the library to be based on the most generic measurement.

The strongly typed system of F# and the extensive usage of the type constraints, that certainly have been both a pleasure and a pain during the

development of the thesis, will guide the user towards following the declarative model in the development of his events and will prevent him from using the interface in an incorrect way.

With a good knowledge of the sensor by the developer, and some practical testing of the statistical values on gesture to represent, we think that our solution may help to solve with a good approximation part of the cases that may arise, without requiring additional work.

Using our solution developers can easily to create good approximations of the gestures that they want to represent with no additional work beyond a good knowledge of the sensor and some practical testing of the statistical values.

We think that the focus of future developments should be aimed towards two aspects:

- Studying the possibility to define the timing of the event-checking routine independently from the data received;
- Executing the testing of the library in different domains and with different kind of sensors.

With the separation of the event firing routine from the data, we can allow a mayor customization of the events, for example it is possible to set the desired interval between consecutive triggers of an event.

The testing of the library in different domains or with different kind of sensors may lead to the identification of properties and pattern of gesture definition that a theoretical approach would be unable to recognize as useful. In this cases some testing might find new elements that have not been identified yet, which are required in some application domains.

7 Acknowledgements

Siamo alla fine di questa tesi, nonché alla fine del percorso universitario, mi sento di dire che è giunto il momento di passare ai ringraziamenti di tutte le persone con cui ho condiviso gioie e fatiche in questi anni.

Innanzitutto voglio ringraziare i miei genitori, che hanno creduto in me e senza il loro sostegno, sia morale che economico, sicuramente non avrei potuto realizzare tutto ciò. Ringrazio la sorellina, Stefi, ci sentiamo e vediamo poco ma è sempre vicina, senza contare le volte ormai per cui dovrei ringraziarla solo per avermi rimesso in sesto dopo i vari infortuni.

Ringrazio il Professor Antonio Cisternino, per la disponibilità, l'umanità e la pazienza offerta durante tutto il percorso della tesi, e per avermi fatto conoscere quel bell'ambiente che è il lab.

Un grazie a tutti i ragazzi del lab, Andrea, Marta, Leonardo, Davide, Simone, Marco, Marco, Francesco e Gabriele. Oltre per avermi fatto ritrovare un po' l'entusiasmo per l'informatica, posso dire di aver conosciuto persone eccezionali, ognuna a modo suo. Grazie per i momenti condivisi, di lavoro e collaborazione reciproca, ma soprattutto i vari improbabili discorsi sui massimi sistemi; roba che Kant, in confronto, ha scritto fuffa!

Un grazie a Occhioni, Carmela, che mi ha supportato (o sopportato?) per la maggior parte della laurea… santa donna armata di pazienza, cercava (e ci riusciva) a mettere ordine al mio casino, nonostante le "peggio cose" subite (la lista per lei, sia dei ringraziamenti, sia delle peggio cose, sarebbe lunga, ci fermiamo qua). Un grazie a Fox e Nadia, che uno con l'ottimismo e pragmatismo, l'altra con il suo savoir-faire e ormai signorilità, mi sono stati

sempre vicini… ah, che poi, se non fosse per un paio di scarpe, magari, a Nadia, gli starei pure simpatico! Un grazie al buon Becciu, il nostro infermiere di fiducia nonché salvatore della patria, sempre disponibile, un generoso, peccato va il doppio più veloce di noi tutti e non gli si stia dietro! Grazie ad Andrea, il buon Cirino, famoso per flemma e visione alternativa: sappi che ho temuto una tua domanda durante la presentazione, che avrebbe potuto sconvolgere sia me, ma soprattutto l'intera platea.

Un grazie al variegato mondo del Paci conosciuto e frequentato durante questi studi, e alla Cumpa: Gigio, Marti, Giudi, Adele, Pols, Chiara, Fede, Erika e Andreas; un grazie perché poi soprattutto mi ricordano che, alla fine, ho solo tipo 27-28 anni, e sono gggiovane!

Un grazie alla Pocket, Irene, il tuo "anno giovane" era troppo il top… le camminate notturne in bici e a piedi, resteranno nella storia. Un grazie a Ciuffo per essere Ciuffo, al buon Paolo Cois, ormai in quel di Torino, per le disavventure condivise, al Pedro, Mau e Lik.

Un grazie a tutti i personaggi più improbabili in cui mi sono imbattuto in queste serate pisane, ce n'è di ogni: dalla mitica "Alice che non crede ai cani", ad "Air Oristano"…

Un grazie alla mia vecchia città, L'Aquila, magari sperando che qualcuno si ricordi di ricostruirla veramente e non facendo spot, e ai miei ex-coinquilini e agli amici conosciuti là, un grazie agli amici di sempre, di Bari e di Foggia, anche se, per distanza, ci si vede sempre meno…

Grazie a tutti quelli che si saranno detti: "e che cavolo, ma non mi ha citato?" … e non se la sono presa.

Alla fine, diciamolo… se oggi sono così, è anche un po' colpa vostra!

Leo

8 Bibliography

[1] Nintendo, "Nintendo Wii - Hardware Information," Nintendo, 9 May 2006. [Online]. Available: http://wii.nintendo.com/controller.jsp.

[2] Apple, "Apple IPhone Announcement," Apple, 2010. [Online]. Available: https://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html.

[3] Microsoft, "Microsof Kinect Announcement," Microsoft, 2010. [Online]. Available: http://www.microsoft.com/en-us/news/features/2010/jun10/06-13kinectintroduced.aspx.

[4] D. Wigdor and D. Wixon, Brave NUI World: Designing Natural User Interfaces for Touch and Gesture, Elsevier, 2011.

[5] S. K. Lee, W. Buxton and K. Smith, "A multi-touch three dimensional touch-sensitive tablet," San Francisco, 1985.

[6] J. Davis and A. Bobick, "The representation and recognition of human movement using temporal templates," in *Computer Vision and Pattern Recognition*, San Juan, 1997.

[7] B. H. Juang and R. Rabiner, "Hidden Markov Models for Speech Recognition," *Technometrics,* vol. 33, no. 3, pp. 251-272, 1991.

[8] M. Brand, N. Oliver and A. Pentland, "Coupled hidden Markov models for complex action recognition," in *Computer Vision and Pattern Recognition*, San Juan, 1997.

[9] K. Murakami and H. Taguchi, "Gesture recognition using recurrent neural networks," in *CHI '91 Human Factors in Computing Systems Conference*, New Orleans, 1991.

[10] A. Kar, "Skeletal tracking using Microsoft Kinect," *Methodology,* vol. 1, pp. 1-11, 2010.

[11] Microsoft, "Kinect 2.0 improvements," Microsoft, 17 9 2013. [Online]. Available: http://blogs.msdn.com/b/kinectforwindows/archive/2013/09/16/updated-sdk-with-html5-kinect-fusion-improvements-and-more.aspx.

[12] K. Kin, B. Hartmann, T. DeRose and M. Agrawala, "Proton++: A Customizable Declarative Multitouch Framework," in *User Interface Software and Technology*, Cambridge, Massachusetts, 2012.

[13] K. Kin, B. Hartmann, T. DeRose and M. Argawala, "Proton: Multitouch Gestures as Regular Expressions," in *ACM CHI 2012*, Austin, 2012.

[14] Visual Lab Berkeley, "Proton++ Source," 2012. [Online]. Available: https://github.com/ucbvislab/Proton.

[15] Blue Mountain Capital LCC, "Deedle," Blue Mountain Capital LCC, 11 2013. [Online]. Available: https://github.com/blueMountainCapital/Deedle.

[16] Blue Mountain Capital LCC, "Deedle Documentation," 2013. [Online]. Available: http://bluemountaincapital.github.io/Deedle/index.html.

[17] Microsoft, ".Net Framework," Microsoft, [Online]. Available: http://www.microsoft.com/net.

[18] Microsoft, "Microsoft Visual Studio," Microsoft, [Online]. Available: http://www.visualstudio.com/.

[19] Novell Inc, "Mono Develop," Novell Inc., [Online]. Available: http://monodevelop.com/.

[20] "ISO/IEC 23271:2012: Common Language Infrastructure," ISO/IEC, 2012. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=58046.

[21] "Standard ECMA-335: Common Language Infrastructure," ECMA, 2012. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[22] "F Sharp Software Foundation," [Online]. Available: http://fsharp.org/.

[23] D. Syme, A. Granicz and A. Cisternino, Expert F♯ 3.0, Apress, 2012.

[24] Leap Motion, Inc, "Leap Motion Controller," Leap Motion, Inc, 2013. [Online]. Available: https://www.leapmotion.com/.

[25] Leap Motion Inc, "Leap API," Leap Motion Inc, 2013. [Online]. Available: https://developer.leapmotion.com/documentation/csharp/index.html.

[26] E. Gamma, R. Helm, R. Johnson and J. Vissides, Design Patterns: Elements of Reusable, Pearson Education, 1994.

[27] L. D. Spano, A. Cisternino, F. Paternò and G. Fenu, "GestIT: a declarative and compositional framework for multiplatform gesture definition," in *5th ACM SIGCHI symposium on Engineering interactive computing systems*, London, 2013.

[28] Università di Pisa, "GestIT," 2013. [Online]. Available: https://github.com/GestIT/GestIT.

[29] R. David and H. Alla, Discrete, Continuous and Hybrid Petri Nets, Springer-Verlag, 2010.

[30] C. A. Petri, Kommunikation mit Automaten, 1962.

[31] W. Van der Aalst and K. Van Hee, Workflow Management: Models, Methods, and Systems, Cambridge, Massachussets: The MIT Press, 2004.

[32] K. Jensen, Coloured Petri Nets and the invariant method, Elsevier, 1981.

[33] L. D. Spano, A. Cisternino and F. Paternò, "A compositional model for gesture definition," in *4th international conference on Human-Centered Software Engineering*, Tolouse, 2012.

[34] Microsoft, "Units of Measure," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-US/library/dd233243.aspx.

[35] L. Festa, "BufferData code repository," 2013. [Online]. Available: https://github.com/leonardofesta/Projects/tree/master/BufferedData.

[36] "Math.Net Numerics homepage," [Online]. Available: http://numerics.mathdotnet.com/.

[37] L. Festa, "LeapMouse Repository," 2013. [Online]. Available: https://github.com/leonardofesta/LeapMouse.