MAURICIO CIRELLI

THE MITABLE ENGINE FOR MULTI-TOUCH AND MULTI-USER TABLETOP APPLICATIONS

São Paulo
2015

MAURICIO CIRELLI

THE MITABLE ENGINE FOR MULTI-TOUCH AND MULTI-USER TABLETOP APPLICATIONS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

São Paulo
2015

MAURICIO CIRELLI

THE MITABLE ENGINE FOR MULTI-TOUCH AND MULTI-USER TABLETOP APPLICATIONS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de concentração:
Engenharia de Computação

Orientador:
Prof. Dr. Ricardo Nakamura

São Paulo
2015

FICHA CATALOGRÁFICA

Mauricio Cirelli
The MiTable Engine for Multi-Touch and Multi-User Tabletop Applications / M. Cirelli. – São Paulo, 2015. 98 p.

Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Tabletops. 2. Frameworks. 3. Interfaces Naturais I. Cirelli, Mauricio II. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais. II. t.

À minha família pelo seu incondicional apoio
durante toda a minha trajeória.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, irmão e à Nathalia. A família é o principal pilar que sustenta um homem durante toda sua vida e sem eles nada seria possível e, tampouco, teria sentido. Eles acreditaram no valor que um mestrado agrega às nossas carreiras e me apoiaram incondicionalmente durante mais esta trajetória.

Em seguida, agradeço ao professor Ricardo, pela orientação, apoio, auxílio e interminável dedicação a este trabalho. Não posso deixar de agradecer, também à professora Lucia, que me acompanha desde o início da graduação e é grande responsável por muito do que consegui profissionalmente e academicamente. Ambos me mostraram a importância que um mestrado acadêmico tem na vida profissional e hoje sou capaz de colher os bons frutos desta escolha.

Por fim, agradeço aos meus amigos Marylia, Eric e Helder pela qualidade excepcional do trabalho que conduzimos juntos durante a graduação. Aquele trabalho deu a motivação para este e é a base do desenvolvimento desta pesquisa.

RESUMO

A definição e o reconhecimento de gestos multi-toque são dois dos maiores desafios encontrados por desenvolvedores de aplicações para tabletops. Após a escolha dos gestos, geralmente após um longo e custoso estudo de usuário, os desenvolvedores precisam selecionar ou criar um algorítimo para reconhecê-los e integrá-lo à aplicação e ao hardware.

Muitas bibliotecas e arcabouços para o reconhecimento de gestos multi-toque foram propostos nos últimos anos. Cada um deles buscou endereçar um dos diversos desafios encontrados pelos desenvolvedores quando desenvolvendo protótipos e implementando novas aplicações para tabletops, como a integração entre a camada de aplicação e a interface de hardware. Em uma das etapas de nossa pesquisa, foram identificados quatorze requisitos para tais arcabouços, variando desde o suporte ao multi-toque ao suporte a gestos colaborativos. Entretanto, as propostas anteriores não conseguiram endereçar todos os requisitos identificados.

Neste trabalho, nós apresentamos o MiTable Engine: um arcabouço flexível e configurável, criado com o objetivo de atender a todos os quatorze requisitos. Esta proposta pode ser utilizada tanto para suportar aplicações em mesas interativas para diversos usuários quanto aplicações para tablets e smartphones.

O MiTable Engine foi construído a partir de uma arquitetura de quatro camadas com uma nova proposta de reconhecimento de gestos baseada em pipeline. Nossa proposta é capaz de processar diversas entradas de toque simultaneamente com grande desempenho e se torna muito flexível para personalizações. O MiTable também inclui alguns dos algorítmos do estado-da-arte para reconhecimento de gestos além de um conjunto de ferramentas para criação e inclusão de novos gestos nas aplicações.

Neste trabalho, nós discutimos a engine proposta em detalhes, incluindo sua arquitetura, algorítmos e como cada requisito é endereçado. Para exercitar a engine e verificar seu funcionamento, nós apresentamos duas provas de conceito e desenvolvemos diversos testes unitários automatizados.

ABSTRACT

Gestures definition and recognition are two of the major challenges for tabletop developers. After choosing the gestures, usually after a costly user study, developers must select or create an algorithm to recognize them and integrate it to the main application layer and to the hardware interface layer.

Several multi-touch gestures recognition systems and frameworks were proposed in the past years. Each of them tried to address one of several challenges developers have when prototyping and implementing new tabletop applications and to provide a seamless integration between the hardware interface and the main application. During our research, we identified fourteen requirements for multi-touch frameworks, ranging from supporting multi-touch to collaborative gestures. Although current state of art multi-touch gestures frameworks addresses several of them, there is no unique solution which addresses all the developers needs.

In this work, we present the MiTable Engine: a flexible and configurable multi-touch gestures engine aimed to address all these requirements. The proposed engine is suitable for both large multi-user surfaces and for small single-user tabletops, such as tablets and smartphones.

The MiTable Engine is built on top of a four layers architecture and introduces a novel multi-touch gestures recognition pipeline which can process several multi-touch inputs simultaneously with high performance and flexibility for customizations. The Engine also includes some of the state-of-art multi-touch gestures recognizers and a set of tools for creating and adding custom gestures to the application.

In this work, we discuss the proposed engine in deep details, including its architecture, its algorithms and how it addresses each requirement. In order to exercise the engine and verify its functionality, we present two proof of concept applications and developed several automated unit tests.

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API Application Programming Interface

CLI Command-line interface

CSCW Computer-Supported Collaborative Work

DI Diffusion Illumination

DTW Dynamic Time Warping

FTIR Frustrated Total Internal Reflection

GRANDMA Gestures Recognizers Automated in a Novel Direct Manipulation Architecture

GUI Graphical User Interface

HAL Hardware Abstraction Layer

HCI Human-Computer Interaction

IR Infrared

KNN K-Nearest Neighbours

LDA Linear Discriminant Analysis

NUI Natural User Interface

RFID Radio-Frequency Identification

SDK Software Development Kit

TUIO Tangible User Interface Objects

XML eXtended Markup Language

CONTENTS

1 INTRODUCTION

The way humans interact with computers has changed a lot in the past two decades. While during the 90's, we had basically desktops and some notebooks with which users interact using mice and keyboards, during the past decade we have seen the mass production of smartphones and tablets, which allows users to interact with in a much more natural way: by touching.

With the recent development of smart televisions, new video-game consoles and interactive tables, users can now interact with these devices without even touching them. Devices such as Kinect[1] and LeapMotion[2] can detect movements users perform and identify their actions, generating commands to the applications. Baudisch presented several of these novel interaction techniques in keynotes at important conferences, such as the International Conference on Entertainment Computer 2013 (ICEC '13) in São Paulo (BAUDISCH, 2013).

These novel devices are not only providing more intuitive ways of interaction, but are changing the way computers perceive their users. In the 70's and 80's, computers *perceive* users as keyboards. With the introduction of graphical user interfaces and mice, computers started to *perceive* users as coordinates on the screen (given by the mice's pointers). With these novel devices, computers perceive users as humans and try to *understand* their actions, their bodies, their voices and their *gestures*.

Although the way computers perceive touch input is not much different than the way they perceive the input from mice (as a set of coordinates on the screen), the way users interact with computers is completely different and much more natural. The human-computer interaction (HCI) evolved from the Command-line Interfaces (CLI) to the Graphical User Interfaces (GUI) and, now, is moving towards the Natural User Interfaces (NUI) (SEOW et al., 2010), putting together touch devices, computer cameras, movement sensors and other technologies in order to build a completely new way of interaction.

These new multi-touch based devices have raised several needs from the software point

[1] Microsoft Kinect: `http://www.xbox.com/en-US/KINECT` - accessed at 15th, March, 2014
[2] LeapMotion: `https://www.leapmotion.com/` - accessed at 15th, March, 2014

of view. The issues software developers must solve in order to support these new ways of interaction are the main topic of this research and will be further discussed in the following chapters.

1.1 Motivation and Objectives

The movements users perform on the touch surface with their fingers or other tangible objects are called *gestures*. These gestures vary from usual *tap* (to click) or *drag* (to move) to more complex trajectories with multiple fingers at once, according to what is more natural for the user in order to activate some function in the applications. Chapter 2 discusses in deep details the definition of *gestures* and how they are used in tabletop applications.

Although there may be a set of common gestures that we can identify in different applications, each application has a set of unique functions that may be activated differently. While in common GUI applications, these functions are activated by a keyboard shortcut or a menu item, in new NUI applications, these functions may be activated by a very intuitive gesture.

However, as what is intuitive for one user might not be as intuitive for another one, this set of gestures may became very complex and large. Hinrichs and Carpendale (HINRICHS; CARPENDALE, 2011) studied how differently users may interact with the same application by using gestures. Their study showed that for a single action, users may perform a different set of gestures. This set of complex gestures must be identified by the application very accurately, in order to execute the right action, and, at the same time, it must be very fast, in order to not harm the user's interaction. The discussion of which gesture is more natural for a given task is beyond the scope of this research. However, in order to support these natural interfaces, it is required to develop an accurate and fast recognition system.

Our goal with this research is to develop a novel framework for gestures recognition which can be integrated to tabletop applications in order to enhance the interaction based on multi-touch gestures and to ease the development of new tabletop applications.

In chapter 3, we review the state of art multi-touch gestures recognition systems and discuss their strengths and weaknesses, enumerating fourteen requirements for multi-touch gestures recognition systems.

The result of this research is a multi-touch gestures recognition engine, which addresses those requirements and provides a set of tools and Application Programming Interfaces (API) to ease the creation and integration of new gestures to tabletop applications.

1.2 Organization

This work is organized in four chapters, besides this one. Our goal is to build a gestures engine for tabletop applications. Thus, we start by understanding tabletop interaction and its applications. In Chapter 2, we introduce the definition of *gestures*, how they are used in several tabletop applications and scenarios and the underlying hardware technology used in tabletop devices.

The most important task of a gestures engine is to interpret the gestures performed by each user and execute its respective action. In Chapter 3, we review the state of art multi-touch gestures recognition systems and discuss their strengths and weaknesses. The result of this analysis is a set of fourteen requirements for such recognition system which guides the development of the MiTable Engine.

In Chapter 4, we present the MiTable Engine itself, exploring its strengths, discussing how it addresses these fourteen requirements with a novel gestures recognition pipeline. Finally, we conclude our work in Chapter 5, summarizing this research, its limitations and pointing out directions for future researches.

2 TABLETOP INTERACTION

In this research, we define tabletops as computer devices with an integrated multi-touch surface (a multi-touch sensor). This sensor may be attached to the computer monitor, but it is not required. Several devices fall into this definition, such as smartphones, tablets, interactive tables, multi-touch notebooks and multi-touch desktops.

Interactive tables are a subtype of tabletops. They are most often table-like shaped artifacts with integrated displays or projections from above or below. They are based on computing devices embedded into them and use an infrastructure of sensors and actuators (GEYER et al., 2011). The provided interface is similar to smartphones, but with the difference of not having the same physical limitations, allowing more complicated gestures to be performed.

As we are going to discuss in this chapter, users can interact with the device using many different techniques: from voice commands and traditional mice or keyboard to cellphones and other tabletops. However, the most fundamental way to interact with tabletops consists in one or more users controlling the device by making gestures or putting objects into the surface (called *tangible objects*).

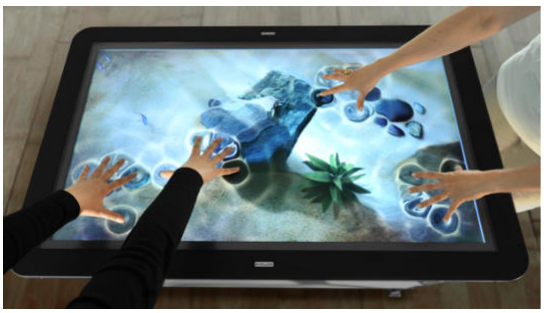

Figure 2.1: Evoluce One

In this chapter we are going to present the current state of art concepts in the tabletop interaction domain from the human-computer interaction perspective. We focus on large interactive tables, which allow the interaction between several users at the same time. Examples

of such devices are the Evoluce One[1] (Figure 2.1) and the ReacTable[2] (Figure 2.2).

Figure 2.2: ReacTable

We start by briefly describing the most recent and most common multi-touch technologies used in these devices in order to detect when one or more users touch the screen and, then, we go deep into the details of important concepts such as gestures and collaboration on tabletops and their applications. In the following sections, we have restructured and updated the literature review provided by our previous work (LIOU et al., 2012). These concepts will support the requirements for developing tabletop applications, multi-touch engines and gestures recognizers, as shall be discussed in the following sections and chapters.

2.1 Multi-touch Technologies

In 2009, the NUI Group[3] published a book (NUIGroup, 2009) discussing the most common multi-touch technologies. They define multi-touch technology as a set of interaction techniques which allow users to control the application using several fingers. It is worth to note that this definition does not require the fingers to *touch* the surfaces directly.

Common smartphones and tablets use resistive or capacitive sensors in order to detect touch inputs. Capacitive sensors detect users input by sensing the human capacitance when the fingers get close to the sensing surface. Only conductive materials and those which have a dielectric different from that of air can be used to touch the surface. On the other hand, resistive touch consists in two flexible sheets coated with a resistive material. One of them has vertical lines and the other has horizontal ones. When the finger touches the surface, it presses one sheet to the other and a contact is made in the exact point where the finger is located.

[1] Evoluce One: `http://www.evoluce.com/multitouch-table.htm` - accessed at 12th, March, 2014.
[2] ReacTable: `http://www.reactable.com/` - accessed at 12th, March, 2014.
[3] NUI Group: `http://www.nuigroup.com/` - accessed at 12th, March, 2014.

However, using these technologies to build large multi-touch surfaces is almost impossible, due to the loss of precision and high costs. For these large devices, other technologies have been proposed and are commercially available, which we are going to briefly describe in this section. Although *touch* interaction may suggest that the fingers must actually *touch* a surface, this is not the case for the most of the current state of art multi-touch technologies for large multi-touch surfaces.

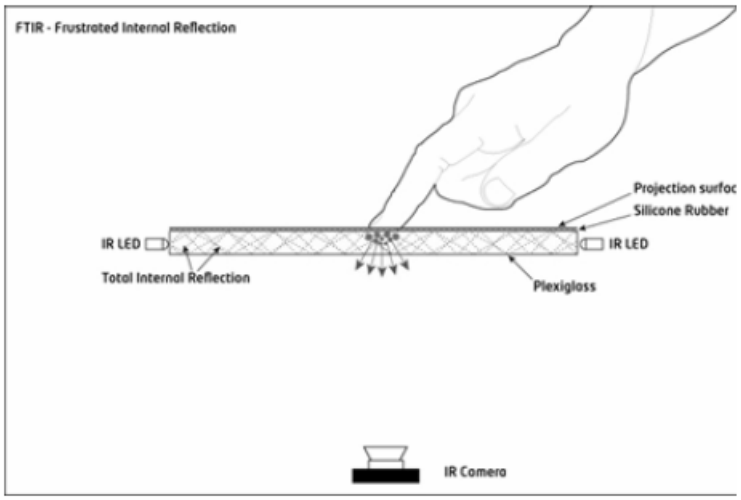

Figure 2.3: FTIR schematic diagram, from the NUI Group's Book

Frustrated Total Internal Reflection (FTIR) is an optical technique which consists in detecting reflected infrared (IR) signals using an IR camera sensor. In this technique, two IR emmitters generate reflected IR signals inside the touch surface material. When a finger gets close to the surface, it reflects these internal IR signals towards the IR camera sensor (Figure 2.3).

The Diffusion Illumination (DI) is another technique based on IR light. The Infrared light is shined at the screen from below the touch surface with a diffuser material placed on the top or on the bottom of this surface. When an object touches the surface, it reflects more light than the diffuser material, allowing it to be detected by the camera. This is the technique employed at the Microsoft's Surface Table (Figure 2.4).

Several authors (MORRIS et al., 2006; HARTMANN et al., 2009; KO et al., 2011) have used computer vision to detect the screen touches, using one or more cameras to capture the images of users' hands. Masoodian et al. (MASOODIAN; McKoy; ROGERS, 2007) have also used cameras, but each user has his own private *touchpad* to interact with the system.

Another approach to build a large multi-touch surface has been proposed by Qin et al. (QIN et al., 2011). They have combined several smaller multi-touch surfaces into a single and large table, with a capacity for up to ten simultaneous users.

Figure 2.4: Microsoft Surface Table

In some applications, there is the need to identify which user each finger belongs to. Marquardt et al. (MARQUARDT et al., 2011) proposed an electronic glove to identify which user is touching the surface. Their system is also capable to identify if the hand is touching the screen from its front or back and if it is a left or right hand. Hutama et al. (HUTAMA et al., 2011) employed movement sensors to identify the users and Meyer et al. (MEYER; SCHMIDT, 2010) used an IR emitter wristband. Puckdeepun et al. (PUCKDEEPUN et al., 2010) investigated the use of a stylus (pen) and IR accessories in educational interactive boards.

Recently, some new approaches to identify the users by their touch input have been proposed. Holz and Baudisch presented a novel multi-touch surface which is capable of capture the user's fingerprint while touching the screen. This way, it it possible to identify which user has made which action very accurately (HOLZ; BAUDISCH, 2013).

Regardless of the technology employed by an interactive table, we are going to refer to the detection infrastructure as *multi-touch sensor* or, simply, *sensor*. For simplicity, this definition considers both electronic devices which are part of the multi-touch sensing system (cameras, sensors, cables and other elements which might be present in a given technique) as their software level drivers and their API's for a given operating system.

Regarding the software technology, most of these interactive tables come with a Software Development Kit (SDK) and a high-level API in order to support the development of new applications. While some of these toolkits come with a set of predefined basic gestures such as *drag and drop* or *tap*, there are some frameworks which allow the developers to identify their own gestures from the input data.

The Tangible User Interface Objects Protocol (TUIO Protocol) (KALTENBRUNNER et al., 2005) has became the *de facto* standard protocol to provide touch information from the sensors to the applications and is implemented by most of current state of art interactive tables. This protocol is capable of reporting the touch-point position and its acceleration for each finger touching the surface. Depending on the sensor's capabilities, it is also possible to

receive information about objects in contact to the screen. The second version of the protocol includes support for many different sensors, such as Radio-Frequency Identification (RFID) readers and accelerometers (KALTENBRUNNER, 2014).

This protocol provides an abstraction of the sensor hardware because any device which is able to detect the position of each finger and track its trajectory is able to report the touch information to the applications using the messages defined in the protocol. While the ReacTable (JORDA et al., 2007) uses computer vision to detect the tangible objects and fingers, the PQLabs Table[4] uses a technique similar to FTIR, both of them implement the TUIO protocol to report the touch events.

In this section we have discussed the most common and important technologies used in multi-touch interactive tables. However, this is definitely not an extensive discussion. Other multi-touch tabletops have been proposed by Wilson et al. (WILSON; SARIN, 2007) and Dietz et al. (DIETZ; LEIGH, 2001) with their own toolkits and features.

As we could see in this brief introduction, there is not a predominant hardware technology and this field is still under heavy development. From the software perspective, although TUIO protocol has been implemented by several devices, each device comes with its own toolkit for applications development.

For more in depth analysis of multi-touch technologies from the hardware and software point of view, readers are directed to the NUI Group's book (NUIGroup, 2009), Hakvoort paper (HAKVOORT, 2009) and Zeitler's survey (ZEITLER; HUSSMAN, 2009) on multi-touch technologies and toolkits.

2.2 Multi-touch Gestures

In this section we are going to discuss the definition of gestures in the context of interactive tables and how they can be classified according to their characteristics from the HCI perspective.

Gestures are the most fundamental way users interact with multi-touch tables. Back to 1986, Rhyne and Wolf defined gestures as hand markings, entered with a stylus or a mouse, that indicate scope and commands (RHYNE; WOLF, 1986). We may rewrite this definition without the need of a stylus or mouse: gestures are movements users perform with their hands in order to activate an application's function. This is a very general definition which is applied

[4]PQLabs Multi-Touch Table: `http://multitouch.com/` - accessed at 14th, March, 2014

to several different contexts, such as multi-touch surfaces (HINRICHS; CARPENDALE, 2011), accelerometer-based gestures (WU et al., 2009) and combining hand movements and tangible objects in the screen (JORDA et al., 2007).

With respect to the space in which the gestures are performed, we can classify them into two categories: two-dimensional (the gestures are performed on the touch surface) and three-dimensional (hand gestures are performed freely in the air) (CHEN; FU; HUANG, 2003; YOON et al., 2001). 3D gestures often use cameras and computer vision techniques to extract information from the input data. An example of this method is the work by Wilson (WILSON; BENKO, 2010) that shows the interaction between multiple large screens.

In the context of interactive tables, gestures may also be classified as *on the surface* or *above the surface*, with no physical contact, gathering information about one or more hands and their projections to the touch surface. Strothoff et al. (STROTHOFF; VALKOV; HINRICHS, 2011) use the projection of users hands to control a triangle in a three dimensional virtual space.

According to their semantics, gestures may be classified into two categories: symbolic and direct manipulation gestures (WOBBROCK; MORRIS; WILSON, 2009). The former category contains gestures defined by their trajectory and they represent a context-dependent sign. The same gestures may have no meaning if performed in other contexts. The latter are the most common gestures for direct manipulation of interface objects, like moving or rotating an image in a picture viewer application (MORRIS et al., 2006) or sharing ideas in a brainstorming activity (CLAYPHAN et al., 2011). They are used in many different applications and, thus, are weakly dependent on the application's context.

Nygärd proposed a set of criteria to classify gestures with respect to their shape and trajectory (NYGARD; THOMASSEN, 2010). According to his classification, gestures may belong to three groups: *open trajectories*, *closed trajectories* and *crossing trajectories*. Examples of such gestures are given in Figure 2.5.

Figure 2.5: Examples of gestures classification accordingly to Nygard's criteria

In large multi-touch tables, it is common to have several users interacting to each other through gestures performed on the surface. In this context, gestures may be *collaborative* or *individual* gestures. When the gestures are collaborative we have two or more different users

combining gestures in order to execute a single action in the application (MORRIS et al., 2006). The collaborative interaction will be further discussed in the next section, as it plays a major role in interactive tables.

Finally, gestures may have different meanings when combined in a well-defined sequence (CIRELLI; NAKAMURA, 2014). Atomic gestures are single movements which users perform with their hands in order to activate a single function in the application. Several examples fall into this category, such as dragging an image around the screen, tapping a button or activating a character's skill in a game.

Sequential gestures are sets of movements which users must perform with their hands respecting a set of time constraints in a well-defined sequence, in order to activate a single application's function. As an example of use of this kind of gesture, we may describe the use of special character's skill in a game (also known as *combos*). Analogously, in traditional devices, such as controller based video-games and computer games, users must perform a sequence of commands in order to activate a character's special action. Games such as those from the *Mortal Kombat* and *Harry Potter* trademarks make heavy use of this kind of interaction. In gesture-based devices, such as *Nintendo Wii*, we also have games, such as *Dragon Ball Budokai Tenkaichi 3*, which uses several accelerometer-based gestures in a well-defined sequence in order to execute a full special action.

Therefore, *sequential* gestures are sequences of *atomic* gestures in which the time constraints between each gesture in the sequence must be respected. In order to have *sequential* gestures available in applications, the sensors must report the gesture information (id and coordinates) as well as the time each event has been detected.

In this section we started with a very generic definition of gestures. Then, we have discussed this concept in the interactive tables domain from different perspectives. Table 2.1 summarizes how different gestures may be classified accordingly to several criteria.

Table 2.1: Gestures classes and criteria

Criteria	Classes
Degrees of Freedom	2D or 3D
Spatial	On the surface or above the surface
Semantics	Symbolic or direct manipulation
Trajectory Complexity	Open, closed or crossing gestures
Multiple users	Individual or collaborative
Timing	Atomic or sequential

2.3 Collaboration with Interactive Tables

Historically, ordinary tables have played an important role in real-time and co-located collaboration. In this section we are going to show in deep details how interactive tables can enhance the collaboration mechanism, enhance the content management and integrate paper media to several other digital devices, such as tablets, smartphones, auxiliary displays and notebooks (WIGDOR et al., 2006).

Accordingly to Tse et al. (TSE et al., 2007) a good collaboration is a consequence of:

- A shared screen
- Users awareness about other users activities
- How people communicate and share ideas through voice commands and gestures

The collaboration in multi-user environments is strictly related to a more general concept: the Computer-Supported Collaborative Work (CSCW). In this context, Scott et al. (SCOTT; GRANT; MANDRYK, 2003) classify tabletops in four different categories of CSCW devices:

Digital Desks
: Designed to replace the traditional tables, integrating paper and digital media

Workbenches
: Users interact with the digital media from a virtual reality environment projected above the table surface

Drafting Tables
: For single users, this device has been designed to replace the traditional artist tables

Collaboration Tables
: Tabletops designed to support the collaboration and content sharing of small groups

From the software perspective, *groupware* is another general and important concept related to CSCW and large interactive tables. Greenberg defines *groupwares* as *collaborative softwares*. They are computer systems designed to support several people working together in order to achieve the same goal (GREENBERG, 1991).

By definition, all Tse et al. criteria are met by collaboration tables running a *groupware*, without loss of generality for individual tasks. In fact, some individual tasks may also benefit

from a large screen and a more intuitive input. However, the collaboration happens naturally on interactive tables (KRUGER et al., 2003) due to the following unique characteristics of these tabletops (APTED; COLLINS; KAY, 2009):

Collaborative Interaction

Collaborative work is a key activity and shall be focused in small groups

Context of Use

Interactive tables shall be located in shared spaces, suggesting a collaborative work

Orientation

Elements on the screen or surface may assume arbitrary orientations as users may sit around the tabletop

Tabletop Size

There is significant differences between the surface size in the tabletop domain

Human Reach

Due to its big size, sometimes a user may not reach an element on the surface without interfering in other users activities

All of these characteristics suggest collaborative works. Due to the large screen dimensions, an user may not reach an element without asking help for another user. Analogously, due to the lack of a fixed orientation, users may also need help from a better positioned user in order to understand some content on the screen. Finally, due to the proximity between the users, all of them share the same context of use, stimulating the communication between the participants and helping them to achieve the same goal.

Morris et al. (MORRIS et al., 2006) introduced the concept of *cooperative gestures*. These special gestures are also known in literature as *collaborative gestures* or *multi-user gestures*. These gestures are defined as a set of gestures performed by two or more users in order to activate a single application's function.

Note that this is different from the *sequential gestures* discussed in Section 2.2. Collaborative gestures must be performed by two or more users, simultaneously or sequentially. On the other hand, sequential gestures are performed by only one user sequentially.

Collaborative gestures enhance the participation of all users during the collaborative task and enhance the awareness of the group about important actions, such as saving the current

work or closing the application. They also ease the content sharing process, enhancing the user's reach, and improve the social aspects of the collaborative task.

Collaborative gestures are useful and desirable when the result of the action triggered by them affects all the users involved in the gesture. Designing collaborative gestures overlooking this characteristic may sadly harm the user experience. An special attention must be taken when parts of a collaborative gesture are also a valid single gesture for the application. These situations should be avoided during the design of the gestures vocabulary.

Collaborative gestures may be classified according to six criteria:

Symmetry

A collaborative gesture is said to be *symmetric* when all participants involved performs the same movements. Otherwise, the gesture is said to be *asymmetric*.

Parallelism

A collaborative gesture is said to be *parallel* when all involved users perform a movement at the same time. Otherwise, the gesture is said to be *serial* or *sequential*.

Proxemic Distance[5]

A collaborative gesture is said to be *intimate* when participants must physically touch each other. It is said to be *personal* when participants must touch the same digital object. It is said to be *social* when participants must touch the same display. Finally, it is said to be *public* when users perform their actions on separated devices (not in the same display).

Additivity

A collaborative gesture is said to be additive if it is meaningful when executed by only one user, but its effect is amplified as more users perform the same gesture. It is a special case of *symmetric* and *parallel* gestures.

Number of Users and Number of Devices

The complexity of the collaborative gestures increases with the number of users who participate on it. Also, collaborative gestures are simpler when performed on a single and shared display, as users may learn easier from others.

In this section we have discussed how interactive tables are suitable for collaborative tasks and we have introduced the concept of *collaborative gestures*. However, there are two major concepts, strictly related to the unique characteristics of collaborative tables which are useful

mechanisms for collaboration: territoriality and orientation (TANG, 1991). The first one is related on how people use and share content in the table, while the other describes how content orientation influences the collaboration. Finally, we end this section discussing some heuristics and guidelines for interactive tables software development.

2.3.1 Territoriality

Scott et al. (SCOTT et al., 2004) published a study on how the interactive surface area is used by several users during collaboration. They have found that this space is split into three parts, called *territories*: the *personal territory*, the *group territory* and the *storage territory*. Each territory has its own functional and spatial characteristics, which we are going to describe in this section.

2.3.1.1 Personal Territory

In their experiments, they found that people used the space near the borders as their personal and private territories. In these areas, users are able to perform individual tasks, without interfering on other users activities.

Although their name suggests that the personal territories are useful only for individual activities, they play an important role in collaboration. Despite of allowing users to perform their own tasks, they are a safe place to test and experiment new ideas before introducing them to the rest of the group.

These territories are located in front of their owners, very near to the border of the surface. Thus, their size and quantity depend on the number of participants, the size of the table and the way people are standing (or sitting) around it. The size of these territories are also dependent on the activity being performed: if it is mostly collaborative, the personal territories tend to shrink; on the other hand, if it is mostly individual, they tend to grow. They found also that people are very opportunistic when disputing territories: they tend to take ownership of as much space as they can.

2.3.1.2 Group Territory

The group territory is all the surface area that is not used by personal territories. In these territories, people share content between the participants, help other users in several tasks and collaborate in the main activities.

2.3.1.3 Storage Territory

During their study, authors found that users organize resources into piles around the table. These piles of resources move around all the surface, from the personal territories to the group territories and vice-versa.

One of the most important characteristic of these territories is the ownership. These territories belong to a respective user if they are over its personal territory. This means that only this user has access to their contents, reserving them for his private purposes. However, when these piles are near the center of the table, their contents are shared among all users (in the group territory). Thus, the storage territories inherit their ownership from the territory they are located on.

The storage territories are located around the surface and may be replicated, if several users need them at the same time; or destroyed, when their contents are no longer needed. Their shape and size depends directly on the amount of content stored on them.

2.3.2 Orientation

Kruger et al. (KRUGER et al., 2003) studied how orientation influences collaboration in deep details, using a collaborative puzzle application. As with traditional tables, when people stand around it they see the content from different angles.

One idea to solve this problem would be to automatically reorient items on the screen towards the user who is manipulating it (SHEN et al., 2004). Another approach would be to reorient items on the table automatically towards the border of the screen or towards the nearest personal or group territory (orthogonality). However, they found that these solutions are overly simplistic and do not care about how people actually use orientation during collaboration. They found that orientation plays three major roles in such contexts:

Comprehension

Comprehension is related to easing the understanding of an object on the screen. Orientation might be used here to ease reading of texts or to provide an alternative perspective of some content.

Coordination

Coordination is related to the overall organization of the objects on the screen. People use orientation to take or give ownership of objects and to define private and group territories. If an object is oriented towards a particular user, he owns it implicitly.

Analogously, if this object is oriented towards a group of users, then this object belongs to that group territory. It is worth to note that users do not need to tell the others if they are releasing an object or taking ownership of it: it is completely understood by the orientation of that object.

Communication

When people orient an object towards themselves, they are implicitly telling the others that they are starting a personal task, with no intention to communicate. On the other hand, if an user orients an object towards another user (or group), then he is intentionally starting a communication to that user (or group) about that object.

Authors suggest that the computer system should support freely oriented objects, besides of being capable of automatically reorient items accordingly to the users or their positions on the screen.

2.3.3 Heuristics and Guidelines

After discussing how interactive tables work and how they are used (mainly for collaboration purposes), we are able to enumerate several heuristics and guidelines for tabletops software development. These heuristics may not only guide the applications development, but also give the foundation to some requirements we identified for multi-touch gestures engines, which will be discussed in further details in Chapter 3.

We start by presenting the most general usability heuristics for systems development, initially proposed by Nielsen (NIELSEN, 1993), which are clearly applicable for tabletop software development (APTED; COLLINS; KAY, 2009):

1. Visibility of system status
2. Match between system and the real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use

8. Aesthetic and minimalist design
9. Help users recognise, diagnose, and recover from errors
10. Help and documentation

Later, a set of heuristics for generic *groupwares* have been proposed (BAKER; GREENBERG; GUTWIN, 2002):

1. Provide the means for intentional and appropriate verbal communication
2. Provide the means for intentional and appropriate gestural communication
3. Provide consequential communication of an individual's embodiment
4. Provide consequential communication of shared artifacts (i.e. artifact feedthrough)
5. Provide protection
6. Manage the transitions between tightly and loosely-coupled collaboration
7. Support people with the coordination of their actions
8. Facilitate finding collaborators and establishing contact

Some of these heuristics are applicable for tabletop interaction, such as the #6 and #7 (APTED; COLLINS; KAY, 2009). A tabletop *groupware* developer or engine must handle such heuristics in order to ease the collaborative process.

The study of usability heuristics for collaborative systems continued with the study published by Scott et al. They suggest some guidelines to enhance the collaboration on tabletops (SCOTT; GRANT; MANDRYK, 2003):

1. Support interpersonal interaction
2. Support fluid transitions between activities
3. Support transitions between personal and group work
4. Support transitions between tabletop collaboration and external work
5. Support the use of physical objects
6. Provide shared access to physical and digital object

7. Consider the appropriate arrangement of users

8. Support simultaneous users actions

In particular, their last recommendation makes a huge impact on how to design tabletop applications and engines, as, differently from other devices, tabletops allow the shared and simultaneous use of its main input method: the touch surface.

Finally, more recent work has brought several heuristics more specific to tabletop softwares (APTED; COLLINS; KAY, 2009):

Design independently of table size

This may require to adapt screen objects accordingly to the table size and resolution.

Support reorientation

This heuristic corroborates to what have been discussed in Section 2.3.2. Although some systems are able to automatically change the orientation of objects according to some criteria, it is mandatory to allow the user to reorient them as they want.

Minimize Human Reach

This heuristic also corroborates to what have been discussed in Section 2.3. Due to its big size, it is possible that an user can not reach an object on the screen. Although that may stimulate the collaboration between users and enhance social aspects of the interaction, it may also be prejudicial to the main activity, stalling or delaying the personal tasks.

Use large selection points

This is directly related to users input. Tabletop interaction is, in general, harmed by the *occlusion problem*: when an user is typing, tapping or dragging an object, his fingers usually hide the object which is being manipulated, harming the interaction.

For smartphones and tablets, an interesting approach has been employed at the Lucid Touch project (WIGDOR et al., 2007): using a back-facing camera, they can draw a lucid shadow of fingers in the main scene. The touch input is, thus, made from two sensor surfaces: one on the back of the main display and another one over it (Figure 2.6).

However, their approach is not suitable for large tabletop surfaces. In these surfaces, their large size must be explored in a way to provide a good feedback on what is being touched by the users.

Manage Interface Clutter and Use Table Space Efficiently

These heuristics are related to the overall organization of screen objects. As in ordinary tables, due to its large size, it is easy to lose control of its content and objects. Applications must allow users to hide or remove screen objects when they are no longer needed.

Support private and group interactions

This is directly related to the main purpose of collaborative softwares and collaborative tables: to support people sharing their ideas through an intuitive and natural interface.

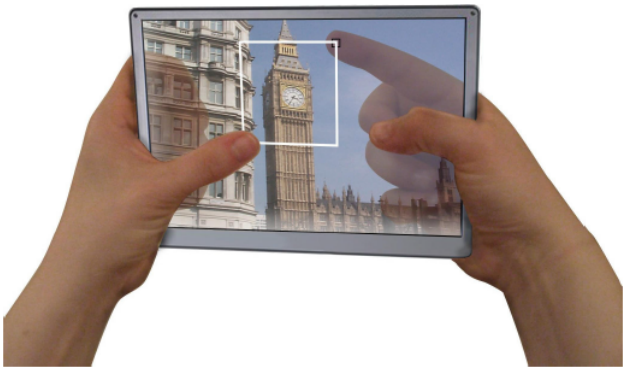

Figure 2.6: Lucid Touch: a conceptual sketch

According to the authors, these heuristics provide a solid foundation to evaluate *groupwares* on tabletops. However, they recognize that some of them may not be suitable to a particular application, such as the free orientation: there might be some applications whose orientation is fixed (e.g. digital desks and drafting tables). It is up to the developers to identify which heuristics are suitable to their applications and ensure they are applied in order to achieve a good collaborative interaction.

In this section we have provided an overview of the most general usability heuristics before specializing them to the context of interactive tables. These heuristics provide a set of guidelines to tabletop developers in order to support good collaborative interactions. In the following section we shall present several applications of tabletops and how these heuristics are used. Also, as shall be discussed in Chapter 3, these heuristics will support the definition of requisites to multi-touch engines and frameworks.

2.4 Applications of Interactive Tables

Most of the work which have been done on interactive tables are collaborative applications. Those applications try to take the most from the naturalness of collaboration these devices

provide. In this section, we will present the most common applications for tabletops, focusing mostly on the collaborative tables. However, this is not intended to be an extensive discussion as the possibilities for tabletop applications are many: from multimodal games (TSE et al., 2007) to biology systems (TABARD et al., 2011).

A tabletop application enhances the physical participation, stimulates reflection and the collaborative affection between the participants (SHAER et al., 2011). Also, collaborative works may be of several types: *planning*, *intellectual*, *creation* and *competition* (MCGRATH, 1984). Examples of these applications are *meetings tools*, *concept mapping*, *brainstorming* and games, respectively.

CMate (Figure 2.7) is a *concept mapping* collaborative application (MALDONADO; KAY; YACEF, 2010), which has been designed following the heuristics and guidelines discussed previously (APTED; COLLINS; KAY, 2009; SCOTT; GRANT; MANDRYK, 2003). Authors showed experimentally that collaborative *concept mapping* promotes better acknowledgement and richer concept maps. However, as their application was based on turns, users could not collaborate simultaneously on the tabletop.

Figure 2.7: CMate: Collaborative Concept Mapping Application

Collaborative *brainstorming* (CLAYPHAN et al., 2011) (Figure 2.8) and *meetings manager* (HUNTER et al., 2011; GEYER et al., 2011; MASOODIAN; McKoy; ROGERS, 2007) applications have also been proposed. The *MemTable*, proposed by Hunter et al. is capable of recording all information generated during the meeting, such as text and speech. In both applications, authors employed physical keyboards to allow the text input.

The text input method is still under discussion in the tabletop interaction community. Despite of existence of some applications which employ physical keyboards and get better results (HARTMANN et al., 2009; HUNTER et al., 2011), there are some studies defending the use of virtual keyboards (KO et al., 2011).

An application for a drafting table has also been proposed (VANDOREN et al., 2008). It

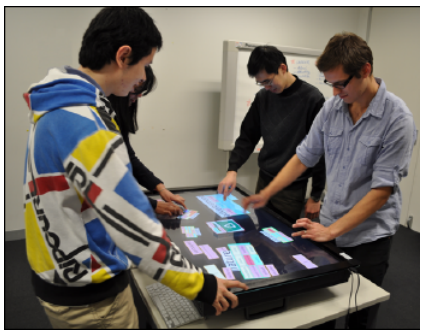

Figure 2.8: Participants of the Firestorm experiments

allows artits to paint a virtual canvas simulating the way they would do to a regular one. The user selects the brush and the color and the system paints where the user touches using the brush.

Despite of the several examples of collaborative applications for tabletop we have found in our literature review, an important concept is barely covered: the *competitiveness* on tabletops (GROSS; FETTER; LIEBSCH, 2008). Developing tabletop applications which explores competitiveness has at least one big challenge to overcome: how to hide information between the competitors? With a single and shared surface, it is very difficult to have privacy. They have found by experimenting that users try to create mimics and eye signals between members of the same team to communicate privately. Another option would be to add other devices to the interaction, such as tablets and smartphones, creating a multi-modal interaction environment.

In this section, we have briefly presented the most common tabletop applications, which are mostly collaborative, and we have reinforced that the competitive aspects of tabletops have been barely covered in recent researches, being an open area for further development.

2.5 Summary

In this chapter we have discussed how tabletops are built, from the hardware and software perspectives. Then, we have discussed what gestures are and how users interact with tabletops using them. We have seen that large tabletops are most often used to support collaborative tasks, although some efforts to support competitiveness on tabletops have also been made.

However, from the software point of view, some questions have not been answered yet, such as:

- How raw input data from sensors, such as TUIO data, is transformed into a meaningful gesture?

- How these data is treated to support multi-touch gestures?
- How to support multi-user (collaborative or competitive) interaction with several different users performing different gestures at the same time?
- How to deal with different orientations at runtime?
- How to support different gestures in different territories, as each territory has its own set of possible actions?

Such questions are discussed in Chapter 3, where we are going to discuss how gestures are interpreted by computer programs, called *gestures recognizers*. Such programs must overcome these challenges in order to support all tabletop interaction concepts we have discussed in this chapter and respect the heuristics and guidelines in order to support the best multi-user tabletop interaction possible.

3 GESTURES RECOGNITION

In this chapter we start the discussion of the main subject of this research: multi-touch gestures recognition. In Chapter 2, we discussed how gestures are used as the main input method for tabletops and how an application may benefit from a large variety of gestures.

A gestures recognizer is a piece of software that is responsible for interpreting the touch input provided by the sensors and for generating a gesture event to the application whenever a valid gesture is performed by the users.

Using the above definition, we can build a general architecture for tabletop applications using three layers (Figure 3.1): the hardware abstraction, which is responsible for getting the touch input from the sensors (e.g. the TUIO protocol API (KALTENBRUNNER et al., 2005)); the gestures recognition, which is responsible for interpreting these data and for generating the gesture event; and the main application, which receives the gesture event and process it according to the application needs.

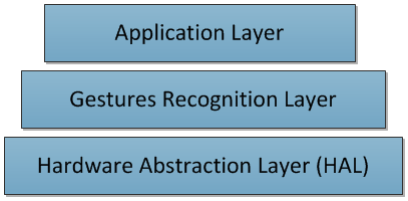

Figure 3.1: A general architecture for tabletop applications

Previous work showed experimentally that gestures depend on interaction and social contexts and that many different gestures could be performed in order to activate a single action (HINRICHS; CARPENDALE, 2011). This shows that the gestures set may be larger than the actions set in a given application. Game Designers would say that there might be more verbs than actions and two or more verbs may be used to perform one action (SCHELL, 2008). Their study provides us two major requirements for gestures recognizers: to be very fast, in order to not harm the interaction; and to be very flexible and accurate, in order to allow a large gestures set to be correctly processed by the applications.

Several authors agree that supporting a new gesture in a given application requires pro-

cessing low-level touch events (raw input events from the sensors device drivers): a tedious, complicated and error-prone task which usually delivers a confused and hard to maintain source code (KIN et al., 2012b; SCHOLLIERS et al., 2011; WOBBROCK; WILSON; LI, 2007). This has been the core motivation for developing extensible and flexible gestures recognizers.

In our previous work (CIRELLI; NAKAMURA, 2014), we presented a survey on multi-touch gestures recognition systems. We found two main approaches to build gestures recognizers: by defining multi-touch gestures formally and, then, using a mathematical or computational method to identify them; or by specifying the gestures by examples and using a pattern recognition technique to identify them among other possible gestures. The former method is discussed in Section 3.1, while the latter is discussed in Section 3.2. We end this chapter in Section 3.3, presenting a more general architecture for tabletop frameworks and the fourteen requirements for gestures recognition systems we identified during our research.

3.1 Formal Gestures Definitions

In this section we discuss different models for multi-touch gestures and how a recognizer would match the raw input data received from the multi-touch sensor to a meaningful gesture according to the proposed model.

Despite of the differences between these models and their respective interpreters, they share a common characteristic: the developers must describe each gesture used in the application using the same formal model. We present two of the most advanced state-of-art gestures recognizers that fall into this category: Proton++ (KIN et al., 2012a) and Midas (SCHOLLIERS et al., 2011). For more examples and a chronological review of such techniques, we direct readers to our survey (CIRELLI; NAKAMURA, 2014).

3.1.1 Defining gestures using regular expressions

Proton (KIN et al., 2012b), and its evolution, Proton++ (KIN et al., 2012a), are gestures recognizers based on the formalism of regular expressions. Regular expressions are easy to interpret using finite automata and many modern languages, such as C# and Java, comes with built-in tools to process them, without the need of a specific compiler.

The idea behind Proton is that a gesture is a sequence of touch events. If we represent each touch event (and its properties) by a symbol, then a gesture can be described by a string of symbols of a given language (in this case, a regular language). Their approach has several

unique features which we shall describe in details in this section.

Proton defines each gesture event as a symbol composed of three properties: the action (touch down, touch move or touch up), the touch identification (id) and the object touched (shape, background or anything). The object touched is obtained from a function that developers must implement and attach to a low-level stream generator. One may have noticed that the action and the touch identification can be obtained directly from the TUIO protocol (KALTENBRUNNER et al., 2005).

A *drag* gesture might be defined according to Proton's notation as: $D_1^s M_1^{s*} U_1^s$, which means that finger 1 touched down a shape followed by a sequence of touch moves on the same shape, ending with a touch up over the same shape.

The Proton's stream generator captures the input data and generates a string of symbols according to the above notation. Then, the recognition process tries to match this string of symbols against each of the regular expressions that defines each gesture.

The system also cares about the possible ambiguity between different gestures. For instance, the *tap* gesture may be defined as $D_1^s U_1^s$, which would also be matched by the above *drag* regular expression. One could fix the *drag* gesture in order to solve this ambiguity, modifying it to $D_1^s M_1^{s+} U_1^s$, however, this might be impossible in many cases. In order to solve ambiguities, the system allows developers to define a score function, which estimates the confidence degree of a decision.

Proton has also other two important features: it comes with a gestures editor inspired in the music tablatures, called *gestures tablature* (Figure 3.2), and the possibility to generate events when any part of the regular expression is matched, which is desirable for direct manipulation gestures.

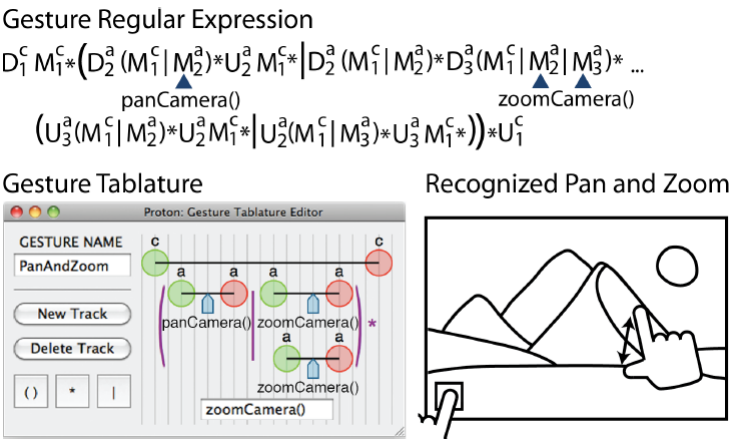

Figure 3.2: The Proton's *gestures tablature* application

Proton has some limitations: it is defined for single-user applications and it only supports those three properties on each symbol. Proton++ allows developers to define their own properties on each symbol (this requires coding a custom property processor) and plug them into the recognition engine, which overcomes the second limitation. However, the stream generator still assumes that only one gesture is being performed at once. This makes the system unsuitable for multi-users applications, but very powerful for single-user ones.

Proton and Proton++ have an additional limitation: regular expressions are, by definition, a sequence of symbols from a fixed set (the language's alphabet), which implies in hard constraints. Using Proton's notation it is hard to describe the complex trajectories found in many symbolic gestures. To illustrate that, suppose that we have added four direction properties to the set of symbols: N, S, E and W. We would have problems if we would like to create a gesture that moves towards NE. Although we could add NE, SE, NW and SW directions, we would fall into the same problem if the trajectory is somewhere between N and NE, for example.

3.1.2 Using logical deduction to recognize gestures

Midas (SCHOLLIERS et al., 2011) is a multi-touch gestures engine which describes gestures as a set of logical rules. These logical rules are processed by a logical inference engine against a set of facts, deciding which gesture has been performed. The main motivation of Midas is to allow the gestures re-usability and extensibility. As gestures are a logical rule, they can be used to build new gestures.

The facts base consists of all current input detected, called *cursor* (the moving finger). Each cursor has the following properties: position (x and y coordinates), speed (x and y coordinates), time and id. Note that all these information are obtained directly from the TUIO protocol (KALTENBRUNNER et al., 2005). The rules base consists of the gestures definitions. An example of a rule to print the X and Y coordinates is given in Listing 3.1 (the *Cursor* predicate is a core fact).

Listing 3.1: Printing X and Y coordinates using a logical rule

```
(defrule PrintCursor(Cursor(x ?x, y ?y))
  =>
(printout t "A cursor is moving at (" ?x "," ?y ")."))
```

They have defined a set of built-in operators (predicates), which are used to build custom

gestures:

Temporal Operators

The built-in set of temporal operators includes predicates to compare two facts regarding their timing data. They can be equal to, near (within a tolerance ϵ) by, contained in, before or after each other.

Spatial Operators

The built-in set of spatial operators includes predicates to calculate the distance between two facts, to check if they are near or inside another fact, regarding their X and Y coordinates.

List Operator

It is common to perform operations against a set of events. Thus, Midas introduced a special operator to group facts into a list of common facts. The list will contain all facts that follow the given constraints, such as time period and id.

Movement Operators

Using the List Operator, it is possible to create rules to obtain the direction of the movement. For instance, *moving up* can be defined as an operation on a given list as all facts $\forall i, j : i < j \wedge list_i(y) > list_j(y)$. *Moving down*, *moving left* and *moving right* are defined analogously and are part of the built-in operators.

One may have noted that the logical inference engine may deduce two or more possible gestures for the same set of facts (inputs). However, Midas solves this ambiguity by allowing developers to define *priorities* for each gesture. When two or more gestures are possible to be deduced from the facts base, Midas will chose the one with the highest priority.

The authors provide the *Flick Left* gesture as an example, which they define as *"an ordered list of cursor events within a small time interval where all events are accelerated to the left"* (Listing 3.2).

Listing 3.2: The Flick Left gesture in Midas notation

```
(defrule FlickLeft
  ?eventList[] <-
    (ListOf (Cursor (same: id) (within: 500) (min: 5)))
  (movingLeft ?eventList)
  =>
  (assert (FlickLeft (events ?eventList))))
```

One may have noticed that authors define *within a small time interval* as a time interval of 500 units of time (one may assume, by the TUIO specification, that it means 500 milliseconds). However, this also adds a hard constraint to the logical definition. Flick gestures will be ignored if they are slightly slower than this maximum time span, the same way as previously described techniques. A future development of the Midas language might address this issue using *fuzzy operators*, defining *vague* rules, such as *a small time interval*. It is also unclear how developers can define complex trajectories by means of the built-in operators and plug them into the proposed recognition engine.

To help developers create custom gestures based on complex trajectories, Hoste et al. have included a *gesture spotting* algorithm before running the Midas recognition process, which is, essentially, a pattern matching algorithm based on a set of control points extracted from examples. This gives developers the ability to create logical rules based on the shape of such gestures (HOSTE; ROOMS; SIGNER, 2013). It would also be possible to define a logical rule based on the result of a template matching algorithm and to deny gestures when they become invalid or if they have been poorly matched.

Midas has also been incorporated to the Mudra multimodal fusion engine, supporting multi-user environments and collaborative gestures (HOSTE; DUMAS; SIGNER, 2011). Finally, Midas declarative approach has been exploited in parallel events processing techniques, providing soft real-time gesture recognition (RENAUX et al., 2012).

Midas is a powerful gestures recognizer, which addresses two important issues seen on other recognizers: modularization (reuse of logical rules) and composition (defining new gestures using previously defined ones), which helps developers to build their custom set of gestures for their applications.

3.2 Defining Gestures by Examples

In this section we are going to discuss a different approach to build multi-touch gestures recognizers. This new approach can be seen as the *dual* of the formalisms one: while the formalism approach defines how users must perform gestures for a given application (users must reproduce the gestures model), *defining gestures by examples* lets users define the gestures they want to use in such application.

This technique is also known as *user-defined gestures* (WOBBROCK; MORRIS; WILSON, 2009) and has got a great attention from the tabletop community since the $1 recognizer was published in 2007 (WOBBROCK; WILSON; LI, 2007). However, recognizing gestures from

examples is an old idea: back to 1991, when touch surfaces were starting to get developed in academic researches and were far away from the mainstream market we have today, Rubine proposed GRANDMA: a model to recognize single-touch gestures from a set of examples (RUBINE, 1991). In this section, we discuss both techniques. For additional techniques and the chronological evolution of user-defined gestures researches, we direct readers to our survey (CIRELLI; NAKAMURA, 2014).

3.2.1 Defining Gestures by Example: the GRANDMA recognizer

In his paper, Rubine presents the GRANDMA (Gestures Recognizers Automated in a Novel Direct Manipulation Architecture) algorithm and is one of the first to use the term *gestures recognizer* to name the part of the software which is responsible to identify the gestures performed by users. As all *user-defined gestures* techniques which have been proposed so far, the recognition is done in two steps: *training* and *classification*. These are the fundamental concepts of any machine learning technique for pattern classification.

GRANDMA models a gesture by its trajectory in the two dimensional plane, which means that each gesture is a single sequence of (x, y, t) points, ignoring points which are less then 3 pixels away from the previous one. Then, it extracts a set of 13 *features* (properties) from a given gesture, which should be able to distinguish different gestures properly:

Sin (f_1) and Cosine (f_2) of the gesture's initial angle:

$$f_1 = cos(\alpha) = \frac{(x_2 - x_0)}{\sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}}$$
$$f_2 = sin(\alpha) = \frac{(y_2 - y_0)}{\sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}}$$

The length (f_3) and the angle (f_4) of the gesture's bounding box diagonal:

$$f_3 = \sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2}$$
$$f_4 = arctg(\frac{y_{max} - y_{min}}{x_{max} - x_{min}})$$

The distance (f_5) between the first and last points:

$$f_5 = \sqrt{(x_{last} - x_0)^2 + (y_{last} - y_0)^2}$$

Sin (f_6) and Cosine (f_7) between the first and last points:

$$f_6 = sin(\beta) = \frac{(y_{last} - y_0)}{f_5}$$
$$f_7 = cos(\beta) = \frac{(x_{last} - x_0)}{f_5}$$

The gesture length (f_8):

Let $\Delta x_p = (x_{p+1} - x_p)$ and $\Delta y_p = (y_{p+1} - y_p)$, then:

$$f_8 = \sum_{p=0}^{p-2} \sqrt{\Delta x_p^2 + \Delta y_p^2}$$

Sum of transversal angles (f_9, f_{10} and f_{11}):

Let $\theta_p = arctg(\frac{\Delta x_p \Delta y_{p-1} - \Delta x_{p-1} \Delta y_p}{\Delta x_p \Delta x_{p-1} + \Delta y_{p-1} \Delta y_p})$, then:

$f_9 = \sum_{p=1}^{p-2} \theta_p$

$f_{10} = \sum_{p=1}^{p-2} |\theta_p|$

$f_{11} = \sum_{p=1}^{p-2} \theta_p^2$

The maximum speed squared (f_{12}):

Let $\Delta t_p = t_{p+1} - t_p$, then:

$f_{12} = max(\frac{\Delta x_p^2 + \Delta y_p^2}{\Delta t_p^2}), p \in [0, p-2]$

The gesture duration (f_{13}):

$f_{13} = t_{p-1} - t_0$

These features can be combined in a 13-dimensional feature vector $\mathbf{f}$ and for each possible classification c, the training process consists on finding the best feature weights (w_c) with the associated bias ($w_{c,0}$) for that class, accordingly to Equation 3.1:

$$\gamma_c = w_{c,0} + \sum_{i=1}^{F} w_{c,i} f_i \tag{3.1}$$

This matrix of weights (a 13-dimensional vector of weights for each class c) is calculated during the training step, from the set of examples provided by the users (or developers), using the classic *Linear Discriminant Analysis* (LDA) technique.

The classification process is as simple as finding the class c which maximizes γ_c from Equation 3.1.

The last step in the GRANDMA's classification process is the rejection of poorly classified gestures. Given a gesture g classified as i among the C classes, the estimated probability of a correct classification is given by Equation 3.2. If the estimated *a posteriori* probability is smaller than 0.95, then the gesture is rejected and discarded.

$$P(i|g) = \frac{1}{\sum_{j=0}^{C-1} exp(\gamma_j - \gamma_i)} \tag{3.2}$$

Rubine also discusses the possibility of rejecting gestures using the Mahalonobis distance, which measures how far a sample is from its respective class mean. However, he understands that this criteria may reject correct classified gestures.

Rubine's classifier may be extended to support multi-touch gestures if one considers each

finger trajectory independently and, then, combines the results to classify the unknown gesture (using a *Decision Tree*, for example). In this case, one would need more features to identify gestures with different number of fingers.

From this very first approach one may have noted that building a gestures recognizer which *learns* from a given set of examples is a challenging task: it is required to understand some machine learning techniques and to find features which allow unknown gestures to be classified properly. However, once one build such recognizer, the effort involved in creating a gesture-based application is drastically reduced. This is the main motivation for further development of *user-defined gestures* recognizers.

3.2.2 A Nearest Neighbours approach

We continue our investigation on *user-defined gestures* recognizers by discussing the $-Family of touch gestures recognizers. This family of recognizers has been developed since the release of the $1 recognizer in 2007 (WOBBROCK; WILSON; LI, 2007). These algorithms share a common approach: they are based on the *K-Nearest Neighbours* (KNN) technique, which consists in defining a *distance* measure between the unknown gesture and those which have been previously added to the system during the *training* step[1].

$1 employs the *Average Euclidean Distance* (also known as *path distance*) as a *distance* measure (Equation 3.3) between a gesture $\mathbf{g}$ in the gestures base and the unknown gesture $\mathbf{t}$. From this distance function, one may derive the $1's similarity measure between these gestures, normalized to the $[0, 1]$ interval (Equation 3.4)[2].

$$d(\mathbf{t}, \mathbf{g}) = \frac{\sum_i^N \|\mathbf{t_i} - \mathbf{g_i}\|}{N} \tag{3.3}$$

$$S(\mathbf{t}, \mathbf{g}) = 1 - \frac{d(\mathbf{t}, \mathbf{g})}{0.5 * \sqrt{size^2 + size^2}} \tag{3.4}$$

Different sensors may sample touch-inputs at different rates, generating a different number of touch-points per trajectory. We have discussed previously that GRANDMA does not care about the sensors capabilities and extracts exact 13 different features from the sequences of (x, y, t) coordinates. $1 introduces a new approach: it re-samples each trajectory in order to have a set of N touch points per sequence of (x, y) coordinates. This process is done for

[1] Some authors do not consider that such recognizers have a proper training step, as there is no parameters tuning process. Most often, this step is just a pre-processing phase.

[2] $size$ is the gesture's bounding box length used in the pre-processing phase

each *sample gesture* during the *pre-processing phase* and repeated for each *unknown gesture* during the *classification phase*. For a given number of points per trajectory N, the algorithm performs following steps (Listing 3.3):

1. Calculate the trajectory length (sums of distances between one point to the next one)
2. Divides the obtained distance by $N - 1$, obtaining the mean distance $\bar{d}$
3. From the first to the last point, it interpolates the trajectory such as the distance between each pair of points is $\bar{d}$.

Listing 3.3: Resampling algorithm proposed by $1

```
public static List<Point> Resample(List<Point> points, int n)
{
  // gets the mean distance between points in the new trajectory
  double I = PathLength(points) / (n - 1);
  double D = 0.0;
  List<Point> oldTrajectory = new List<Point>(points);
  List<Point> newTrajectory = new List<Point>(n);
  newTrajectory.Add(oldTrajectory[0]);

  for (int i = 1; i < oldTrajectory.Count; i++)
  {
    Point pt1 = (Point) oldTrajectory[i - 1];
    Point pt2 = (Point) oldTrajectory[i];

    double d = Distance(pt1, pt2);
    if ((D + d) >= I)
    {
      double qx = pt1.X + ((I - D) / d) * (pt2.X - pt1.X);
      double qy = pt1.Y + ((I - D) / d) * (pt2.Y - pt1.Y);
      Point q = new Point(qx, qy);
      newTrajectory.Add(q);
      oldTrajectory.Insert(i, q);
      D = 0.0;
    }
    else
    {
      D += d;
    }
  }
```

```
  // we may miss the last point due to rounding-errors
  if (newTrajectory.Count == n - 1)
  {
    newTrajectory.Add(oldTrajectory[oldTrajectory.Count - 1]);
  }

  return newTrajectory;
}
```

Then, in order to make gestures invariant to translation and scale, $1 re-scales all gestures to fit into the $[-1, 1]$ square interval and translates their center of mass to $(0, 0)$. These algorithms are presented in Listing 3.4 and Listing 3.5, respectively.

Listing 3.4: Re-scaling Trajectories

```
public static List<Point> ScaleTrajectory(List<Point> points)
{
  double maxX = MaxX(points);
  double minX = MinX(points);
  double maxY = MaxY(points);
  double minY = MinY(points);

  Point fromMax = new Point(maxX, maxY);
  Point fromMin = new Point(minX, minY);
  Point toMax = new Point(1, 1);
  Point toMin = new Point(-1, -1);

  List<Point> newTrajectory = new List<Point>(points.Count);
  for(int i = 0; i < points.Count; i++)
  {
    // Scales each coordinate of p from [min, max] to [-1, 1]
    Point p = Scale(fromMin, fromMax, toMin, toMax, points[i]);
    newTrajectory.Add(p);
  }
  return newTrajectory;
}
```

Listing 3.5: Translating Trajectories

```
public static List<Point> TranslateTrajectory(List<Point> points)
{
```

```
  Point centroid = Centroid(points);
  List<Point> newTrajectory = new List<Point>(points.Count);
  for(int i = 0; i < points.Count; i++)
  {
    Point p = points[i] - centroid;
    newTrajectory.Add(p);
  }
  return newTrajectory;
}
```

In order to achieve rotation invariance, $1 rotates each gesture by its *indicative angle* (the angle between its centroid and the first point) and applies the *Golden-Section Search* approach, which consists in generating new gestures by rotating a gesture from the base by an angle $\theta \in (0, 2\pi)$, step-by-step, while the similarity between the unknown gesture and this rotated gesture grows.

This approach assumes that the distance function has a global minimum on θ and, therefore, the maximum likelihood between an unknown gesture and a gesture from the gestures base can be found iteratively. The gesture from the base that shows the best similarity to the unknown one will be the chosen one.

The $1 recognizer has addressed the single-touch gestures recognition problem uniquely: it has brought an easy-to-implement algorithm (KNN); it has presented a pre-processing step that will be reused in many subsequent approaches and it has defined gestures as trajectories: a sequence of (x, y) points.

However, in order to obtain rotation invariant gestures, it employs a brute force approach which has at least one important drawback: minimizing the distance function on θ requires lots of iterations per gesture in the gestures base, slowing down the classification process. It has also some limitations:

- Due to its pre-processing transformations, $1 can not distinguish gestures that depend on aspect ratio, location or orientation, such as *up-arrows* or *down-arrows*
- $1 can not distinguish gestures that depend on time or speed, such as distinguishing between a *tap* and a *press and hold* gestures

Almost at the same time, in 2010, two extensions of the $1 recognizer were published: the *Protractor* (LI, 2010) and $N (ANTHONY; WOBBROCK, 2010) recognizers. The former presented a closed-form solution to the rotation invariance and the aspect-ratio problems of

$1, improving significantly the time spent to classify an unknown gesture. On the other hand, the latter presented an extension of $1 for multi-touch gestures. In 2012, both solutions were merged into a new recognizer: $N-Protractor (ANTHONY; WOBBROCK, 2012), which we describe now.

We have discussed previously that the *Golden-Section Search* method iterates through the $(0, 2\pi)$ interval until it finds the best angle θ which maximizes the similarity between the rotated gesture from the gestures base and the unknown gesture. *Protractor* introduces the new *Inverse Cosine Distance* (Equation 3.5) function[3] and formulates the problem as finding the angle θ which, if applied to a given gesture $\mathbf{g}$, maximizes the similarity function $S(\mathbf{t}, \mathbf{g})$ between this gesture and the unknown gesture $\mathbf{t}$ (Equation 3.6).

$$d(\mathbf{t}, \mathbf{g}) = arccos(\frac{\mathbf{g} \cdot \mathbf{t}}{\|\mathbf{g}\| \, \|\mathbf{t}\|}) \tag{3.5}$$

$$S(\mathbf{t}, \mathbf{g}) = \frac{1}{d(\mathbf{t}, \mathbf{g})} \tag{3.6}$$

In order to maximize Equation 3.6 one can minimize its denominator, by finding θ which equals its first order derivative to zero (Equation 3.7).

$$\frac{\mathrm{d}}{\mathrm{d}\theta} \left(\frac{\mathbf{g}(\theta) \cdot \mathbf{t}}{\|\mathbf{g}(\theta)\| \, \|\mathbf{t}\|} \right) = 0 \tag{3.7}$$

Solving Equation 3.7 gives us the angle $\theta_{optimal}$ which we need to rotate the gesture $\mathbf{g}$ in order to maximize its similarity to the unknown gesture $\mathbf{t}$ (Equation 3.8). This method significantly outperforms the *Golden-Section Search* method because, instead of iteratively rotating each gesture from the gestures base and testing if θ is optimal, one needs to rotate each gesture from the gestures base by the calculated $\theta_{optimal}$ only once.

$$\theta_{optimal} = arctan(\frac{\sum_i^N \mathbf{g_{x,i}} \mathbf{t_{y,i}} - \mathbf{g_{y,i}} \mathbf{t_{x,i}}}{\|\mathbf{g}\| \, \|\mathbf{t}\|}) \tag{3.8}$$

After its pre-processing phase, $1 models each gesture $\mathbf{g}$ as a sequence of (x, y) coordinates, defining a $(N * 2)$-dimensional vector, where N is the number of touch points per gesture. Extending this model to multi-touch gestures is as easy as considering each finger trajectory as a $(N * 2)$-dimensional vector and concatenating them into a new $(N * F * 2)$-dimensional vector, where F is the number of fingers used to perform the gesture. This

[3] The cosine distance actually finds an angle between two vectors in an n-dimensional space

model works the best if we compare an unknown gesture only to gestures in the gestures base that contains the same number of fingers (which is a reasonable assumption for most of applications).

$N has been well-accepted by the tabletop community and the proposed algorithm has been ported to several different languages, such as C#, Java, Javascript and C++. It is a powerful prototyping tool which allows developers to easily create new gestures and evaluate them in their applications. Authors have also published a web-based tool[4] to help developers to create custom gestures and evaluate the $N algorithm. $N has also been modified by Kratz and Rohs to recognize 3D gestures using accelerometer sensors (KRATZ; ROHS, 2010). However, $N has some limitations[5]:

Fingers Permutations

When users input multi-touch gestures, fingers can be placed into the screen in any order. For instance, a two-fingers gesture (F1 and F2) would generate two different vectors: [F1, F2] and [F2, F1], which would be classified differently. In order to avoid this issue, $N automatically adds all possible fingers permutations to the gestures base. This combinatoric increases the number of gestures in the base, slowing down the classification process and making this process much more error-prone. This issue has been addressed by the $P recognizer (VATAVU; ANTHONY; WOBBROCK, 2012), as we describe later in this section.

Large Gestures Base

$N is good for prototyping because the KNN technique does not require a real *training* process, as opposed to GRANDMA. However, this is also a drawback, because the classification routine must compare the unknown gestures to all gestures (and their fingers permutations) in the base. In real applications, when tens of different gestures (and different numbers of fingers) are available, the classification process suffers a significant loss of speed and, sometimes, precision.

Time-constrained gestures

$N does not use the time information to define gestures. Therefore, it is impossible to distinguish gestures that differ only by their duration or speed, such as *tap* and *press and hold* gestures.

[4] $N tool: `https://depts.washington.edu/aimgroup/proj/dollar/ndollar.html` - accessed at 22nd, April, 2014

[5] $N limitations: `http://depts.washington.edu/aimgroup/proj/dollar/limits/` - accessed at 22nd, April, 2014

The Drag Gesture

$N is not capable of recognizing the *drag* gesture because its trajectory is not defined, since users can freely drag objects around the screen.

Division by Zero

$N's similarity measure (Equation 3.6) has an issue: if we compare identical gestures, it will generate a division-by-zero error. It is unlike to happen in real applications, but for testing purposes, it may be an issue. Fortunately, it is possible to change this function without loss of precision to the classification process (Equation 3.9):

$$S(\mathbf{t}, \mathbf{g}) = \frac{1}{1 + d(\mathbf{t}, \mathbf{g})} \tag{3.9}$$

The last member of the $-Family of touch gestures recognizers is $P (VATAVU; ANTHONY; WOBBROCK, 2012). $P drastically changes the $N gestures model in order to solve the combinatoric overhead issue from its predecessors. While $N defines gestures as an ordered sequence of touch-points (x, y), $P defines gestures as unordered points-clouds (or graphs).

We have seen that $1 iteratively rotates each gesture from the gestures base in order to find the maximum similarity between different gestures. $P needs to do a similar task in order to find which touch-point minimizes its distance to a given point in the unknown gestures cloud. Thus, one needs to calculate the distance between a combination of $N!$ pairs of points in each cloud. This problem has already been solved in the *Graph Theory* and is known as the classic *Assignment Problem* (finding the minimum distance between two bipartite graphs). $P uses an approximation of the *Hungarian Algorithm* (KUHN, 1955) to solve it.

This approach loses the trajectory information and does not care about the number of fingers in each gesture. However, authors claim that such information is not relevant in most applications and only adds more complexity to the recognizer. Authors performed an evaluation study in order to compare $P to $N and $N-Protractor and found out that $P has a slightly better accuracy at the cost of a significant loss of performance ($P takes more than 300ms to classify a gesture while $N-Protractor takes about 0.10ms).

In this section we have covered how $1 and its evolutions have addressed the multi-touch gestures recognition problem from a learning perspective. The $-family has presented lots of contributions to the tabletop community:

- A pre-processing algorithm which deals with variations in translation, scale and sampling
- A closed-form distance function which is fast to calculate and is rotation-invariant

- An easy to implement distance-based classification algorithm
- An extendible model for multi-touch gestures to any number of fingers
- A set of tools for rapid prototyping of gestures-based interfaces

However, time-constrained, free-form or multi-user gestures can not be identified by any of presented techniques from this family. Also, performance may be harmed if these techniques are used in a complex system with many different gestures and several fingers per gesture. Finally, the $-family of algorithms does not include a way to reject poorly classified gestures nor how to provide continuous feedback to the applications while a user is performing a direct manipulation gesture.

3.3 Multi-touch Gestures Frameworks

In this section we describe the state of art multi-touch frameworks. A framework is a set of abstractions, interfaces and standards developed to solve a class of problems in a flexible and extensible manner (GOVONI, 1999). In our context, a framework abstracts multi-touch gestures concepts, allowing developers to create custom gestures and handle their respective events in their applications.

Several multi-touch frameworks were proposed in the past few years, each of them trying to address different developers needs. Although their implementation varies, their most high-level architecture (Figure 3.3) remains almost unchanged, because their base goal are still the same: to link between different input hardware and different graphical toolkits (application level) (ECHTLER; KLINKER, 2008).

The link between these two parts of the system is built on top of a layered architecture, which looses coupling between each part of the system and allows great extensibility and customization. The Hardware Abstraction Layer (HAL) abstracts the sensors hardware and device drivers, defining a common interface for receiving raw touch events. The TUIO protocol has became the *de facto* standard to provide such integration between different frameworks and the input sensors (KALTENBRUNNER et al., 2005), but other implementations have also been proposed, such as the Surface SDK (MICROSOFT, 2012).

On top of the HAL, Echtler and Klinker define a *transformation layer*, which is responsible for converting input data from the sensors to screen coordinates[6]. The *interpretation layer* does

[6]In some frameworks, this step is embedded into the HAL on top of the TUIO API

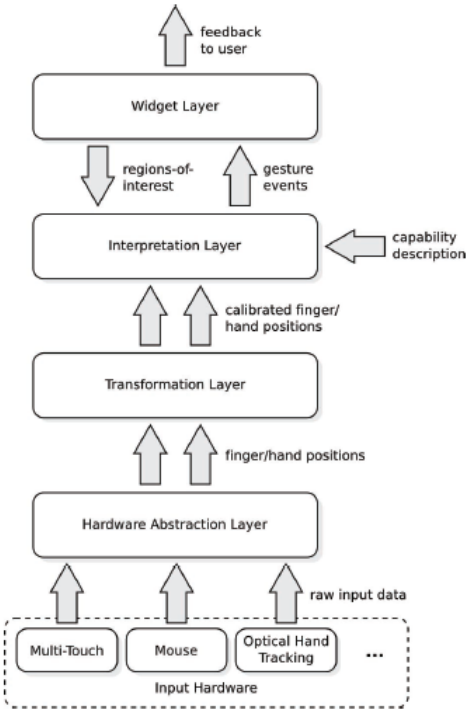

Figure 3.3: Echtler's and Klinker's General Architecture for Tabletop Frameworks

the actual *gestures recognition*, embedding gestures *features* into an *event* object, transmitted to the *widgets layer*, which represents the GUI components (polygonal *regions* where gestures are performed).

It is worth to note that the *gestures recognizer*, which we have discussed previously, is only part of this more complex architecture, which abstracts the gestures handling and its integration to interface objects, allowing developers to easily build new applications on top of it.

Currently there are lots of frameworks with the most recurring multi-touch gestures like *tap* or *drag* built-in. Examples of these frameworks are MT4j (FRAUNHOFER-INSTITUTE, 2011), Surface SDK (MICROSOFT, 2012), Multitouch Vista (CODEPLEX, 2009), Sparsh-UI (SPARSH-UI, 2010), .NET4 (MICROSOFT, 2011), Breeze Multitouch (MINDSTORM, 2010) and Miria (CODEPLEX, 2011), ranging between proprietary and open source. Usually, when a vendor releases a new tabletop, it also provides a custom multi-touch framework, such as the *DiamondSpin Framework* (SHEN et al., 2004) for the *DiamondTouch Table* (DIETZ; LEIGH, 2001) or the reacTIVision (KALTENBRUNNER, 2009) for the reacTable (JORDA et al., 2007).

Kammer *et al* have defined a set of criteria to compare different multi-touch frameworks, such as platform and hardware independences, gesture extensibility, standard gestures and framework's scope. Their criteria are categorized into three groups (Table 3.1): *features*,

scope and *architecture*, which are useful to compare different multi-touch frameworks and find the most suitable for a given application (KAMMER et al., 2010).

Table 3.1: Kammer et al Criteria

Classes	Criteria
Features	Visualization Support Gesture Extensibility Standard Gestures
Scope	Gesture Parameters Tangible Objects Touch
Architecture	Event System Hardware Independence Platform Independence

The *architecture* criteria are concerned with the applicability of a given framework on a given operating system or hardware. Regarding their *scope*, although some frameworks handle touch and tangible objects, authors claim that most frameworks are focused on only one of them. Regarding its *features*, a framework usually provide a set built-in gestures, most often for direct manipulation of interface objects, but also can provide abstractions to allow developers to build their custom gestures. Finally, some frameworks implement the full Echtler's and Klinker's stack, providing a set of abstractions for GUI components integrated to their internal gestures handling system.

Accordingly to the authors, some of the mentioned frameworks are platform independent (MT4j, Miria and Splash-UI) and others are not (Surface SDK). There are also frameworks which are focused on tangible objects (reacTIVision) or on touch (MT4j, Sparsh-UI and Breeze Multitouch). Some frameworks also provides a presentation layer, which contains some widget controls specially developed for multi-touch applications (MT4j, Surface SDK and Breeze Multitouch), implementing the full stack proposed by Echtler and Klinker.

3.4 Requirements for Multi-Touch Frameworks

Kammer's *et al* criteria are mostly focused on the developers needs, by evaluating how a given framework would answer to questions like:

- How to create new gestures?
- How to port my application to different platforms?

- Does it handle touch input or tangible objects?
- Is it integrated to custom GUI components?

However, from the user experience point of view, which we have discussed extensively in Chapter 2, Kammer's *et al* criteria do not give enough information to decide the most suitable framework for a given application. As we have discussed previously, different *gestures recognizers* have their own limitations, which affects users interaction in different ways. Problems such as *orientation invariance*, *recognition performance* and *continuous feedback* are ignored by their set of criteria. Due to this problem, we proposed an extended set of fourteen requirements that a full-featured tabletop framework should meet (CIRELLI; NAKAMURA, 2014). In the following list, we describe each requirement and highlight the heuristic each requirement is addressing.

1. Be Flexible and Extensible

This is the major motivation for all researches presented in this chapter: to allow developers to create their custom gestures without needing to handle low level input data. This requirement supports the *match between system and the real world* heuristic, as it makes it possible to add more intuitive gestures to the applications. Together with requirements 2 and 3, this requirement adds *flexibility and efficiency of use*.

2. Be Fast

Gestures recognition algorithms must be fast in order to not harm the users interaction due to lagging.

3. Be Accurate

Misidentification of gestures may cause several issues to the interaction, as users may lose current work or may lose time undoing unwanted actions. This requirement is related to some of the most basic heuristics, such as *error prevention* and *recognition rather than recall*.

4. Support Multi-Touch and 5. Multi-Users Applications

Tabletops are most often used to support collaborative applications. Thus, handling multi-touch gestures from several users simultaneously is a requirement for recognition systems. This requirement has been designed according to the *support simultaneous users actions* heuristic.

6. Support Orientation Invariance

One of the heuristics discussed previously states that tabletop applications should *support*

reorientation. Thus, most of tabletop applications must not enforce a fixed orientation and participants may interact from any position around the table and rotate objects around the screen.

7. Provide Continuous Feedback

Applications must be continuously notified about gesture events when these gestures are being used to manipulate objects on the screen, such as rotating, moving or resizing them. Such gestures changes the system status continuously. By providing continuous feedback, applications are addressing the *visibility of system status* heuristic.

8. Allow Easy Prototyping

During the prototyping phase of any tabletop application, developers and interaction designers work together in order to run several user experiments to identify which gestures will be used in their applications. This is an iterative process which requires it to be easy to add new gestures to the gestures set and handle their respective events in the application level.

9. Support Symbolic Gestures

Symbolic gestures are often used to execute single-shot actions in tabletop applications, playing the role of command's *shortcuts* or *hotkeys*, which are common in other platforms. These gestures are defined by their trajectory, both in spatial and time coordinates. This requirement also supports the *match between system and the real world*, as symbolic gestures are, in its essence, a gestural representation of a real world object or action.

10. Allow Time-Constrained Gestures

Most of applications uses *tap* or *press and hold* gestures in order to manipulate objects on the screen. These gestures are examples of gestures spatially identical, but very different from the timing perspective.

11. Support Territories

Users perform different actions in different tabletop territories. Therefore, a gestures framework must allow developers to define which gestures are allowed in each territory, avoiding this decision in the application level. This requirement is based on the *support transitions between personal and group work* heuristic.

12. Recognize Free-form Gestures

Gestures such as *drag* and *lasso* can not be described by their trajectories. Therefore,

frameworks must also handle such kind of free-form gestures. Such gestures help to *minimize human reach* and to *design independently of the table size*.

13. Support Sequential Gestures

A well-defined sequence of multi-touch gestures can trigger a new action in the application level (*combos of gestures*). A gestures framework must allow developers to specify these sequences and must notify the application whenever these sequences are detected. This is a special kind of gestures that also helps to *support fluid transitions between activities* and to *support people with the coordination of their actions*.

14. Support Cooperative Gestures

Collaborative applications make use of cooperative gestures, which are gestures performed by different users in order to trigger a single command in the application. As tabletops are often used to support Collaborative Applications, frameworks must support cooperative gestures. This requirement adds *support to private and group interactions*.

Table 3.2 evaluates some of the multi-touch gestures frameworks discussed in this chapter accordingly to the proposed criteria (●: criteria is met; ○: criteria is not met; −: not enough data). Unfortunately, none of these frameworks meets all of the proposed criteria.

Table 3.2: Frameworks Comparison using the proposed Criteria

Criteria	Midas	Proton++	$N-Protractor	MT4J
1	●	●	●	●
2	●	−	●	−
3	−	−	●	−
4	●	●	●	●
5	●	○	○	○
6	○	○	●	○
7	●	●	○	●
8	○	●	●	○
9	●	○	●	○
10	●	●	○	●
11	●	○	○	●
12	●	●	○	●
13	●	●	○	○
14	●	○	○	○

3.5 Summary

In this chapter, we presented the state of art techniques and frameworks for defining and recognizing multi-touch gestures for tabletop applications. We have discussed their strengths and weaknesses based on the requirements we have enumerated in our previous research.

Unfortunately, none of the state of art techniques or frameworks discussed in this chapter meets all of those requirements (Table 3.2). In Chapter 4 we present our research on the MiTable Engine and how it addresses them.

4 THE MITABLE ENGINE

So far, we discussed how tabletops are used, which are their typical applications, how people use gestures to interact with these devices and how a piece of software, called *gestures recognizer*, interprets different inputs from different users.

Now, it is time to put all of these concepts together: our main goal is to support the development of tabletop applications that follows the heuristics and guidelines discussed previously in Chapter 2. Such applications are controlled by gestures, which must be *as natural as possible* and must be well-interpreted by the gestures engine. In order to support and ease such a complex development task, we propose a new multi-touch gestures engine, called *MiTable*, which we discuss in this chapter.

4.1 Introduction

Before presenting the engine itself, we recall the set of requirements from Section 3.3, which guided the development of the *MiTable* Engine, from both the architecture and implementation perspectives:

1. Be Flexible and Extensible
2. Be Fast
3. Be Accurate
4. Support Multi-Touch
5. Support Multi-Users Applications
6. Support Orientation Invariance
7. Provide Continuous Feedback
8. Allow Easy Prototyping

9. Support Symbolic Gestures
10. Allow Time-Constrained Gestures
11. Support Territories
12. Recognize Free-form Gestures
13. Support Sequential Gestures
14. Support Cooperative Gestures

Each requirements is addressed in a different level of the *MiTable* engine. On one hand, requirements such as *Be Fast* and *Be Accurate* are very implementation dependent (although some bad architectural decisions may also harm the engine's performance), on the other, criteria such as *Be Flexible and Extensible* and *Support Multi-Touch and Multi-Users Applications* define architectural requirements.

In the following sections, we discuss the details of the proposed engine. We start by presenting the MiTable's architecture in Section 4.2 and its benefits. The MiTable Engine consist not only of a set of classes and API's for application development, but also includes a set of tools to ease the creation of new gestures and the configuration of different gestures recognizers. Such tools are presented in Section 4.3. Finally, we end this chapter presenting our two proof of concept applications and discuss how the proposed engine addresses each of the fourteen requirements.

4.2 MiTable's Architecture

In this section we present the *MiTable*'s architecture and the its internal behaviours. *MiTable* is a built on top of a 4-layers architecture (Figure 4.1):

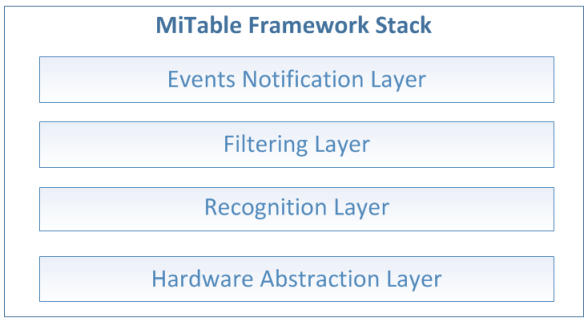

Figure 4.1: MiTable's Architecture

Hardware Abstraction Layer (HAL)

This is the bottom-level layer. Its responsibility is to interface the engine to different hardwares, operating systems and device drivers, abstracting such complexity from the layers above it and the main application.

Recognition Layer

The recognition layer is responsible for interpreting the raw data received from the sensors by the HAL components into a meaningful gesture object. For each gesture interpreted, the recognition algorithm must provide its confidence level about that recognition. The recognition layer is structured in a novel six customizable steps pipeline, which are further described in Section 4.2.2.

Filtering Layer

In this layer, the engine filters the gesture objects according to some criteria, such as the gesture name, the recognizer confidence level or the area (territory) where the gesture has been performed.

Events Notification Layer

This is the top-level layer which is responsible to notify the main application about the recognition of a new gesture.

The most basic structure is the *multi-touch gesture* representation, which consists of an ordered collection of *strokes* (fingers movements), which are an ordered collection of raw touch coordinates (trajectory) as shown in Figure 4.2. Each type of gesture is uniquely identified by its $Name$. When recognized, a *MultiTouchGesture* object receives a $Score$, which is the confidence level of that recognition (ranges between 0 and 1) and the classification $Timestamp$.

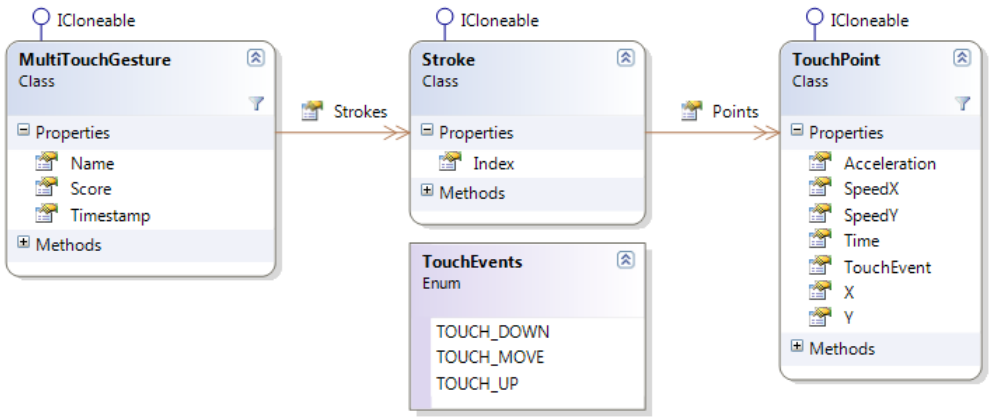

Figure 4.2: MiTable's Gesture Model

Before we go deeper into each layer, it is worth to look at the engine from an intermediate level. Understanding the core components and how they are connected to each other is important before we dig into the implementation details. Figure 4.3 provides a big picture of the engine's internals and the most important components in each layer.

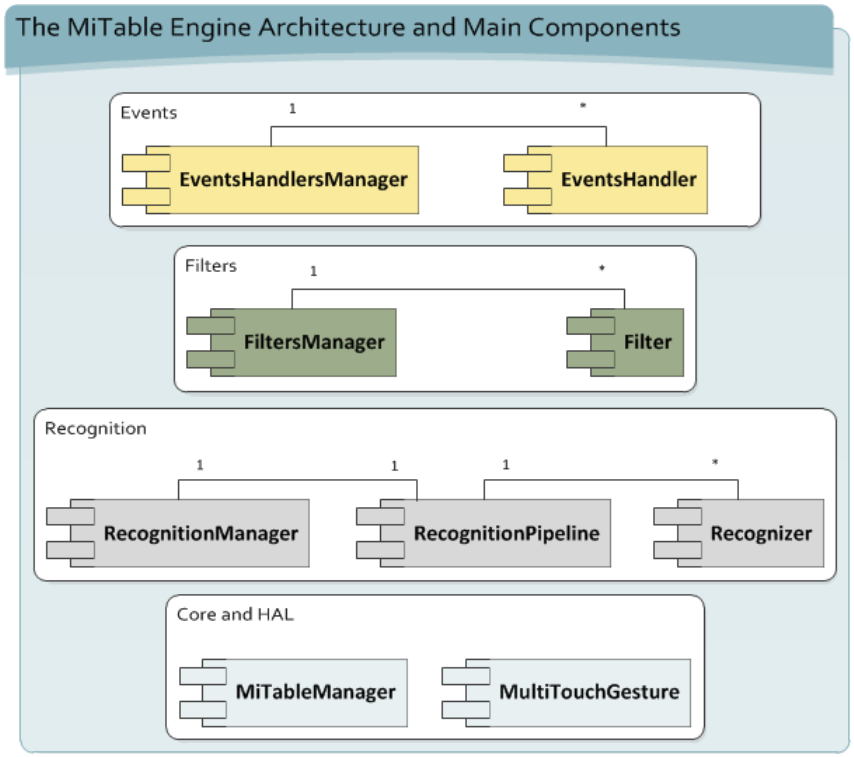

Figure 4.3: MiTable's Main Components

From top to bottom, the *EventsHandlersManager* is responsible for registering the callbacks from the applications for each possible gestures and for calling them accordingly. The *FiltersManager* is responsible for running the filtering pipeline, which rejects a gesture according to a set of developer-defined criteria. The *RecognitionManager* is responsible for managing the recognition pipeline. Finally, the *MiTableManager* is the core engine component, which integrates all layers of the engine and provides an abstraction to the underlying hardware.

As we shall discuss in the following sections, the *MiTable* engine is very flexible and can be customized by developers according to the application's and device's needs. All engine's components are defined by a very high-level interface, allowing developers to create their own implementations. Obviously, the engine comes with several built-in implementations to support the most usual cases out of the box.

4.2.1 The Hardware Abstraction Layer (HAL)

The HAL is the engine's entry point and the *AMiTableManager* class is its core component. It defines the interface to the underlying hardware and device drivers and also manages the communication between all layers (Figure 4.2).

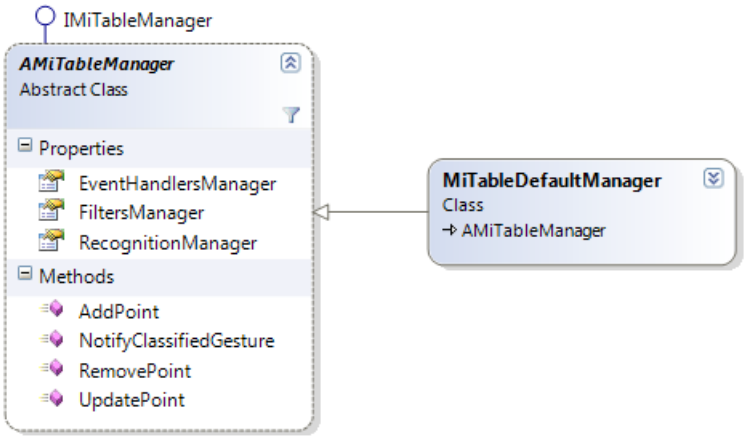

Figure 4.4: MiTable's Manager

AMiTableManager is an abstract class: it implements the engine's dynamics and defines the prototypes for the lower level hardware interface, according to the *Template Pattern* (GAMMA et al., 1998). The prototype functions define the *Data Acquisition Flow* and are called by the platform-dependent hardware interface class. When new touch events are reported by the lower-level hardware interface, this manager redirects the calls to the recognition layer (Figure 4.5).

The platform-specific implementation of this manager must connect to the hardware-specific device driver and call the default *Data Acquisition Flow* functions according to the device's capabilities. The engine has one built-in platform-specific implementation of this manager: the *MiTableTuioAdapter*, which connects the engine to TUIO-supported devices, according to the TUIO Protocol (KALTENBRUNNER et al., 2005).

When the recognition layer identifies a new gesture, it reports back to this manager with an asynchronous function call. The recognized gesture will, then, be filtered and reported to the application. The framework's dynamics is represented in Figure 4.6.

Code provided in Listing 4.1 provides an example of how to initialize the TUIO adapter using the engine's API. It will be further developed during the following sections in order to provide a complete example of how to initialize the *MiTable* engine and start receiving gestures events in the main application.

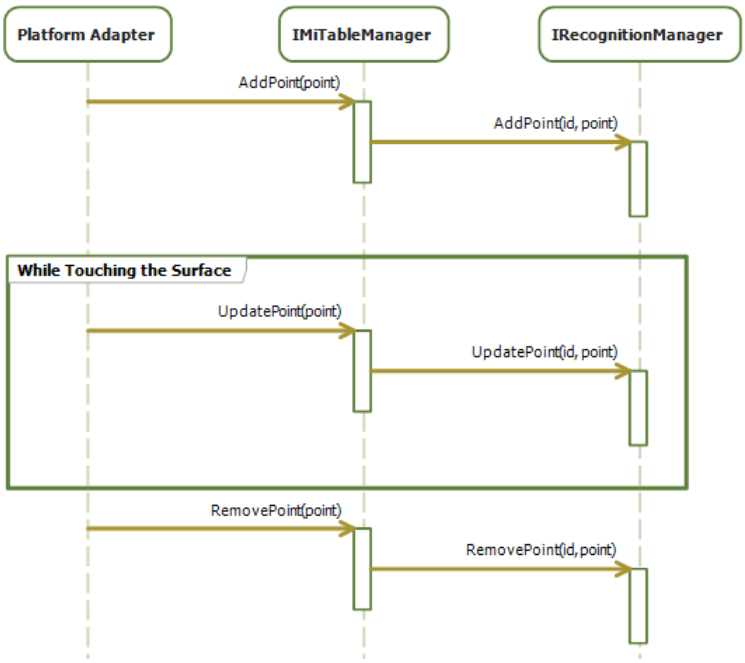

Figure 4.5: Data Acquisition Flow

Listing 4.1: Initializing the TUIO Adapter

```
IMiTableManager miTableManager;

public void InitializeEngine()
{
  // screen resolution (e.g. Full HD)
  int width = 1920;
  int height = 1080;
  // TUIO's socket port
  int tuioPort = 3333;
  miTableManager = new MiTableTuioAdapter(tuioPort, width, height);
}
```

4.2.2 The Recognition Layer

The *Recognition Layer* provides the infrastructure developers need to add standard and custom gestures to their applications. While some frameworks discussed earlier in Chapter 3 provide a single recognition algorithm (SCHOLLIERS et al., 2011; KIN et al., 2012a), others provide a generic interface for developers implement their own recognition algorithm (FRAUNHOFER-INSTITUTE, 2011). There are pros and cons on both approaches. On one hand, having a single recognition algorithm, developers are limited by the algorithms capabilities and limitations, but everything

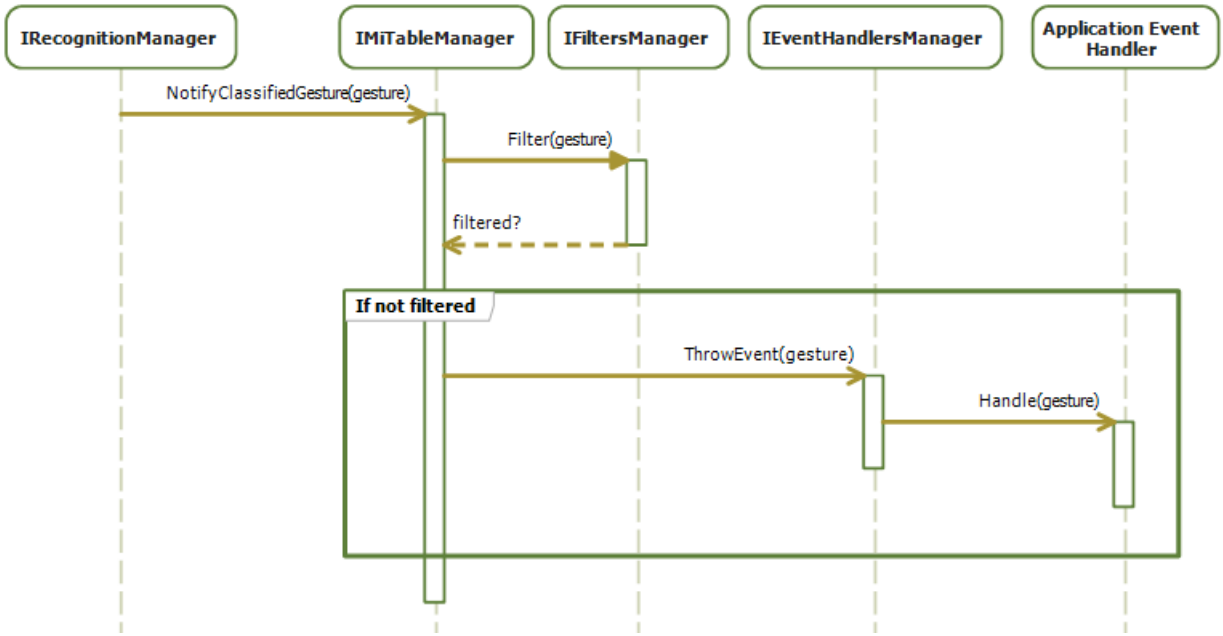

Figure 4.6: Processing Flow

is ready to use out of the box. On the other, by defining a generic interface for recognizers, developers have flexibility to develop their own algorithms, but it is not a trivial task and code-reuse is almost impossible.

From our research on multi-touch gestures recognizers, presented in Chapter 3, we conclude that a gestures recognizer tries to answer to the following questions:

1. **When does a trajectory start and when does it end?**

   This question is related to a problem in computer vision known as *Segmentation*. The goal of segmentation algorithms is to determine the boundaries of the object in study in a image (MITRA; ACHARYA, 2007), removing the parts that are not important for the application. By analogy, in multi-touch gestures recognition, we can define the *Segmentation* process by the task of identifying when a trajectory starts and finishes, filtering possible noisy or corrupted data.

2. **Which fingers belongs to each gesture (or each user)?**

   This question is related to scenarios where it is possible to perform different gestures at the same time (usually, but not restricted to, multi-user applications). When only one gesture can be performed at the same time, it is clear that all fingers touching the screen belongs to that gesture. This is a common scenario for smartphones applications, in which a single user performs just one action at a time. However, in multi-user applications, users are interacting with the device simultaneously and gestures are performed

concurrently. Thus, there is the need to identify which fingers belongs to each gesture being performed.

3. Which gesture is being (or has been) performed?

This is the main purpose of all gestures recognition techniques discussed in Chapter 3: to identify the gesture performed correctly and take its respective action.

4. How to combine gestures to identify sequential or collaborative gestures?

We have seen in Chapter 2 that gestures can be performed in a sequence to generate an additive effect or to trigger a completely new event. These gestures are called sequential gestures. Gestures can be performed by different users in combination to have a different effect, which are the definition of collaborative gestures. In both cases, the resultant gesture is the combination of previously identified gestures.

5. What to do when more than one gesture is possible?

This is the ambiguity problem discussed in Chapter 3, which consists in deciding for a single gesture when more than one classification is possible. This problem happens in two different scenarios: when we have more than one gestures recognition algorithms running in parallel and they provide different answers to the same gesture data; or when a gesture is part of a sequential or collaborative gesture at the same time it is a valid gesture alone (e.g *tap* is a valid gesture while it is part of a *double-tap* gesture). While the former is usually addressed by a *dis-ambiguity function* (KIN et al., 2012a), the latter is usually addressed by a prioritization function (SCHOLLIERS et al., 2011).

When developing the MiTable Engine, we opted for breaking down the recognition process into a six-steps pipeline (Figure 4.7). This novel architecture allows developers to reuse existing pipeline blocks while providing a generic interface for custom algorithms. The Pipeline Pattern (VERMEULEN; BEGED-DOV; THOMPSON, 1995) also allows easy parallelism (to improve performance) and the customization of the recognition process by skipping or adding custom blocks.

Each pipeline block is responsible for answering to one of the aforementioned questions and is defined by a generic interface (Figure 4.8). As fingers touch the screen in a unpredictable manner, the recognition process is intrinsically asynchronous. Thus, the recognition pipeline runs in parallel threads inside the engine and, when the whole process is finished for a set of input data, an event is generated to notify the engine about the captured gesture.

Now, we start a discussion on each of these blocks, explaining how it works and the algorithms available for use in the engine out of the box. As mentioned earlier, one of the

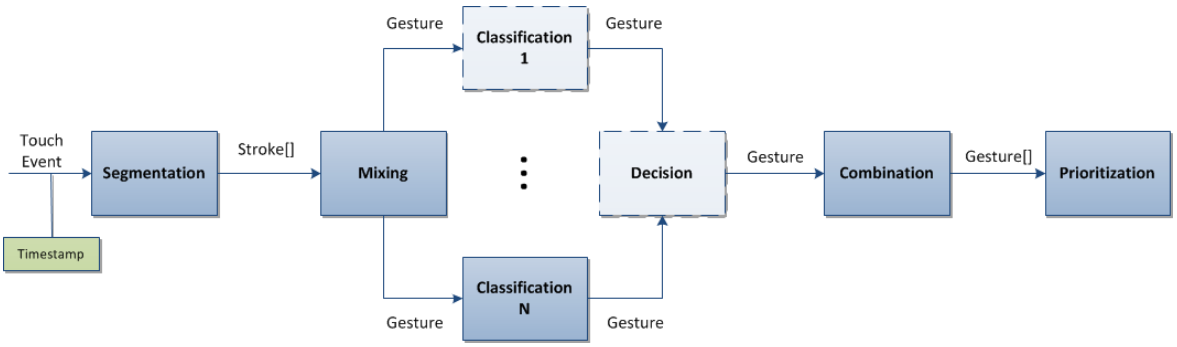

Figure 4.7: Recognition Layer

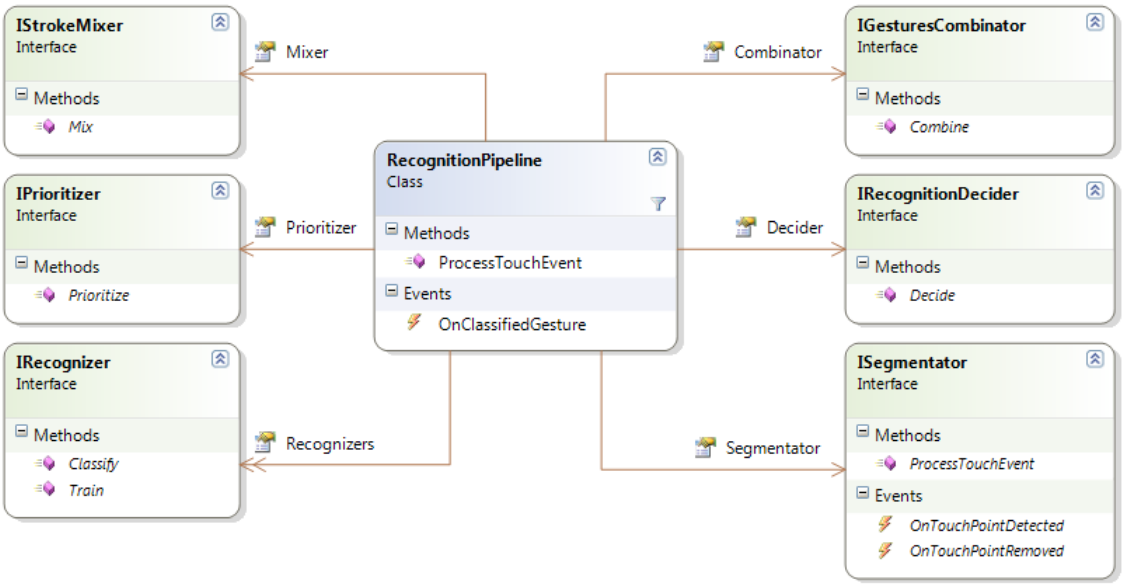

Figure 4.8: Recognition Pipeline Blocks

benefits of the pipeline pattern is that each block is customizable to fit an specific need.

4.2.2.1 The Segmentation Block

The first block in the MiTable Recognition Pipeline is the Segmentation Block. This block is responsible for identifying valid and independent trajectories from the raw touch events provided by the HAL. It receives the touch events from the HAL and organizes them into *Strokes*, which is our representation of a single moving finger and its trajectory.

Its most challenging task is to decide when to transmit the segmented *Stroke* object to the Mixing block. An example of a segmentation algorithm is provided by the GRANDMA recognizer, discussed earlier in Chapter 3. It ignores touch points that are too close to each other, to improve performance and remove noisy data (RUBINE, 1991). Another example of a segmentation algorithm was proposed for 3D gestures based on accelerometers (PREKOPCSAK,

2008). This is very application-specific and, sometimes, is tight-coupled to the recognition algorithm itself.

In order to provide continuous feedback, the Segmentation algorithm needs to return the identified *Strokes* continuously (e.g. by periodic batches or synchronized to the raw touch events). Once a *touch up* event is received from the HAL, the last *Stroke* information is returned and the object can be discarded to free up memory.

The MiTable Engine comes with some common implementation of Segmentation algorithms:

The OnRemoveStrokeSegmentator class

This class implements the basic segmentation algorithm for applications that make use of single-touch symbolic gestures only. A symbolic gesture is defined as a trajectory and it is identified when the trajectory is completed. Thus, whenever a *touch up* event happens, this segmentation algorithm returns the whole *Stroke* information at once.

The GestureRecorderSegmentator class

This class implements a segmentation algorithms which consists of tracking the *Stroke* information whenever the application requests to start recording touch gestures. It tracks raw touch events until the application requests it to stop. This implementation is very similar to the one used in the Nintendo WiiMote controller, in which the user holds a button pressed while performing a gesture.

The PeriodicSegmentator class

This class implements a segmentation algorithm that keeps tracking all raw touch events continuously and returns the set of identified *Strokes* periodically. This is useful to provide continuous feedback to applications that rely on direct manipulation gestures and to avoid an excessive pipeline process when lots of fingers are touching the screen simultaneously as the raw touch information can be passed through the pipeline in periodic batches.

The AllowedAreaSegmentator class

This class implements a segmentation algorithm that tracks only the raw touch events that happens inside a set of allowed areas in the screen. This is a useful algorithm to avoid unnecessary pipeline process in GUI-based applications where gestures can only be performed on the visible widgets or in specific territories.

4.2.2.2 The Mixing Block

The second block in the MiTable Recognition Pipeline is the Mixing Block. This block is responsible for grouping the set of *Strokes* provided by the Segmentation Block into a new set of one or more gesture objects. In smartphone applications, the Mixing block can be very simple: it may just map all provided *Strokes* into the same multi-touch gesture object. When the application uses only single-touch gestures, the Mixing algorithm is also simple: it just returns one gesture object for each *Stroke* performed. However, in multi-user applications with multi-touch gestures this problem is very challenging.

When the hardware is capable of identifying which fingers belongs to each user, the Mixing algorithm is also simple: it returns one gesture object containing the *Strokes* performed by each user. However, this is not the case for several multi-touch surfaces. Usually, there is no information about which user is touching the screen at a given time, nor the location of each user around the tabletop.

Due to these hardware limitations and the different applications requirements, the Mixing block is very dependent on the platform capabilities and the application design. We provide a set of four common implementation of Mixing algorithms, but developers can make use of the Pipeline Pattern to create their own implementation that is more suitable for their devices and applications:

The AllInOneGestureMixer class

This class implements the simple algorithm for single-user applications: it maps all *Strokes* to the same gesture object.

The SingleTouchMixer class

This class implements the simple algorithm for applications based only on single-touch gestures: each *Stroke* is mapped to a different gesture object.

The MaxDistanceMixer class

This class implements a Mixing algorithm that consists in grouping near *Strokes* into gestures objects. The threshold distance is provided by the developer and all *Strokes* within that radius are mapped to the same gesture object. This algorithm is based on the assumption that fingers from the same gesture cannot be far away from each other due to the limited reach of the users hands.

The RectangleAreaMixer class

This class implements a Mixing algorithm that consists in grouping *Strokes* that are

performed within an area on the screen. The set of areas where gestures are allowed to be performed is given and may be changed by the application at runtime. This algorithm is useful for GUI and territory-based applications which define the regions on the surface where users are allowed to interact with.

4.2.2.3 The Classification Block

The third block in the MiTable Recognition Pipeline is the Classification Block. Since the gesture is well-defined, we are able to run the classification process. This block receives the unknown (but segmented and well-defined) gesture and execute its recognition algorithm, regardless of being based on a machine-learning algorithm or on a formal gestures model. The goal of this algorithm is to fill the unknown gesture's $Name$, $Score$ and $Timestamp$ properties.

We have discussed several different classification techniques in Chapter 3. There are those based on machine learning principles and those based on formal models. In order to accomplish both kinds of gestures recognizers, the Classification Block is defined by just a single $Classify(gesture)$ method, which can be implemented regardless of the underlying classification technique used.

As several machine learning techniques have been proposed so far, the engine's API provides a set of classes to support the development of new classifiers. All machine learning-based approaches we discussed previously follow the same recognition flow, such as the $N-Protractor (ANTHONY; WOBBROCK, 2012) and the GRANDMA (RUBINE, 1991) recognizers. There are basically three steps: the *pre-processing*, where the unknown gesture data is normalized and noise is filtered; the *features-extraction*, where the characteristics that are important to distinguish one gesture from the others are obtained; and the *classification* process, that runs the pattern recognition algorithm itself (Figure 4.9).

Figure 4.9: General Classification Pipeline

Besides the Classification Block itself, represented by the *IRecognizer* interface, the engine's API provides abstractions for both the pre-processing and features-extraction steps, which can be used by custom classification blocks.

Lots of pre-processing algorithms are provided out of the box within the engine's API,

such as Catmull-Rom Spline, Re-sampling, Translation Normalization, Scale Normalization, Douglas-Peucker Polygon Simplification (DOUGLAS; PEUCKER, 1973; XIAO-LI; DE, 2010), among others (Figure 4.10).

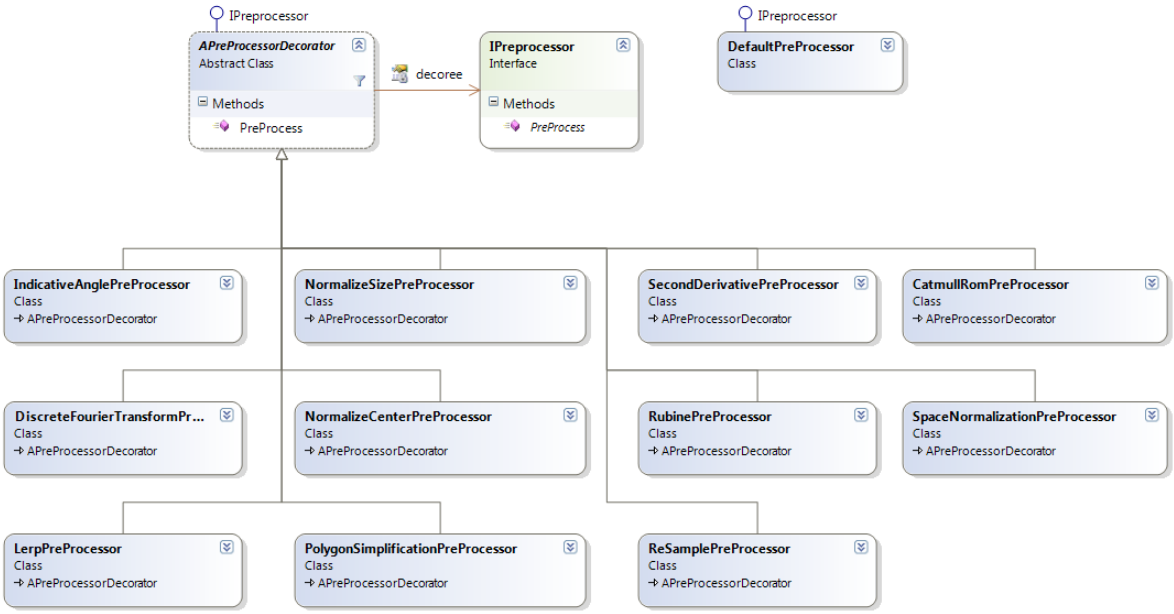

Figure 4.10: Built-in Pre-Processors

Sometimes, the pre-processing phase consists in several steps. For instance, the $N recognizer first re-samples the gestures to a certain number of touch-points. Then it rescales and translates it in order to make it scale and translation invariant (ANTHONY; WOBBROCK, 2012). Each step may be performed by a different pre-processing class and they can be combined using the *Decorator Pattern* (GAMMA et al., 1998).

The same structure is used to the features-extraction interfaces. A features extractor may be designed to extract only one feature and several different extractors may be combined using the Decorator Pattern (Figure 4.11).

Using the *Decorator Pattern* to extract features and pre-process the raw gestures enhances code reuse and allows developers to compare the performance between recognition algorithms with the same set of features and same normalization processes easily.

For multi-touch gestures recognition, our engine provides some of the state-of-art techniques built-in: the $1-Protractor and $N-Protractor; a generic K-Nearest Neighbours algorithm (using configurable distance measures); a Dynamic Time Warping (DTW) based method (SALVADORE; CHAN, 2004); and a Finite-State Machine to recognize the most common direct manipulation gestures (such as *drag*, *drop*, *tap*, *press* and *flick*), which is, in its essence, a recognition algorithm based on the formalism of Finite Automata and Regular Expressions.

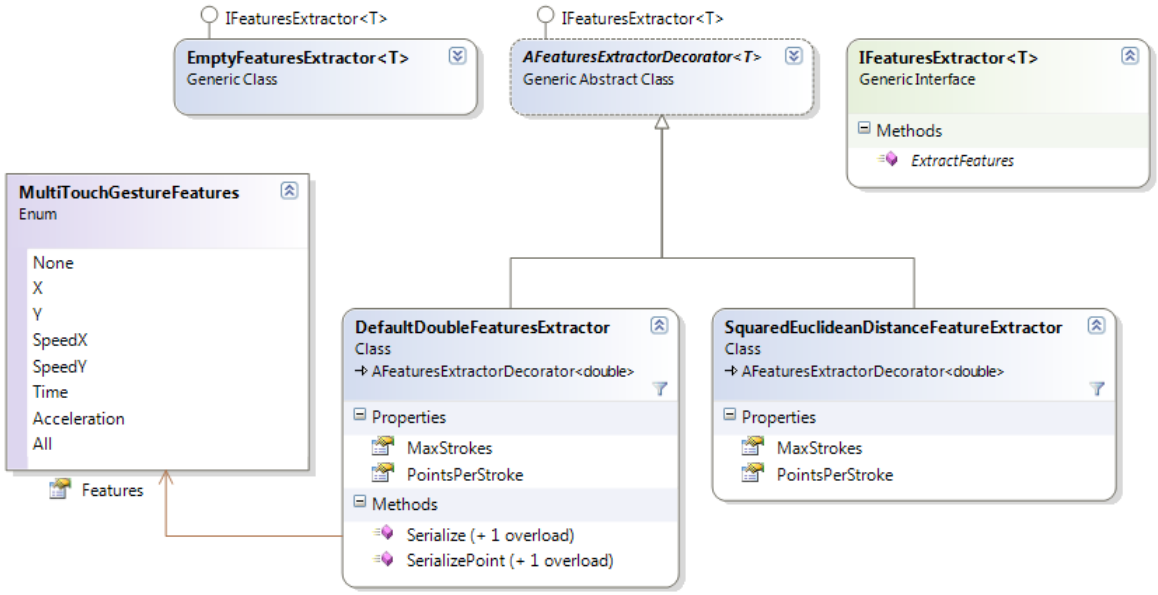

Figure 4.11: Built-in Features Extractors

4.2.2.4 The Decision Block

The fourth block in the MiTable Recognition Pipeline is the Decision Block. As we discussed in Chapter 3, there are several different techniques to recognize gestures, each of them with its own strengths and weaknesses. Sometimes, an algorithm is very specialized for a single type of gesture, performing just a binary decision: the gesture is c or is not. In others, the recognition has some degree of confidence that a gesture is c. In such cases, there might be one algorithm for each possible gesture.

As one may have noticed in Figure 4.7, the recognition pipeline allows the addition of more than one Classification Block to the pipeline. Each of them runs in parallel to each other and provides a different gestures classification.

In order to solve this decision problem, we added a Decision Block to the pipeline. In the simple case where there is just one Classification Block, the decision algorithm just returns recognized gesture. However, in the general case, there might be more than one Classification Block in the pipeline and a decision must be taken.

Taking this decision in the general case is not trivial. Different recognition algorithms estimate the confidence level of their classification in different ways. Thus, using just the *Score* property of the classified gesture is usually not enough. In the general case, when we have different machine learning algorithms in the pipeline, this is worse than insufficient: it is statistically wrong.

There are basically two different scenarios for such decision making algorithms: the different classification blocks have the same gesture set or they are specialized for different gestures. In the former case, a voting algorithm could be applied to have the decision taken. Another possibility would be to apply a statistical test, such as the Cohen's Kappa Test, to decide if the classification algorithms agree to a common decision. However, in the latter case, things are even more complicated, because the universe of possible gestures is not the same for each classifier.

Due to these technical challenges, current engine provides just a default Decision Block, which returns the first classified gesture in the given set (which is fine for pipelines with just one recognition block). The implementation of more complex decision algorithms between different classifiers is a subject for future researches.

4.2.2.5 The Combination Block

The fifth block in the MiTable Recognition Pipeline is the Combination Block. This block allows the combination of gestures into a new gesture. As gestures events are triggered to the application layer, the application would be able to test the combination of different gestures into new ones. However, as this is a very common scenario we decided that it is worth to add a specific block for this purpose into the pipeline. The benefit of doing this computation inside the pipeline is that the gestures can be prioritized and filtered before being handled in the application layer and it also enhances the code-reuse.

We have basically two kinds of gestures that are identified by the combination of other gestures: the sequential gestures and the collaborative gestures. As an example, let's consider the *double tap* gesture. This gesture is defined as two *tap* gestures performed within a given timespan. After the classification, the Decision Block would report a *tap* gesture followed by another *tap* gesture. It is the responsibility of the Combination Block to create the new *double tap* gesture based on the sequence of two *tap* gestures. This is usually implemented with Finite Automata and State Machines, but could also be implemented using machine learning techniques, such as Hidden Markov Models and Decision Trees.

The Combination Block is very application specific and depends on the gestures set and its definitions. The default implementation of this block in the MiTable Recognition Pipeline returns the same gesture received by the Decision Block. This is due to the fact that generalizing the combination algorithm for any application (or, at least, for a set of common use-cases) is not trivial. One possibility would be to implement a scripts interpreter that receives the gestures from the Decision Block and runs an application-dependent script engine to try to

identify if there is a valid combination of gestures.

However, as writing this script is a task to the application developers, it is not much different than implementing a custom Combination Block class. For this reason, we have just the default Combination Block implementation and leaves it to the developers to customize it according to their needs using the provided pipeline block interfaces. A more generic implementation is a subject for future researches.

4.2.2.6 The Prioritization Block

The sixth and last block in the MiTable Recognition Pipeline is the Prioritization Block. Let's consider the *tap* and *double tap* case again. When the Combination Block identifies a *double tap* it returns the new *double tap* and the last *tap* gesture, because they are both possible gestures and the Combination Block does not have the responsibility to decide for one of them.

It is the responsibility of the Prioritization Block to decide which gesture event should be sent to the Filtering Layer. In most of applications, in this case, the *double tap* gesture is more valuable to the application than the *tap* gesture. Thus, the *double tap* gesture has a higher priority than the *tap* one, which is discarded.

The purpose of the Prioritization Block is to solve ambiguities when more than one gesture is equally possible (e.g. a second *tap* and a *double tap*). The default Prioritization Block available in the pipeline implements the algorithm used in the Midas (SCHOLLIERS et al., 2011) recognizer: for each gesture, developers may assign a priority. When more than one gesture is equally possible, the default Prioritization Block returns the one with the highest priority. As any other block in the pipeline, this behaviour can be customized according to the developers needs.

Listing 4.2 includes the *Recognition Layer* configuration code for our hypothetical example:

Listing 4.2: Configuring the Recognition Algorithm

```
IMiTableManager miTableManager;
// We set it as a global variable so we can
// add/remove areas as GUI widgets are created
RectangleAreaMixer mixer;
PeriodicSegmentator segmentator;
// The Standard Gestures Classifier
// for drag, drop, tap, press, flick, zoom and rotate gestures
StandardGesturesRecognitionMachine machine;

public void InitializeEngine()
{
  // screen resolution (e.g. Full HD)
  int width = 1920;
  int height = 1080;
  // TUIO's socket port
  int tuioPort = 3333;
  miTableManager = new MiTableTuioAdapter(tuioPort, width, height);

  // Creates the Segmentation Block
  TimerComponent timer = new TimerComponent(this);
  PeriodicSegmentator segmentator = new PeriodicSegmentator(timer);
    // Creates the Mixing Block
    mixer = new RectangleAreaMixer();
    // Creates the Classification Block
    StandardGesturesRecognizer classifier = new
        StandardGesturesRecognizer();
    // Builds the Recognition Pipeline using default Decision,
    // Combination and Prioritization Blocks
    RecognitionPipeline pipeline = new RecognitionPipeline(segmentator,
        mixer, classifier);
    // Attaches the Pipeline to the Engine
    miTableManager.RecognitionManager.AddPipeline(pipeline);
    // Starts the periodic segmentation task
    segmentator.Start();
}
```

4.2.3 The Filtering Layer

The *Filtering Layer* is responsible for rejecting recognized gestures based on any kind of criteria. For instance, developers can program the framework to reject poorly classified gestures (low recognition confidence) or reject gestures which are not allowed to be performed in current application state. The filtering process is handled by the *Filters Manager*, which can handle any number of *filters* (Figure 4.12).

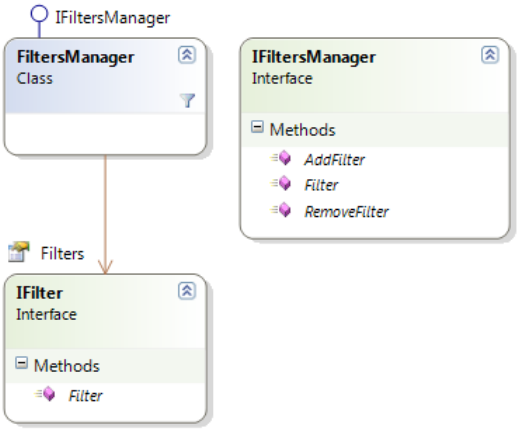

Figure 4.12: Filtering Layer

MiTable comes with four built-in filters (Figure 4.13). The $NameFilter$ class, which filters a given set of gestures, and its special class ($StandardGesturesFilter$), which filters the most common direct manipulation gestures; the $ScoreFilter$ class, which rejects poorly classified gestures based on a pre-defined minimum confidence level; and the $AllowedAreasFilter$ class, which rejects gestures performed on a set of areas of the screen (allowing developers to filter gestures in disabled GUI components, for example).

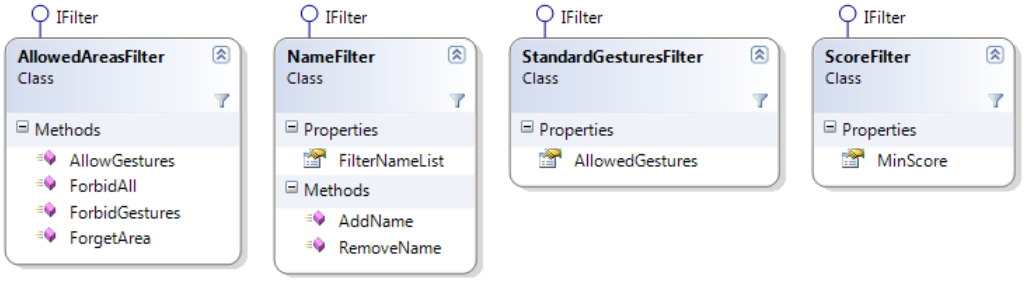

Figure 4.13: Built-in filter classes

The filtering process is also pipelined. The *Filters Manager* acts as the Pipeline Manager, each *filter* class is a *pipeline block or handler* and the recognized gesture is the *pipeline command* (VERMEULEN; BEGED-DOV; THOMPSON, 1995). Listing 4.3 uses the *AllowedAreasFilter* class to reject gestures performed out of GUI components bounds.

Listing 4.3: Configuring the Recognition Algorithm

```
IMiTableManager miTableManager;
// We set it as a global variable so we can
// add/remove areas as GUI widgets are created
RectangleAreaMixer mixer;
PeriodicSegmentator segmentator;
// The Standard Gestures Classifier
// for drag, drop, tap, press, flick, zoom and rotate gestures
StandardGesturesRecognitionMachine machine;

public void InitializeEngine()
{
  // screen resolution (e.g. Full HD)
  int width = 1920;
  int height = 1080;
  // TUIO's socket port
  int tuioPort = 3333;
  miTableManager = new MiTableTuioAdapter(tuioPort, width, height);

  // Creates the Segmentation Block
  TimerComponent timer = new TimerComponent(this);
  PeriodicSegmentator segmentator = new PeriodicSegmentator(timer);
    // Creates the Mixing Block
    mixer = new RectangleAreaMixer();
    // Creates the Classification Block
    StandardGesturesRecognizer classifier = new
        StandardGesturesRecognizer();
    // Builds the Recognition Pipeline using default Decision,
    // Combination and Prioritization Blocks
    RecognitionPipeline pipeline = new RecognitionPipeline(segmentator,
        mixer, classifier);
    // Attaches the Pipeline to the Engine
    miTableManager.RecognitionManager.AddPipeline(pipeline);
    // Starts the periodic segmentation task
    segmentator.Start();

    // Create the filter
  filter = new AllowedAreasFilter();
  miTableManager.FiltersManager.AddFilter(filter);
}
```

4.2.4 The Events Notification Layer

The last step in the *Processing Flow* is the events notification process (Figure 4.14). The top-level application registers event handlers for each gesture, using the *Events Handlers Manager* methods. Then, when a gesture is processed by the previous layers, it is sent to the specific application handler. This structure is based on the *Observer-Observable* Design Pattern: the gesture event is the observable object and the application's handlers are the observer objects (GAMMA et al., 1998).

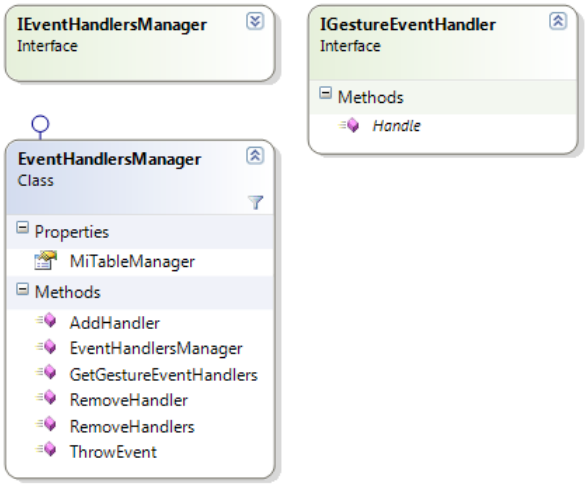

Figure 4.14: Events Notification Layer

The entry point for receiving gesture events from the application is the $Handle(gesture)$ method. Application can define a single handler for all gestures or a specific handler object for each kind of gesture. Listing 4.4 shows how to attach an application event handler to handle *drop* gesture events.

Listing 4.4: Configuring the Recognition Algorithm

```
IMiTableManager miTableManager;
// We set it as a global variable so we can
// add/remove areas as GUI widgets are created
RectangleAreaMixer mixer;
PeriodicSegmentator segmentator;
// The Standard Gestures Classifier
// for drag, drop, tap, press, flick, zoom and rotate gestures
StandardGesturesRecognitionMachine machine;

public void InitializeEngine()
{
  // screen resolution (e.g. Full HD)
  int width = 1920;
  int height = 1080;
  // TUIO's socket port
  int tuioPort = 3333;
  miTableManager = new MiTableTuioAdapter(tuioPort, width, height);

  // Creates the Segmentation Block
  TimerComponent timer = new TimerComponent(this);
  PeriodicSegmentator segmentator = new PeriodicSegmentator(timer);
    // Creates the Mixing Block
    mixer = new RectangleAreaMixer();
    // Creates the Classification Block
    StandardGesturesRecognizer classifier = new
        StandardGesturesRecognizer();
    // Builds the Recognition Pipeline using default Decision,
    // Combination and Prioritization Blocks
    RecognitionPipeline pipeline = new RecognitionPipeline(segmentator,
        mixer, classifier);
    // Attaches the Pipeline to the Engine
    miTableManager.RecognitionManager.AddPipeline(pipeline);
    // Starts the periodic segmentation task
    segmentator.Start();
    // Create the filter
  filter = new AllowedAreasFilter();
  miTableManager.FiltersManager.AddFilter(filter);
  // Drop Event Handler
  miTableManager.EventHandlersManager.AddHandler(this,
      StandardGesturesNames.DROP_GESTURE);
}
```

Passing the recognized gesture object to the application handler gives application developers lots of flexibility to obtain the information they need from the recognized gesture, such as speed, initial or last position and bounding area. In order to ease the developers task, the engine comes with a set of classes to extract common information from a given gesture.

Such classes extracts information from the gestures which is relevant to the application context. For each gesture, a different set of data can be extracted using the built-in classes. As always, there is a generic data extraction interface which can be used by developers to create their own data extraction algorithms. This API is shown in Figure 4.15.

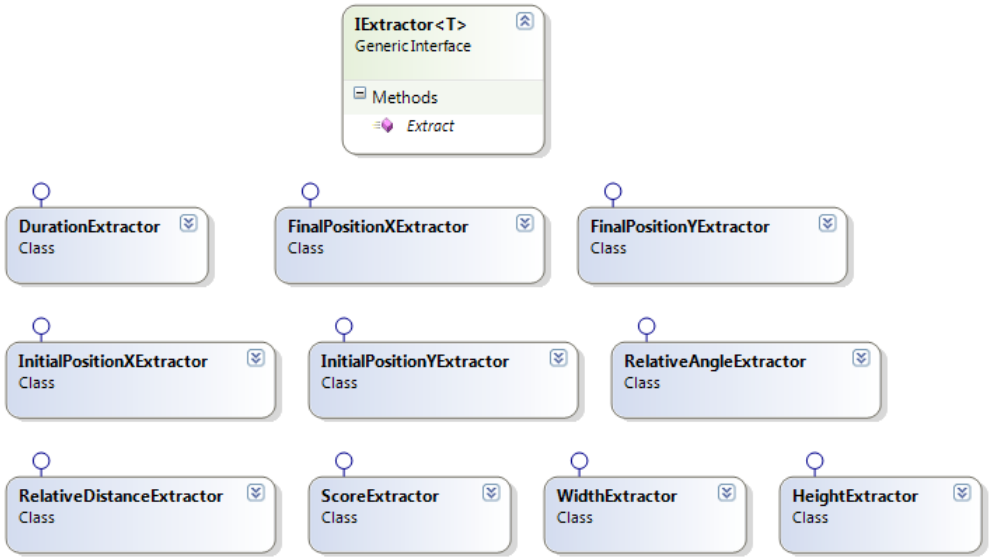

Figure 4.15: Built-in data extraction classes

4.3 The MiTable Tools Package

MiTable Engine provides the abstractions developers need to create their multi-touch and multi-user tabletop applications with more efficiency. However, developing new gestures for such applications is still a difficult task. Also, deciding which gestures recognition package to use is a very challenging task, which may require deep statistical and machine learning concepts depending on the nature of the recognition algorithm. In order to help developer with such tasks, we developed a set of tools, built on top of the *MiTable Engine*, which eases the development of new gestures and the recognizer configuration.

In section 4.3.1, we present the *MiTable Gestures Recorder*, a tool for capturing multi-touch gestures which can be used in an application. The output of this tool can be used to train a gestures recognition system or evaluate different systems using the *MiTable Gestures*

Benchmark tool (Section 4.3.2).

4.3.1 MiTable Gestures Recorder

The *MiTable Gestures Recorder* is a tool for capturing multi-touch gestures and storing them into a text file compatible with the *MiTable Engine*. These files can be used in other *MiTable* tools or in code with the engine's API.

This tool has three tabs: the *Configuration Tab*, where new gestures are added; the *TUIO Tab* (Figure 4.16), where raw gestures data is obtained with the *MiTable TUIO Adapter* for TUIO-compatible devices and the captured gesture is drawn into the screen; and the *Gestures Tab*, where developers can visualize each recorded gesture and, if desired, remove them from the list.

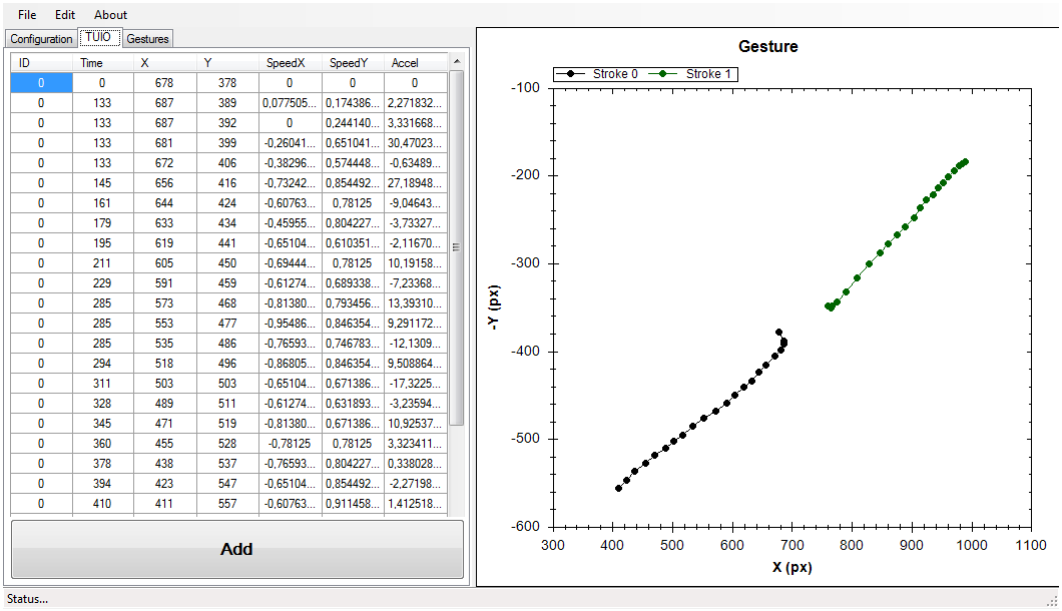

Figure 4.16: Gestures Recorder: TUIO Tab

4.3.2 MiTable Gestures Benchmark

The *MiTable Gestures Benchmark* is a tool for evaluating different multi-touch gestures recognition algorithms. It is a GUI tool for configuring different algorithms and train them with the gestures recorded using the *Mitable Gestures Recorder* tool. It supports all *user-defined* multi-touch gestures recognition algorithms supported by the engine itself, helping developers to chose the best algorithm for their application (Figure 4.17).

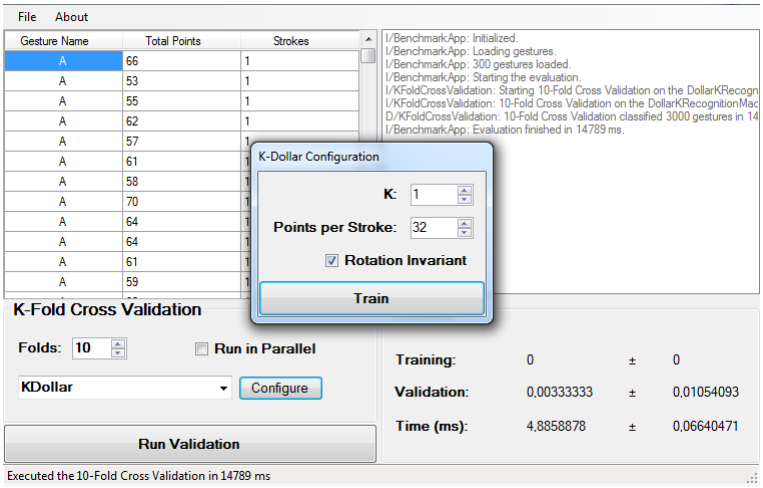

Figure 4.17: MiTable Gestures Benchmark: Evaluating the $N Algorithm

4.4 Proof of Concept: Brainstorming Application

In order to exercise the engine and the interaction between its components, we have designed (with help of dedicated undergrad students) a simple *brainstorming* application. In that application, users can create their *post-its* to fill out their ideas and share them among other participants.

This is a multi-user application that makes use of the following standard gestures: *tap*, *drag* and *drop*. Other standard gestures are filtered out. The *tap* gesture is used to type in the virtual keyboard. The *drag* gesture is used to move *post-its* around the table and the *drop* gesture is used to discard a *post-it*, by dropping it into the thrash can.

The application uses the *Periodic* Segmentation algorithm to receive continuous feedback from the underlying gestures recognizer. Also, as gestures can only be performed over the GUI widgets (buttons, text boxes and images), the *RectangleArea* Mixing Block has been used, providing true multi-user experience in a surface that cannot detect user's information. This has been achieved because, in this application, one widget is controlled by only one user at a time. In this case, the assumption that all *Strokes* captured inside the widget belongs to the same user and the same gesture holds true.

With this application we have been able to validate the concepts proposed, mainly the new recognition pipeline, which have successfully allowed different users to perform gestures on different GUI components at the same time, demonstrating how the engine can be used in true multi-touch and multi-user scenarios.

4.5 Proof of Concept: Symbolic Gestures Recognition

In order to exercise the Recognition Pipeline using state-of-art recognition algorithms, we developed an application that recognizes twelve multi-touch symbolic gestures (Figures 4.18 and 4.19). The gestures names are drawn into the screen when they are recognized.

Figure 4.18: Single-touch symbolic gestures used in the Proof of Concept Application

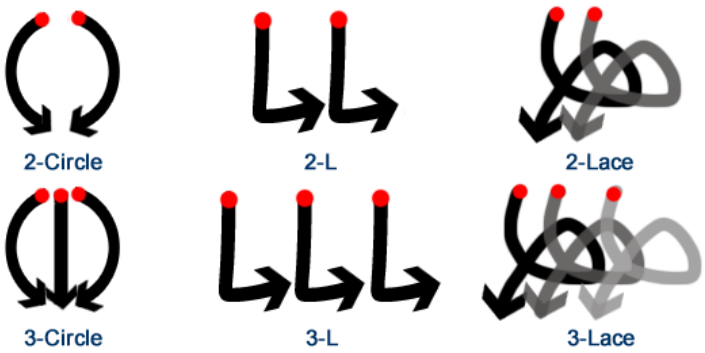

Figure 4.19: Multi-touch symbolic gestures used in the Proof of Concept Application

The applications uses the built-in $N-Protractor algorithm, with its pre-processing and features extraction steps configured as described in previous work (ANTHONY; WOBBROCK, 2012). The Segmentation Block captures *Strokes* performed inside an active area for input, while the Mixing Block considers a single-user scenario, in which all captured *Strokes* belongs to the same user. This is a reasonable scenario for most of the smartphone and tablets applications. This proof of concept application also represents a scenario where user can execute special actions or take short-cuts by performing gestures.

With this application, we have been able to evaluate the performance of the $N-Protractor algorithm within the engine, demonstrating the capabilities of the engine regarding symbolic gestures in a orientation invariant application. This simple application has also demonstrated the flexibility of the engine's API's, which helps developers to integrate existing recognition algorithms or creating customized ones.

4.6 Summary

In this chapter we have presented the *MiTable Engine*, a layered framework for multi-touch and multi-user applications with a novel recognition pipeline. So far we have presented the MiTable Engine, its API and infrastructure. Now, we shall summarize how its architecture and API's addresses each of the fourteen requirements recalled in Chapter 3.

1. Be Flexible and Extensible

The engine allows developers to define their own gestures. The *MultiTouchGesture* class abstracts the multi-touch gesture concept and its type is identified by its $Name$ property. The recognition pipeline identifies which gesture has been performed and fills this property accordingly.

2. Be Fast and 3. Be Accurate

The proposed recognition pipeline enhances the performance of the gestures recognition process by running in separate threads. Also, as the process is broken down into six simpler blocks, the algorithm is more manageable: only the parts needed to support the gestures in a specific application are used. However, we understand that the performance and accuracy is directly related to the performance of the classification block. Inefficient classification algorithms may harm the user experience. The engine comes with some of the state-of-art multi-touch gestures recognizer, which have had their performance accepted by the tabletop community. The engine also allows developers to create their own algorithm that may fit better to their applications.

4. Support Multi-Touch

This requirement is addressed in the core architecture of the engine by the *Multi-TouchGesture*, the *Stroke* and the *TouchPoint* classes as shown previously in Figure 4.2.

5. Support Multi-Users Applications

This requirements is addressed by the initial stages in the recognition pipeline. The Segmentation and Mixing Blocks are responsible for identifying the set of *Strokes* that belongs to each user in order to support multi-user applications. This is not a trivial task, but can be simplified when the hardware provides user information about touch inputs or when some assumptions about the location and size of users hands hold true.

6. Support Orientation Invariance

This requirement is addressed by the Classification Block algorithm. The $N-Protractor

implementation supports Orientation Invariance. This is a requirement that can be addressed not only by the pattern recognition algorithm, but also by the pre-processing and the features extraction implementations as gestures can be normalized to be rotation invariant and rotation-invariant features can be extracted before running the pattern recognition algorithm.

7. Provide Continuous Feedback

This requirement is also addressed by the initial stages of the Recognition Pipeline. Periodic and touch event driven Segmentation algorithms are available out of the box in the engine's API.

8. Allow Easy Prototyping

This requirement is addressed by the engine's Tools Package. The Gestures Recorder tool allows developers to record user-defined gestures for their applications and the Gestures Benchmark tool can be used to tune the recognition algorithm properly. The engine still lacks a tool to ease the creation of gestures based on formal models, like the Gestures Tablature (KIN et al., 2012b; KIN et al., 2012a). This is a subject for future works.

9. Support Symbolic Gestures

This requirement is addressed by the Classification Block algorithm. The engine provides some of the state-of-art recognition algorithms, such as the $N-Protractor, which supports symbolic gestures (ANTHONY; WOBBROCK, 2012).

10. Allow Time-Constrained Gestures

This requirement is also addressed by the Classification Block algorithm. The engine provides a standard gestures recognizer which can distinguish a *tap* from a *press and hold* or a *flick* from a *drag*. Future works shall include support for other recognizers, such as the Midas, Proton++ or GRANDMA, in order to have a better support for time-constrained gestures.

11. Support Territories

This requirement is addressed by the initial stages in the Recognition Pipeline. The engine's API comes with Segmentation and Mixing Blocks implementations that allows developers to define territories where gestures can be performed. Also, gestures that are not allowed to be performed in some territories can be filtered at the Filtering Layer.

12. Recognize Free-form Gestures

This requirement is also addressed by the Classification Block algorithm. The engine provides a standard gestures recognizer which recognize one free-form gesture (*drag and*

drop). In order to support more complex free-form gestures (such as *lasso*), developers must implement their own Classification Block. This is a limitation of the engine which can also be addressed by including other state-of-art recognizers, such as Midas and Proton++.

13. Support Sequential Gestures and 14. Support Cooperative Gestures

These requirements are addressed by the last stages in the Recognition Pipeline. The Combination Block is responsible for combining previously (or simultaneously) made gestures into new sequential or cooperative gestures. Later, the Prioritization Block is responsible to decide which gesture should be passed to the filtering layer and sent to the application handlers. As discussed previously, the Combination algorithm is very application specific and the engine just provides a default implementation.

We understand that all the requirements have been successfully addressed by the engine. However, some of them require some effort from the developers to create custom blocks in the Recognition Pipeline, because their implementation are very application-specific. We claim that the abstractions and API's provided by the engine are enough for such task, but we still have directions for future works. Finally, Table 4.1 compares *MiTable* to other state-of-art multi-touch frameworks, according to the criteria we proposed in Chapter 3.

Table 4.1: Multi-touch frameworks comparison

Criteria	Midas	Proton++	$N-Protractor	MT4J	MiTable
1	●	●	●	●	●
2	●	–	●	–	●
3	–	–	●	–	●
4	●	●	●	●	●
5	●	○	○	○	●
6	○	○	●	○	●
7	●	●	○	●	●
8	○	●	●	○	●
9	●	○	●	○	●
10	●	●	○	●	●
11	●	○	○	●	●
12	●	●	○	●	●
13	●	●	○	○	●
14	●	○	○	○	●

5 CONCLUSION AND FUTURE WORKS

In this work we presented the results of our research on multi-touch gestures recognition systems with focus on multi-user applications. We started with a review on tabletop technologies from the hardware and software perspectives. Then, we studied how tabletops are used, which kinds of interactions are supported, which are their benefits and which kinds of applications have been developed for large multi-touch surfaces.

From that review, we studied the software engineering aspects of such applications. Most of those applications are developed on top of a layered architecture, which consists of the presentation layer, a gestures recognition layer and a hardware abstraction layer. We saw that the TUIO protocol has been widely used to abstract the hardware technology, while the presentation layer is driven by GUI engines specific for multi-touch applications.

For the gestures recognition layer, we have seen several different techniques, which can be categorized as based on machine learning (user-defined gestures) or on formal gestures models. Each technique has its own strengths and weaknesses. Thus, the gestures recognition problem is still an open issue for the tabletop community and lots of work have been done in the past years. In the next sections we summarize our contributions and point out directions for future researches.

5.1 Conclusion

We studied several different multi-touch gestures frameworks and APIs. Some of them provide a HAL, others provide a presentation layer with widgets specially designed for tabletop applications. Also, some of them support tangible objects, while others are focused on multi-touch gestures. From that research on multi-touch interaction and gestures recognition, we had our first contribution: the set of fourteen requirements for multi-touch gestures frameworks:

1. Be Flexible and Extensible

2. Be Fast
3. Be Accurate
4. Support Multi-Touch
5. Support Multi-Users Applications
6. Support Orientation Invariance
7. Provide Continuous Feedback
8. Allow Easy Prototyping
9. Support Symbolic Gestures
10. Allow Time-Constrained Gestures
11. Support Territories
12. Recognize Free-form Gestures
13. Support Sequential Gestures
14. Support Cooperative Gestures

These requirements were designed from the set of heuristics and guidelines for tabletop applications and from the limitations of current gestures recognition systems, as we discussed in Chapter 2 and 3.

Some of the issues in other state-of-art frameworks are due to the fact that the recognition layer is hard-coded for an specific algorithm. On some frameworks, this issue is addressed by providing an abstraction of the gestures recognition algorithm for customization. However, this abstraction enforces developers to implement complex algorithms for gestures recognition and code is hard to be reused and to maintain. This problem gets worse when developers need to implement one algorithm for each available gesture. Also, this architecture makes it hard to combine two or more gestures into another (for sequential or collaborative gestures).

In order to address such issues, we proposed a novel recognition layer. In our research, we have found that the recognition process can be split into six well-defined steps and each step can be reused in different applications individually. Our second contribution is a novel Recognition Pipeline with six processing blocks that can run in parallel for best performance:

Segmentation

Responsible for delimiting the boundaries of a valid finger movement

Mixing

Responsible for identifying which fingers belongs to each gesture

Classification

Responsible for interpreting the unknown gesture

Decision

An optional block used to decide between multiple classification results

Combination

Responsible for combining classified gestures into new gestures

Prioritization

Responsible for solving ambiguities, when more than one valid gesture is possible

Our last contribution is the MiTable Engine: a set of API's and tools for easing the development of multi-touch and multi-user applications. Besides the HAL and the new Recognition Pipeline that is able to run multiple gestures recognition algorithms at the same time, the engine provides a Filtering Layer, which is responsible for rejecting undesirable gestures (e.g. the application does not allow a gesture in a given state or the gesture has a low confidence level). The connection between the engine and the main application is done through an event-driven API based on the Observer-Observable design pattern. Thus, the engine addresses the Kammer's et al criteria for multi-touch frameworks (with exception to the presentation layer) and puts together all findings from this research in a single software solution.

Although the engine has been developed with focus on large multi-touch surfaces, the engine can also be used in small tabletops, such as smartphones and tablets. For multi-touch gestures recognition, the engine incorporates some of the state-of-art algorithms, such as the $N-Protractor, K-Nearest Neighbours and a Finite State Machine for standard gestures.

We have demonstrated by the architecture discussion and the two proof of concepts applications that the resultant engine supports all of the fourteen requirements, although we understand that some of them still require some work from developers because they are very application-dependent.

Finally, the engine comes with two tools to support the tabletop application development: the Gesture Recorder Tool, to record the raw data from gestures into files compatible to the

engine; and the Benchmark Tool, to ease the tuning of the recognition algorithms available for use within the engine's API.

5.2 Future Works

Finally, we would like to point out some directions for future researches:

Include other state-of-art recognizers

It would be a good contribution to the engine if other recognizers were incorporated to MiTable. Mainly those based on formal gestures models, such as Proton++ and Midas. The engine also lacks a tool for building gestures according to a formal model, such as the Proton's Gestures Tablature.

Include a GUI integration layer

Current engine does not provide a presentation layer. The reason is that there are lots of good GUI engines for multi-touch applications that could be used together with MiTable. However, providing an interface for integrating widgets to the recognition pipeline would improve code reuse and speed up the development, as events could be directed to the respective widgets.

Include algorithms for the Decision Block

Making the decision between the classification of different recognizers is not a trivial task. It was out of the scope of this research, but, in order to provide an out of box solution that supports more than one Classification Blocks we need to study how to make decisions based on the results of different classifiers.

Include algorithms for the Combination Block

The implementation of the Combination Block is very application-dependent. However, a research on this subject may find a more general solution for combining previously identified gestures into new ones, enhancing the support for sequential and collaborative gestures.

Develop a real application

We have tested the engine with hundreds of unit tests and two proof of concept applications. However, when developing real applications one may find that the API needs changes or lacks some functions.

Besides future works related to the engine itself, the tabletop community would also benefit from novel recognition algorithms, as there are still open issues on the state-of-art techniques. Also, an important contribution would be a benchmark for evaluating those algorithms with gestures from real applications, in order to allow statistical comparisons between them. We have partially addressed this issue with the Benchmark Tool, with some of the most used statistical comparison methods for classifiers, but we do not have a set of gestures to build a benchmark that would be widely accepted by the tabletop community.

BIBLIOGRAPHY

Anthony, L., and Wobbrock, J. O. A lightweight multistroke recognizer for user interface prototypes. In *Proceedings of Graphics Interface 2010*, GI '10, Canadian Information Processing Society (Toronto, Ont., Canada, Canada, 2010), 245–252.

Anthony, L., and Wobbrock, J. O. $N-protractor: a fast and accurate multistroke recognizer. In *Proceedings of Graphics Interface 2012*, GI '12, Canadian Information Processing Society (Toronto, Ont., Canada, Canada, 2012), 117–120.

Apted, T., Collins, A., and Kay, J. Heuristics to support design of new software for interaction at tabletops. In *CHI '09 Workshop on Multitouch and Surface Computing* (2009).

Baker, K., Greenberg, S., and Gutwin, C. Empirical development of a heuristic evaluation methodology for shared workspace groupware. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, ACM (New York, NY, USA, 2002), 96–105.

Baudisch, P. Natural user interfaces, June 2013.

Chen, F.-S., Fu, C.-M., and Huang, C.-L. Hand gesture recognition using a real-time tracking method and hidden markov models. *Image and Vision Computing 21*, 8 (2003), 745–758.

Cirelli, M., and Nakamura, R. A survey on multi-touch gesture recognition and multi-touch frameworks. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, ACM (New York, NY, USA, 2014), 35–44.

Clayphan, A., Collins, A., Ackad, C., Kummerfeld, B., and Kay, J. Firestorm: a brainstorming application for collaborative group work at tabletops. ITS '11, ACM (New York, NY, USA, 2011), 162–171.

Codeplex. Multi-touch vista, 2009.

Codeplex. MIRIA SDK, 2011.

Dietz, P., and Leigh, D. DiamondTouch: a multi-user touch technology. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, UIST '01, ACM (New York, NY, USA, 2001), 219–226.

Douglas, D., and Peucker, T. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization 10*, 2 (1973), 112–122.

Echtler, F., and Klinker, G. A multitouch software architecture. In *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*, NordiCHI '08, ACM (New York, NY, USA, 2008), 463–466.

Fraunhofer-Institute. MT4j - multitouch for java, 2011.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional, May 1998.

Geyer, F., Pfeil, U., Hochtl, A., Budzinski, J., and Reiterer, H. Designing reality-based interfaces for creative group work. C&C '11, ACM (New York, NY, USA, 2011), 165–174.

Govoni, D. *Java Application Frameworks*, 1 ed. Wiley, June 1999.

Greenberg, S. Computer-supported cooperative work and groupware. Academic Press Ltd., London, UK, UK, 1991, 1–10.

Gross, T., Fetter, M., and Liebsch, S. The cuetable: cooperative and competitive multi-touch interaction on a tabletop. CHI EA '08, ACM (New York, NY, USA, 2008), 3465–3470.

Hakvoort, M. A unifying input framework for multi-touch tables (2009).

Hartmann, B., Morris, M. R., Benko, H., and Wilson, A. D. Augmenting interactive tables with mice & keyboards. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, UIST '09, ACM (New York, NY, USA, 2009), 149–152.

Hinrichs, U., and Carpendale, S. Gestures in the wild: studying multi-touch gesture sequences on interactive tabletop exhibits. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, ACM (New York, NY, USA, 2011), 3023–3032.

Holz, C., and Baudisch, P. Fiberio: A touchscreen that senses fingerprints. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, ACM (New York, NY, USA, 2013), 41–50.

Hoste, L., De Rooms, B., and Signer, B. Declarative gesture spotting using inferred and refined control points. In *Proceedings of the 2nd International Conference on Pattern Recognition Applications and Methods (ICPRAM 2013)* (Barcelona, Spain, 2013), 1–6.

Hoste, L., Dumas, B., and Signer, B. Mudra: A unified multimodal interaction framework. In *Proceedings of the 13th International Conference on Multimodal Interfaces*, ICMI '11, ACM (New York, NY, USA, 2011), 97–104.

Hunter, S., Maes, P., Scott, S., and Kaufman, H. MemTable: an integrated system for capture and recall of shared histories in group workspaces. CHI '11, ACM (New York, NY, USA, 2011), 3305–3314.

Hutama, W., Song, P., Fu, C.-W., and Goh, W. B. Distinguishing multiple smart-phone interactions on a multi-touch wall display using tilt correlation. CHI '11, ACM (New York, NY, USA, 2011), 3315–3318.

Jorda, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, TEI '07, ACM (New York, NY, USA, 2007), 139–146.

Kaltenbrunner, M. reacTIVision and TUIO: a tangible tabletop toolkit. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, ACM (New York, NY, USA, 2009), 9–16.

Kaltenbrunner, M. TUIO protocol specification 2.0, 2014.

Kaltenbrunner, M., Bovermann, T., Bencina, R., and Costanza, E. TUIO: a protocol for table-top tangible user interfaces. In *6th International Gesture Workshop* (2005).

Kammer, D., Keck, M., Freitag, G., and Wacker, M. Taxonomy and overview of multi-touch frameworks: Architecture, scope and features. In *Workshop on Engineering Patterns for Multitouch Interfaces* (2010).

Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++: a customizable declarative multitouch framework. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, UIST '12, ACM (New York, NY, USA, 2012), 477–486.

Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: multitouch gestures as regular expressions. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, ACM (New York, NY, USA, 2012), 2885–2894.

Ko, S., Kim, K., Kulkarni, T., and Elmqvist, N. Applying mobile device soft keyboards to collaborative multitouch tabletop displays: design and evaluation. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '11, ACM (New York, NY, USA, 2011), 130–139.

Kratz, S., and Rohs, M. A $3 gesture recognizer: Simple gesture recognition for devices equipped with 3D acceleration sensors. In *Proceedings of the 15th International Conference on Intelligent User Interfaces*, IUI '10, ACM (New York, NY, USA, 2010), 341–344.

Kruger, R., Carpendale, S., Scott, S. D., and Greenberg, S. How people use orientation on tables: comprehension, coordination and communication. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, GROUP '03, ACM (New York, NY, USA, 2003), 369–378.

Kuhn, H. W. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly 2*, 1-2 (Mar. 1955), 83–97.

Li, Y. Protractor: a fast and accurate gesture recognizer. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, ACM (New York, NY, USA, 2010), 2169–2172.

Liou, E., Paulo, H., Gutierrez, M., Cirelli, M., Filgueiras, L., and Nakamura, R. *Framework para reconhecimento de gestos em mesa multi-toque e multi-usuario*. PhD thesis, Universidade de Sao Paulo, Sao Paulo, 2012.

Marquardt, N., Kiemer, J., Ledo, D., Boring, S., and Greenberg, S. Designing user-, hand-, and handpart-aware tabletop interactions with the TouchID toolkit. ITS '11, ACM (New York, NY, USA, 2011), 21–30.

Martínez Maldonado, R., Kay, J., and Yacef, K. Collaborative concept mapping at the tabletop. ITS '10, ACM (New York, NY, USA, 2010), 207–210.

Masoodian, M., McKoy, S., and Rogers, B. Hands-on sharing: collaborative document manipulation on a tabletop display using bare hands. CHINZ '07, ACM (New York, NY, USA, 2007), 25–31.

Mcgrath, J. *Groups: Interaction and Performance*. {Prentice Hall College Div}, 1984.

Meyer, T., and Schmidt, D. IdWristbands: IR-based user identification on multi-touch surfaces. In *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '10, ACM (New York, NY, USA, 2010), 277–278.

Microsoft. .NET framework, 2011.

Microsoft. Microsoft surface SDK, 2012.

Mindstorm. Breezemultitouch, 2010.

Mitra, S., and Acharya, T. Gesture recognition: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 37*, 3 (2007), 311–324.

Morris, M. R., Huang, A., Paepcke, A., and Winograd, T. Cooperative gestures: multi-user gestural interactions for co-located groupware. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, ACM (New York, NY, USA, 2006), 1201–1210.

Nielsen, J. *Usability Engineering*. Morgan Kaufmann, 1993.

NUIGroup, T. *Multitouch Technologies*, 1 ed. The NUI Group, 2009.

Nygard, E. S., and Thomassen, A. *Multi-touch Interaction with Gesture Recognition*. PhD thesis, 2010.

Prekopcsak, Z. *Accelerometer Based Real-Time Gesture Recognition*. 2008.

Puckdeepun, T., Jaafar, J., Hassan, M. F., and Hussin, F. A. Investigating collaborative interaction using interactive table and IR devices. In *2010 International Conference on User Science and Engineering (i-USEr)*, IEEE (Dec. 2010), 83–88.

Qin, Y., Yu, C., Liu, J., Wang, Y., Shi, Y., Su, Z., and Shi, Y. uTable: a seamlessly tiled, very large interactive tabletop system. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '11, ACM (New York, NY, USA, 2011), 244–245.

Renaux, T., Hoste, L., Marr, S., and De Meuter, W. Parallel gesture recognition with soft real-time guarantees. In *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! '12, ACM (New York, NY, USA, 2012), 35–46.

Rhyne, J. R., and Wolf, C. G. Gestural interfaces for information processing applications. Tech. Rep. RC12179, IBM Corporation, Yorktown Heights, NY, 1986.

Rubine, D. Specifying gestures by example. *SIGGRAPH Comput. Graph. 25*, 4 (July 1991), 329–337.

Salvadore, S., and Chan, P. FastDTW: toward accurate dynamic time warping in linear time and space. In *KDD Workshop on Mining Temporal and Sequential Data* (2004), 70–80.

Schell, J. *The Art of Game Design: A Book of Lenses*. Elsevier/Morgan Kaufmann Publishers, 2008.

Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, TEI '11, ACM (New York, NY, USA, 2011), 49–56.

Scott, S. D., Grant, K. D., and Mandryk, R. L. System guidelines for co-located, collaborative work on a tabletop display. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, Kluwer Academic Publishers (Norwell, MA, USA, 2003), 159–178.

Scott, S. D., Sheelagh, M., Carpendale, T., and Inkpen, K. M. Territoriality in collaborative tabletop workspaces. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, ACM (New York, NY, USA, 2004), 294–303.

Seow, S. C., Wixon, D., Morrison, A., and Jacucci, G. Natural user interfaces: The prospect and challenge of touch and gestural computing. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '10, ACM (New York, NY, USA, 2010), 4453–4456.

Shaer, O., Strait, M., Valdes, C., Feng, T., Lintz, M., and Wang, H. Enhancing genomic learning through tabletop interaction. CHI '11, ACM (New York, NY, USA, 2011), 2817–2826.

Shen, C., Vernier, F. D., Forlines, C., and Ringel, M. DiamondSpin: an extensible toolkit for around-the-table interaction. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, ACM (New York, NY, USA, 2004), 167–174.

Sparsh-UI. Sparsh-UI, 2010.

Strothoff, S., Valkov, D., and Hinrichs, K. Triangle cursor: interactions with objects above the tabletop. ITS '11, ACM (New York, NY, USA, 2011), 111–119.

Tabard, A., Hincapié-Ramos, J.-D., Esbensen, M., and Bardram, J. E. The eLabBench: an interactive tabletop system for the biology laboratory. ITS '11, ACM (New York, NY, USA, 2011), 202–211.

Tang, J. C. Findings from observational studies of collaborative work. *International Journal of Man-Machine Studies 34*, 2 (1991), 143–160.

Tse, E., Greenberg, S., Shen, C., and Forlines, C. Multimodal multiplayer tabletop gaming. *Comput. Entertain. 5*, 2 (2007).

Vandoren, P., Van Laerhoven, T., Claesen, L., Taelman, J., Di Fiore, F., Van Reeth, F., and Flerackers, E. Dip - it: digital infrared painting on an interactive table. In *CHI '08 extended abstracts on Human factors in computing systems*, CHI EA '08, ACM (New York, NY, USA, 2008), 2901–2906.

Vatavu, R.-D., Anthony, L., and Wobbrock, J. O. Gestures as point clouds: a $p recognizer for user interface prototypes. In *Proceedings of the 14th ACM international conference on Multimodal interaction*, ICMI '12, ACM (New York, NY, USA, 2012), 273–280.

Vermeulen, A., Beged-dov, G., and Thompson, P. *The Pipeline Design Pattern*. 1995.

Wigdor, D., Forlines, C., Baudisch, P., Barnwell, J., and Shen, C. Lucid touch: A see-through mobile device. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, ACM (New York, NY, USA, 2007), 269–278.

Wigdor, D., Shen, C., Forlines, C., and Balakrishnan, R. Table-centric interactive spaces for real-time collaboration. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '06, ACM (New York, NY, USA, 2006), 103–107.

Wilson, A. D., and Benko, H. Combining multiple depth cameras and projectors for interactions on, above and between surfaces. UIST '10, ACM (New York, NY, USA, 2010), 273–282.

Wilson, A. D., and Sarin, R. BlueTable: connecting wireless mobile devices on interactive surfaces using vision-based handshaking. In *Proceedings of Graphics Interface 2007*, GI '07, ACM (New York, NY, USA, 2007), 119–125.

Wobbrock, J. O., Morris, M. R., and Wilson, A. D. User-defined gestures for surface computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, ACM (New York, NY, USA, 2009), 1083–1092.

Wobbrock, J. O., Wilson, A. D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, ACM (New York, NY, USA, 2007), 159–168.

Wu, J., Pan, G., Zhang, D., Qi, G., and Li, S. Gesture recognition with a 3-d accelerometer. In *Ubiquitous Intelligence and Computing*, D. Zhang, M. Portmann, A.-H. Tan, and J. Indulska, Eds., no. 5585 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2009, 25–38.

Xiao-li, W., and De, Z. Selecting optimal threshold value of douglas-peucker algorithm based on curve fit. In *2010 First International Conference on Networking and Distributed Computing (ICNDC)* (2010), 251–254.

Yoon, H.-S., Soh, J., Bae, Y. J., and Seung Yang, H. Hand gesture recognition using combined features of location, angle and velocity. *Pattern Recognition 34*, 7 (2001), 1491–1501.

Zeitler, A., and Hussman, H. *Survey and Review of Input Libraries, Frameworks, and Toolkits for Interactive Surfaces and Recommendations for the Squidy Interaction Library*. PhD thesis, Universitat Munchen, 2009.