

UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA

PH.D. THESIS

A Model-Based Approach for Gesture Interfaces

Lucio Davide Spano

SUPERVISORS

Antonio Cisternino

Fabio Paternò

REFEREES

Gaëlle Calvary

Kris Luyten

November 2013

SSD INF/01

Largo Bruno Pontecorvo 3, 56127 Pisa, Italy.
Email: spano@di.unipi.it

This one goes out to the one I love

“People shouldn’t have to read a manual to open a door, even if it is only one word long (push/pull).”

Don Norman

“La vita in Sardegna è forse la migliore che un uomo possa augurarsi: ventiquattro mila chilometri di foreste, di campagne, di coste immerse in un mare miracoloso dovrebbero coincidere con quello che io consiglierei al buon Dio di regalarci come Paradiso.”

Fabrizio De André

“And the only way to do great work is to love what you do. If you haven't found it yet, keep looking. Don't settle. As with all matters of the heart, you'll know when you find it. And, like any great relationship, it just gets better and better as the years roll on. So keep looking until you find it. Don't settle.”

Steve Jobs

“A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.”

Douglas Adams

“Est mezus ainu biu qui non doctore mortu”.

Sardinian proverb

“Do. Or do not. There is no try”

Master Yoda

Abstract

The description of a gesture requires temporal analysis of values generated by input sensors, and it does not fit well the observer pattern traditionally used by frameworks to handle the user's input. The current solution is to embed particular gesture-based interactions into frameworks by notifying when a gesture is detected completely. This approach suffers from a lack of flexibility, unless the programmer performs explicit temporal analysis of raw sensors data.

This thesis proposes a compositional, declarative meta-model for gestures definition based on Petri Nets. Basic traits are used as building blocks for defining gestures; each one notifies the change of a feature value. A complex gesture is defined by the composition of other sub-gestures using a set of operators. The user interface behaviour can be associated to the recognition of the whole gesture or to any other sub-component, addressing the problem of granularity for the notification of events.

The meta-model can be instantiated for different gesture recognition supports and its definition has been validated through a proof of concept library. Sample applications have been developed for supporting multi-touch gestures in iOS and full body gestures with Microsoft Kinect.

In addition to the solution for the event granularity problem, this thesis discusses how to separate the definition of the gesture from the user interface behaviour using the proposed compositional approach.

The gesture description meta-model has been integrated into MARIA, a model-based user interface description language, extending it with the description of full-body gesture interfaces.

Acknowledgments

This thesis, like many others, ends a chapter. And this is not only related to a dot on paper. For me, this thesis completes a journey, which I started ten years ago, when I finished the high school and I moved to Pisa to learn something about Computer Science and for becoming a good programmer. During this journey, I tried to learn all the skills that a good programmer and a good researcher needs. But that was only a part of the journey. I met great people who taught me a lot more than algorithms, programming languages and techniques. The passion they put in their work, together with their competence created an inspiration that has been and will be of primary importance for me.

Now I have the possibility to represent for other people what they are for me, and I hope I will be as good as they are.

The first persons in this list are of course my two supervisors: Antonio Cisternino and Fabio Paternò. Antonio is able to inject enthusiasm into his students, and this always helped me in reaching the next level. Every time I discuss something with him, I can write down a list of ideas for the next five or six years.

Fabio, with his great knowledge and experience, guided me through different projects, giving me the opportunity to start my work in research, at a European level. He supported me day by day and I owe him all the experience I have in the HCI field.

Many thanks to the two reviewers Gaëlle and Kris, who have been so kind to accept reading this work and for all their useful suggestions.

Special thanks go to Paolo Mancarella, for his help and his kindness during these years. Thanks also to Pierpaolo Degano, for the effort he puts in managing the PhD course.

I would like to mention here one by one all the people I met while working at the Human Interfaces in Information Systems lab at ISTI-CNR: Giuseppe, Carmen, Barbara, Giulio Galesi, Giulio Mori, Claudio, Marco,

Giuliano, Valentina, José, Christian, Mauro, Ariel. I really miss you all. A fond farewell to you, Antonio, wherever you are now.

Many friends contributed in making Pisa my second home in these years. First of all, the CdP members: Agostino, Giancarlo, Marta, Antonio, Zulio, Alessandra, Federica, Raimondo, Giancossu, Pier. My “colleague” Claudio and Cristina, Alessandro, Elena that never left us alone, Carlo, Rita, Chiara, Anna, Paolo, Claudia, Antonio, and all the people that shared this wonderful experience with me.

I really missed my hometown friends during these years, and it was always nice to come back to you: Nicola, Ilaria, Gavinuccio, Barbara, Maura, Manuela, German, Mattia, Nicoletta, Andrea, Nicola Giorgioni, Carlo, Patty, Roberto, Carla.

I would like to thank also all the people that welcomed me in my new adventure in Cagliari: the professors Gianni Fenu and Riccardo Scateni, Maurizio Atzori, Marco, Paolo, Fabio, Samuel.

A special thank, with love and gratitude goes to my family. My parents gave me the possibility to make such a wonderful experience, and they were always at my side. I always do my best to make you proud of me, and this work is largely yours. Thanks to my two brothers Andrea and Emanuele, who supported in all these years, filling the distance that separated us. Thanks also to Roberta and Laura, it is really nice to have you in our family.

Many thanks to all my uncles and cousins and every member of my second family in Dorgali, who accepted me as one them.

Finally, my greatest thanks to my fiancée Pinuccia. Writing down what you mean for me is harder than writing a PhD thesis. Only one word comes into my mind: “everything”. If we are together, we can face challenges and changes, double the joy and halve the sadness. Without you, this work would not have been possible. Without you, my work in Cagliari will not be possible. Without you, I am lost.

Thank you all.

Contents

Chapter 1 Introduction.....	19
1.1 Context and motivations	19
1.2 Objectives of the thesis	21
1.3 Requirements summary	21
1.4 Overview of the results	22
1.5 Thesis organization.....	23
1.6 Peer-reviewed publications.....	24
Chapter 2 Background and Related Work	25
2.1 Enabling recognition technologies	25
2.1.1 Multitouch.....	26
2.1.2 Remote-based gesture recognition	27
2.1.3 Image-based gesture recognition.....	28
2.1.4 Floor devices.....	30
2.2 Input modelling with formal approaches	31
2.3 Declarative approaches for gesture definition	33
2.3.1 Multitouch.....	34
2.3.2 Full-body	41
2.4 Model-based approaches for User Interfaces.....	44
2.4.1 Historical Background.....	45
2.4.2 The CAMELEON reference framework.....	50
2.4.3 ConcurTaskTrees.....	53
2.5 Non-Autonomous Petri Nets.....	54
Chapter 3 Gesture Meta-Model Definition.....	58
3.1 Meta-Model Definition.....	58
3.1.1 Basic Building Blocks: Ground Terms	59
3.1.2 Composition Operators	61
3.1.3 Handling recognition errors.....	72
3.2 Modelling multitouch gestures	73
3.3 Modelling full-body gestures	73
3.4 Comparison with Proton++	76
3.4.1 Proton++ literals	76

3.4.2	Proton++ operators.....	78
Chapter 4	Gesture Models	81
4.1	Common multitouch gestures models.....	81
4.1.1	Tap.....	81
4.1.2	Double Tap.....	82
4.1.3	Pan.....	82
4.1.4	Slide.....	83
4.1.5	Pinch	83
4.1.6	Rotate.....	84
4.2	Common full-body gesture models	85
4.2.1	Pointing.....	89
4.2.2	Grab	90
4.2.3	Push	91
4.2.4	Push back.....	92
4.2.5	Lateral push.....	93
4.2.6	Kick	94
4.2.7	Wave	94
4.2.8	Swipe	97
4.2.9	Walk.....	98
4.2.10	Turn	100
4.2.11	Converge or Diverge Hands.....	101
4.2.12	Steering wheel.....	103
4.2.13	Roll.....	104
4.2.14	Universal Pause.....	105
Chapter 5	Library Support.....	107
5.1	Library Architecture.....	107
5.1.1	Library core	108
5.1.2	Multitouch package.....	111
5.1.3	Full-body package.....	111
5.2	Creating a multitouch application	112
5.3	Creating a full-body gesture application	116
5.4	Sample applications	119
5.4.1	Pilot study: Simple canvas.....	119
5.4.2	Photo viewer.....	122
5.4.3	3D viewer.....	124
5.4.4	Touchless recipe browser.....	128
Chapter 6	A Gestural Concrete User Interface in MARIA.....	135
6.1	MARIA.....	135
6.1.1	Abstract User Interface.....	136
6.1.2	Concrete User Interface.....	138
6.2	Gestural Concrete User Interface	139

6.2.1	Modelling device data	140
6.2.2	Gestures definition.....	140
6.2.3	Gesture effects	143
6.2.4	Interactors	145
6.3	Model to code transformation	146
6.4	Sample application.....	148
Chapter 7 Discussion		157
7.1	Granularity problem	158
7.2	Spaghetti code problem	159
7.3	Selection Ambiguity Problem	160
7.4	Cross-platform gesture modelling.....	165
Chapter 8 Evaluation.....		169
8.1	Requirements review.....	170
8.1.1	Temporal evolution.....	170
8.1.2	Granularity	170
8.1.3	Separation of concerns	171
8.1.4	Multiple recognition devices.....	171
8.1.5	Parallel interaction	171
8.1.6	Equivalent descriptions.....	172
8.1.7	Selection ambiguity.....	172
8.2	Five themes in evaluating tools.....	172
8.2.1	Parts of the user interface that are addressed	173
8.2.2	Threshold and ceiling.....	173
8.2.3	Path of Least Resistance.....	174
8.2.4	Predictability	174
8.2.5	Moving Targets.....	175
8.3	Cognitive Dimensions Framework.....	176
8.3.1	Abstraction gradient.....	176
8.3.2	Closeness of mapping	176
8.3.3	Consistency.....	177
8.3.4	Diffuseness	177
8.3.5	Error proneness.....	177
8.3.6	Hard mental operations.....	178
8.3.7	Hidden dependencies	178
8.3.8	Premature commitment	178
8.3.9	Progressive evaluation	179
8.3.10	Role expressiveness	179
8.3.11	Secondary notation	179
8.3.12	Viscosity	180
8.3.13	Visibility	180
8.4	Performance analysis	181

8.4.1	CPU (sampling)	183
8.4.2	CPU (instrumentation)	186
8.4.3	Memory	188
8.5	Summary	190
Chapter 9	Conclusion.....	191
9.1	Future work.....	192

List of Figures

Figure 2.1 An example of gesture definition with Proton++	39
Figure 2.2 An example of Petri Net	54
Figure 2.3 Transition firing in Petri Nets from [36], p. 3.	55
Figure 2.4 Non-Autonomous Petri Net for a traffic light, from [36] p.4	56
Figure 3.1 Gesture recognition building block.....	61
Figure 3.2 The Iterative operator.....	64
Figure 3.3 The Sequence operator.....	64
Figure 3.4 The Parallel operator	65
Figure 3.5 The Choice operator (immediate variant)	66
Figure 3.6 Choice operator (best effort variant).....	67
Figure 3.7 The Disabling operator	68
Figure 3.8 Order independence operator Petri Net	71
Figure 3.9 Skeleton joints.....	74
Figure 3.10 Full-body gesture coordinate system	75
Figure 4.1 The touch gesture	81
Figure 4.2 Double tap gesture	82
Figure 4.3 Pan gesture	82
Figure 4.4 Pinch gesture	84
Figure 4.5 Rotate gesture.....	84
Figure 4.6 The pointing gesture	89
Figure 4.7 The grab gesture	91
Figure 4.8 The push gesture.....	92
Figure 4.9 The push-back gesture	92
Figure 4.10 Lateral push gesture.....	93
Figure 4.11 The kick gesture.....	94
Figure 4.12 The wave gesture	95
Figure 4.13 The swipe gesture.....	97
Figure 4.14 Walk gesture	98
Figure 4.15 The turn gesture.....	101
Figure 4.16 Converge or diverge hands gesture.....	102

Figure 4.17 Steering wheel gesture	103
Figure 4.18 Roll gesture.....	104
Figure 4.19 The Universal Pause gesture.....	105
Figure 5.1 GestIT class diagram.....	108
Figure 5.2 Recognition of a pinch gesture with the GestIT library	114
Figure 5.3 Recognition of a pinch gesture (sequence diagram)	115
Figure 5.4 Touchless recipe browser, dish type selection	117
Figure 5.5 Simple canvas UI, multitouch version	121
Figure 5.6 Simple canvas UI, full-body version	122
Figure 5.7 The photo viewer application	124
Figure 5.8 3D viewer interaction	126
Figure 5.9 3D viewer UI, grab gesture.....	127
Figure 5.10 3D viewer UI, roll gesture.....	127
Figure 5.11 Recipe category selection	131
Figure 5.12 Recipe selection.....	131
Figure 5.13 Recipe browser.....	132
Figure 6.1 MARIA gesture description meta-model	141
Figure 6.2 Task model for the TV control application	149
Figure 6.3 TV application AUI.....	150
Figure 6.4 MARIA application: function selection presentation	151
Figure 6.5 MARIA application: channel information presentation	152
Figure 6.6 MARIA application: channel selection presentation	152
Figure 7.1 Common prefix handling for the choice operator (1).....	163
Figure 7.2 Common prefix handling for the choice operator (2).....	164
Figure 8.1 Finite State Machine for the 3D viewer interaction.....	182
Figure 8.2 3D viewer CPU usage (FSM version)	184
Figure 8.3 3D viewer CPU usage (GestIT version).....	184

List of Tables

Table 2.1: Comparison of different multitouch gestures definition approaches in literature	38
Table 2.2 Comparison of different full-body gestures definition approaches in literature	44
Table 3.1 Composition Operators.....	62
Table 3.2 Mapping a Proton++ literal to a GestIT ground term	77
Table 3.3 Mapping Proton++ operators to GestIT	78
Table 4.1 Common full-body gestures in literature	88
Table 5.1 XAML Gesture definition.....	118
Table 6.1: Channel selection gesture	154
Table 7.1 Grab and Drag gestures defined with GestIT.....	158
Table 7.2: Grab and Drag gestures e parametric definition	160
Table 7.3 Gesture definition for the 3D viewer application.....	161
Table 7.4 Simple drawing canvas gesture modelling	166
Table 7.5 Mapping multitouch ground terms to the full-body platform	167
Table 8.1 3D viewer CPU profiling (sampling, FSM version)	185
Table 8.2 3D viewer CPU profiling (sampling, GestIT version).....	186
Table 8.3 3D viewer CPU profiling (instrumentation, FSM version)	187
Table 8.4 3D viewer CPU profiling (instrumentation, GestIT version)...	187
Table 8.5 3D viewer memory profiling (FSM version)	189
Table 8.6 3D viewer memory profiling (GestIT version).....	189

Chapter 1

Introduction

1.1 Context and motivations

In recent years, a wide variety of new input devices has changed the way we interact with computers. Nintendo Wii in 2006 has broken the point and click paradigm with the Wiimote controller, based on gestures in a 3D space; iPhone has shown better usability by means of multi-touch in 2007, while Microsoft introducing Kinect in 2010 has expressed a way of interaction without wearing sensors of any kind. All these new devices exploit gestures performed in different ways, such as moving a remote, touching a screen, or through whole-body movements.

The introduction of such novel interaction techniques in the mass market has not yet affected the current user interface programming frameworks: the underlying model is still bound to the observer pattern [141] where events occur atomically in time and they are notified through messages or callbacks. The support for gestures has been mostly forced in the same paradigm by hiding the gesture recognition logic under the hood, which usually means providing high-level events when the gesture is completed, and leaving the possibility to provide intermediate feedback to the handling of low-level events, which are not correlated with the high-level ones.

Indeed, it is difficult to create gestural interfaces following the observer pattern for two main reasons. The first one is that the temporal extension of a gesture is significant with respect to the time scale of a system, since a gesture may require seconds to complete. The observer pattern is particularly effective when applied to events that can be considered atomic from the system's and the user's point of view: a button click takes such a small amount of time that both the user and the application can ignore what happens *during* the click. Gestures break this assumption, since they have a

longer duration in time. In addition, the application usually has to provide feedback during the gesture execution, in order to guide the users. Therefore, a single event does not fit a gesture in general.

In addition, the observer pattern has been successfully adopted in the development of user interfaces (UIs), since it is particularly effective in describing actions that do not have temporal relationships between them. For instance, the handlers that deal with the pointer interaction work independently from the timing sequence of e.g. the keyboard events.

Such property, which is a strength for classic WIMP (Window, Icon, Menu, Pointing device) UIs, is the second problem in modelling gestural interaction with the observer pattern. Indeed, in order to recognize a gesture, a developer has to define the *temporal relationships* among different low-level device events, through code that tracks the order of the received events. For instance, in order to recognize a pinch gesture, the developer has to ensure that at least two fingers are currently touching the screen before reacting to touch move events. The code that establishes whether the event sequence is correct or not is mixed with the definition of the user interface behaviour, increasing the code complexity and limiting its reuse.

Another aspect that is difficult to model with the observer pattern in UI development is related to the animations. A simple approach may rely on a timer tick notification. Each time the tick handler is triggered, the code changes some visualization attributes and repaints the view. However, when we want to compose more than one animation, such approach is difficult to maintain since the actions that deal with the different animations are mixed. The point is that it is easier to describe an animation as a *continuous* rather than a *discrete* process. Most modern UI toolkits describe the animations providing an initial state, a final state and an interpolation algorithm between the two states. Given the duration, the UI toolkit can define a set of discrete steps that changes the UI state from the initial to the final one.

An effective approach for modelling gestures solves the dual problem: for each single discrete step (the low-level device events) in a given set (how the user performs the gesture), it should be able to provide information on the distance between the initial state (the start of a gesture) and the final state (the end of a gesture), *continuously*.

This means that gesture description should have different levels of *granularity*: it should be possible to consider it as a whole, reacting to its complete execution, but it should be also possible to associate feedback and

UI behaviour to gesture sub-parts, in order to support users during the execution of a complex gesture.

1.2 Objectives of the thesis

The objective of this thesis is the definition of a gesture meta-model that can be effectively used for creating descriptions at the desired level of granularity.

The meta-model should be abstract enough to describe gestures recognized by different devices (e.g. touch screens, remotes, Microsoft Kinect etc.). We follow a compositional approach: the definition of a complex gesture is created through the composition of smaller sub-gestures, connected through a set of operators. Such approach allows to declarative define a gesture and to reuse its definition in more than one application, independently from a given UI control. In this way, it is possible for instance to separate the pinch gesture from the image viewer that exploits it for e.g. enlarging a photo.

In addition, the developer should be able to attach the behaviour definition to the different parts of a gesture, either if its recognition completes successfully or in case of partial recognition.

Once such meta-model has been defined, it should be instantiated for at least to two different gesture recognition techniques, in order to validate it with different sources of input.

Finally, we want to demonstrate its effectiveness through a set of applications that define gestural interaction through the modelling elements.

We consider out of scope for this thesis an evaluation of the overall usability of the applications created with the proposed approach. The effort required for investigating the correlation between modelling and usability forced us to focus on the meta-model definition and validation, but we plan to consider this aspect in further research.

1.3 Requirements summary

In this section, we summarize the requirements we identified as success criteria for the definition of our gesture description meta-model. The motivation for such requirements is discussed in Chapter 2.

- R1. *Temporal evolution.* The meta-model must describe the gesture temporal evolution. The developers should be able to define the behaviour of the user interface according to this temporal

evolution, without the need of tracking explicitly the different stages of the gesture performance outside the model definition.

- R2. *Granularity.* Provided that a gesture may take seconds to complete, it must be possible for developers to define user interface reactions to partially completed gestures, not only to their complete recognition.
- R3. *Separation of concerns.* The definition of gestures and the user interface behaviour must be separated, in order to allow the reuse of the same gesture model in different applications.
- R4. *Multiple recognition devices.* The meta-model must support different recognition devices, abstracting from a particular recognition technology.
- R5. *Parallel interaction.* The meta-model must handle the recognition of different gestures at the same time, in order to allow parallel interactions with the same application.
- R6. *Equivalent descriptions.* The same gesture can be performed in different ways (e.g. a pinch may be performed either with one hand or with two hands). The meta-model must support the definition of equivalent gestures.

During the development of the proposed modelling approach, we identified another requirement that does not apply to the gestural interaction modelling in general, but only to compositional approaches:

- R7. *Selection ambiguity.* The recognition support must provide means for identifying or managing the selection between two different gestures that shares the same initial sequence.

1.4 Overview of the results

This thesis describes the following research results:

- The definition of GestIT (Gesture In Time), an abstract gesture description meta-model, based on the composition of a ground terms (which represent atomic gestures) through a set of composition operators. The semantics of the meta-model elements have been defined through Non-Autonomous Petri Nets [36].
- The instantiation of the abstract gesture description meta-model for describing two different recognition supports: multitouch and full-body.

- The implementation of a proof-of-concept library that allows creating user interfaces exploiting the gesture models. The library can be exploited for creating multitouch and full-body gesture applications.
- The implementation of a set of sample applications that demonstrate the effectiveness of the meta-model.
- The integration of the gesture modelling technique into MARIA [111], a state of the art User Interface Description Language

1.5 Thesis organization

The thesis is organised as follows:

- Chapter 1 introduces the context and the motivation of the thesis.
- Chapter 2 discusses the related work and different devices and solutions for gesture recognition.
- Chapter 3 introduces the abstract gesture meta-model, and its instantiation for multitouch and full body gestures
- Chapter 4 defines a set of gestures for multitouch and full-body interaction using the proposed modelling approach.
- Chapter 5 discusses a proof-of-concept library that supports the meta-model, together with a set of sample applications.
- Chapter 6 extends the MARIA [111] User Interface Description Language with gestural interaction.
- Chapter 7 discusses how the proposed modelling approach addresses three different problems in modelling gestural interaction: the support for different granularity levels, the separation between the gesture recognition code and the definition of the UI behaviour, the ambiguities in the definition of different gestures that have a common prefix.
- Chapter 8 reports an evaluation of the proposed meta-model, according to the success parameters established for the thesis. We report also an inspection of the notation according to two different frameworks: Myers et al. [97] and the cognitive dimensions [51]. Finally we report on a preliminary analysis of the GestIT library performance.
- Chapter 9 summarizes the results and describe possible directions for further research.

1.6 Peer-reviewed publications

The following is the list of peer-reviewed publications that are a direct outcome of this research:

- Spano, L.D. A model-based approach for gesture interfaces. *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, ACM (2011), 327–330.
- Spano, L.D., Cisternino, A., and Paternò, F. A Compositional Model for Gesture Definition. *Proceedings of the 4th International Conference in Human-Centered Software Engineering (HCSE 2012)*, LNCS, Springer (2012), 34–52
- Spano, L.D. Developing Touchless Interfaces with GestIT. In F. Paternò, B. de Ruyter, P. Markopoulos, C. Santoro, E. van Loenen and K. Luyten, eds., *Ambient Intelligence*. Springer Berlin / Heidelberg, 2012, 433–438.
- Spano, L.D., Cisternino, A., Fabio, P., and Fenu, G. A Declarative and Compositional Framework for Multiplatform Gesture Definition. *EICS 2013, 5th Symposium on Engineering Interactive Computing Systems*, ACM Press (2013).

Chapter 2

Background and Related Work

This chapter provides an overview on the research topics that have inspired or that are advanced in this thesis. The discussion starts with a quick overview of the different devices that can be used for creating gestural interfaces. After that, we provide a background on previous work on formal techniques for modelling the user input. Then, we focus more on other declarative approaches for modelling gestural interaction, briefly comparing them with the one proposed in this dissertation. Next, we provide background information on model-based approaches for user interfaces. Finally, we introduce the Non-Autonomous Petri Nets, since we exploit them for formally defining our gesture meta-meta model.

2.1 Enabling recognition technologies

This section discusses the main advances and innovations that introduced gestural interaction to the mass-market. Some of them are due to commercial innovation in existing platforms, such as mobile devices or game consoles. Others have a long history, and eventually the technology evolution (e.g. the increase of computing capabilities of mobile phones) created the possibility to make them available to a wider set of users.

First, we have to define the meaning of the word “gesture”. Gestures consist of movements of hands, face or other parts of the body that are used for communication between people, replacing or enhancing speech. Gestural interfaces emulate such kind of communication, recognizing a set of gestures and exploiting them as input for computers [76].

Many tracking and sensing technologies have been employed in order to recognize gestures through about thirty years of research. For instance, in 1986 Zimmerman et al. [148] already created gloves equipped with sensors

and force feedback for measuring finger bending and recognizing hand motions.

With respect to the recognition techniques, in [95] it is possible to find a survey on the different approaches for gesture recognition. In particular, we can list the following techniques that have been employed for arm and hand gestures: Hidden Markov Models [132,143], Particle Filtering and Condensation [60,84], Finite State Machines [17] and Neural Networks [144].

Other computer-vision techniques that have been applied for recognizing facial motions are important also for full-body gestures, such as Hidden Markov Models, Principal Component Analysis [132], Contour Models [69] Feature Extraction[101], Gabor Filtering [83].

The quest for a technique that is able to combine the recognition of natural movements with a high level of precision is the “Holy Grail” for the gestural interaction [40] and it is both one of the most investigated aspects and one of the most challenging and open research question. However, the techniques that enable the recognition of the gestures either on the hardware or on the software side are not in the scope of this thesis. We aim to define an effective model for defining the gesture structure according to the different features that are provided by the recognition platform.

2.1.1 Multitouch

Multitouch UIs recognize the position of different touches on the same screen simultaneously. Even if such kind of interaction has become popular after the iPhone launch in 2007 [4], it is possible to find in literature systems that used such screen interaction technique already in 1984 [68]. A survey on the history of touch-based systems can be found in [25].

The technology support has been refined through the years the technique was applied mainly for large and collaborative projective walls as, for instance, in the Diamond Touch system [37]. It allowed multiple-user and touches recognition using an array of antennas embedded in a table top, used as a projector screen.

The application of the Frustrated Total Internal Reflection [54] introduced an innovative and low-cost implementation for a multi touch surface.

However, the industrial success of multitouch arrived with its application on mobile devices: Apple introduced in 2007 the iPhone, which combined this interaction technique with the idea of having a phone without hard keys

(already experimented with Simon by IBM & Bell South). This led to the possibility of having a larger screen and an enhanced interaction vocabulary with respect to its competitors.

In the same year, Microsoft introduced Surface (today renamed Pixel Sense [86]) an integrated hardware and software table-top system that enable multitouch and multi-user interaction on the same application. The system allows also to place and move physical objects on the table-top, for enhancing the interaction with tangible tags.

Besides the research on new technologies for enhancing the multitouch support, different work focused on the definition of a set of gestures that can be commonly accepted by users for executing different actions (e.g. undo-redo, object selection etc.). For instance, in [140] the authors conducted a user study on user-defined gestures, with the aim of finding a consensus on the interactive meaning of gestures.

Nowadays multitouch interaction is mature in its application. All major mobile device vendors created multitouch enabled smartphones, and all major desktop operating systems support multitouch interaction.

2.1.2 Remote-based gesture recognition

The release of the Nintendo Wii in 2006 [102] leveraged the gesture-based interaction from the research scope to the entire entertainment market. This game console introduced an innovative controller called Wii Remote (or Wiimote in short), which is equipped with a three axis linear accelerometer for sensing controller accelerations, an infrared camera for exploiting the remote as a pointing device, and a set of buttons

The IR camera senses the light coming from ten emitters, positioned into the Sensor Bar, another device placed near the screen. The controller has a shape that makes it suitable to be used with one hand, similar to a normal TV remote controller. It has no wires and it communicates with the console through a Bluetooth connection.

Such hardware configuration broke the static game-pad interaction, where the player has to stay motionless and control the actions pressing buttons. The user started to control avatars moving the remote, performing movements immediately replicated by her virtual counterpart. For instance, in a golf game, the player mimes the club control with the remote, and the power of the stroke can be associated to the movement speed, rather than to a bar displayed on a GUI.

Such kind of natural interaction opened the video game market (at least for Nintendo) to “casual gamers”, people who do not have a deep knowledge of video games and do not play videogames very often [65].

The Sony PlayStation 3 adopted a similar controller in 2010 [123]. The hardware configuration of this controller includes a three-axis accelerometer and a three-axis gyroscope, which enables sensing the also the angular speed of the movement. Differently from its Nintendo counterpart, the remote is equipped with a spherical RGB light emitter. The orb changes its colour in order to be recognizable for a camera in the surrounding environment, enabling a precise 3D position tracking.

A similar operation principle is shared by the Gyration “in air” mouse [52], a wireless device designed for the manipulation of 3D environments, exploited in [38] for creating a virtual orchestra game.

Such kind of remote controllers are not able to track the movements of the whole body if compared to e.g. Microsoft Kinect. However, they provide haptic feedback to the user, which is particularly useful when there is the need to manipulate virtual objects, offering a graspable counterpart in the physical world. For instance, it is simpler for the user to understand the aforementioned golf club metaphor if she has a physical object that represent the club itself, rather than performing an in-air gesture pretending to have something in her hands.

In addition, it is possible for the interface designer to exploit the physical buttons in order to mitigate the well-known Midas Touch problem [62], starting the gesture recognition only when the user presses a button, otherwise avoiding the movement tracking.

2.1.3 Image-based gesture recognition

Another option that is widely adopted in both research and industry solutions for gesture recognition is based on image analysis coming from RGB, infrared and depth cameras, which can be also exploited in combination.

One of the first examples for this kind of approach is CamSpace [29], a software tool that exploits webcams for turning any object into a game controller. Such generic approach comes at the price of losing possible haptic feedback coming from the system: the tracked object cannot be used to send output to the user.

A similar approach that led to a great change in the way people interact with games has been produced with the launch of Microsoft Kinect [87], released in 2010. The first version of the device was designed as a game controller for the Xbox 360 console. The device is composed by a bar placed on top of a motorized pivot, which has to be placed horizontally below or above the screen. It is equipped with an RGB camera, a depth sensor and an array of microphones.

A newer version with a similar configuration, improved in its hardware components, was created in 2011 together with the launch of the official Microsoft SDK for Kinect applications. It supported the gesture tracking at a nearer distance with respect to the previous version, which makes the sensor suitable for the usage in desktop settings.

The hardware configuration enables the tracking of the whole body and the recognition of facial expressions. The microphones allow the speech recognition.

The Kinect was the Microsoft's answer to Nintendo Wii, and with this new type of devices all game consoles in the market were equipped with gesture recognition devices.

An improved version of this successful device (the Kinect 2) is expected at the end of 2013 [137]. At the time of writing, only a set of specifications and a presentation video are available, but it should include an improved version of the hardware and a more powerful SDK. The new available features are a smoother joint tracking, the recognition of the hand state (open or closed), and the measurement of biometrical indices such as the heart rate or the muscle tension.

Another promising device that exploits such kind of approach is the Leap Motion sensor [80], which is a small bar to be placed under the screen of a desktop computer or a laptop. It is able to track the position of the fingers (or even sticks or pencils) with a precision of up to 0.01 mm. Such precision enables the creation of touchless interfaces with a robust 3D hand tracking.

Two infrared cameras and three infrared emitters compose the device, which tracks the hand position into a hemispherical surface of about one meter. It is currently available only for developers from October 2012, and it has been delivered to consumers on September 2013.

The image processing approach has a higher flexibility on the supported gesture types, since it is able to track the whole body or both hands. However, such configuration limits to the visual and audio channels the

possibility to provide output to the user, which is a clear disadvantage with respect to the haptic feedback that can be supported by a remote controller.

2.1.4 Floor devices

Considering again the entertainment field, it is possible to find another device type that was largely exploited for enhancing the playing experience: the so-called dance pads. Introduced by Konami with the game Dance Dance Revolution, they essentially are a huge directional pad with big arrow-shaped buttons that can be pressed with feet. This configuration allows the player to move following the music and the button sequence displayed on screen.

Even if such configuration is not able to track the body movements, the button sequence to be pressed with the feet forced user to dance.

A different kind of floor device is the Wii balance board, which is a rectangular feet panel that is equipped with two pressure sensors. It is mainly used in snowboard emulation games and in aerobic and yoga activities.

It is possible to find different work in literature that exploit such device for the interaction: a virtual reality controller [53], in combination with hand gestures for table tops [118] or 3D touch devices [70]. In addition, there are many example in literatures that use such devices for medical purposes (e.g. [49] and [145]).

In [9] the authors proposed Multitoe, an high-resolution frustrated total internal reflection floor, which is able to detect the shape and the shape of the users' footprints. Based on such shape and on the estimation of the pressure on the floor (using the brightness of the different footprint parts), it is possible to reconstruct postures and to interact with different widgets (keyboard, buttons, menu etc.).

An extension of this approach based on sensing the floor pressure is provided by GravitySpace [23], which exploits a high-resolution pressure sensitive floor for tracking the position of both furniture an multiple users in the room. The system is able to reconstruct the movements of each object and person on the surface analyzing the changes on the pressure image, and to provide a real-time 3D reconstruction of the room scene on the floor through a mirror metaphor.

2.2 Input modelling with formal approaches

In this thesis, we exploit a formal notation for defining a gesture meta-model. The idea of describing different types of input through a formal notation has been widely investigated in literature, using different formalisms. Such kind of research is recurrent when new type of input of devices are available to the mass market.

The first category considers Finite State Machines (FSM), which have been exploited not only for gestural interaction, but also for modelling input coming from standard input devices such as mouse and keyboard. For instance, Myers [98] defined a set of reusable interactors that encapsulate the interactive behaviour, hiding the details of the underlining window-manager events. The control part of such interactors, which managed the input coming from the different devices, was modelled with FSMs.

In the same years, Henry et al. [56] used FSMs for solving the problem of modelling non atomic actions on the UI, such as the drag and drop technique. Indeed, such kind of interaction is particularly tedious for developers, since they need to track the event sequence in order to implement describe the temporal relationship of the user's actions, which is close to the definition of a gesture.

The same problem has been addressed also in [120], where FSMs are exploited together with a set of intermediate layers between the input and the application. They separate the UI object picking and the sequence recognition from the definition of the UI behaviour. In this way, the authors were able to increase the reuse of tracking code, isolating it in a component library.

The FSMs approach for modelling the UI dialogues has been also integrated into widely adopted window toolkits, such as Java Swing, by Appert et al. [3]. In this work, the authors integrate FSMs inside the definition of the UI classes, in order to define in a single place the interaction code. Different FSMs can work together at the same time, in order to avoid the state explosion problem. One of the motivating examples was again the drag and drop interaction technique.

Jacob et al. [61] applied FSM to non-WIMP user interfaces: they separated two aspects of such kind of interfaces. The first one is the response to continuous input, which is managed by data-flow oriented variables. The second aspect is the connection among these continuous variables that can

change according to different discrete events. The different set of connections that are active among the continuous variables is described through FSMs.

Increasing the number of modelled dialogues, the number of states in the FSM definition explodes, and it may be difficult for designers to manage them. In order to mitigate such problems, in [14] rich interactions are defined using a hierarchical variant of FSMs, which includes sub-machines that are composed together for defining the UI behaviour.

Besides FSMs, context-free grammars or the equivalent push-down automata have been exploited for modelling the user input. We can remember here the work in [103], where the authors described a user interface generator that defined the accepted input through context-free grammars. The same formalism was the core of a formal UI specification defined in [15]. The authors exploited it not only for experimenting with different designs for the same UI, but also for proving the UI conformance to a set of guidelines.

The combination of different interaction modalities needed a formalism that was able to integrate different concurrent information sources. In [2], Accot et al. used Petri Nets for modelling low-level graphical interaction events. In addition, they showed how it was possible to create multimodal models starting from single-modalities, and composing them into one Petri Net. They exemplified the composition technique defining a bimanual interaction model for a direct-manipulation interface. A similar approach for modelling bimanual interaction has been proposed in the same years in [57].

More recently, Bo et al. [16] proposed an extension of Petri Nets that integrate the unification of typed feature structures [31]. Petri Nets provide a seamlessly definition of concurrent user's input, while typed feature structures support the specification of partial meaning and the integration different modalities, together with the specification of the constrains on such unification.

In this dissertation, we exploit a particular type of Petri Net, called Non-Autonomous [36], in order to provide the semantics of the temporal operators for our compositional model. Our approach is able to support and integrate different modalities, since we provide an extensible definition for the ground terms involved in the temporal expressions.

2.3 Declarative approaches for gesture definition

In this section, we review different approaches in literature that model gestures following a declarative and/or compositional approach. We differentiate between the work that addresses multitouch and the work that addresses full-body gestural interaction.

Through the analysis of the different work, we identified a set of problems that are addressed by the different notations, in order to define a set of requirements for our gesture meta-model. The following is the list of the requirements identified:

R1. *Temporal evolution.* The meta-model must describe the gesture temporal evolution. The developers should be able to define the behaviour of the user interface according to this temporal evolution, without the need of tracking explicitly the different stages of the gesture performance outside the model definition.

In the different work we analysed, such requirement was supported by a formal description of the gesture, through different notations: grammars [66], Petri Nets [7] or regular expressions [72,73].

R2. *Granularity.* Provided that a gesture may take seconds to complete, it must be possible for developers to define user interface reactions to partially completed gestures, not only to their complete recognition.

The granularity requirement was supported in the different work that proposed a compositional approach for defining gestures, where it was possible to combine different definitions for obtaining a new one. For instance, this was possible with grammars [66], rule-based [50,59,119] and regular expressions [72,73].

R3. *Separation of concerns.* The definition of gestures and the user interface behaviour must be separated, in order to allow the reuse of the same gesture model in different applications.

Independently from the different modelling approach and from the supported interaction device, the notations that raised custom events for notifying the gesture recognition supported such requirement [7,39,58,66,71,72,73,82]. This requirement it is not supported by most rule-based approaches, which usually define the behaviour in the rule body [50,59,119].

R4. *Multiple recognition devices.* The meta-model must support different recognition devices, abstracting from a particular recognition technology.

Such requirement is usually supported creating an abstraction layer between the recognition support and the actual device used for tracking the different features [7,39,58,59,82].

R5. *Parallel interaction.* The meta-model must handle the recognition of different gestures at the same time, in order to allow parallel interactions with the same application.

There are two main techniques for supporting the parallel interaction. The first one is allowing the simultaneous recognition of a set of gesture description, which are provided as a list. Such gestures are always matched against the updates coming from the recognition device. Such approach is common to the rule-based notations [59,71,119] or in custom events engines [7,115]. Another approach is to provide a composition operator that allow the developer to specify such parallel recognition as a temporal relationship among different gestures, which may be not available from the beginning [66].

R6. *Equivalent descriptions.* The same gesture can be performed in different ways (e.g. a pinch may be performed either with one hand or with two hands). The meta-model must support the definition of equivalent gestures.

Most of the work we analysed support this feature, providing a composition operator for specifying the different alternatives.

2.3.1 Multitouch

In this section, we discuss different work in literature that provided different notations for modelling multitouch gestures. We compare the different approaches against the set of elicited requirements.

At the end of this section, we summarize the support provided by all the approaches in Table 2.1.

Kammer et al. [66] introduced GeForMT, a formalization of multitouch gestures that aimed to fill the gap between the high level complex-gestures (such as pinch to zoom) and the low level touch events provided by different

toolkits. The description language, defined through an Extended Backus-Naur form grammar, is based on five different elements:

- the *pose function* describes the shape of the tracked touch;
- the *atomic gestures* describe the basic movements of the different touches (move, point, hold, line, circle and semicircle)
- the *composition operators* define composite gestures through a parallel or a sequential temporal relationship;
- the *focus* specifies the currently manipulated application object or objects;
- the *area constraints* defines the relative movements among the different touches (e.g. two touches that cross their positions).

The grammar productions of these five elements represent interactive gestures. With respect to the approach described in this dissertation, GeForMT is limited in scope, since it can be applied only to multitouch gestures. The composition operators do not provide a way for defining the equivalence of two different gesture definitions (e.g. through a choice operator).

In [50], Gorg et al. modelled multitouch gesture recognition through a labelled deductive system [44]. In order to define the interaction, the designer has to specify a set of rules that is able to recognize the expected sequence of touch-related events. Two different types of rules are exploited in this approach: the first is an *inclusion rule*, where the designer defines the expected sequence of events; the second is the *exclusion rule*, which specifies explicitly which sequences break the recognition.

Through the rule system, the designer has a fine-grained control on the recognition process, in particular when two gestures share the same common prefix, since it is possible to define priorities. However, exclusion rules make it difficult to compose gestures designed for different applications, since the developer has to find out if they inhibit the recognition of the composed gesture. In addition, the temporal evolution of the gesture is not stated explicitly, but it should be reconstructed from the rule set.

Regarding the separation of concerns, the rule body defines the reaction to the triggered events. This approach mixes the logic for the gesture recognition with the behaviour of the UI.

Scholliers et al. [119] defined Midas, an architecture for recognizing gestures according to a set of rules, which are matched against a set of input

facts using a logical rule inference engine. The rules are able to recognize multitouch gestures, taking into account different features such as the touch positions and speed, and the touch state (appear, move and disappear). Each rule has a prerequisite part, which defines the input fact pattern to be recognized, and an action part that specifies the UI behaviour.

The rules have different priorities in order to control the effects of the overlapping ones. The composition is possible through a set of temporal operators, which are able to compare the distance in time between two input facts. In this way, it is possible to define complex gestures asserting that the different components occurred with the specified temporal relationship.

With a rule-based approach, the designer has to figure out the temporal relationship between gestures reading and understating the rules. The approach proposed in this thesis describes this aspect more explicitly. Similarly to [50], the definition of the UI behaviour is contained into the same rules that matches a gesture.

In a follow-up work, Hoste et al. [59] extended the Midas approach for describing multimodal interfaces. Mudra (which is the extension name) is able to unify the input stream coming from different devices, which exploits even different modalities. It provides the designer with a way to define both the low-level handling events, and the high-level rules that combine them into a unique software architecture. The rule language has been extended for supporting facts coming from e.g. voice and hand movements, but its structure still mix the gesture recognition and the behaviour definition. Even if we do not explore deeply the multimodality aspect in this thesis, we demonstrate with the application in section 5.4.4 that is possible to combine the gestural and the vocal modality in GestIT.

Khandkar et al. [71] proposed GDL (Gesture Description Language), which separated the gesture recognition code from the definition of the UI behaviour. The description language focus on multitouch gestures, and it is defined through three components: the gesture *name*, the code for the gesture *validation* and a *return* type, which represents the data notified with a callback to the application logic, containing all the relevant information (e.g. the entire sequence of touch positions, the number of touches etc.).

The approach is compositional, since it is possible in the validation part to reuse different gesture recognizers. However, since each recognizer represents simply a boolean function, it not possible to define all the

temporal sequences that we can define with GestIT, but it is possible to provide different equivalent version of the same gesture. In addition, once the composed gesture is defined, it is not possible to register a handler to its sub-parts, since the only event that is notified is the completion of the entire gesture.

GISpL [39] proposes a JSON-based syntax for describing gestural interfaces and it supports different interaction modalities such as multitouch, digital pens, regular mouse (or mice), tangible tokens and mid-air gestures. The syntax defines how to monitor a set of features observed in the input stream, such as e.g. the count of different objects in a region, the matching accuracy between a predefined path and the one travelled by a given input object etc. Each feature can be related to single or multiple sources.

When one among the different gestures is recognized, the target application receives a notification in the form of a specific event. It is possible that more than one gesture is detected at the same time. The approach enables the reuse of the gesture definition in different applications, and the separation between the gesture recognition and the application behaviour aspects. However, the language does not provide compositional operators. Therefore, there is no way to create complex gestures describing the temporal relationships among simpler ones.

The maturity of the multitouch support on different devices makes this interaction suitable to be adopted in safety-critical settings. However, in such environments it is necessary to prove a set of properties of the UI, such as invariants or constraints on the behaviour. In [7], Arnaud et al. provide a formalization for multitouch gestures, which can be exploited in order to prove a set of UI characteristics for employing them in a plane cockpit. The proposed architecture is based on the Interactive Cooperative Objects [99], a formalism that exploits an object-oriented description of the structural and static aspects of the UI, while it exploits Petri Nets for describing the dynamic behaviour. The instantiation of such kind of objects for multitouch interaction defines a set of layers that are similar to the ones proposed in this thesis for supporting generic gestures: the first level creates and abstraction of the low-level device events (called *low-level transducer*). Such events are passed to the second layer, between the device and the

application, which contains a set of gesture recognizers that raise high-level events according to the successful completion of a gesture (e.g. pinch or tap).

Finally, the application reacts to such high-level events. In the work by Accot et al., Petri Nets are used to define directly the behaviour of an interaction object that recognizes a gesture, therefore the expressive power of the modelling language can be considered the same, since it is possible to directly “reuse” the Petri Net definition of the composition operators we propose in this thesis.

However, the approach is affected by the granularity problem: the high-level events are raised only when a gesture completes successfully, without any intermediate notification. In addition, all the interaction objects that represent the different gesture receive the low level events in parallel, and this limits the possible temporal relationships that can be defined among the different gestures.

In Table 2.1, we summarize the comparison of all the approaches discussed in this section against the set of requirements we identified. None of them satisfied the full set of requirements.

	Temporal evolution	Granularity	Separation of concerns	Multiple recognition devices	Parallel interaction	Equivalent description
Kammer et al. [66]	✓	✓	✓	✗	✓	✗
Gorg et al. [50]	✗	✓	✗	✗	✓	✓
Scholliers et al. [119]	✗	✓	✗	✗	✓	✓
Hoste et al. [59]	✗	✓	✗	✓	✓	✓
Khandkar et al. [71]	✗	✗	✓	✗	✗	✓
Echtler et al. [39]	✗	✗	✓	✓	✓	✓
Arnaud et al. [7]	✓	✗	✓	✓	✓	✓
Kin et al. [72,73]	✓	✓	✓	✗	✗	✓

Table 2.1: Comparison of different multitouch gestures definition approaches in literature

2.3.1.1 Proton++

In this section, we analyse the gesture description that, to the best of our knowledge, is the closest one to the approach described in this dissertation.

Proton++ [72,73] is a multitouch framework allowing developers to declaratively describe custom gestures, separating the temporal sequencing of the events from the code related to the behaviour of the UI.

Multitouch gestures are defined as regular expressions, where literals are identified by a triple composed of:

1. The event type (e.g. touch down, move and up)
2. The touch identifier (e.g. 1 for the first finger, 2 for the second etc.)
3. The object hit by the touch (e.g. the background, a particular shape etc.).

It is possible to define a custom gesture exploiting the regular expression operators (concatenation, alternation, Kleene's star).

Figure 2.1 shows an example of gesture definition using Proton++. The gesture is a simple two-hand scale (pinch) gesture. The different colours in the lower part of the figure correspond to the different gesture parts in the expression.

The entire expression is built composing touch events represented in the E_T^O form, where E is an event (D for touch down, M for touch move and U for touch up), O is a touchable object (in our example s is the star, while a can be any object) and T is the touch identifier (simply an integer). It is possible to create the gesture definition composing such literals through the usual regular expression operators.

Scale

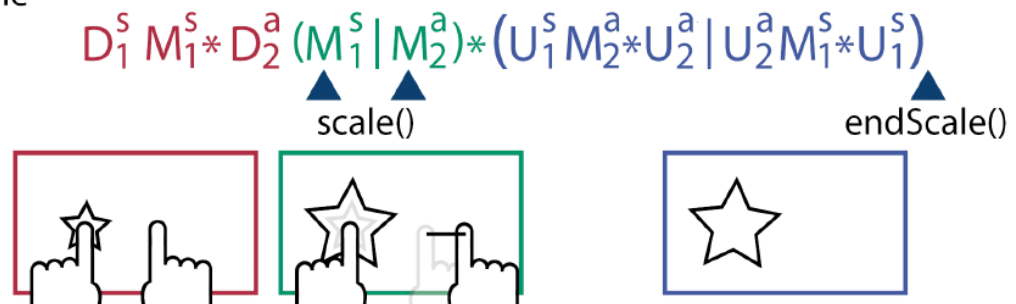


Figure 2.1 An example of gesture definition with Proton++

The red part in Figure 2.1 describes starting part of the gesture, where the user touches the screen with two fingers. After that, she can converge or diverge the hands with an iterative movement of both fingers (the green

part in Figure 2.1). The gesture ends when both fingers are lift from the screen (the blue part in Figure 2.1).

The underlining framework is able to identify conflicts between different composed gestures and to return their common longer prefix in order to let the developers remove the ambiguous expression or assign different probability scores to the two gestures.

The runtime support receives the raw input from the device, transforms it into a touch event stream that is matched against the defined regular expressions.

When one or more gestures are recognized, the support invokes the callbacks associated to the related expressions, selecting those with higher confidence scores (assigned by the developer in case of conflict between the expression definitions at design time).

An improved version of the framework (presented in [73]) included also the possibility for the developer to calculate a set of attributes that may be associated to an expression literal. For instance, it is possible to associate the current trajectory to a touch move event, and let the framework raise the associated events (read recognize the literal) only if its movement direction is the one that the designer specified (e.g. north, north-west, south etc.).

Other examples of such attributes are the touch shape, the finger orientation etc. In Proton++ it is possible to define the custom gestures through a graphical notation (called tablature), which has been demonstrated to be more understandable for the developers if compared with normal code.

Since this language shares different features with GestIT, such as the separation between the gesture description and effects, and the possibility to create gesture descriptions composing ground terms trough a well-defined set of operators, we compare its expressiveness against our approach in section 3.4.

2.3.1.2 Selection ambiguity

The framework described in [120] does not take into account gesture modelling, but it focused on the input uncertainty problem. The same problem affects in particular the compositional approaches when two different gestures, which have a common starting prefix, are connected through a choice operator (it is possible to execute either one or the other). Starting from the fat finger problem in multitouch interfaces [42], the

authors propose to assign a probabilistic score to the input meaning (read the selected object). For each possible interpretation of the input, the different interactors send a notification of the events to an intermediate layer between the interface view and the behaviour, called *mediator*. Such component is in charge of handling the uncertainty according to different policies. Once the mediator is able to pick one among the different actions, it performs the selection and only one among the possible interpretation is sent to the regular interface behaviour definition.

In this dissertation, we propose a different solution for this problem in section 7.3, which exploits the possibility to split a gesture into different sub-parts. Such solution is able to seamlessly provide a support not only for the developer that needs a way to manage the uncertainty, but also for providing guidance to the users during the gesture performance, which has been demonstrated effective for learning the interaction vocabulary [10].

Therefore, we add to our set of requirements the following:

- R7. *Selection ambiguity.* The recognition support must provide means for identifying or managing the selection between two different gestures that shares the same initial sequence.

2.3.2 Full-body

The compositional and declarative modelling techniques have been scarcely applied to full-body gestures in literature. The state of the art abstraction for creating gestural interfaces with depth cameras is based on tracking the sequence of skeleton frames [115,138]: the sensor driver, according to images captured by the cameras, updates the number of skeletons and the position of their joints, using inverse kinematic techniques [122,147].

The documentation for such programming toolkits presents such abstraction as the usual instantiation of the observer-pattern: the developer has to register to an event (the skeleton position change) and then the application has to react to such change. Therefore, since there is nothing new for the window managers, it is possible to build an intermediate layer between the device drivers and the application logic in order to uniform the new interaction devices to the existing input techniques, allowing the developers to map them to mouse or keyboard events [128]. This has the obvious advantage of reusing existing applications with different interaction devices with little effort.

However, gestural interaction is intrinsically continuous, and the event-handlers for the skeleton position change are filled by code that tries to filter the notifications that do not correspond to the expected temporal sequence, as we better detail in Chapter 7.

The research community and the device vendors have obviously identified such problem and different solutions.

The first one is offering an extension point for the device driver SDK, where it is possible to concentrate some recognition code that defines a new custom gesture. When such gesture is recognized completely, the library raises a custom event and the developer can attach different handlers in different points of the application code. Such solution does not allow developers to provide intermediate feedback during the gesture performance, since the complex gesture cannot be decomposed in smaller parts, violating our *granularity* requirement.

Such approach is adopted in NITE [115] but, given the amount of time needed by the user for completing a gesture, is not sufficient for providing an adequate support to developers. It is possible to provide a simple explanation for this point simply considering the sample code that NITE provides to developers for demonstrating such SDK feature. The sample application recognizes a circular hand motion and changes the background colour of the screen when the user completes the movement. Even in this simple case, the application has to provide an intermediate feedback during the gesture execution, otherwise the user is not able to understand if it is tracking her movements correctly. Therefore, the sample provides such feedback showing a circle and a line that represents its radius on the application UI. When the user moves her hand, the radius rotates around the centre of the circle. If the user does not perform the gesture correctly, the radius returns in the initial position.

The sample code shows, even in this simple case, that the single-event approach is not suitable for gestures: the screen-background change is attached to the custom event, raised at the end of the gesture. However, in order to provide the intermediate feedback, the sample code tracks the hand position again, since it is not possible to access the inner components of the custom gesture. However, such solution allows the recognition of different gestures at the same time and, raising the same event for different gestures, it is possible also to define the equivalence between two gestures.

A more effective solution is presented in [58], where the authors provide a declarative syntax for defining complex gestures. The different gesture

recognition devices are exposed as a data stream, which is analysed by the AnduIN [75] processing engine. The custom events are defined with an SQL-like syntax that create triggers for the sequences that are compliant to a specific selection rule. Even if the declarative syntax enhances the reuse and the possibility to inspect the gesture definition, the notification of the gesture recognition is still based on a single event. Therefore, in order to provide intermediate feedback, the developer is in charge to define a set of recognizers for the gesture sub-parts and to coordinate them in the recognition handlers.

Such rule-based approach guarantees the parallel recognition of different gestures and also the possibility to define equivalent gestures. In addition, the data stream abstraction allows to support different recognition devices.

Another approach that tries to lower the complexity for gesture definitions consists in providing an abstraction layer, which hides the complexity of the underlying machine-learning algorithms that perform the recognition. An example of this approach is GART [82], where the developer can provide a set of training examples for different sensors in order to define the gesture vocabulary.

The main difference with the approach proposed in this thesis is that classifiers are bound to raise the events only when the whole gesture is recognized. Therefore, such approach is good for gestures that have a limited duration in time. In addition, such approach does not support neither the definition of gesture composition nor the temporal sequencing.

Such approaches usually allow the recognition of only one gesture at time. However, it is possible to map multiple gestures on the same event, providing the mechanism for expressing the equivalence of gestures. In addition, a classifier can be trained with features coming from different devices.

Table 2.2 summarizes the comparison of the different modelling approaches against our requirement set. As it is possible to see, the approaches for full-body gestures provide a narrower support for the required features if compared with multitouch work.

	Temporal evolution	Granularity	Separation of concerns	Multiple recognition devices	Parallel interaction	Equivalent description
NITE [115]	✗	✗	✗	✗	✓	✓
Hirte et al. [58]	✗	✗	✓	✓	✓	✓
Lyons et al. [82]	✗	✗	✓	✓	✗	✓

Table 2.2 Comparison of different full-body gestures definition approaches in literature

Besides the obvious exploitation of gesture recognition for games or different gestural applications (we show different examples in literature while discussing common control gestures in section 4.1), it is worth pointing out here that such devices have a wider impact on HCI with respect to other input techniques. Indeed, as discussed for instance in [130], it is possible to exploit such hardware to differentiate the user’s feedback according to their personality (introvert or extrovert), inferring it through a user’s posture analysis. The authors proved that such empathetic feedback enhanced the experience in video games.

The same configuration can be also exploited in ubiquitous settings. In [131], Tan et al. provided an off-the-shelf solution for tracking the user’s affective state, that can be employed by intelligent user interfaces for modifying the feedback and/or the content according to her current feelings.

The gesture description discussed in this thesis may be employed in such configuration for instrumenting the posture recognition with a human-understandable notation.

2.4 Model-based approaches for User Interfaces

Model-Based User Interface design is sub-area of the Human-Computer Interaction research field that aims to lower the complexity for the design of an interactive system. This objective is achieved creating a set of abstractions for the design and the development of a User Interface.

The proposed approach for modelling gestures has been integrated into MARIA [111], a state of the user interface modelling language that belongs to this research field.

2.4.1 Historical Background

During the last two decades, the research in this field has tried to deal with the evolution of the technological settings and the consequent changes and challenges in the development and design of UIs.

In [111], three generations of approaches are identified. The first generation focused on Graphical User Interfaces (GUIs).

In this category, MIKE [104] attempted to leverage the UI development to non-programmers, introducing a command syntax for defining the interface functionalities. After the command list was created, the tool generated the UI, and it was possible to edit the result adding descriptive information.

Jade [146] is a tool that automatically created input dialogs out of a layout independent content description, created by programmers. Combining this specification with a layout database created by artists, the tool was able to generate the graphical dialogs.

In the same category, we can remember ITS [139], which defined a four layered architecture for defining interactive systems. The different layers are the application back-end functionalities (action layer), the content without style information (dialog layer), the layout rules for choosing the appropriate interaction technique (style rule layer) and the dynamic changes in the interface (style program layer).

Humanoid [129] created an abstract description that allowed the declarative specification of both presentation and behaviour.

Finally, UIDE [41] is a development environment able to exploit models in order to generate automatically the implementation of the UI and also to derive data schemas for databases and help for the application usage.

The second generation defined the shift of focus from the graphical modality to the interaction semantics, using task models in order to describe the actions that users have to perform in order to achieve a specified goal. This trend was driven by the psychological theory on how people perform tasks. Indeed, in [30] the execution was explained in terms of GOMS, which stands for Goals, Operators, Methods and Selection rules. Goals refers to the intended user's targets, the Operators are actions performed in order to achieve a given goal, the Methods are sequences of operators and sub-goals that allow accomplishing a goal, while the Selection rules drive the execution of a certain method when more than one option is available.

In [63] Johnson et al. described ADEPT, an environment for prototyping user interfaces. The tool supported the creation of a model of the tasks that

the user and the system have to perform jointly, together with a UI prototype editor that take as input the task model. The designer can refine the user interface model in order to create the application prototype.

Van der Veer et al. [134] created a conceptual framework for the design of an interactive system, which envisioned a three staged modelling methodology. At the first step, the designer creates a first task model (Task Model 1) from the domain knowledge and the work practice. This model has to be refined with the specification of task decomposition procedures, task allocations to people and technology, communication structures and management procedures (Task Model 2). After that, the designer should create a User's Virtual Machine (UVM) that represents the knowledge of the system that is relevant from the user perspective, without hardware or implementation details. This abstraction has to be iteratively validated through a prototyping phase, in order to obtain a UVM specification suitable for creating the system.

Another example can be found in [114], which describes the ConcurTaskTrees notation. It allows the designer to specify the task with a graphical tree-shaped notation, decomposing high-level tasks (abstract) down to atomic actions that can be performed by the user, the system or by an interaction between them.

The various tasks are connected through operators in order to specify their temporal relationship. It is also possible to specify which kind of objects are manipulated while performing actions.

Since the temporal operator of this task modelling language provided the inspiration for the ones we exploit in our compositional approach for the definition of a complex gesture, we describe it in detail in a dedicated section (2.4.3).

As explained in [97], although such approaches for the development of interactive systems were promising, they did not found a wide acceptance (aside for task modelling), because they were generally affected by the unpredictability of the final result due to a set of generation heuristics. In addition, the standardization of the vocabulary of GUI toolkits lowered the importance of having specific models. However, a new generation of model-based approaches is now pushed by the increasing availability of a large number of different devices, each one with specific characteristics and features, which creates the need for device-independent user interface specifications [97]. In this thesis, we try to provide a unified approach for the different devices that enable the recognition of a gesture.

The third generation of model-based approaches is currently trying to take into account such issues, providing models and languages able to support multi-device development, and the desired level of control to designers. The effort is generally on the definition of User Interface Description Languages (UIDL) to describe such models. The best-known projects in this field are XIIML [116], UIML [1,55], UsiXML [81], Teresa XML [96], MARIA [111] (which we extend in this dissertation in Chapter 6) CAP3 [11] and, to some extent, XForms [20].

XIIML [116] is the acronym of eXtensible Interface Markup Language, which is an extensible language based on XML developed by RedWhale Software. The language aims to create a framework for supporting the entire UI engineering process (design, operation, evaluation). The XIIML vocabulary contains a collection of interface elements categorized into an extensible number of components, which should be in a relatively small number. Such components are:

- user tasks, which represent definition of activities with a hierarchical decomposition
- domain objects, which represent a collection of data objects and classes
- user types, which represent a hierarchical categorization of the various user profiles
- presentation elements, which represent the hierarchy of abstract interaction elements (such as windows, buttons, sliders etc.)
- dialog elements, which are actions that are available to the users of an interface (e.g. clicks, gestures, voice responses etc.).

These components are linked using relations, which are definitions or statements for the runtime operations on the UI. The language itself does not specify the relation semantics, but the specification is left to each single application. The interface elements have a set of attributes, which characterize better their role.

The User Interface Markup Language [1,55] (UIML) is a XML-based language that addresses the multi-device problem, with the definition of UI elements that are independent from the target device, delegating the mapping between elements and their rendering to style-sheets. The runtime behaviour of the elements is described through events, which can be either local (that affect only the interface elements) or global (that affect also the application back-end). In a UIML document, the UI is described through the following sections: the structure (a list of abstract part of the UI), the

style (a list of properties for rendering UI parts for a given device), the content (text, images and data contained into the UI), the behaviour (a set of rules that define how the UI reacts to actions), the logic (the application programming logic for connecting the UI with its back-end), the presentation (a mapping list between the UIML vocabulary and the target implementation language constructs).

The USer Interface eXtensible Markup Language [81] (UsiXML) is a XML-compliant language that employs different models for describing various UI aspects in different context of use. The set includes the following specifications: tasks (through an extension of the CTT [114] language), abstract UI (a description of the UI elements independent from any particular device or modality), a set of concrete UIs (a description of the UI elements that is modality dependent, e.g. graphical, vocal, 3D etc.), domain model (description of the classes of objects manipulated by the UI), a set of mappings (declaration of inter-model relationship between elements semantically related), a model of the context of use (properties regarding the current end user, platform and surrounding environment), and a set of transformations (a set of graph rewriting rules depending on attribute conditions).

UsiXML has been exploited in different applications, and is maintained with a dedicated project [133]. Among the different tools that support the different models, we can remember here UsiComp [46] an environment that support both the design and the generation of applications based on OSGi [106] services. The environment can be extended for exploiting other meta-models that describe different aspects of the application. Such extensions enter in the final application generation through a set of custom transformations (model-to-model or model-to-code).

In [96] is described TERESA, an XML language with the associated tool that is able to support the definition of UIs with different levels of abstraction (see section 2.2.2).

At the abstract level, the elements can be of three types: interactors (single interaction objects), composition operators (that groups together interactors logically connected) and presentations (a set of interactors and composition operators presented to the user at the same time). The interactors belong to different classes according to their interaction semantics (e.g. edit, control, selection, only-output etc.). Each target platform (graphical desktop, mobile, vocal, multimodal etc.) refines this

abstract representation introducing modality dependent implementations of the interactor classes.

XForms [20] is an attempt to create a new generation of web forms that can be integrated into different markup languages, exploiting the model-view-controller pattern. The relevant point for this discussion is that, being tailored for being embedded into other XML markups, the view layer cannot rely on a specific interaction modality, e.g. the forms can be embedded either into HTML or Voice XML. Therefore, the input items should focus more on the interaction semantics rather than the appearance as currently happens in HTML. Examples of the interaction objects included in XForms are the following: *select* (choice of one or more items from a list), *trigger* (that activates a defined process), *output* (display-only form data), *secret* (entry of sensitive information etc.). The XForms vocabulary represents an attempt for a device-independent specification of UI controls.

CAP3 [11] is a user interface modelling language, designed to be integrated in a user-centered design process. The language contains both structural and behavioural specifications, combining such aspects into a model that can be exploited by different stockholders while discussing the design of interactive applications. In order to express the relationships with other models of the same applications, it contains explicit references to external models representing different aspects of the system, such as the domain, user and context models.

Nowadays, there are different initiatives that aim to create international standards for adopting the model-based approach into industrial settings. The ANSI/CEA-2018 is a standard for the specification of task models [117], published in November 2007, together with an XML interchange syntax. The task definition is provided through a hierarchical structure, the sub-tasks are by default executed sequentially, but it is also possible to define partial orderings. Tasks are also optionally associated to input/output parameters and pre/post execution conditions.

The World Wide Web Consortium has a working group for providing a standardization of the different languages related to the Model Based User Interface approach. The Model Based User Interface Working Group (MBUI-WG) aims to provide a standard definition for task models and the abstract user interface level that, according to the CAMELEON [27] reference framework, provides a description of the UI that is independent from the current device and interaction modality.

As a second step, the group aims to provide different specifications of UI languages that describe the interface for a given set of homogenous devices (the concrete level) and a way for representing context-dependent adaptation rules.

The working group produced a public working draft for the task model specification in August 2012 [109].

2.4.2 The CAMELEON reference framework

In this section, we introduce the CAMELEON [26,27] reference framework, that provided the theoretical background for different model-based languages for user interfaces (MBUI), and in particular for the definition of MARIA [111], which we discuss more in detail in Chapter 6, since in this dissertation we define an extension for supporting gestural interaction, which can be therefore inserted into the broader scope of MBUI approaches.

The CAMELEON Reference Framework offers a unified representation of the models, methods and processes for creating multi-target user interfaces.

The UI context of use is defined along three dimensions: the *users* that are intended to use or effectively use the system, the *platform* that is the hardware and software configuration of the interactive system, and the *environment* that specifies the physical conditions where the interaction occurs. A multi-target UI is able to support different contexts of use. The reaction to a context change in a multi-target UI is called *adaptation*. If the adaptation is performed preserving usability, the UI is *plastic*. Preserving UI plasticity for cross-platform design and for context-aware applications is currently one of the main challenges in this research field. Indeed in [121], the authors demonstrated through a case study that the overall UI quality (in terms of ergonomic criteria) increases when the UI plasticity is preserved. This happens since plasticity has an impact on a set of usability criteria that influence positively different usability aspects.

Given the increasing number of devices that people uses in their everyday life, the engineering techniques able to preserve plasticity from the early phase of the applications development have an impact on different fields of Computer Science [28], and their number will increase in the future. Besides HCI, fields such as Software Engineering (e.g. aspect-oriented programming, model-driven-engineering) and Artificial Intelligence can provide effective solutions for this problem.

With respect to the application modelling, the CAMELEON framework distinguishes three kinds of models. The *ontological models* are defined as the meta-models, independent from any interactive system, which are able to describe the concepts and their relationship involved in multi-targeting. The *archetypal models* are instantiations of the ontological models and represent an interactive system that deals with a given domain. The *observed models* are executable models that support the adaptation process at run-time.

The ontological models can be of three different types:

1. *Domain Models*, which support the description of a domain-related concepts and tasks.
2. *Context Models*, which support the description of the context (user, platform and environment).
3. *Adaptation Models*, which support the description of the reactions in case of context change and the commutation process.

After the identification or the specification of the needed meta-models (e.g. UML class diagram for describing the domain-concepts, CTT for describing the tasks etc.), it is possible to define various configurations that describe a specific interactive system using the different meta-model constructs. This instantiation of the ontological model produces different archetypal models, which represent the application for classes of potential devices (e.g. the archetypal model for a multi-touch mobile device applies for the iPhone, Samsung Galaxy S4 etc.). The observed models are exploited at runtime in order to perform both the UI execution and the context switches.

The design-time phase creates a set of executable UIs, each one targeted to a particular archetypal model configuration, called *initial model*. The process envisions the creation of a set of different *transient models*, produced using different *operators*. At the end of the process, we have the *final context-sensitive interactive system*.

The framework specifies four different transient models, with a decreasing level of abstraction:

- *Concepts and Tasks model*: description of the concepts and the tasks that is produced by the designer for a particular context of use.
- *Abstract User Interface (AUI)*: user interface description that is independent with respect to the device and the interaction modality.

- *Concrete User Interface* (CUI): user interface description that is abstract with respect to the technology used for the implementation.
- *Final User Interface* (FUI): the final implementation of the user interface, expressed in source code.

The operators transform a model into another one. Such transformation can be implemented in different ways: with a completely automatic process, without any automatic support (completely defined by a designer), or with a semi-automatic solution. The latter process envisions the automatic creation of a target model draft, with an intervention of a designer, which modifies it in order to achieve the desired result. On one hand, a full automation leads to a very quick development process. However, this produces only standard solutions that are not tailored for the specific application, otherwise the designer should specify a huge number of details that invalidates the model convenience. On the other hand, a completely manual solution has a high development cost that would make the multi-targeting expensive. It is a general opinion that a good balance between the automation and human intervention is the best solution for this problem.

Operators can be classified according to the abstraction level of the models involved in the transformation process:

- An operator performs a *vertical* transformation if the source and the target models are at different levels of abstraction. The top-down approach (from a higher level to a lower one) is called *reification*, while a reverse engineering step is called *abstraction*.
- An operator performs a *horizontal* transformation if the source and the target models are at the same abstraction level. If it involves two different targets, it is called *translation*.

In the run-time phase, the designed UIs and the runtime infrastructure cooperate in order to support the adaptation. The process consists of three steps that include the recognition of the situation, the computation of a reaction and the execution of the reaction.

The recognition of the situation needs the ability to sense the context of use (or at least the part that are interesting for triggering a change), to detect context changes (comparing the sensed attributes with the previous values) and to identify context changes (classifying the change into the modelled categories).

2.4.3 ConcurTaskTrees

We dedicate this section to the description the ConcurTaskTrees [114], a task modelling language that provides a set of temporal operators, which inspired the ones we defined for composing gestures. In addition, it provides the task-level language that is exploited by MARIA [111], the UIDL we extend in this thesis.

The ConcurTaskTrees model hierarchically different task: at the top level there is an abstract task that represents the whole application, which is decomposed into a set of subtasks until the desired level of detail is reached, building a tree. Four types of tasks exist: *user* (that involve only the human user), *system* (that involve only the system), *interaction* (that involve both the system and the user), and *abstract* (used for grouping together task of different type at the intermediate levels).

At each level of the tree, it is possible to connect two tasks using the following temporal operators, reported here in order of priority:

- *Choice*. It is possible to choose one of the connected task. Once one task is selected, it is the only one that can be performed, while the other is disabled.
- *Concurrency*. The connected tasks can be performed concurrently, without any specific constraint.
- *Order Independence*. The connected tasks can be performed in any order. However, once one of them is selected, it has to be completed before executing the others.
- *Synchronization*. The connected tasks can be performed concurrently, but they have to synchronize in order to exchange information.
- *Disabling*. The first task is deactivated when the second is performed.
- *Suspend-Resume*. The second task interrupts the first one. When it is finished, the first can be reactivated from the state it was before the interruption.
- *Sequential Enabling*. The first task enables the second when it is finished.
- *Sequential Enabling with information passing*. The first task enables the second when it is finished, passing some information.

The modelling language defines also two operators that are applied to a single task. The *optional* operator indicates that the execution of a task is

optional. The *iteration* operator re-enables the beginning actions of a task when it is completed.

2.5 Non-Autonomous Petri Nets

Before introducing the meta-model, we briefly summarize the formal notation we exploit for defining the semantics of the gesture meta-model entities. As we discuss in detail in Chapter 3, we used Non-Autonomous Petri Nets since they allow us to define easily a parallel interaction. In addition, they offer a straightforward way for modelling the reaction to events that are external with respect to the application logic, such as the data coming from gesture tracking devices.

A Petri Net is a bipartite graph consisting of two types of nodes: transitions (represented as black rectangles) and places (represented as circles), which are connected by directed arcs. A place contains a positive number of tokens and the state of the net is represented by the distribution of the tokens among the places.

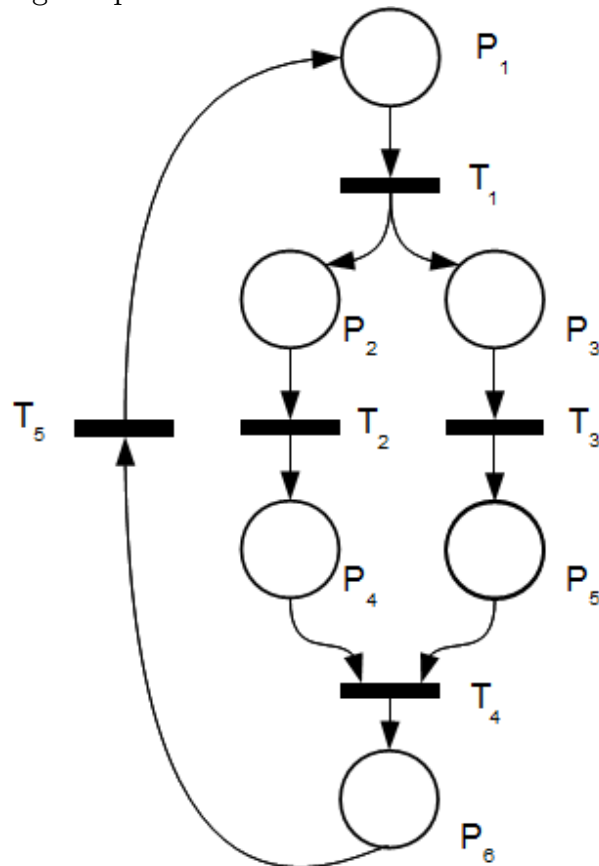


Figure 2.2 An example of Petri Net

The Figure 2.2 shows a Petri Net example, which contains six places, represented by the set $P = \{P_1, P_2, P_3, P_4, P_5, P_6\}$, and five transitions, represented by the set $T = \{T_1, T_2, T_3, T_4, T_5\}$. The arcs are represented by the following set of pairs: $A = \{(P_1, T_1), (T_1, P_2), (T_1, P_3), (P_2, T_2), (P_3, T_3), (T_2, P_4), (T_3, P_5), (P_4, T_4), (P_5, T_4), (T_4, P_6), (T_5, P_1)\}$.

When all the places that are connected to a given transition contain at least one token, the transition fires, withdrawing a token from all the incoming places and adding one token to all the out-coming ones.

Figure 2.3 shows different sample conditions for firing the same transition. In the example (a), each one of the incoming places (P_1 and P_2) contains exactly one token, while no one of the outgoing places (P_3 , P_4 and P_5) contains any token (upper part). In this situation, the transition T_1 fires and each one of the outgoing places receives a token (lower part).

In the example (b), the initial situation is different: before firing the transition, the outgoing place P_3 already contains a token. Therefore, after the transition firing, P_3 contains two tokens.

In the example (c), the incoming place P_1 contains two tokens, while P_2 contains only one token. When the transition fires, only one token is removed from each one of the incoming places, therefore after the transition P_1 contains one token.

In the example (d), it is not possible to fire the transition since there must be at least one token in each one of the incoming places, while in this case P_1 is empty.

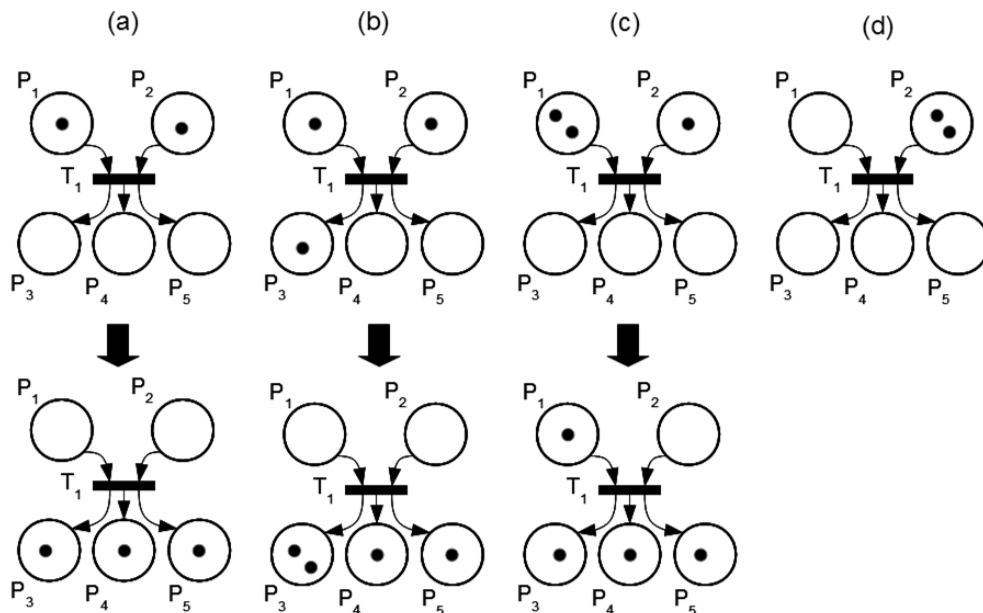


Figure 2.3 Transition firing in Petri Nets from [36], p. 3.

In Non-Autonomous Petri-Nets, the transition firing is controlled not only by the presence or absence of the token in the incoming places, but also by an external event. The transition fires if there is at least one token in all the incoming places *when* the external event occurs. External in this case means that the Net is able to react to changes that are not directly connected with its internal state.

For instance, we can consider the Petri Net in Figure 2.4, which models a semaphore. It contains one place for each one of its states, namely green, yellow or red traffic light (respectively the G, Y and R places). One of the three lights is on if the correspondent place contains the only token in the Petri Net. If we consider the previously discussed basic version of Petri Nets, the different transition would continue to fire indefinitely changing the light colour as soon as the token enters into the one of the three places. Instead, we would like to model the fact that the semaphore waits for a predefined amount of time before changing the light colour.

In Non-Autonomous Petri Nets it is possible to model such situation specifying that a transition fires when there is at least one token in all the incoming places and the predefined amount of time has passed. Obviously, the time is an external entity for the Petri Net, which has no control on it. The Petri Net receives a notification when one of the specified external events occur. In our example T_1 fires after 65s after the token arrives in G, T_2 fires 5s after the token arrives in Y and T_3 fires 65s after the token arrives in R.

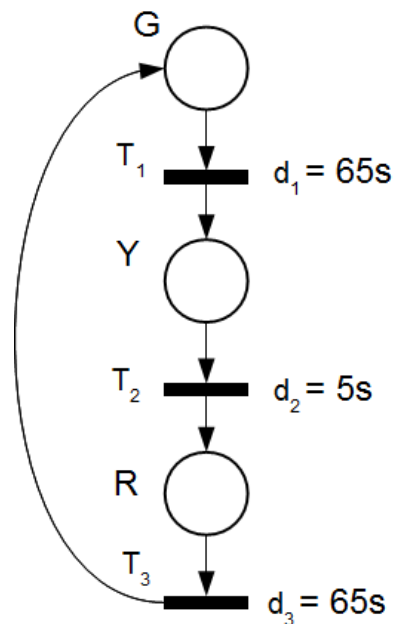


Figure 2.4 Non-Autonomous Petri Net for a traffic light, from [36] p.4

In our case, we use the internal state of the Net for modelling the gesture recognition phases, which are driven by the data received by the gesture recognition device.

An extensive description of Petri Nets and their properties can be found in [36].

Chapter 3

Gesture Meta-Model Definition

This chapter provides the formal definition of the meta-model we use for defining gestures. It can be applied to different recognition platforms and for different types of gestures.

We start from the specification of the ground-term semantics. After that, we define a set of composition operators that allow defining complex gestures in a declarative and compositional way.

Next, we apply the meta-model to different recognition platforms: namely multitouch and full-body. For each one of them, we provide the specification of a set of commonly used gestures.

This chapter is an extended version of the work discussed in [125].

3.1 Meta-Model Definition

In this section, we theoretically define our gesture description meta-model. Such meta-model is abstract with respect to a specific gesture recognition support, which means that it is possible to instantiate it for different devices (e.g. multitouch screens, body tracking devices, remotes etc.).

We start from the definition of the basic building blocks (ground terms), which represent the set of basic features observable through a specific device. Composed terms represent complex gestures (that can be further decomposed) and they are obtained connecting ground or composed terms through a well-defined set of composition operators.

The definition of the UI behaviour can be associated to the recognition of basic or composed gesture definition. Once the Petri Nets for a basic building block and for all the composition operators have been defined, the designer can create complex gestures through expressions of basic building blocks and/or complex gestures composed through the set of operators. The actual Petri Net for the complex gesture is derived visiting bottom-up the complex

gesture expression definition and can be executed by a run-time library that we introduce in section 5.2.

3.1.1 Basic Building Blocks: Ground Terms

Ground terms of our language are the basic building blocks of our gesture description model, since they cannot be further decomposed. They are defined by the events that developers currently track in order to recognize gestures. Ground terms do not have a temporal extension, though their values may be obtained by computing a function of the raw sensor data (the current gesture support). For instance, if we are describing a gesture for a multitouch application, the ground terms are represented by the low-level events that are available for tracking the finger positions, which are usually called touch start, touch move and touch end.

Besides, for creating full body gestures, the current recognition devices and libraries offer means for tracking specific skeleton points, such as hands, head, shoulders, elbows etc.

As happens for multitouch gestures, also full body ones are recognized tracking the skeleton points positions over time. Here, we define an abstract building block that can be instantiated for different gesture recognition supports. In order to do this, we have to consider that a gesture support provides the possibility to track a set of *features* that change through the time. As said before, the meaning of each feature (and the associated low-level event) depends on the concrete gesture recognition support. A feature is a n-dimensional vector (e.g., the position of a finger is a vector with two components, the position of a skeleton joint has three components, etc.).

A set of features can be also represented with a vector that contains a number of components equals to the sum of the dimensions of its elements. A set of features is the abstract representation of a *gesture recognition support* at a given time, since it describes the data provided by a given hardware and software configuration.

We will provide examples for the definition of a gesture recognition support in the following sections. The *state of a gesture support* at a given time is represented by the current value of each feature. The state of a gesture recognition support over time can be represented by a *sequence* of such states, considering a discrete time sampling.

Equation 3.1 defines a feature f , a gesture recognition support G_S , a gesture recognition support state G_{S_i} and a gesture recognition support state sequence S .

$$\begin{aligned}
 f &\in \mathbb{R}^n && (3.1) \\
 G_S &= [f_1, f_2, \dots, f_m] && G_S \in \mathbb{R}^k \quad f_i \in \mathbb{R}^{n_i} \quad \sum_{i=1}^m n_i = k \\
 G_{S_i} &= [f_1(t_i), f_2(t_i), \dots, f_m(t_i)] && t_i \in \mathbb{R} \\
 S &= G_{S_1}, G_S, \dots, G_{S_n} && n \in \mathbb{N}
 \end{aligned}$$

A gesture building block notifies a change of a feature value between t_i and t_{i+1} . Such notification can be optionally associated to a condition, which can be exploited for checking properties of the gesture state sequence such as trajectories for hand movements.

For instance, it is possible to check whether the path of a tracked point is linear or not, avoiding the notification of different movements.

The gesture support is responsible for the notification of the feature change, which is external with respect to the current state of the gesture recognition.

This aspect is modelled by the Non-Autonomous Petri-Nets, since the firing of a transition is enabled not only by the presence of the tokens, but also by the occurrence of an event that does not depend on the considered Net. Therefore, in Non-Autonomous Petri Net, the transition fires only if the incoming places contain a token *and* if an event of a given type occurs. We need such kind events in order to model the notification of a feature change by the considered gesture support.

We define an event type for each observed feature. In addition, we define a boolean predicate for each gesture state sequence constraint. As we already specified previously, such predicates are optionally associated to a feature change and constraints its recognition.

In our Petri Net it is possible to model the external notification with the definition of a function *raise*, which establishes if the external event is raised at a time t , as defined in equation 3.2.

$$\begin{aligned}
 \text{raise}(E_{f_i, P(S)}, t) &\Leftrightarrow (f_i(t) \neq f_i(t-1)) \wedge p(S) && (3.2) \\
 p: S &\rightarrow \{\text{true}, \text{false}\}
 \end{aligned}$$

In order to model the current progress in the gesture recognition, we use a control state token (Cs) on the Petri Net. The recognition of a basic block is enabled by the presence of such token, and it is inhibited by its absence. As we explain better in the following sections, the parallel recognition of different gestures in a composed Net is possible managing multiple instances of such control state token. The Petri Net in Figure 3.1 defines a basic building block for gesture recognition.

The two dotted arrows connect the ground term net to transitions that are “externals” with respect to the building block, namely the previous and the following parts of the gesture Net.

The place *Start F1* receives the control state token from its incoming transition. If we are considering the first place in the recognition net, it contains the token associated with the entire recognition process. The transition after this place fires only when the event $f1, p(S)$ occurs.

Finally, the control state token reaches the place *End F1*, concluding the basic gesture recognition. The actions that react to the basic gesture recognition are associated to the latter place. The out-coming arrow that starts from the *End F1* place connects the considered block with the next part of the gesture net.

In order to represent a basic building block we use the notation $F_i[p]$: we assign a name to the considered feature (F_i in this case) and to the boolean function (p), which is omitted if it is true for every gesture support state.

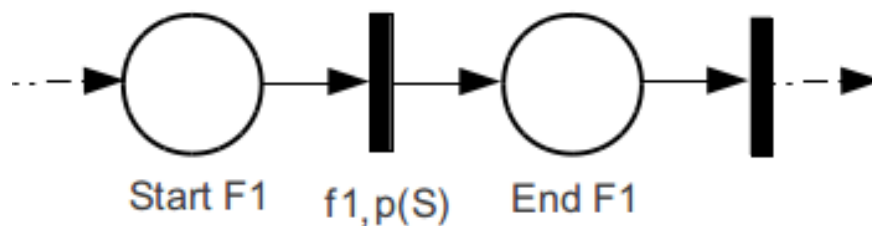


Figure 3.1 Gesture recognition building block

3.1.2 Composition Operators

A gesture description model is based on the composition of the aforementioned ground terms. The connection is performed through a set of operators, which express different temporal relationships among them. Such set has as starting point those defined in CTT [114], which has been proved

effective in defining the temporal relationship for task modelling, and that are defined also in process algebras (e.g. [18]).

Some of them (sequence and choice) have been already defined through Petri Nets in [107]. We provide here a complete definition of all operators.

Operator	Notation	Arity
Iterative	G^*	1
Sequence	$G1 \gg G2$	2 (n)
Parallel	$G1 G2$	2 (n)
Choice	$G1 [] G2$	2 (n)
Disabling	$G1 [> G2$	2 (n)
Order Independence	$G1 = G2 = \dots = Gn$	n

Table 3.1 Composition Operators

Table 3.1 lists the composition operators that we describe in the next sections. All binary operators are associative, therefore the n-ary version of a binary operator (e.g. choice) is defined applying such property.

During the discussion in the following sections, we need also the definition of three different sets of ground terms, given a complex gesture definition.

The first one is the set containing all its ground terms. We refer such set as GS (Ground terms Set).

Equation 3.3 defines how to construct the GS for a gesture G , which consists of a recursive set union on the sub-blocks connected through the composition operators.

$$\begin{aligned}
 G = F_i[p] &\Rightarrow GS_G = \{F_i[p]\} \\
 G = G1^* &\Rightarrow GS_G = GS_{G1} \\
 G = G1 \text{ op } G2 &\Rightarrow GS_G = GS_{G1} \cup GS_{G2} \\
 &\text{op} \in \{\gg, ||, [], [>\} \\
 G = G1 |=| G2 |=| \dots |=| Gn &\Rightarrow GS_G = \bigcup_{i=0}^n GS_{Gi}
 \end{aligned} \tag{3.3}$$

The second set we need to define contains only the ground terms not appearing as the right operand in a sequencing temporal relation, so they are immediately recognizable when the gesture execution starts. The operators that express such relation are *sequence* and *disabling*.

We call such set Starting Ground terms Set, or SGS and it is defined in equation 3.4. Obviously $SGS \subseteq GS$.

$$\begin{aligned}
G = F_i[p] &\Rightarrow SGS_G = \{F_i[p]\} \\
G = G1^* &\Rightarrow SGS_G = SGS_{G1} \\
G = G1 \text{ op } G2 &\Rightarrow SGS_G = SGS_{G1} \quad \text{op} \in \{\gg, [\>]\} \\
G = G1 \text{ op } G2 &\Rightarrow SGS_G = SGS_{G1} \cup SGS_{G2} \quad \text{op} \in \{||, []\} \\
G = G1 \text{ |=| } G2 \text{ |=| } \dots \text{ |=| } Gn &\Rightarrow SGS_G = \bigcup_{i=0}^n SGS_{Gi}
\end{aligned} \tag{3.4}$$

The last set we define contains the complementary features with respect to a given one in a gesture expression, and we call it $CGS_G(F_i)$, where G is a gesture and F_i is a ground term.

In other words, this set contains all the features used in the gesture expression that are different from the one specified. This set can be obtained simply subtracting the specified feature from the GS set for the considered expression. If the feature has an associated predicate, we have to add the specified feature with the logical negation of the predicate to the $CGS(F_i)$ set. The complete definition can be found in equation 3.5.

$$\begin{aligned}
CGS_G(F_i) &= GS_G \setminus F_i \\
CGS_G(F_i[p]) &= GS_G \setminus F_i[p] \cup F_i[\bar{p}]
\end{aligned} \tag{3.5}$$

3.1.2.1 Iterative Operator

The iterative operator repeats the recognition of gesture subnet for an indefinite number of times. In order to avoid an infinite gesture definition, each iterative basic block should also be coupled with a disabling operation. As already specified in Table 3.1, we use the $*$ symbol in order to represent the iterative operator (e.g. F^* recognizes an infinite number of value changes for the feature one).

It is possible to define this operator simply creating a cycle from the ending transition of a gesture subnet to its starting place. In this way, the recognition subnet is fed again with the control state token, immediately after the gesture has been recognized.

Figure 3.2 shows the Petri Net definition of the iterative operator. The thicker arrow represents the operator definition.

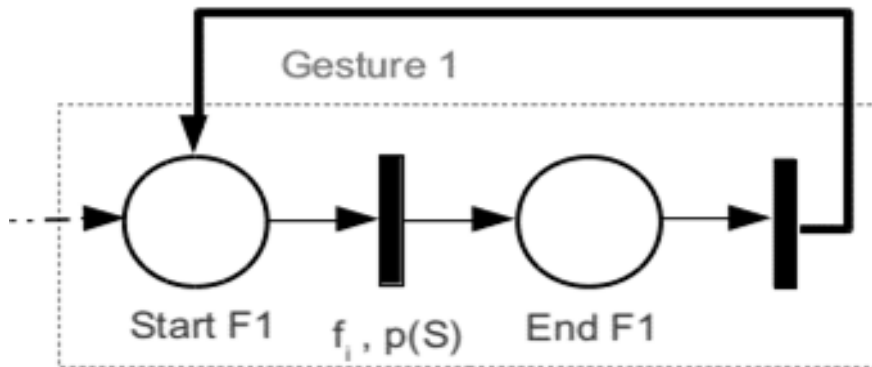


Figure 3.2 The Iterative operator

3.1.2.2 Sequence Operator

This operator simply defines that two gesture subnets should be performed in sequence. We use the \gg symbol in order to represent this operator. It is possible to define such operator connecting the last transition of the first gesture with the starting place of the second one.

Figure 3.3 shows a gesture consisting of the sequential composition of two basic feature recognizers. The thicker arrow represents the sequence operator.

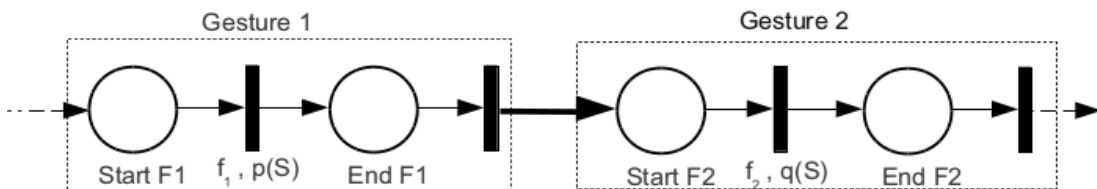


Figure 3.3 The Sequence operator

3.1.2.3 Parallel Operator

The parallel operator defines the recognition of two or more different gestures at the same time. We use the \parallel symbol in order to represent the parallel operator.

From the Petri Net definition point of view, the blocks representing the parallel gestures should be simply put in different recognition lines. In order to do this, we assign a different control state token to each line. This can be obtained, as shown in Figure 3.4, inserting a transition that “clones” the control state token and dispatching a copy to the starting place of each different recognition lines.

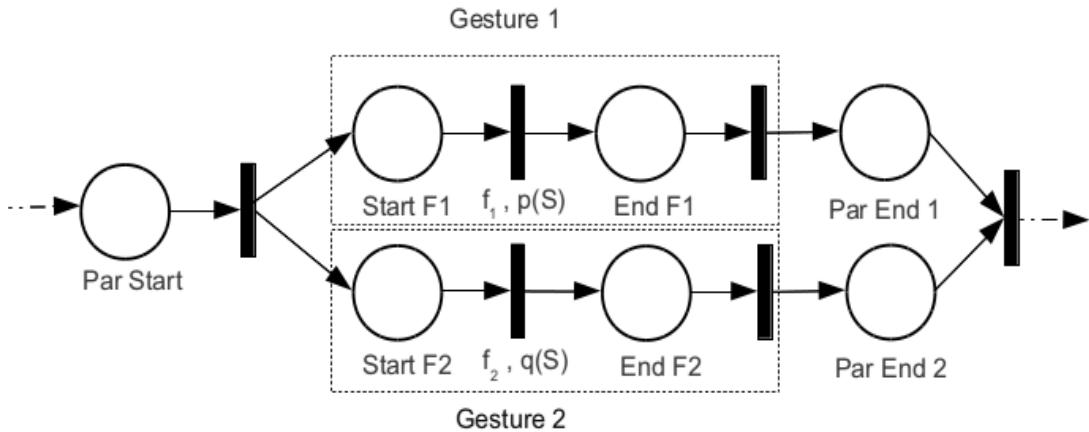


Figure 3.4 The Parallel operator

We add a place at the end of each recognition line that forwards the “cloned” control state token to the last transition that, once all gestures terminated, restores only one token in the net.

3.1.2.4 Choice Operator

The choice operator defines a gesture that is recognized if exactly one between its first and its second component is detected (either one or the other). We use the symbol $[]$ for representing it.

The net can be defined as it is shown in Figure 3.5, and its construction is similar to the parallel operator. The transition after the *Choice Start* place splits the control state token between two subnets, each one representing a component involved in the choice. The two lines cannot evolve independently as happens for the parallel operator. Therefore, when one subnet starts its recognition, the other one should be interrupted. In order to do this, it is sufficient to connect the first place of the first gesture subnet with the first transition of the second one and vice versa. In this way, once one of the two feature events is raised, the control state token from the other gesture subnet is deleted.

More precisely the steps to be followed for constructing a Petri Net for $G1[]G2$ in the general case are the following:

1. Calculate SGS_{G1} and SGS_{G2}
2. Connect the first place of each element of SGS_{G1} with the first transition of each element in SGS_{G2}
3. Connect the first place of each element of SGS_{G2} with the first transition of each element in SGS_{G1}

The last transition of each gesture subnet is connected to the *Choice End* place, which forwards the control state token to the following part of the recognition net.

This definition of the choice operator envisions an immediate selection between the two sub-gestures involved in the choice. Immediate selection means that the choice is performed taking into account elements from the *SGS*, thus it considers only the Ground Terms which can be recognized at the beginning of the choice.

Such approach has the advantage that is sufficient to recognize only a ground term in order to perform the choice. The main problem is that most of the times the sub-gestures that are connected with the choice have a common prefix, which is a set of ground terms at the beginning of the expression. For instance, it is possible to take into account one finger and two fingers multitouch gestures. The definition for both categories start always with the detection of one finger on the screen. If we consider the previous definition of the choice operator, the selection is ambiguous.

As we discuss more in detail in Chapter 7, having a shared prefix between the choice operands is really common. Therefore, in order to ease the definition of gestures in choice, we defined a variant that applies a best effort approach for performing such selection. The basic idea is to delay the selection until only one of the two operands can continue in the recognition process. This means that the two gestures are recognized in parallel until one of them is blocked and the choice is performed.

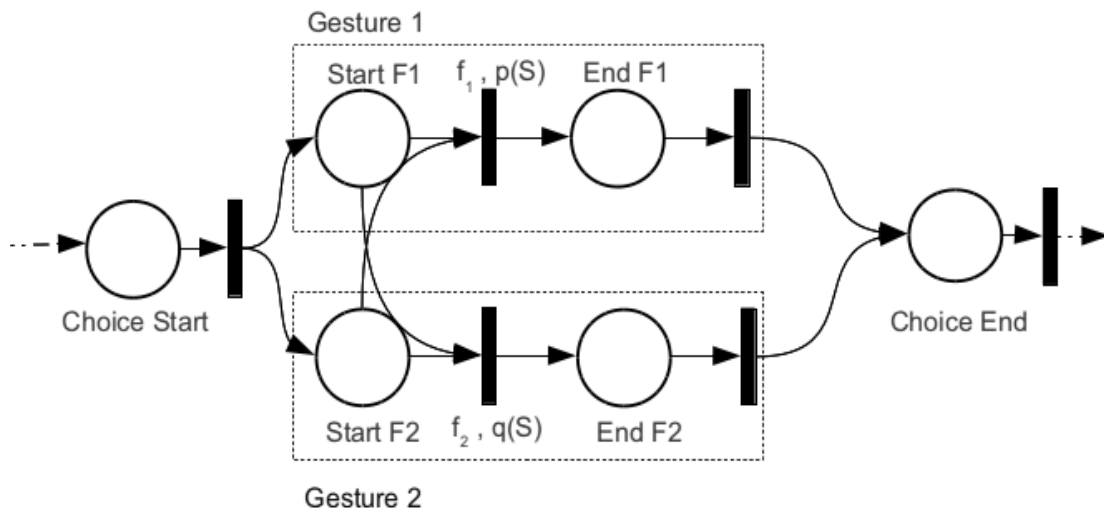


Figure 3.5 The Choice operator (immediate variant)

The structure of the Petri Net for the best effort variant of the choice operator is shown in Figure 3.6.

As in the previous variant, we put the two operands on two parallel recognition lines, duplicating the token. The difference is the way we use for disabling one of the two lines. For each one of the depicted operands, the normal recognition flow for the gesture is disabled if it is no more possible to continue the recognition. In the Petri Net, such concept is modelled adding, for each ground term contained into the two operands, a transition that fires if one of the elements of the *CGS* set is recognized. We recall that such set contains all the features of the considered gestures which are different from the one considered (see section 3.1.2). In Figure 3.6, such transitions are labelled *Comp F1* and *Comp F2* respectively for the first and the second operand in the choice.

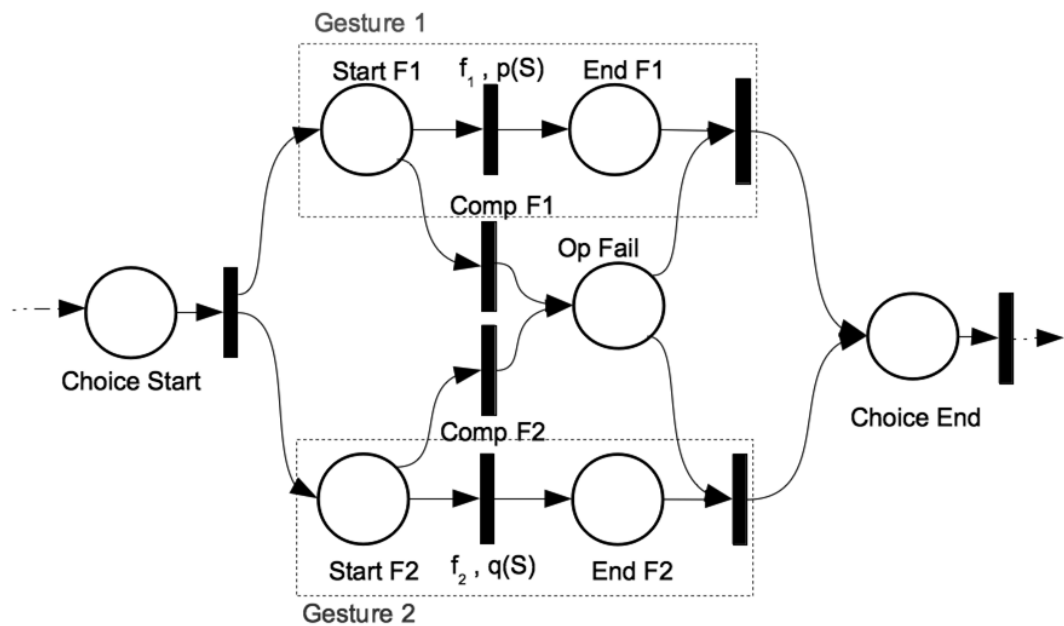


Figure 3.6 Choice operator (best effort variant)

If one of the two lines cannot recognize the gesture, its token goes to the *Op Fail* place. Here we have two possibilities: the first one is that the other line successfully completes the recognition. In this case, since the place is connected with the last transition of both the operands, the token is consumed and the recognition proceeds as usual. The second possibility is that both tokens arrive at the *Op Fail* place. This means that both

recognition lines failed the recognition, and so the choice. In such situation, the Net cannot proceed and it is in an error state (described in section 3.1.3).

In the general case, it is possible to construct the Net as follows:

1. Connect the place *Op Fail* to the ending transition of all choice operands ($G1$ and $G2$)
2. For each ground term F_i in GS_{G1} and GS_{G2} calculate the sets $CGS(F_i)$
3. For each ground term F_i in GS_{G1} and GS_{G2} , add a transition between F_i and *Op Fail* which fires if one of the features in $CGS(F_i)$ is recognized.

From now on, we consider the choice operator the best effort variant.

3.1.2.5 Disabling Operator

The disabling operator defines a gesture that stops the recognition of another one, thus “disabling” it. The operator symbol is $[>]$. It is typically needed when a gesture is iterative, in order to define the condition that stops the loop. Figure 3.7 shows the definition of the disabling operator using Petri Nets for $G1[> G2$.

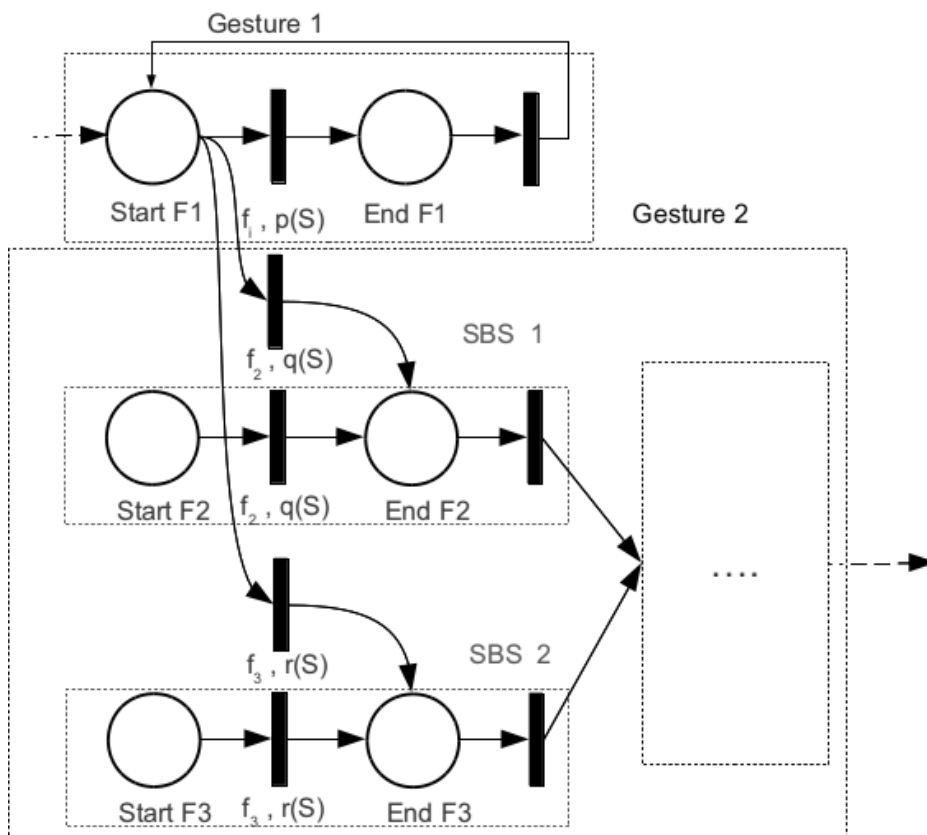


Figure 3.7 The Disabling operator

The basic idea is to connect the first place of each basic component belonging to $G1$ to a “copy” of the first transition of the starting blocks of the second one. In Figure 3.7 we can see an example of this kind of net, where the first gesture is composed by only one building block.

This gesture can be disabled by the second one, which starts with an event related either to the feature $f2$ or $f3$. In order to obtain the desired effect, we connect the *Start F1* place with a copy of both the transitions after the *Start F2* and *Start F3*. In order to construct the net for $G1[> G2$ in the general case, we need to perform the following steps:

1. Calculate the sets GS_{G1} and SGS_{G2}
2. Connect the starting place of each element of GS_{G1} with a copy of the first transition of each element in SGS_{G2} , possible duplicates (transitions that have the same incoming places and the same external event) are merged. In case of order independence operator, a transition duplicate is added also to each *OI Flag* and *OI End* (see section 3.1.2.6)
3. Connect the second place of each element in SBS_{G2} with the transitions generated at step 2. Such connection has to preserve the single control state token property for each sub-gesture, so we need to collapse recursively the recognition lines with net in the case $G1$ sub-components contain the parallel or the order independence operator. The technique is the same shown in Figure 3.4 for the parallel operator.

3.1.2.6 Order Independence

The order independence operator is used when two or more gestures can be performed in any order. The composed gesture is recognized when all of its subcomponents have been recognized. We use the symbol $|=|$ for this operator.

It is worth pointing out that such operator is not strictly needed, because it is possible to derive it according to the property in equation 3.6.

$$G1 \mid = \mid G2 = (G1 \gg G2) [] (G2 \gg G1) \quad (3.6)$$

In general, we can define an order independence composition of a set of n gestures as a choice between all the permutations of its elements. Inside each permutation the gesture set elements are connected through the sequence operator.

Obviously, such kind of definition creates $n!$ options for the choice that makes it too expensive both from the space and time point of view. It is possible to provide a more compact net for defining this operator, which is shown in Figure 3.8. The idea is to create a Petri Net that repeats n times the choice between the composed subnets, removing one option at each iteration.

The place *OI Start* receives the control state token and creates two copies of it for each gesture connected by the operator, in the same way we do for the parallel operator.

For each gesture component, we create a place called *OI Flag*, which receives one of the two control state token copies. Such token is used in order to remember whether the corresponding gesture subnet has been recognized in a previous iteration or not.

We guarantee two construction properties for this place. The first one is that each *OI Flag* loses its token only when the corresponding gesture subnet ends its execution. This is enforced by an out coming connection between *OI Flag* and the last transition of the gesture sub-component net. The second property is that the *OI Flag* maintains its control state token until the gesture sub-component has been recognized. This is obtained with an incoming and out-coming connection of the *OI Flag* place with each event transition of the gesture sub-component net. This property guarantees that, when the sub-gesture has been already recognized, it is not possible to restart it until a new token arrives from the *OI Start* place. The presence of a token in an *OI Flag* place indicates that the corresponding sub-gesture has not been recognized yet, while its absence indicates that the recognition has already happened.

The second copy is received by the first place of the gesture component sub-net. With this construction we guarantee that a gesture sub-net will be chosen only once for each iteration. Now we need to add something in order to avoid that two or more gesture sub-nets can start their recognition in parallel. We already discussed a technique that guarantees this for the choice operator. We reapply the same technique here, connecting the starting transition of each order independence component to all the starting transition of all the other components. In Figure 3.8 such connections are the following:

- The one that connects the *Start F1* place and the $f_2 q(S)$ transition
- The one that connects the *Start F2* place and the $f_1 p(S)$ transition.

In order to guarantee that the choice is performed more than once, we connect the last transition of each order independence component with its starting place, in the same way we explained for the iterative operator. In addition, we create also an *OI End* place for each component, which receives a copy of the control state token when the corresponding gesture sub-net ends its recognition. All the *OI End* places are connected to the last transition of the order independence subnet. When they all contain a control state token, all sub-gestures have been recognized, and the entire gesture is completed.

The starting places of the different components are connected with the last transition, in order to consume the control state tokens that returned back after the n -th iteration.

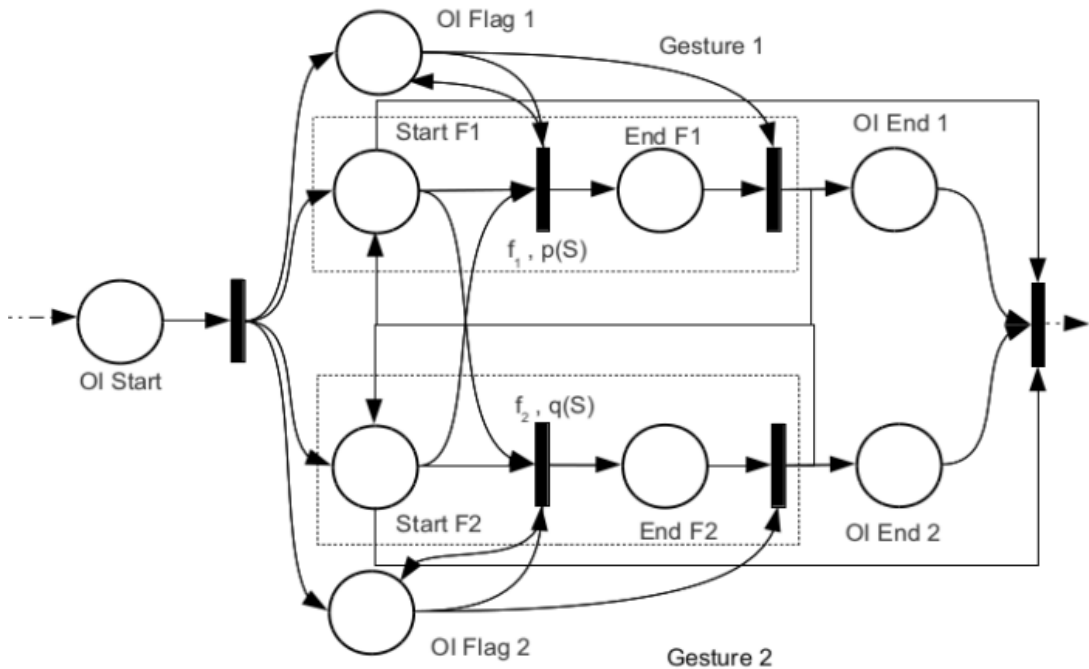


Figure 3.8 Order independence operator Petri Net

The steps to construct this net for $G1|=|G2|=|\dots|=|Gn$ are the following:

1. Calculate $SGS_{Gi} \forall i \in [1, n]$
2. Create an *OI Flag* place for each Gi and connect it with its last transition.
3. Create an *OI End* place for each Gi and connect it with the same transition at the end of the net.
4. Connect the transition after the *OI Start* place with each starting place of all elements in SGS_{Gi} and with all the *OI Flag* places.

5. For each $i \in [1, n]$, connect the starting places of each element of SGS_{G_i} with all the starting places of each element in $\cup_j SBG_{G_j}$, with $j \in [1, i - 1] \cup [i + 1, n]$
6. For each $i \in [1, n]$, connect the event-driven transitions of each element of GS_{G_i} with $OI\ Flag_i$ and vice versa.
7. For each $i \in [1, n]$ connect the ending transition of the net associated to G_i with all the elements in SGS_{G_i}
8. For each $i \in [1, n]$, connect the starting places of each element of SBS_{G_i} with the last transition of the order independence net.

3.1.2.7 Short-hands

Even if they are not strictly required for the definition of the meta-model, we consider a set of short-hands that are useful for the definition of the temporal relationships among gestures.

The first one is useful when the designer wants to recognize a gesture a given number of times (e.g. five). We specify the number of times as a superscript for the gesture, in brackets. Such kind of iteration can be obtained obviously through a chain of sequence operators, as shown in equation 3.7.

$$F^{\{n\}} = \underset{i=1}{\overset{n}{\gg}} F \quad n \in \mathbb{N} \quad (3.7)$$

The second short-hand we use is related to the definition of iterations that should be recognized at least a given number of times. The shorthand is again a superscript for the gesture symbol and contains the minimum number followed by a comma and the Kleene star, inside brackets.

The semantics of the shorthand can be defined again with a chain of sequence operators, followed by the gesture with the iterative operator, as shown in equation 3.8

$$F^{\{n,*\}} = \underset{i=1}{\overset{n}{\gg}} F \gg F^* \quad n \in \mathbb{N} \quad (3.8)$$

3.1.3 Handling recognition errors

Besides the recognition of a gesture, it is important also to define how to react if the sequence of events received does not match the gesture definition.

This case can be detected when the notification of an external event related to some observed property does not fire any transition. In such case, the gesture recognition should be interrupted, and the developer should have the possibility to define the interface reaction to such interruption.

This can be supported associating a handler not only for the successful recognition of a gesture (either basic or composed), but also for the recognition failure. Obviously, the recognition failure is propagated from through the composition tree, from the component to its parent.

From the Petri Net point of view, such handling can be modeled adding a transition for each ground term to a place that represents the recognition error. Such transition fires if one of the elements in $CGS(F_i)$, being F_i the feature associated to the ground term (see section 3.1.2).

3.2 Modelling multitouch gestures

A multitouch screen can detect multiple simultaneous touches. For each touch, the device can detect its screen position (usually expressed in pixel). In addition, it is possible to detect the current time.

According to our abstract meta-model, we have n features related to the touch positions (one for each detectable touch) and a feature related to the current time. If a touch is not currently detected on screen, we say that its current position is the point (\perp, \perp) .

We identify the feature related to the i -th touch with p_i , while we use the *time* symbol for the time. In order to have a uniform terminology with the current multi-touch toolkits, we define the simplest set of multitouch gestures in equation 3.9. From these building blocks it is possible to define complex gestures using the composition operators, which are described in the following subsections.

$$\begin{aligned} Start_i &= p_i [p_i(t-1) = (\perp, \perp) \wedge p_i(t) \neq (\perp, \perp)] \\ Move_i &= p_i [p_i(t-1) \neq (\perp, \perp) \wedge p_i(t) \neq (\perp, \perp)] \\ End_i &= p_i [p_i(t-1) \neq (\perp, \perp) \wedge p_i(t) = (\perp, \perp)] \end{aligned} \tag{3.9}$$

3.3 Modelling full-body gestures

The devices that enable the recognition of full-body gestures (e.g. Microsoft Kinect [87], Asus Xtion PRO [8] etc.), are able to sense the 3D position of the complete skeleton joints for up to two users, while they can sense the

body centre position of up to four more users, in meters. The SDKs provide facilities for projecting the position on the image space of the RGB camera or depth sensor, obtaining the corresponding coordinates in pixels (obviously, without considering the depth axis). In addition, they are also able to track the joint orientations, providing a 3D vector.

Moreover, it is possible to have more information using Computer Vision techniques. For instance, it is possible to detect fingertips if the user is really close to the sensor, or to detect if a hand is open or not at intermediate distances (e.g. calculating the convex hull and convexity defects [21]).

It is clear that for this kind of devices the available toolkits share most of the features, but we have still a set of differences which is larger if compared with multitouch SDKs.

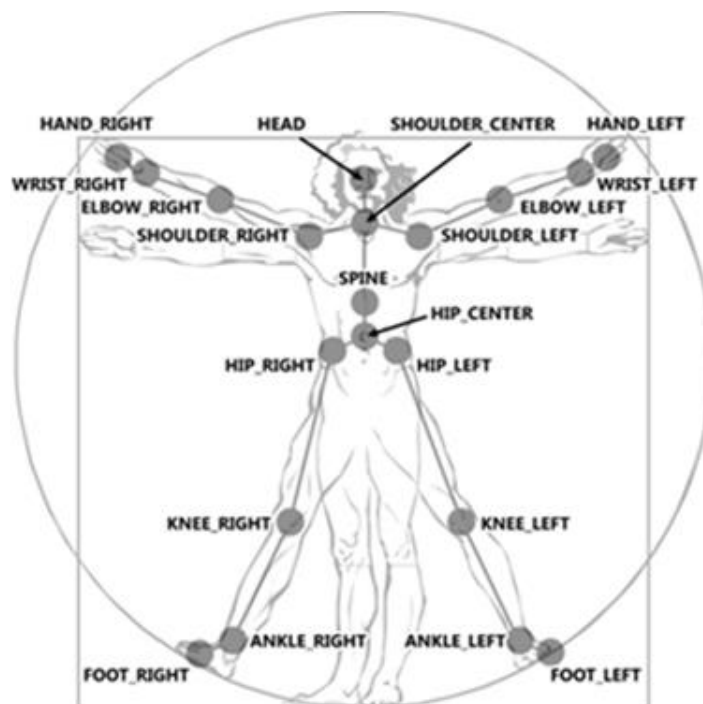


Figure 3.9 Skeleton joints

From the point of view of our abstract meta-model, it is possible to include all the features provided by all the frameworks. However, in order to be able to provide a proof-of-concept implementation, we had to fix a set of features we deal with.

Therefore, from now on we limit the scope of the full body gesture features to the following list, unless otherwise specified:

- The time

- 3D position and orientation of the skeleton joints, depicted in Figure 3.9
 - Head
 - Shoulder center
 - Shoulder left
 - Shoulder right
 - Elbow left
 - Elbow right
 - Wrist left
 - Wrist right
 - Hand left
 - Hand right
 - Spine
 - Hip center
 - Hip left
 - Hip right
 - Knee left
 - Knee right
 - Ankle left
 - Ankle right
 - Foot left
 - Foot right
- Left hand open (true if open, false otherwise)
- Right hand open (true if open, false otherwise)

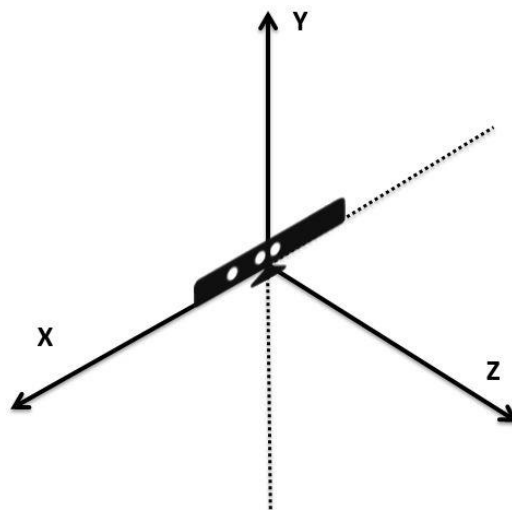


Figure 3.10 Full-body gesture coordinate system

Each feature is available for each user tracked by the device. We indicate the user id in the gesture expression only if it involves more than one user.

The coordinate space representation used by our meta-model is shown in Figure 3.10. It considers a right-handed coordinate system that has its origin in the position of the tracking device. For other tracking systems that are not based on depth sensors, it is possible to consider the screen as the origin of the axes.

3.4 Comparison with Proton++

In this section we demonstrate that possible gestures modelled using Proton++ [72][73] are a subset of those that may be defined with GestIT. Proton++ is the declarative approach closest to GestIT in literature, as described in section 2.3.1.1.

We prove it showing a general way for mapping the regular expressions used in Proton++ towards the GestIT notation. In addition, we show that it exists a class of GestIT models, which is not possible to define using Proton++.

Obviously, since Proton++ describes only multitouch gestures, we define the correspondence between the regular expression literals and the ground terms only for the multitouch platform.

However, it is worth pointing out that the higher expressiveness of the modelling approach is not related to the gesture recognition support, but it is related to a less expressive set of operators provided by Proton++. Indeed, it would be possible to model full-body gestures using the Proton++ approach providing a set of literals related to a full-body tracking device, but even in this case there is a set of gestures that can be expressed with GestIT but not with Proton++.

3.4.1 Proton++ literals

A Proton++ literal is identified by:

1. An event type (touch down, touch move, touch up)
2. A touch identifier
3. An object hit by the touch
4. A set of custom attributes values (one or more), such as e.g. the touch trajectory, shape etc.

In GestIT for multitouch, a ground term is identified by an event type (touch start, touch move or end) and by a touch identifier. Therefore, the correspondence between the first two elements of the Proton++ literal and the GestIT ground term is straightforward. The third and fourth component of a Proton++ literal can be all modelled constructing a correspondent predicate associated to a GestIT ground term.

We recall that a predicate associated to a ground term in GestIT is a boolean condition checks whether the gesture performance conforms to a set of gesture-specific constraints or not. According to this definition, the third component can be modelled with a predicate that checks if the current touch position is contained into an object with a given id or belonging to a particular class.

The fourth component can be modelled considering, for each Proton++ custom attribute value, the function that computes its value. Such computation may depend on the current or previous touch positions, or it may depend also on other gesture features. In brief, such function depends on what we call the gesture support state sequence.

The function that calculates the attribute value has been defined in Proton++ for associating it to a literal. Therefore it is also possible to provide a predicate that compares the current attribute value with the specified in the regular expression, in order to be translated in a boolean form that can be exploited in GestIT. If more than one value is acceptable, the predicate can be defined simply through a boolean OR of the comparison for the different values. Obviously, if the touched object and a set of custom attributes for the literal need to be modelled, it is sufficient to define a single predicate that is composed by the boolean AND of the corresponding predicates.

Proton++	GestIT
$E_{Tid}^{O V_1 \dots V_n}$	$E_{Tid}[p]$ where: $o = true \Leftrightarrow O_{type} = O$ $a_i = true \Leftrightarrow A_i = V_i \quad i = 1 \dots n$ $p = o \wedge (a_1 \vee \dots \vee a_n) \quad i = 1 \dots n$

Table 3.2 Mapping a Proton++ literal to a GestIT ground term

Table 3.2 summarizes how to transform a Proton++ literal into a GestIT ground term. E represents an event type T_{id} a touch identifier, O_{type} is a

property that maintains the current object type, O is a concrete value for the object type (e.g. start, rectangle etc.), A_i is a property that maintains the value of an attribute, while V_i is the actual attribute value, while p is a boolean predicate associated to the ground term in GestIT.

3.4.2 Proton++ operators

The correspondence between the Proton++ and the GestIT ones is straightforward, since all the operators defined by the former have an equivalent in the latter. Table 3.3 summarizes how to transform the operators from Proton++ to GestIT.

Proton++	GestIT
Concatenation: $A_P B_P$	Sequence: $A_G \gg B_G$
Alternation: $A_P B_P$	Choice: $A_G [] B_G$
Kleene star: A_P^*	Iterative: A_P^*

Table 3.3 Mapping Proton++ operators to GestIT

Applying recursively the transformations defined in Table 3.3 and Table 3.2, it is possible to build a GestIT gesture definition corresponding to a Proton++ one.

The vice-versa is not possible in general, since there is no way to transform the *Disabling* and the *Parallel* operators from GestIT to Proton++.

The *Disabling* operator is important in order to stop the recognition of iterative gestures, in particular the composed ones.

Most of the times, it models how to interrupt the iterative recognition of a gesture. For instance in a grab gesture, the iterative recognition of hand movements is interrupted by opening the hand. In addition, it may be used also for modelling situations where the user performs an action that interrupts the interaction with the application. For instance, all the Kinect games have a “pause” gesture that disables the interaction. In some applications we describe in this thesis, the disable operator is used for modelling the fact that the application tracks the user only if she is in front of the screen. Therefore, the gesture “shoulders not parallel to the screen plane” disables the interaction.

This is particularly relevant while interacting with devices that track the user continuously (e.g. Microsoft Kinect), since it is important to provide the user with a way to disable the interaction at any time.

The *Parallel* operator has a clear impact when modelling parallel input for e.g. multi-user applications. For instance, the parallel operator can be useful in a scenario where a user zooms a photo on a multitouch table while another user drags another picture, simply composing two existing gestures. In addition, it is also possible that parallel interaction occurs with a single user. A user may drag an object through a single-hand grab gesture and point with the other hand for selecting where to drop it.

Chapter 4

Gesture Models

In this chapter, we provide the definition of different gesture models for both multitouch and full-body interaction.

4.1 Common multitouch gestures models

In this section, we provide a definition for the most common multitouch gestures, using the GestIT notation, showing some modelling examples. It is worth pointing out that all the following gestures can be in turn composed in order to obtain more complex interactions.

In order to graphically show the gesture performance, we exploit the representation in [142].

4.1.1 Tap

The tap gesture is simply a touch immediately released from the screen, and it is shown in Figure 4.1. It can be simply described with equation 4.1: the gesture starts with the touch of the first finger, which is immediately released from the screen.

$$Start_1 \gg End_1 \tag{4.1}$$

Tap



Briefly touch surface
with fingertip

Figure 4.1 The touch gesture

4.1.2 Double Tap

A double tap is a tap followed by another tap in the same position, with a maximum distance in time. The gesture is shown in Figure 4.2.

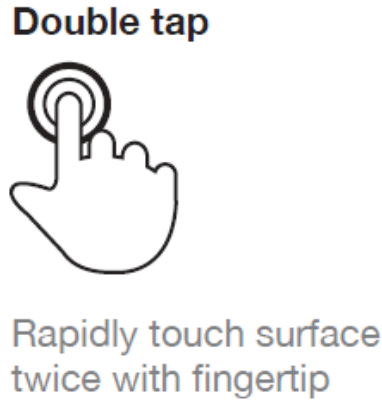


Figure 4.2 Double tap gesture

We specify two constraints: the first checks that the two touch start points are (almost) in the same position (modelled with the predicate *pos*), while the second one that their difference in time is not above a given threshold (modelled with the predicate *timeDiff*). The description is simply a sequence of taps, with the constraints to be checked on the second touch start, shown in equation 4.2.

$$Start_1 \gg End_1 \gg Start_2 [pos \wedge timeDiff] \gg End_2 \quad (4.2)$$

4.1.3 Pan

The pan gesture consists on a single finger that touches the screen, changes its position a certain number of times, and then it is released from the screen, as shown in Figure 4.3.

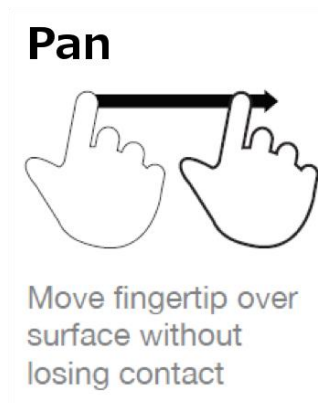


Figure 4.3 Pan gesture

The definition of the gesture using the GestIT notation is shown in equation 4.3. After the touch is detected on the screen, we have an iterative movement of the touch position. The loop is ended when the user releases the touch from the screen. It is possible to add constraints to the finger trajectory simply specifying an additional property for the *Move* feature.

$$Start_1 \gg Move_1^* [> End_1 \tag{4.3}$$

4.1.4 Slide

The slide gesture is simply a linear pan with a moving speed higher than a certain threshold.

The modelling of the temporal relationships between the touch features is exactly the same of the pan gesture. The difference is a specific constraint for the path. We define such constraints through two predicates, one that checks whether the trajectory is linear (*linear*) and another one that compares the current speed with the specified threshold (*speed*). Thus, the slide gesture can be defined with the expression in equation 4.4.

$$Start_1 \gg Move_1^*[linear \wedge speed] [> End_1 \tag{4.4}$$

4.1.5 Pinch

The pinch gesture is usually exploited in multi touch devices for zooming in or out a view. It consists of the contemporary touch of two fingers on the screen, followed by an increase or decrease of the distance between them, due to a parallel movement of the two fingers. Lifting the two fingers from the screen ends the gesture. The pinch gesture is depicted in Figure 4.4.

In order to model the gesture with the GestIT notation, we split the execution in three different phases.

In the first one, the user touches the screen with two fingers. Obviously, the touch order is not important, therefore we can use an order independence relationship for the touch start features. After that (sequence) the user can move both fingers on the screen independently an indefinite number of times. In this case, we can use a parallel operator for connecting the two *Move* ground terms. Finally, such iterative movements are disabled by the lift of one of the two fingers (the *End* features), again without any constrains in the lifting order.

The complete expression is shown in equation 4.5.

Pinch



Touch surface with two fingers and bring them closer together

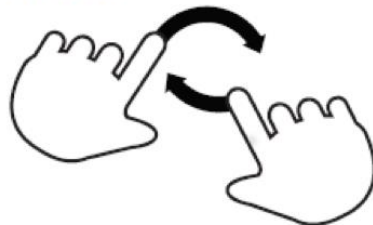
Figure 4.4 Pinch gesture

$$(Start_1 \mid = \mid Start_2) \gg ((Move_1^* \mid \mid Move_2^*) [> (End_1 \mid = \mid End_2)]) \quad (4.5)$$

4.1.6 Rotate

The rotate gesture is similar to the pinch, but instead of increasing or decreasing the finger distance, the user moves the two fingers in a circular path, as shown in Figure 4.5.

Rotate



Touch surface with two fingers and move them in a clockwise or counterclockwise direction

Figure 4.5 Rotate gesture

The gesture description is the same as the pinch from the temporal point of view, but we should check the circular trajectory, represented by the *circle* property.

$$(Start_1 \mid = \mid Start_2) \gg (Move_1^*[circle] \mid \mid Move_2^*[circle]) \quad (4.6)$$

[> (*End*₁ |=| *End*₂)

4.2 Common full-body gesture models

Considering the full-body gestures, it is more difficult to find a well-established vocabulary with respect to the multitouch interaction, which can be used as a benchmark for the proposed meta-model.

Therefore, we tried to create a list of common gestures through a literature review, trying to identify the common ones and to provide a unified naming convention for those that are exploited in different work, but called in different ways by different authors.

We do not consider applications that exploit the full-body tracking device in order to mimic the user's movement through an avatar, as happens to the wide majority of the Kinect enabled games for Xbox 360, since the effects of the body movements are mapped one-to-one with the user's virtual counterpart.

The following is the list of papers we considered for identifying the common gestures:

- In [45], the authors propose the integration of full-body gesture interaction into a medical image viewer.
- In [79], the authors selected a set of gestures for developing a machine-learning recognizer based on a restricted set of features.
- In [12], the authors propose a gestural interface for the remote control of a robot
- In [67], the authors propose a set of gestures for controlling a Google Maps through gestures
- In [74], the authors enhanced a book story telling application, providing the possibility to select different paths on the plot through a set of gestures. A user study demonstrated that the users prefer such selection mechanism if compared with pressing buttons.
- In [33], the authors describe another gestural controller remote control interface for robots
- In [35], the authors propose a gestural interface for controlling Power Point presentations.
- In [135], the authors defined a set of gestures for navigating in a virtual 3D environment.
- In [24], the authors provided an interface for controlling the movements of a robot.

- In [78], the authors created a 3D model visualizer, which can be controlled by gestures.
- In [43], the authors describe a flexible way for adding gestural interaction to applications that do not support it. They propose a set of gestures that can be employed in different settings
- In [22], the authors present Code Space, a system for enabling collaboration among developers exploiting touch and on-air gestures.
- The book in [138] describes the basics of the development of Kinect enabled applications exploiting the Microsoft Kinect SDK. The Chapter 6 is dedicated to gestures, and the authors describe a set of typical gestures and how they can be recognized.
- The middleware described in [115] provide a set of reusable graphical controls for creating gestural interfaces
- The work in [125] and [127] is reported in section 5.4, since it discuss the application developed exploiting the proof-of-concept library for our gesture modelling approach. The applications are respectively a 3D model visualizer and a touchless recipe browser.

In the following sections, we discuss the performance of each identified gesture and we provide the correspondent model. In addition, we explain how it has been exploited in the different work selected in literature. A summary of the different identified gestures together with their exploitation in the different selected papers is provided in Table 4.1

Gesture	[45]	[79]	[12]	[67]	[74]	[33]	[35]	[135]	[24]	[78]	[43]	[22]	[138]	[115]	[125]	[127]
Pointing	Black	White	Black	Black	White	White	White	White	White	White	White	Black	Black	Black	White	Black
Grab	Black	White	White	White	White	White	White	White	White	Black	White	Black	White	White	White	White
Push	White	Black	White	Black	Black	Black	Black	White	White	Black	Black	White	White	Black	White	White
Push back	White	Black	White	White	White	White	White	Black	Black	White	White	White	White	White	White	White
Lateral push	White	White	White	White	White	White	White	Black	White	White	White	White	Black	White	White	White
Kick	White	White	White	White	Black	White	White	White	Black	White	White	White	White	White	White	White
Wave	Black	White	White	White	White	Black	White	White	White	White	White	White	White	Black	White	White
Swipe	White	Black	White	White	White	White	White	White	White	Black	White	White	White	Black	White	Black

Gesture	[45]	[79]	[12]	[67]	[74]	[33]	[35]	[135]	[24]	[78]	[43]	[22]	[138]	[115]	[125]	[127]
Walk				■	■	■										
Turn				■												■
Diverge/Converge hands	■	■	■							■	■	■				
Steering wheel	■	■								■	■	■				
Roll															■	
Universal Pause																■

Table 4.1 Common full-body gestures in literature

4.2.1 Pointing

The pointing gesture consists of the usage of the dominant hand (or optionally the non-dominant hand), for selecting an object on the screen. Figure 4.6 graphically shows the gesture performance.

The relationship between the hand position on the real world and the corresponding position on the screen can be defined in different ways. For instance, it is possible to exploit the *image-plane* approach described in [64], where the on-screen position is obtained tracing a ray from the user's eye location, passing from the finger tip and intersecting it with the screen plane. In [45], the authors approximate this approach replacing the eye position with the head point of the skeleton and the fingertip with the position of the dominant hand.

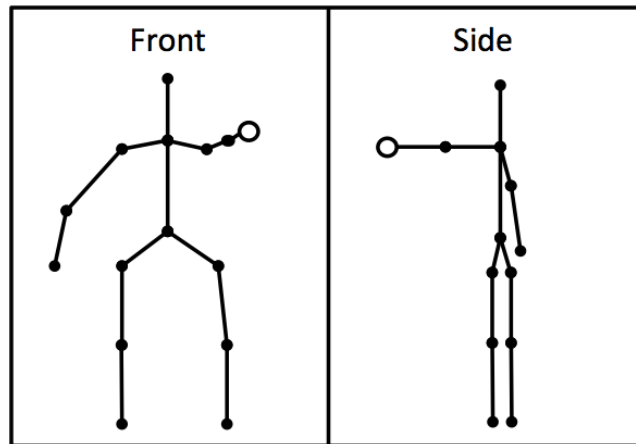


Figure 4.6 The pointing gesture

A different approach is a direct mapping between the screen and the real world, defining a scale matrix between the two spaces. This is the a typical approach since it is adopted by the Kinect SDK, and it is possible to find it in literature for instance in [138].

Another source of variation for the pointing gesture is related to the space where the hand movements are tracked. It is possible to define a depth barrier where the hand position is tracked only if its Z coordinate is lower than a certain threshold. Another possibility is to define a 2D plane in front of the user.

In order to model this gesture with our declarative approach, it is sufficient to iteratively track the position of the dominant hand (e.g. the right one). It is possible to optionally associate a predicate to the recognition of the hand feature in order to limit the tracking space. Equation 4.7 shows

the definition of the pointing gesture for the right hand (represented by the mH_r feature). A symmetric definition is possible for the left hand. The predicate ts can be instantiated in different ways in order to limit the tracking space.

From the interaction semantics point of view, this gesture has the obvious effect of selecting an area on the screen, or provides a direction for controlling a robot

$$mH_r^*[ts] \tag{4.7}$$

4.2.2 Grab

In its simplest form, the grab gesture consists of simply closing one hand. In this form it has been exploited for instance in [45] and in [127].

A different definition of the same gesture can be found in [78], [22], [125] and [127], where the hand closure is followed by a change of the closed hand position until the hand is reopened.

This variant is exploited for providing a manipulation metaphor for rotating [78] or moving [125] a 3D model, changing the position of video timeline [127] or for implementing an on-air drag and drop [22].

The grab gesture performance is shown in Figure 4.7: the bigger black dots represent a closed hand, while the white dots represent an open hand. The first phase is the same for all the gesture variants: the user closes the hand. The phase number 2 and number 3 belong to the second variant of the gesture: the user can move the closed hand in different directions (represented with the arrows in the second part of Figure 4.7). Finally, the user opens the hand (the third part of Figure 4.7).

The gesture modelling for the two variants is shown in Equation 4.8, considering the right hand (the left one is symmetric). The feature oH_r represents a change on the open/closed state of the hand, while the predicate c ensures that the hand is closed.

The second version of the gesture offers the possibility to drag the grabbed object with the closed hand and then release it. This is modelled using a sequence operator after the hand closure, which allows the closed hand to be moved iteratively (represented by the mH_r feature). The loop is disabled by a change in the hand closure state that changes from closed to opened. The expression models such change exploiting the feature oH_r , which is associated to the \bar{c} predicate, which is the logical negation of c .

$$\begin{aligned}
 Grab_{v1} &= oH_r^*[c] \\
 Grab_{v2} &= oH_r^*[c] \gg (mH_r^*[\gt oH_r^*[\bar{c}]]) \\
 &= Grab_{v1} \gg (mH_r^*[\gt oH_r^*[\bar{c}]])
 \end{aligned}
 \tag{4.8}$$

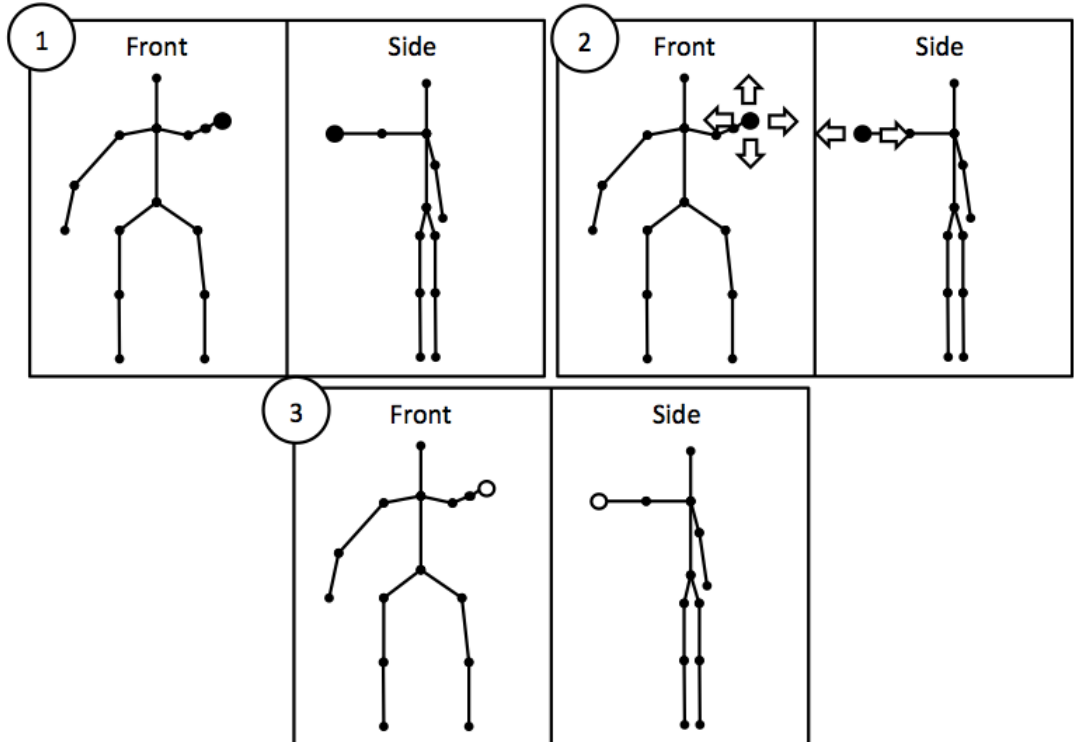


Figure 4.7 The grab gesture

4.2.3 Push

The push gesture mimics the action for pressing a virtual on-air button, stretching out one hand towards the screen. The approach for recognizing this gesture is based simply on a depth barrier definition between the user's position and the screen. If one of the hands crosses the barrier, the push is detected.

Figure 4.8 graphically shows how the gesture can be performed. The depth barrier is depicted using a dotted line in the side view.

From the modelling point of view, the push gesture is a simple change in the position of the hand feature, which has to cross the depth barrier. This can be simply defined by a sequence of two hands movements: the former has a depth value greater than the depth barrier (see Figure 3.10), while the latter has a depth value lower than the depth barrier.

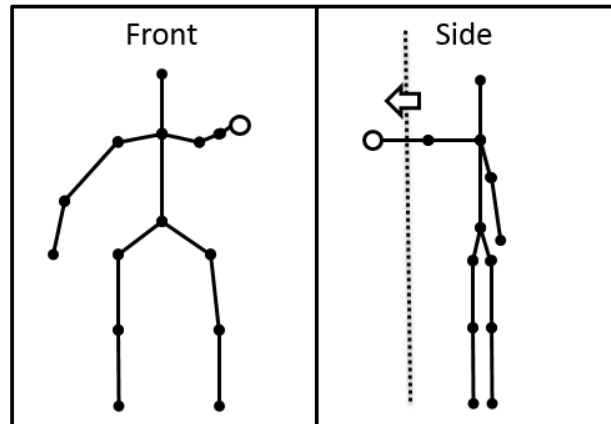


Figure 4.8 The push gesture

We formalize the definition through the equation 4.9, where the mH_r feature tracks the changes in the position of the right hand (for the left one the definition is symmetrical). The depth-barrier test is performed by the d predicate: it is true if the Z coordinate for the hand position was lower than the considered barrier value and false otherwise. The predicate \bar{d} is the logical negation of d .

$$mH_r[\bar{d}] \gg mH_r[d] \quad (4.9)$$

4.2.4 Push back

The push-back gesture mimics the action for releasing a virtual on-air button. The gesture performance is symmetric to the one described in the previous section: this time the user pulls-back the hand from the depth barrier.

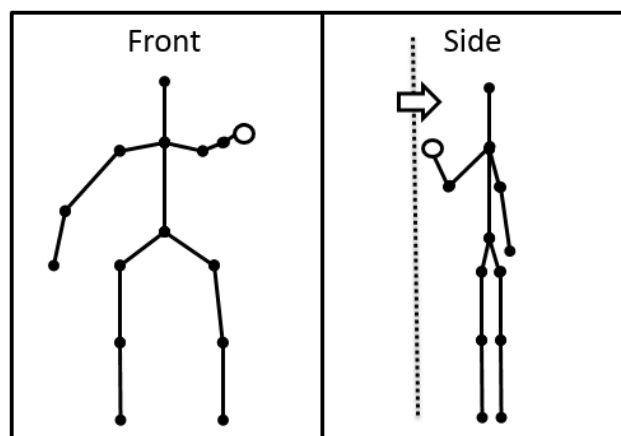


Figure 4.9 The push-back gesture

Figure 4.9 graphically shows how the push back gesture is performed, with the depth barrier represented by a dotted line in the side view.

The gesture modelling with our meta-model notation is symmetric with the one discussed in the previous section, and it is shown in equation 4.10. This time, the first ground term accepts values that are lower than the depth barrier value (modelled with the d predicate), while the second one accepts values that are greater than the depth value (the \bar{d} predicate).

It is worth pointing out that the value of the depth barrier needs to be updated according to the user's position. For instance, it is possible to consider a relative displacement calculated on the position of Z coordinate of the hip center joint.

$$mH_r[d] \gg mH_r[\bar{d}] \quad (4.10)$$

4.2.5 Lateral push

The lateral push gesture is equivalent to the push gesture, the only difference is the change of the axis for defining the barrier, which relies no more on the depth axis but on a value defined on the X axis.

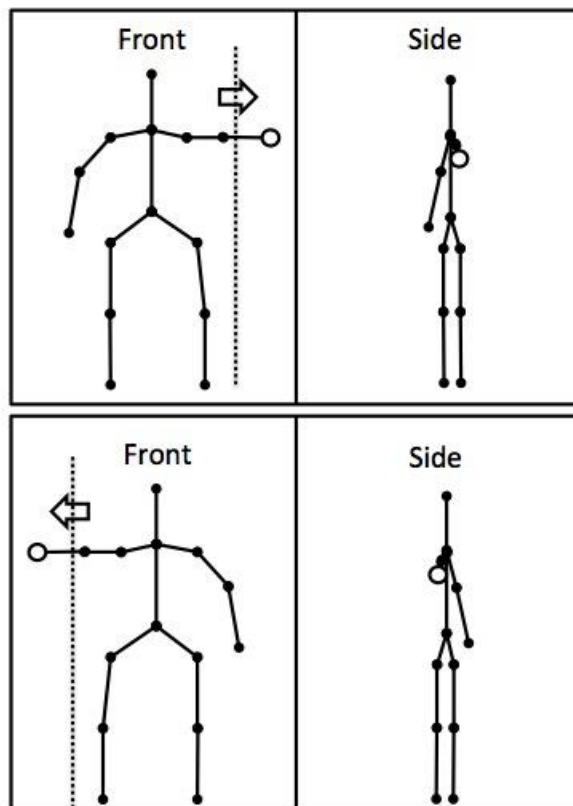


Figure 4.10 Lateral push gesture

Figure 4.10 shows the performance of the lateral-push gesture. From the modelling point of view, it is possible to reuse the definition in the equation 4.9, changing the definition of the d predicate.

4.2.6 Kick

The kick gesture, as the name already explains, consists in recognizing when the user mimics a kick for interacting with the application. As it depicted clearly by Figure 4.11, the recognition of this gesture can be defined through the same patterns we use for recognizing the push gesture (front or lateral): we again set a depth barrier and the gesture is completed when it is crossed by the considered foot.

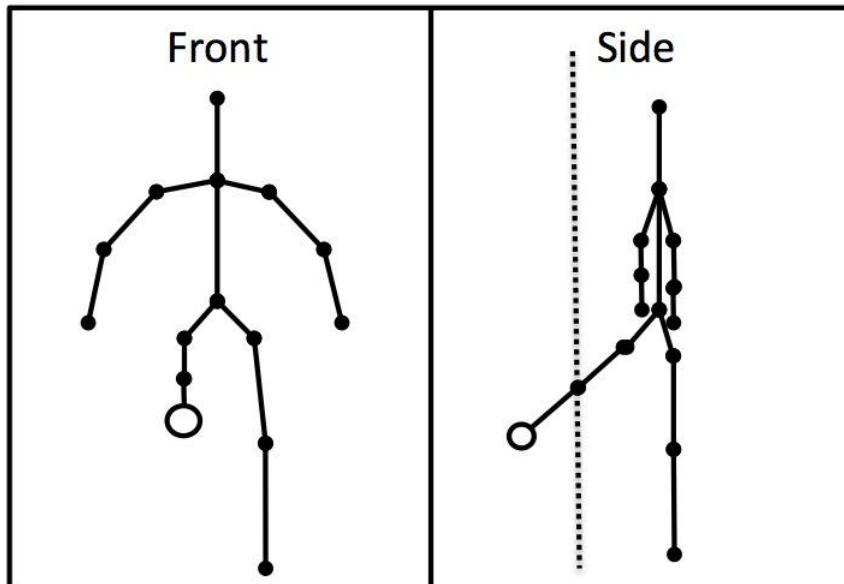


Figure 4.11 The kick gesture

Equation 4.11 shows the definition of a GestIT expression for the kick gesture. The mF_r is the feature for the right foot (symmetrically it is possible to define the same gesture for the left one), which has to be detected first outside the depth barrier (represented by the d predicate) and then inside it (the \bar{d} predicate).

$$mF_r[d] \gg mF_r[\bar{d}] \quad (4.11)$$

4.2.7 Wave

The wave gesture is commonly used by people to say hello and goodbye from a distance, simply moving one hand. Different applications exploit this

gesture for communicating the intention of the user to interact with them, using a “greet the screen” metaphor, especially in games for Xbox.

In order to define an expression for recognizing it, we consider the algorithm described in [138].

The gesture recognition phases are depicted in Figure 4.12. We describe the recognition of the wave gesture for the right hand, but it can be defined symmetrically also for the left hand. For convenience, in this paragraph we exploit a different coordinate system for the hand point: we set its origin on the elbow of the considered hand, preserving the orientation for the axes in Figure 3.10. Such coordinate system can be obtained at each frame simply defining a translation of the original coordinate system, using as vector the one defined by the currently tracked elbow position.

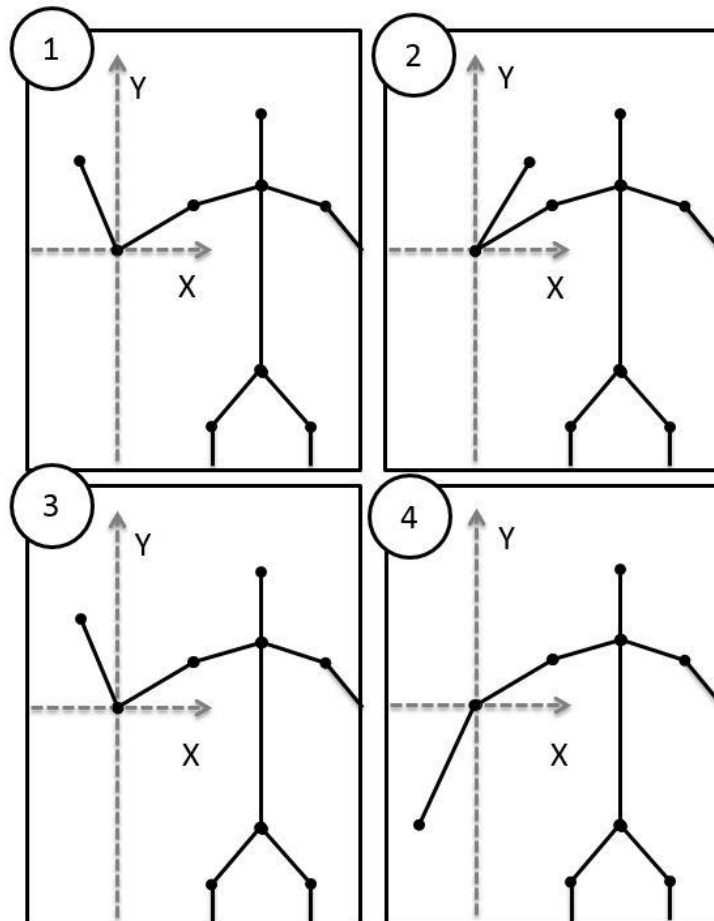


Figure 4.12 The wave gesture

The gesture starts when the hand point reaches the second quarter in our coordinate system, with a positive Y and a negative X value. The situation is depicted in Figure 3.10, part 1. Then, the user has to move the hand in

the first quarter of the coordinate system, with both values of X and Y positive, as shown in Figure 3.10, part 2. After that, the hand has to return in the second quarter (Figure 3.10, part 3). At this point, there are two alternatives: either the user repeats the wave, returning to the situation in Figure 3.10 part 2 and then back to hand position in Figure 3.10 part 3, or she can conclude the gesture moving the hand in the third quarter, as depicted in Figure 3.10, part 4.

In order to model this gesture with the GestIT notation, we define four different predicates, to be applied to the feature that describes the position of the right hand (mH_r):

1. x is true if the hand point has a positive value for the X coordinate
2. \bar{x} is true if the hand point has a negative value for the X coordinate
3. y is true if the hand point has a positive value for the Y coordinate
4. \bar{y} is true if the hand point has a negative value for the Y coordinate

With such predicates, we can model the recognition of the hand position as follows:

- $mH_r[\bar{x} \wedge y]$ recognizes the hand in the second quarter
- $mH_r[x \wedge y]$ recognizes the hand in the third quarter
- $mH_r[\bar{x} \wedge \bar{y}]$ recognizes the hand in the fourth quarter

Having defined the different parts of the gesture, we can compose them using the temporal operators in order to obtain the wave. For defining such temporal relationships, we have to consider that, during the gesture performance, the hand position inside the different quarters changes an indefinite number of times. Therefore, the recognition of each gesture subpart is iterative, it has to be executed at least once, and it is stopped by the recognition of one of the other components.

The equation 4.12 shows the definition of the wave gesture with the GestIT notation, and it clearly shows the four phases of the gesture. The first ground term corresponds to the first gesture phase. The iterative hand movement inside the second quarter of our coordinate system is disabled by the expression for the phases 2 and 3, contained in round brackets, which correspond respectively to the $mH_r[x \wedge y]$ and the $mH_r[\bar{x} \wedge y]$ ground terms.

These phases can be repeated an indefinite number of times (the user can wave more than once), but they have to be completed at least once.

Finally, the user put down the hand, positioning the hand point inside the third quarter, modelled by the $mH_r[\bar{x} \wedge \bar{y}]$ ground term.

$$\begin{aligned}
& mH_r^{\{1,*\}}[\bar{x} \wedge y] \text{ [}> \\
& (mH_r^{\{1,*\}}[x \wedge y] \text{ [}> mH_r^{\{1,*\}}[\bar{x} \wedge y] \text{]})^{\{1,*\}} \text{ [}> \\
& mH_r[\bar{x} \wedge \bar{y}]
\end{aligned}
\tag{4.12}$$

4.2.8 Swipe

The swipe gesture is a rapid movement of one hand, in a direction roughly parallel to the X or Y axis. In this paragraph, we consider a swipe on the X axis with the right hand, but it is easy to modify the definition to obtain any combination of hand and axis for recognizing all the variants for this gesture.

The gesture performance is depicted in Figure 4.13: the user moves her hand rapidly maintaining it in the area between the two dotted lines. Obviously, it is possible to define different tolerance thresholds for both the height of the area and the movement speed, in order to fine-tune the recognition.

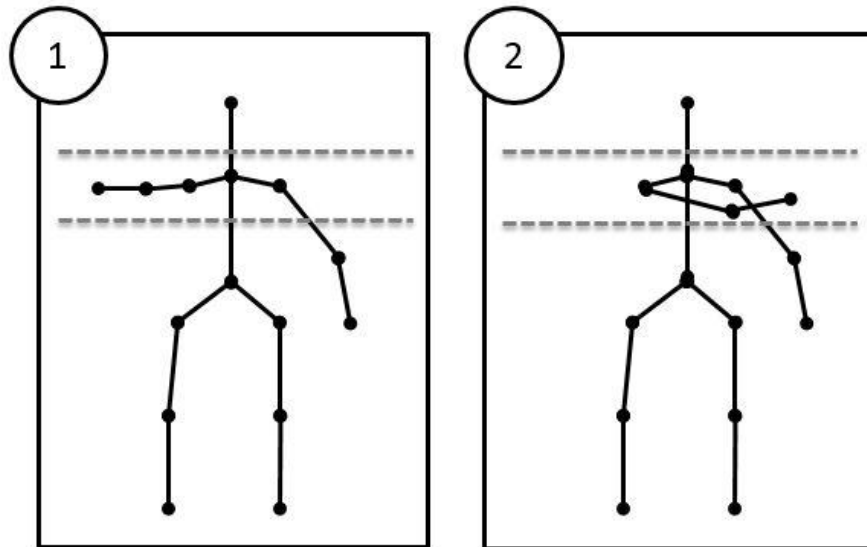


Figure 4.13 The swipe gesture

It is possible to model the gesture using the expression in equation 4.13. The swipe gesture is simply an iterated hand movement, which is constrained to be in an area with a specific height (modelled by the *linear* predicate) and with a speed higher than a specific threshold (modelled by the *speed* predicate).

It is possible to specify that the recognition of such hand movement has to be repeated at least a given number of times, changing the iterative operator with the second one of the short-hands we defined in section 3.1.2.7.

Finally, the first hand movement that do not satisfy the constraints disables the iteration, concluding the gesture.

$$mH_r^*[linear \wedge speed] [> mH_r \quad (4.13)$$

4.2.9 Walk

The walk gesture is an in-place imitation of the movements we perform while walking. In literature, we can find two different types of such imitation.

In the first type, the user mimics the walking movement raising alternatively the left and the right foot. Such kind of definition is exploited for instance in [74] and [33].

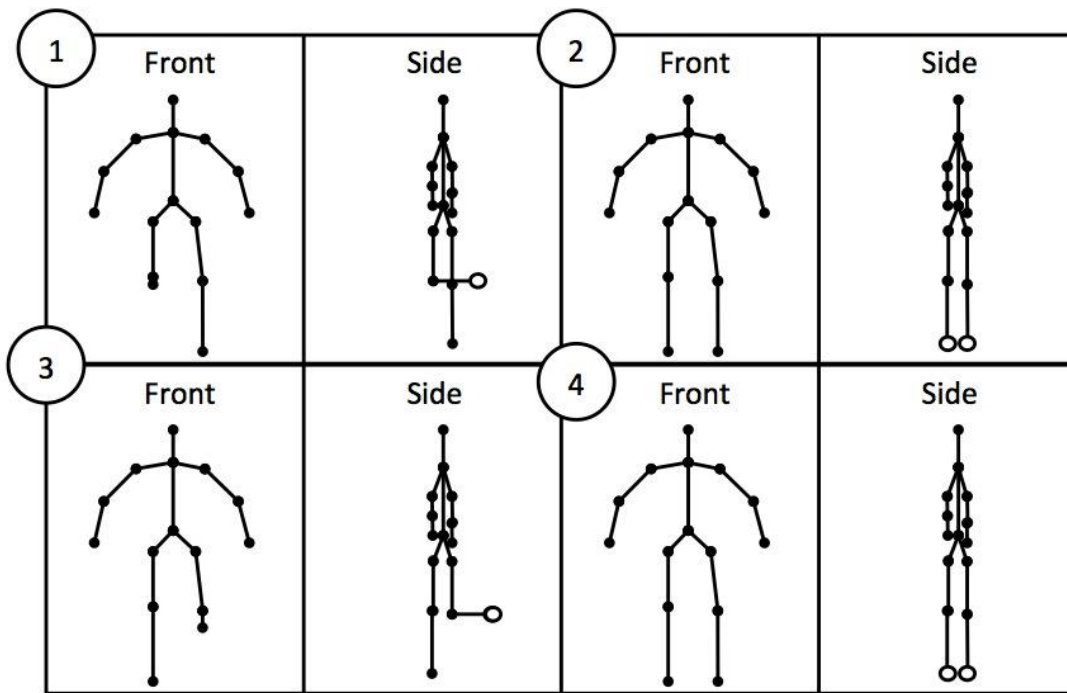


Figure 4.14 Walk gesture

The second one is more recognition-oriented and tries to mimic the walking movements with less physical effort for the users. Indeed, it consists of simply putting one of the feet forward with respect to the other one. This definition is exploited for instance in [67] and [24].

We define a GestIT expression for both variants here. The first one is depicted in Figure 4.14.

The gesture performance can be decomposed in four phases, each one depicted by a number in Figure 4.14.

In the first phase, the user raises the first foot (we consider the right one here, but the order is not fixed) until it reaches the height of the knee. After that, the same foot has to return to the rest position. The other two phases are symmetric: the user raises the second foot (the left one in our example) to the knees height and then she returns to the rest position.

We can model each one of the different phases with a GestIT expression, considering an iterative movement of the foot point (right for the first two, left for the other ones) that is disabled by reaching the position that concludes the considered phase. Therefore, we can define the follow four expressions:

1. $mF_r^* [> mF_r[*rightUp*]$, where *rightUp* is a predicate that tests if the right foot is in the position depicted in Figure 4.14, part 1.
2. $mF_r^* [> mF_r[*rest*]$, where *rest* is a predicate that tests if the right foot is in the position rest position depicted in Figure 4.14, part 2 and 4.
3. $mF_l^* [> mF_l[*leftUp*]$, where *leftUp* is a predicate that tests if the left foot is in the position depicted in Figure 4.14, part 3.
4. $mF_l^* [> mF_l[*rest*]$, where *rest* is a predicate that tests if the right foot is in the position rest position depicted in Figure 4.14, part 2 and 4

We can compose the four phases using the GestIT temporal operators in order to define the complete gesture. The phases 1 and 2, and symmetrically the phases 3 and 4, have to be executed in sequence: when the user starts raising one of the feet, she must raise it at the knee height and put it back in the rest position before starting the same movement with the other foot.

Obviously, there is no need to force the user to start with the left or the right foot, but we must ensure that the execution of the in-place steps is alternated between the right and the left foot. For this purpose, GestIT provides the order independence operator that, as defined in section 3.1.2.6, does not impose any order on the two operands but forces both of them to be completed in order to successfully recognize the entire expression.

The resulting GestIT expression for the first variant of the walk gesture is shown in equation 4.14.

$$\begin{aligned} & ((mF_r^* [> mF_r[*rightUp*] \gg mF_r^* [> mF_r[*rest*]]) \models | \\ & (mF_l^* [> mF_l[*leftUp*] \gg mF_l^* [> mF_l[*rest*]]))^* \end{aligned} \quad (4.14)$$

In the second variant of the walk gesture, the user put forward one of the feet with respect to the rest position. The recognition of this variant is quite similar to the one described for the kick gesture, the only difference is that the foot is not raised from the ground. Therefore, from the modelling point of view, the difference between the second type of the walk gesture and the kick gesture is simply the definition of the d predicate in the expression 4.11.

4.2.10 Turn

The turn gesture is a change in the user's position that, from being in front of the screen, turns the entire body either left or right. This gesture has been exploited in [67] for turning the field of view in a 3D-space control application. The same gesture has been exploited in [127] in order to distinguish when the is willing to interact with the application (and then she stays in front of the screen) from the situation where she was focused on cooking and her movements should not be tracked: if the turn gesture was recognized, the interaction tracking was disabled.

The recognition of this gesture is quite simple, and it is based on a comparison of the position of the two shoulders points. If they are both on a plane that is roughly parallel to X axis, we can consider that the user is in front of the screen (or, more precisely, in front of the sensor).

Starting from this consideration, it is simple to define how to recognize the gesture: we need simply to track the movements of the two shoulders and as long as they are no more on the aforementioned parallel plane, the gesture is recognized. Figure 4.15 depicts the performance of the turn gesture.

In order to model this gesture using the GestIT notation, it is sufficient to track the shoulder movements in parallel, checking the position of the shoulders at each movement. The gesture can be modelled in two steps: in the first one the user's shoulder has to be parallel to the X axis of our coordinate system, represented by the p predicate. The shoulders can move independently (even if they actually do not move independently, but we can abstract from such correlation), modelled by the parallel operator. The movement of one shoulder (or both) that does not fulfil the predicate constraint ends such situation (\bar{p} predicate). The complete definition is

shown in equation 4.15, S_l and S_r represent respectively the feature associated to the left and the right shoulder points.

$$(S_l[p] || | S_r [p])^* [> (S_l[\bar{p}] || | S_r [\bar{p}]) \quad (4.15)$$

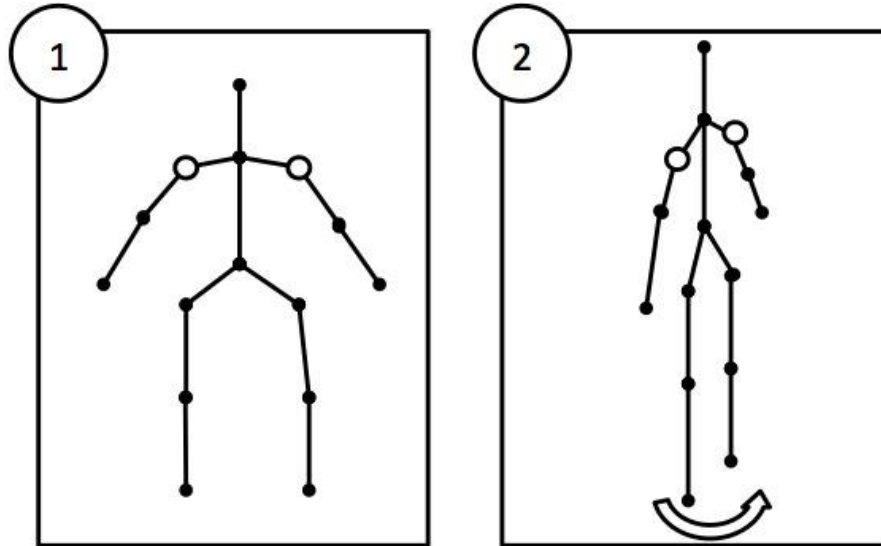


Figure 4.15 The turn gesture

4.2.11 Converge or Diverge Hands

In different work in literature, it is possible to find the definition of a full-body gesture for controlling the zoom level of a 2D or 3D view with an interaction style similar to the pinch gesture for multitouch screens. The touches are replaced with the position of the hands and the zoom level is controlled through the current distance between them: if it increases during the movement, the view is zoomed-in otherwise is zoomed-out.

The main difference with the multitouch counterpart of this gesture is the way we establish when it starts. For multitouch screens is straightforward: the gesture starts when the user touches the screen. Instead, for the full-body gesture we have two main alternatives. The first one is relying on the depth barrier concept we introduced for instance for the push: the gesture begins when both hands cross a given depth threshold (e.g. [45] and [79]). The second one exploits the recognition of the hand closure, and the gesture starts when the user closes both hands (e.g. [78] and [43]).

The gesture performance, based on hands closure, is shown in Figure 4.16, but it is obviously similar also for the depth barrier case.

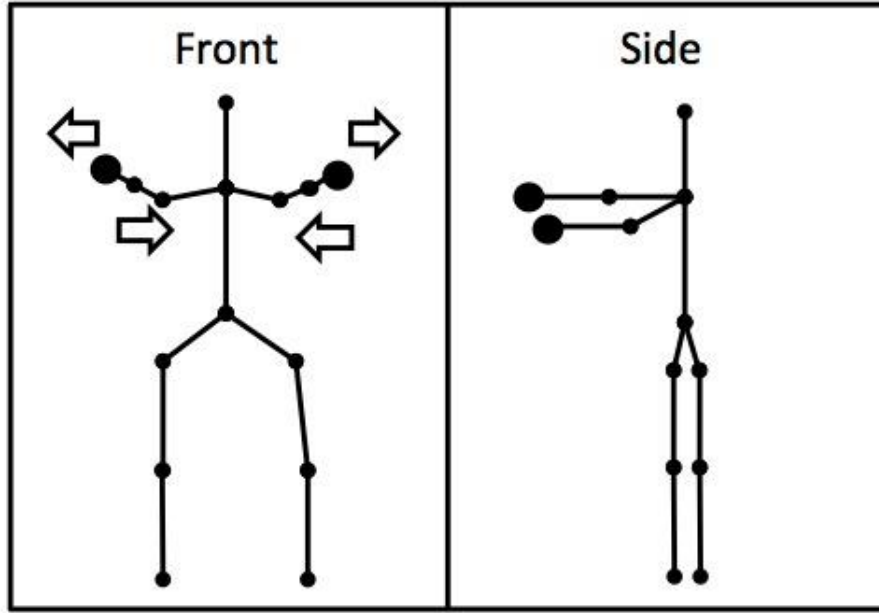


Figure 4.16 Convergence or diverge hands gesture

The expression for modelling this gesture is quite similar to the one shown in equation 4.5. Indeed, the gesture is defined exploiting the same temporal relationships, but we substitute the multitouch feature with the full-body ones, according to the different variations we are considering.

If we exploit the depth barrier, we have the GestIT definition in the upper part of the equation 4.16: the gesture starts when both the right and the left hands cross the depth barrier (modelled with the d predicate), continues with an iterative movement of the hands “inside” the barrier and then finishes withdrawing the hands from the barrier (the logical negation of d holds).

The second variant works without considering the position of the hands in the depth axis. The gesture starts when the user closes both hands (represented by the c predicate), continues with a parallel hand movement and it is ended when the user reopens the hands (and the logical negation of c holds). The GestIT definition for this variant is shown in equation 4.16.

$$\begin{aligned}
 & (mH_r[d] \mid = \mid mH_l[d]) \gg ((mH_r^*[d] \mid \mid mH_l^*[d]) \\
 & \quad [> (mH_r[\bar{d}] \mid = \mid mH_l[\bar{d}])) \\
 & (oH_r[c] \mid = \mid oH_l[c]) \gg ((mH_r^*[c] \mid \mid mH_l^*[c]) \\
 & \quad [> (oH_r[\bar{c}] \mid = \mid oH_l[\bar{c}]))
 \end{aligned} \tag{4.16}$$

4.2.12 Steering wheel

Together with the definition of a pinch equivalent for the full-body gesture recognition support, different work defined a full-body equivalent for the rotation gesture for multitouch. What we call “steering wheel” is exactly such equivalent, which can be found in literature with the same variants described for the previous gesture: based on the depth barrier [45] or on hand closure [78][43].

The gesture performance consists in mimicking a rotation of the hands along a circular path, as if the user holds a steering wheel. The gesture is depicted in Figure 4.17, the dotted line shows the path that constrains the hand movements. In practice such area cannot be defined simply as a circle, but it must contain an outer and an inner circle where the hands can move, tolerating some degree of deviation from a perfect trajectory.

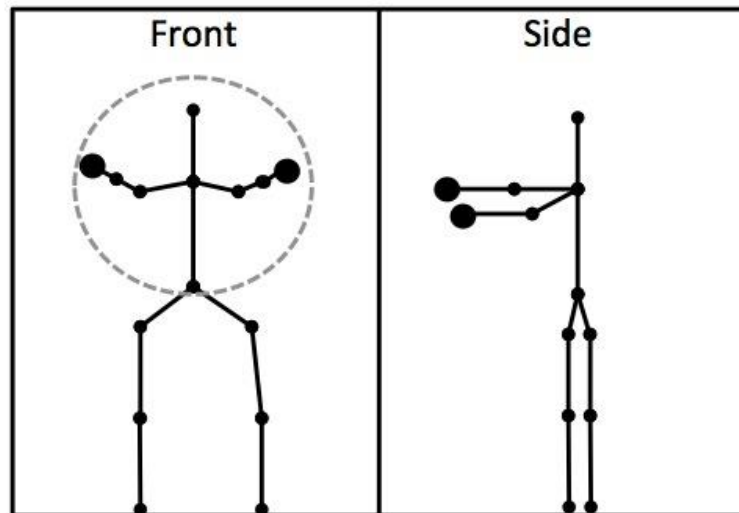


Figure 4.17 Steering wheel gesture

It is possible to model such gesture in GestIT through an expression similar to the converge or diverge gesture. The two definitions differs only on the path that constrains the “steering wheel”: we add a *circle* predicate for checking such property to the iteration of the parallel movement of both hands.

We keep the two different definitions also in this case: one for the depth barrier and one for the hand closure exploitation for starting and finishing the gesture. The expression is shown in equation 4.17.

$$\begin{aligned} & (mH_r[d] = |mH_l[d]) \gg \\ & ((mH_r^*[d \wedge circle] || mH_l^*[d \wedge circle]) [> (mH_r[\bar{d}] = |mH_l[\bar{d}])) \end{aligned} \quad (4.17)$$

$$\begin{aligned} & (oH_r[c] = |oH_l[c]) \gg \\ & (mH_r^*[circle] || mH_l^*[circle]) [> (oH_r[\bar{c}] = |oH_l[\bar{c}])) \end{aligned}$$

4.2.13 Roll

The roll gesture is similar to the grab one defined in section 4.2.2: The difference is that performs the on-air grab with two hands. The user has to close both hands before moving them.

As we already explained for other gestures, when the information on the hand closure is not available it is possible to exploit a depth barrier technique for starting the recognition.

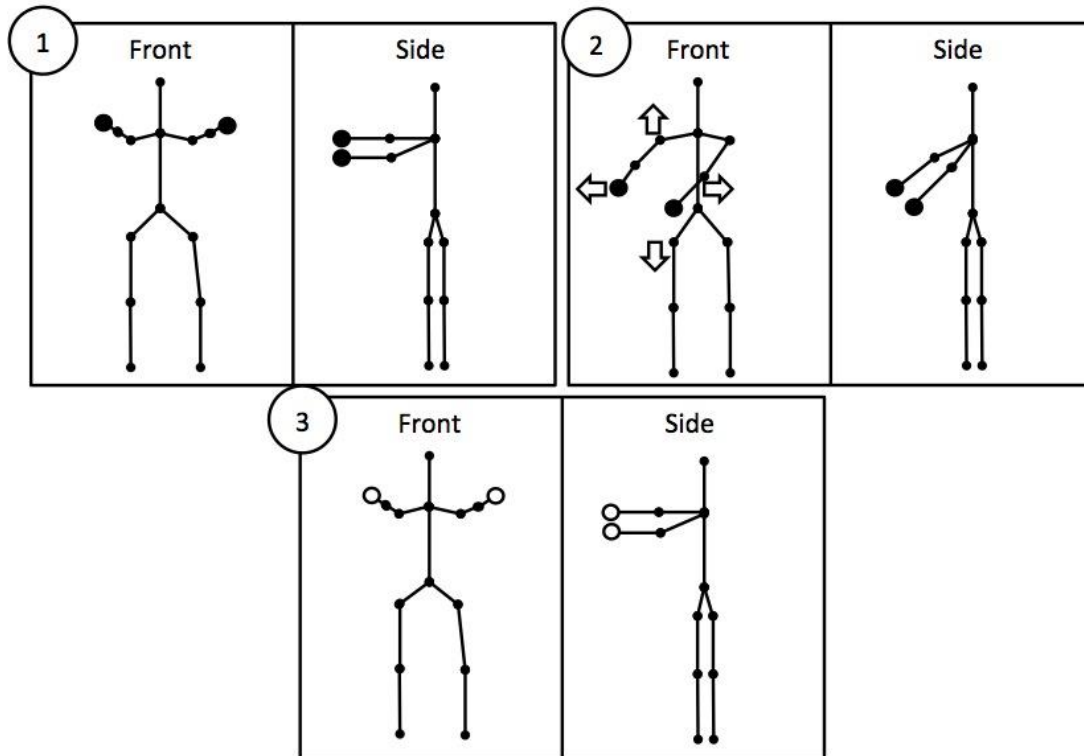


Figure 4.18 Roll gesture

In addition, the two hands cannot move independently: they should be maintained close to each other, as if the user holds a stick during the movements. This kind of gesture is exploited in [45] and [125] for rotating a 3D models using a virtual trackball mechanism [32], and it was included by Franke et al. [43] in their benchmark gestures.

The definition of a GestIT expression for recognizing this gesture is similar to the one we used for the steering wheel gesture. The main difference is the predicate that constraints the parallel movement of the hands: in this case we have to ensure that the distance between the hands remains roughly the same for the whole gesture performance. For this purpose, we define a *dist* predicate, which performs this check.

In addition, we define two variants for this gesture, one that exploits a depth barrier for recognizing when interaction start and one that exploits the hand closure. The resulting expressions are shown in equation 4.18.

$$\begin{aligned}
 & (mH_r[d] = |mH_l[d]) \gg \\
 & ((mH_r^*[d] \wedge dist) || mH_l^*[d] \wedge dist) [> (mH_r[\bar{d}] = |mH_l[\bar{d}])) \\
 & (oH_r[c] = |oH_l[c]) \gg \\
 & (mH_r^*[dist] || mH_l^*[dist]) [> (oH_r[\bar{c}] = |oH_l[\bar{c}]))
 \end{aligned} \tag{4.18}$$

4.2.14 Universal Pause

The universal pause gesture is exploited in the Xbox games in order to pause the interaction. The gesture has been defined with the purpose to be an unnatural pose for the interaction, in order to reduce the accidental recognition of this gesture, with a consequent undesired pause [138].

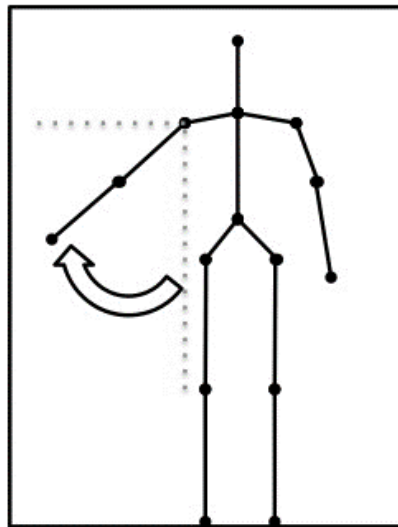


Figure 4.19 The Universal Pause gesture

The gesture performance is shown in Figure 4.19: the user has to maintain the position of the arm roughly at 45 degrees from the body. Obviously, it

is difficult that the user is able to hold the exact position, therefore the user will actually move the hand, but keeping it roughly in the same position. The other parameter to establish is how long the user has to wait before the recognition. It obviously depends on the designer's choice (e.g. two seconds).

In order to model the gesture with the GestIT notation, we have to consider the hand position as the tracking feature. We have also to ensure that the position of the hand is at roughly 45 degrees from the body, and for this purpose, we defined the *deg45* predicate. Moreover, we take into account the time spent by the user in this position, through the definition of a *time* predicate that checks whether the required time has passed or not.

The complete definition of the gesture is shown in equation 4.19: it iteratively recognizes the hand movement in the specified position until the time has passed.

$$mH_r^*[deg45 \wedge \overline{time}][> mH_r[time] \quad (4.19)$$

Chapter 5

Library Support

This chapter presents the implementation of a proof of concept library for the development of gestural interfaces according to the meta-model definition described in Chapter 3. The library is open source and it is publicly available at <http://gestit.codeplex.com/>.

In the first part, we discuss the overall library architecture, which includes the classes shared by the different gesture recognition platforms. After that, we show how the abstract classes are refined in for supporting multitouch and full-body gestures.

The second part of this chapter shows how it is possible to define gestures through the library, discussing some code samples for iOS (multitouch) and for the Microsoft Kinect (full-body).

The third part briefly discuss a set of applications developed exploiting the library for both the multitouch and the full-body platforms.

The last part of the chapter introduces the possibility of a cross-platform reuse of the gestures definition.

5.1 Library Architecture

We designed and implemented a proof-of-concept library starting from the meta-model defined in Chapter 3. The library architecture has been designed in order to isolate the definition of the temporal relationships once for all the supported platforms. Therefore, we created an abstraction layer that is exploited by all recognition supports.

In this way, it is easy to add the support for new recognition platforms, since there is no need to redefine how to compose expressions and the semantics of the temporal operators.

According to this idea, the library consists of different packages, one for each supported platform, which share the implementation of the temporal relationships between gestures.

The library class diagram is shown in Figure 5.1. It has a core independent from the actual gesture recognition support, which is contained in the *core* package. Each platform is an extension of the core package and it deals with an actual device. The ones that are currently supported are iOS and Android devices (*multitouch* package) and Microsoft Kinect (*fullBody* package).

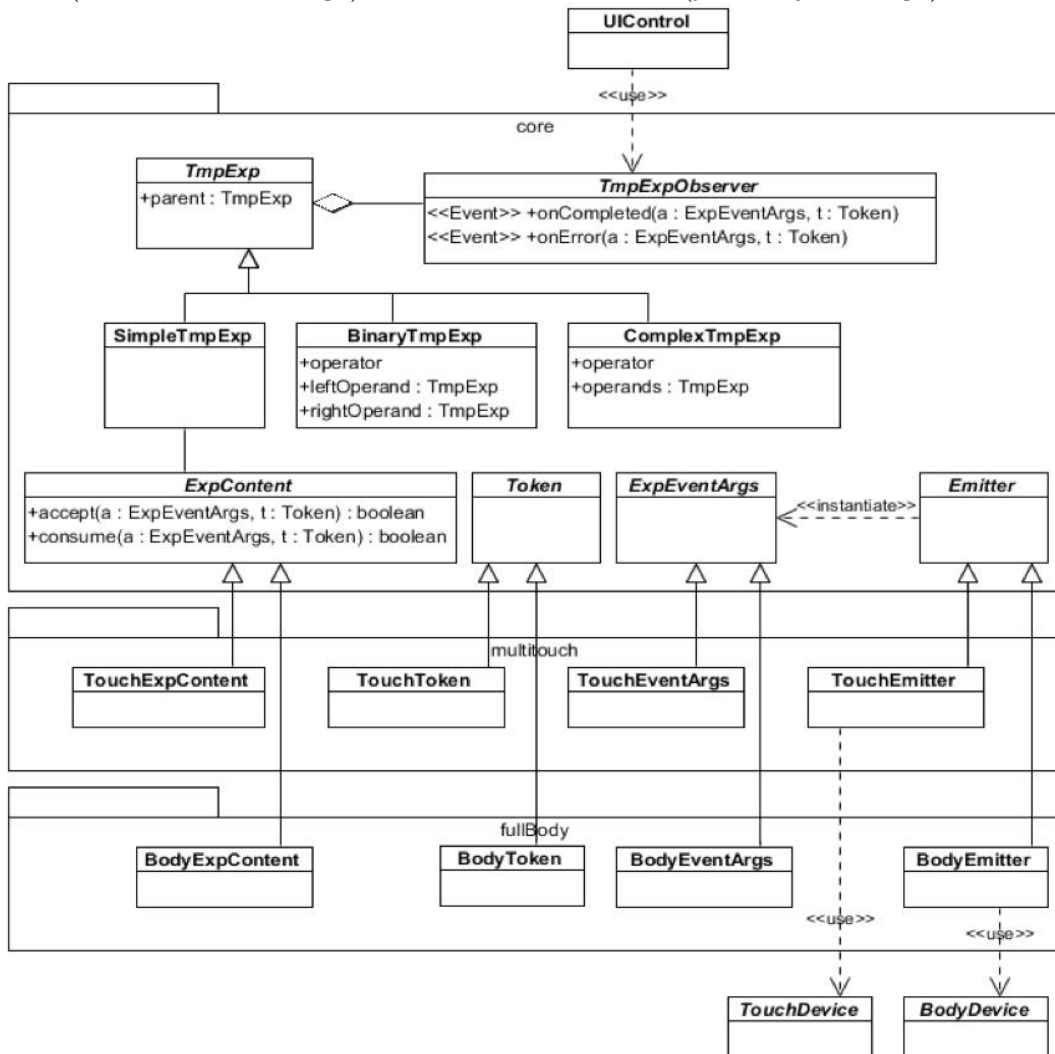


Figure 5.1 GestIT class diagram

5.1.1 Library core

The library core contains the classes for the defining gesture expressions. The abstract class **TmpExp** represents such expressions, either ground terms

or composed ones. The class contains a composition of *TmpExpObservers*, which define the protocol for receiving notifications about the recognition of the gesture expression. It is possible to receive two types of notifications:

- *onCompleted*, which notifies the successful completion of the gesture expression;
- *onError*, which notifies that it was not possible, given the current gesture state sequence, to recognize the gesture expression.

Both events are parametric on two abstract classes, the *ExpEventArgs* and the *Token*. The first one is the extension point for the information about the current gesture recognition support state that contains, as we defined in section 3.1.1, the value of all the features recognizable by the considered device.

The second one instead maintains the gesture recognition state sequence, which is the history of the previously sensed features. As we describe in more detail during the discussion of the platform refinements, it is not feasible for the concrete implementations of the *Token* class to maintain the whole history of the feature values. Therefore, the implementation provides the developer with mechanism to control the amount of information to maintain.

The iterative operator is represented by a boolean flag on all *TmpExp* instances.

The subclasses of *TmpExp* refine the gesture expression concept with according to different roles.

The *SimpleTmpExp* class implements the Petri Net for recognizing a generic basic building block, and it is a subclass of *TmpExp*. The actual feature changes and the optional conditions on them (see section 3.1.1) are defined by a delegate object associated to the *SimpleTmpExp* instances, which are obviously device-dependent. Such delegate object must implement the *ExpContent* protocol, which is the second extension point defined for the *core* package. Such interface consists of two different methods:

- *accept*, which receives the current gesture recognition support state (represented by the abstract class *ExpEventArgs*) and the Petri Net *Token* that, for convenience, contains the information on the previous gesture recognition support state sequence. A concrete implementation of the delegate returns a boolean value indicating whether the feature change is recognized or not, according to the parameter values;

- *consume*, which allows to specify the amount of gesture data to be maintained during the gesture recognition. As we better detail in section 5.2, it is not feasible to maintain the entire sequence of feature values because of memory space, storing into a *Token* only the subset of the gesture support state sequence that is needed.

The possibility to combine building blocks and composed gestures is provided by other two *TmpExp* subclasses: *BinaryTmpExp* and *ComplexTmpExp*.

The first one implements all Petri Nets representing binary operators, namely sequence, parallel, choice, disabling. Obviously, an instance of this class behaves differently according to the *operator* property and its *left* and *right* operands, which belong to the *TmpExp* class. This make it possible to connect both building blocks and complex gestures.

The N-ary versions of such operators can be obtained associating the operands, exploiting the associative property of all the binary operators.

The second *TmpExp* subclass implements the Petri Net for the order independence and contains a list of *operands* (again belonging to the *TmpExp* class).

A gesture definition is represented by a *TmpExp* tree, where all leafs are *SimpleTmpExp* instances, while the other nodes belong either to the *BinaryTmpExp* or the *ComplexTmpExp* class. At runtime, the tree is managed by a device dependent implementation of the *Emitter* class. Its responsibility is to listen to device updates and to forward them to the leafs that currently contain a token. For each one of them, the *Emitter* invokes the *accept* method. If the return value is true, the *Emitter* calls the *consume* method. Then, the *SimpleTmpExp* notifies the recognition to its parent expression that, according to the Petri Net semantics, moves the *Token*, propagating the notification up to the tree hierarchy and proceeding with the gesture recognition. In section 5.2 we provide a concrete example of this mechanism.

It is possible that the device raises an update that is not *accepted* by any leaf. In this case, the gesture recognition should be interrupted, and the developer should have the possibility to define how the interface should react to the interruption. The library offers the possibility to associate a handler not only for the successful recognition of a gesture (either basic or composed), but also for the recognition failure (the aforementioned *onError* event of the *TmpExp* class). The recognition failure is also propagated to the upper levels of composition tree as in the successful case.

5.1.2 Multitouch package

In order to recognize multitouch gestures described through this formal definition with our library, we need to define the concrete implementation of the abstract classes discussed in section 4, represented as the *multitouch* package in Figure 5.1.

The first one is *TouchEventArgs*, an *ExpEventArgs* subclass, which contains the information about a device feature update (touch identifier, touch point, time). The instances of this class are created by a *TouchEventEmitter*, an *Emitter* subclass, which translates the touch screen updates into a format manageable by the library.

The *TouchEventArgs* instances are forwarded to the leafs of the *TmpExp* tree that, as already discussed in the previous section, are *SimpleTmpExp* instances. These leafs are connected with *TouchExpContent* instances, which are *ExpContent* refinements. The *TouchExpContent* class has two instance variables, which represent the touch identifier and the type of a basic building block for touch gestures (start, move and end).

Therefore, the *accept* method checks the conditions defined in equation 3.9, according to the specified type. Further conditions to be checked can be defined by developers sub-classing *TouchExpContent* and overriding the *accept* method. The *TouchToken* class contains the information on the gesture sequence, and represents the concrete implementation of a *Token*. Obviously, it is not possible to store in memory each single feature update especially when programming for mobile devices. Therefore, it is possible to specify the maximum number of updates to be buffered and, for convenience, if the starting point of each touch should be maintained or not.

5.1.3 Full-body package

The structure of the *fullBody* package is symmetric with respect to the *multitouch* one. The *BodyContent* class is the concrete implementation of the *ExpContent* for this package, and it defines all the possible ground terms for the full body platform through an enumeration that contains a value for each different joint type, joint orientation, the hands status (open or closed) and the time, as discussed in section 3.3.

The constraints on the recognition of a given feature are defined by the *accept* instance method. In the particular case of the C# implementation, we exploited a delegate method [88], which allow the developer to customize

the definition of a predicate on the given feature without the need of subclassing *BodyContent*.

The data manipulated by the expressions is contained into the concrete refinement of the *ExpEventArgs*, the *BodyEventArgs*. This class provides the information tracked by the Kinect for Windows SDK [87] to the *core* package, wrapping it in a format that can be manipulated by the library. It contains the current position and orientation of the skeleton joints, together with the information on the time feature.

The information on the gesture recognition state sequence is maintained into an object of the *BodyToken* class, a refinement of the *Token* class, in the same way we already explained for the multitouch platform. The objects of this class are able to maintain a finite number of gesture recognition support states, which can be specified at the moment of the object instantiation. This is obvious, since it is not possible to maintain the whole history of the features changes in memory.

The task to interface the Kinect device with the expression library is accomplished by the *BodyEmitter* class. It observes the changes of the device state through the API provided by the Kinect SDK [87] and, when a change occurs, it creates an object of the *BodyEventArgs* class that contains the same updates in a format that can be processed by the library, and forwards them to the expression tree that represents the gesture description definition.

5.2 Creating a multitouch application

We better clarify how a developer can use the library for providing multitouch gesture support for a UI control with an example. We consider a pinch gesture (defined in equation 4.5), exploited in a multitouch application that we detail in section 5.4.2.

We recall in the equation 4.1, the GestIT expression that defines the pinch gesture.

$$(Start_1 \text{ |=| } Start_2) \gg ((Move_1^* \text{ || } Move_2^*) \text{ [> } (End_1 \text{ |=| } End_2)) \quad (4.1)$$

The following are the steps that have to be followed by the UI control initialization code.

1. Construct the tree of *TmpExps* represented by the UML object diagram in Figure 5.2, starting from the leafs, and then associate

each *SimpleTmpExp* to the delegate for recognizing the desired feature

2. Create a *TouchToken* instance, specifying the number of updates to be buffered and whether the initial position of each touch has to be stored or not.
3. Create an instance of the *TouchTmpEmitter* class, passing the token created at step 2, and the current UI control (exploited in order to receive the touchscreen updates from the OS).
4. Attach the handlers to the completion and/or error event of the entire gesture and/or its subparts, represented by the instances of *TmpExps* created at step 1.

This initialization code at step 1 can be created directly by the developer with the considered programming language (e.g. Objective C or Java). Otherwise, it can be created exploiting an XML-like description of the gesture, which eases the definition of the gesture tree. As we better detail in the next section it is convenient to extend the interface description markup (e.g. XAML [89]) if available.

In addition, it is possible to store such initialization code in a separate class (e.g. *PinchTmpExp*) and reusing it for different UI controls.

The flow of notifications that allows the library to manage the recognition and to raise the appropriate intermediate events is shown with a sequence diagram in Figure 5.3. In Figure 5.2 we show the how such notifications are propagated in the object tree, visualizing the numbers of the messages in Figure 5.3 inside arrows.

The numbered arrows represent the sequence of notifications when the user touches the screen with the second finger, the squares represent the handlers attached to gesture sub-components, the solid circle represents the position of the token before the second touch, and the dotted circles the position of the token after the second touch. The lower part shows the effects of the attached handlers on the UI.

We suppose that net has already recognized a touch start with id 1. Therefore, it is waiting for another touch start, this time with id 2. Such “waiting” is defined by the token position (represented as a circle-enclosed T on the *s2* object in Figure 5.2).

When the touch screen senses a new touch, the *TouchEmitter* forwards such notification to *s2*, the tree leaf that currently contains the token (arrow 1). After that, *s2* tries to recognize the touch, invoking the *accept* method of its *TouchExpContent* delegate, which returns *true* (arrow 2). Then *s2*

notifies the successfully completion to its parent, $c1$, which represents the expression ($Start_1 \mid = Start_2$).

All the building blocks enclosed in this expression are recognized, thus the order independence expression is completed. Therefore, the event handler attached to $c1$ is executed. In our example, it paints two circles on the currently visualized image in correspondence of the touch points (A square in Figure 5.2), providing intermediate feedback to the user while executing the gesture.

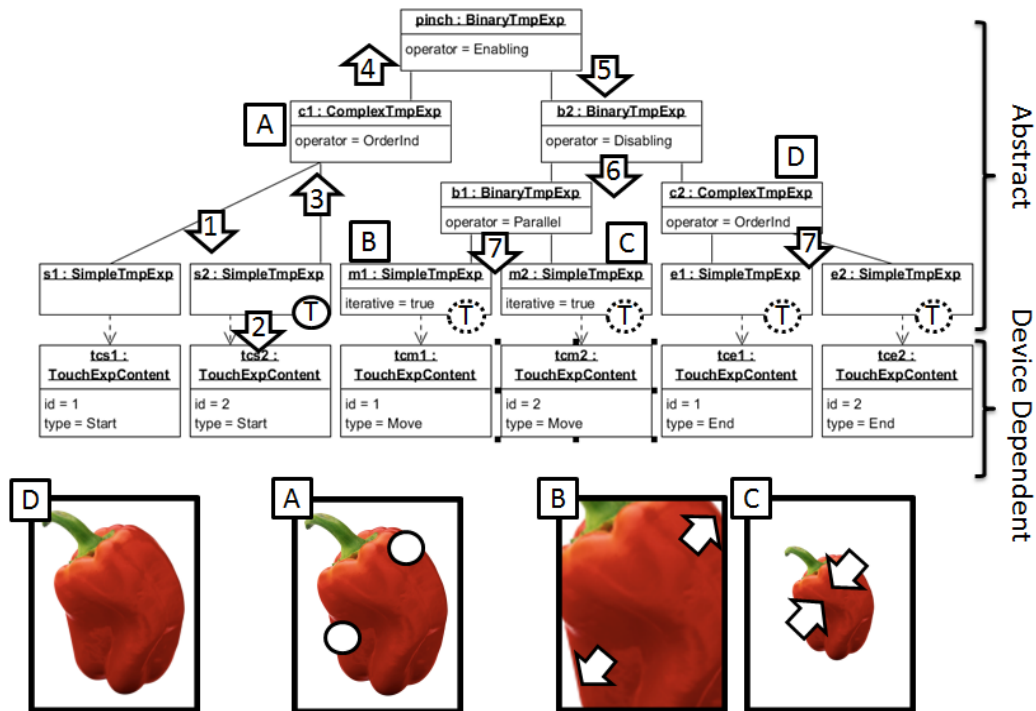


Figure 5.2 Recognition of a pinch gesture with the GestIT library

This is the point where our approach breaks the standard observer pattern: the gesture recognition is not already finished, but it is possible to define UI reactions to the completion of its sub-parts, without re-coding the entire recognition process, as happens for instance when a viewer has a built-in pinch for zoom gesture recognition.

After that, $c1$ notifies the completion to its parent, $pinch$ (arrow 4), which represents an enabling expression. Having completed its left operand, $pinch$ passes the token to its right operand $b2$ (arrow 5), which represents a disabling expression, and $b2$ passes the token to both its operands (arrow 6), which both duplicate it (arrow 7) at next step. The left one represents a

parallel expression, while the right one represents an order independence (see section 3.2.3 and 3.2.5).

Finally, we have four different basic gestures that can be recognized as next ones: touch 1 move, touch 2 move, touch 1 end or touch 2 end. The dotted circles in Figure 5.2 represent the new token positions.

It is worth pointing out that the device dependent part of the recognition support is concentrated on delegates for the *SimpleTmpExp* object (represented at the bottom of the tree in Figure 5.2). Therefore, the remaining part of the support is implemented by classes that are not bound to a specific device (identified by the “Abstract” label in Figure 5.2) and can be exploited not only for multitouch, but also for full body gestures and other recognition supports.

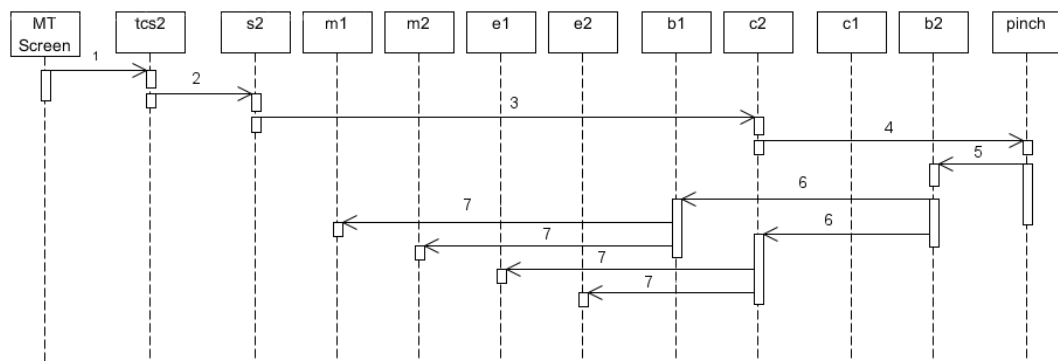


Figure 5.3 Recognition of a pinch gesture (sequence diagram)

As already mentioned at the beginning of this section, the example discussed here is a part of an iOS proof of concept application that allows zooming the current view through the pinch gesture and drawing with a pan gesture. The application gives intermediate feedback during the pinch, showing two divergent arrows while zooming in and two convergent arrows while zooming out (respectively square B and C in Figure 5.2). The two gestures are composed through the parallel operator, so it is possible to draw and to zoom the view in at the same time (e.g. using one hand for zooming and one for drawing).

From the developer point of view, the difference in handling them at the same time or separately is a matter of selecting the choice or the parallel operator for the composition. No further code is required, which is not the case for current multitouch frameworks. In addition, both gestures have been defined separately from the application (they are contained as samples in the iOS library implementation) and nevertheless the developer can

associate UI reactions at different levels of granularity (to the whole gesture, or part of it).

5.3 Creating a full-body gesture application

In this section, we detail how a developer can use GestIT in order to create a full-body gestural UI.

We consider here a touchless recipe browser application that, as we detail better in section 5.4.4, is organised into three presentations: the first one allows the user to select the recipe type (e.g. starter, first dish, main dish etc.), the second one is dedicated to the selection of the recipe, while the last one presents the steps for cooking the selected dish with video and subtitles.

In the latter presentation, it is possible to go through the steps back and forth or to jump randomly from one point to the other of the procedure.

We consider here the C# version for Windows Presentation Foundation (WPF) of the GestIT library. An interface in WPF is described by two different files. The first one contains the definition of the UI appearance and layout specified using XAML [89], an XML-based notation that can be used in .NET applications for initializing objects. In this case, it initializes the widgets contained into the application view.

The second file involved in the UI definition contains the behaviour, and it is a normal C# class file. Since the two files are part of the same view class definition, the latter is called the “code-behind” file. Objects defined by the XAML file are accessible in the code-behind file and the methods defined in the code behind file are accessible in the XAML definition.

In this example, we discuss the implementation of the first presentation, which is shown in Figure 5.4. The view is composed of a title on the upper part and a fisheye panel in the centre. The bottom part is dedicated to the status messages: the application notifies if it is tracking the user’s movements or not.



Figure 5.4 Touchless recipe browser, dish type selection

The gestural interaction is defined inside the associated view through a set of custom XAML tags, which are shown in Table 5.1. The tree structure of the tags is equivalent to the expression notation we used in Chapter 3.

The high-level description of the gesture interaction is the following: if the user is not in front of the screen, the application does not track her movements. When the user is in front of the screen, she can highlight one of the recipe types, which can be selected by a grab gesture (closing the hand). The definition of such gestural interaction is highlighted with comments in Table 5.1.

The interaction is a *sequence* of different gestures, which starts with the user that stands in front of the screen (a turn gesture, from line 7 to line 11). This is modelled checking the position of the shoulder points, which have to be almost parallel to the sensor plane on the depth axis (see section 4.2.10). Such constraint is modelled using a predicate associated to the left shoulder ground term, which is specified by the *Accepts* attribute containing a value the name of the C# method that calculates it (*screenFront*). The latter method is defined in the code-behind file associated to a XAML specification.

When this gesture is completed, the user needs to be aware that the application is tracking her position, therefore the completion method associated to the gesture changes the message on the label at the bottom of the UI in Figure 5.4, setting its text to “Tracking User” with a green background.

```

1 <TabItem Background="#FF92BCED" x:Name="recipeType">
2   <Grid Background="#FF92BCED">
3     <!-- gesture definition -->
4     <g:GestureDefinition x:Name="moveSelection" >
5       <g:Sequence Iterative="True">
6         <!-- turn gesture (front of the screen) -->
7         <g:Change Feature="ShoulderLeft" Accepts="screenFront">
8           <g:Change.Completed>
9             <g:Handler method="screenFront_Completed"/>
10          </g:Change.Completed>
11        </g:Change>
12        <g:Disabling>
13          <!-- grab gesture -->
14          <g:Disabling Iterative="True">
15            <g:Change Feature="HandRight" Iterative="True">
16              <g:Change.Completed>
17                <g:Handler method="moveHand_Completed" />
18              </g:Change.Completed>
19            </g:Change>
20            <g:Change Feature="OpenRightHand" Accepts="rightHandClosed">
21              <g:Change.Completed>
22                <g:Handler method="rightHandClosed_Completed"/>
23              </g:Change.Completed>
24            </g:Change>
25          </g:Disabling>
26          <!-- turn gesture (not in front of the screen) -->
27          <g:Change Feature="ShoulderLeft" Accepts="notScreenFront">
28            <g:Change.Completed>
29              <g:Handler method="notScreenFront_Completed"/>
30            </g:Change.Completed>
31          </g:Change>
32        </g:Disabling>
33      </g:Sequence>
34    </g:GestureDefinition>
35    <!-- view definition -->
36    <ui:FisheyePage x:Name="heading" Grid.ColumnSpan="2" />
37    <kt:KinectSensorChooserUI Grid.Column="0" Grid.ColumnSpan="2"
38      Name="kinectSensorChooser1" VerticalAlignment="Center" Width="328"/>
39  </Grid>
40 </TabItem>

```

Table 5.1 XAML Gesture definition

The definition of this behaviour is again in the code-behind file, and it is linked with the gesture declaration through the *method* attribute for the *handler* element inside the *change.completed* tag (line 8 and 9 in Table 5.1). The method name in this case is *screenFront_Completed*.

Once this gesture is completed, it is possible to interact with the screen, and the grab gesture implements the selection of the recipe type. First, we listen iteratively to the change of the right hand position (the *Change* tag with *Feature="HandRight"* at line 15 in Table 5.1, which implements a pointing gesture). Every time such ground term is completed, (read the user moves the hand), the *moveHand_Completed* method is executed. It updates

the currently highlighted recipe type (the one with the red border in Figure 5.4).

The recognition iteration may be interrupted in two cases. The first one is when the user closes the right hand (the *Change* tag with *Feature=“OpenRightHand”* at line 20 in Table 5.1) and the method *rightHandClosed_Completed* handles the completion of the grab gesture, changing the current presentation.

The second case is when the user goes away and she is not in front of the screen anymore (the turn gesture at line 27 in Table 5.1). This situation is modelled symmetrically with respect to the gesture at line 7, the only difference is the *Accepts* method (*notScreenFront*), which is exactly the logical negation of *ScreenFront*. In both cases, the interruption is modelled using a *disabling* operator, declared respectively by the inner and the outer *Disabling* tags (respectively at line 14 and 12 in Table 5.1).

As it should be clear from the description, in order to create a gestural interface with GestIT in XAML is sufficient to:

1. Create the UI view
2. Define the gestures associated to a view (in the same file), composing declaratively existing gestures or creating new ones starting from ground-terms.
3. Provide the methods for calculating the predicates associated to the specified gestures in the code-behind file (if any)
4. Provide the UI behaviour associated to the gesture completion

5.4 Sample applications

In this section, we provide the description of the different applications that developed as showcases for the GestIT library.

5.4.1 Pilot study: Simple canvas

The first example we discuss is a simple drawing application for both the multitouch and the full-body gesture recognition supports, which we exploited in order to drive the design of the gesture meta-model we defined in Chapter 3, with a proof-of-concept implementation. The preliminary results for this pilot study were discussed in [126].

We used two different supports and SDKs, such as the iPhone and the iOS SDK [5] and the Microsoft Kinect with the NITE framework [115], since

it was created before a stable version of the Microsoft Kinect SDK was released. The application is a simple canvas where the user can draw with her finger in the iPhone or with one hand in the Kinect version.

Though the applications are really simple, they have two important things in common. The first one is the support for the temporal operator definition, shared by both versions, which have been initially developed in C++ and compiled for both platforms.

The second one is the gesture definition: we selected for this sample application the *Pan* (see section 4.1.3) and the *Pointing* (see section 4.2.1) gestures for drawing respectively for multitouch and full-body, which can be considered two equivalent gestures in the two different platforms.

For the same reason, we selected the *Pinch* for multitouch (see section 4.1.5) and the *Diverge or Converge Hands* for the full-body (see section 4.2.11) in order to implement the zoom feature.

In both versions, the gestures are connected through the choice operator, showing already the main advantage of our modelling technique: we can reuse the definition of two gestures and combine them in order to obtain a more complex interaction.

In addition, with the same definition it is possible to support the zooming feature while drawing in the multitouch version changing only the composition operator (*Parallel*), without any additional effort for the developer.

The equation 4.1 shows the definition of the gestures for the simple canvas application: the first two expressions model the multitouch application, while the third one defines the full-body interaction.

The definitions show how the *Pan* and the *Pinch* gestures may be connected first through the choice and then with the parallel operator. The choice operator connects also the *Point* and the *Diverge or Converge Hands* in the full-body version.

$$\begin{aligned} &\text{Multitouch with choice operator} && (4.1) \\ &(Start_1 \gg Move_1^* [> End_1] []) \\ &(Start_1 |=|Start_2) \gg ((Move_1^* || Move_2^*) [> (End_1 |=| End_2)]) \end{aligned}$$

$$\begin{aligned} &\text{Multitouch with parallel operator} \\ &(Start_1 \gg Move_1^* [> End_1] || \\ &(Start_1 |=|Start_2) \gg ((Move_1^* || Move_2^*) [> (End_1 |=| End_2)]) \end{aligned}$$

Full-body

$$\begin{aligned} & (mH_r^*[ts]) [] \\ & (mH_r[d] |= |mH_l[d]) \gg ((mH_r^*[d] || mH_l^*[d]) \\ & \quad [> (mH_r[\vec{d}] |= |mH_l[\vec{d}])) \end{aligned}$$

The UI behaviour associated to the gesture definition can be summarized as follows:

- To the *Move* block of the pan gesture and to the mH_r block of the pointing gesture, we associated an event handler that draws a line from the previous touch position to the current one.
- To each one of the *Move* blocks of the pinch gesture and to the mH_r blocks of the diverge or converge gestures, we associated an event handler that computes the difference between the previous and the current distance between the two touches. If it is increased, the canvas zooms in the view, otherwise it zooms out the view accordingly.

The iPhone version included the definition of the *TmpExp* that describes the gesture definition as discussed in section 5.1.2 and 5.2.

The UI controls and the listeners that define the UI behaviour for the multitouch version are coded exploiting the UI Kit framework [6] for iOS. The resulting user interface is shown in Figure 5.5.

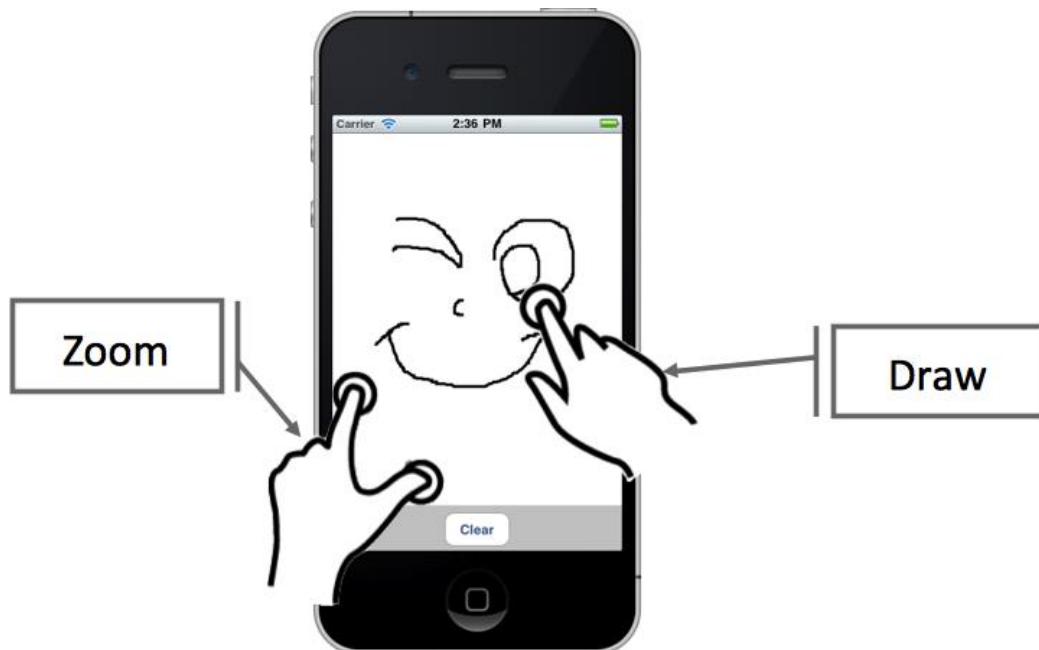


Figure 5.5 Simple canvas UI, multitouch version

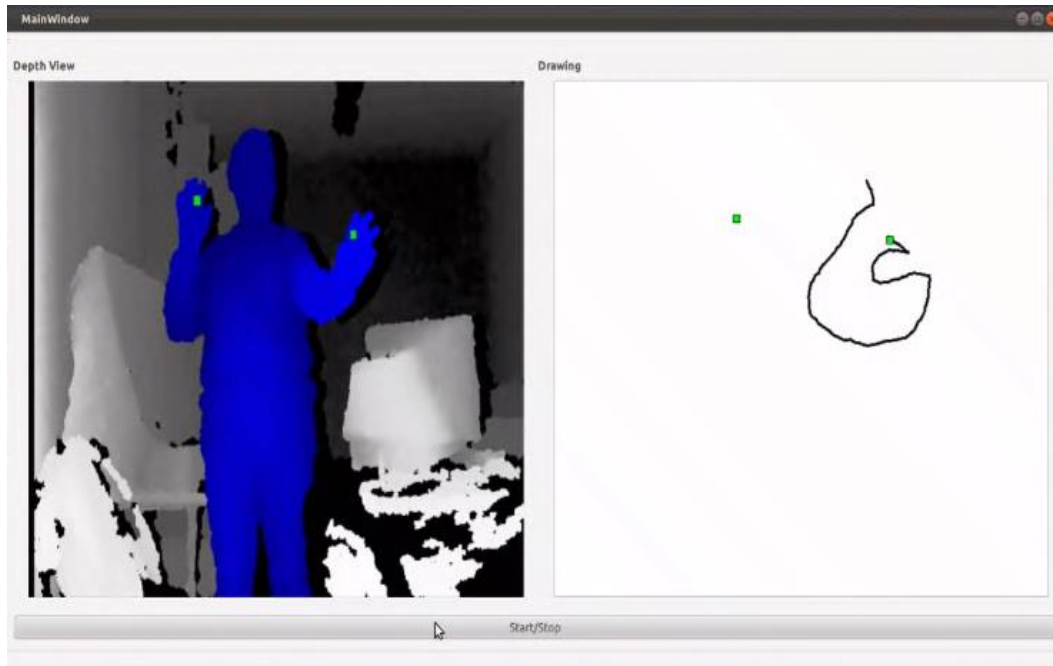


Figure 5.6 Simple canvas UI, full-body version

For the Kinect version, the definition of the UI reaction with respect to the notification of the gesture recognition is symmetric with respect to the iPhone version. However, this time the application exploits the Qt4 for the application UI, which is shown in Figure 5.6.

5.4.2 Photo viewer

The second application we discuss is a multitouch photo viewer for iOS devices, which is the sample application that is shipped with the GestIT library in order to show how it is possible to create multitouch interfaces with it.

With all the currently available UI toolkits for multitouch mobile devices, the creation of an application that is able to simply show a photo is simply a matter of exploiting an image view widget. Usually, such widget provides the possibility to interact with the contained photo, using the pinch for zooming and a single touch for panning the view. When the device recognizes one of these gestures, it raises an event corresponding respectively to the change of the image scale factor or position.

It is possible to identify two problems with this approach from the gesture interaction design point of view:

1. The gestures that have been selected for the interaction cannot be modified. They are completely tied to the implementation of the UI graphic control
2. If the events are raised only when the corresponding gesture has been completely recognized, it is difficult for the developer to provide intermediate feedback during the gesture execution.

Therefore, in order to show the GestIT library capabilities, we started from this simple photo viewer application, but with a small variant: when the user is panning or zooming the photo, the application has to show one or more arrows under the user's fingers, which change their orientation according to the finger movement direction.

Exploiting directly the image viewer widget is still possible but, in order to provide the intermediate feedback, the developer has to register to the following low-level touch events:

1. Detect when a new touch is detected, in order to show the arrow(s)
2. Maintain the count of the currently detected fingers
3. Track the movement of the different touches for detecting the movement direction
4. Detect when a touch ends, in order to hide the arrow(s)

The application we discuss in this section shows a different way to create an application for this simple yet exhaustive scenario.

Through the GestIT library, we separated the UI control for visualizing the photo from the definition of the gestures that manipulates it. We exploited the existing image viewer shipped with the iOS UI toolkit, but we “deviated” the touch events to a GestIT expression.

Such expression is a composition of the pan and the pinch gestures through a choice operator, respectively discussed in section 4.1.3 and 4.1.5.

The photo scale factor and viewport position are now changed through two different handlers attached to the gesture expression ground terms. Such expression is exposed by the image control. In this way, the developer can inspect such definition and it is possible to easily add behaviour to an existing definition.

In our case, it is possible to add the arrow feedback through three simple handler methods, one for showing or hiding one arrow, one for changing its position and orientation.

Without re-implementing the touch tracking logic, it is possible to connect such handlers to the recognition of the appropriate ground term (e.g. touch start for showing the arrow, touch move for changing the position and the

orientation etc.) and the developer can really reuse the UI control *and* the definition of the gestures.

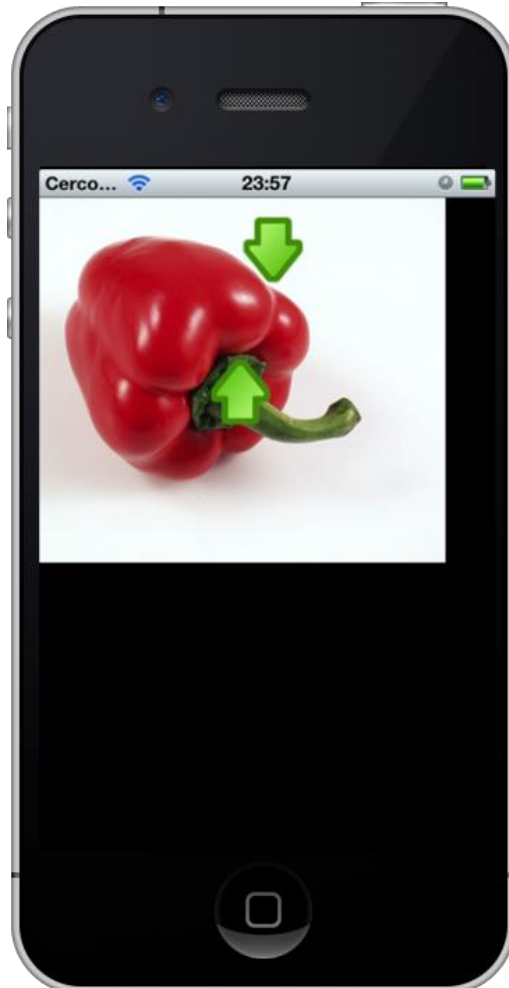


Figure 5.7 The photo viewer application

5.4.3 3D viewer

In this section, we describe a 3D viewer we created for demonstrating the library capabilities in [125]. The application visualizes a 3D car model, which can be moved and rotated by the user through a set of on-air gestures.

In order to avoid unwanted interactions, we specified that users have to stand with the shoulders in a plane (almost) parallel to the sensor, before starting the interaction with the car. Thus, if the user is not in front of the device that means most of the times in front of the screen, the interface will not give any response.

The car position can be changed with a grab gesture (see section 4.2.2), which consists of closing the right hand, moving and reopening it.

In addition, the car can be rotated performing the roll gesture (an on-air grab with both hands), which means closing two hands, moving them maintaining almost the same distance in between, and then reopening them (see section 4.2.13)

We want also to display the 2D projected hand position on the screen, in order to provide an immediate feedback to the user for each hand movement.

The resulting gesture model is defined in equation 4.2. The *Front* and *NotFront* gestures respectively activate and deactivate the UI interaction. When a change in the feature associated to the left and right shoulder (indicated as S_l and S_r) occurs, they respectively check if the sensor parallel plane property (p) is true or false.

The UI interaction consists of three gestures in parallel. The first and the second one are simply a hand position change. The UI reacts to their completion moving a correspondent (left or right) hand icon. The *Grab* gesture is the one associated to the car position change, and consists of a sequence of a right hand close (represented $oHr[c]$) and a unbounded number of right hand moves (mH_r^*), interrupted by the opening of the right hand ($oH_r[o]$).

The *Roll* gesture is represented by the same sequence, performed with both hands in parallel, almost maintaining the same distance (the d condition).

$$\begin{aligned}
 &Front \gg (mH_r^* | | mH_l^* | | (Grab [] Roll)))^* [> NotFront & (4.2) \\
 &Front = (S_l[p] | | S_r [p]) \\
 &NotFront = (S_l[!p] | | S_r [!p]) \\
 &Grab = oH_r[c] \gg (mH_r^* [> oH_r[o]) \\
 &Roll = (oH_r[c] | | oH_l[c]) \gg \\
 &\quad ((mH_r[d] | | mH_l[d])^* [> (oH_r[o] | | oH_l[o]))
 \end{aligned}$$

The intermediate feedback associated to different sub-parts of the composed gestures is shown in Figure 5.8, the upper part shows the UI feedback provided while performing the gestures represented in the lower part.

The interaction proceeds as follows: when the correct pose is detected (the *Front* gesture is completed), the car passes from a grayscale to a full-colour visualization, indicating that it is possible to start the interaction (the B square in Figure 5.8).

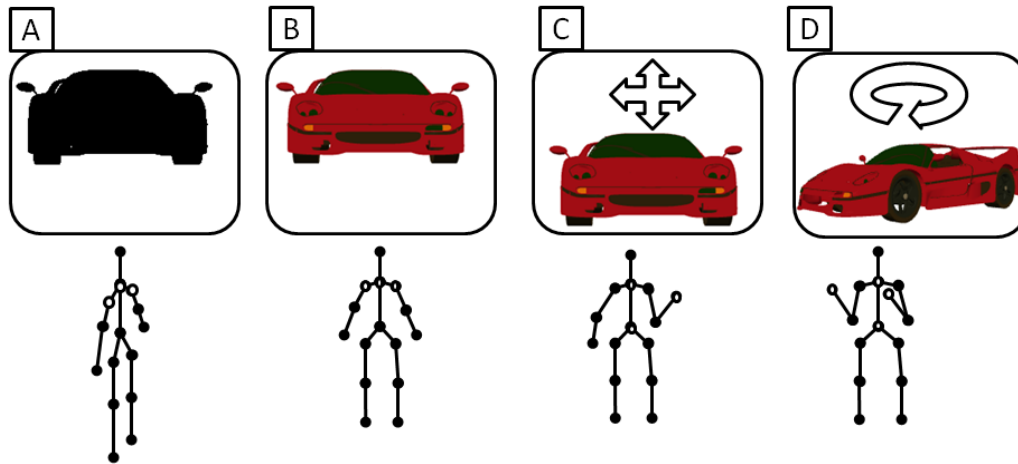


Figure 5.8 3D viewer interaction

When the user “grabs” the car with one hand (completes cH_r), a four arrow icon is shown on top of the car (C square). The change of the car position is associated to the following hand movements (mH_r^*).

The interface during the grab gesture is shown in Figure 5.9: the central part shows a car model with the user feedback for the grab gesture on top. The sidebar shows the representation of the user’s skeleton, the video coming from the RGB camera of the Kinect sensor and a label with the current tracking state (true or false) of the application.

The other interaction command is associated with a two-hands closure in parallel (completion of $(oH_r[c]||oH_l[c])$, the roll gesture of section 4.2.13), a circular arrow is displayed (D square), suggesting the gesture function. The car rotation is associated to the parallel movement of the two hands (the completion of $(mH_r[d]||mH_l[d])^*$). The car returns inactive when the user is not in the front position any more (A square).

Figure 5.10 shows the interface during the rotation of the 3D model: the central part shows the rotation feedback, while the right part still shows the position of the skeleton, the RGB video and the tracking label.

Writing such application with the support of the GestIT library has a set of advantages, which is possible to notice also in this simple case.

First of all, the defined gestures are separated from the UI control. Indeed, the car viewer is a standard WPF 3D viewport, enhanced with full body gestures at the application window level.

Second, the possibility to inspect the gesture definition and to attach handlers at the desired level of granularity allowed us to define easily when and how to react to the user input, without mixing the logic of the reactions with the conditions that need to be satisfied for executing them.

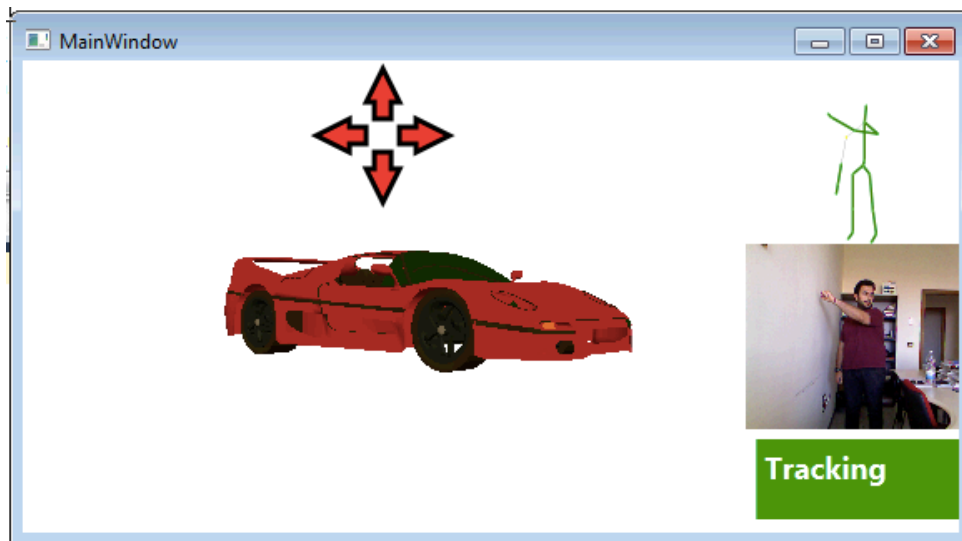


Figure 5.9 3D viewer UI, grab gesture

Finally, we do not define any additional UI state for maintaining the gesture execution. Indeed, if we created such application simply with the Kinect for Windows SDK, we would have needed at least a state variable for maintaining what the user has already done and, consequently, for deciding what she is allowed to do next (e.g. when the user closes the right hand the state has to change for moving the car at next hand movement).

Most of the times, this is managed with the implementation of a state machine inside the handler of the skeleton tracking update, which mixes the management of all gestures together. Especially when we want to support parallel gestures, mixing the different gestures leads to code difficult to understand and maintain.

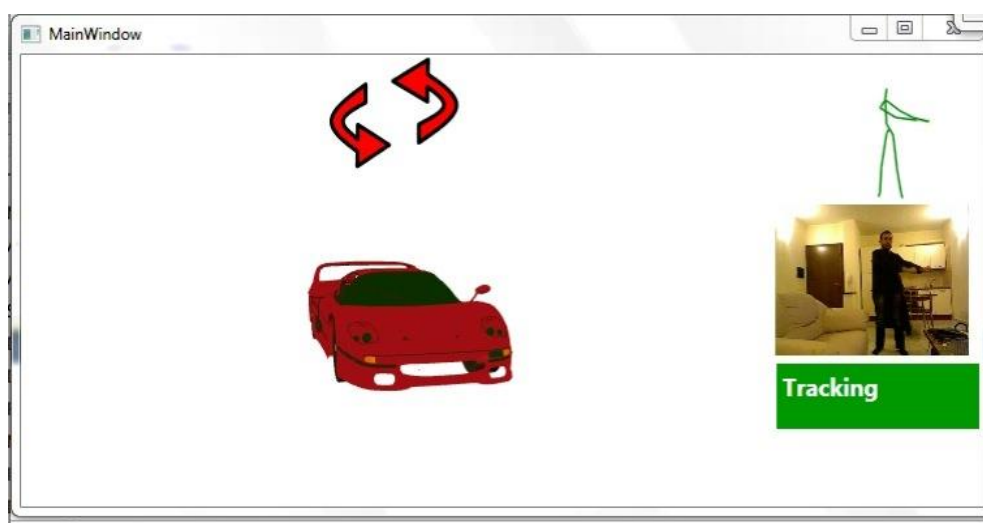


Figure 5.10 3D viewer UI, roll gesture

The approach discussed in this thesis helps the developer to separate the temporal aspect and the UI reaction and to reuse gesture definition in different applications, while maintaining the possibility to define fine-grained feedback.

5.4.4 Touchless recipe browser

In this section, we describe how we exploited the GestIT library in order to create an interactive support to be used in a kitchen environment, which has been presented in [127].

Indeed, depth sensors are useful when users are performing tasks that do not allow the use of traditional pointing devices or keyboards. For instance, the primary user's task may be the creation of an artefact in the real world, which requires several steps to be completed, like assembling furniture or replacing a part of an appliance.

An interactive support that enables the user to browse the information while performing the primary task can be really effective in such situations.

Its advantages and its risks have been analyzed in [77], where the authors concluded that a touchless direct manipulation is well accepted by the users, but designers should be careful while choosing the vocabulary, which has to be immediately understandable for them.

It is possible to find in literature examples of touchless interfaces for specific appliances in the kitchen environment [48], or for getting a full control of different devices [108], but problems such as gesture reuse or how to distinguish movement aimed to interact with the system from those that are not (the well-known Midas Touch) are still open.

In this section, we consider the kitchen environment as an example for such kind of applications. The case-study scenario envisions the assistance during the dish preparation through information displayed on a screen, which can be browsed while touching the food or using kitchen tools. In such situation, the touchless interaction has the advantage of avoiding the contact with the input devices, which can create hygiene problems or the risk of damaging the electronic equipment (e.g. touching it with wet hands).

For the development of the touchless user interface, we considered a scenario in which the user wants to cook a dish, but she does not really master the particular procedure. Therefore, she needs a description of the steps to be accomplished in order to complete the preparation, which is usually provided through books or specialised magazines.

We try to enhance such experience with an interactive support for delivering the information: the steps are described by the interactive system through a combination of text and video. In order to browse the recipes, the user does not need to touch any particular input device, which has the advantage of supporting the interaction while the cooker is manipulating tools or she has dirty hands.

Instead, she controls the application through a multimodal combination of voice and gestures. In order to enable such kind of interaction, we exploited a Microsoft Kinect, together with a computer screen or TV that displays the user interface.

The touchless recipe browser supports two tasks: the first one is the recipe selection, while the second one is the presentation of the cooking step. The selection of the recipe consists of two screens: the first one for selecting the recipe category (starter, first course, second course, dessert etc.) and then the selection of the recipe itself.

The presentation of the cooking steps is performed through a combination of text and video. The user can watch the entire video with subtitles that show how to cook the selected dish, or she can browse back and forth among the different steps with a previous and next function or controlling a timeline.

In order to combine the vocal and the gestural modality, we extended the GestIT library adding the possibility to react to vocal input, representing the different keywords that activate vocal commands as features that can be detected by the Kinect support. Therefore, it is possible to combine in the gesture description expression also vocal inputs.

With respect to the design of the user interface, we decided to assign commands that do not need any argument (e.g. going back to the previous screen) to the vocal modality, while we assigned commands related to object selection and/or manipulation to the gesture modality. The rationale behind this choice is trying to keep the user's focus on her main task (cooking the dish) as much as possible: gestures have a higher cognitive load with respect to speech interaction.

In addition, the design of such kind of user interface must take into account the well-known Midas Touch problem. We exploited the possibility to define the temporal relationships between gestures provided by GestIT in order to mitigate it. Indeed, we chose to enable the interaction with the user interface only if the user stands in front of the screen, while we do not consider any movement or interaction otherwise. The rationale behind this

design choice is that, being the dish cooking the main task, we assume that most of the times the user does not want to interact with the application. When the user wants to get some information from the application, she will look at the screen, positioning herself in front of it.

Using the GestIT library, the interaction with the different application presentation follows the schema defined in equation 4.3. The *Front* gesture enables the *ScreenInteraction*, which represents the allowed gestures or vocal commands for the considered presentation, and it is disabled by the *NotFront* gesture. Such expression term is refined in different ways according to the considered presentation.

$$\begin{aligned}
 \textit{Front} &\gg \textit{ScreenInteraction}^* [> \textit{NotFront} & (4.3) \\
 \textit{Front} &= (S_l[p] \mid | S_r[p]) \\
 \textit{NotFront} &= (S_l[!p] \mid | S_r[!p])
 \end{aligned}$$

As it is possible to observe in equation 4.3, *Front* and *NotFront* are symmetric: they respectively check whether the shoulder position (S_l and S_r) are parallel with respect to the sensor (and screen) plane (the p predicate) or not. This means that as long as the user stays in front of the screen, it is possible to interact with the application.

The *Front* and *NotFront* gestures have handlers that provides the user with feedback for signalling whether the application is ready to receive inputs (a green “Tracking” label) or not (a red “Not Tracking” label).

Figure 5.11 shows the presentation for selecting the recipe category. The user points the screen and moves the hand in order to highlight the different categories, which are magnified using a fisheye effect. The selected one has a thick red border. When the user closes her hand, the presentation changes, and the application shows the interface in Figure 5.12, which supports a similar interaction for selecting one of the recipes in the different categories.

As already discussed before, we assigned the commands without arguments to the vocal modality: in this screen it is possible to use the following commands: *back* for going back to the previous screen and *exit* for closing the application.

$$\begin{aligned}
 \textit{ScreenInteraction} &= V[\textit{back}][] V[\textit{exit}][] \textit{Grab} & (4.4) \\
 \textit{Grab} &= mH_r^* [> cH_r
 \end{aligned}$$



Figure 5.11 Recipe category selection



Figure 5.12 Recipe selection

Equation 4.4 shows the *ScreenInteraction* gesture definition for the selection presentation (we describe movements only for the right hand for simplicity, but the actual implementation provide a symmetric support also for the left hand). The features marked with $V[word]$ are those related to the voice and indicate the pronunciation of the specified word, with the obvious effect on the user interface (respectively going back to the previous screen or closing the application).



Figure 5.13 Recipe browser

The *Grab* gesture is used for selecting the recipe category and it is composed by an iterative hand movement (mH_r^*) disabled by a closure of the hand (cH_r). As already explained in section 5.1.1, it is possible to attach event handlers not only to the whole gesture completion (which performs the category selection and therefore changes the screen), but also to its sub-parts. In this case, the fisheye effect in Figure 5.11 is driven by an event handler attached to the completion of the hand movement (mH_r^*).

Figure 5.13 shows the screen for the preparation of a dish. In the upper part, it is possible to read the recipe name, in the centre there is a video tutorial for the preparation¹ together with a text describing the procedure

¹ The sample recipes included with the application prototype have been created using some videos from the public website of the Italian cooking TV show “I Menu di Benedetta” (<http://www.la7.it/imenu dibenedetta/>)

to follow in order to complete the current step. In the lower part, a slider represents the video timeline.

The interaction for this presentation is defined in equation 4.5. The vocal commands *back* and *exit* are still available in this screen. The video playback can be *continuous* or it can stop at each *step*. A vocal command is available for activating both modes.

It is possible to pronounce the words *next* and *previous* respectively to show the previous or the next step of the preparation. Such command can be activated also through the *Swipe* gesture, an iterative linear hand movement performed at a certain speed (verified by the properties *linear* and *speed*), disabled by a hand movement that does not have this characteristics (for finishing the iteration loop).

$$\begin{aligned}
 \text{ScreenInteraction} &= V[\text{back}][]V[\text{exit}][]V[\text{previous}][] & (4.5) \\
 &V[\text{next}][]V[\text{continuous}][]V[\text{step}] \\
 &\text{Swipe}[]\text{Drag} \\
 \text{Swipe} &= mH_r^*[\text{linear} \wedge \text{speed}][> mH \\
 \text{Drag} &= \text{Grab} \gg \text{Release} \quad \text{Release} = mH_r^* [> oH_r
 \end{aligned}$$

If the swipe movement has been performed from left to right, the tutorial proceeds to the next step, if it has been performed from right to left the tutorial goes back to the previous step.

Finally, it is possible to control the video timeline through the *Drag* gesture. The latter is a composition of two sub gestures: the first one is *Grab* (already defined for the recipe and dish selection) and the second one is *Release*, which is an iteration of hand movement disabled by its opening (oH_r).

Through such gesture description is possible to notice the reuse possibility offered by the library (we defined the *Grab* gesture and reused it for the *Drag* one). Different handlers have been assigned to the different gesture sub-parts, which allow to define easily the user interface reactions while performing the gesture: when the user closes the hand (completion of cH_r in the *Grab* gesture), the user interface changes the colour of the slider knob, after that its position is changed according to the hand movement direction together with the displayed video frame (completion of $.mH_r^*$ in the *Release* gesture) and finally, when the whole gesture is completed, the video playback restarts from the point selected by the user.

Chapter 6

A Gestural Concrete User Interface in MARIA

In this chapter we extend MARIA [111], a model-based user interface definition language with different abstraction levels, in order to define full-body gestural interfaces.

We first describe the general modelling concepts of the language, and then we detail how we extended the entities in order to model gestural interfaces. After that, we discuss the implementation of a model to code transformation that we exploit in order to create running applications starting from the MARIA model definition, showing a sample application.

6.1 MARIA

MARIA [111](Model-based Language foR Interactive Applications) is a set of XML languages for defining UIs at different levels of abstractions.

Created as an evolution of TERESA [96], the different languages inherit the CAMELEON [27] reference framework structure.

Indeed, the set includes an abstract language that has multiple extensions for the different interaction platform supported.

For designers of multi-device user interfaces, one advantage of using a multi-layer description for specifying UIs is that they do not have to learn all the details of the many possible implementation languages supported by the various devices, but they can reason in abstract terms without being tied to a particular UI modality or, even worse, implementation language. In this way, they can better focus on the semantics of the interaction, namely what the intended goal of the interaction is, regardless of the details and specificities of the particular environment considered.

The languages have also an associated authoring tool called MARIAE [112] (MARIA Environment), publicly available for download².

Exploiting the CTT [114] language for the task modelling, the tool is able to support the CAMELEON design process for different platforms, allowing the designer to create and edit models at different levels of abstraction, exploiting also different reification functions (see section 2.4.2).

The tool is able to derive an AUI from a CTT task model, to derive different CUIs from an AUI definition and provides at least one code generator (FUI) for each supported platform.

6.1.1 Abstract User Interface

The *Abstract User Interface* (AUI) level describes a UI only through the semantics of the interaction, without referring to a particular device capability, interaction modality or implementation technology.

An AUI is composed by various *Presentations*, which groups logically connected model elements to be presented to the user at once.

A presentation contains modelling elements that belong to two different categories: *Interactors* or *Interactor Compositions*. The former represents every type of interaction object, the latter groups together elements that have a logical relationship.

According to its interaction semantics, an interactor belongs to one the following categories:

- The *Selection* interactors allow the user to select one or more values among the elements in a predefined list. It contains the selected value and the information about the list cardinality. According to the number of values that can be selected by the user, the selection interactor can be either a *SingleChoice* or a *MultipleChoice*.
- The *Edit* interactors allow the user to manually edit the data associated to them, which can be textual (*Text Edit*), numerical (*Numerical Edit*), related to a position (*Position Edit*) or a generic object (*Object Edit*).
- The *Control* interactors allow the user to switch between presentations (*Navigator*) or to activate UI functionalities (*Activator*).
- The *Only Output* interactors represent information that is presented to the user but it is not affected by the user actions. An interactor of

² <http://giove.isti.cnr.it/tools/MARIAE/home>

this category can be a *Description*, which represents different types of media, an *Alarm* a *Feedback* or a generic *Object*.

The different types of interactor-compositions are:

- *Grouping* a generic group of *Interactor* or *InteractorComposition* elements.
- *Relation* a group where two or more elements are related to each other.
- *Composite Description* that represents a group aimed to present contents through a mixture of *Description* and *Navigator* elements.
- *Repeater* which is used to repeat the content according to data retrieved from a generic data source

MARIA allows describing not only the presentation aspects but also the associated behaviour. In addition, the interface definition contains also the description of the data types that are manipulated by the user interface. The interactors can be bound with elements of the data model, which means that, at runtime, modifying the state of an interactor changes also the value of the bound data element and vice-versa. This mechanism allows the modelling of correlation between UI elements, conditional layout, conditional connections between presentations, input values format. The data model is defined using the standard XML Schema Definition constructs.

MARIA has a set of features that allow the creation of multidevice applications, in particular based on web services [110,113] or able to adapt to the context of use [19].

- *Generic Back End*. The interface definition contains a set of *External Functions* declarations, which represent functionalities exploited by the UI but implemented by a generic application back-end support (e.g. web services, code libraries, databases etc.). One declaration contains the signature of the external function that specifies its name and its input/output parameters.
- *Event Model*. Each interactor definition has a number of associated events that allow the specification of UI reaction triggered by the user interaction. Two different classes of events have been identified: the Property Change Events that specify the value change of a property in the UI or in the data model (with an optional precondition), and the Activation Events that can be raised by activators and are intended to specify the execution of some application functionalities (e.g. invoking an external function).

- *Dialog Model*. The dialog model contains constructs for specifying the dynamic behaviour of a presentation, specifying which events can be triggered at a given time. The dialog expressions are connected using CTT operators in order to define their temporal relationships.
- *Continuous update of fields*. It is possible to specify that a given field should be periodically updated invoking an external function.
- *Dynamic Set of User Interface Elements*. The language contains constructs for specifying partial presentation updates (dynamically changing the content of entire groupings) and the possibility to specify a conditional navigation between presentations.

This set of features allow having already at the abstract level a model of the user interface that is not tied to layout details, but it is complete enough for reasoning on how UI supports both the user interaction and the application back end.

6.1.2 Concrete User Interface

A *Concrete User Interface* (CUI) in MARIA provides platform-dependent but implementation language independent details of a UI. A platform is, as stated in [27], a set of software and hardware interaction resources that characterize a given set of devices. MARIA currently supports the following platforms:

- *Desktop CUI*: models graphical interfaces for desktop computers.
- *Mobile CUI*: models graphical interfaces for mobile devices.
- *Multimodal Desktop CUI* models interfaces that combine the graphical and vocal modalities for desktop computers.
- *Multimodal Mobile CUI* models interfaces that combine the graphical and vocal modalities for mobile devices.
- *Vocal CUI* models interfaces with vocal message rendering and speech recognition.

Each platform meta-model is a refinement of the AUI, which specifies how a given abstract interactor can be represented in the current platform. For instance, if we consider a *Single Choice* interactor, it can be implemented with a radio button, a drop down list or a list box in the graphical modality, while on the vocal platform it can be rendered with a list of vocal messages for each option associated to a given keyword.

The same applies for the interactor compositions: a grouping can be implemented in a desktop platform using background colours, borders etc.,

while in a vocal platform it is possible to e.g. use sounds before the first group element.

The model definition can be exploited for creating (or deriving with a code generator) final implementations in different target languages. Indeed, it is possible to exploit the same mobile CUI for representing an interface for e.g. iOS or Android devices.

6.2 Gestural Concrete User Interface

As it should be clear from the CAMELEON [27] reference framework discussion, we extended MARIA with the definition of gestural interfaces simply providing a refinement of the AUI language that covers the modelling concepts needed by gestural interfaces.

In order to provide MARIA with these concepts, we have to create a set of modelling entities for the following parts:

1. A description of the data provided by the device
2. The description of the gestures and the temporal relationships between them
3. The description of the effects that the gestures have on the other parts of the interface
4. The description of the interface layout

We recall that the interaction semantics (which kind of task is supported by different interactors) is inherited from the AUI level.

The first point is needed in order to define the constraints and the effects of the gestures according to the data received by the recognition device. The description of such data needs to be abstract with respect to the actual programming language or development toolkit.

The second point is covered by the gesture meta-model discussed in Chapter 3. We detail in section 6.2.2 the entities we included in MARIA in order to define gestures.

The third point deals with two different aspects of the UI model. The first one is how it is possible to model the visual feedback that the user has to receive during the gesture performance. The second aspect is the need to provide the “glue” between the definition of the UI behaviour at the abstract level and the recognition of the gestures at the concrete one (section 6.2.3).

The last point is related to the visual part of the gestural UI. In brief, since MARIA already have a Concrete Desktop User Interface definition, we describe what was missing in the existing definition of the graphic controls

in order to be easily exploited also in the gestural model. It is worth pointing out that the solution to modelling problems that we discuss in section 6.2.4 are general and they can be applied not only to MARIA, but also to all the graphical control toolkits that are actually used in order to create the GUIs that have a gestural support.

6.2.1 Modelling device data

The body data can be modelled with a structure that contains the collection of the joint positions (a 3D point) and the joint orientation (a 3D vector) as defined in section 3.3. One instance of such data structure is available for each tracked user.

In addition, it is available for modelling the gestures also the history of the body data at the previous steps during the recognition. Therefore, for each user, the first instance provided by the runtime support is the current body data, while the following ones are related to the previous recognition step, providing access to what we called the gesture recognition support state sequence in section 3.1.1.

This structure is referenced in both event handlers and the modelling of the recognition constraints that are detailed in the following sections. The implementation of the runtime support for has to provide the access to this data in order to execute the model.

6.2.2 Gestures definition

In a gestural interface, the description of the gestures provides the temporal sequence for the exchange of information between the user and the application. Such sequence defines on the one hand how the application reacts to the user inputs while, on the other hand, provides the description of the set of actions that are available for the user in order to interact with the application.

In MARIA, as already described in section 6.1.1, the *Dialog Model* contains constructs for specifying the dynamic behaviour of a presentation, specifying what events can be triggered at a given time. The dialog expressions are connected using CTT [114] operators in order to define their temporal relationships.

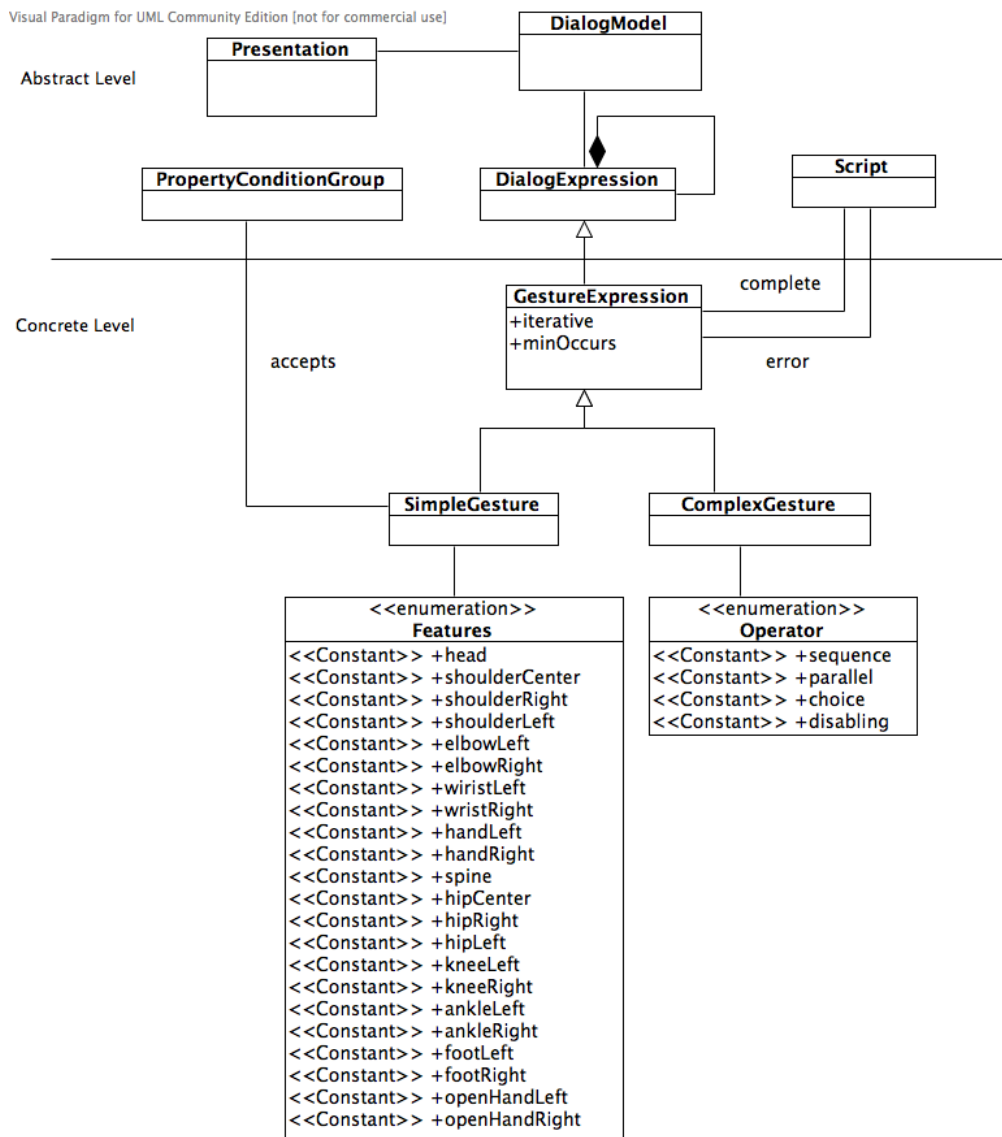


Figure 6.1 MARIA gesture description meta-model

A gesture description is a concrete example of such dialog expressions. In its simplest form, the dialog expression is related to the recognition of a ground term (see Chapter 3), which is an event triggered by the recognition device. The complex gestures, can be defined simply connecting ground terms and/or other complex gestures through the set of composition operators.

Therefore, we identified the *Dialog Model* as the point to extend in the gestural concrete user interface in order to define the gesture description. In order to do this, we created a refinement of the dialog expression that represents a generic expression in GestIT: the *GestureExpression*.

Figure 6.1 show the UML class diagram for the *DialogModel* extension, which introduces the gesture model in MARIA.

Using the same modelling approach we exploited in Chapter 5, the *GestureExpression* has two attributes. The first one models the only unary operator in the set: the *iterative* operator. The second one models the minimum number of times that one gesture has to be recognized, the *minOccurs* attribute.

Exploiting the combination of both attributes it is possible to specify the short-hands defined in section 3.1.2.7, in order to model gestures that can be repeated iteratively, but starting from a minimum number of times.

The abstract *GestureExpression* class is in turn refined into two different classes: the *SimpleGesture* which represents the ground term expressions and the *ComplexGesture*, which represent a composed gesture.

The *SimpleGesture* class has an associated *feature* attribute, which defines the feature change that is recognized by the simple gesture. Such attribute has an enumerated value for each feature described in section 3.3.

In addition, the *SimpleGesture* instances may specify some constraints on the recognition of the ground term they represent. In MARIA, it is possible to specify such constraints directly modelling them with instances of the *PropertyConditionGroup* class, which represents a boolean expression. The literals of the expression are represented by:

1. The value of an interactor attribute
2. The value of a data model element
3. The result of the execution of an *ExternalFunction*, which represents a functionality that is external to the definition of the UI model.

In particular, exploiting the external functions in order to model the gesture predicates allows the designer to reuse the predicate definition across various UI models. For instance, if the FUI exploits the GestIT library, it is possible to define as external functions each one of the predicates involved in the modeling of the common full-body gestures described in section 4.2.

The *ComplexGesture* class has an *operator* attribute, which specifies the temporal operator used for combining the set of sub-gestures. This attribute is specified through an enumeration that has one value for each one of the operator discussed in section 3.1.2.

From the XML syntax point of view, we created an element for each one of the features that can be recognized with a ground term. Therefore, the instances of the *SimpleGesture* class are serialized in XML using a different

tag according to the *feature* attribute. This leads to a more readable XML code.

For the same reason, we introduced an element for each one of the composition operators. Therefore, the *ComplexGesture* instances are serialized with different tags according to the *operator* attribute.

The relation between a composite gesture and its sub-components is specified with the XML element hierarchy in the document tree.

6.2.3 Gesture effects

According to the discussion in Chapter 3, our modelling approach allows the designer to attach the UI behaviour to both the successful recognition of a gesture component and also in case of a recognition error.

In the same way, we need to include such possibility also in the MARIA meta-model. In MARIA, the dynamic changes to the UI and to the data model state are defined through the *Script* class, an element of the AUI meta-model. It contains elements that represent expressions and statements that define such changes at both the abstract and the concrete level.

In order to distinguish the behaviour for the successful recognition from the error handling, we connected the *GestureExpression* class with two instances of the *Script* class: the first one represent the reaction to the *complete* event raised by a generic gesture expression, while the second one defines the reaction to the *error* event.

It is worth pointing out that in MARIA the behaviour which is independent from the concrete platform is already defined in the AUI model. The concrete model inherits the definition of such behaviour. The completion of a given gesture must be able not only to trigger the execution of some concrete-platform dependent behaviour, but it should be also able to activate the behaviour that is defined at the abstract level.

One simple example of the situation is represented by a presentation with two different *activators*, each one associated to a different application functionality, for instance save and new file. The triggering of the functionalities is associated in the AUI to the abstract event *activation*. When the AUI is refined to the concrete level, the activators can be in turn refined into two buttons. The abstract event (and its handlers) are inherited also in the CUI.

In a classical desktop interface, the buttons are activated using the mouse pointer, which is a singleton for the entire window system. In addition, for

activating them, the user (and obviously the designers) has the only option of clicking one of the mouse buttons.

In the case of the gestural interaction, the designer may use different paradigms for both the interactor selection and activation. The gestural interaction provides a richer vocabulary for selecting and activating a graphical control in general, and the buttons in particular. For instance, it is possible that the user activates the first button raising the left hand, while she activates the second one raising the right hand. Another possible interaction is that the user points with the hand one of the two buttons and closes the hand for activating it.

From the previous description, it should be clear that the binding between the gestures and the abstract events cannot be derived implicitly as in the classical desktop interfaces, but it has to be defined explicitly.

The way we identified for connecting the recognition of a gesture expression with the behaviour defined at the abstract level is to explicitly raise the abstract events inside the definition of the behaviour associated to a gesture expression. Indeed the MARIA meta-model contains, among the other statements for the definition of the UI behaviour, the *Raise* element. This modelling construct allows raising a specific event (either abstract or concrete) specifying the event name, the interactor identifier and the event arguments (if needed).

Therefore, the schema for binding the definition of the behaviour associated to the gestures to the abstract one consists of first managing the changes that involve the concrete level (most of the times providing the intermediate feedback) and then raising the abstract event that the designer wants to trigger.

If we consider the hand-pointing interaction in our example, when the user changes the hand position, the interface should give some feedback for identifying which button she is currently pointing (e.g. drawing the button border in a different colour). When she closes the hand, the behaviour associated to the gesture completion has first to identify which is the button currently pointed (and this part is related to the concrete level) and then it is possible to raise the activation event of the selected button, executing the behaviour defined at the abstract level. We provide a real modelling example for such binding in section 6.4.

6.2.4 Interactors

The definition of the graphical part of the gestural CUI is based on the interactors that are already defined for the graphical desktop CUI. In order to do this, the meta-model of the gestural CUI imports the classes that refine the abstract interactors as described in section 6.1.2.

As we already explained in the previous section, in a gestural interface the binding between how the user selects the different interactors (e.g. pointing them with the mouse) is not implicit anymore, and the designer may select different ways for let the user start the interaction with a concrete UI object.

From the modelling language point of view, the events defined by both the concrete desktop interactors and the ones raised when the gestures are completed in the dialog model are already sufficient for defining different selection techniques. However, the resulting models are complex to define and consequently to read and to understand. Indeed, the following definitions are necessary:

- In the completion event of the interactor selection gesture, the designer has to define how to calculate which interactor has been selected by the user, according to a given selection logic. For instance, it is possible to directly point one interactor (and therefore specify a pick-correlation algorithm). Another example is a list of interactors where the user can change the currently selected one in a sequential manner. The previous one in the list may be selected with a swipe gesture from right to left, while the following one may be selected with a swipe from left to right.
- After that, the designer should define how to provide feedback to the user for recognizing which interactor is currently selected, tracking the currently selected interactor.
- Once an interactor has been selected, the UI has to execute a conditional handler that behaves differently according to the selection, in order to define different reactions.

This problem obviously recurses in different interfaces. Therefore, we extended the definition of the *Interactor Composition* refinement in order to ease the definition of such recurring interaction scheme.

We exposed a property called *focusPoint*, which can be used in order to specify a specific point that currently focuses the user's attention. When such point is changed, the runtime support automatically calculates which interactor is the currently selected one. With this protocol, the designer is

no more in charge of defining the pick-correlation between the point and the interactors. However, she can still define different ways for selecting the interactors in a composite UI, modelling the selection of the actual point with different strategies.

In addition, each refinement of the *Interactor Composition* category contains a new element, which defines the style for showing which one is the selected interactor. The element was added in the *GroupingSettings* class, which contains the styles for rendering the grouping (and the other classes of the *Interactor Composition* category). Such element contains attributes for defining e.g. the border for the selected element, a different background colour etc.

In addition, we added a property to the presentation class in order to maintain which interactor currently has the interface focus. The property has to be automatically updated by the runtime support, according to the focus point selected. This eases the definition of the interaction making such information always accessible without specifying the logic for the property update.

6.3 Model to code transformation

Having defined the various components of the gestural CUI modelling elements, we created a model to code transformation, which shows how it is possible to exploit the modelling language for creating the FUI.

Differently from the other generators provided with the MARIAE tool, which transform the models into running web applications, we defined one of the first transformations that creates a standalone application.

The target implementation exploits the following technologies:

- Windows Presentation Foundation as presentation layer [89]
- C# for defining the application behaviour
- The GestIT library for defining full-body gestures
- The Kinect SDK [87] for managing the data coming from the Kinect sensor device.

The transformation process consists of two steps. The first one transforms the MARIA model, defined through the usual XML syntax into a XAML [89] definition of the presentation layer.

The second step takes as input the same MARIA model and creates a C# file that contains the definition of the application behaviour.

Both files represent a partial definition of the application window, but their combination defines the application completely (exploiting the partial class definition mechanism [85]). The communication between the two parts relies on a naming convention for the class methods, which is shared between the two transformations (e.g. the event handler for a button specified in the presentation layer is then implemented with the same name in the behaviour file). Both transformations are defined using an XSLT [136] stylesheet.

The whole interface is mapped into a single window, while the different presentations are mapped into a separate panel in the window content.

For each one of the interactors and interactor compositions contained into the different presentation, the transformation selects the corresponding widget in the WPF framework. In particular:

- For each interactor that specifies the *id* attribute, the transformation fills the *Name* attribute in the corresponding WPF widget. In this way, the behaviour part (a C# file) can access the interactor and its properties simply considering it as an instance variable.
- The interactor composition refinements are mapped into different WPF panels, according to their specification. For instance, if a grouping is implemented using the *Grid* technique, it is mapped into a grid panel in WPF, if a grouping is implemented with the tab technique, it is mapped into a *TabPanel* in WPF etc. Otherwise, the transformation uses a vertical *StackPanel*, which positions the inner elements vertically, according to the their definition order.
- The interactors are mapped into the correspondent widgets in WPF (buttons, images, videos etc.).
- The connections between the different presentations are mapped a change of the currently visualized content inside the main window.
- The event handlers, which are defined in the C# part of the UI, are attached to the WPF widgets simply specifying the method name in the XAML code.

The gesture model is mapped into the corresponding XAML elements provided by the GestIT library, with a straightforward transformation. The recognition constraints and the handlers for the successful or erroneous performance of the gesture are attached to the expression definition specifying the name of the corresponding method in the C# class.

The second transformation defines the UI behaviour in C#. The first part defines the overall structure of the class (importing the external libraries, defining the class name and the constructors). After that, the class defines a set of methods for changing the gesture model together with the presentation: we recall that the gesture model is associated to a specific presentation. Such methods are triggered in correspondence of the activation of an interactor specified in a *Connection*.

The third part of the class contains the gesture recognition constraints, associated to the *accept* property of the ground term expression. The C# code implements the definition expressed with the *PropertyCondition* and *InvokeFunction* constructs in the model (see section 6.2.2).

The fourth part symmetrically defines the UI reaction to the completion of the gesture expression (both simple and composed). In the same way are transformed also the reaction to the erroneous recognition of the different gestures (if defined).

The last part is dedicated to the implementation of the event-handlers associated to the different interactors. In this part are also considered the handling of the connections among the presentations.

In the following section, we present a sample application generated from a MARIA model definition.

6.4 Sample application

We show in this section a concrete modelling example in MARIA. We modelled a simple gestural interface for controlling a digital TV.

We first describe the tasks that need to be supported by this application and the implementation of the UI at the abstract level. After that, we discuss the concrete gestural refinement and we show the final result.

The application allows the user to watch a TV show. In addition, the user should be able to change the current TV channel and to retrieve information on the program scheduling.

The temporal sequencing of the tasks supported by the application is shown in Figure 6.2, using the CTT notation [114]: the application normally shows a TV program, represented by the *showChannel* task, until the user request the control of the device (the *requestControl* task).

The user can control the TV through two commands in choice: the first one allows changing the current selected channel and the second one for retrieving information on the program scheduling.

The first functionality is the channel selection (the select channel task), followed by the actual change of the channel performed by the application (the save selection task).

The second functionality allows the user to browse the information (select info), which is provided by the control application (the show info task). Finally, the user goes back and watches again a TV show (back to tv).

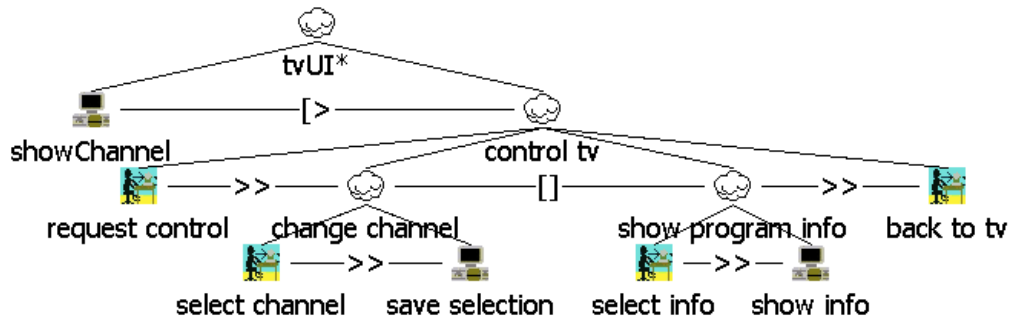


Figure 6.2 Task model for the TV control application

At the AUI level, the interface can be modelled using four different presentations, which are shown in Figure 6.3. The first one is dedicated to watching the TV show. Inside this presentation, a description interactor provides the information on the TV show (which will be obviously refined in a video). In addition, the presentation contains the navigator for changing the presentation to the second one, shown in Figure 6.3 (2).

The second presentation allows the user to select the two controls functions: changing the current channel or retrieving the schedule information.

The third presentation implements the first control functionality: the selection, which consists in changing the current TV channel. The user selects the current channel in a predefined list of values (Figure 6.3 part 3).

The fourth presentation provides the information about the TV show scheduling (Figure 6.3 part 4).

It is possible to go back to the first presentation after the completion of the TV control task, in order to continue watching the show.

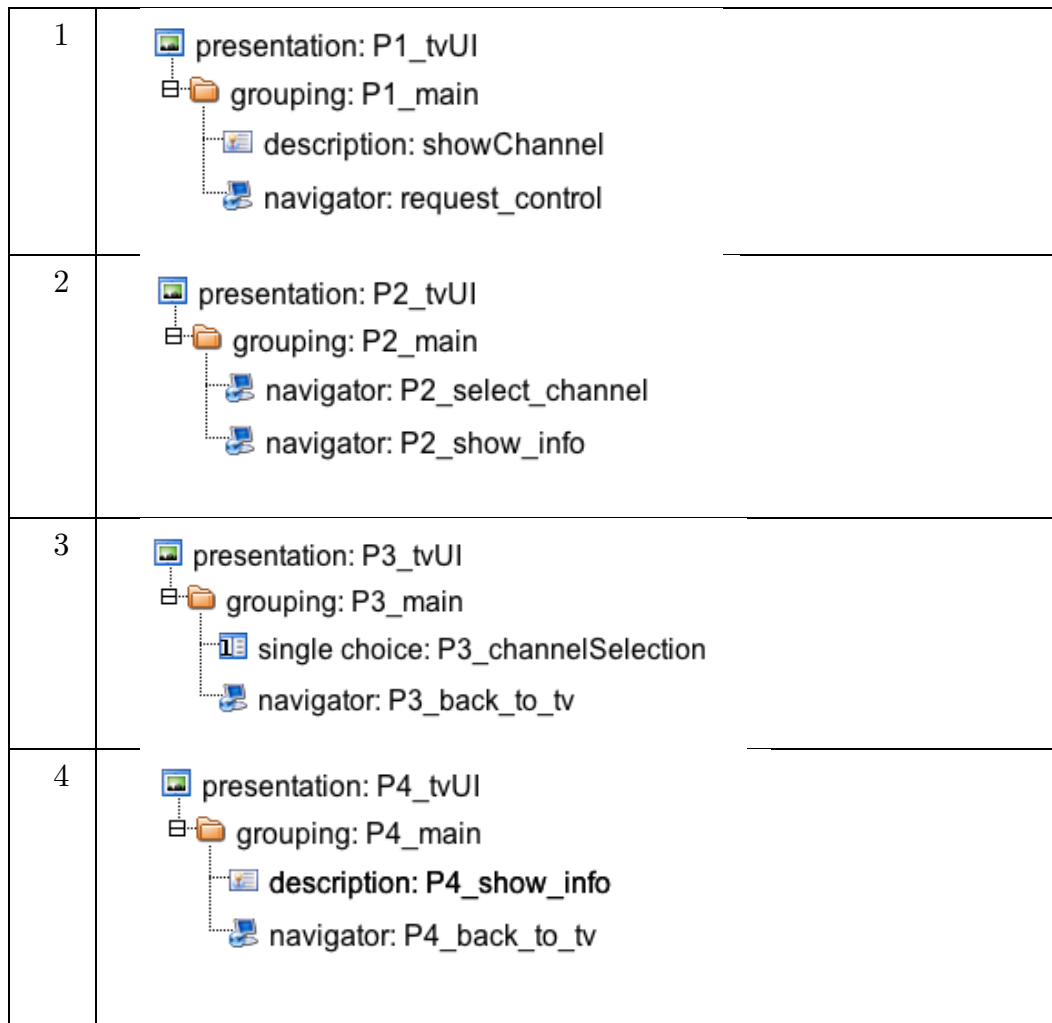


Figure 6.3 TV application AUI

For providing a gestural refinement for the proposed AUI, we have to select the concrete implementation for the different abstract interactors.

In the first presentation, the description is obviously refined into a video, which allows watching the selected TV show. In our design, the navigator is refined into a simple link that we do not want to be visible in the concrete UI. Therefore, we set its hidden attribute to true, leaving the full screen to the video. In order to activate the navigator, we chose to exploit the wave gesture (see section 4.2.7), which has the effect to change the current presentation, showing the second one. We selected the wave gesture because it is more difficult to have false positive in the recognition. Watching the TV show is the main task and it is important to avoid unwanted interruptions.

The second presentation contains two different links: the first one takes the user to the channel selection presentation, while the second one to the

visualization of the show schedule. We decided to position the two links horizontally, providing both an image and a textual label. The user can select one between the two links pointing at the screen. The application highlights the currently selected link showing a thick blue border around the link. The link can be activated simply closing the hand. In summary, we exploited in the dialog model of this presentation the grab gesture (see section 4.2.2).

As in other applications discussed in this thesis, we exploited the turn gesture (section 4.2.10) for mitigating the Midas touch problem: the user is allowed to perform the grab gesture only if she is in front of the screen. Otherwise, the application does not react to any gesture. Figure 6.4 shows the interface for the functionality selection.

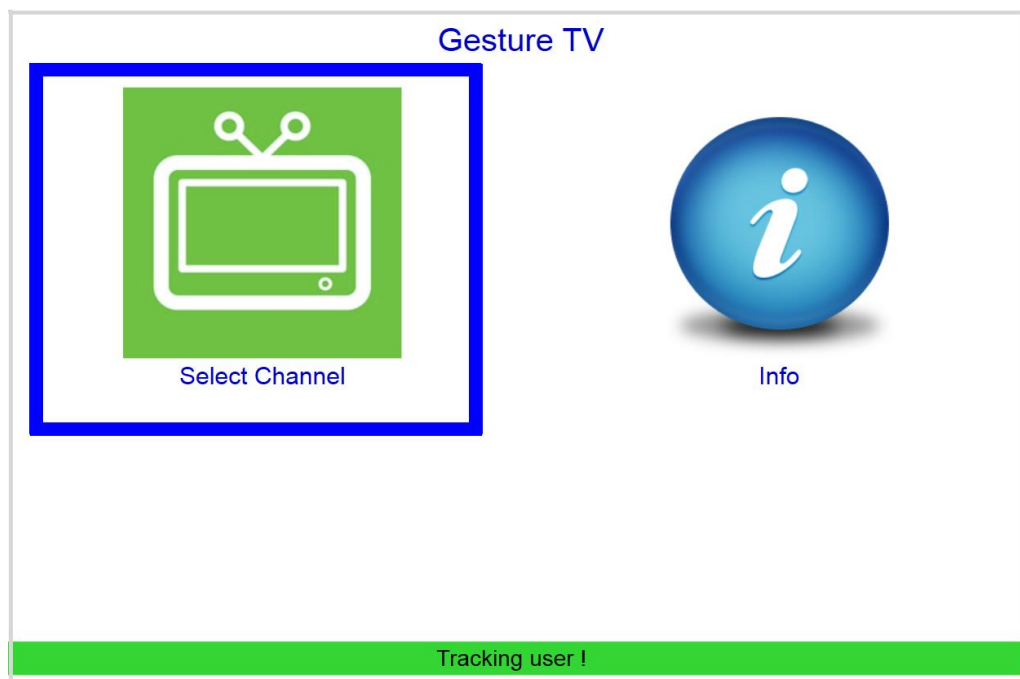


Figure 6.4 MARIA application: function selection presentation

If the user selects the show info functionality, the application shows the presentation in Figure 6.5. The TV programs schedules are grouped per day. The user can change the selected day with a swipe gesture (see section 4.2.8). A left-to-right swipe selects the next day while a right to left the previous one. Instead, if the user selects the channel selection functionality, the channel list is shown using a grid (see Figure 6.6). It is again possible to select among the different options (channels) pointing one of the elements in the grid and confirming the selection closing the hand. Each element changes the current value of the video URL contained in the first presentation and then changes the currently visualized presentation.



Figure 6.5 MARIA application: channel information presentation



Figure 6.6 MARIA application: channel selection presentation


```

1 <bodyGesture name="channelSelectionGesture">
2   <sequence iterative="true">
3     <shoulderLeft>
4       <accepts>
5         <invoke_function name="Predicate.axisParallel">
6           <parameter name="point1" data_ref="body:shoulderLeft"/>
7           <parameter name="point2" data_ref="body:shoulderRight"/>
8           <parameter name="axis" data_ref="x"/>
9         </invoke_function>
10      </accepts>
11      <completed>
12        <script>
13          <change_property interactor_id="trackingState"
14            property_name="properties/background/background_color"
15            property_value="#CC00CC00"/>
16          <change_property interactor_id="feedback"
17            property_name="text/string"
18            property_value="Tracking user !" />
19        </script>
20      </completed>
21    </shoulderLeft>
22    <disabling>
23      <disabling iterative="true">
24        <handRight iterative="true">
25          <completed>
26            <change_property interactor_id="channelMain"
27              property_name="focusPoint"
28              property_value="body:handRight"/>
29          </completed>
30        </handRight>
31      <openHandRight>
32        <accepts>
33          <invoke_function name="Predicate.handClosed">
34            <parameter name="point"
35              data_ref="body:openHandRight"/>
36          </invoke_function>
37        </accepts>
38        <completed>
39          <script>
40            <raise event_name="activation"
41              interactor_id="ui:channelMain/focusInteractor"/>
42          </script>
43        </completed>
44      </openHandRight>
45    </disabling>
46    <shoulderLeft>
47      <accepts operator="not">
48        <invoke_function name="Predicate.axisParallel">
49          <parameter name="point1"
50            data_ref="body:shoulderLeft"/>
51          <parameter name="point2"
52            data_ref="body:shoulderRight"/>
53          <parameter name="axis" data_ref="x"/>
54        </invoke_function>

```

55	<code></accepts></code>
56	<code><completed></code>
57	<code><change_property interactor_id="trackingState"</code>
58	<code>property_name="properties/background/background_color"</code>
59	<code>property_value="#CCCC0000"/></code>
60	<code><change_property interactor_id="feedback"</code>
61	<code>property_name="text/string"</code>
62	<code>property_value="Not tracking... " /></code>
63	<code></completed></code>
64	<code></shoulderLeft></code>
65	<code></disabling></code>
66	<code></sequence></code>
67	<code></bodyGesture></code>

Table 6.1: Channel selection gesture

We conclude this section describing in detail an XML excerpt from the sample application definition in order to provide a complete view of the different elements described in this section. The XML definition is shown in Table 6.1 and it defines the gesture for selecting a channel in the presentation shown in Figure 6.6.

The gesture definition is a sequence of two sub-gestures, that can be repeated an indefinite number of times (line 2). The first one is exploited in order to detect whether the user is in front of the screen or not through the turn gesture (line 3). The ground term predicate test (represented by the *accept* tag at line 4) is provided by a GestIT library function. Therefore, in MARIA we can model this functionality through an external function, and we can invoke it using the *invoke function* tag at line 4. Such function needs two body points (namely the shoulder left and right) and the axis for the comparison (X in our case, lines 5-9).

If this gesture completes successfully (line 11), we change the background colour of the *tracking state* grouping (which is visible at the bottom of Figure 6.6) to green (lines 13-15) and then we change the text in the feedback label to “Tracking user!” (lines 16-18).

The next step is modelling the hand pointing gesture (line 23-45), which is disabled (line 22) if the user turns and she is no more in front of the screen (line 46). The latter gesture is symmetric with respect to the one at line 3, it exploits the same external function for computing the ground term predicate (line 48). The only different is that this time the invocation result is logically negated (the *operator* attribute at line 47). If this gesture is recognized and the interaction is disabled, the *tracking state* grouping colour is reset to red and the feedback label is reset to “Not tracking”, providing

the user with a feedback on the system state (similar to the one shown at the bottom of Figure 6.5).

The hand pointing gesture (line 23-45) is an iterative repetition of hand movements (line 24-30), which is disabled by the hand closure (line 31-44). For each change of the hand position, the gesture model updates the *focusPoint* of the grouping containing all the presentation interactors, which has id *channelMain*, using the current right hand position (line 25-29). As already discussed in the previous sections, this has the effect of providing feedback for identifying the currently pointed interactor (Figure 6.6).

Finally, the channel is selected closing the right hand. The ground term accepts only the change from open to close provided that, at line 33, the model exploits another library function that returns if the hand is closed or not. If this is the case, it is possible to complete the interaction with the current presentation.

In order to do this, it is sufficient to raise the *activation* event of the currently pointed *activator* that, as already explained, it is possible simply exploiting the *raise* statement. Such statement is specified at line 40: the activator is maintained by the runtime support in the *focusInteractor* property of the *channelMain* grouping.

Chapter 7

Discussion

In this chapter, we discuss how it is possible to address a set of problems in the engineering and development of gestural interfaces, how they can be addressed by declarative approaches and, in particular, the one proposed in this thesis. The discussion contained in this chapter has been published in [124].

The first and the second problem are related to the gesture modelling in general, while the third is related to the compositional approach for gesture definition. The three problems we address can be summarized as follows:

1. *It is difficult to model a gesture only with a single event raised when its performance is completed.* The need for intermediate feedback forces the developer to redefine the tracking part. From now on, we refer to this issue as the *granularity problem*.
2. In [72], the authors state “*Multiouch gesture recognition code is split across many locations in the source*”. This problem is even worse if we consider full-body gesture recognition, which has a higher number of points to track, in addition to the other features (e.g. joints orientation, voice etc.). We refer to this issue as the *spaghetti code problem*.
3. A compositional approach for gestures has to deal with the fact that “*Multiple gestures may be based on the same initiating sequence of events*” [72]. This means that a support for the gesture composition has to manage possible ambiguities in the resulting gesture definition. We refer to this issue as the *selection ambiguity problem*.

In this chapter, we discuss the advantages of a declarative and compositional approach for gestural interaction, which are able to solve the aforementioned problems and we discuss how it is possible to support a cross-platform gesture definition exploiting the discussed approach.

7.1 Granularity problem

The granularity problem derives from the modelling of complex gestures with a single event notification when it completes. Due to the time duration of the interaction gestures, it is usually needed to provide intermediate feedback during the performance, with the consequent need to split the complex gesture in smaller parts.

In order to show the impact of such problem even for simple interactions, here we focus on two specific hand gestures we exploited in the touchless recipe browser (see section 5.4.4): the first one is a simple hand grab, which is used in the first and the second presentation for selecting an object. The second one is a hand-drag gesture we used for controlling the recipe preparation video: the user grabs the knob of the video timeline and then it moves it back and forth before “releasing” it by simply opening the hand.

Table 7.1 shows how it is possible to model such gestures with GestIT. The grab gesture is composed by an iteration of the hand movement (mH_r^*), which is disabled by a change on the feature that tracks the opened or closed status of the hand (cH_r in the expression).

We force the recognition only of a hand closure specifying the *closed* predicate, which accepts only changes from opened to closed. The grab gesture is a prefix for the drag one. Indeed, it is defined by a grab gesture followed in sequence by an iterative movement of the hand, disabled again by a change on the hand status, this time from opened to closed (modelled by the *open* predicate).

Grab	$mH_r^* [> cH_r[closed]$
Drag	$Grab \gg Release$ $Release = mH_r^* [> cH_r[open]$

Table 7.1 Grab and Drag gestures defined with GestIT

With GestIT it is possible to reuse the definition of the grab gesture for defining the drag one, as it is shown Table 7.1. However, the possibility to compose gestures with a set of operators does not guarantee the reusability of the definition.

Indeed, even in this simple example, the programmer needs a fine-grained control not only on the gesture itself, but also on its subparts. In the first two screens of the recipe browser application the grab gesture is exploited for an object selection, and the user has to be aware of which object she is currently pointing. Therefore, there is the need to provide intermediate feedback during the grab gesture execution. This is supported in the

application exploiting the fact that GestIT notifies the completion of the gesture sub-parts.

With this mechanism, the application receives a notification when each time mH_r is completed, highlighting the pointed object. The handler associated to the completion of the entire gesture performs the recipe selection and the presentation change.

It is worth pointing out that our meta-model does not make any assumption on the distance in time between two notifications for an iterative gesture. Since all device notifications are external with respect to the Petri Net that models the gesture, the device controls the event notification rate. If needed, the designer may specify some timing constraint using the predicates associated to the ground terms.

While performing the drag gesture, there is no need to attach a handler to the hand movement in the grab part, but it is sufficient to specify that the position in the video stream is changing after the grab completion, and to update it during the movement of the hand in the release part of the gesture.

It should be clear now how the declarative and compositional pattern offered by GestIT solves the granularity problem: the application developer is not bound to receiving a single notification when the whole gesture is completed. If needed, she is able to attach the behaviour also to the gesture sub-parts, handling them at the desired level of granularity.

In our approach, the finest granularity is represented by ground terms. They cannot be further decomposed into smaller components since they represent the features tracked by the recognition device.

7.2 Spaghetti code problem

The previous example may be used also for showing how it is possible to solve the problem of having the gesture recognition code spread in many places (spaghetti code problem). Indeed, the declarative and compositional approach to the gesture definition allow the developer to separate the temporal sequencing aspect from the UI behaviour while defining a gesture. This allows maintaining the gesture recognition code isolated in a single place.

In the example, the recognition code corresponds to the declaration of the gesture expression. The handlers define the UI behaviour, but they are not

part of the recognition code, since they are simply attached to the run-time notification of the gesture completion (or its sub-parts).

In this way it is not only possible to isolate the recognition code into a single application, but it is also possible to provide a library of complex gesture definitions, which may be reused in different scenarios, maintaining the possibility to attach the UI behaviour at the desired level of granularity.

In addition, the definition of the gesture is separated from the UI graphic control: it is not shipped with a particular image viewer or canvas, but it can be exploited in different UI configurations.

In this particular example, it would be possible to model the entire interaction instantiating a single complex gesture. Indeed, the *Grab* and the *Release* gestures differ only for the predicate on the change of the hand status feature. Therefore, it is possible to define with GestIT a complex gesture that is parametric with respect to this predicate.

Table 7.2 shows a different definition of the gestures in Table 7.1, which demonstrates the level of flexibility in the factorization of the gesture recognition code in the proposed framework.

Hand Status	$HandStatus[p] = mH_r^* [> cH_r[p]$
Grab	$HandStatus[closed]$
Drag	$HandStatus[closed] \gg HandStatus[open]$

Table 7.2: Grab and Drag gestures e parametric definition

7.3 Selection Ambiguity Problem

In this section, we show how the problem of possible ambiguities that may arise when composing gestures is handled in GestIT. We exemplify the problem through the simple 3D viewer application we introduced in section 5.4.3.

The interaction with the 3D model is the following: the user can change the camera position performing a “grabbing” the model gesture with a single hand and moving it, while it is possible to rotate the model executing the same gesture with both hands. The complete definition is shown Table 7.3.

<pre> Move [] Rotate Move = cH_r[closed] >> (mH_r* [> cH_r[open]]) Rotate = (cH_r[closed] cH_l[closed]) >> ((mH_r[d] mH_l[d])* [> (cH_r[open] cH_l[closed]))) </pre>

Table 7.3 Gesture definition for the 3D viewer application

The *Move* and the *Rotate* gestures are composed through a choice operator but, as it is possible to see in the definition, both gestures start with $cH_r[closed]$. Therefore, it is not possible to perform the selection immediately after the recognition of the first ground term, but the recognition engine needs at least one “look ahead” term, and the selection has to be postponed to the next event raised from the device. However, the two instances of $cH_r[closed]$ may have different handlers attached to the completion event, which should be executed in the meantime.

In general it is possible that, when composing a set of different gestures through the choice operator, two or more gestures have a common prefix, which does not allow an immediate choice among them. We identified three possible ways for addressing this problem. The different solutions have an impact on the recognition behaviour while traversing the prefix.

The first solution is the one proposed in [72], where the authors define an algorithm for extracting the prefix at design time. After having identified it, it is possible to apply a factorization process to the gesture definition expression, removing the ambiguity. This solution has the advantage that, since there is no ambiguity anymore, the recognition engine is always able perform the selection among the gestures immediately. The main drawback is that it breaks the compositional approach: after the factorization the two gesture definitions are merged and it is difficult for the designer to clearly identify them in the resulting expression. This leads to a lack of reusability of the resulting definition.

The second possible solution is again to calculate the common prefix at design time, without changing the gesture definition. In this case, the recognition support is provided with both the gesture definition and the identified prefix. During the selection phase at runtime, the support buffers the raw device events until only one among the possible gestures can be selected according to the pre-calculated prefix, and then flushes the buffer considering only the selected gesture.

This approach has the advantage of maintaining the compositional approach, while selecting the exact match for the gestures in choice: the

runtime support suspends the selection until it receives the minimum number of events for identifying the correct gesture to choose. Once the gesture has been selected, the application receives the notification of the buffered events.

The latter is the main drawback of this approach: the buffering causes a delay on the recognition that is reflected on the possibility to provide intermediate feedback while performing the common prefix gesture. Another drawback is that the common prefix has to be calculated at design time, which may need an exponential procedure for enumerating all the possible recognizable event sequences, which are needed for extracting the common prefix. For instance, an order independence expression with n operands in GestIT recognizes $n!$ event sequences, since we should consider that the operands can be performed in any order.

The third solution is based on a best effort approach, and is the one implemented by GestIT. When two or more expressions are connected with a choice operand, the recognition support executes them as if they were in parallel. If the user correctly performed one of the gestures in choice, when the parallel recognition passes the common prefix only one among the operands can further continue in the recognition process.

At this point the choice is performed and only one gesture is successfully recognized, and the support stops trying to recognize the others. This approach solves the buffering delay problem of the previous solution, since the effects of the gestures contained into the common prefix is immediately visible for the user.

However, in this case the recognition support notified the recognition of the gestures included in the common prefix of all the operands involved in the choice. Consequently, the UI showed the effects associated to all of them, while only the ones related to the selected gesture should be visible. In order to have a correct behaviour, we need a mechanism to *compensate* the changes made by the gestures that were not selected by the recognition support, which means to revert the effects they had on the UI. Such mechanism can be supported through another notification, signalling that the recognition of a gesture (ground term or complex) has been interrupted. In this way, it is possible for the developer to specify how to compensate the undesired changes. This is the main drawback for this solution: the developer is responsible of handling the compensating actions.

In order to better explain how this solution works, we present a small example of compensation. We consider the gesture model in Table 7.3, which

allows the user to move and to rotate a 3D model. The UI provides intermediate feedback during the gesture execution in the following way: a four-heads arrow while the camera position is changing, and a circular arrow while the user is rotating the model.

We suppose in our example that the user performs the grab gesture with both hands and we describe the behaviour of the recognition support during the recognition of the common prefix (in this case $ch_r[closed]$) and after the gesture selection has been performed.

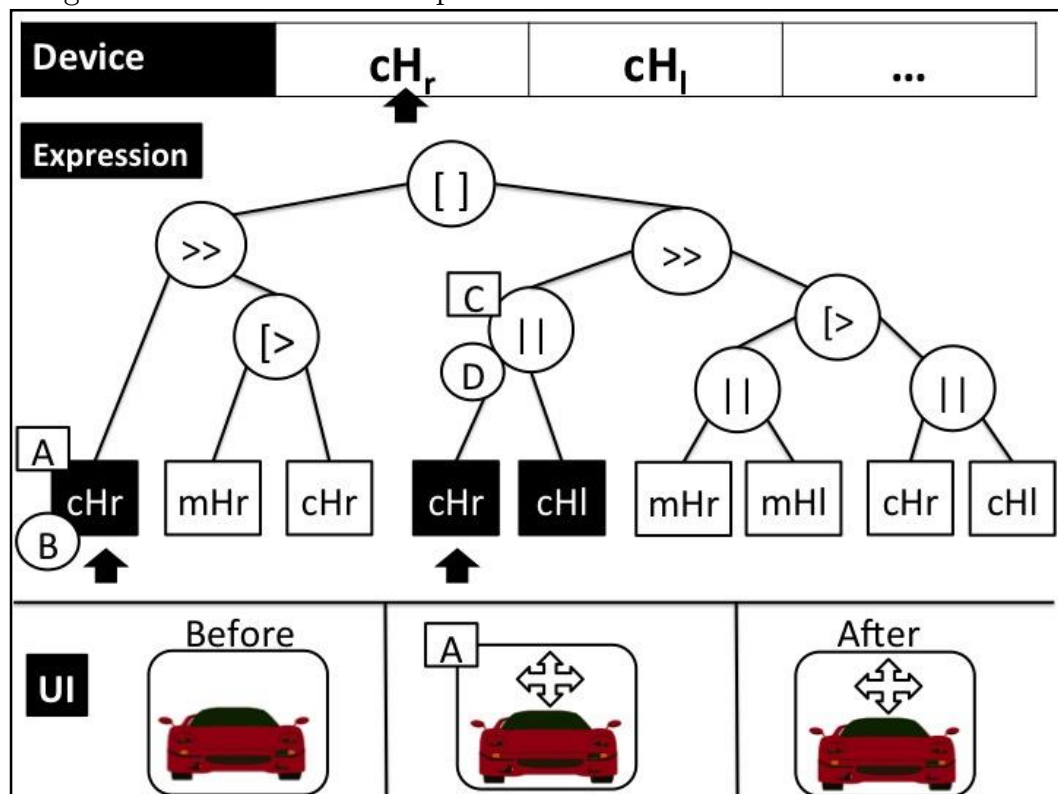


Figure 7.1 Common prefix handling for the choice operator (1)

The common prefix handling is depicted in Figure 7.1: the upper part represents the stream of updates that comes from the device, the black arrow highlights the one that is currently in progress. The central part shows the gesture expression represented as a tree, with the ground terms that can be recognized immediately highlighted in black (we do not show the predicates associated to the ground terms, since for this example we suppose that they are always verified).

Some tree nodes are associated to rectangular and circular badges, which represent respectively the completion and the compensation behaviour. Such handlers are external with respect to the gesture description and are defined by the developer. The lower part shows the effects on the UI of the gesture

recognition. The left part depicts the UI before the recognition, the middle part shows the intermediate effects, while the right one shows the resulting state after the recognition.

During the recognition of the common prefix, the support behaves as follows: after receiving the update coming from the device, the support executes the two instances of cH_r , highlighted by the black arrows in Figure 7.1, central part. Since the leftmost one has an associated completion handler (the A rectangular badge), the recognition support executes it. Therefore the UI changes its state and an arrow is shown above the 3D model (Figure 7.1, lower part).

After that, the expression state changes (two ground terms have been recognized) and we have the situation depicted in Figure 7.2: the ground terms with a grey background have been completed, therefore the ground terms that may be recognized at this step are mH_r or cH_l . Since the next device update we are considering is cH_l (Figure 7.2, upper part), the recognition support is now able to perform the selection of the right-hand part of the expression tree, while the left-hand part cannot be further executed.

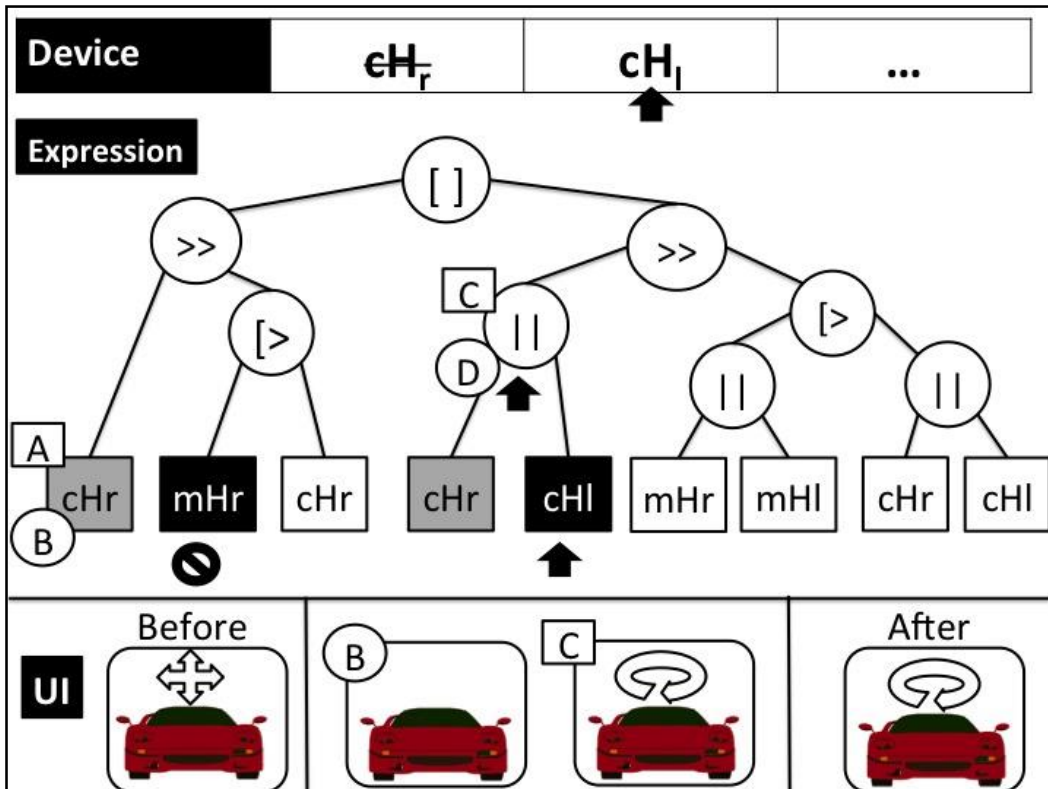


Figure 7.2 Common prefix handling for the choice operator (2)

Therefore, the latter sub-tree needs compensation, which consists of invoking the handlers associated to all the expressions previously completed (CH_r). In our example, this corresponds to the execution of the handler identified with the B circular badge, which hides the four-heads arrow. After that, it is possible to continue with recognition of the gesture: the CH_l ground term in the right-hand part of the expression is completed and also the parallel expression highlighted with a black arrow in Figure 7.2.

Consequently, the recognition support executes the completion handler represented with the C rectangular badge, which shows the circular arrow for providing the intermediate feedback during the model rotation, and the gesture recognition continues taking into account only the *Rotate* gesture. The effects of the handlers on the UI for this step are summarized by the lower part of Figure 7.2: before the recognition of the ground term, it was visible on the UI the four-head arrow, which has been hidden by the B compensation handler. The C completion handler instead showed the circular arrow that determines the state of the UI after the ground term recognition.

From a theoretical point of view, the proposed solution considers the set of gestures in choice as instances of long-running transactions [47] but in this case the components involved are not distributed. In case of failure of such kind of transactions, it is not possible in general to restore the initial state, as happens with the effects on the UI of the gestures that are not selected by the choice. Instead, a compensation process is provided, which handles the return to a consistent state. There is a large literature on how to manage long-running transactions, in [34] the authors provide a good survey on this topic.

7.4 Cross-platform gesture modelling

In this section, we discuss an advantage provided by the compositional and declarative approach for modelling gestural interaction. Since such definition is based on a set of building blocks (ground terms), connected through a set of well-defined composition operators, it is possible to create interfaces that share the same gesture definition across different recognition platforms finding a meaningful translation of the source platform ground terms towards the target one.

This opens the possibility to reuse the gesture definition not only for different applications that exploit the same recognition device but also, if

the interaction provided still makes sense, with different devices that have different recognition capabilities.

In order to explain how such reuse is possible, we report here on a first experiment we conducted with the two platforms supported by GestIT: multitouch and full-body.

We started from the simple drawing canvas application for iPhone we described in section 5.4.1, which supported the pan gesture for drawing and the pinch gesture for zooming. Such gestures were connected through the choice operator (see Table 7.4).

$ \begin{aligned} &Pan \ [\] Pinch \\ &Pan = Start_1 \gg Move_1^* \ [> End_1 \\ &Pinch = (Start_1 \ = \ Start_2) \gg (Move_1^* \ \ Move_2^*) \ [> \\ &\quad (End_1 \ = \ End_2) \end{aligned} $
--

Table 7.4 Simple drawing canvas gesture modelling

In order to create a Kinect version it is not possible to reuse directly the gesture definition, because concepts as pan, pinch, touch etc. do not have any meaning in such device. However, having a precise definition of the gestures allows us also to define precisely new concepts. In our case, what is missing is a precise definition of what a touch start, a touch move and a touch end are. If we add a precise definition of these concepts, all the gestures that have been constructed starting from such building blocks will be defined consequently.

One simple idea is to associate a point that represents a finger position on the iPhone to the position of one hand with the Kinect (therefore, the maximum number of touch points is two). In addition, we have to define a criterion for distinguish when the touch starts and when the touch ends. A simple way we discussed many times in this thesis, is to rely on the depth value of the position of a given hand: if it is under a certain threshold, we can consider that the user is “touching” our virtual screen, otherwise we do not consider the current hand position as a touch.

More precisely, we need to define the multitouch basic gestures according to the 3D position of the left and right hand, indicated respectively as $l = (x_l, y_l, z_l)$ and $r = (x_r, y_r, z_r)$. Moreover, we have to define a plane, which represents the depth barrier for the touch emulation, as $T_p = (x, y, k)$ where k is a constant. The complete definition can be found in Table 7.5.

It is worth pointing out that, even if we used such definition for a quite “extreme” change of platform, the redefinition of the ground term allows us

to support with the Kinect platform all the multitouch gestures that involve no more than two fingers, which are the large majority of those used in such kind of applications.

Obviously, from the interaction design point of view it may be a bad idea to port multitouch gestures to the full body gesture recognition support directly, and the example should be considered only as a proof of concept. However, such kind of approach may be used for those devices that are exploited for recognizing gestures in similar settings.

For instance, it can be useful for designing applications that recognize the same full body gestures with a remote or a depth camera-based optical device. In this case, having such kind of homomorphism may reduce the complexity in supporting different devices.

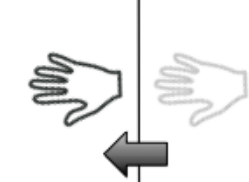
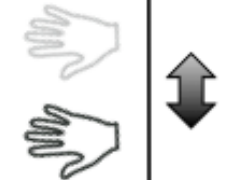
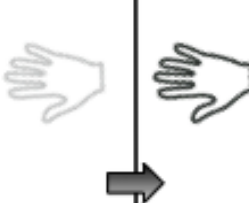
Multitouch Ground Term	Interaction
$Start_1 = r[z_r(t - 1) > k \wedge z_r(t) \leq k]$ $Start_2 = l[z_l(t - 1) > k \wedge z_l(t) \leq k]$	
$Move_1 = r[z_r(t - 1) \leq k \wedge z_r(t) \leq k]$ $Move_2 = l[z_l(t - 1) \leq k \wedge z_l(t) \leq k]$	
$End_1 = r[z_r(t - 1) \leq k \wedge z_r(t) > k]$ $End_2 = l[z_l(t - 1) \leq k \wedge z_l(t) > k]$	

Table 7.5 Mapping multitouch ground terms to the full-body platform

Chapter 8

Evaluation

The definition of methods and techniques for the evaluating a new user interface description language or tool is an open problem for the HCI community [105]. An evaluation with real users (gesture interface developers in our case) requires time for setting-up a community around the new tool, waiting for the development of real-world applications with the proposed solution. Therefore, in this thesis we opted for an inspection-based approach, similar to a heuristic evaluation [100].

The inspection we report in this chapter follows three list of criteria:

1. A review of the meta-model requirements that we identified from the state of the art analysis, detailed in section 8.1.
2. The five themes identified by Myers et al. [97] for assessing a specification tool: the tool target, the threshold and ceiling, the path of least resistance, predictability and the adaptation to moving targets. In section 8.2, we define each one of these aspects and we inspect our modelling language accordingly.
3. The *Cognitive Dimensions Framework* by Green and Petre [51], which sets a small vocabulary for cognitive aspects of a language structure. In section 8.3, we provide the definition of the different dimensions and we discuss the weakness and the strength of our approach.

Finally, we provide some data on the recognition library performance, comparing a version of the 3D viewer application created tracking gestures with a simple Finite State Machine and the version we created with GestIT.

8.1 Requirements review

In this section, we review the requirements for definition of the gesture meta-model, which have been identified through an analysis of the state of the art in section 2.3.

8.1.1 Temporal evolution

The meta-model must describe the gesture temporal evolution. The developers should be able to define the behaviour of the user interface according to this temporal evolution, without the need of tracking explicitly the different stages of the gesture performance outside the model definition.

The meta-model proposed in this thesis describes the temporal evolution of different gestures through a compositional approach. Gestures are modelled starting from a set of ground terms, which are connected together through a set of formally defined composition operators.

Each one of the different terms that compose the gesture (either simple or complex) provides an event for its recognition. The developer can attach the definition of the UI behaviour to this event, separating it from the gesture definition.

In addition, each term provides an event for notifying an error during its recognition, in order to allow the developer to recover the UI changes due to a partial recognition of the considered gesture.

8.1.2 Granularity

Provided that a gesture may take seconds to complete, it must be possible for developers to define user interface reactions to partially completed gestures, not only to their complete recognition.

Each one of the terms that compose a gesture definition compliant with the meta-model proposed in this thesis provides a notification for its successful completion. Therefore, the developer can attach handlers not only to the completion of the whole gesture, but also to all its sub parts. Such handlers define the UI behaviour with different levels of granularity, allowing to provide intermediate feedback during the gesture performance.

The maximum level of granularity is provided by the ground terms, which correspond to all the features that may be tracked by the gesture recognition device (e.g. the position of the touches for multitouch screens, the joint position for depth cameras etc.).

8.1.3 Separation of concerns

The definition of gestures and the user interface behaviour must be separated, in order to allow the reuse of the same gesture model in different applications.

The definition of a gesture model does not contain any information on the associated behaviour. That aspect of the UI can be defined specifying a set of event handlers for the completion of the whole gesture or its subparts.

Such notification mechanism allows reusing the definition of the same gesture in different applications with different effects.

8.1.4 Multiple recognition devices

The meta-model must support different recognition devices, abstracting from a particular recognition technology.

The proposed meta-model is abstract with respect to a particular recognition device. In this thesis, we discussed how it is possible to support multitouch and full-body interaction. In the same way, it is possible to add further recognition platforms: through the definition of the set of ground terms that are specific to the new platform.

The proposed set of composition operators has been defined independently from any recognition technology.

8.1.5 Parallel interaction

The meta-model must handle the recognition of different gestures at the same time, in order to allow parallel interactions with the same application.

We included a parallel composition operator in our meta-model. Such operator allows the simultaneous recognition of the different connected terms. The parallel operator support parallel interactions for both single and multiple users.

8.1.6 Equivalent descriptions

The same gesture can be performed in different ways (e.g. a pinch may be performed either with one hand or with two hands). The meta-model must support the definition of equivalent gestures.

It is possible to specify two equivalent gestures connecting them with the choice operator, which guarantees the recognition of exactly one among the connected terms. In this way, it is possible to provide different gestures with an equivalent effect.

8.1.7 Selection ambiguity

The recognition support must provide means for identifying or managing the selection between two different gestures that shares the same initial sequence.

GestIT provides a best-effort solution for the selection ambiguity problem: when two gestures are connected in choice and they share a common prefix, the library supports the parallel recognition of both gestures. When such prefix has been entirely recognized, only one between the two gestures can continue its recognition. The other one raises an event related to its erroneous recognition, which can be exploited by the developer for compensating the previous changes to the UI. In this way, the support allows the developer to identify such situation. For further details, see section 7.3.

8.2 Five themes in evaluating tools

The work by Myers et al [97] contains a review of different tools that have been used in both research and industrial settings for creating user interfaces. The authors identify a set of themes that help in identifying strength and weaknesses and in explaining the reasons behind the success or the failure of different approaches.

In this section, we report the definition of the five themes and we inspect GestIT accordingly.

8.2.1 Parts of the user interface that are addressed

In [97], Myers et al. stated that the successful tools in the development of UIs had a precise target, and they limited their scope only to the task that was needed.

Our modelling approach is focused on the description of gestures, limiting to their temporal evolution. We do not aim to redefine again the other parts of the UI, such as the layout or the behaviour. Instead, we provide a different approach for describing an aspect of the UI definition that is currently spread among different parts of the code, creating what we call the *spaghetti code* problem (see section 7.2).

In addition, the proposed modelling technique allows reusing a gesture definition in different applications, since it is possible to attach the behaviour not only to the whole gesture, but also to its sub-parts (the *granularity problem* see section 7.1).

The proposed solution, as demonstrated by the supporting library, can be employed with different UI toolkits and do not enforce the developers to select a specific technology, therefore it does not interfere with others aspects of the development.

8.2.2 Threshold and ceiling

According to [97], the *threshold* is “how difficult is to learn how to use the system”, while the *ceiling* is “how much can be done using the system”.

In the ideal tool, the threshold is low, while the ceiling is high. This means that the developer or the designer are able to use the tool with little or no training at all and the tool is able to cover appropriately every type of UI that should be created.

In order to evaluate the threshold we should have data on the time needed for learning how to model gestures with GestIT, starting from scratch. Unfortunately, at the time of writing we have no sufficient data for drawing any conclusion. However, we can point out here that the model is based on two different concepts that are familiar for UI developers.

The first are the device related events (e.g. the one related to the touches or the joint positions) that should be understandable for people who design gestural interaction, since they are commonly used in all toolkits.

The second concept is the description of the evolution of the gesture through the time with a set of temporal operators. Such operators are well

known in other contexts and languages, for instance such the CTT [114] for task modelling or LOTOS [18] for process modelling. Therefore, people who already know the semantics of the different temporal operators may apply such knowledge in a different context. Otherwise, the learning path should not take longer with respect to the aforementioned languages and, since they are widely applied in their respective areas, it should be reasonable to claim that the temporal operators will not constitute a problem for adopting GestIT.

With respect to the ceiling aspect, we can claim that the proposed modelling technique covers adequately the target interaction. This is supported by the different examples of models that we provided in Chapter 4, which cover a broad set of gestures. In addition, we demonstrated that it is possible to apply the model to different existing gesture recognition platforms and that the approach can be easily extended for new ones.

8.2.3 Path of Least Resistance

This *path of least resistance* aspect is about how “tools influence the kinds of user interfaces that can be created. Successful tools use this for their advantage, leading implementers towards doing the right things, and away from doing the wrong things” [97].

We are confident that our modelling technique is able to “force” the developers to:

1. Create gesture definitions separated from other UI aspects, such as the layout and the behaviour (see section 7.2)
2. Provide means for inspecting the gesture definition and to define reactions at the desired level of granularity (see section 7.1).

Such advantages are provided by the way the developer creates the gesture definition in GestIT, and they require the only effort of adopting the model, without assuming any additional technique or pattern.

8.2.4 Predictability

The *predictability* aspect is about the fact that “tools which use automatic techniques that are sometimes unpredictable have been poorly received by programmers”.

Although we do provide an automatic support for recognizing gestures modelled through the proposed notation, we can claim that such aspect do not represent an issue for GestIT. Indeed, we provided a precise formal

definition of both the terms and the composition operators that are involved in a gesture definition. This helps the developers in understanding and predicting the runtime behaviour of the defined model.

However, we are aware that not all developers may be interested in studying the formal definition of the meta-model. Therefore, it may be useful to provide an high level (but obviously imprecise) description of the compositional operators and support the development with an interactive simulator, that may help the developer in finding out himself the recognizer behaviour against a particular event sequence. Such approach has been proved useful for the same set of composition operators in task modelling [114].

8.2.5 Moving Targets

The *moving targets* aspect is related the fact that, in order to provide a useful support, designer must have a different understating of the target tasks. However, since the development of UI evolves with at a high speed, once the knowledge about how to support a given task is mature, it is possible that such support is no more needed, since the task has become obsolete.

In our case, the moving targets problems does not apply to the gestural interaction itself, since it exists from at least 30 years now, and we can be positive that it will last for a long time. However, it may be related to the change of the supporting technology for recognizing gestures. Indeed, this field proposes an increasing number of recognition devices, and it may happen that a new one device overtakes the capabilities of the existing ones. Therefore, it may be reasonable to shift the development towards a new device even if the supporting tools for the old one are more mature and stable.

We tried to create a model that can be tailored for supporting new devices, providing a definition of ground term that can already cover devices that employ different technologies for tracking gestures. We considered this aspect from the very beginning in order to create a modelling technique able to last more than the recognition devices.

8.3 Cognitive Dimensions Framework

The cognitive dimensions [51] define set of cognitive-related aspects that capture how the structure of a notation influence their usability. The aspects are called “dimensions” since they are supposed to be orthogonal characteristics of a given notation, which may need a trade-off against each other.

Such kind of evaluation enables scientists to broaden the scope while evaluating a given notation, without limiting to the notation expressiveness. The framework is applied usually to visual notations but, as stated by the authors, may be used also for non-interactive notations.

The methodology we followed is similar to a heuristic evaluation [100]: we inspected our notation considering each one of the cognitive dimensions and we report on the results of such inspection. In order to guide our inspection, we used the questionnaire in [13], which has been created by the cognitive dimensions framework authors.

8.3.1 Abstraction gradient

What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

The minimum level of abstraction depends on the feature that can be tracked by a specific device. Obviously, this is a lower bound for the notation, since it is not possible to split in sub-parts such features.

We do not impose any upper bound to complex gestures, they can ideally consist of any combination of complex and simple gestures. However, the fact that we are modelling interactive applications sets an upper bound for the model complexity: the feedback has to be delivered in a timely manner during the gesture performance, otherwise the abstractions are useless.

8.3.2 Closeness of mapping

What ‘programming games’ need to be learned?

This aspect is related to the distance between the mental model that a developer has about a specific notation and the construct that such notation provides.

The elements of our model corresponds exactly to the notation we provided. The different phases of a gesture performance can be mapped to those

different terms that can be connected together, according to the gesture analysis. We provide different examples of such kind of mapping while we describe how we modelled different gestures for multitouch and the full-body interaction in Chapter 4: we provide a high level description of the different phases and then we map them to different terms, composed through temporal operators.

8.3.3 Consistency

When some of the language has been learnt, how much of the rest can be inferred?

The notation exploits similar programming language constructs for similar elements in the model, which would help the developers in inferring i.e. how it is possible to connect complex gestures starting from the knowledge they acquire connecting ground terms.

Understanding such compositional concept through ground terms is easier, since the space of all possible combinations that can be recognized remains small. In order to stress such similarity, simple and complex gestures share the same base class. Consequently, they can be connected through temporal operators in the same way even at the programming level.

8.3.4 Diffuseness

How many symbols or graphic entities are required to express a meaning?

The notation is reasonably brief and it contains an element for each of the different ground terms that need to be composed and for all the temporal relations that need to be expressed.

The predicates take more space to define, since they may include different accesses to the gesture state and may contain a complex logic.

8.3.5 Error proneness

Does the design of the notation induce ‘careless mistakes’?

The most common mistake so far is forgetting to set the iterative flag to a term. This obviously causes a strange UI behavior, and sometimes it results difficult to find exactly where this slip happened

In addition, the object-oriented notation has the problem of connecting the objects through variable names that may be in a high number. Therefore, it is possible that the developer erroneously connects two or more terms that should not be connected.

8.3.6 Hard mental operations

Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?

Most mental effort is required for identifying the different parts of the gesture performance and to generalize them in a way that is appropriate for different “styles” that may be encountered with different people. Therefore, it may be useful to sketch on paper or on different media some graphs or schemas for identifying such different parts.

In addition, it may be difficult to work out all the different combinations that are possible when two gestures are connected in parallel. This may have consequences if the UI resources they access enter in conflict.

8.3.7 Hidden dependencies

Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?

In our notation, it is possible to have hidden dependencies among the predicates associated to the different ground terms. Indeed, some of them may depend on each other (e.g. one is the logical negation of the other) but, since the different predicates are referenced by name, such kind of relationships are not immediately visible. The same holds for the behavior definition, but is less frequent to reuse exactly the same definition.

8.3.8 Premature commitment

Do programmers have to make decisions before they have the information they need?

The programmers can follow different paths for reaching the same model. They can start by defining all the terms that need to be composed and then define the associated predicates, or they can choose to define completely each one of the terms before composing them. In addition, they may also choose to define the effects of the different commands in advance and to provide gestures for executing them, or they can first select the gestures and then define the effects.

This means that the developer may take decisions about the interaction when he has all the information needed.

8.3.9 Progressive evaluation

Can a partially complete program be executed to obtain feedback on ‘How am I doing?’

The compositional structure enables you to create the whole gesture definition iteratively, trying the different parts in isolation, or composed with a subset of terms. This allows achieving a good gesture model even by trial and error.

8.3.10 Role expressiveness

Can the reader see how each component of a program relates to the whole?

The different modelling constructs are mapped on different syntactical elements in both the XML and the code notation. However, the tree structure of the XML notation allows the developer to visualize the relations between the different gestures and sub-gestures in the whole expression.

Understanding such relationship through the code notation is more difficult, since the developer has more freedom on the declaration structure and may interleave the gesture specification logic with code related to other aspects of the UI (e.g. graphics controls).

8.3.11 Secondary notation

Can programmers use layout, colour, other cues to convey extra meaning, above and beyond the ‘official’ semantics of the language?

With the current notation, it is not possible to provide hints to the developer for identifying different parts of the model other than writing some comments on the XML or object-oriented code.

This is an aspect that can be considered in future work in an appropriate authoring environment, since it would be useful to immediately identify expressions that belong to two different complex gestures that are composed in order to define the whole gestural interaction. For instance, if we consider the 3D viewer application in section 5.4.3, there should be some way for differentiating the sub-expression belonging to the *Grab* from those belonging to the *Roll* gesture.

8.3.12 Viscosity

How much effort is required to perform a single change?

Making a change is easy once the expression corresponding to the phase of the gesture performance has been identified. It may be difficult to find the phase if predicates or the attached methods for defining the behavior does not have meaningful names. Eventually, for really long expressions, it is possible to use comments for identifying the expression parts.

There are no changes that are more difficult than others, all of them require about the same effort.

8.3.13 Visibility

Is every part of the code simultaneously visible (assuming a large enough display), or it is at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

The various part of the notation can be identified easily, since only the different terms of the expression can be instantiated as objects, while the connection between the different terms are possible through methods. In the XML definition, the tag names and their structure allow to distinguish the different parts of the defined gestures.

If the gesture is defined through the XML notation, there can be some difficulties in identifying the predicates that can be optionally attached to the ground terms and the methods that define the behavior, since they are defined in a different file.

There is space for improvement, and in a possible authoring tool it should be possible to navigate from the XML definition to the code behind.

It is possible to see the different parts if they are defined in the same UI (e.g. the same code file). Otherwise, the user should work of two different files before the combination. The same holds for the comparison.

8.4 Performance analysis

In this section, we discuss the results of a preliminary analysis of the GestIT library performance. Even if the implementation described in this thesis is a proof of concept, we show here that the overhead introduced by the library does not invalidate the entire application performance.

The discussed analysis is not complete, but it shows that the required resources are reasonable for the advantages provided by the library. A throughout discussion of how to create a high-performance version of the library is beyond the scope of this thesis.

We analyze the performance of the 3D car viewer application, discussed in section 5.4.3. We recall that the application is able to show a 3D model of a car, which can be moved through a grab gesture and/or rotated through a roll gesture. The application tracks the user only if she stands in front of the screen (with the shoulders contained in a plane roughly parallel to the one of the sensor).

Using the GestIT notation, the interaction can be modelled with the equation 8.1.

$$\begin{aligned}
Front &\gg (mH_r^* | | mH_l^* | | (Grab [] Roll)))^* [> NotFront] & (8.1) \\
Front &= (S_l[p] | | S_r [p]) \\
NotFront &= (S_l[!p] | | S_r [!p]) \\
Grab &= oH_r[c] \gg (mH_r^* [> oH_r[o]) \\
Roll &= (oH_r[c] | | oH_l[c]) \gg \\
&\quad ((mH_r[d] | | mH_l[d])^* [> (oH_r[o] | | oH_l[o]))
\end{aligned}$$

In order to estimate the overhead introduced by the library, we created a version of the 3D viewer without using the GestIT library, which recognizes the gestures through a simple Finite State Machine (FSM). This application provides the same interaction capabilities with respect to the GestIT version, thus it can be considered as a baseline implementation for the 3D viewer.

The FSM defined in the baseline version of the 3D viewer application is shown in Figure 8.1. The recognition starts with the *not front* state, where the application the position and the rotation of the 3D model cannot be changed. When the users is in front of the screen, the current state changes to *front* (firing the *parallel* transition). The application is now ready for accepting the input through the grab and the roll gestures. In this state, the UI shows a green label with the text “Tracking” for informing the user that the application is ready for tracking gestures.

From this state, the FSM is able to recognize the grab gesture firing a transition for each hand. If the user closed the right hand (*close DX*), the state is updated to *DX closed*, and the interface shows a four arrow icon, indicating that the 3D model can be moved iteratively. The firing of the *move DX* transition updates the model position. The recognition of the grab gesture for the left hand is symmetric (*close SX* – *SX closed* – *move SX*).

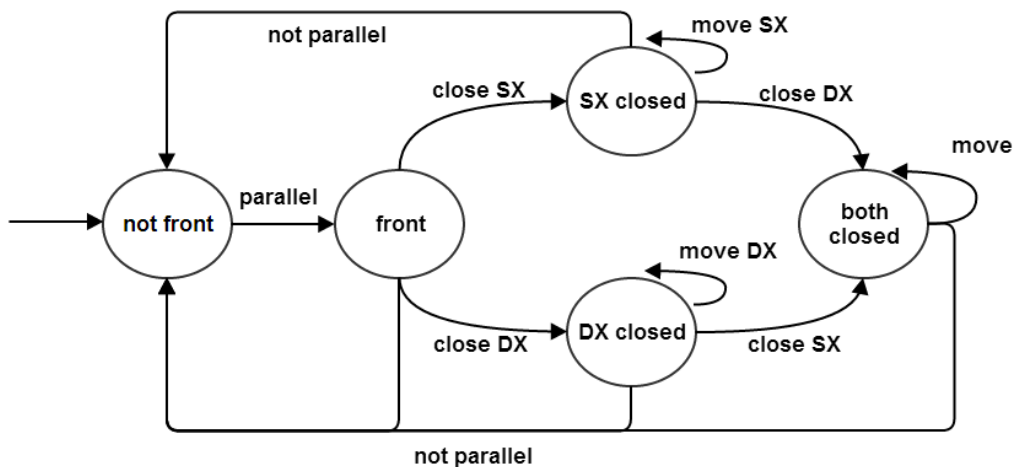


Figure 8.1 Finite State Machine for the 3D viewer interaction

The roll gesture can be recognized closing right or the left hand when the current state is respectively *SX closed* or *DX closed*. The associated transitions *close DX* and *close SX* changes the current state to *both closed*, where the model can be rotated and the application shows a circular arrow for suggesting that the rotation angle can be updated moving both hands (the *move* transition). From all the states included in the FSM, if the user is no more in front of the screen, the gesture recognition is interrupted and the current state is set to *not front*.

In the following sections, we report the resources (CPU and memory) consumed by both versions of the application. In order to remove the input variability in the comparison, we recorded the interaction sequence with Kinect Studio [90]. After that, we profiled with Visual Studio 2012 [94] both

versions of the applications in order to collect the CPU and memory consumption data.

All tests were performed with the following configuration:

- CPU: Intel Core i5-3470, 3.20 GHz
- RAM: 8.00 Gb
- OS: Windows 8 Pro, x64
- Kinect for XBOX 360

8.4.1 CPU (sampling)

In this section, we report the CPU profiling data, obtained through sampling. This profiling method interrupts the processor at set intervals, collecting the list of functions contained in the call stack. At the end of the profiling session, we obtain for each function the number of times that it was contained in the call stack. Therefore, the functions using more CPU have a higher sample count.

We start the analysis from the overall CPU usage percentage. Figure 8.2 shows the CPU consumption for the FSM-based version, while Figure 8.3 shows it for the GestIT version. As it is possible to see, there is no meaningful difference between the two line graphs, which have a similar trend. In both versions, the CPU consumption never went above the 40%. The overall trend of the two lines indicates that the resources consumed by the GestIT library does not have a sensible impact on the overall performance of the application.

In order to analyse this aspect more in detail, we report in Table 8.1 and Table 8.2 the sample count respectively for the FSM and the GestIT versions. In both tables, the counters are grouped by DLL, since we are not interested in establishing exactly which function is consuming more resources, but we limit granularity of our analysis at the software component level. For the FSM version, we isolated the state machine definition into a specific library, the *FsmGestureRecognition.dll*. In order to evaluate the consumption for the GestIT version, we have to consider two different DLLs: the *Gestit.dll*, which contains the definition of the temporal operators and the abstract classes for the ground terms, and the *BodyGestit.dll*, which contains the extension of the base classes for supporting the Kinect device through our modelling approach. The *FsmGestureRecognition.dll* is highlighted in Table 8.1 while the *Gestit.dll* and the *BodyGestit.dll* are highlighted in Table 8.2

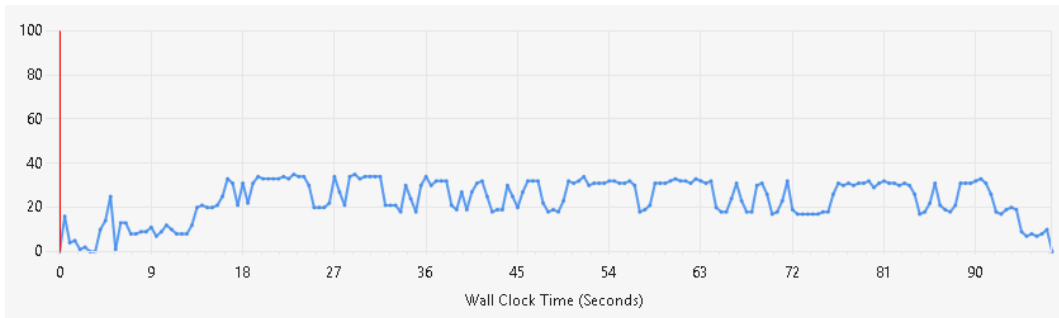


Figure 8.2 3D viewer CPU usage (FSM version)

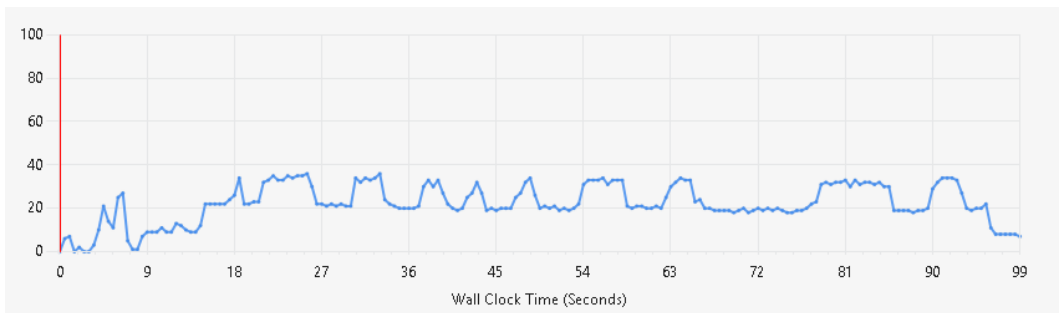


Figure 8.3 3D viewer CPU usage (GestIT version)

In both tables we report the following data, as defined in [91]:

- *Inclusive samples*: the total number of samples that are collected during the execution of the target function. It includes the samples collected during the execution of child functions, which have been called by the target one.
- *Exclusive samples*: the total number of samples that are collected during the execution of the instructions of the target function, without counting those belonging to child functions.
- *Inclusive percent*: the percentage of the total number of inclusive samples in the profiling run.
- *Exclusive percent*: the percentage of the total number of exclusive samples in the profiling run.

As it is possible to see in Table 8.1 and Table 8.2 the gesture recognition DLLs produced similar results for both versions: the state machine DLL was included in the 18.07% of the samples and occupied exclusively the CPU for the 0.02%. The *BodyGestit* functions inclusively occupied the CPU for the 19.22% and exclusively for the 0.07% of the samples, while the *Gestit* functions for the 0.18% inclusively and 0.01% exclusively.

The difference is about 1.5% for the inclusive samples and the 0.08% for the exclusive samples.

Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
nvd3dum.dll	9253	7542	51.40	41.90
PresentationFramework.ni.dll	8747	2593	48.59	14.40
CarViewer.exe	8747	19	48.59	0.11
FsmGestureRecognition.dll	3252	3	18.07	0.02
Microsoft.Kinect.Toolkit.Interaction.dll	3050	3050	16.94	16.94
PresentationCore.ni.dll	2066	2066	11.48	11.48
d3d9.dll	1505	1505	8.36	8.36
Microsoft.Kinect.dll	720	720	4.00	4.00
ntdll.dll	163	163	0.91	0.91
WindowsBase.ni.dll	145	142	0.81	0.79
Microsoft.Kinect.Toolkit.dll	132	1	0.73	0.01
clr.dll	117	117	0.65	0.65
mscorlib.ni.dll	36	36	0.20	0.20
gdi32.dll	21	21	0.12	0.12
nvSCPAPI.dll	16	0	0.09	0.00
nvapi.dll	9	0	0.05	0.00
3DTools.dl	8	0	0.04	0.00
setupapi.dll	5	5	0.03	0.03
user32.dll	5	5	0.03	0.03
KernelBase.dl	4	4	0.02	0.02
wow64cpu.dll	4	4	0.02	0.02
kernel32.dll	2	2	0.01	0.01
msvcrt.dl	2	2	0.01	0.01
dxgi.dll	1	1	0.01	0.01
rxinput.dll	1	0	0.01	0.00

Table 8.1 3D viewer CPU profiling (sampling, FSM version)

Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
PresentationFramework.ni.dll	9470	2623	52.67	14.59
CarViewer.exe	9470	23	52.67	0.13
nvd3dum.dll	8508	5710	47.32	31.76
BodyTmpLib.dll	3455	13	19.22	0.07
Microsoft.Kinect.Toolkit.Interaction.dll	3217	3217	17.89	17.89
d3d9.dll	2648	2648	14.73	14.73
PresentationCore.ni.dll	2406	2406	13.38	13.38
Microsoft.Kinect.dll	865	865	4.81	4.81
WindowsBase.ni.dll	144	140	0.80	0.78
Microsoft.Kinect.Toolkit.dll	141	5	0.78	0.03
clr.dll	137	137	0.76	0.76
ntdll.dll	123	123	0.68	0.68
microsoftlib.ni.dll	40	40	0.22	0.22
TmpLib.dll	32	1	0.18	0.01
gdi32.dll	17	17	0.09	0.09
wow64cpu.dll	7	7	0.04	0.04
3DTools.dll	3	0	0.02	0.00
kernel32.dll	2	2	0.01	0.01
msvcrt.dll	2	2	0.01	0.01
nvSCPAPI.dll	1	0	0.01	0.00

Table 8.2 3D viewer CPU profiling (sampling, GestIT version)

8.4.2 CPU (instrumentation)

We repeated the profiling experiment using the instrumentation profiling method, which records detailed timing information about the execution of the application code, injecting some profiling code at the start and the end of each target function [92]. We again grouped all the counters by the containing DLL.

The values recorded in this second experiment are the following:

- *Number of calls*: the total number of calls to the target function
- *Elapsed inclusive time percentage*: the percentage of time spent executing the target and the child functions
- *Elapsed exclusive time percentage*: the percentage of time spent in executing the target function, without considering child functions
- *Elapsed inclusive time*: the total time spent in the target and the child functions (milliseconds).
- *Elapsed exclusive time*: the total time spent in the target function, without considering child functions (milliseconds)

The profiling results show that the GestIT version requires 0.78% more inclusive time and 0.16% more exclusive time with respect to the overall application time, which can be considered a low impact. However, it is possible to notice a sensibly higher elapsed inclusive time and total number of calls for the functions contained in *BodyGestit.dll*. Therefore, it may be reasonable to optimize the code that translates the data coming from the Kinect sensor into a format manageable by the GestIT library.

Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Elapsed Inclusive Time	Elapsed Exclusive Time
CarViewer.exe	207615	100.00	0.07	99191.85	72.60
PresentationFramework.dll	657	100.00	89.98	99188.71	89257.71
FsmGestureRecognition.dll	24331	3.80	0.02	3758.85	2045
Microsoft.Kinect.Toolkit.Interaction.dll	116875	3.57	3.57	3542.77	3541.90
Microsoft.Kinect.dll	1320554	3.28	3.28	3253.83	3253.83
PresentationCore.dll	125643	2.06	2.06	2041.67	2041.67
Microsoft.Kinect.Toolkit.dll	24	0.65	0.65	646.81	646.81
mscorlib.dll	302814	0.34	0.34	333.60	333.60
3DTools.dll	325	0.02	0.02	17.76	17.76
WindowsBase.dll	142961	0.00	0.00	8.50	2.31
System.dll	4	0.00	0.00	3.21	3.21

Table 8.3 3D viewer CPU profiling (instrumentation, FSM version)

Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Elapsed Inclusive Time	Elapsed Exclusive Time
CarViewer.exe	204464	100.00	0.08	98.356.87	74.82
PresentationFramework.dll	449	100.00	89.33	98.353.59	87.864.40
BodyGestit.dll	119676	4.25	0.15	4179.38	151.93
Microsoft.Kinect.Toolkit.Interaction.dll	134141	3.75	3.75	3.684.37	3.683.89
Microsoft.Kinect.dll	3715440	3.38	3.38	3.325.18	3.325.18
PresentationCore.dll	122.804	2.25	2.25	2.215.43	2.215.43
Microsoft.Kinect.Toolkit.dll	24	0.66	0.66	649.62	649.62
mscorlib.dll	685086	0.36	0.36	352.09	352.09
Gestit.dll	174359	0.33	0.03	225.13	18.30
3DTools.dll	213	0.01	0.01	13.02	13.02
WindowsBase.dll	142423	0.00	0.00	3.23	3.23
System.dll	16	0.00	0.00	0.01	0.01

Table 8.4 3D viewer CPU profiling (instrumentation, GestIT version)

8.4.3 Memory

The memory profiler included in Visual Studio 2012 [93] provides information about the size and the number of objects created during the execution of the target function code.

We again grouped all the counters by DLL, and we report here the following data:

- *Inclusive allocations*: the number of allocations made in the target function and its children.
- *Exclusive allocations*: the number of allocations made in the target function, without considering its children.
- *Inclusive bytes*: the number of bytes allocated in the target function and its children.
- *Exclusive allocations*: the number of bytes allocations in the target function, without considering its children.

The comparison of the two profiling session data shows an increase of the allocation and bytes counters for the *BodyGestit.dll* with respect to the baseline implementation. Such increase is especially high for the exclusive allocation number and bytes (respectively 52% and 28%), confirming the need of an optimization in the library code that connects the Kinect sensor with the GestIT library. Instead, the part of the library that defines the temporal operators and the ground terms (the *Gestit.dll*) has a low impact on the memory consumption increase.

However, the amount of memory consumed by the FSM and GestIT DLLs are two orders of magnitude below the amount consumed by the Kinect sensor driver, which is the main responsible for the memory consumption in this application. This means again that the overall impact of using the GestIT library on whole application as a whole cannot be considered high.

Name	Inclusive Allocations	Exclusive Allocations	Inclusive Bytes	Exclusive Bytes
CarViewer.exe	3129945	148232	159320630	3285646
PresentationFramework.ni.dll	3129942	231986	159320424	19016158
FsmGestureRecognition.dll	1287576	5126	23208425	164104
Microsoft.Kinect.dll	1151146	1151146	23362566	23362566
PresentationCore.dll	951919	951919	97159963	97159963
Microsoft.Kinect.Toolkit.Interaction.dll	526203	521090	12891040	12727424
mscorlib.ni.dll	73514	73514	2365703	2365703
WindowsBase.ni.dll	41806	41070	1127672	1101122
Microsoft.Kinect.Toolkit.dll	21048	4740	1925101	78042
3DTools.dll	1735	651	46084	15828
clr.dll	468	468	44014	44014
System.ni.dll	3	3	60	60
PresentationFramework.Aero2.ni.dll	0	0	0	0
System.Core.ni.dll	0	0	0	0
System.Xaml.ni.dll	0	0	0	0
UIAutomationTypes.ni.dll	0	0	0	0

Table 8.5 3D viewer memory profiling (FSM version)

Name	Inclusive Allocations	Exclusive Allocations	Inclusive Bytes	Exclusive Bytes
CarViewer.exe	3424747	143.210	171514376	3204490
PresentationFramework.ni.dll	3424744	488085	171514178	33889952
BodyGestit.dll	1336355	7794	24872174	208982
Microsoft.Kinect.dll	1131988	1131988	22977390	22977390
PresentationCore.ni.dll	944591	944591	92900375	92900375
Microsoft.Kinect.Toolkit.Interaction.dll	517574	512485	12696600	12533752
mscorlib.ni.dll	144819	144819	4427685	4427685
WindowsBase.ni.dll	39389	38653	1063468	1036918
Gestit.dll	30427	55	941082	1812
Microsoft.Kinect.Toolkit.dll	20853	4155	1927916	68658
USER32.dll	8012	8012	208312	208312
3DTools.dll	1120	427	29988	10452
clr.dll	470	470	45538	45538
System.ni.dll	3	3	60	60
PresentationFramework.Aero2.ni.dll	0	0	0	0
System.Core.ni.dll	0	0	0	0
System.Xaml.ni.dll	0	0	0	0
UIAutomationTypes.ni.dll	0	0	0	0

Table 8.6 3D viewer memory profiling (GestIT version)

8.5 Summary

In this chapter, we provided an evaluation of the proposed gesture modelling approach, which consisted of three different sets of inspection criteria.

We first assessed the requirements we identified for the development of gestural interfaces, showing that we advanced the state of the art, providing a meta-model able to describe the temporal evolution of gestures in a reusable and compositional way. We support different gesture recognition devices, defining a programming model that can be instantiated even to other recognition supports that are not covered by our work.

The second set of criteria has been defined in [97], reviewing different tools that have been used for defining UIs and the reasons behind their success or failure. We showed that our modelling technique did not repeated well-known errors, even if some aspects will need further investigation (e.g. how difficult is to learn our modelling technique from scratch).

The third set of criteria is represented by the Cognitive Dimension Framework, defined in [51]. The analysis highlighted the need for a clearer representation of the dependencies among the different predicates associated to the ground terms. In addition, if a gesture model is created programmatically (i.e. without using XML or another declarative notation), it is more likely to have modelling errors, since the composition aspect is not explicit in the notation. Therefore, it may be reasonable to provide a graphical notation that solves these problems in the future.

Finally, we provided a preliminary analysis of the GestIT library performance, comparing two versions of the 3D viewer application. The first one was implemented with a simple FSM and represents the baseline for the application performance. The second version defined the gestural interaction through the GestIT notation.

The comparison results show that the GestIT version requires a low increase of the CPU usage and a sensible increase of the memory consumption. Provided that the main responsible for the increment is the part connecting the Kinect sensor the GestIT temporal operators, the optimization work should start from that part of the library.

Chapter 9

Conclusion

The lack of proper programming models for defining gestures is a major issue in defining gesture-based interfaces and it limits significantly the ability to fully exploit the new multitouch and 3D input devices, now becoming widely available. The observer pattern underlying the traditional event-based programming is largely inadequate for tracking gestures made of multiple inputs over time, forcing the programmer to choose between handling the complexity of this process or picking one of a pre-defined gestures recognized by the framework used.

In this thesis, we proposed GestIT, a declarative, compositional meta-model for defining gestures, addressing this key issue and allowing for simultaneous recognition of multiple gestures and sub-gestures under control of the programmer rather than the framework. The meta-model elements contain ground terms and composition operators that have been theoretically defined using Non Autonomous Petri Nets.

It allows reusing and composing the definition of gestures in different applications, providing the possibility to define UI reactions for the recognition not only for the entire gesture, but also for its sub-components.

The declarative and compositional approach proposed in this thesis for gesture definition solves the single-event granularity problem and provides a separation of concerns (the temporal sequence definition is separated from the behaviour), which allows a more understandable and maintainable code. In addition, we discussed the selection ambiguity problem, which affects the composition of gestures that have a common prefix through a choice operator. The recognition support has different possibilities for dealing with the uncertainty in the selection while performing this common prefix. We discussed different solutions and we adopted the compensation approach in GestIT.

Moreover, we reported on a proof-of-concept library, which has been exploited for managing two different gesture recognition supports (multitouch and full-body), showing the flexibility and the generality of the approach. We developed different sample applications for demonstrating the advantages of the proposed modelling technique in reusing gesture definitions, which can be exploited at the desired level of granularity.

Finally we extended MARIA [111], a state of the art User Interface Description Language, providing it with a concrete user interface model that is able to exploit gestural interaction, according to the proposed meta-model.

9.1 Future work

The work discussed in this thesis can be extended in different directions. The first one is the most obvious: we did not cover the entire set of devices that can be used for recognizing gestures. Adding both the formal modelling and the library support for existing devices, such as remotes and floor boards, or new ones such as the Leap Motion or the new version of Microsoft Kinect, may enforce the validity of the proposed modelling technique and also provide the source for enhancing the model with other features.

We already started this work, through an optimized implementation of the modelling technique that supports web applications (through javascript) and that will be ready for supporting commercial and production-level applications. In addition, such more engineered version of the library will provide a way for creating personalised combination of sensors providing a way for defining new ground terms in a simple way, in order to increase the flexibility of the approach. The implementation of this new version will be open source and available at <http://gestit.github.io/GestIT/>.

Another research direction is the investigation of the impact that such modelling techniques may have on tools and authoring environments for creating gestural interaction. The compositional approach may be exploited for creating a sort of workflow visualization that can be interactively explored for analysing the defined interaction.

A declarative description can be also exploited for describing not only the interaction, but also for estimating different parameters connected to gesture performance. For instance, it is possible to define a cost model based on the composition of ground terms and complex gestures. The cost model may predict different types of efforts that users put in gesture performance: from physical (which may assess ergonomics aspects) to cognitive. An

effective prediction based on the declarative definition may be successfully exploited in both UI design and analysis tools.

In addition, the gesture modelling can be applied for emotion analysis, defining a set of gestures or postures that communicate implicit information on the user's emotional state.

Last but not least, the main future direction that we foresee for this work is its refinement and application in UI toolkits for both desktop and mobile devices. In the future, we believe that this kind of interaction will be embedded in different everyday use devices, such as televisions, home appliances etc., enhancing their interaction possibilities. We think that the proposed approach can have a role on the creation of the future UI toolkits, that will host such kind of interaction as a first-class citizen, as happened for instance for the animation support.

In order to do that, we think that it is necessary to apply the modelling technique in a large-scale application scenario, in order to test it outside the research environment and inside an industrial setting. The feedback provided by such kind of application may benefit both the industry, which will exploit a more efficient and effective way for creating gestural interaction, but also the research per se, since it may provide an engineering pattern that can be applied for all the different continuous input sources we use for interacting with computers.

Beyond further enhancements of the meta-model and a more deep evaluation of the proposed approach, it would be interesting to investigate if our approach provides advantages not only for developers but also for end-users. Our hypothesis is that providing a way for reuse existing gesture definitions encourages developers in reapplying tested definitions against naïve implementations that may be incomplete. In addition, this promotes the adoption of commonly-used gestures for similar functionalities, which may have a positive influence on the overall gesture interface usability

Bibliography

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M., and Shuster, J.E. UIML: an appliance-independent XML user interface language. *Computer Networks* 31, 11-16 (1999), 1695–1708.
2. Accot, J., Chatty, S., and Palanque, P.A. A Formal Description of Low Level Interaction and its Application to Multimodal Interactive Systems. *DSV-IS*, Springer (1996), 92–104.
3. Appert, C. and Beaudouin-Lafon, M. SwingStates: adding state machines to the swing toolkit. *Proceedings of the 19th annual ACM symposium on User interface software and technology*, ACM (2006), 319–322.
4. Apple. iPhone. Available online: <http://www.apple.com/iphone/>. (Accessed: 27-May-2013).
5. Apple. Create Apps for iOS. Available online: <https://developer.apple.com/devcenter/ios/checklist/>. (Accessed: 02-Apr-2013).
6. Apple. UIKit Reference. Available online: http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIKit_Framework/index.html. (Accessed: 02-Apr-2013).
7. Arnaud, H., Palanque, P., Silva, J.L., Deleris, Y., and Navarre, D. Formal Description of Multi-Touch Interactions. *Fifth ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM Press (2013).

8. ASUS. Xtion PRO. Available online: http://www.asus.com/Multimedia/Xtion_PRO/. (Accessed: 28-Mar-2013).
9. Augsten, T., Kaefer, K., Meusel, R., et al. Multitoe: high-precision interaction with back-projected floors based on high-resolution multi-touch input. *UIST*, (2010), 209–218.
10. Bau, O. and Mackay, W.E. OctoPocus: a dynamic guide for learning gesture-based command sets. *Proceedings of the 21st annual ACM symposium on User interface software and technology*, ACM (2008), 37–46.
11. Den Bergh, J., Luyten, K., and Coninx, K. CAP3: context-sensitive abstract user interface specification. *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, ACM (2011), 31–40.
12. Den Bergh, M., Carton, D., de Nijs, R., et al. Real-time 3D hand gesture interaction with a robot for understanding directions from humans. *RO-MAN, 2011 IEEE*, (2011), 357–362.
13. Blackwell, A.F. and Green, T.R.G. A Cognitive Dimensions Questionnaire Optimised for Users. *12th Workshop of the Psychology of Programming Interest Group*, (2000), 137–152.
14. Blanch, R. and Beaudouin-Lafon, M. Programming rich interactions using the hierarchical state machine toolkit. *Proceedings of the working conference on Advanced visual interfaces*, ACM (2006), 51–58.
15. Bleser, T. and Foley, J.D. Towards specifying and evaluating the human factors of user-computer interfaces. *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, ACM (1982), 309–314.
16. Bo, H., Bing-yi, Z., Fang, Z., and Ya-min, S. Modeling multimodal integration based on colored Petri nets and feature structures.

- Control, Automation, Robotics and Vision Conference, 2004. ICARCV 2004 8th*, (2004), 514–516 Vol. 1.
17. Bobick, A.F. and Wilson, A.D. A state-based approach to the representation and recognition of gesture. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 19, 12 (2002), 1325–1337.
 18. Bolognesi, T. and Brinksma, E. Introduction to the ISO specification language LOTOS. *Computer Networks and Systems* 14, 1 (1987), 25–59.
 19. Bongartz, S., Jin, Y., Paternò, F., Rett, J., Santoro, C., and Spano, L. Adaptive User Interfaces for Smart Environments with the Support of Model-Based Languages. In F. Paternò, B. Ruyter, P. Markopoulos, C. Santoro, E. Loenen and K. Luyten, eds., *Ambient Intelligence*. Springer Berlin Heidelberg, 2012, 33–48.
 20. Boyer, J.M. XForms 1.1. Available online: <http://www.w3.org/TR/2009/REC-xforms-20091020/>. (Accessed: 02-Mar-2013).
 21. Bradski, G. and Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Incorporated, 2008.
 22. Bragdon, A., DeLine, R., Hinckley, K., and Morris, M.R. Code space: touch + air gesture hybrid interactions for supporting developer meetings. *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ACM (2011), 212–221.
 23. Bränzel, A., Holz, C., Hoffmann, D., et al. GravitySpace: tracking users and their poses in a smart room using a pressure-sensing floor. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2013), 725–734.
 24. Broccia, G., Livesu, M., and Scateni, R. Gestural Interaction for Robot Motion Control. *Eurographics Italian Chapter Conference*, (2011), 61–66.

25. Buxton, B. Multi-Touch Systems that I Have Known and Loved. Available online: <http://www.billbuxton.com/multitouchOverview.html>. (Accessed: 01-Jun-2013).
26. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A unifying reference framework for multi-target user interfaces. *Interacting with Computers* 15, 3 (2003), 289–308.
27. Calvary, G., Coutaz, J., Thevenin, D., et al. The CAMELEON reference framework. *Deliverable D 1*, (2002).
28. Calvary, G. and Demeure, A. Context-aware and mobile interactive systems: the future of user interfaces plasticity. *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, ACM (2009), 243–244.
29. Cam-Trax. CamSpace. Available online: <http://www.camspace.com/>. (Accessed: 27-May-2013).
30. Card, S.K., Moran, T.P., and Newell, A. *The psychology of human-computer interaction*. CRC, 1983.
31. Carpenter, R.L. *The logic of typed feature structures: with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press, 2005.
32. Chen, M., Mountford, S.J., and Sellen, A. A study in interactive 3-D rotation using 2-D control devices. *SIGGRAPH Comput. Graph.* 22, 4 (1988), 121–129.
33. Cheng, L., Sun, Q., Su, H., Cong, Y., and Zhao, S. Design and implementation of human-robot interactive demonstration system based on Kinect. *Control and Decision Conference (CCDC), 2012 24th Chinese*, (2012), 971–975.
34. Colombo, C. and Pace, G. Long Running Transaction. *ACM Computing Surveys* 4, 3 (2013), (accepted paper).

35. Cuccurullo, S., Francese, R., Murad, S., Passero, I., and Tucci, M. A gestural approach to presentation exploiting motion capture metaphors. *Proceedings of the International Working Conference on Advanced Visual Interfaces*, ACM (2012), 148–155.
36. David, R. and Alla, H. *Discrete, continuous, and hybrid Petri nets*. Springer, 2010.
37. Dietz, P. and Leigh, D. DiamondTouch: a multi-user touch technology. *Proceedings of the 14th annual ACM symposium on User interface software and technology*, (2001), 219–226.
38. Dillon, R., Wong, G., and Ang, R. Virtual Orchestra: an immersive computer game for fun and education. *Proceedings of the 2006 international conference on Game research and development*, (2006), 215–218.
39. Echtler, F. and Butz, A. GISpL: gestures made easy. *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction*, ACM (2012), 233–240.
40. Farley, H. and Steel, C. A quest for the Holy Grail: Tactile precision, natural movement and haptic feedback in 3D virtual spaces. *Same places, different spaces*, (2009), 285.
41. Foley, J. and Sukaviriya, P. History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. *Proceedings of DSV-IS*, 3–14.
42. Forlines, C., Wigdor, D., Shen, C., and Balakrishnan, R. Direct-touch vs. mouse input for tabletop displays. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2007), 647–656.
43. Franke, T., Olbrich, M., and Fellner, D.W. A flexible approach to gesture recognition and interaction in X3D. *Proceedings of the 17th International Conference on 3D Web Technology*, ACM (2012), 171–174.

44. Gabbay, D. *Labelled deductive systems and situation theory*. 1993.
45. Gallo, L., Placitelli, A.P., and Ciampi, M. Controller-free exploration of medical image data: Experiencing the Kinect. *Computer-Based Medical Systems (CBMS), 2011 24th International Symposium on*, (2011), 1–6.
46. García Frey, A., Céret, E., Dupuy-Chessa, S., Calvary, G., and Gabillon, Y. UsiComp: an extensible model-driven composer. *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, ACM (2012), 263–268.
47. Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Modeling long-running activities as nested sagas. *Data Eng.* 14, 1 (1991), 14–18.
48. Garzotto, F. and Valoriani, M. “Don’t touch the oven”: motion-based touchless interaction with household appliances. *Proceedings of the International Working Conference on Advanced Visual Interfaces*, ACM (2012), 721–724.
49. Gil-Gomez, J.-A., Lozano, J.-A., Alcaniz, M., and Perez, S.A. Nintendo Wii Balance board for balance disorders. *Virtual Rehabilitation International Conference, 2009*, (2009), 213.
50. Gorg, M.T., Cebulla, M., and Garzon, S.R. A Framework for Abstract Representation and Recognition of Gestures in Multi-touch Applications. *Advances in Computer-Human Interactions, 2010. ACHI '10. Third International Conference on*, (2010), 143–147.
51. Green, T.R.G. and Petre, M. Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.
52. Gyration. Gyration in-air mouse. Available online: <http://www.gyration.com/>. (Accessed: 27-May-2013).
53. De Haan, G., Griffith, E.J., and Post, F.H. Using the Wii Balance Board™ as a low-cost VR interaction device. *Proceedings of*

- the 2008 ACM symposium on Virtual reality software and technology*, ACM (2008), 289–290.
54. Han, J.Y. Low-cost multi-touch sensing through frustrated total internal reflection. *Proceedings of the 18th annual ACM symposium on User interface software and technology*, (2005), 115–118.
 55. Helms, J. and Abrams, M. Retrospective on UI description languages, based on eight years' experience with the User Interface Markup Language (UIML). *International Journal of Web Engineering and Technology* 4, 2 (2008), 138–162.
 56. Henry, T.R., Hudson, S.E., and Newell, G.L. Integrating gesture and snapping into a user interface toolkit. *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, ACM (1990), 112–122.
 57. Hinckley, K., Czerwinski, M., and Sinclair, M. Interaction and modeling techniques for desktop two-handed input. *Proceedings of the 11th annual ACM symposium on User interface software and technology*, ACM (1998), 49–58.
 58. Hirte, S., Seifert, A., Baumann, S., Klan, D., and Sattler, K.-U. Data3 -- A Kinect Interface for OLAP Using Complex Event Processing. *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, (2012), 1297–1300.
 59. Hoste, L., Dumas, B., and Signer, B. Mudra: a unified multimodal interaction framework. *Proceedings of the 13th international conference on multimodal interfaces*, ACM (2011), 97–104.
 60. Isard, M. and Blake, A. Contour tracking by stochastic propagation of conditional density. *Computer Vision-ECCV'96*, (1996), 343–356.
 61. Jacob, R.J.K., Deligiannidis, L., and Morrison, S. A software model and specification language for non-WIMP user interfaces. *ACM Trans. Comput.-Hum. Interact.* 6, 1 (1999), 1–46.
 62. Jacob, R.J.K. Eye Movement-Based Human-Computer Interaction Techniques: Toward Non-Command Interfaces. *IN ADVANCES IN*

- HUMAN-COMPUTER INTERACTION*, Ablex Publishing Co (1993), 151–190.
63. Johnson, P., Wilson, S., Markopoulos, P., and Pycock, J. ADEPT: Advanced design environment for prototyping with task models. *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, (1993), 56.
 64. Jota, R., Nacenta, M.A., Jorge, J.A., Carpendale, S., and Greenberg, S. A comparison of ray pointing techniques for very large displays. *Proceedings of Graphics Interface 2010*, Canadian Information Processing Society (2010), 269–276.
 65. Juul, J. *A Casual Revolution: Reinventing Video Games and Their Players*. The MIT Press, 2009.
 66. Kammer, D., Wojdziak, J., Keck, M., Groh, R., and Taranko, S. Towards a formalization of multi-touch gestures. *ACM International Conference on Interactive Tabletops and Surfaces*, ACM (2010), 49–58.
 67. Kang, J., Seo, D., and Jung, D. A Study on the control Method of 3-Dimensional Space Application using KINECT System. *International Journal of Computer Science and Network Security* 11, 9 (2011), 55–59.
 68. Kasday, L.R. Touch position sensitive surface. *U.S. Patent*, (1984).
 69. Kass, M., Witkin, A., and Terzopoulos, D. Snakes: Active contour models. *International journal of computer vision* 1, 4 (1988), 321–331.
 70. Kerber, F., Lessel, P., Daiber, F., and Krüger, A. Shift ‘n’ touch: combining Wii Balance Board and Cubtile. *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*, ACM (2012), 789–790.
 71. Khandkar, S.H. and Maurer, F. A domain specific language to define gestures for multi-touch applications. *Proceedings of the 10th Workshop on Domain-Specific Modeling*, ACM (2010), 2:1–2:6.

72. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: multitouch gestures as regular expressions. *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems (CHI 2012)*, ACM Press (2012), 2885–2894.
73. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++ : A Customizable Declarative Multitouch Framework. *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST 2012)*, ACM Press (2012), 477–486.
74. Kistler, F., Sollfrank, D., Bee, N., and André, E. Full Body Gestures Enhancing a Game Book for Interactive Story Telling. In M. Si, D. Thue, E. André, J. Lester, J. Tanenbaum and V. Zammitto, eds., *Interactive Storytelling*. Springer Berlin / Heidelberg, 2011, 207–218.
75. Klan, D., Hose, K., Karnstedt, M., and Sattler, K.-U. Power-aware data analysis in sensor networks. *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, (2010), 1125–1128.
76. Kortum, P. *HCI Beyond the GUI: Design for Haptic, Speech, Olfactory, and Other Nontraditional Interfaces (Interactive Technologies)*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2008.
77. De la Barré, R., Chojecki, P., Leiner, U., Mühlbach, L., and Ruschin, D. Touchless interaction—novel chances and challenges. In *Human-Computer Interaction. Novel Interaction Methods and Techniques*. Springer, 2009, 161–169.
78. Iacolina, S.A., Soro, A., and Scateni, R. Natural exploration of 3D models. *Proceedings of the 9th ACM SIGCHI Italian Chapter International Conference on Computer-Human Interaction: Facing Complexity*, ACM (2011), 118–121.
79. Lai, K., Konrad, J., and Ishwar, P. A gesture-driven computer interface using Kinect. *Image Analysis and Interpretation (SSIAI), 2012 IEEE Southwest Symposium on*, (2012), 185–188.

80. Leap Motion Inc. Leap Motion. Available online: <https://www.leapmotion.com/>. (Accessed: 02-Jun-2013).
81. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. Usixml: A language supporting multi-path development of user interfaces. *Engineering Human Computer Interaction and Interactive Systems*, (2005), 200–220.
82. Lyons, K., Brashear, H., Westeyn, T., Kim, J.S., and Starner, T. GART: the gesture and activity recognition toolkit. In *Human-Computer Interaction. HCI Intelligent Multimodal Interaction Environments*. Springer, 2007, 718–727.
83. Lyons, M.J., Budynek, J., and Akamatsu, S. Automatic classification of single facial images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 21*, 12 (2002), 1357–1362.
84. Maskell, S. and Gordon, N. A tutorial on particle filters for on-line nonlinear/non-Gaussian Bayesian tracking. *IEE Seminar Digests*, (2001).
85. Microsoft, M. Partial Class Definitions (C# Programming Guide). Available online: [http://msdn.microsoft.com/en-us/library/wa80x488\(v=VS.80\).aspx](http://msdn.microsoft.com/en-us/library/wa80x488(v=VS.80).aspx). (Accessed: 27-Apr-2013).
86. Microsoft. PixelSense. Available online: <http://www.microsoft.com/en-us/pixelsense/default.aspx>. (Accessed: 27-May-2013).
87. Microsoft. Kinect for Windows. Available online: <http://www.microsoft.com/en-us/kinectforwindows/>. (Accessed: 28-Mar-2013).
88. Microsoft. Delegate. Available online: [http://msdn.microsoft.com/en-us/library/900fyy8e\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/900fyy8e(v=vs.71).aspx). (Accessed: 29-Mar-2013).
89. Microsoft. XAML Overview. Available online: <http://msdn.microsoft.com/en-us/library/ms752059.aspx>. (Accessed: 02-Apr-2013).

90. Microsoft. Kinect Studio. Available online: <http://msdn.microsoft.com/en-us/library/hh855389.aspx>. (Accessed: 11-Jul-2013).
91. Microsoft. Understanding Sampling Data Values in Profiling Tools. Available online: <http://msdn.microsoft.com/it-it/library/ms242753.aspx>. (Accessed: 11-Jul-2013).
92. Microsoft. Understanding Instrumentation Data Values in Profiling Tools. Available online: <http://msdn.microsoft.com/it-it/library/ms182369.aspx>. (Accessed: 11-Jul-2013).
93. Microsoft. Understanding Memory Allocation and Object Lifetime Data Values in Profiling Tools. Available online: <http://msdn.microsoft.com/library/dd264966.aspx>.
94. Microsoft. Visual Studio 2012. Available online: <http://msdn.microsoft.com/it-it/vstudio/bb984878.aspx>. (Accessed: 11-Jul-2013).
95. Mitra, S. and Acharya, T. Gesture recognition: A survey. *IEEE Transactions on Systems, Man and Cybernetics - PART C* 37, 3 (2007), 311–324.
96. Mori, G., Paterno, F., and Santoro, C. Design and development of multidevice user interfaces through multiple logical descriptions. *Software Engineering, IEEE Transactions on* 30, 8 (2004), 507–520.
97. Myers, B., Hudson, S.E., and Pausch, R. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (2000), 3–28.
98. Myers, B.A. A new model for handling input. *ACM Trans. Inf. Syst.* 8, 3 (1990), 289–320.
99. Navarre, D., Palanque, P., Ladry, J.-F., and Barboni, E. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.* 16, 4 (2009), 18:1–18:56.

100. Nielsen, J. Heuristic evaluation. *Usability inspection methods 17*, (1994), 25–62.
101. Nikolaidis, A. and Pitas, I. Facial feature extraction and pose determination. *Pattern Recognition 33*, 11 (2000), 1783–1791.
102. Nintendo. Nintendo Wii. Available online: <http://www.nintendo.com/wii>. (Accessed: 27-May-2013).
103. Olsen Jr., D.R. and Dempsey, E.P. SYNGRAPH: A graphical user interface generator. *SIGGRAPH Comput. Graph. 17*, 3 (1983), 43–50.
104. Olsen Jr., D.R. MIKE: the menu interaction kontrol environment. *ACM Trans. Graph. 5*, 4 (1986), 318–344.
105. Olsen Jr., D.R. Evaluating User Interface Systems Research. *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, ACM (2007), 251–258.
106. OSGi Alliance. OSGi Service Platform Release 4. Available online: <http://www.osgi.org/Main/HomePage>. (Accessed: 02-Jun-2013).
107. Palanque, P.A., Bastide, R., and Sengès, V. Validating interactive system design through the verification of formal task and system models. *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, Chapman & Hall, Ltd. (1996), 189–212.
108. Panger, G. Kinect in the kitchen: testing depth camera interactions in practical home environments. *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts*, ACM (2012), 1985–1990.
109. Paternò, F., Santoro, C., Spano, L.D., and Ragget, D. (eds). MBUI-Task Models. Available online: <http://www.w3.org/TR/2012/WD-task-models-20120802/>. (Accessed: 27-May-2013).

110. Paterno, F., Santoro, C., and Spano, L.D. The role of HCI models in service front-end development. *Behaviour & Information Technology* 31, 3 (2012), 231–244.
111. Paternò, F., Santoro, C., and Spano, L.D. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transaction on Computer Human Interaction* 16, 4 (2009), 19:1–19:30.
112. Paternò, F., Santoro, C., and Spano, L.D. Exploiting web service annotations in model-based user interface development. *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, (2010), 219–224.
113. Paternò, F., Santoro, C., and Spano, L.D. Engineering the authoring of usable service front ends. *J. Syst. Softw.* 84, 10 (2011), 1806–1822.
114. Paternò, F. *Model-based design and evaluation of interactive applications*. Springer Verlag, 2000.
115. PrimeSense. NITE Middleware. Available online: <http://www.primesense.com/solutions/nite-middleware/>. (Accessed: 04-Feb-2013).
116. Puerta, A. and Eisenstein, J. XIML: A universal language for user interfaces. *White paper*, (2001).
117. Rich, C. Building task-based user interfaces with ANSI/CEA-2018. *Computer* 42, 8 (2009), 20–27.
118. Sangsuriyachot, N., Mi, H., and Sugimoto, M. Novel interaction techniques by combining hand and foot gestures on tabletop environments. *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ACM (2011), 268–269.
119. Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, ACM (2011), 49–56.

120. Schwarz, J., Hudson, S., Mankoff, J., and Wilson, A.D. A framework for robust and flexible handling of inputs with uncertainty. *Proceedings of the 23rd annual ACM symposium on User interface software and technology*, ACM (2010), 47–56.
121. Serna, A., Calvary, G., and Scapin, D.L. How assessing plasticity design choices can improve UI quality: a case study. *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, ACM (2010), 29–34.
122. Shotton, J., Fitzgibbon, A., Cook, M., et al. *Real-time human pose recognition in parts from single depth images*. IEEE, 2011.
123. Sony. PlayStation Move. Available online: <http://iplaystation.com/psmove/>. (Accessed: 27-May-2013).
124. Spano, L.D., Cisternino, A., Fabio, P., and Fenu, G. A Declarative and Compositional Framework for Multiplatform Gesture Definition. *EICS 2013, 5th Symposium on Engineering Interactive Computing Systems*, ACM Press (2013).
125. Spano, L.D., Cisternino, A., and Paternò, F. A Compositional Model for Gesture Definition. *Proceedings of the 4th International Conference in Human-Centered Software Engineering (HCSE 2012)*, LNCS, Springer (2012), 34–52.
126. Spano, L.D. A model-based approach for gesture interfaces. *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, ACM (2011), 327–330.
127. Spano, L.D. Developing Touchless Interfaces with GestIT. In F. Paternò, B. de Ruyter, P. Markopoulos, C. Santoro, E. van Loenen and K. Luyten, eds., *Ambient Intelligence*. Springer Berlin / Heidelberg, 2012, 433–438.
128. Suma, E.A., Lange, B., Rizzo, A.S., Krum, D.M., and Bolas, M. FFAST: The Flexible Action and Articulated Skeleton Toolkit. *Virtual Reality Conference (VR), 2011 IEEE*, (2011), 247–248.

129. Szekely, P., Luo, P., and Neches, R. Beyond interface builders: model-based interface tools. *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, ACM (1993), 383–390.
130. Tan, C.S.S., Schöning, J., Barnes, J.S., Luyten, K., and Coninx, K. Bro-cam: Improving game experience with empathic feedback using posture tracking. In *Persuasive Technology*. Springer, 2013, 222–233.
131. Tan, C.S.S., Schöning, J., Luyten, K., and Coninx, K. Informing intelligent user interfaces by inferring affective states from body postures in ubiquitous computing environments. *Proceedings of the 2013 international conference on Intelligent user interfaces*, ACM (2013), 235–246.
132. Turk, M. and Pentland, A. Eigenfaces for recognition. *Journal of cognitive neuroscience* 3, 1 (1991), 71–86.
133. UsiXML Consortium. UsiXML ITEA 2 project. Available online: <http://www.usixml.eu/about-the-project>. (Accessed: 02-Jun-2013).
134. Van Der Veer, G.C., Lenting, B.F., and Bergevoet, B.A.J. GTA: Groupware task analysis--Modeling complexity. *Acta Psychologica* 91, 3 (1996), 297–322.
135. Vultur, O.M., Pentiuc, S.G., and Ciupu, A. Navigation system in a virtual environment by gestures. *Communications (COMM), 2012 9th International Conference on*, (2012), 111–114.
136. W3C. XSL Transformations (XSLT) Version 2.0. Available online: <http://www.w3.org/TR/xslt20/>. (Accessed: 26-Apr-2013).
137. Wagner, K. Xbox One: Everything You Need to Know About. Available online: <http://gizmodo.com/the-new-xbox-everything-you-need-to-know-about-microso-509033619>. (Accessed: 02-Jun-2013).
138. Webb, J. and Ashley, J. *Beginning Kinect Programming with the Microsoft Kinect SDK*. Apress, 2012.

139. Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S. ITS: a tool for rapidly developing interactive applications. *ACM Trans. Inf. Syst.* 8, 3 (1990), 204–236.
140. Wobbrock, J., Morris, M., and Wilson, A. User-defined gestures for surface computing. ... *on Human factors in computing ...*, (2009), 1083.
141. Wolfgang, P. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.
142. Wroblewski, L. Touch Gesture Reference Guide. .
143. Yamato, J., Ohya, J., and Ishii, K. Recognizing human action in time-sequential images using hidden Markov model. *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR 92., 1992 IEEE Computer Society Conference on*, (2002), 379–385.
144. Yang, M.H. and Ahuja, N. Recognizing hand gesture using motion trajectories. *cvpr*, (1999), 1466.
145. Young, W., Ferguson, S., Brault, S., and Craig, C. Assessing and training standing balance in older adults: A novel approach using the ‘Nintendo Wii’ Balance Board. *Gait & Posture* 33, 2 (2011), 303–305.
146. Zanden, B. Vander and Myers, B.A. Automatic, look-and-feel independent dialog creation for graphical user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, ACM (1990), 27–34.
147. Zhu, Y., Dariush, B., and Fujimura, K. Controlled human pose estimation from depth image streams. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW’08. IEEE Computer Society Conference on*, (2008), 1–8.
148. Zimmerman, T.G., Lanier, J., Blanchard, C., Bryson, S., and Harvill, Y. A hand gesture interface device. *SIGCHI Bull.* 17, SI (1986), 189–192.

