

Making Tabletops Useful with Applications, Frameworks and Multi-Tasking

Carles F. Julià

TESI DOCTORAL UPF / 2014

Dirigida per:

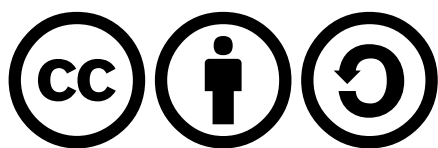
Dr. Sergi Jordà

Departament de Tecnologies de la Informació i les Comunicacions



Universitat
Pompeu Fabra
Barcelona

© 2014 Carles F. Julià



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>.

Als companys de viatge.

Acknowledgements

It's difficult to list all the people that contributed to this thesis. A thesis is not only the result of the work of the author, but also of the many people that contributed, directly or indirectly, to this research culmination.

I'd like to thank the Music Technology Group and its director Xavier Serra for supporting my research. Also and specially my thesis director, Sergi Jordà, for allowing me to choose my path and supporting my decisions along the journey of this PhD, and for always being open to start a powerful discussion and to share his thoughts. I also thank Emilia Gómez and Alba B. Rosado for involving me in the PHENICX project.

This thesis includes work of which I was not the sole author and of which I must very much thank many people for: Daniel Gallardo shared with me many of the presented projects and his contribution has been essential to them; TDesktop, TurTan, ofxTableGestures and MTCF can't be explained without him. Milena Markova was responsible for running the TurTan experiments, Miguel Lechón performed tests on the SongExplorer, and Nicolás Earnshaw tested GestureAgents, as part of their Master Thesis. Also, Marc Colomer restored and re-documented many of the TSI applications that are later presented here. Many students and projects from various courses have contributed to this work, using the frameworks or creating applications. In particular students from the TSI and CDSIM courses, undergraduate final projects from computer science and media students, and master thesis from SMC and CSIM students.

At a personal level, I also want to thank my family, for their constant support; Clara, for her sincere companionship and permanent aid over many years and, of course, her valuable English proof-reading; the current and former MAIn people (Dani, Sebastian, Sergi, Mathieu, Gianmaria, Yuya, Joana, Nuno, Marc, Alex); Alba, Cristina and Sònia; my present and former MTG colleagues; and Lydia and all the university staff.

This work is partially supported by the Spanish Ministry of Tourism and Commerce (MITYC) through the Classical Planet project (Plan Avanza Grant agreement No TSI-070100-2009-407) and by the European Union 7th Framework Programme through the PHENICX project (grant agreement no. 601166)

Abstract

The progressive appearance of affordable tabletop technology and devices urges human-computer interaction researchers to provide the necessary methods to make this kind of devices the most useful to their users. Studies show that tabletops have distinctive characteristics that can be specially useful to solve some types of problems, but this potential is arguably not yet translated into real-world applications. We theorize that the important components that can transform those systems into useful tools are application frameworks that take into account the devices affordances, a third party application ecosystem, and multi-application systems supporting concurrent multitasking.

In this dissertation we approach these key components: First, we explore the distinctive affordances of tabletops, with two cases: *TurTan*, a tangible programming language in the education context, and *SongExplorer*, a music collection browser for large databases.

Next, in order to address the difficulty of building such applications in a way that they can exploit these affordances, we focus on software frameworks to support the tabletop application making process, with two different approaches: *ofxTableGestures*, targeting programmers, and *MTCF*, designed for music and sound artists.

Finally, recognizing that making useful applications is just one part of the problem, we focus on a fundamental issue of multi-application tabletop systems: the difficulty to support multi-user concurrent multitasking with third-party applications. After analyzing the possible approaches, we present *GestureAgents*, a content-based distributed application-centric disambiguation mechanism and its implementation, which solves this problem in a generic fashion, being also useful to other shareable interfaces, including uncoupled ones.

Resum

L'aparició progressiva de tecnologia i dispositius *tabletop* barats urgeix a la comunitat de recerca en interacció persona-ordinador a proveir els mètodes necessaris per transformar aquests dispositius en eines realment útils pels usuaris. Diversos estudis indiquen que els *tabletops* tenen algunes característiques peculiars que poden ser especialment útils per solucionar algun tipus de problemes, però tanmateix sembla que el seu potencial encara no arriba a transformar-se en aplicacions reals. Creiem que els components importants per a transformar aquests sistemes en eines útils són *frameworks* d'aplicació que tinguin en compte les capacitats dels dispositius, un ecosistema d'aplicacions fetes per desenvolupadors independents, i sistemes multi-aplicació amb suport per multitasca concurrent.

En aquesta tesi doctoral ens aproximem a aquests components clau: En primer lloc, explorem les capacitats dels *tabletops*, usant dos casos: *TurTan*, un llenguatge de programació tangible en el context educatiu, i *SongExplorer*, un navegador de col·leccions musicals per a grans bases de dades.

A continuació, amb ànim d'abordar la complexitat a l'hora de crear aquest tipus d'aplicacions de tal manera que usin aquestes capacitats, ens centrem en els *frameworks* de programari per donar suport en el procés de creació d'aplicacions *tabletop*, amb dues aproximacions diferents: *ofxTableGestures*, dirigit a programadors, i *MTCF*, dissenyat per a artistes musicals i del so.

Finalment, reconeixent que crear aplicacions útils és només part del problema, ens centrem en una qüestió fonamental dels sistemes multi-aplicació: la dificultat d'acceptar la interacció multitasca de forma concurrent i multi-usuari amb aplicacions externes. Després d'analitzar-ne les possibles aproximacions, presentem *GestureAgents*, un mecanisme de desambiguació (i la seva implementació corresponent) basat en el contingut, distribuït i centrat en les aplicacions, que soluciona aquest problema d'una forma genèrica, esdevenint útil també per altres interfícies compatibles, incloses les desacoblades.

Contents

1	Motivation	1
1.1	Learning from mistakes	1
1.2	A personal introduction to the thesis	3
1.3	Contributions of this thesis	3
1.4	Structure of this document	4
2	Introduction	7
2.1	The usefulness of the personal computer	7
2.2	Interaction in personal computers	8
2.2.1	WIMP	9
2.2.2	Direct Manipulation	10
2.2.3	Multi-Tasking and third-party applications	11
2.2.4	Interaction Foci	12
2.2.5	Post-Wimp GUI	14
2.3	Gestural Interaction	15
2.4	Tangible Interaction	17
2.4.1	Tangible and tabletop	20
2.5	Making tabletops useful	23
3	Exploring tabletops' distinctive affordances	27
3.1	Affordances of tabletops	27
3.1.1	Simultaneous input	27
3.1.2	Collaboration	28
3.1.3	Physicality	30
3.2	Existing tabletop applications	32
3.3	The presented applications	36
3.4	Hardware Setup	36
3.5	TurTan, a tangible tabletop programming language for education	40
3.5.1	TUI in learning	41

3.5.2	Tangible programming languages	42
3.5.3	TurTan interaction mechanics	43
3.5.4	TurTan language, as an example of a Tangible Programming Lan- guage	46
3.5.5	Is TurTan better than Logo in an education context?	50
3.6	SongExplorer, a tabletop for song database exploration	55
3.6.1	Visualization of music collections	57
3.6.2	Feature Extraction	59
3.6.3	Visualization	59
3.6.4	Interface Evaluation	65
3.6.5	Is map coherence important for music discovery?	67
3.7	Tabletop Applications TSI	71
3.7.1	Puckr	72
3.7.2	TUIGoal	72
3.7.3	Punk-o-Table	73
3.7.4	Smash Table	73
3.7.5	80-table	74
3.7.6	Tower Defense	74
3.7.7	Pulse Cubes	74
3.7.8	Rubik Musical	76
3.7.9	Naus	78
3.7.10	Scrabble	79
3.7.11	Ranas	79
3.7.12	TSI applications' statistics	79
3.8	Conclusions	80
4	Empowering application developers	83
4.1	ofxTableGestures: a framework for programmers	84
4.1.1	The early stages: providing libraries (2009)	85
4.1.2	Creating Gestures is the key (2010)	88
4.1.3	Multi-user gestures (2011)	93
4.1.4	Simplifying the API (2012)	98
4.1.5	Discussion	103
4.2	MTCF: a platform for sound and music tabletop creation	104
4.2.1	A Reactable-like playground	105
4.2.2	Allowing interface design	112

4.3	Conclusions	117
5	Multi-Application systems: GestureAgents	119
5.1	Introduction	119
5.2	Collaboration in shareable interfaces	120
5.3	Multi-Tasking Shareable Interfaces: Current Situation and Related Research	122
5.4	Approaches to Multi-Tasking	125
5.4.1	Input sharing	125
5.4.2	Area-based interaction	127
5.4.3	Arbitrary Shape Area-Based Interaction	128
5.4.4	Content/Semantics -based input sharing	131
5.5	Approaches for Multi-Gesture Applications	135
5.5.1	Gesture recognition and disambiguation in a single application . .	135
5.5.2	Gesture Composition	140
5.5.3	Single-Gesture Certainty and Real Time Disambiguation Strategies	140
5.6	Implementation of GestureAgents Framework	142
5.6.1	Elements of GestureAgents	142
5.6.2	GestureAgents Protocol	144
5.6.3	Restrictions on the Behaviors of Recognizers	148
5.6.4	The GestureAgents System	149
5.7	The GestureAgents Recognition Framework	152
5.7.1	Recognizer composition	152
5.7.2	Recognizer instances as Hypotheses	154
5.7.3	Context polling	156
5.7.4	Link with the application	158
5.7.5	Provided Gestures	158
5.8	A second iteration: Composition with 3rd Party Apps	158
5.8.1	A working gesture composition example	159
5.8.2	Two apps, two developers and composition: Not working	161
5.8.3	Revisiting gesture composition	163
5.8.4	Implementation	164
5.8.5	Effects on Policies	167
5.8.6	Portability	167
5.8.7	Testing	168
5.9	Applications and systems created with GestureAgents	168
5.9.1	Original tests	168

Contents

5.9.2	Example Applications	169
5.9.3	Orchestra Conductor Gesture Identification	171
5.10	GestureAgents Code	176
5.11	Discussion on the framework	177
5.11.1	Accessory agents	177
5.11.2	Temporary Feedback	177
5.11.3	Supporting Other Gesture Recognizing Techniques	178
5.11.4	Security	178
5.11.5	Efficiency	179
5.11.6	Debugging and Testing	180
5.11.7	Considerations on the second iteration	180
5.11.8	Considerations on decoupled interfaces	181
5.11.9	Future steps	183
5.12	Conclusions	183
6	Conclusions	185
6.1	Contributions	186
6.2	Future Work	187
	Bibliography	189
A	List of publications	205
B	Tabletop applications in TEI, ITS, and tabletop conferences	207
C	TSI Applications	209

1 Motivation

The intention of this thesis is to boost the usefulness of tangible tabletop interfaces which are scarcely used today. The tablet PC introduces a good precedent of a new technology that while currently being widely adopted, failed in its first attempt to be integrated into mainstream culture. We argue that the causes of this early failure were related to the lack of support of the affordances provided, something similar to what, in our opinion, is happening to tabletops.

1.1 Learning from mistakes

In recent years, we have witnessed the successful adoption of multi-touch personal devices, bringing ubiquitous computing to a broad population. Tablets, smartphones and other personal devices are now an integral part of our lives. We use them every day and find them somehow essentially useful.

One could think that these very useful devices were destined to rule our lives from their invention but, far from this recent success, the turbulent story of tablets started quite some time before.

In 2001 the tablet PC was introduced to the market by Microsoft. It consisted of a specification to build laptops that would transform into tablets, devices in which the keyboard was hidden and all the interaction was done directly through the screen. The idea was to try to fill the gap between laptops and paper: although paper was preferred over computers for some tasks and situations, such as note-taking or drawing and painting, the resulting document had to be digitized in order to get all the benefits of a digital document. Creating the document directly in the computer was still possible, but the uncoupling of the keyboard and mouse from the visual representation when taking notes, or of graphic tablets in the case of drawing, simply could not beat the directness of pen and paper, where one draws and writes directly on the resulting document.

The affordances theoretically provided by the tablet PC were very valuable: by interacting directly with the screen, users could get rid of the indirection that was supposedly

1 Motivation

preventing them to use computers for the aforementioned activities. However, in the end, it did not achieve its goal of replacing paper in those situations. The lack of adoption of these technologies can be blamed upon several aspects, the first probably being that the additional price requested for a tablet PC over a laptop was too high at the time. The ergonomic solution was not optimal either; as those devices were cased on convertible laptops, and their weight and thickness made them too impractical to replace paper.

While these two factors could explain the phenomenon from a technological point of view (as in "we did not have the technology back then"), we must also recognize the failure of the interaction strategy. The tablet PC was conceived as an extension of the traditional PC, in a sense that most of the input created by the pen-based interaction was equivalent to the mouse-based one: there existed equivalents for click, double click, free mouse movement, etc, because both applications and operating systems needed to support both input methods, and the easiest solution was to emulate a mouse. This obviously forced the interaction to be lowered to the maximum common subset. Those first tablet devices could be practical for either traditional computer activities or paper-like interaction; but not for both at the same time.

This limitation interferes with the original affordances that were expected from this kind of systems and blocks them. Having an expensive version of a laptop which includes technology that does not pay off is probably the reason that drove potential users not to adopt it.

Finally, in recent years, we have witnessed the successful adoption of tablets (together with multi-touch technologies). This time the devices unharness their interactive and ergonomic affordances by not limiting the interaction: the operating system, the applications and the form factor are designed to empower those affordances. The results are well known.

A different kind of multi-touch devices, those which are not personal and small, but big and shared, is still yet to spread, of which many key affordances are often associated with these, such as collaboration and multi-user interaction. It is undeniable that any improvement of co-located collaborative interaction beyond the personal computing device family, which is, by definition, personal, would be extremely useful.

But before we start selling those devices as open, general-purpose computing devices we have to be sure that we understand their affordances and we empower them. The consequences of not preparing the tablet to fulfill the same functions it was designed to support, cost 9 years of misleading device production. We definitively do not want to make the same mistakes, as it would delay a useful technology that would solve many

problems, in a more convenient and accurate way than today.

The goal of the work described in this dissertation is to approach many of the obstacles that prevented (and some still prevent) these kinds of interfaces from being the next paradigm-changing technology that they could be.

1.2 A personal introduction to the thesis

My final undergraduate project was based on creating TDesktop, a Tangible Operating System, e.g. an OS for Tangible Tabletops. My Masters Thesis was centered on creating SongExplorer, a Tabletop application for finding new interesting music using spatial exploration. Those two experiences shaped my understanding of Tangible Tabletop Interaction in ways that I was not aware of and were the first steps in trying to solve a problem that was not explicitly stated and that has guided my entire career. It was not until my third year of my PhD that I started to realize the big conceptual problem that I was trying to solve.

My time working in the Musical and Advanced Interaction team (MAIn) in the Music Technology Group (MTG) at Universitat Pompeu Fabra (UPF) has involved working on several projects, usually around this problem: making the best of Tangible Tabletop interfaces. Some of the work has been driven (or stimulated) by the teaching I did, around interaction topics to very different groups of students.

The consistency of this thesis is, therefore, not given by the typical path of a seminal problem, of its analysis and solution, and its results; but as a large problem space and some steps to solve and explore major areas of it. It is thus composed of several actuations, some of them published, that structure a dissertation about how we should envision the future of this field.

1.3 Contributions of this thesis

This thesis covers many different topics in the path of *Making Tabletops Useful*, and so its contributions are diverse. The first part of the thesis contributes by exploring the distinctive affordances of tabletop devices and how they can be used in the fields of tangible programming, education, collection and map browsing and exploration. In particular, the approaches presented in these first sections are novel in tangible tabletops. An exploration of typical interaction design strategies that emerged in a tabletop application creation course are also presented. These observations and project presentations

1 Motivation

are relevant, given the lack of such observations in the field.

On a second part, two different approaches to support tabletop application creation are presented, addressing the specific needs of potential users. A novel teaching-driven approach is used and described, revealing the main problems new programmers face when creating this type of interfaces.

The last part presents what can be considered the main contribution of this thesis: a problem that has not yet been identified in the state of the art of tabletop systems: In order to support general purpose computing and to be useful to its users, tabletop systems should allow multi-user concurrent multi-tasking. The identification of this problem, its analysis and the proposed solution and development are important contributions. As a consequence, the specific strategy of a protocol-based content-based gesture disambiguation both inside a single application and between multiple applications is a relevant contribution. Finally, the approach of creating a device-agnostic (inter-application) gesture disambiguation mechanism covering coupled and uncoupled interfaces is also novel.

1.4 Structure of this document

The rest of this document is divided in five chapters. In the next one, Chapter 2: *Introduction*, we introduce the concepts and background required to understand the scope and focus of my work, stepping into interaction in personal computers, tangible and tabletop interaction, gestural interaction, interaction in windowing systems and finally presenting the plan for making the best of Tangible Tabletop interfaces by addressing three components: systems, applications and frameworks.

Chapter 3, *Exploring tabletops' distinctive affordances*, focuses on the affordances and capacities of tabletops, and how applications can make use of them. In this chapter we present the work by exploring two of the such specific capabilities: creative learning and programming, with TurTan, and large data exploration using spatial exploration, with SongExplorer.

Chapter 4, *Empowering application developers*, is devoted to frameworks and how they support various programming techniques and solutions to common problems. In this chapter we present two approaches created in different circumstances: when supporting undergraduate students for a tabletop application creation course, and when supporting NIME¹ students and practitioners.

¹NIME stands for New Interfaces for Musical Expression. It is a field that addresses novel interface design related to music performing, both from practitioners' and researchers' perspectives. The name was coined after its main yearly conference (<http://www.nime.org/>)

Chapter 5, *Multi-Application systems: GestureAgents*, focuses on the computing systems that run and manage applications, and their role with applications in the context of tabletop systems. Here we introduce a modern approach to the problem of concurrent multiuser multitasking in a single interface and its possible approximations, we explain how the selected approach can also be used to support multi-user interaction inside applications, and present a framework created to solve these problems.

Finally, Chapter 6, *Conclusions*, summarizes the work and advances made throughout this thesis, and comments on future work.

1 Motivation

2 Introduction

Understanding where we are coming from, is essential to assess the current situation. Why are PCs successful? How are they designed to support their capabilities? How are Tangible User Interfaces providing their own set of capabilities? What should we do to make Tabletops a useful computing platform?

2.1 The usefulness of the personal computer

The advent of computers and personal computing devices has had a very deep impact in recent history. Their usefulness relies in their capacity of performing tasks that previously were inconvenient, difficult or impossible, and the core goal of Human-Computer Interaction is indeed devoted to support users accomplishing these tasks (Shaer and Hornecker, 2010).

Many qualities of computers contribute indeed to this goal of easing task solving processes:

Computing power This was the original function of computers, the ability to compute mathematical operations in an unprecedented speed. There is no need to even try to enumerate the infinity of practical applications that computing power has created, transforming our world.

Convenience Computers make it easy to perform simple but repetitive tasks, instantly. Boring tasks that may not be specially difficult, when automated, can be performed in a fraction of the time originally needed, thus making them accessible to much more people. Spreadsheets are a good example: accounting existed before computers, but it is now more widely accessible.

Connectivity It could be argued that portable computing devices do not specially excel in computing power. Still, their usefulness has proved to be an absolute success, probably because of their connectivity, which allows users to communicate in efficient ways.

General-purpose computing What makes computers even more useful is that they are generic tools. The purpose and function of a computer can be changed by changing its program. This uncoupling of the device from its function, frees computer builders from having to think about every possible use of the machine. Other parties can create programs that will turn that computer into different specific tools such as a word processor or a calculator. In the smartphone revolution, for instance, the availability of third-party applications has also arguably been instrumental to their success (West and Mace, 2010).

These capabilities can be present in all computing devices. We could say that these are their *affordances*, in terms of its original meaning introduced by Gibson (1977), referring to all the possible action possibilities that an object provides to an individual and of which this individual is capable of performing.

The popularization of the concept of *affordance* by the HCI community is only with its second meaning, established by Norman (1988), that includes the requisite of these action possibilities being perceived by the individual. With this definition, affordances of manufactured objects will depend on the ability of designers to facilitate the perception and inference of the offered action possibilities.

In our case, different instantiations of computers will have different success in facilitating access to the aforementioned capabilities to their users, depending on their Norman's affordances embedded in their design. It is also important how those are strengthened with their interaction strategies.

The next sections present how those computing devices are and how their interaction methods provide (or complicate) those affordances.

2.2 Interaction in personal computers

Although most of the discussions and work described in this document relate to tangible interaction, we will introduce concepts from the PC¹-based HCI. Tangible Tabletop interaction techniques have been influenced by previous ones, mostly related to the paradigmatic technology of the time.

In the same way that PC interaction was inspired by the technology of the time (i.e. documents, folders, desktops, typewriters...) the next wave of devices (such as tabletops or smartphones) were, on its turn, inspired by already established personal computers'

¹In this text we will be referring to the Personal Computer as PC, not referring to a particular brand, architecture or operating system.

concepts (Blackwell, 2006).

In order to fully discuss aspects of tangible and tabletop interaction we must therefore first understand the origin of the PC desktop metaphor and of many of its elements, which were later adapted.

2.2.1 WIMP

Modern day computer systems use Graphical User Interfaces (GUI) to interact with the user. This involves the use of visual monitors for output information. But previously the interaction was purely textual, consisting of a Command Line Interface (CLI). The user typewrote commands to the computer (initially they were printed on paper) and the computer answered by typing text in the same medium. After introducing monitors, it was clear that those could be used to show more than text, such as images, and later interactive images for interaction, GUIs.

In most of the current cases of PC operating systems, they have GUIs that make use of the WIMP paradigm. The Window Icon Menu Pointer (WIMP) interaction was developed at Xerox PARC and popularized with Apple's introduction of the Macintosh in 1984. It refers to the GUI that bases its interaction to control programs via several elements drawn in a graphical display:

Windows are rectangular areas that contain the GUI of an individual running program. Instead of occupying the whole screen, programs are contained in windows that can have arbitrary sizes and can be moved around.

Icons are graphical representations of (virtual) objects or actions. Computational actions -such as commands- can be triggered by interacting with them. For instance, most of the interaction in file system operation is based on manipulating icons.

Menus are selection systems based on graphical lists of actions. The items can be textual or iconic and the menu may contain items that hold other menus -sub-menus- creating *hierarchical menus*. Menus that can be invoked anywhere are called *contextual menus* and they only give operations that are possible in the current context.

Pointer is a virtual element on the screen (usually depicted as an arrow) that can be controlled using a physical device (such a mouse or a touchpad) and that is used to interact with all the virtual elements of the screen.

Metaphors (prominently relating to the office world) are largely used in those systems, relating virtual elements and behaviors to real ones using their pictograms as icons: files

2 Introduction

(represented by paper icons) can be removed by placing them into the deleting folder (represented by a bin icon). The whole system is often called the *desktop metaphor*, as most of the references relate to desktop and office objects and actions.

The use of metaphors has been vastly discussed in the HCI field. The concept itself is central to this discipline. Although shaping interaction to loosely mimic real world behaviors seems to be valuable for users, its abuse can be considered harmful. As time passes and computers become more prevalent, some of the metaphors lose their original meaning as the real world references decay in importance, and we see the computer-based behaviors as the reference for newer metaphors in other systems (Blackwell, 2006).

2.2.2 Direct Manipulation

Most of the aforementioned techniques can be placed around the concept of *direct manipulation*, a term that was first introduced by Ben Shneiderman in 1983 within the context of office applications and the desktop metaphor (Shneiderman, 1983). The concept was introduced to criticize the command line interface, dominant at the time, where commands are entered in textual mode by the user into a command prompt, usually adopting a “command argument1 argument2 ...” structure. In contrast Shneiderman proposed replacing this interaction by another one centered on the object: instead of starting with the verb (command) and then adding the arguments, to start with the object (argument) to then select the verb. The object could be a visual representation such an icon, a particular place in a text (the cursor) etc. Direct manipulation refers also to the ability to perform actions on the object using contextual menus that only show the available commands for such object and context, or by using actions that would loosely correspond to the physical world, such as virtually dragging an icon to issue a move or transfer command.

The rationale behind these propositions was to concentrate the action-related information and interaction near (or over) the manipulated object, while maintaining a visual representation of the actual data (as in WYSIWYG (Hatfield, 1981)), assuming that real-world metaphors for both objects and actions would make it easier for users to learn and use an interface.

This idea is important not only to the WIMP interaction paradigm and the desktop metaphor in PCs, but also to further evolutions of the computer user interface, such as in tangible interaction.

2.2.3 Multi-Tasking and third-party applications

General-purpose computing allows a single computing device to change its function according to a program. The computer itself has no specific objective, as a calculator would have. Logical programs are executed by the computer instead, making it useful for a specific calculation or any task completion. On the other side, as the universe of possible complex tasks and problems to be solved with the assistance of a computer is broad and open, it seems rather unpractical to create a single program for every different task. As complex tasks can often be divided in simpler subtasks, the particularity of every task will often require the use of several different more generic programs, which will address some of the subtasks we can divide the original problem in. Those programs, on its turn, can then be reused for other different tasks.

Let us imagine, for instance, that someone is writing a report on the discoveries of new wildlife in a country. This task will require writing, editing and formatting text, capturing, classifying and editing images, calculating statistics and displaying charts, creating and manipulating maps, etc. Instead of having a single program for “new wildlife finding report writing” involving all these activities, several programs addressing the needs of every single activity can be used: a word processor, an image editor, a file browser, a spreadsheet editor, a map browser, etc. These programs, such as text editors or image viewers, usually designed to solve domain-specific problems, can be also created by parties that do not relate to the creators of the hardware or the programmers of the OS. These third-parties can be programmers or teams that have an expert knowledge of the field the program is focused on. Allowing third party software to be created without the prior consent of the computer’s and OS’ designers or other software creators, allows new programs to continuously appear for filling potential new needs.

There is indeed a common agreement that allowing third-party applications is an important factor for success on commercialization of computing platforms. A classic example would be the effect of commercialization of the Lotus 1-2-3 spreadsheet program exclusively for the IBM’s PC. Sales of IBM’s PC had been slow until 1-2-3 was made public, and then increased rapidly a few months after Lotus 1-2-3’s release². As a more contemporary example, Apple trademarked the slogan “*there is an App for that*” for its iPhone 3g selling campaign on 2009³, advertising the availability of third-party apps as its main appeal. This move by Apple revolutionalized the smartphone scene (West and Mace, 2010).

²https://en.wikipedia.org/wiki/Killer_application

³<http://www.trademarkia.com/theres-an-app-for-that-77980556.html>

2 Introduction

Modern operating systems and computers allow several programs to be ran in parallel, and to switch interaction with the user at any desired time. This ability, multi-tasking, helps to use computers to solve a particular task that involves several steps and requires potentially different programs, in a more convenient way than having to stop the current program to start another. Multi-tasking would be indeed very convenient in our hypothetical new wildlife finding report writing activity: while our user is writing the report, she has the need of inserting a picture of a new specimen. She switches the interaction from the word processor to a file browser to find the picture she wants, she then opens it inside an image editing program (another switch), where she crops the marginal part of the picture. She then switches again to the word processor (that still holds the document she was working on) to insert the modified image.

2.2.4 Interaction Foci

Some characteristics of PC interaction are directly related to the very purpose of personal computing and often overlooked, assuming its universality. A notable example could be that PCs are designed as single-user appliances, and every aspect of them is only supposed to be used by one single user at a time. This focus can be explained by the context for which personal computing was developed: the office. Many office-related activities are individual: writing, accounting, reading, drawing... it was logical to shape PCs with this in mind; the setup of a monitor, a keyboard and mouse is optimized for individual focused work.

One evident consequence of this is that PCs are supposed to have one single keyboard and mouse, to be operated by one single individual. This constrain has very deep implications in the mechanisms of current systems: *Only one virtual pointer is available, so multiple pointing physical devices will refer to the single virtual pointer on the screen.*

The consequence is that there is a single application with a single widget that will receive all the keyboard input events: one privileged application and widget that is in focus. So multiple keyboard devices will insert text into the same application and position in the text (cursor) or receiver widget.

The assumption of having only one keyboard and mouse, used by only one user (but not both at the same time), renders interaction with a GUI much more easy to design in a way that is predictable. As the mouse and keyboard are uncoupled from the output device (the graphical display) allowing multiple graphical and textual cursors would require an explicit mapping to know which device is paired with which cursor.

The mechanism used in personal computing to assign a receiver for the keyboard input

is called *focus*. This mechanism makes it easy to predict the receiver of keystrokes from the keyboard by defining a single target at all times. This target is often highlighted (using color or intermittent visibility) and its container application window shown on top (occluding the others).

Focus can be changed both by using the keyboard or the pointer. The keyboard *tab* key is used to switch the sink inside the application, or between different applications (in combination with *alt* or *command* keys). Whenever a user interacts with a pointer with an specific application and widget, it receives the focus.

This mechanism assumes that the mouse and the keyboard will not be used in different applications at the same time, for instance writing a letter while drawing a picture, effectively making interaction with a PC a single-tasked one. Although different applications can be running simultaneously, even producing video or audio, only one can be interacted with by the user at the same time.

Research projects and implementations exist that try to cover this many-mice and many-keyboards issue. MTX (Hutterer and Thomas, 2007) is an extension of X11 graphical system (present in several flavors of UNIX) that allows having virtual input pairs of visual cursors and keyboards that can operate at the same time with adapted and legacy X11 applications. Each pair has a focus target and multiple visual cursors can interact with different applications at the same time or even with the same application if it supports MTX extension. The main problem MTX has encountered is that widget frameworks used to build X11 desktop applications may not support this extension and still assume that there is only one single cursor and keyboard.

Dynamo (Izadi et al., 2003) was another example of many-mice many-keyboards system, focused also on ownership and permissions over programs, documents, etc in a shared multiuser WIMP system. Users may use mouse-keyboard pairs to interact with a system that presents local and shared interfaces. In shared interfaces it focuses the attention on methods for preserving and sharing control over applications and files. More examples of systems avoiding a single focus of interaction can be found in Section 5.3.

We have to consider that having a single *focus* at a time is extremely useful, and preventing multi-user or multi-application interaction poses no problem to the PC use, as the PC is in every aspect designed to be single-tasked and single-user. Even when using solutions like MTX, the ergonomic characteristics of the PC prevent users from effectively sharing it with others.

2.2.5 Post-Wimp GUI

Other than the CLI, PCs sometimes expose other types of GUI, not (totally) based on WIMP. Because of the practical ubiquity of WIMP most of the GUI-building libraries are WIMP-compliant by default, making the task of building non-WIMP-compliant application a difficult one. For this reason, only the applications that are forced to leave out WIMP principles in order to expose WIMP-incompatible functionality do so.

In particular, immersive experiences such as video games tend to violate some of the WIMP principles and therefore are forced to give up on them:

Grabbing the cursor Instead of using the mouse to control the movement of the usual virtual cursor, their movements are used directly. The mouse becomes a controller, as it would be a joystick, for instance to manage a virtual 3D camera by rotating its angle or moving its position. In particular, first-person shooter video games make heavy use of this mapping.

Disabling control keys By using the keyboard as a multi-button controller instead of a text-input device, games allow the players to control elements of the game very efficiently. By pressing many keys independently at the same time, users can control multiple aspects of the game simultaneously, something rather complex to achieve in the WIMP paradigm. Even multiple players can interact with the application simultaneously by sharing the keyboard.

As a result of this shift on the keyboard mapping, and because the effective lack of a single focus point of interaction, the use of some keys as focus control mechanism disappears.

Full screen By removing the window and adopting the whole screen, a program can create a more immersive experience, as all other programs and desktop environment items are hidden. This breaks the multi-tasking capability of WIMP systems on purpose.

By giving up on the cursor, control keys and windowing elements and capabilities of WIMP, those applications can offer an immersive, multi-user experience. This means, however, that the features provided by WIMP, such as multi-tasking support, are also lost.

Not only games give up WIMP capabilities to overcome its limitations. Coming from practices of personal multi-touch devices, we have lately seen the introduction of operating systems that cease using some WIMP interaction windows in favor of full screen

applications (with optional screen tiling as multitasking approach⁴) probably due to their limited space and inaccurate pointing mechanisms, and that abandon the mouse as a pointing device because of convenience. Microsoft Windows 8 native operation follows this approach⁵ even in PCs.

2.3 Gestural Interaction

Before the appearance of the WIMP, research on interaction with computers addressed many now exotic (or just now recovered) modes. The seminal work of Ivan E. Sutherland in the 1960s and 1970s is essential for the development of new HCI disciplines, and representative of this open view. With SKETCHPAD, Sutherland (1964) introduced the first direct pen-based visual interface to a computer, where the pen input device was used directly into the screen, manipulating directly the drawn lines and other graphical elements, in order not only to draw, but to program graphically. This work was extremely influential to the modern GUIs and object-oriented programming languages, and appeared almost simultaneously as the mouse, created by Douglas Engelbart but not publicly demoed until 1968.

Four years later, Sutherland also introduced the first ever head mounted display (HMD) with the aim to present the users with stereoscopic images according to the position of their heads (Sutherland, 1968), effectively inventing the field of virtual reality. A field that after a period of practical inactivity seems to be recovering thanks to the improvements on displaying and tracking hardware that allowed consumer-level HMDs such as OCULUS RIFT⁶ to appear.

If Sutherland can be considered the founder of many HCI fields, Myron Krueger is the pioneer of gestural interaction, understood as body movement-based interaction. Krueger experimented with the body silhouette images processed in real time to interact with the computer. In VIDEOPLACE, (Krueger et al., 1985) the participants entered a room where their silhouette captured by means of a camera was projected on the wall, along with digitally generated images of virtual elements. Users could interact with such virtual objects through their mediating projected image.

The same work explores a desktop version of the system, where only the hands and arms of the user on top of the table are captured, instead of the full body. Then the interaction

⁴Microsoft Windows 8 multitasking by tiling <http://windows.microsoft.com/en-us/windows-8/getting-around-tutorial#apps>

⁵Microsoft Windows 8 <http://windows.microsoft.com/en-us/windows-8/meet>

⁶[HTTP://WWW.OCULUSVR.COM/](http://WWW.OCULUSVR.COM/)

2 Introduction

takes place with the silhouette of the hands, displayed in a monitor, ideally embedded into the wall. This can be considered as the first attempt to create a bimanual user interface, one designed for the user of the two hands simultaneously. The combination of the two settings was used also, allowing participants in the interactive room to interact with the hands of the operator, in another place, with very interesting results.

Following the steps of Krueger, Penny et al. (1999) developed many years later a real-time video-based capture system for the 3D shape of a user, to be used in 3d interactive immersive environments, such as the CAVE. In this work, the system could create an enclosing volume with roughly her shape by intersecting the silhouette images of the same body from different angles. Penny originally used this system in a series of artistic experiences in interaction with 3D spaces and entities, from creating voxel-based automata life in 3d, to leaving 3D shadow traces of the user's volume. Also, in later installations, it was used for simple user tracking in 3D space.

Nowadays, volumetric user tracking has been, in fact, popularized thanks to new depth-perceptive cameras developed for video games, such as Microsoft Kinect⁷. These use a structured light approach to capture the depth of the image, in addition to the color one. This is then processed to extrapolate the pose of the user's body and used to control the system using body gestures.

The path of bimanual input laid by Krueger was soon pursued further by Bill Buxton (1986), first at the University of Toronto and later at Xerox Park and at the Alias|Wavefront company, developing some prototypes such as a multi-touch tablet (Lee et al., 1985).

An interesting approach into this field was presented by Bier et al. (1993) TOOLGLASS AND MAGIC LENSES. In it, two pointing devices were used simultaneously by the user; one as the cursor and the other as a transparent palette between the cursor and the application. This palette may specify commands, properties, which can be executed by clicking into the virtual object through the palette. They also provided a preview of the effect the command was going to perform into the object. A two handed strategy that was then proved to be natural to the users (Kabbash et al., 1994).

Using two hands to manipulate 3D props in space proved to be also a fruitful approach: 3-DRAW system by Sachs et al. (1991) allowed to draw in 3D space by using a tablet representing an arbitrarily oriented plane and a stylus; THE VIRTUAL WORKBENCH (Poston and Serra, 1994) presented the users with a 3D scene that they could manipulate by using the two tool handles; Hinckley (Hinckley et al., 1998, 1994) proposed a method

⁷<http://www.xbox.com/en-US/kinect>

of MRI visualization for neurosurgical purposes in which the user specified a cutting plane by manipulating a doll's head and a cutting plane prop in free space. This field would rapidly become the origin of Tangible Interaction (see Section 2.4).

Recently, successful consumer-level devices for bimanual gestural interaction have appeared. Leap Motion Controller⁸ attempts to address this type of interaction by tracking hands and fingers close to the structured light sensor⁹, usually placed near the screen.

Aside from camera-based approaches, using accelerometers and gyroscopes to track the body movement has been a common strategy. A notable example is the Wii REMOTE¹⁰, which introduced motion-based interaction to the general public in late 2006, triggering many studies and applications in the academic world, such as (Schlömer et al., 2008; Vlaming et al., 2008; Lee et al., 2008). A similar use case has been being used for a while in smartphones, recently with better precision by the use of sensor fusion techniques (Sachs, 2010).

An specific area where gestural interaction and bimanuality has an important role is music performing. Music performing and improvisation is a field that often can require many dimensions of control of the instrument, and thus the instruments tend to take advantage of the directness and abilities of the hands and body movement. Even the strangely popular Theremin, the first fully functional electronic musical instrument, was controlled by moving the hands in the air.

Gestural interaction and bimanuality in music performance can be found in custom hand controllers such as in The Hands (Waisvisz, 1985), profiting from joysticks and other standard computer peripherals (Jordà, 2002; Arfib and Kessous, 2002) or even using special globes for per-finger control (Kessous and Arfib, 2003).

2.4 Tangible Interaction

In 1995, Buxton together with then PhD students George W. Fitzmaurice and Iroshi Ishii, had already demonstrated how the sensing of the identity, location and rotation of multiple physical devices on a digital desktop display, could be used for controlling graphics using both hands (Bier et al., 1993; Kabbash et al., 1994). This work introduced the notion of *graspable interfaces*, formalized in BRICKS (Fitzmaurice et al., 1995), which

⁸<https://www.leapmotion.com/product>

⁹Although it seems that the technology used by LEAP MOTION remains undisclosed, external observation indicates that infrared structured light is the base of their solution.

¹⁰It seems that NINTENDO does not provide an official website to this product, so you can instead go to WIKIPEDIA: https://en.wikipedia.org/wiki/Wii_Remote

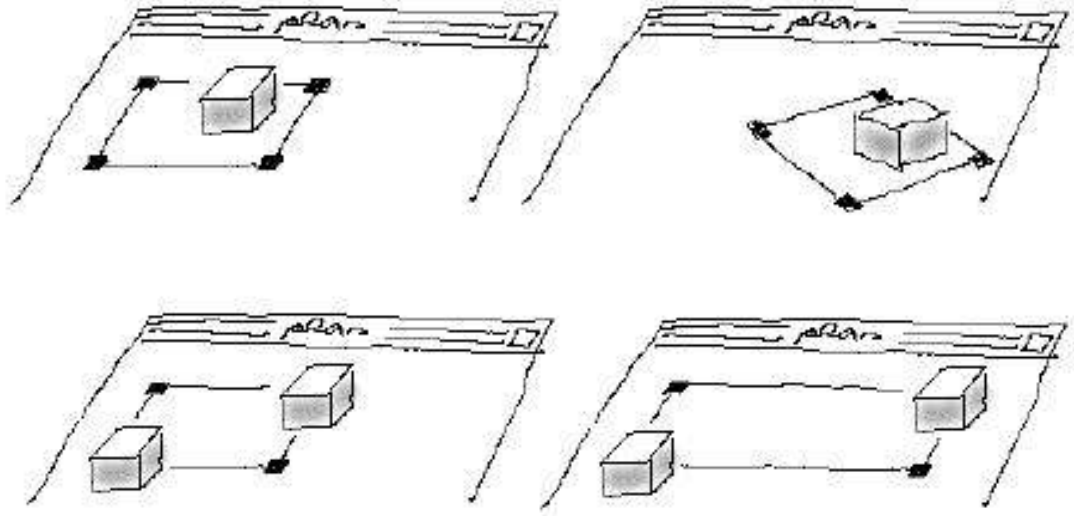


Figure 2.1: Different operations of physical manipulation of digital elements in bricks, as shown in (Fitzmaurice et al., 1995)

were to become two years later “tangible interfaces”.

In BRICKS, Fitzmaurice presents a system where wooden blocks are placed on top of an horizontal display (ACTIVEDESK) and uses them to control virtual elements. By moving the bricks over the display, the coupled virtual elements would be transformed accordingly. Using a single brick would result in translation and rotation, while using two or more would allow zoom, pivotal rotation and other non homogeneous transformations (see Figure 2.1). Many of the created interaction techniques are the ones used nowadays in multi-touch devices.

Two years later, Hiroshi Ishii at the MIT Media Lab coined the term *tangible user interface* in 1997 (Ishii and Ullmer, 1997), although several related research and implementations predate this concept. Ishii picked the abacus as the source of inspiration and the ultimate tangible interaction metaphor because, unlike pocket calculators or computers, in the abacus, input and output components coincide and arithmetical operations are accomplished by the direct manipulation of the results. Following this captivating idea, Ishii envisioned TUIs as interfaces meant to augment the real physical world by coupling digital information to everyday physical objects and environments, literally allowing users to grasp data with their hands, thus fusing the representation and control of digital data and operations with physical artifacts. Instead of having separated phys-

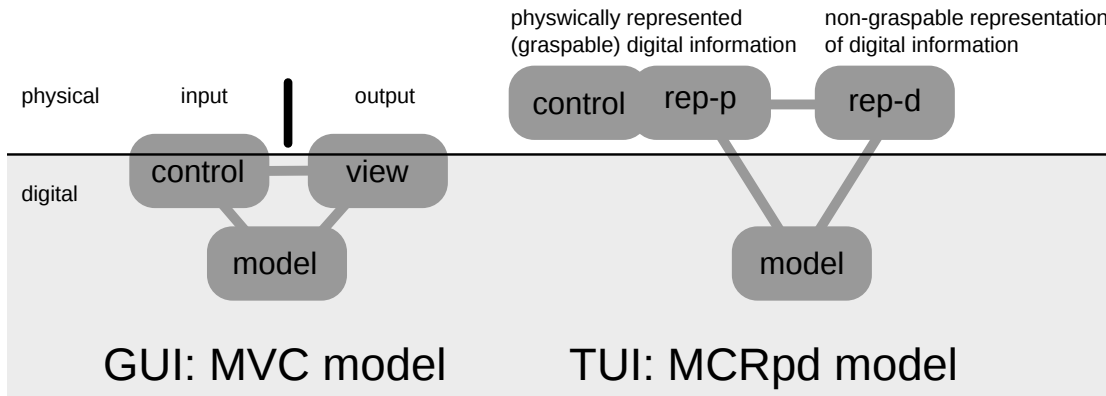


Figure 2.2: GUI and TUI interaction models according to Ullmer and Ishii in (Ullmer and Ishii, 2000).

ical controls and digital data and logic, with separated input and output, tangible user interfaces created a more tangible and coupled physical and digital representation and control, while leaving only data as purely digital (see figure 2.2).

One of the inspirational examples identified by Ishii was the Marble Answering Machine conceived by Durrel Bishop in 1992 (Bishop, 2009). In this never implemented concept, physical marbles were attached to digital information (phone messages). When the answering machine registered a new message it released the corresponding marble. When the user wanted to listen to the pending messages, she only had to take each marble and introduce it into a slot. Although the marbles did not carry any digital information in them, they served as a handle to reach to it, and physical actions could have digital consequences.

In a way, tangible interaction can be seen as an extension and deepening of the concept of “direct manipulation”. With GUIs, users interact with digital information by selecting graphic representations (icons, windows, etc.) with pointing devices, whereas tangible user interfaces combine control and representation in a single physical device (Ullmer and Ishii, 2000), emphasizing tangibility and materiality, physical embodiment of data, bodily interaction, and the embedding of systems in real spaces and contexts.

Although WIMP concepts are widely used in Ishii’s seminal paper, posterior works, such as Urp (Underkoffler and Ishii, 1999), try to avoid them as much as possible, while maintaining their direct manipulation flavor. Urp was an urban planning workbench, where buildings are represented by three-dimensional physical miniatures on an horizontal surface while the system simulates their shadows or sun reflections in real time. The manipulation of the data is purely physical, and the resulting image takes physical and

2 Introduction

digital part, as the buildings are in scale.

Ishii's vision in *tangible bits* presented various interface possibilities other than objects-on-surfaces-based (i.e. metaDESK and transBOARD) ones. Ambient displays are one example, now often regarded as a separated field, present information to the individual in the background so it is unconsciously processed. An inspirational example mentioned by Ishii is Live Wire (originally *Dangling String*) (Weiser and Brown, 1996), a plastic spaghetti hanging from the ceiling, where an electric motor connected to the Ethernet network makes it to oscillate every time a packet is detected. The result is that when the network is calmed, the string moves every phew seconds, while when busy it starts making a distinctive noise because of its movement, proportional to the business of the network. Inhabitants have then an ambient measure to be aware of the network load.

Another interface possibility, and probably the most identified with the tangible interaction field, is the one suggested by the Marble Answering Machine, objects with embedded input and output capabilities. Examples could be objects that connect between them to create combined high-level behaviors such as Flow Blocks (Zuckerman et al., 2005), a series of peaces to explore causality, or AlgoBlock (Suzuki and Kato, 1993), a programming language made of connectible blocks as instructions; or objects that can be interacted by manipulating their shape or part configuration, such as Topobo (Raffle et al., 2004), a robotic creature construction kit with kinetic memory, or roBlocks (Schweikardt and Gross, 2006), a modular robot constrction kit where the physical modules are also logical functions.

2.4.1 Tangible and tabletop

Tabletop interfaces are a kind of tangible user interface. "Tabletop" relates to its physical shape and proportions: horizontal surface of the height and size of a table. Any computing system that uses a table-shaped interface could be described as a tabletop. Usually, the visual output is projected or displayed on the surface, while allowing touch interaction with fingers and other instruments as the input.

In Tangible Bits, Ishii already describes tabletop-like interfaces: metaDESK, and Bricks was described even earlier. In fact, ActiveDesk, the platform where bricks was built upon, is one of the first tabletop systems that combined a sensing camera and a projector (Buxton, 1997).

Additionally, physical objects can be placed on the surface as part of the interaction. In this case we can refer to them as *tangible tabletop interfaces*, and the *tangible tabletop interaction field* (or *surface computing*, in reference to Microsoft Surface, a commercially

available tangible tabletop device). We could say that the first tabletop systems, such as ActiveDesk, were *tangible tabletop* as they were using physical objects. However, they were not because of a preference for the concept, but because of the limitation of touch sensing technologies, which were unable to recognize direct hand touch to the interface.

Urp is, instead, one of the earliest Tangible Tabletop applications which unveils and suggests its potential. Developed as a town planning aid in the late 1990s at the MIT Media Lab (Ben-Joseph et al., 2001), in Urp, various users can analyze in real time the pros and cons of different urban layouts by arranging models of buildings on the surface of an interactive table, which represents the plan of a town or a district. The surface provides important information, such as building shadows at different times of the day (see Figure 2.3).

Urp is executed on an augmented table with a projector and a camera pointed at the surface from above. This system permits the detection of changes in the position of the physical objects on the table and also projects visual information concerning the surface. Other elements can be included, such as a clock to control the time of day in the system. Urp detects any changes made in real time and projects shadows according to the time of day. All these properties could obviously be achieved with the usual mouse-controlled software on a conventional screen, but the interest of this simple prototype does not lie in its capacity to calculate and simulate shadows, but rather in the way in which information can be collectively manipulated, directly and intuitively. This example, although quite simple, already unveils some of the most important benefits of tabletop interaction: collaboration, naturalness, and directness.

We can see how, with the combination of the tracking of control objects on the table with projection techniques that do convert the table into a flat screening surface, these systems can fulfill the seminal ideal of tangible interaction, of “adding digital information to everyday physical objects”, allowing digital entities to coexist as fully digital non-physical form and as shared digital-physical form.

We can think of tangible tabletops as a pragmatic version of tangible interfaces. The tracking of objects’ state is easy to do because of its limits, as they are usually rigid and therefore the only parameters to be recognized are position and orientation, which can be tracked with video processing methods. It also facilitates their digital enhancement by projecting the image on it or around it (as auras), instead of having to physically enhance them by embedding electronics and display mechanisms. Also, because of the transitoriness of this visual enhancing technique, the physical objects can adopt many different roles and become generic; this way each application can define their role at any

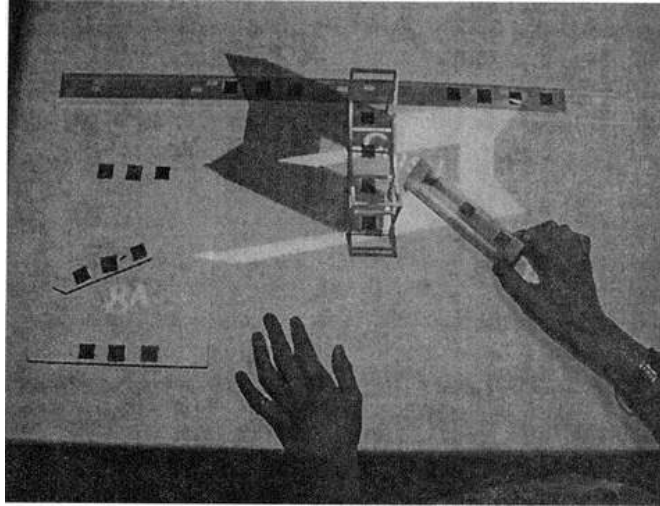


Figure 2.3: Urp (image from (Underkoffler and Ishii, 1999))

time. By gaining leverage by using those pragmatic approaches, we can get a versatile, affordable tangible interaction solution.

It should be stressed, though, that not all existing tabletop technologies allow dual touch-object interaction, being very common the ones only accepting multi-touch input. To be able to place objects on a surface, the tabletop must be horizontally oriented, not tilted. In that sense, some studies analyze the benefits and disadvantages of tilted versus horizontal surfaces (for example (Morris et al., 2008)), suggesting that in many individual use cases, if tangible objects where not to be supported, tilted interfaces (like traditional architect’s tables) seem to be more engaging and feel more natural (Muller-Tomfelde et al., 2008).

The “popularization” of multi-touch tabletop interfaces came with the Frustrated Total Internal Reflection (FTIR) technique by Han (2005), an inexpensive and very powerful method to track multiple fingers on a translucent surface that finished with the original need of using physical elements to interact with the surface. This method providing the possibility of using the fingers directly to interact, accompanied by a very inspiring talk¹¹ and demos¹², were probably the origin of the multi-touch culture we have seen from DIY and commercial devices.

In contrast, a similar technique called Diffused Illumination (DI) (Schöning et al., 2008), although less reliable than FTIR when tracking touching fingers, allows the recognition of fiducial markers, which can be stucked into objects, allowing tangible tabletop

¹¹http://www.ted.com/talks/jeff_han_demos_his_breakthrough_touchscreen

¹²<https://www.youtube.com/watch?v=EiS-W9aeG0s>

interaction. However, the improvements on the related hardware and software have progressively decreased the performance difference between the two technologies. For a detailed description of a complete DI system see Section 3.4.

Available commercial and noncommercial tabletop platforms (and applications) are reviewed in Section 3.2.

2.5 Making tabletops useful

In the 1 chapter we introduced the motivation of our research, that we intend to make tangible tabletop devices useful to the population, and also exposed two successful cases of making a computing platform useful and widely used, PC and Multi-touch personal devices. Here we analyze the positive factors that contributed to their success, and compare them to the current situation of tangible tabletops.

Price and Availability constitutes one of the major factors for the popularization of a product or a technology. Obviously the widespread acquisition of a device will be limited by its prize and other market factors such as expectation, advertising, availability, etc. The PC has been historically decreasing prices until a point where buying a PC represents a expenditure equivalent of a fraction of the average monthly income. Personal multi-touch devices have been lowering prices too and now it is difficult to see new cheaper non-multi-touch cellphones being marketed.

In the case of tangible tabletop devices, the situation has been slowly improving. Since the first examples of tabletop computing (or surface computing) the price has been decreasing, but it is still fairly expensive. Latest developments by MICROSOFT and SAMSUNG (Pixelsense¹³, based on (Hodges et al., 2007)) led to a mass-produced device that would have to theoretically drop the prices, by using a different embedded technology.

Affordances and Ergonomics would define if the device is capable of being used in situations or in ways not possible before. PCs base their design on focused single-user single-task interaction, making it valuable for high-concentration tasks such as productivity applications.

Multi-touch personal devices are designed to be portable and small. Their size and lack of an ergonomically correct keyboard makes them unsuitable for office applications, but very useful to perform some tasks away from the PC. Their mobility allows users to perform tasks in places and situations that before were

¹³<http://www.pixelsense.com>

not possible. Also importantly enough, its multi-touch interaction, extremely tied to direct manipulation, is direct and expressive enough to be attractive for real-time artistic applications such as musical or gaming ones.

Tangible tabletop devices differ from the PC and multi-touch personal devices in its big size and focus on multi-user settings. We will see the benefits of such interfaces afterwards in Section 3.1, but let us just summarize that they are known to encourage creativity and expression (Catalá et al., 2012) as well as collaboration (Hornecker and Buur, 2006). Some of those properties are totally absent if not penalized on the PC and multi-touch personal devices.

Application frameworks that take into account the aforementioned affordances are needed in order to create applications that can be used in consonance with the physical device's abilities. The WIMP-based frameworks have been shaping the way applications behave for the whole history of PCs, allowing them to correctly use the single focus mechanism, understand the double-click or to present pull-down menus. The first wave of tablet PC devices still used those frameworks, totally ignorant of their ergonomics and affordances, different from PCs, and that was surely a factor that prevented its success.

Multi-touch personal devices also have their application frameworks that make it easy to use the default gestures of the device, use its sensors and other features. By using these frameworks and not WIMP oriented ones, the applications are prepared to behave well and to profit from all device affordances.

There are many application frameworks in tangible tabletop, although there is not a clearly defined set of the features that they should implement. Notable missing parts are facilities for creating new complex and multiuser gestures.

Third party useful applications are a very important factor for success. The creators of a general computing device or system are not able to predict all the possible applications of it and, therefore, if they create a closed system with a set of applications that should cover all the needs of the users, the device will become useless for many users. The PC was already created with the philosophy of allowing third party apps. This was not the case of the personal device market; for many years personal devices such as cellphones, address manager, digital phone books, digital personal assistants existed with a limited set of functions and applications set by the builder.

This factor is indeed so important that the path of multi-touch personal devices to distributed third party applications, through a centralized catalog, proved to be

so successful that is being copied over other devices, such as PCs.

The temptation of creating a closed complete system that supports many tasks that are foreseen by the developers can be seen in tangible tabletops, although most of the systems in the market are still designed and sold as a single application appliance.

The ability to support third party applications that make the best from the device and makes it truly useful to the users depends at much extent on having good and coherent application frameworks.

Multi-Application systems, or operating systems in general, windowing systems in PC, are the last part of this puzzle for allowing the device to be useful. WIMP systems allowed multiple applications to be running in parallel, and being seen at the same time by placing its interface inside windows that would not occupy the whole screen, they provided drag-and-drop interaction for inter-application communication, and many other capabilities that made applications work well together. In multi-touch personal devices, the designers realized that small screens did not allow having multiple applications showing their state, and forced them to spread to the whole screen.

While some systems exist for tangible tabletop devices, they usually make the assumption of using either the full-screen approach, well suited for small screens, or WIMP derivatives, well suited to vertical single-user screens.

The application management system will define the way applications can cooperate, the way users will relate to applications and their capabilities. Its impact on third party applications behavior and design as well as on the application frameworks is extremely important. The lack of a well defined management system (or operating system, or windowing system) can be fatal for the appearance of good frameworks and applications.

With the goal of making a useful device from tangible tabletops in mind, we approach the solution by attacking the several factors aforementioned.

- In the first part we explore the affordances of this type of devices.
- Then we focus on the frameworks we need for developers to make use of those affordances.
- Finally we approach the system problem, by trying not to make the same mistakes of previous attempts.

Only by addressing these three aspects we can be sure that no other obstacles lie before these devices road to success.

3 Exploring tabletops' distinctive affordances

Tabletops provide some affordances that emerge from their shape, size and interaction capabilities, differing from those provided by PCs. Learning how applications can exploit them will allow application creators to shape their programs so that they are most useful to the users.

3.1 Affordances of tabletops

Since the conception of tangible tabletops by Ishii and Ullmer (1997), researchers have been intuitively assigning positive properties to this type of interface. The fact that tangible and tabletop interaction substantially changed the way people could physically relate to computers unharnessed the imagination of those that identified the original PCs restrictions as the main contributor to their limitations. Claims about its effect on learning, collaboration, creativity were rapidly assumed from informal evaluation (Marshall et al., 2007). Formal user studies have been, since then, sorting out the real affordances of these types of interfaces, which we can classify in three different types:

- Simultaneous input: the ones related to the ability of multiple simultaneous input points.
- Collaboration: representing the effects of allowing multiple users to work on the same goal.
- Physicality: the ones related to the use of real physical objects and devices as input as opposed to using virtual objects, in big interactive surfaces.

3.1.1 Simultaneous input

Several of the tabletop prototypes mentioned in Chapter 2 were, in fact, tables that often allowed their users to interact in two complementary ways: touching the table's

3 Exploring tabletops' distinctive affordances

surface directly, and manipulating specially configured real physical objects on its surface. Typically, this type of interface allows more than one input event to enter the system at the same time. Instead of restricting input to an ordered sequence of events (click, click, double click, etc.), any action is possible at any time and position, by one or several simultaneous users.

This has two different consequences. Firstly, interaction by using multiple input events simultaneously (such as in multi-touch) allows the appearance of *rich gestures*. In fact, multi-touch interaction is arguably the most commercially successful capability of horizontal surfaces. And the “pinch-zoom” technique is only one example of hundreds of possibilities. Apart of multi-touch, having multiple simultaneous input points can also provide means for bimanual interaction, promoting a richer gesture vocabulary (Fitzmaurice et al., 1995; Fitzmaurice, 1996). Also, rich gestures as such, lighten the cognitive load and help in the thinking process while taking advantage of kinesthetic memory (Shaer and Hornecker, 2010).

Secondly, multiple simultaneous input enables multi-user interaction, in form of multi-user gestures and collaboration.

3.1.2 Collaboration

The social affordances associated with tables directly encourage concepts such as “social interaction and collaboration”(Hornecker and Buur, 2006) or “ludic interaction” (Gaver et al., 2004). Many researchers do, in fact, believe that the principal value of tangible interfaces may lie in their potential for facilitating several kinds of collaborative activities that are not possible or poorly supported by single user technologies (Marshall et al., 2007). A research community has been growing around these technologies and the concept of “shareable interfaces”, a generic term that refers to technologies that are specifically designed to support physically co-located and co-present groups to work together on and around the same content. With vast horizontal table-shaped screens, tabletops seem to be a paradigmatic example of a “shareable interface”.

Until recently, most research on computer-supported cooperative work (CSCW, a term coined by Irene Greif and Paul M. Cashman in 1984 (Bannon, 1993)), has oftenly concentrated on remote collaboration. But, if we restrict ourselves to co-located collaboration, the type of devices used (for example screens versus tables) seem to make a manifest difference.

Rogers and Rodden (2004) have shown for example that screen-based systems inevitably lead to asymmetries concerning the access and the creation of information. While these

systems make it possible for all participants to view the external representations being displayed (for example, through using whiteboards and flipcharts), it is more difficult for all group members to take part in creating or manipulating them. In particular, one person can often dominate the interactions by monopolizing the keyboard, mouse, or pen when creating and editing a document on a shared interactive whiteboard. Once a person is established in a particular role (for example note-taker, mouse controller) she or he tends to remain in it. Moreover, those not in control of the input device, can find it more difficult to get their suggestions and ideas across.

Rogers and Rodden (2004) have done some user studies around interactive tables, for learning the new opportunities for collaborative decision-making that shared interactive tabletops can provide. They conclude that collaborative decision-making can indeed be promoted by providing group members with equal access and direct interaction with digital information, displayed on an interactive table surface. They observe that these interfaces also foment discussion, and that the sharing of ideas and inviting others to take a turn, to respond, to confirm, or to participate, all tended to happen at the same time the participants were interacting with the table, supporting their opinions with gestures and with the table responses.

Some tabletop implementations have even strengthened this collaborative aspect with idiosyncratic design decisions. Such is the case of the collaborative tabletop electronic music instrument *Reactable* (Jordà et al., 2007)(see more in Sections 3.2 and 3.4) or the *Personal Digital Historian System* (Shen et al., 2003), all of which are based on circular tables and use radial symmetry, for promoting collaboration and eliminating head position, leading voices, or privileged points of view and control.

Sharing control in collaboration

All this collaboration with tabletops can be seen as acts of sharing digital data between users, for example in the form of photo collections (for example (Shen et al., 2003; Crabtree et al., 2004)), which probably constitutes nowadays, together with map navigation, the most popular demo for tabletop prototypes (for example Microsoft Surface). Communication in general is definitely about sharing data (Shannon, 2001), but it is not about sharing documents or files—it is about sharing real-time, on-the-fly-generated data.

This idea of sharing control versus sharing data has been becoming more frequent on tabletops. One clear example is *Reactable* (Jordà et al., 2007), which is better described as a contraption for sharing real-time control over computational actions, rather than

3 Exploring tabletops' distinctive affordances

for sharing data among its users.

Although the idea of sharing control can be strongly linked to music performance, tangible applications with a similar philosophy are also becoming more frequent in non performance-related domains. The Patcher (Fernaesus and Tholander, 2006) presents a set of tangible resources for children in which tangible artifacts are better understood as resources for shared activity rather than as representations of shared information. Fernaeus et al. (2008) identify a "practice turn" in tangible interaction and HCI in general, which is moving from a data-centric view of interaction to one that focuses on representational forms as resources for action. Instead of relying on the transmission and sharing of data, the action-centric perspective is looking for solutions that emphasize user control, creativity, and social action with interactive tools.

No matter how influential this paradigm shift may be felt in a near future, the truth is that it is hard to think on shared control when models and inspirational sources come from WIMP based, single-user interactive computer applications. Much of the efforts taken until today in the field of CSCW have been in that direction, trying to convert single-user applications into multi-user collaborative applications. But sequential interaction has proved to be too inflexible for collaborative work requiring concurrent and free interactions (Begole et al., 1999; Stefik et al., 1987; Olson et al., 1992; Sun et al., 2006).

3.1.3 Physicality

Independently to the capability of having simultaneous input actions, and providing a big and shareable interface that drives the previous affordances, the simple fact that table-shaped interfaces have the ability to literally support physical items on them has many consequences. Users can interact with objects of various volumes, shapes, and weights, and when their position and orientation is identified and tracked by the system, the potential bandwidth and richness of the interaction goes thus far beyond the simple idea of multi-touch. While interacting with the fingers still belongs to the idea of pointing devices, interacting with physical objects can take us much farther. Such objects can represent abstract concepts or real entities. They can relate to other objects on the surface. They can be moved and turned around on the table surface, and all these spatial changes can affect their internal properties and their relationships with neighboring objects. They can even add metaphoric parameters (e.g. size, texture, weight... (Hurtienne et al., 2009)).

Objects can have different purposes and meanings which can be used to help the in-

teraction. Different taxonomies have been proposed in order to classify the objects; Underkoffler and Ishii (1999) classify objects depending on whether they are more associated with data or with actions (pure object, attribute, noun, verb, reconfigurable tool). Holmquist et al. (1999); Ullmer et al. (2005) distinguishes between tools, tokens and containers, depending on whether if they can manipulate, access or carry information.

In particular it seems that the use of those objects can have many benefits to problem solving processes:

- Epistemic actions (Kirsh and Maglio, 1994), those physical manipulations intended to understand the task or problem rather than to interact with the system, are then possible with those objects, thus facilitating mental work and helping problem solving (Shaer and Hornecker, 2010).
- Problem solving can be also empowered by physical constraints (Shaer and Hornecker, 2010), which can be possible with objects in tabletops. Physical constraints are confining regions that prevent objects from moving freely, limiting the degrees of freedom of their movement, preventing object positions that would represent unacceptable problem solutions or situations (Ullmer et al., 2005).
- Tangible representations of a problem may contribute to problem solving and planning (Shaer and Hornecker, 2010). This is specially true when the physical properties of the manipulated objects map exactly the ones of the problem to solve. A very good example is Urp (Underkoffler and Ishii, 1999), using building miniatures to plan an urbanization while seeing the consequences of the planning (more in Sections 2.4 and 2.4.1).
- Tangible tabletops seem to facilitate divergent thinking and creative thought, an important factor in problem solving processes (Catalá et al., 2012).

Additionally, to support tangible interactive objects, vast screens favor real-time, multidimensional as well as explorative interaction, which makes them especially suited for both novice and expert users (Jordà, 2008). They also permit spatial multiplexing, allowing for a more direct and fast interaction (Fitzmaurice, 1996) while leveraging the cognitive load (Shaer and Hornecker, 2010). Also, the visual feedback possibilities of this type of interfaces, make them ideal for understanding and monitoring complex mechanisms, such as the several simultaneous musical processes that can take place in a digital system for music performance (Jordà, 2003).

3.2 Existing tabletop applications

Most of the end-user applications in tabletops belong to several tabletop application ecosystems, built around the different hardware platforms. Those platforms are better known than the applications they may hold and run (and that itself could be a sign of the immaturity of the application market). The most disseminated devices are undoubtedly the ones by Microsoft: Microsoft Surface¹ and PixelSense² (see Figure 3.1). These are conceived as a platform to allow third party applications to be run in it: they have a tabletop-ready application management environment working on top of a windows desktop that allows users to instantiate and change applications from the available ones, only one being active. They also offer a set of basic applications and samples (not fully working concepts), but the market is composed mainly by 3rd party applications, which users need to buy and/or download and install by themselves in the underlying desktop environment. As opposed to Microsoft Surface (tablet and desktop) ecosystem³, a centralized application distribution system is still not available for tabletops, a clear sign of the current weakness of the market.

Nonetheless, many software development companies specialize on tabletop applications for Microsoft products. Of course, it is impossible to chart every commercial application offered by a third party as there is not a centralized list, but a quick search for (commercialized on-line) applications results in a profile of the usual application categories that are offered: Point-of-Sale kiosks; collection visualization (such as documents, pictures or videos); casual gaming and music; education and scientific visualization. None of the activities from these software applications seem to center on the collaboration or on leveraging the affordances to solve complex domain specific problems, as one may think it would be its main application.

SMART table⁴ is another of the commercially available tabletop systems that combines hardware and software in commercialization. Its primary focus is education: the embedded software supports loading learning activities designed by SMART and customized by teachers. A runtime execution environment is provided to support 3rd party applications, but there is not a centralized database of 3rd party applications or any other reference to any.

DiamondTouch⁵ is commercialized as tabletop hardware with some software tools in-

¹<http://www.microsoft.com/en-us/pixelsense/supportlinks10.aspx>

²<http://www.pixelsense.com>

³<http://windows.microsoft.com/en-us/windows-8/apps>

⁴<http://education.smarttech.com/en/products/smart-table>

⁵<http://www.circletwelve.com/products/diamondtouch.html>

3.2 Existing tabletop applications



Figure 3.1: Microsoft Surface (left) and PixelSense (right).



Figure 3.2: SMART table

3 Exploring tabletops' distinctive affordances

tended to use desktop applications emulating a mouse, and an SDK to develop custom applications on it. In their website there are no references to any tabletop application, own or third party.

As an exception to this platform-centric focus, although many commercial tabletop devices have existed for many years now (DiamondTouch first publication appeared in 2001, for instance), there are arguably nearly no commercially successful applications. As a sole example of a commercially successful tabletop application there is the Reactable (Jordà et al., 2005), a musical instrument that is sold as a hardware and software bundle⁶.

The Reactable is a musical instrument inspired in modular music synthesizers. A circular tabletop surface (for a more complete explanation of Reactable hardware see Section 3.4) allows performers to place objects that either make sound, transform sound or control other objects or global parameters. By placing these objects near each other, data and audio connections are formed, creating a mesh that produces music and sound. Interaction with fingers allows manipulating the audio stream as well as modifying parameters of the objects (see Figure 3.4).

In fact, the Reactable is also the only commercial application known to the author to use custom tabletop hardware, very popular among low-budget or custom projects, thanks to the various software tools and techniques, large knowledge bases of resources and many easy-to-follow tutorials collected and publicly available to the DIY community (Schöning et al., 2008). Particularly, reacTIVision (Bencina et al., 2005) and Community Core Vision (CCV)⁷ offer very easy to use finger and fiducial tracking for optical based tabletop projects. Its very difficult to keep track of those projects because of its Do It Yourself nature.

In an academic environment, many tabletop applications are presented in conferences every year. If we search for all publications introducing a tabletop application in the most related conferences TANGIBLE AND EMBEDDED INTERACTION (TEI), TABLETOP (tabletop), and INTERACTIVE TABLES AND SURFACES (ITS) since their creation until 2013 (see Appendix B for a complete list), we can see that Microsoft Surface and PixelSense are becoming popular choices, while a slightly decreasing amount of custom solutions (using reacTIVision or CCV) is still present (see Figure 3.5).

The types of tabletop application presented on those conferences (see Figure 3.6) favor in the education setting, being a big part of the application developing effort. Tools (common, professional, coordination) represent another big part of them, believing in

⁶Reactable products web page <http://reactable.com/products/>

⁷<http://ccv.nuigroup.com/>



Figure 3.3: DiamondTouch



Figure 3.4: The Reactable. Photo by Daniel Williams from NYC, USA.

the impact of tabletops on current processes, in order to increase their performance or capability.

3.3 The presented applications

Education is indeed a very common approximation vector not only for commercial and research tabletop applications but for TUI in general. There have been a number of works focused on studying TUIs and their effects on learning (especially in children) in recent years. Marshall (2007) categorizes learning activities with TUIs as two types: exploratory and expressive. These two categories precisely relate to the approaches we explored in order to exploit tabletops affordances. TurTan (see Section 3.5) focuses in an eminently expressive activity: graphical programming, in an exploratory way. By using objects to represent instructions, users create programs by creating a sequence of objects, which can be modified in real time while continuously seeing the graphical result; and, by manipulating several objects simultaneously, they can even be collaboratively edited by several people at the same time.

Data exploration and discovery (very present in apps such as map and collection browsing) is addressed by SongExplorer (see Section 3.6), although not in an educational context. It presents a large collection of songs as a two-dimensional map that can be browsed with regular map-browsing gestures with the intent to help users to discover new, interesting music. This map is constructed by using the analyzed contents of the songs and placing similar songs close together, and visualized in ways that enhance their high-level detected qualities (such as the mood or danceability of the music). This strategy makes use of the everyday orientation and map interpretation skills to explore a completely different space: music.

3.4 Hardware Setup

Before starting to describe the tabletop applications we created to explore those affordances, we will briefly introduce the hardware platform that we used, common in all the presented projects in this thesis.

Analyzing and improving hardware platforms for tabletop computing is outside the scope of this thesis. Instead of building a custom tabletop, a Reactable has been used for convenience. This setup has been used over the years in many projects and has proved to be reliable enough to simply focus on the software side.

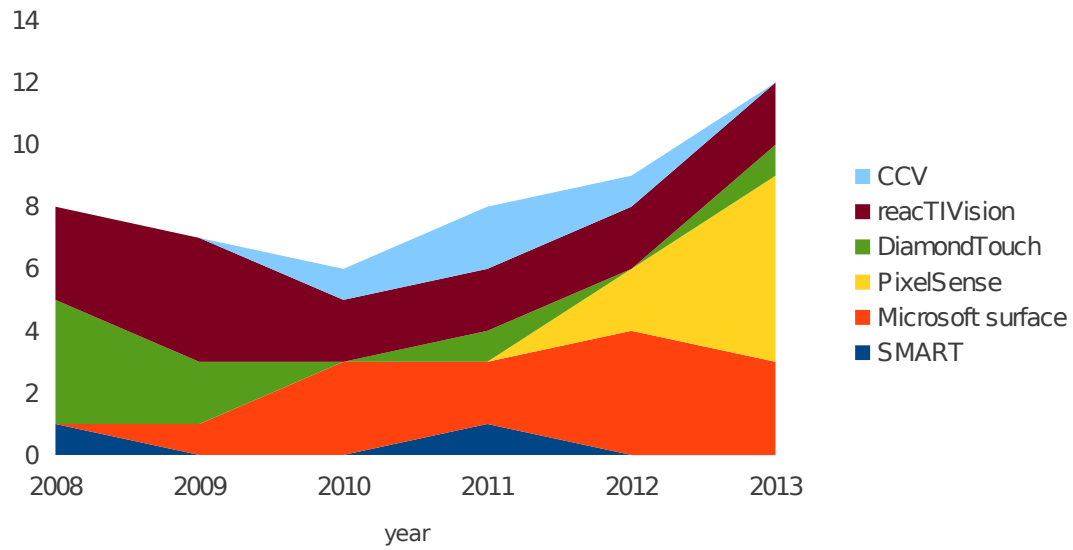


Figure 3.5: Tabletop technologies and devices used in applications presented at TEI, ITS and TABLETOP conferences.

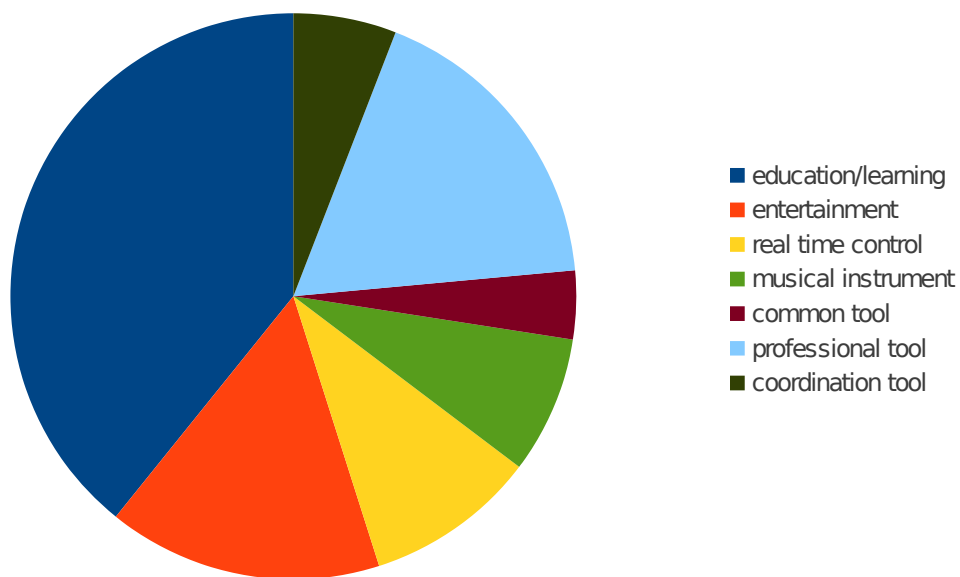


Figure 3.6: Types of tabletop applications presented at TEI, ITS and TABLETOP conferences.

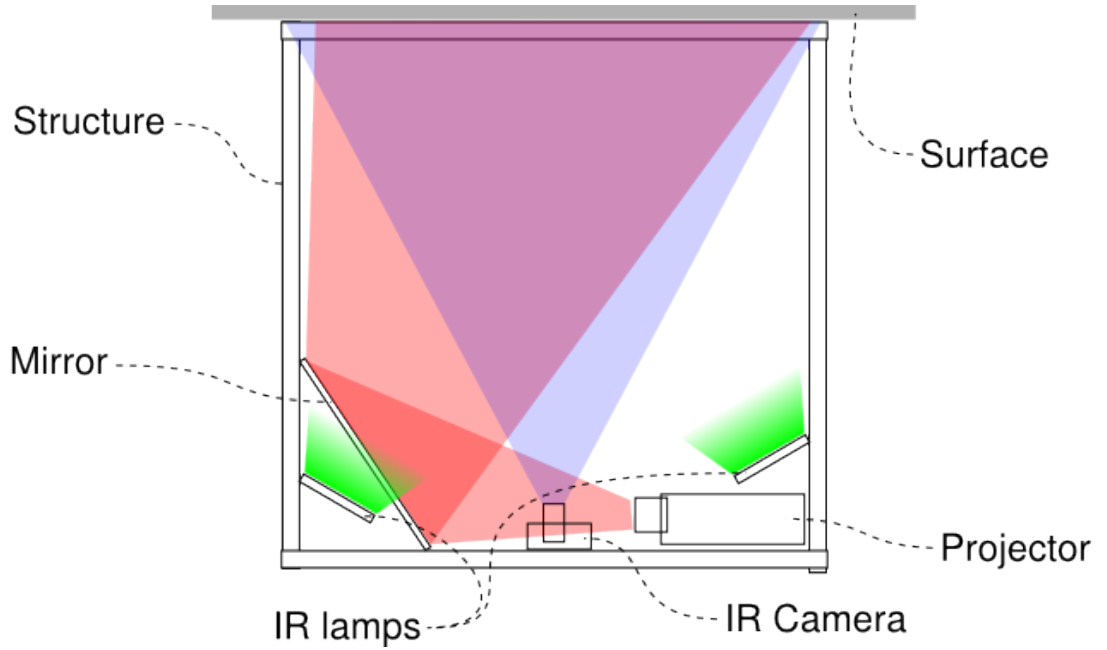


Figure 3.7: Hardware setup

A diagram shown in Figure 3.7 shows the main elements and processes of a Reactable device. It consists of a round horizontal surface of a translucent material seated into a structure, infrared light emitters and camera, and a video projector, which are all underneath the surface. It is about 90cm in diameter, and roughly the same in height.

The round shape of the surface is specially designed to promote collaboration, by erasing any possible privileged orientation and so any predominant role (as we briefly discussed in Section 3.1.2). This was a requirement of the Reactable instrument, meant for collaborative musical performance (Jordà et al., 2005; Jordà, 2008), which also has its consequences on the design of any application running on the device: apart of having to plan for a circular interface, the lack of a predominant position constrains to the appearance of text and other elements that have a correct orientation.

The displayed image is projected from underneath with the projector, optionally using a mirror to maximize the distance and thus the projected area. As the surface is translucent it serves as the projector screen, displaying the image. The distortion created by the projection (and aggravated by the mirror) has to be corrected by software (see 4.1.1).

The technique used for finger and object detection is Diffused Illumination (Schöning et al., 2008) (see Section 2.4.1): infrared (IR) LED lamps⁸ illuminate the elements

⁸The design of the lamps has evolved over time, from LED matrices to high power SMD LEDs

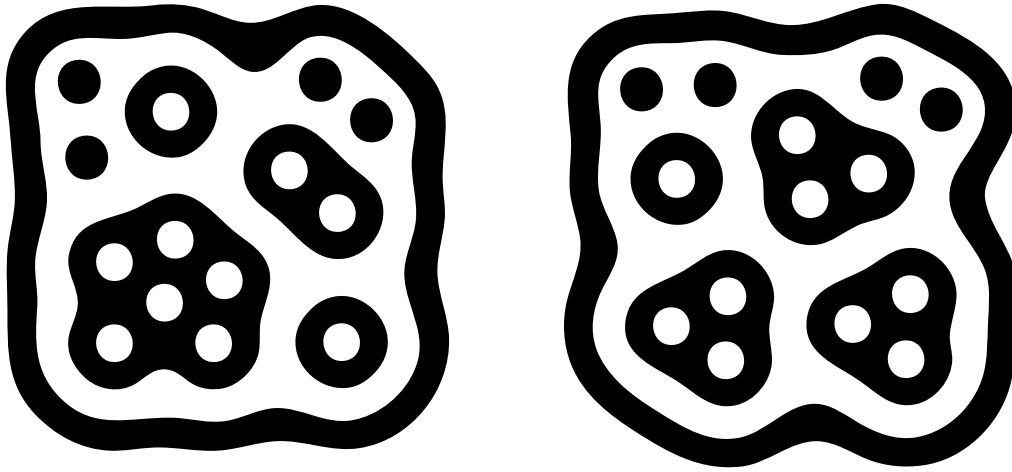


Figure 3.8: reacTIVision fiducial markers

beyond the surface, while an IR camera records the image as seen through the surface. The infrared spectrum is used instead of visible to avoid interferences with the projected image.

Because of the translucent diffused surface, only objects that are close enough ($\sim 1cm$) can be seen in the camera's image, while more distant ones remain invisible, so the tracking software can detect objects only in contact with the surface and not hovering. This is very useful to detect finger contact, as fingers touching the surface appear as small circles while hiding the rest of the hand.

ReacTIVision (Bencina et al., 2005) is used to track the fingers and objects using the image from the camera. Objects must be tagged with special fiducial markers in order to be recognized. Those are configurable and can have arbitrary sizes and shapes, though the set already provided with the software is usually convenient (see Figure 3.8). Finger interaction, in contrast, does not need any additional marker or apparatus in order to be detected.

ReacTIVision processes the image from the camera and sends the identified finger and object information to the actual program using the TUIO protocol (Kaltenbrunner et al., 2005), which is the de facto standard in low level gesture input protocol, based on Open Sound Control (OSC) over UDP, and specifically designed to simplify the communication between processes in a tangible user interfaces environment. This way the tabletop application does not have to implement any tracking mechanism and will receive all processed events detected on the surface just by implementing a TUIO client. The

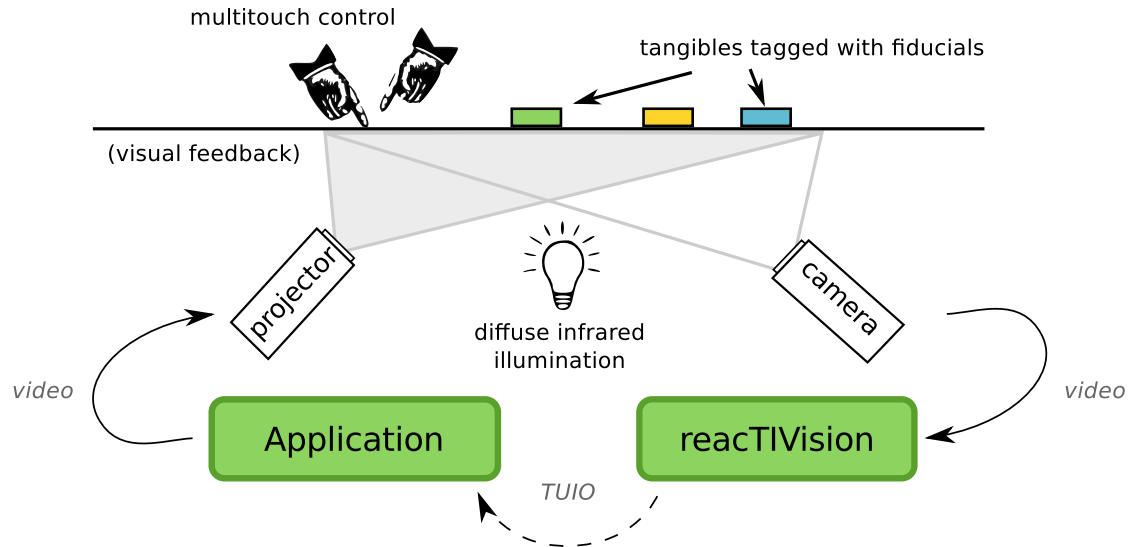


Figure 3.9: Tracking setup

whole process is pictured in Figure 3.9.

3.5 TurTan, a tangible tabletop programming language for education

One of the major research targets involving tabletops and tangibles is enhancing the education process. The possibility of affordances such as expressivity, exploration and collaboration, making the tangible version of a particular activity better suited for learning, is particularly interesting (Shaer and Hornecker, 2010). In parallel, the challenge of creating a tabletop-based tangible programming language was particularly appealing, as we envisaged that the directness of tangible interaction would essentially change the way users would relate to the activity, into a more exploratory one.

With TurTan (Gallardo et al., 2008) we approached these two big subjects: tangible programming languages and tabletops in a learning context.

TurTan implements a tangible programming toy language (Ledgard, 1971), especially conceived for children and non-programmers, which may help to introduce and explore basic programming concepts in a playful, enjoyable and creative way. It is inspired in Logo (Feurzeig et al., 1969), and thus designed for performing *turtle geometry* (Abelson and Di Sessa, 1986). As the original Logo language, one of the TurTan design goals was to use it for teaching some programming concepts, mainly to children. Unlike the

original Logo, TurTan is totally controllable and manipulable in real-time, meaning that there is no distinction between coding and running the program, everything happening simultaneously. In that sense it could also be considered as a *Dynamic programming language*.

3.5.1 TUI in learning

Historically, children have been encouraged to play with physical objects (individually and collaboratively) such as building blocks, shape puzzles and jigsaws to learn a variety of skills. When Friedrich Froebel established the first kindergarten in 1837, he used the technology of the time to develop a set of toys (known as “Froebel’s gifts”) to help young children to learn basic concepts such as number, size, shape, and color through their manipulation (Brosterman and Togashi, 1997).

Later, Montessori (1912) observed that young children were attracted to and spontaneously used sensory development apparatus independently, repeatedly and with deep concentration. While Froebel’s theory encouraged children to do more practical work and directly use materials that exist in the real world, Montessori’s method supported the belief that children should follow their inner nature, and learn by following an autonomous development (Zuckerman et al., 2005).

There is significant research evidence supporting that physical action is important in learning. Piaget and Bruner showed that children can often solve problems when given concrete materials to work with before they can solve them symbolically (Bruner, 1966; Piaget, 1953). Constructivism, introduced by Piaget, according to which individuals learn from their experiences (Piaget and Jean, 1999), states that “knowledge and the world are both constructed and interpreted through action, and mediated through symbol use” (Ackermann, 2004). Piaget’s theory sees children as individuals capable of rational thinking and approaching scientific reasoning.

Based on the foundation of Constructivism, Papert’s Constructionism (Papert and Harel, 1991), according to which children learn through making things, puts emphasis on providing children with technologies with which they get to be authors, rather than experiencing worlds and situations that are pre-scripted, or absorbing facts provided by a computer (Papert, 1993).

Papert’s constructionism lead to the invention of the Logo programming language (Feurzeig et al., 1969). In Logo, a program consists of a series of instructions that “tell a virtual turtle what to do”. This turtle, which is in fact a cursor, has a position and an orientation and will leave a trace when moving. A series of instructions relate to different

3 Exploring tabletops' distinctive affordances

available movements (such as FORWARD or RIGHT) while others relate to control flow (such as REPEAT).

As an example, drawing a (50-units side) square in Logo could be done with REPEAT 4 [FORWARD 50 RIGHT 90]. The turtle would move 50 units while leaving a trace and rotate to the right 90 degrees 4 times, leaving a square on the screen.

Resnick was the first to bridge these concepts by bringing the possibilities of digital technologies and applying them to educational domain, coining the term 'digital manipulatives', which he defines as familiar physical items with added computational power, aimed at enhancing children's learning (Resnick, 1998; Resnick et al., 1998). Zuckerman et al. (2005) create their own categorization of learning by referring to the schools of Froebel and Montessori, and to what they call "manipulatives". They classify them as two types: Froebel-inspired Manipulatives (FiMs) and Montessori-inspired Manipulatives (MiMs). "FiMs are design materials, fostering modeling of real-world structures, while MiMs foster modeling of more abstract structures". Marshall (2007) categorizes learning activities with TUIs as two types: exploratory and expressive, and argues that TUIs "can help learners to make their ideas concrete and explicit, and once externalized, they can reflect upon how well the model or representation reflects the real situation".

Educational TUI example projects are the TICLE (Tangible Interfaces for Collaborative Learning Environments) (Scarlatos, 2002), where children study basic math and geometry concepts, or Read-It (Weevers et al., 2004), used to learn how to read.

3.5.2 Tangible programming languages

Many artist-oriented programming languages exist and are used everyday. Most of them are still text based, like *processing*⁹ or *actionscript*¹⁰, while others such as *MAX*¹¹ or *Pure Data*¹² use visual paradigms. All of them do still use mouse and keyboard to program, and are more conceived for the creation of interactive art pieces, than for the intrinsic exploration of programming creativity.

Beyond the PC, other languages use tangibles to program. This is the case of the tangible programming systems *Quetzal* (Horn and Jacob, 2007) and *AlgoBlock* (Suzuki and Kato, 1993): while the first uses objects to embody the meaning of the instructions (tangibles are just representations), the second embeds electronics on the objects that actually perform the commands themselves. However, these languages lack a direct real

⁹<http://www.processing.org/>

¹⁰<https://www.adobe.com/devnet/actionscript.html>

¹¹<http://cycling74.com/products/max/>

¹²<http://puredata.info/>

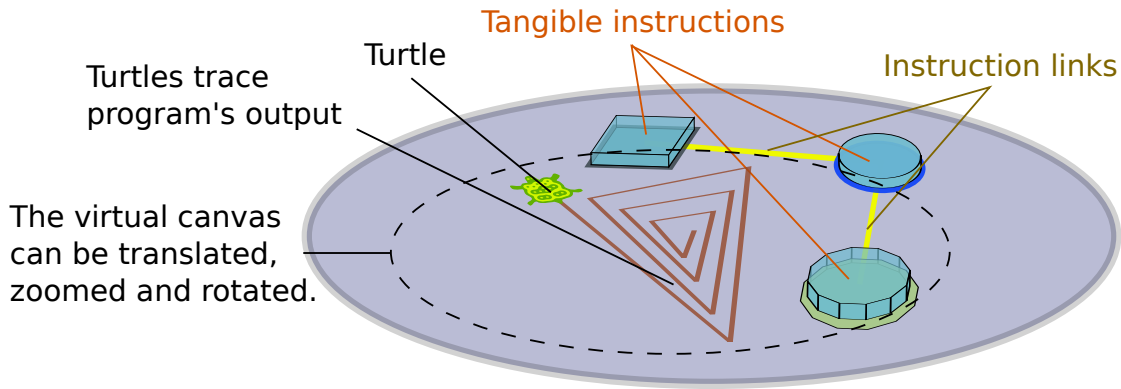


Figure 3.10: TurTan elements.

time control of their parameters, which makes fluid interaction between the programmer and the program difficult to reach.

A more creative-oriented real-time programming language using tangibles is *Task Blocks* (Terry, 2001), which consists of tangible instructions (named task blocks), physical controls and physical connectors. *Task Blocks* is a language for data-processing in which processing modules are programmed connecting task blocks together, while its parameters are controlled by physically wiring the controls to these blocks. This structure allows to have real time control over the program's parameters and structure.

Another example of tangible programming, is the Robot Park exhibit, which is a permanent installation at the Boston Museum of Science (Horn et al., 2009).

3.5.3 TurTan interaction mechanics

When beginning a session, TurTan users find an empty interactive surface, only with a virtual turtle on the center. Various plastic objects are placed around this surface, tagged with symbols representing their function. To program, users will have to place those objects on the surface, creating a structure that will represent the program sequence. The output of this program, in the form of vector graphics, will be continuously displayed and updated in the background, in real time (see Figure 3.10). This is one of the key components of TurTan as a dynamic programming language: there is no compiling or executing steps, the result is instantaneous, so the manipulation of the program becomes a real-time interactive activity.

While users create the program by placing and manipulating the objects, the graphical output of the program can be manipulated using hand gestures. By using only one finger, the graphical output can be translated; by using two fingers it can be rotated and scaled

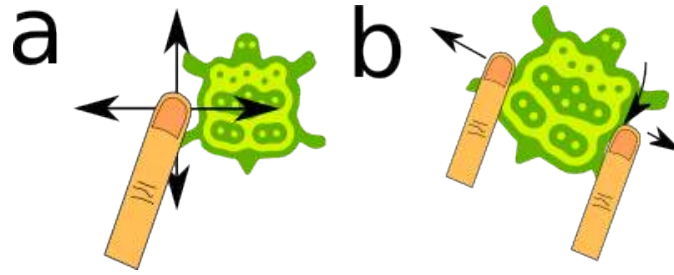


Figure 3.11: Scale, rotate and translate with fingers.

(see Figure 3.11). These are already de facto standard gestures for 2D transformation in multi-touch devices. When users want to return to an untransformed state, they can remove all the objects off the surface, and the output will return to its original state, with the turtle in the center of the surface.

This separation of the interactive elements between the program editor and output is created on purpose, to enforce the sense that they are two independent activities: editing the program will not alter the output transformation nor the other way around, making it safe to engage in either of the activities when someone is already working in the other one. By using different “interaction layers” to present two activities in the same space, and not creating separated areas, we do not sacrifice valuable space nor define privileged positions for each activity.

Creating programs

In order to create a program, which is in fact a sequence of instructions, users need to establish links between the physical objects and to control their order inside the sequence. Links are represented as virtual lines between the objects, and are automatically created by the system depending on its proximity with other objects. Depending on the type of the object, two different strategies to create links are used in TurTan: dynamic linking and static linking.

By using static links (Figure 3.12 (i→ii)), the position of an object inside the sequence is decided only once, when the tangible is first placed on the surface. Any later modification of its relative distances with other objects (by moving or manipulating a linked object) will not affect its position within the sequence.

In contrast, with dynamic linking the order inside the program sequence may be modified whenever the relative distances are changed, so that by moving and manipulating any instruction object on the table it might actually change its position within the sequence

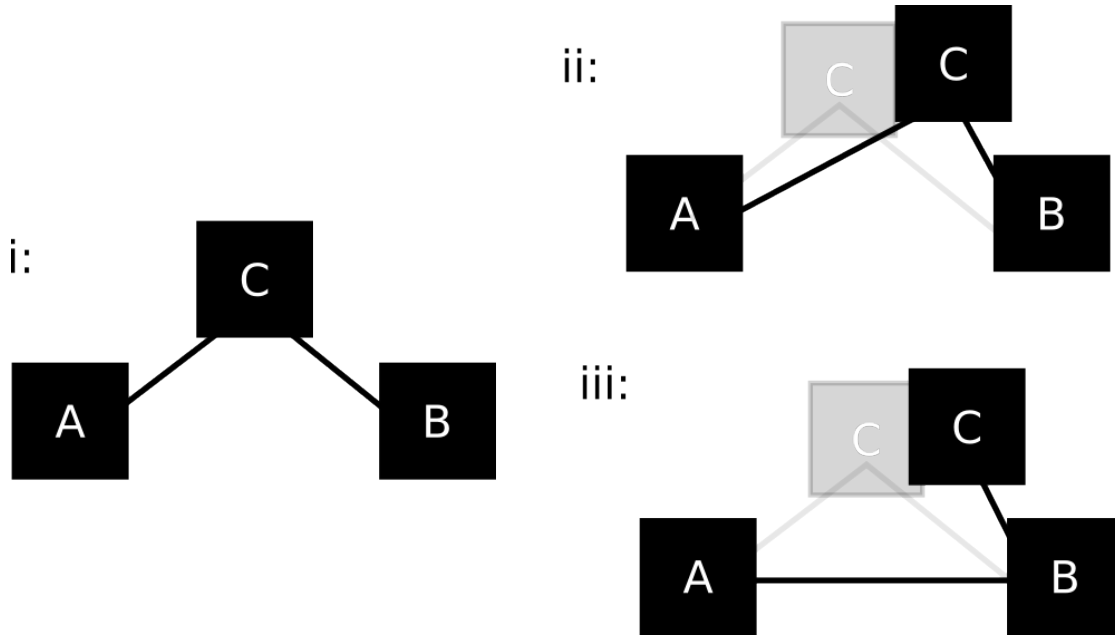


Figure 3.12: Static (i→ii) and dynamic (i→iii) linking approaches.

(see Figure 3.12 (i→iii)).

Dynamic linking, which allows to swap positions easily between two elements, is a good choice for applications where the order is not so relevant. Also, when there is a large number of objects present on the table, the resulting sequence might be rather complex and the user may have difficulties for anticipating where the tangible will be linked at the moment of introducing it. Dynamic linking therefore allows for easier adjustments of the resulting sequence, while static linking provides a better flexibility for manipulating the tangibles on the table without the danger of destroying the correct order.

In TurTan, objects in the program sequence are linked using static linking, as the order of the instructions in a program is extremely sensitive to its meaning. Other less essential objects (such as modifiers) use dynamic linking to profit from its versatility.

Object parameters

Objects may have parameters that users might want to change. For instance the “move painting” instruction needs an “amount of movement” parameter to be defined. Those parameters are defined by spinning the object on the surface.

Because of the different possible scales of the parameter, we noticed that a linear mapping from the amount of instantaneous rotation $\Delta\alpha$, and the variation of the object parameter

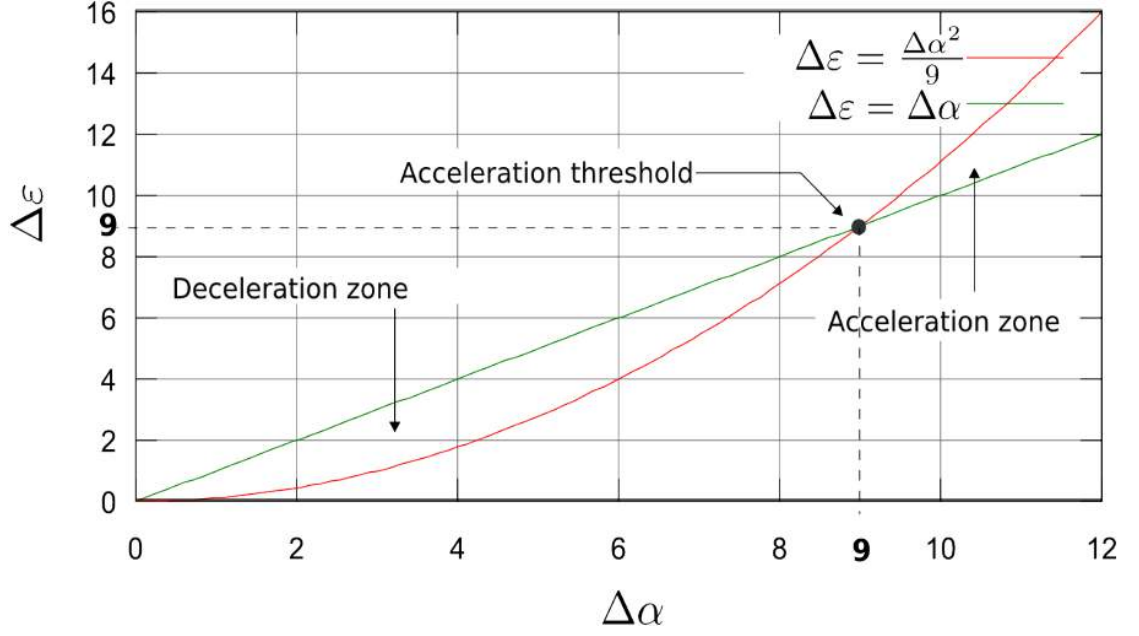


Figure 3.13: Non-linear scaling of angle.

$\Delta\epsilon$ in the form of $\Delta\epsilon = k\Delta\alpha$ implied having to trade between high precision (wasting time spinning the object to get a higher value) or rudeness (making it difficult to be accurate).

Instead, a parabolic mapping in the form of $\Delta\epsilon = \text{sign}(\Delta\epsilon) k\Delta\alpha^2$ proved being much more effective providing both high precision for and roughness. The difference can be seen in Figure 3.13. As you can see, the border between the deceleration zone and the acceleration zone is $\frac{1}{k}$.

3.5.4 TurTan language, as an example of a Tangible Programming Language

The TurTan language and goal is based on Logo. It can be defined by the types of objects that can be used to create a program and the syntax or possible relationships between them. It consists of three different object types: *instructions* (equivalent to Logo instructions), *modifiers* and *cards*.

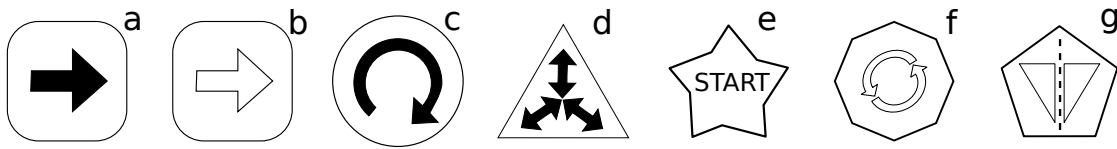
Instructions

As Logo, TurTan is an imperative programming language, and therefore, it places a great importance on instructions and their order (sequence). Every instruction object repre-

sents an atomic action that the turtle will perform. This action can have consequences on the output (can leave a trace) and in the turtle's state (can transform the turtle). As instructions output depend on the turtle internal state (position, orientation, size) and they can change this state, the order of execution of the instructions is extremely relevant to get the correct output.

Instructions are placed on the surface and become a sequence by linking them together using static linking, otherwise, the object's position on the table is not relevant. The linking process creates a single ordered chain of instructions. The direction is defined by the order in which the objects were placed on the surface: when the second object is placed, it becomes the last in a sequence of two, beginning with the object that was placed first. All other future objects are linked to this ordered sequence (at the beginning, in the middle, or at the end) without changing the direction.

The provided instructions are:



- a) Move painting** The turtle moves forward or backward leaving a line trace.
- b) Move without painting** The turtle moves forward or backward without leaving a trace.
- c) Rotate** The turtle turns right or left.
- d) Scale** The turtle scales its body size and also paints its following trace distances at a different scale.
- e) Return to start** The turtle returns to its original state at the beginning of the sequence, in the center.
- f) Repeat** The turtle repeats all the actions since the beginning of the sequence as many times as indicated by the parameter.
- g) Mirror** The turtle mirrors herself so instead of rotating right it rotates left and vice versa. Mirroring a second time cancels the effect.

See Figure 3.14 for an example using only instructions.

Modifiers

Instead of representing instructions or sequences, *modifiers* dynamically change the parameter of instruction objects. This idea is very similar to the Reactable controllers

3 Exploring tabletops' distinctive affordances

(Geiger and Alber, 2010).

Modifiers can embed a periodic function, such as an oscillator, or serve as a means of control from external programs, using OSC messages. Using modifiers, users can achieve automatic animation of the drawings. One of the possible applications of this OSC communication is animating TurTan programs at the rhythm of music and sound, and using it as a VJing tool by analyzing the sound in real time (see Figure 3.15).

Unlike instructions, modifiers are linked using dynamic linking (see Section 3.5.3), and they are not on the program sequence of instructions, but instead link directly to the neighboring objects, affecting the parameter of the closest one. This means that moving around a modifier may eventually change its target.

By rotating a modifier, users can change the amplitude of the modification, which is selecting the amount of effect that will perform the modifier (oscillator or external) on the parameter of the instruction.

See Figure 3.16 for an example of a TurTan program using a modifier to create automatic animations.

Cards

Laminated paper *cards* can be used in TurTan for storing and running programs. The idea is using cards to store the current sequence of instructions and modifiers into an object, which can be used later to recover the program.

The idea of using objects as information containers in order to transfer and keep data was already introduced in mediaBlocks (Ullmer et al., 1998). We use cards to not only keep and transfer information, but to label it, as users can write or draw on one side of the card with a marker pen, to remember its contents.

To save a program into a card, users must place that card on the surface, away from the instruction sequence, and rotate it 180 degrees to confirm a possible overwriting. Once recorded, a preview of the program will be displayed next to the card. Later, when the surface is empty, a card can be placed anywhere on the table and the program will be loaded and displayed.

The most interesting way of using cards is not only to store and load programs, but to use stored programs as functions. In fact, cards are also instructions, and can be part of another program. For instance, we can create a program that draws a square, save it into a card, and use that card as a building block of a circle made of squares.

See Figure 3.17 for an example of a program using a card as function. Note that the

3.5 TurTan, a tangible tabletop programming language for education

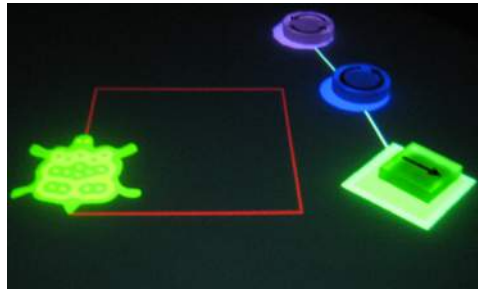


Figure 3.14: Simple TurTan program example: a square.

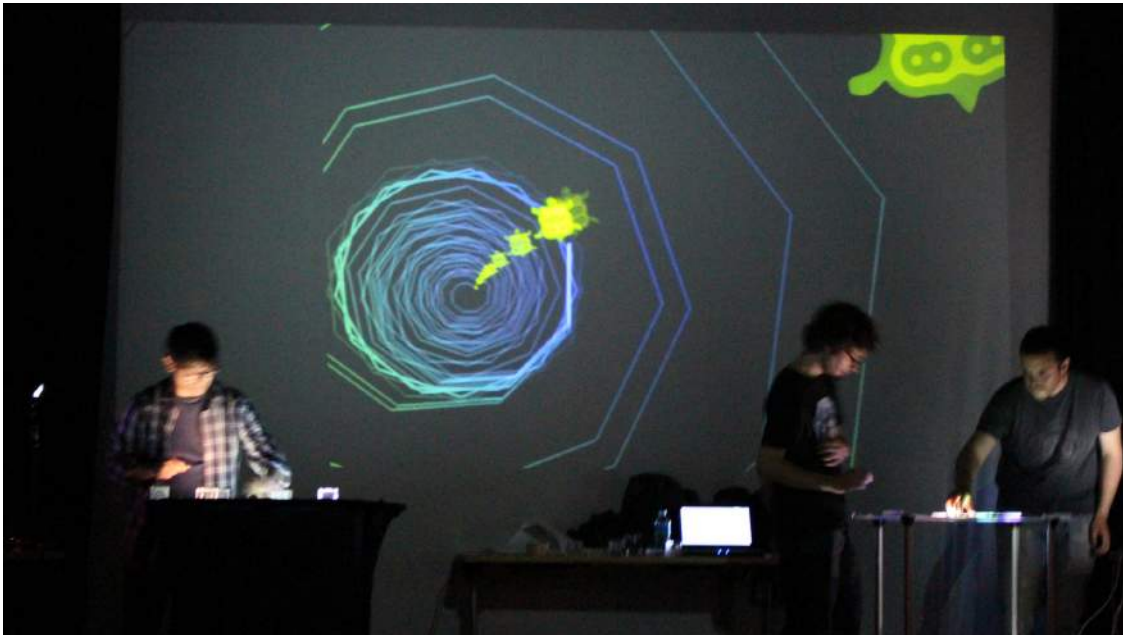


Figure 3.15: TurTan used as a VJing tool.



Figure 3.16: A TurTan program using a sinusoidal modifier to dynamically change the parameter of a *rotate* instruction of a spiral.

program saved into the card ends with a *return to start* instruction.

3.5.5 Is TurTan better than Logo in an education context?

To find out if TurTan had any kind of advantage over Logo, due to its Tangible interface compared to graphical interfaces, we decided to perform an experiment in the context of an education institution, with children of a suitable age (Markova, 2010).

Experimental setup

The study was conducted at a public primary school in Barcelona, Spain, with 24 students from 5th grade (10 year old), because that is the age children should be familiar with most basic geometry concepts, also as recommended by the Montessori Method. The class was divided into eight groups, three students each, to promote collaboration.

To compare the two environments, we used a Catalan version of FMSLogo¹³ running on a portable computer and TurTan, running on a tabletop interface (Reactable), to compare their learning progress. Groups 1 to 4 started learning TurTan first (TurTan, Logo), and groups 5 to 8 started learning Logo first (Logo, TurTan), during one whole week. Halfway through the study, the groups swapped subjects and were tested in the other condition.

We taught five identical or similar lesson sessions for both interfaces, that were approximately 30 minutes long, and included a short revision section. The lesson plans included learning concepts ranging from drawing a rectangle to drawing flowers. The commands that were taught both in TurTan and in Logo were Forward, Right, Left, Back, Clear Screen, Pen Up, Pen Down, Start, Repeat and Procedures. All the sessions were recorded on video and photographs of the progress of each group were taken.

A short individual final task was required to the students at the end of the study, where they were asked to perform similar tasks in TurTan and in Logo. In it, their activity was timed and recorded on video and screen activity (in the case of Logo). The students were informed at the beginning that they had 2.5 minutes to complete the task, although the time limit was never strictly enforced, because we wanted to compare the actual time it took each student to complete the task. The final task included drawing stairs or a square (as a second option), and the students were encouraged to use the Repeat command of both user interfaces to save time and to test understanding of the programming concepts.

¹³<http://fmslogo.sourceforge.net/>

Results

Data from 13 students (2 males and 11 females), who were present at the final task, was collected for analyzing learning speed and difficulty. The data was evaluated both by individual results and as a within-subjects experiment with two conditions - TurTan studied first (TurTan, Logo) or Logo studied first (Logo, TurTan). This was done to check if learning one affected learning the other.

The variables used to compare results between the two interfaces and different conditions were time, steps and errors made during the course of training, as measures used to study learnability, according to Albert and Tullis (2008). This data was collected and analyzed from 44 hours of video recorded during the training sessions and the final task.

The time variable is defined by the time from the student commencing the task until saving her work. That also includes all the attempts that the student made to complete the task. An error is noted when removing all the pucks from the tabletop in TurTan or cleaning the screen in Logo to commence the task anew, as well as when having an unsatisfactory result (e.g. wrong geometrical figure). A step is considered to be a puck placed on the tabletop in TurTan, or a command written in Logo, excluding measurements or pressing the Space or Enter keys.

On the final task, 85% of students completed their task with TurTan faster than with Logo (11 students out of 13). The average task completion time per student for TurTan and Logo was 90 seconds and 213 seconds, respectively. 77% of students used the Repeat command with TurTan and 69% used it with Logo. Figure 3.18 shows the different task completion times per student and per subject on the final task, as well as if the student used the Repeat command. Using the Repeat command is important because it shows a more systematic work (by simplifying the command sequence), and the final result is usually achieved faster - students use less steps both in Logo and in TurTan. It also shows a more advanced level of programming skills because it shows that the students understand the commands they are using, since using the Repeat command requires some thinking ahead.

On average, the final task was completed in less than half the time with TurTan. In terms of errors, the students made only 2 errors in total during their TurTan task, but they made 41 errors in total when using Logo (only 2 resulting from typing errors). A possible explanation of why the students completed their task faster with TurTan could also be the fact that they did not have to make as many attempts, and go through as many steps, to arrive at the final satisfactory result.

There is a significant difference between the completion time for the final task with

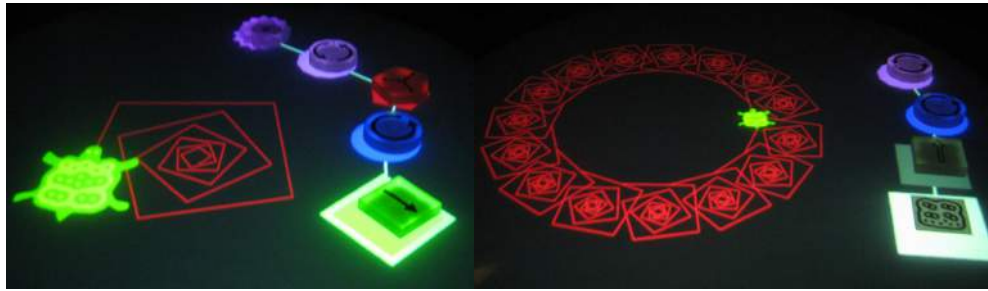


Figure 3.17: A TurTan program using a function card. In the left picture, a simple program that will be saved into a card. In the right picture, a program using the card as the basis of the drawing.

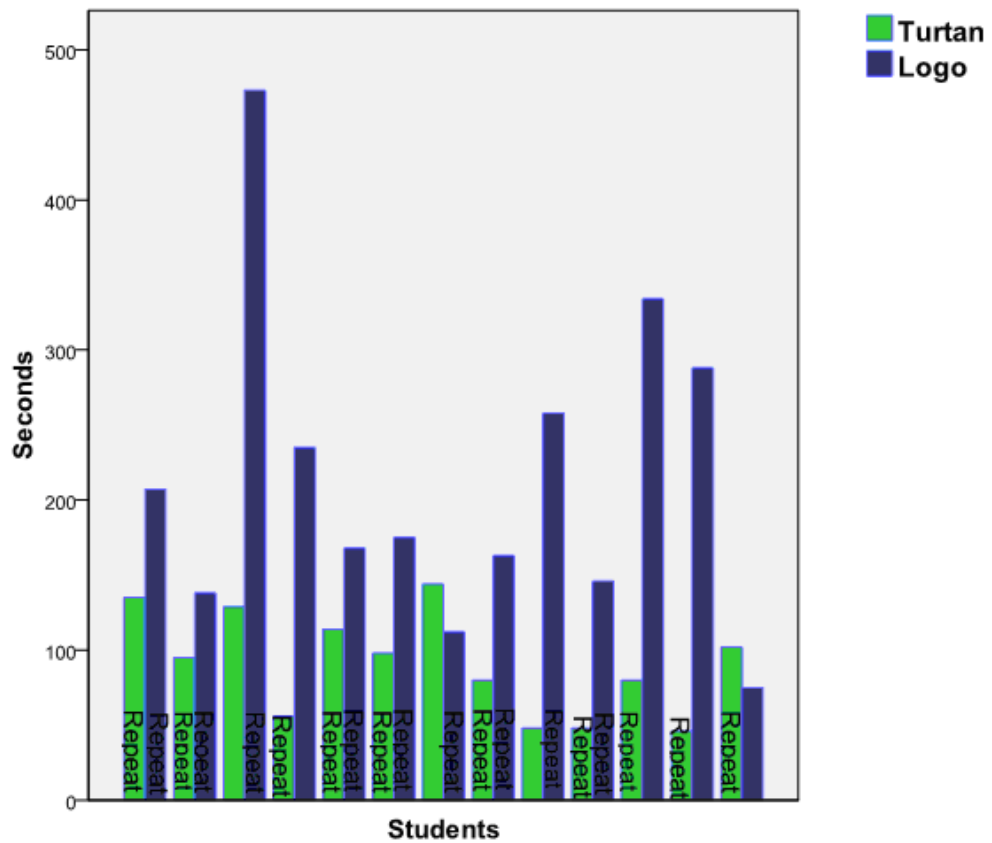


Figure 3.18: Final task completion results. For each student the completion time for TurTan and Logo are plotted. *Repeat* marks the tests where the Repeat instruction was used.

3.5 *TurTan, a tangible tabletop programming language for education*

TurTan and Logo, as well as the amount of errors made using both interfaces (the Paired-Samples T Test shows 0.2% significance in time and 0.4% significance in errors - within the 5% confidence interval). The difference in steps (efficiency) is not significant, therefore this leads us to conclude that, in this case, it is not a relevant variable to determine learnability, as described by Albert and Tullis (2008). We can state that using TurTan results in shorter times and fewer errors than Logo.

Relationship between learning orders

When analyzing learning progress, we compared values from the final task with values from the the second session, for both TurTan and Logo. In the second session, the students were learning how to draw squares, stairs and castles, which are comparable to the final task.

Figures 3.19 and 3.20 show the difference in the relationship between earlier and final sessions of TurTan and Logo. On average, in TurTan, students slightly improved their performance (performed the task faster) in the final task in the Logo-TurTan condition. On the other hand, in Logo, in both conditions, (Turtan, Logo) and (Logo, Turtan), they performed the final task slower than in the earlier sessions, therefore did not improve their performance. It is clear that the students performed their TurTan final task faster in the Logo-TurTan condition because TurTan was the interface they had studied immediately before the final task. However, no such improvement was observed with Logo in either condition however, even when they had studied Logo immediately before the final task. In fact, students took more time to complete their tasks in Logo on the final task (almost twice as much on average), even when they had studied TurTan first and then Logo, and their memory of it was more recent.

The two T Test tables (Table 3.1 and 3.2) show a significant difference between time and steps made with Logo and steps with TurTan in earlier and final sessions. There was no significant difference between final task completion time between earlier and final sessions of TurTan, but a small improvement can be observed in the (Logo, Turtan) condition (Figure 3.20). We can state then that in Logo, the final autonomous task is more difficult (because of greater time, steps and errors) than the guided session, while in TurTan, although there tend to be more steps involved, the resulting time is not affected.

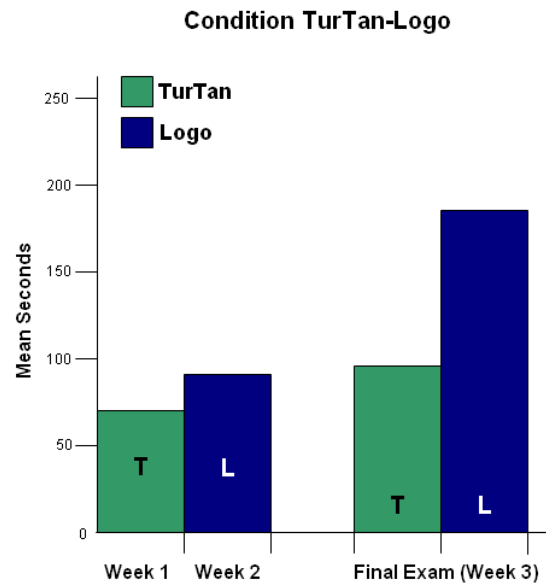


Figure 3.19: Completion time comparison between earlier and final sessions in the (TurTan, Logo) condition.

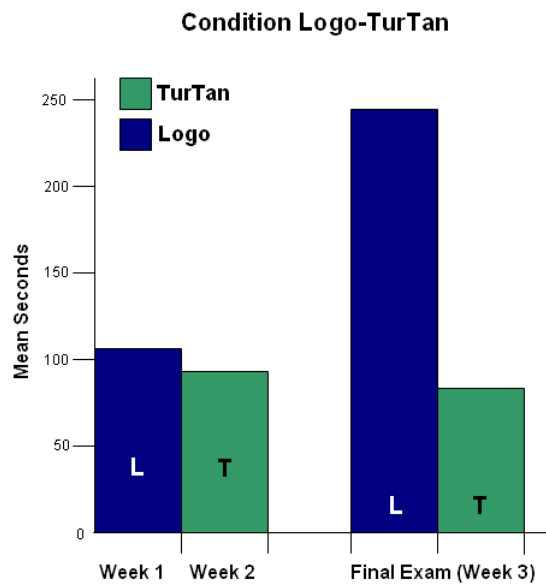


Figure 3.20: Completion time comparison between earlier and final sessions in the (Logo, TurTan) condition.

3.6 SongExplorer, a tabletop for song database exploration

Variable	Mean Initial	Mean Final	Significance
Logo time	$\bar{X}_i = 99$	$\bar{X}_f = 213$	$p = 0.006$
Logo steps	$\bar{X}_i = 5$	$\bar{X}_f = 12$	$p = 0.007$
Logo errors	$\bar{X}_i = 0$	$\bar{X}_f = 3$	$p = 0.003$

Table 3.1: T Test significance values between Logo earlier and final results

Variable	Mean Initial	Mean Final	Significance
TurTan steps	$\bar{X}_i = 5$	$\bar{X}_f = 9$	$p = 0.017$

Table 3.2: T Test significance values between TurTan earlier and final results

Conclusions

The analysis of the videos and on-screen activity taken during the study shows that the students performed faster with TurTan than with Logo, made less errors and less steps to complete the final task. A significant difference between final task completion time and errors leads us to conclude that TurTan takes less time to learn than Logo. Furthermore, the final task results show that, on average, students performed their final task twice as fast with TurTan compared to their results with Logo.

The results also showed that there was a small improvement in completion time with TurTan when the final task was compared to earlier sessions in the Logo-Turtan condition (when TurTan was studied immediately before the final task), while no such improvement was shown in the Logo results. On the contrary, students took almost twice as much time on the final task with Logo than in earlier sessions.

Additionally, the students always tried to work collaboratively, which supports the statement that TUIs are a collaborative type of technology, as shown in the TICLE (Scarlatos, 2002) and Read-It (Weevers et al., 2004) projects mentioned earlier.

3.6 SongExplorer, a tabletop for song database exploration

As seen earlier, many tabletop applications address the exploration of maps or collections of items. With SongExplorer, we wanted to combine such promising modes, by searching within an item database represented as a map (Julià and Jordà, 2009).

SongExplorer is a system for the exploration of large music collections on tabletop interfaces. It addresses the problem of finding new interesting songs on large unfamiliar music databases, from an interaction design perspective. Using high level descriptors of musical songs, SongExplorer creates a coherent 2D map based on similarity, in which

3 Exploring tabletops' distinctive affordances

neighboring songs tend to be more similar. This map can be browsed using a tangible tabletop interface. As noted before, it was proposed (Jordà, 2008) that tangible tabletops can be especially adequate for complex and non-task oriented types of interaction, which could include real-time performance, as well as explorative search. This topic (N-Dimensional navigation in 2-D within tabletop interfaces) was never addressed before.

With the popularization of the Internet and broadband connections, the amount of music which we are exposed to has been increasing permanently. Many websites do offer very large collections of music to the user, either free of charge (e.g. Magnatune¹⁴, Jamendo¹⁵) or on a fee-paying basis (e.g. iTunes¹⁶, The Orchard¹⁷). Such a number of available and still undiscovered music records and songs seems too difficult to manage in a sorting and searching-by-keyword way. This is known as the Long Tail Problem (Celma, 2010). In order to solve this problem and help users to discover new music, many online music recommendation services exist (e.g. Pandora¹⁸, Last.fm¹⁹). One of the main drawbacks of most current music recommenders, independently of the recommendation mechanisms and algorithms they employ (user profiling, experts-based knowledge, statistical models, etc.), is that they apply information filtering techniques to the entire collections, in order to extract and display only a subset of songs that the system believes the user could enjoy. By doing it this way, the user loses the opportunity to discover many new songs which are left out by the system, whatever the cause may be.

To solve this problem, we proposed to construct maps of the entire collection of songs and allowing users to explore them in novel ways. Maps are widely used to explore spaces and also concepts. Although most commonly used to depict geography, maps may represent any space, real or imagined, without regard to context or scale. We use conceptual maps to discuss ideas, we organize data in 2D spaces in order to understand it, and we can get our bearings using topographical maps. SongExplorer's maps are constructed using Music Information Retrieval (MIR) techniques that provide the high-level descriptors needed to successfully organize the data; they do not filter or hide any content, thus showing the complete collection while highlighting some of the songs' characteristics.

Therefore, SongExplorer provides intuitive and fast ways for promoting the direct exploration of these maps. Several successful projects have shown that tangible, tabletop and

¹⁴<http://www.margatune.com>

¹⁵<http://www.jamendo.com>

¹⁶<http://www.apple.com/itunes/>

¹⁷<http://www.theorchard.com>

¹⁸<http://www.pandora.com>

¹⁹<http://www.last.fm>

multi-touch interfaces exhibit useful properties for advanced control in general (such as continuous, real-time interaction with multidimensional data, and support for complex, skilled, expressive and explorative interaction) (Jordà, 2008) and for the exploration of two-dimensional spaces in particular (Han, 2006). Following this trend, SongExplorer allows users to interact with the maps directly with their hands, touching the surface with their fingers and manipulating physical tokens on top of it.

3.6.1 Visualization of music collections

In the field of visualization, there is extensive bibliography on the representation of auditory data. In the particular case we are focusing on, that of the visual organization of musical data, solutions often consist in extracting feature descriptors from sound files, and creating a multidimensional feature space that will be projected into a 2D surface, using dimensionality reduction techniques.

A very well known example of this method is the work *Islands of Music* by Pampalk (2003), which uses a landscape metaphor to present a large collection of musical files. In this work, Pampalk uses a Self Organizing Map (SOM) (Kohonen, 2001) to create a relief map in which the accumulation of songs are presented as the elevation of the terrain over the sea. The islands created as a result of this process roughly correspond to musical genres (see Figure 3.21). A later attempt to combine different visualizations on a single map was also created by Pampalk et al. (2004). By using different parameters to organize the SOM, several views of the collection were created. The results were later interpolated to construct a smooth parameter space affecting the visualization.

Beyond the 2D views, *nepTune*, an interactively explorable 3D version of *Islands of Music* supporting spatialized sound playback (Knees et al., 2007). Leitich and Topf (2007) describes a *Globe of Music*, which distributes the songs on a spherical surface, thus avoiding any edge or discontinuity.

A different and original visualization approach is chosen in *Musicream* (Goto and Goto, 2005), an interesting example of exploratory search in music databases, using the search by example paradigm. In *Musicream*, songs are represented using colored circles, which fall down from the top of the screen. When selected, these songs show their title on their center, and they can be later used to capture similar ones (see Figure 3.22).

In the tabletop applications category, *Musictable* (Stavness et al., 2005) takes a visualization approach similar to the one chosen in Pampalk's *Islands of Music*, to create a two dimensional map that, when projected on a table, is used to make collaborative decisions to generate playlists. Another adaptation into the tabletop domain is the work

3 Exploring tabletops' distinctive affordances

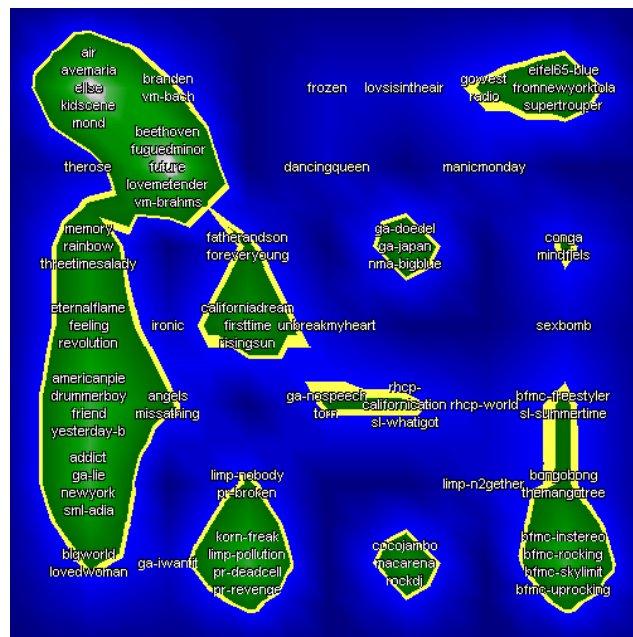


Figure 3.21: Islands of Music (Pampalk, 2003)

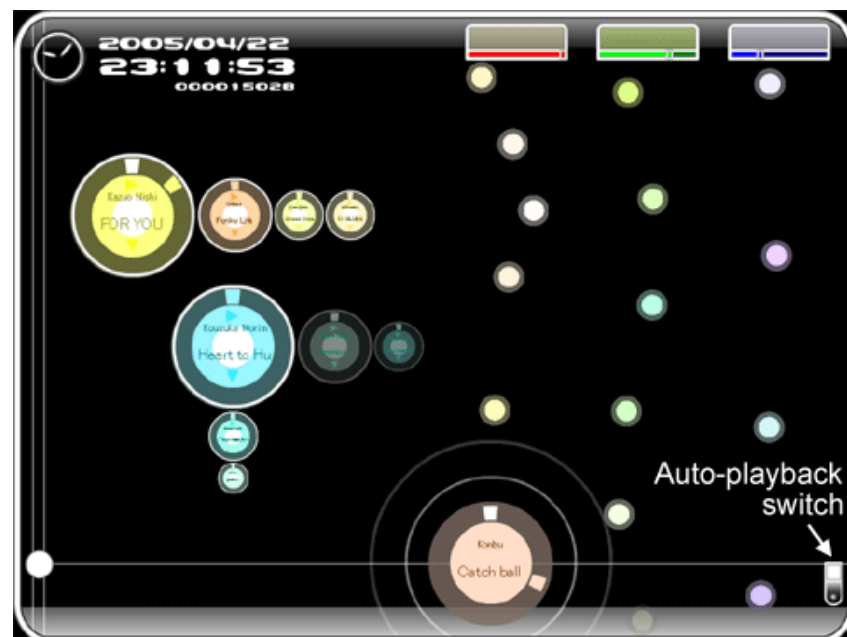


Figure 3.22: Musicream (Goto and Goto, 2005)

of Hitchner et al. (2007), which uses a SOM to build the map and also creates a low resolution mosaic that is shown to the user. The users can redistribute the songs on this mosaic, thus changing the whole distribution of SOM according to the user's desires.

3.6.2 Feature Extraction

SongExplorer uses a total of 6666 songs included in the Magnatune online database, weighting more than 26 GB. Being Creative Commons-licensed²⁰, this library is used in many research projects. These songs are processed by Essentia²¹, a music annotation library developed at the Music Technology Group (MTG)(Laurier et al., 2009), and the results are transformed to binary files that can be loaded by the system using the Boost²² C++ library.

3.6.3 Visualization

From the whole set of available annotated features generated by the annotation library, we are currently using a set of high-level properties and the BPM:

- Beats Per Minute (BPM)
- Happy probability
- Sad probability
- Party probability
- Acoustic probability
- Aggressive probability
- Relaxed probability

All these high level features are independent and, even the moods, which try to cover all the basic emotions, do not depend on each other (i.e. a song could be both sad and happy) (Laurier et al., 2009).

With this data, a multidimensional feature space (of 7 dimensions) is constructed, in which each song is a single data point with its position defined by these 7 features, all of them being normalized between 0 and 1. From this multidimensional data we construct a 2D space which preserves its topology, and we present it to the user, who will then be able to explore it.

²⁰<https://creativecommons.org/>

²¹<http://essentia.upf.edu/>

²²<http://www.boost.org>



Figure 3.23: Detail of the hexagonal structure of the grid.

Similarly to other visualization works, a SOM is used to distribute the data on the 2D space. Our implementation of the Kohonen network uses a circular, hexagonally-connected node grid, in order to fit the shape of the interactive surface. As opposed to the original implementation of SOM (Kohonen, 2001), a restriction is added to prevent more than one song falling into a single node, so that every representation in the 2D space should be visible and equally distant from its direct neighbors, as shown in Fig. 3.23.

The SOM defines a mapping from the multi-dimensional input data space onto a two-dimensional array of nodes. With every node, a parametric model vector, of the same dimensionality as the former space, is associated, so every node has coordinates both in the former space and the target space. All model vectors are initialized with low random

values.

It then starts an iterative process that gradually distributes the data points in the original space (analyzed songs) into the nodes in the target space (the position in the hexagonal grid). For every data point d_i in the original space, taken randomly, it finds the not assigned node N_j which $j = \operatorname{argmin}_k(\operatorname{dist}(N_k, d_i))$ where $\operatorname{dist}(a, b)$ is the distance function, in this case the euclidean distance function is used. When the node is found, it assigns the model vector the value of the data point. Because we want one data point only assigned to every node, we mark the node as already assigned until the next iteration.

Once this first step is done, we have all the data points assigned to nodes. It is time to filter the node array so the values of the model vectors are weighted with their neighbors, producing a “blurring” effect. This SOM uses an hexagonal topology (optimal for visualization) and thus in the filtering step calculates the influence of the other nodes using the hexagonal distance (see Figure 3.24).

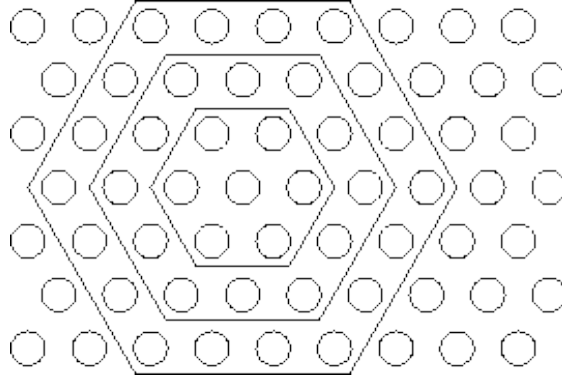


Figure 3.24: Hexagonal topological neighborhood. Note that nodes in the same delimited ares have the same hexagonal distance to the node in the center.

This “blurring” process smooths the model vector map, approaching the nodes values together. This allows to obtain a node array without high differences between neighbors, as the goal of SOM is putting similar things together.

This process is repeated until the desired order is achieved. The optimality of the ordering can be evaluated using the accumulated distance between all the neighbors in the hexagonal grid (the smoother the map, the better)

In the visualization plane, every song is represented by a colored circle, throbbing at the song’s BPM. Since there seems to be a strong agreement about the usefulness of artwork to recognize albums or songs (Leitich and Topf, 2007; Pabst and Walk, 2007), depending

3 Exploring tabletops' distinctive affordances

on the zoom factor, the actual artwork may be shown in the center of each song.

Additionally, colors are used to highlight the different properties of the songs. The coupling {feature \rightarrow color} was defined with an online survey where 25 people had to pair the high-level tags to colors. The color candidates were 7 distinctive colors with maximum saturation and lightness: red, blue, green, brown, cyan, yellow and magenta. An online questionnaire was created where subjects were only able to choose the best color representation for each tag. The results were: aggressive-red (with an agreement of 100%), relaxed-cyan (43.5%), acoustic-brown (52%), happy-yellow (39%), party-magenta (48%) and sad-blue (56.5%).

For every song, its corresponding property value is mapped into the saturation of the related color (0 meaning no saturation thus resulting on a grey color, 1 corresponding to full saturation), while the lightness is kept to the maximum and the hue is obviously linked to the emotional feature selected, as described in the previous color pairings (Fig. 3.25 shows the effect of different highlights on the songs). An option to see colors representing genres is also provided, although in that case the pairing between genres and colors is done randomly.

Multi-touch interaction

Basic finger interaction includes single and multiple finger gestures, and the use of one or two hands. The simplest gesture, selecting and tapping, is implemented by touching a virtual object shown on the table surface, with a single finger and for more than 0.4 seconds. In order to distinguish them from this selection action, other finger gestures involve the use of two simultaneous fingers for each hand. That way, using only one hand, users can translate the map and navigate through it, while the use of both hands allows rotating and zooming the map (see Fig. 3.26).

When the user puts two fingers close together a visual confirmation is shown over the map (see Figure 3.27). The navigation is designed in order to maintain the relationship between the physical position of the hands and the virtual points of the map. When using one hand, moving it will cause the map to move, in order to maintain the same position under the fingers. This gesture is extremely intuitive, since it is the same movement used to move a real map. Zooming and rotating the map is based on the same principle. The map scales and rotates himself to match the position of the fingers. It should be noted that most of these gestures have become de facto standards in multi-touch and tabletop interaction (Kim et al., 2007).

3.6 SongExplorer, a tabletop for song database exploration

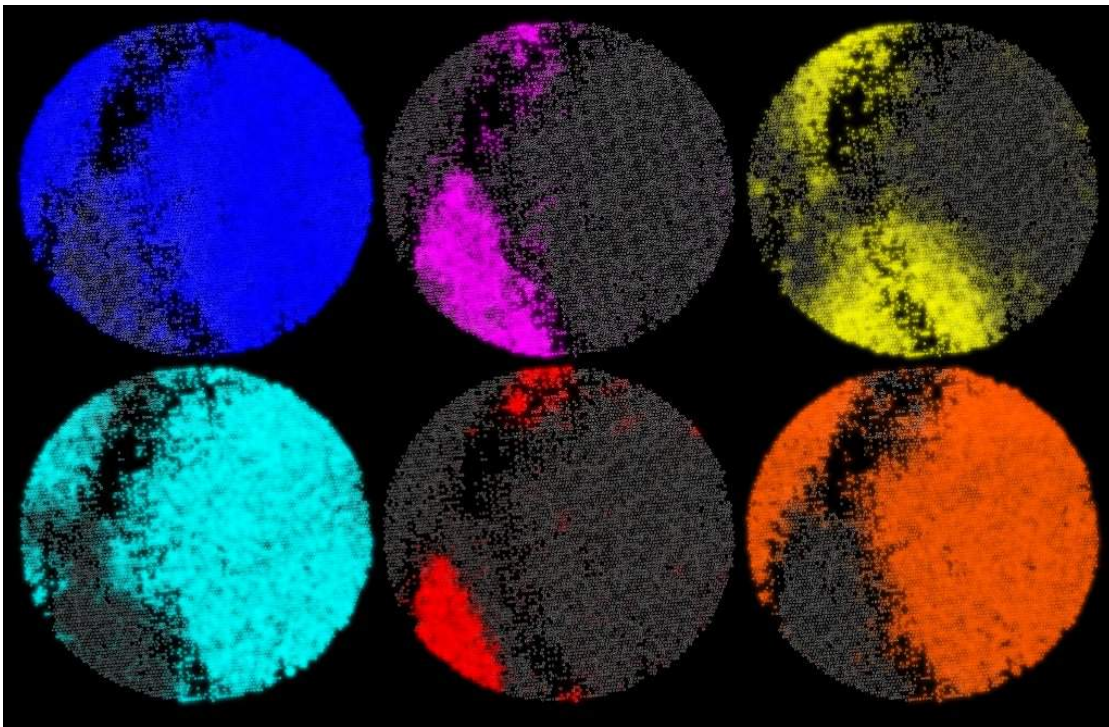


Figure 3.25: Colors highlighting high-level properties: sad, party, happy, relaxed, aggressive and acoustic.

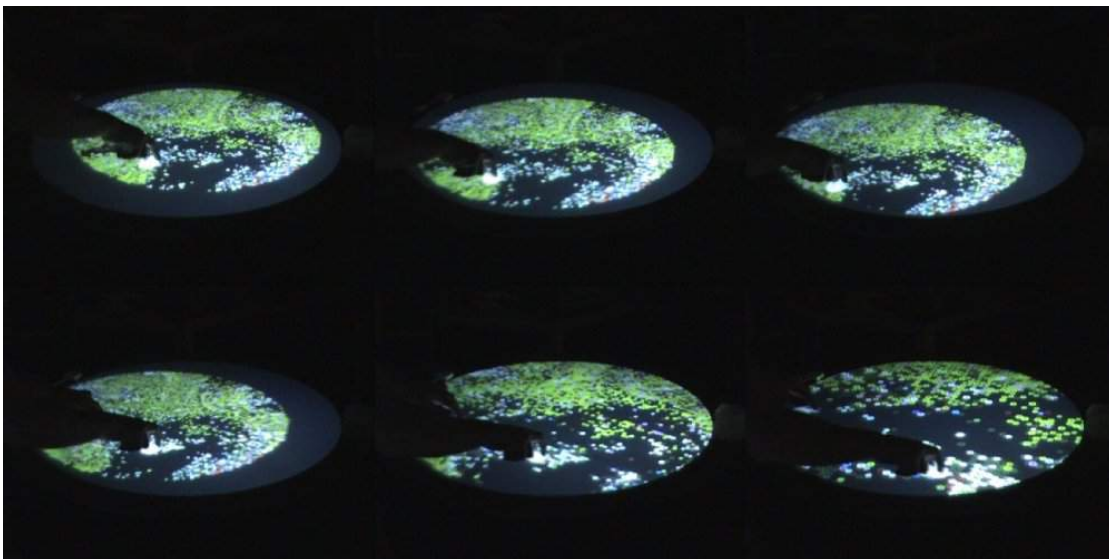


Figure 3.26: Virtual Map movement (up) and zooming (down)

3 Exploring tabletops' distinctive affordances





Symbol	Name	Description
	playlist navigator	Permits to run over the songs on the playlist
	color changer	Allows to highlight features of the songs
	magnifying glass	Shows information about songs
	navigation menu	Provides a way to return to known situations

Table 3.3: Tangibles used in SongExplorer

Tangible interaction with pucks

Additionally, SongExplorer tangibles also include 4 transparent Plexiglas objects, each one with a different shape and a different icon that suggests its functionality, as described in Table 3.3. These pucks, become active and illuminated when they get in contact with the interactive surface. As indicated below, some (like the color changer or the navigator) will apply to the whole map, while others (such as the magnifying glass) apply to the selected song.

- The **color changer** puck allows selecting and highlighting one of the different emotional properties of the whole song space. For example, changing the map to red allows us to see the whole map according to its aggressive property, with fully red dots or circles corresponding to the more aggressive songs, and grey dots to the least aggressive ones. Apart from helping to find songs based on a given property, the resulting visual cues also help to memorize and recognize the already explored zones of the map.
- When placed on top of a song, the **magnifying glass** puck allows seeing textual information on this particular song, such as the song title, the album, the author's name, as well as its artwork.
- The **navigation** puck displays a navigation menu, which allows the user to perform actions related to the movement and zooming of the map, such as returning to the initial view, or zooming on the currently playing song.
- The **playlist navigator** puck allows the creation and management of the playlist, as described below.

Managing playlist and playing back songs

SongExplorer has the ability of creating and managing song playlists. Playlist are graphically represented on the surface as a constellation, in which the stars (i.e. the corresponding songs it contains) are connected by lines establishing their playing order (see Fig. 3.28). Most stars have white stroke, except for the one that is currently playing (red), and the one the playlist navigator is focusing on (green).

Playlists allow several actions using both the fingers and the playlist navigator puck. When tapping on a song, this is automatically added to the playlist. Users can start playing a song by tapping on any star of the playlist. Similarly, crossing out a star removes the corresponding song from the list. A song will stop playing either when it reaches its end, when the song is deleted from the playlist or when another song is selected for playing, and a playlist will keep playing until its end, unless it is stopped with the playlist navigator puck. This object allows several additional actions to be taken on the playlist, such as navigating through its songs and showing information about the focused song in the same way the magnifying glass does.

3.6.4 Interface Evaluation

Some user tests were undertaken in order to evaluate the system, focusing on the interface design. The evaluation consisted in three areas: subjective experience, adequacy of the visualization and the organization, and interaction.

Experiment Procedure

To carry out the tests, an interactive tabletop with SongExplorer up and running was provided. The system was always on an initial state at the beginning. One subject at a time was doing the test. First of all, a little explanation about the purpose, visualization and interaction was given. Then the subject was asked to *find something interesting* in the music collection. No time limit was imposed, and the subject was observed throughout the process. At the end of the activity, the subject was told to fill a questionnaire, on which she had to rate, using a Likert scale of 11 levels (10: Totally agree, 0: Totally disagree), the several aspects of each area. They could also write suggestions at the end of the test.

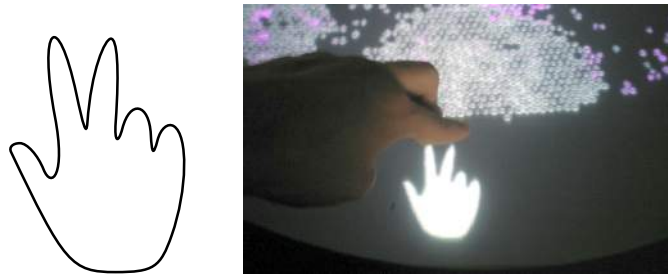


Figure 3.27: Visual feedback for 2-fingers gestures, original and live



Figure 3.28: Playlist and Playlist navigator

3.6 SongExplorer, a tabletop for song database exploration

	$\mu_{1/2}$	IQR
Enjoyed the experience	8	1
Discovered new music	8	1
Felt comfortable	8	1.5
Found it useful	9	0.5
Found colors correct	8	1.5
Found categories suitable	7	1
Found graphics adequate	9	1.5

Table 3.4: Evaluation Results. $\mu_{1/2}$: Median, IQR : Interquartile range.

Results

After doing the tests, the results were quite positive (see Table 3.4). Regarding the personal experience with SongExplorer, the subjects enjoyed the experience, discovered new and interesting music, felt comfortable, and found it useful to find interesting music. So the overall experience seemed to be good; we have to notice the low deviation, indicating that there was an agreement about these opinions.

Focusing on the visualization process, there was also a common opinion about the suitability of the colors used. This is not a surprise, as they were extracted from an online poll (details on subsection 3.6.3). According to the subjects, the categories (formerly the high-level properties from the annotation software) were suitable for the purpose of describing music. The graphics were also evaluated (meaning the adequacy of icons, the metaphor song-circle, the panels...) and also appreciated.

The level of understanding of every gesture and tangible of SongExplorer was tested, as well as their difficulty of use and usefulness. The only noticeable result was that there seemed to be an inverse correlation between previous experiences with tabletops and the perceived difficulty of finger gestures.

Finally there was a general demand for more music-player capabilities like pause or a progress bar for jumping to the middle of the song. The option of bookmarking and storing playlist was also desired.

3.6.5 Is map coherence important for music discovery?

While it may seem reasonable that clustering the presented songs together according to their similarity is an important factor contributing to the good navigation of the map, the extent to which this affects the user's experience and the result of the interaction needs to be quantitatively studied(Lechón, 2010). Are users finding new interesting

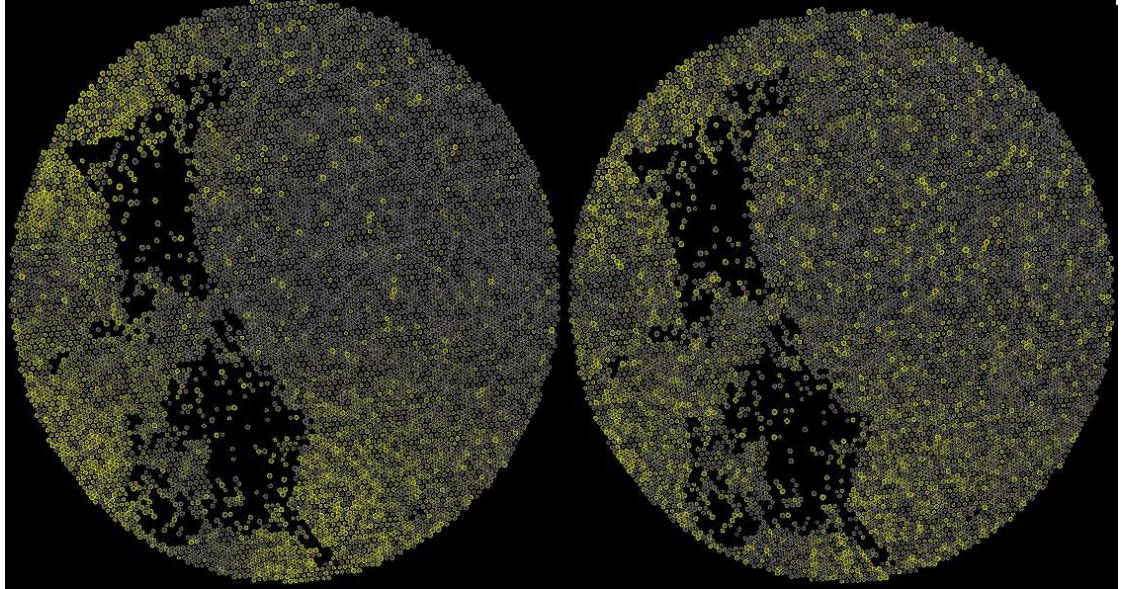


Figure 3.29: Comparison between two maps, one in the normal condition (left) and the other on the scrambled condition (right), highlighting the Happy probability. Notice the different dispersion.

songs because they are able to find similar songs close together and refine its search, or the simple fact of having random music presented to the users is enough?

Our hypothesis was that the spatial ordering is indeed important and we tested it by providing two experimental conditions. The first one is the original condition (the *normal* condition), the second one differs from the former in that the *goodness* of the generated map is substantially degraded through a process of random scrambling (the *scrambled* condition).

The process used to worsen the quality of the maps presented on the scrambled condition is by taking two random non-overlapping quarters of the songs and exchanging their coordinates on the map. By following this procedure, half of the songs are randomly displaced throughout the map, contaminating the coherent distribution of the musical pieces (see Figure 3.29). By comparing the users' subjective perception of the system and their relative performance carrying out a task in both conditions, we expect to accept or reject our hypothesis.

Experimental setup

Twenty participants (13 female, 7 male, all university students or junior university staff, mean age 26.9 ± 2.29) volunteered for this study. Half of them (8 female, 2 male, mean age 26.9 ± 2.02) were exposed to the normal condition while the other half (5 female, 5 male, mean age 26.9 ± 2.64) were exposed to the scrambled condition. After introducing the system to the participant, she was asked to complete a task consisting in navigating the map looking for songs she enjoyed and tagging them for 15 minutes. During the task, several events from the interaction were logged in order to further analyze them. The participant would fill a questionnaire afterwards.

Several measures can be used to answer the question of whether the SOM ordering plays a primary role in the interaction in SongExplorer:

- Question 4 from the questionnaire asks the subjects whether they agree with the statement "The ordering of the songs on the map was confusing".
- The number of songs the users liked during the interaction is a direct performance measure that can be normalized by dividing it by the total number of songs played.

Analyzing the recorded data we can expose other measures. Due to the fact that a correct ordering of the map keeps similar music pieces together, in the normal condition the subjects are expected to find music they like lying closer than users of the scrambled condition. This difference should lead to a sparser navigation of the map for the users of the scrambled condition. Two related measures can give us a sense of the sparsity of the navigation performed by a subject:

- Total traveled distance through the map, reconstructed by taking the length of the playlist including the originally removed songs.
- Mean distance between consecutive songs added to the playlist.

Results

We compared the means of the four measures across the two conditions using an unpaired two-tailed T-Test. Regarding the questionnaire, users of the normal condition rated the perceived ordering higher than those from the scrambled condition (5.1 ± 2.51 vs 3.1 ± 2.23 Likert scale, range 1 to 10); however, the difference among the two conditions was not significant ($p=0.33$) (see Figure 3.30).

Both the number of liked songs, and the proportion of liked songs over played songs, however, seem to yield unexpected results (number of liked songs: 5.7 ± 2.12 normal, 7.1

3 Exploring tabletops' distinctive affordances

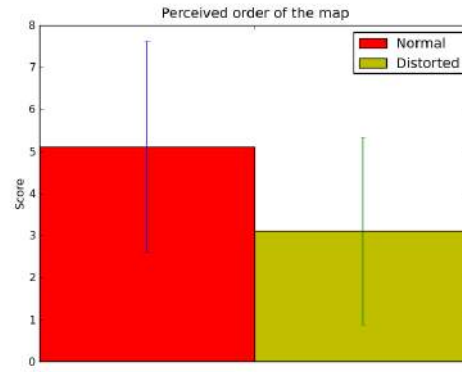


Figure 3.30: Answer to item four on the questionnaire: perceived order of the map.

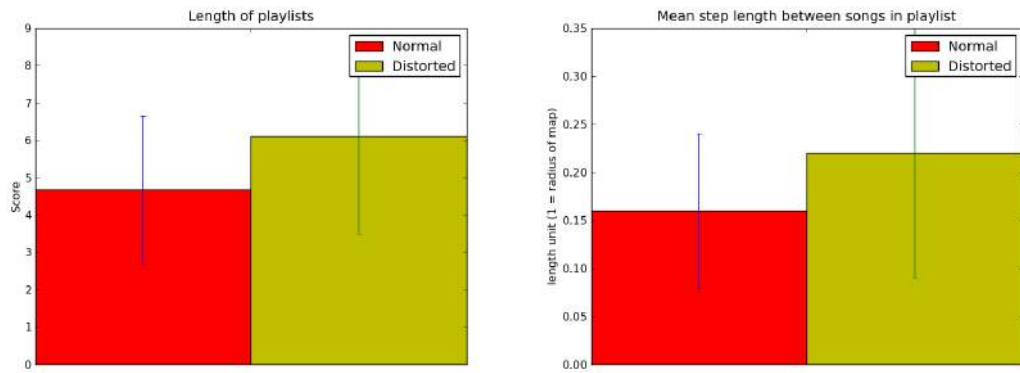


Figure 3.31: Total length of playlists (left) and mean step between songs in the playlist (right)

± 4.2 scrambled; proportion of liked songs: 0.29 ± 0.11 normal, 0.31 ± 0.15 scrambled). Again, there was no significant difference across conditions ($p = 0.47$).

The length of the playlists is found shorter in the normal condition (4.67 ± 1.97 vs 6.11 ± 2.62). Although without significant difference ($p=0.28$) (see Figure 3.31, left). The mean distance between consecutive songs added to the playlist was still shorter for the normal condition (0.16 ± 0.08 vs 0.22 ± 0.13), although again there is no significance ($p=0.41$) (see Figure 3.31, right).

Conclusions

Considering the results, we cannot confirm our hypothesis that the ordering of the song map affects the discovery of new interesting music.

There are several possible explanations for these results. First of all, maybe users prefer diversity when looking for new music, and thus ordering the songs is unnecessary. In this case, as stated previously, presenting the songs in the form of a map, and allowing users to quickly jump between different songs, is good enough for the users in order to find interesting music.

However, the benefits of an ordered song map may appear more in the long term, when the big song database starts being familiar to the users, who are then able to recall whether a particular area was interesting in the past and therefor can find new interesting music faster. Because of the nature of the experiments, this effect would pass undetected.

We cannot rule out the possibility of problems with the experimental setup (we used a bad ordering method, chose song features irrelevant to the user perception of the song collection, used a song collection unsuitable for the task or relied on a badly tagged song collection).

We are still convinced that plotting the songs on a coherent 2D map can help users navigate big song databases. In particular, the SongExplorer interface seems to be specially pleasing and approved by its users. Although there are some doubts on the ordering method regarding its influence of new music discovery, its influence on recall is still to be assessed, in future sessions of a same user.

We also think that this browsing method could also be applied to other kinds of data. Pictures, for instance, could be analyzed to extract features (such as color-based, face identification, etc) from them, in order to be sorted in a map, becoming a non-linear viewer for family photos.

3.7 Tabletop Applications TSI

Acknowledging the need of dissemination of tabletop applications, which rarely become public for various reasons (private projects, student work, work in nonacademic environments), we think it would be useful to briefly expose some of the projects done in the context of the author teachings in a course in tabletop application making. This course was taught for four consecutive years, and it was influenced by its approach into tabletop application making from a multi-user real-time perspective with an important implementation part.

The outcome of this course was a finished tabletop application, ready to be used on a real Reactable device. In this section we describe 11 works that were selected from a total of 35. They were selected for their final quality, their novel interaction techniques

3 Exploring tabletops' distinctive affordances

or their good concept. The complete details of the teaching, used materials and tools are explained in detail in Chapter 4.

By revealing these works, the tabletop and TUI community can learn from the common aspects and practices that arise in the tabletop application making process, which otherwise remain hidden.

3.7.1 Puckr

Original author Roger Pujol

Soundtrack authors Pablo Molina, Francisco Garcia and Frederic Font (2009)

Air hockey game, with generative music. Every player (up to 4) has to score the disk into the goal of other players. After receiving 4 goals a player loses and is eliminated from the game. After a period of time the level passes.

As the game progresses, new actions are available to players (as using fingers to create a temporary wall of ice, or the ability to use objects that power up the disc or create a trajectory-changing roulette).

The sound track of the game is generative, and depends on the current level and events of the game. The pucks used to “hit” the disc have a comfortable shape to apply the needed speed and force (see Figure 3.32).

3.7.2 TUIGoal

Authors Arturo Lascorz, Javier Salinas, Juan Luis Lopez and Juan Manuel Lentijo (2009)

This game is a replica of a traditional children game where bottle tops are used as players to fake a soccer game. The field is presented on the surface with a goal and three bottle tops per team. By dragging their finger on a bottle top, the players can define the direction and power of a kick that will project the bottle top to hopefully hit on the ball and make it to through the other team’s goal.

This adaptation improves the original game by removing turn-based game mechanics. Instead players can manipulate the bottle tops at any time. This also allows multiple players to join the game at the same time.

Objects are used to customize parameters of the field: scenario, bottle top’s weight, friction of the ground, etc.

3.7.3 Punk-o-Table

Authors Dani Garcia, Iván Hernández and Miguel Ferreiro (2009)

A musical sequencer centered on the *punk* musical genre. Three objects represent three instruments that can be manipulated: changing their volume by rotating them or defining its sequence of sound loops in different time scales by dragging a finger on a cell of the row presented around the object. Shaking the object erases all registered loops from the sequence.

Other objects are provided to change the playing speed or the number of the beat subdivisions, to save and load the whole song or instruments sequence, or to pause the whole application.

3.7.4 Smash Table

Author Alejandro Sánchez (2010)

A replica of a competitive card game where players win by getting rid of all their cards. The cards are managed by the system (sorting, distributing) as well as the games logic; leaving the four players with the active part.

Players will uncover a card from their personal pile, in order. Every time a card is uncovered, the rules of the game may require *battle* between two or many players, depending on the combination of the four visible cards:

- When two or more cards have the same figure on them, a *battle* is called between their owners.
- If a *color star card* is present on the table, the above rule applies but instead of comparing their figures, their color is compared.
- If a *four-way battle card* is on the table, all the players engage into the *battle*.

A battle consist of attempting to place their token at the center of the table first. The last player to place his token loses, and collects all the played cards into his pile. If a player places her token mistakenly (when there is no call for *battle*), she also loses, with the same consequences.

It is interesting that because of the *violent* nature of this game, using rigid tokens could be dangerous. The game creator came out with an idea that was proved so effective, that it has been copied in many later projects: using soft objects easy to handle. In his case he realized that dish sponges had a convenient handle and their size allowed

to place a fiducial marker underneath (see Figure 3.34, right). This solution perfectly allows to safely play the game.

3.7.5 80-table

Authors David Peñuela, Marc Ramírez, Miquel Cornudella and Piero Sacco (2010)

A tribute to PacMan. Every player (up to four) has a character that moves freely in a circular endless space filled with dots. *Eating* those dots (by passing through them) gives the player points. The player that has more points after a time period wins.

The characters are commanded with rotatory tangibles that control the direction of their movement, while the speed cannot be controlled. When two characters collide, they lose part of their points in a form of dots around the collision place.

Around the driving tangible, points are displayed, and also attacks and bonuses are offered. To use them, players hit a character (its own if it is a bonus, another players' if it is otherwise) with a fly swatter (see Figure 3.35, right). This tool is extremely useful for fast movements with precision, and has been replicated in later projects.

3.7.6 Tower Defense

Authors Roger Tarrago and Sergi Juanola (2011)

A classic tower defense game. Two players take different roles in the game. While the first one creates caves, where attackers come out from, using a hammer, the second one places defensive towers around the center (see Figure 3.36, center and left).

Attackers are automatically created periodically in the caves, while the towers automatically shoot them if they pass through the targeting space. Both the type of attackers and the weapon of the towers is chosen from a surrounding menu. The available set of attackers depends on the level (that changes with time), while the available set of weapons depends on the amount of interconnected towers (the amount of available towers increases with the level).

It is interesting how the authors used the appearance of the object to convey meaning, instead of just using generic shapes, which is the most common strategy in past projects.

3.7.7 Pulse Cubes

Authors Antonio Ruiz, Iván Mahindo and Adrià Ruiz (2011)

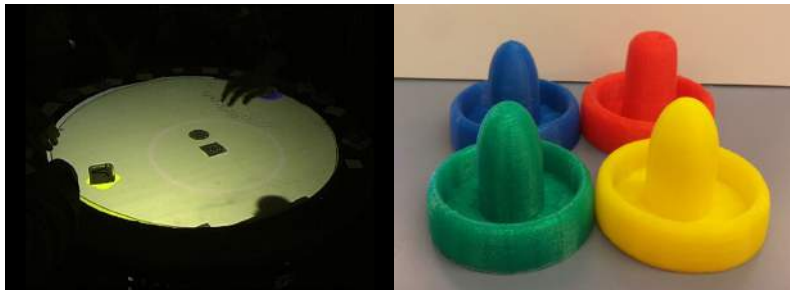


Figure 3.32: Puckr

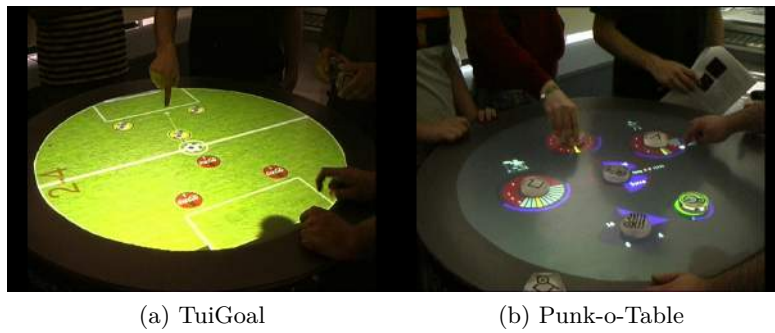


Figure 3.33



Figure 3.34: Smash Table



Figure 3.35: 80-table

3 Exploring tabletops' distinctive affordances

Pulse Cubes creates rhythms from networks of interconnecting objects. Those objects are cubes where each side represents a different sound sample. When placed on the table, each cube can be aimed to another one to establish a connection. Those connections transport pulses that travel at a fixed speed from one object to another, meaning that between more distant objects the pulse will take more time.

When users touch the circle created around the object or when it receives a pulse from a connection, its sample is played and immediately a pulse is sent through all its connections.

Connections can be configured to be one way or two way. The network created by the objects and connections will create a space of rhythm possibilities, which can be self-sustaining, extinguish themselves, or explode exponentially.

The network can be changed in real time provoking the resulting rhythms to change dynamically.

3.7.8 Rubik Musical

Authors Andres Bucci and Álvaro Sarasúa (2011)

A circular step sequencer. Instead of using a horizontal score, the authors decided to present time traveling from the center of the table to the outer edge, thus putting the sequence of notes in radial position. Three different instruments, each with a distinct color, can be controlled with five notes, each in a pentatonic scale. By placing the finger onto the circle of a particular note and time position, users switch the activation of such note.

The evocation of the Rubik Cube relates to two functions that can be controlled with objects. One moves the score forward and backward in time, either for one instrument or for the whole sequencer. While the sound effect is delaying or advancing the sequence, the visual effect is moving the notes away or closer to the center.

The other one also moves notes, but instead of moving them to or from the center, it moves them around the center. With a clear visual effect, the sound effect is interestingly unpredictable: while for most notes of every instrument there will simply be a pitch shift, some notes will overflow from one instrument to another. Taking into account that one of the instruments is *percussion*, the results are quite interesting.

Another object is provided in the form of a sponge (see Figure 3.38, right). Instead of for safety reasons, the decision here is purely aesthetic and metaphorical, as it is used to clean the score.



Figure 3.36: Tower Defense

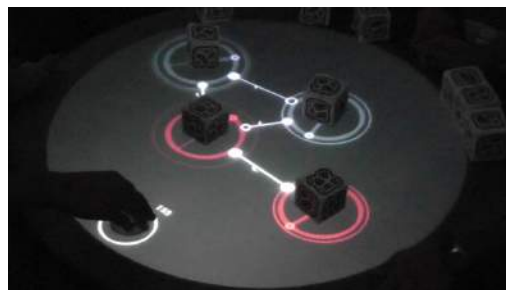


Figure 3.37: Pulse Cubes

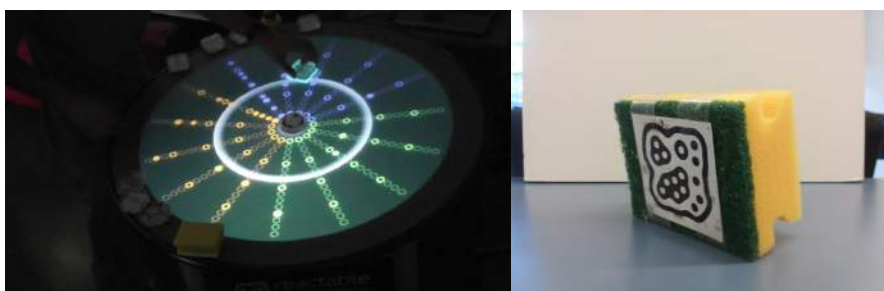


Figure 3.38: Rubik Musical

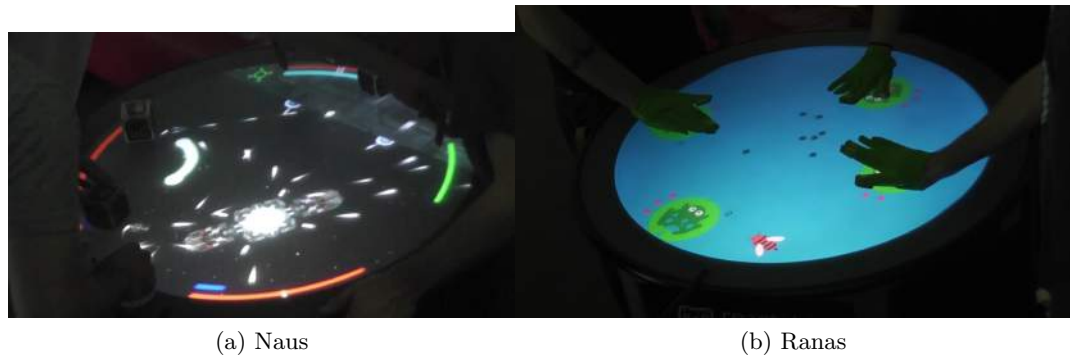


Figure 3.39



Figure 3.40: Scrabble

3.7.9 Naus

Authors Juan Cava, Oscar Cosialls and Eduard Soler (2012)

A three player spaceship battle game. One of the players controls a big ship with two cannons at one side of the table. Using an object she can control the direction of the cannons. Also, by doing gestures with the fingers, the player can try to create new smaller secondary ships.

The other two players control two tiny ships with an object as a handle, so they can move and aim them easily. When a new ship is summoned by the other player, they have a small period of time to cancel such creation, by performing a complex gesture in place. The gesture is described by a series of numbered dots that the players have to connect with a single finger stroke. Other power ups are available by the two players to gain special temporary powers.

The decision of using the gesture complexity as a challenge in the game is interesting, usually gestures are selected by their ease of use and not the other way around.

3.7.10 Scrabble

Authors Joan Anselm Cardona, Adrián Ponz and Xavier Zanatta (2012)

A word game. Up to four players situated around the table try to form words by picking letters from a common pool. To pick a letter, players take their picking object and hit the letter, this is then added to their forming word and removed from the pool. After a limited time, players that formed a correct word are awarded with points computed in the way of the original Scrabble²³ game.

As the process of picking letters is competitive and can result in violent hits, the team learned from previous projects and used soft objects (see Figure 3.40, right).

3.7.11 Ranas

Authors Fidel Montesino and Iris Serrano (2012)

A tribute to *Hungry Hungry Hippos*. With its same mechanics four players control four frogs to eat flies in the center of the table. Each frog can also shoot a missile to kill other frogs. Wasps also appear and users must avoid eating them.

What is interesting in this game is the way players control the frogs. Wands were created with fiducial markers on the palm, the thumb and the index and middle fingers together. Users placed the pal on the surface to move the frog, touched with the index and middle fingers to unroll the tongue to chase flies, and touched with the thumb to shoot a missile (see Figure 3.39b).

3.7.12 TSI applications' statistics

Taking into account all the projects created in the TSI course, we can extract and comment some interesting statistics regarding various aspects of the interaction (a complete List of the projects and their characteristics can be found in Appendix C). For instance, we saw a predominant preference for creating games, specifically 71.43% of the applications. This preference can be explained by the real-time multi-user nature of this kind of applications. Accordingly, we would expect a high percentage of music related applications, because of their similar real-time requirements, but that sector constituted only 20.00% of the total. This can probably be explained by the low level of music training of the students, as opposed to video-game related experience.

²³<http://www.scrabble.com/>

3 Exploring tabletops' distinctive affordances

Applications designed to solve specific problems or tasks accounted for 14.29% of the projects, while 5.71% were concept demonstrations regarding specific interaction techniques (percentages of application types overlap because of entries entering into multiple categories).

Object-based interaction (94.29%) was preferred over finger-based (71.43%). Several factors can explain this preference; firstly, we insisted on the students using objects where possible and coherent. Secondly, in most of the cases, finger tracking is less reliable than object tracking in the Reactable. A third possibility is that, because of interaction design requirements, objects could be more suitable than fingers. An example of this is the use of objects to identify the user interacting with the surface, 39.39% of the applications using objects actually used them to identify the user, accounting for 52,00% of the games (it seems that identifying players in games was an important issue).

We have also seen how soft objects were first introduced in Smash Table and later copied in other projects. This is probably the most evident example, but other object strategies have also been influencing later projects, such as the use of fly swatters.

3.8 Conclusions

We reviewed the distinctive affordances of tabletops, which come from the capability of interacting with objects, from the shape of the interface (big, table-shaped) or from the possibility of having multiple interaction points. We showed that, despite those capabilities, there is still a limited number of successful commercial tabletop applications.

Two examples of tabletop applications using its capacities were presented, describing their purpose, their functioning and their evaluation. Those can serve as a guide of how to take advantage from the tabletop capacities to design a coherent interaction.

Addressing the apparent lack of interest on tabletop application making, we also tried to promote its practice by teaching a dedicated course to undergraduate students. We gave a theoretical background, as well as practical tools, best practices and examples helping them to achieve the best results. The results were presented to contribute to the knowledge of the current practice and general trends in tabletop building communities, a topic that is rarely documented.

While supporting those projects (and because of our own experience in the field) we assessed the added difficulty of programming such systems, which we addressed by creating specific frameworks to support their typical challenges. This effort is described in next chapter, *Empowering application developers* along with the consequences on the

resulting projects for each introduced feature and enhancement.

3 Exploring tabletops' distinctive affordances

4 Empowering application developers

If I have seen further it is by standing on the shoulders of giants.

- Isaac Newton

Documenting the distinctive affordances and features of tabletops, as well as providing relevant examples of good interaction design for this type of devices, may not be sufficient to successfully support the creation of applications for them. Frameworks that help developers to create applications that take tabletops' advantages into account and encourage the use of related good practices is equally necessary.

In particular, frameworks allow programmers to focus on the details of the application and not on the common set of repetitive tasks needed to build a tabletop application. General solutions to common problems are supplied so that programmers do not have the temptation to implement them themselves again, which would not only be time consuming but also introduce potential associated problems such as new bugs.

Simultaneously, by defining the programming idiom and common strategies in the framework, developers are forced to code in a specific fashion that avoids common bad designs or pitfalls to appear.

In this section we describe two frameworks that try to address the aforementioned challenges, *ofxTableGestures* and MTCF, both created and evolved in the context of teaching how to create tabletop applications for the Reactable hardware (see Section 3.4).

The two frameworks have two different purposes. While *ofxTableGestures* is intended to be used by programmers and students of computer science, MTCF is designed for sound and music computing students and artists. Despite of this, we, their authors, also use them a great deal.

As opposed to creating applications, where users have to be taken into account, a framework developer has to think about future developers as well. When developing a framework for internal use, many practices, idioms and programming conventions are introduced without noticing. Features that may seem important, because of past experiences, are added while others are left out because they were considered too trivial. Contact

with other developers and practitioners is essential to try not to be affected by such deviation.

In this sense, it is important to consider that those frameworks were developed for students assisting courses where they were encouraged to use them. Having a set of new developers using the frameworks, presented itself as a great opportunity to test and improve them, according to the results and problems found by the students.

So, although we used the frameworks internally most of the time, new fresh students' troubles served as a driving force for their improvement, trying to address the type of challenges that a new developer would face when creating such type of application.

4.1 ofxTableGestures: a framework for programmers

The reasons that led us to create a framework for programing tangible tabletop applications are two. First, through the experience of implementing various of such applications, we started to notice that some parts were just being copied throughout different projects, and we considered that we should unify these pieces of code into a single package. Second, for the upcoming teaching duty on *Taller de Sistemas Interactivos* (TSI, interactive systems workshop), we needed to provide a base to the students where they could develop their projects.

The course started, in fact, almost a decade ago, initially conceived as an “advanced interface design course”, and progressively evolved into a more specific “tabletop design course” the last years. It was given in one trimester (10 weeks), with four teaching hours every week, which are divided into theoretical and practical classes (2 h/week for each). The about 30 students that integrated the course generally were skilled at programming (e.g. they had good knowledge of C++), and had studied a more traditional course on HCI the previous year, but had no prior knowledge of tangible or tabletop interaction, and had never studied any design-oriented discipline.

The course objectives were the following: (a) to deduce and learn from observation and analysis what may constitute some valuable design criteria for tabletop interaction, understanding the main benefits and drawbacks of these technologies, in order (b) to conceive, design, implement and informally test a real tabletop application. Given the short time available and the ambitious scope of the course, the traditional iterative process of designing, implementing, evaluating, redesigning, re-implementing, etc., could not be applied. To solve this limitation, much emphasis was given in the initial design process, which was attained by studying, analyzing and criticizing many (both successful

and unsuccessful) examples, fostering the discussion and the active participation among all the participants.

The practical part was done in groups of three. This aimed at teaching all the implementation steps to create an application that could eventually be used on the real device, covering the whole tangible tabletop application implementation process. In previous years it even combined hardware construction, with students being asked to build their own simple tabletops using a cardboard box, a source of visible light and a cheap webcam. Although instructive, the outcomes of these works were hardly ever used (mainly because of the lack of a space to store them in), consuming also too much of the students' time and energy. Lately, we therefore decided to concentrate on the software part, given that the students' final results will be finally tested on a real and fully working tabletop interface anyway.

The framework resulting from this support evolved over time, deciding to enhance and correct bugs every year in function of the feedback and application results of the students. We present 4 versions of the framework corresponding to the courses for the years 2009 through 2012. In each iteration we will describe the implementation and changes with respect to the previous version and the observed consequences on the resulting projects.

4.1.1 The early stages: providing libraries (2009)

Creating a new tabletop application from scratch, can be a cumbersome activity involving many complex tasks. Several well-known open-source solutions do exist addressing the highly specialized input component, both for the tracking of multi-touch fingers, such as the NUIGroup's Community Core Vision¹, or for the combined tracking of fingers and objects tagged with fiducial markers, such as reacTIVision (Bencina et al., 2005). These software tools greatly simplify the programming of the input component, essential for this type of interfaces, but this is only one part of the problem. The visual feedback or the graphical user interfaces, which often also include problems specific to tabletop computing, such as aligning the projector output with the camera input or correcting the distortion that results from the use of mirrors, still have to be manually programmed.

In earlier years, only one piece of software technology was provided to the students: reacTIVision. The choice of programming language, libraries, etc to use for implementing the application was left to the students, as long as it accepted the TUIO protocol.

This choice often resulted into applications that were not usable in the Reactable hardware. A typical problem was the graphical distortion; because of the position of the

¹<http://ccv.nuigroup.com/>

projector, the mirror and the surface, the image is projected with a considerable keystone deformation. As many technologies used by the students were limited to 2D output (such as Java 2D API²), they were unable to apply a correcting distortion to the resulting image. The result often was that, after a hard work of several weeks, their applications worked correctly only with the simulation software provided with reacTIVision, but when used on the Reactable, the images did not meet the objects and fingers on the surface. Needless to say, this was not very encouraging for the students.

Another undesirable consequence of this freedom of choice for the students was the compatibility problems between different operating systems and building platforms. Because of that, very few of the projects from this period have survived the passage of time, as the necessary versions of frameworks, libraries and environments often get increasingly difficult to obtain and install with time.

To avoid these problems, we decided to provide a framework that would be used by the students to create their applications, which complied:

- Multi-platform: students have to be able to work at home with the platform of their choice, so we needed a framework that could at least run on Windows, Mac OSX and Linux. This would contribute in solving the perishability problem.
- Input independent: making it TUIO compatible, any TUIO-based tracking program could be used, ensuring the independence from a particular library or program.
- Graphics distortion correction and input alignment: The Reactable (and many other tabletops) projects through a mirror in order to maximize the projection distance. This inevitably causes some graphic distortion and complicates the perfect alignment of the input with the graphical output. The new framework should transparently solve these programming problems, or at least facilitate this solution.
- Without an existing set of widgets or gestures: we feared that if the framework provided simple ways to create WIMP-like elements, the students would use them to save time; even if that was not in accordance with the goals of the tabletop interface, mainly supporting multi-user activity. We wanted to encourage the students to implement the gestures and visual components from scratch, so they could think about its appropriate use.
- Simulation software: since we did not have enough interactive tables for all the students, we needed a simulator solution that would allow the students to easily develop and test their applications, even at home.

²<http://docs.oracle.com/javase/tutorial/2d/overview/>

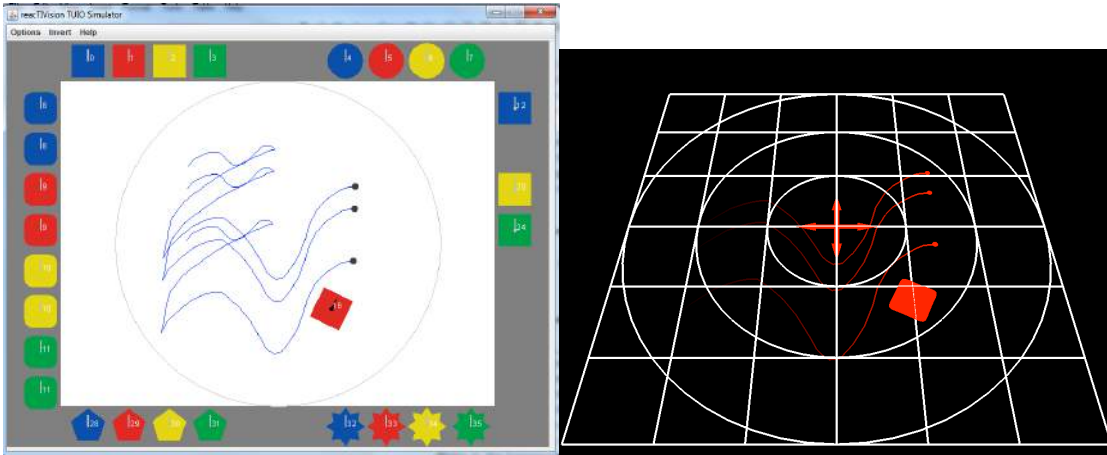


Figure 4.1: TUIO Simulator and graphical distortion.

Some of these requirements prevented us to use existing solutions from that time. So we decided to put together a C++ framework with the minimal needed functionality: Integrating a TUIO client so students would receive object and finger events directly in the application code and using OpenGL libraires to draw graphics, with the code to distort the image and calibrate this distortion already implemented and transparent to the students (see Figure 4.1). Those choices were multi-platform and we provided means for compiling the framework in all platforms.

The simulator software solution was provided by using TUIO Simulator, a program from the reactIVision suite that allows to simulate finger and object interaction on the table with the mouse and sending the TUIO messages directly to the developed application (see Figure 4.1).

The results at the end of the course were very positive. Almost all of the groups could correctly use their application in the tabletop hardware, thanks to the graphical distortion already implemented in the framework. Only one group had problems with the calibration and it was because they did not use the framework at all, but wanted to use Java.

We detected some difficulties and caveats the students had in the process of creating the applications. Those should be corrected and used to evolve the framework.

One of the problems students faced was the lack of provided libraries for common tasks such as image and texture loading, sound output, access to time and date, etc. Much time was used to solve these technical difficulties that did not specifically form part of the main task of the course: designing and making a tabletop application.

Students also encountered problems when implementing gesture recognizers for their applications. As the framework did not enforce a distinction of gesture and program code, the two would typically get mixed (i.e. in the same class) and evolved into an unstable situation, where implementation of complex recognizers was difficult, and allowing multi-user input very hard to achieve. A clear example was happening in Punk-o-Table (see Section 3.7.3); the students implemented a *shake* gesture recognizer for a single object, integrated with the main program and loop, and debugging the recognizer was encountered to be very difficult because of the integration with other code, eventually realizing that in order to allow this gesture to be performed with various objects simultaneously, they would need to rewrite everything.

A third problem that added difficulty to the students was using the TUIO Simulator to test their applications. Because the simulator is a separated program, input events such as finger and object input happened in a different window than the visual output. Touching a particular element on the program from the simulator was not easy. Some workarounds were used to attempt to unify those input and output elements: as some Linux window managers allowed to present a particular window as translucent, the simulator could be placed above the application in order to simulate this integration, but this situation was not optimal.

Those problems would be addressed in the next version of the framework.

4.1.2 Creating Gestures is the key (2010)

To solve the issues identified during the first year we transformed the framework in several ways. First of all, we adapted the framework to work with openFrameworks³ (OF), a group of multi-platform libraries written in C++ designed to assist the creative process, which integrates libraries for easily loading textures, videos, sounds and music (Noble, 2009). OF also brings advanced OpenGL methods for drawing figures and an abstraction layer between the drawing and the input loops.

The renewed framework⁴ (now called ofxTuioGestures and ofxTableDistortion⁵) is an add-on to OF, and is now divided in two parts: TUIO input and graphics output. This division forces the separation of gesture detection from the application's logic, solving the problem of mixing the two, present in the previous version.

The TUIO input part processes the messages that arrive to the framework from any

³<http://openframeworks.cc/>

⁴<https://github.com/chaosct/ofxTableGestures/tree/legacy>

⁵It is common practice to prepend names of OF add-ons with *ofx*.

```

class testApp : public CanDirectObjects < CanDirectFingers
    <tuioApp <TableApp> > >
{
public:
    void Setup();
    void Update();
    void Draw();
    void WindowResized(int w, int h);
    //CanDirectFingers
    void newCursor(int32 id, DirectFinger *);
    void removeCursor(int32 id);
    //CanDirectObjects
    void newObject(int32 s_id, int32 f_id, DirectObject *)
        ;
    void removeObject(int32 s_id, int32 f_id);
};

```

Listing 4.1: Example main app receiving DirectObjects and DirectFingers events.

application (e.g reacTIVision or any other applications that can send TUIO messages). Once these messages are processed, this component detects and generates high-level gestural events to be interpreted by the main application loop. It also allows programmers to implement new complex gestures as *plug-ins*, self-contained pieces of code that recognize the gesture independently from the application logic.

The specification of the type of gesture events that the main program will be receiving is done by creating a series of template-based mixin⁶ classes associated to the different gestures, for instance to receive the events from the very basic gesture recognizer *Input-GestureBasicFingers* (that provides finger-related events) the main class should inherit from the mixin *CanBasicFingers*. As you can see in the example in Listing 4.1 this technique required a great amount of C++ template jargon.

In addition, gesture recognizers are run in a different thread, as we were thinking that computing-expensive recognizers could be implemented. As depicted in Figure 4.2, gesture events are embedded into instances that are transferred from the gesture recognition thread to the main application thread through a queue.

This modularity introduced the possibility of using *gesture composition* to define gestures. That is, defining a gesture in terms of another simpler one. A double tap gesture, for instance, can be defined as a sequence of tap gesture events, instead of using raw touch events. As shown in Listing 4.2, composition involved manually processing the events generated by the gesture recognizer used as the basis of the new gesture. Also,

⁶<https://en.wikipedia.org/wiki/Mixin>

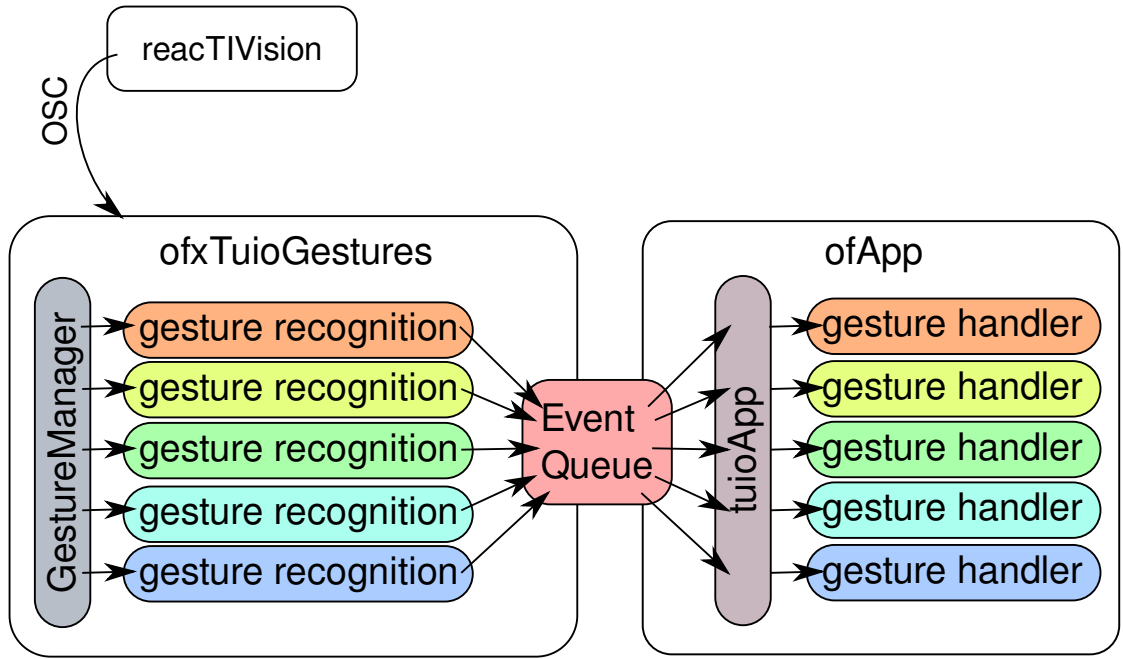


Figure 4.2: Diagram of ofxTuioGestures event flow. Notice a strong separation between gesture recognizers and the program, using different threads.

when creating a new gesture recognizer, the corresponding mixin should be created. Listing 4.3 shows the related mixin from the previous gesture composition example.

This new version also includes an embedded simulator. The students no longer needed to run TUIO Simulator to test the interaction when in absence of a real tabletop device. Instead, as the simulator is now included in the same application, they would test object and finger input directly over the generated graphical output (see Figure 4.3).

When the simulator is enabled, a right panel with a subset of figures is shown on one side of the screen. These figures are labeled with the identifier that will be reported by the TUIO messages to the system and have various configurable shapes. *ofxTableGestures* includes six different figure shapes (circle, square, star, rounded square, pentagon and dodecahedron), which are defined in a configuration file that includes the figure shape, the figure identifier and the figure color.

The effort put in separating gesture code from the application logic yielded good results. Complex gestures were more present in the projects developed with this framework version. Even one project, *ProjectWalk*, (by Ciro Montenegro and Ricardo Valverde), implemented the *walk* gesture, which consists of using the index and middle fingers to imitate the walking of a person (see Figure 4.4). The complex task of correctly

4.1 ofxTableGestures: a framework for programmers

```

class InputGestureDummyTab : public CanBasicFingers <
    tuioApp <InputGesture> >
{
    std::map<int32, tabcursor *> tstamps;
    InputGestureBasicFingers * basicfingers;
public:
    InputGestureDummyTab()
    {
        basicfingers = Singleton< InputGestureBasicFingers
            >::get();
    }
    virtual void ReceiveCall(const char * addr, osc::
        ReceivedMessageArgumentStream & argList)
    {
        for (std::list<TEvent *>::iterator it =
            basicfingers->events.begin() ; it !=
            basicfingers->events.end() ; ++it)
        {
            processTevent(*it);
        }
    }
    //From CanBasicFingers
    void addTuioCursor(int32 id, float xpos, float ypos,
        float xspeed, float yspeed, float maccel)
    {
        tstamps[id] = new tabcursor(xpos, ypos,
            ofGetElapsedTimef());
    }
    void updateTuioCursor(int32 id, float xpos, float ypos,
        float xspeed, float yspeed, float maccel)
    {
        tstamps[id]->update(xpos, ypos);
    }
    void removeTuioCursor(int32 id)
    {
        if (tstamps[id]->istab())
        {
            TeventDummyTabTabed * evt = new
                TeventDummyTabTabed();
            evt->x = tstamps[id]->x;
            evt->y = tstamps[id]->y;
            events.push_back(evt);
        }
    }
};

```

Listing 4.2: Example of gesture using composition. In this case the (naive) tap gesture is implemented by receiving basic finger events from *InputGestureBasicFingers* gesture recognizer.

4 Empowering application developers

```
template <class Base>
class CanDummyTab : public Base
{
public:
    //Interface redefined by ofApp
    virtual void tab(float x, float y) {}
    //processing events callbacks
    TEventHandler(TeventDummyTabTabled)
    {
        TeventDummyTabTabled * e = static_cast<
            TeventDummyTabTabled *>(evt);
        tab(e->x,e->y);
    }
    //registering
    CanDummyTab()
    {
        TRegistraCallback(CanDummyTab, TeventDummyTabTabled)
        ;
        registerMeToInputGestureManager(Singleton<
            InputGestureDummyTab>::get());
    }
};
```

Listing 4.3: The mixin used to receive *InputGestureDummyTab* events form Listing 4.2.

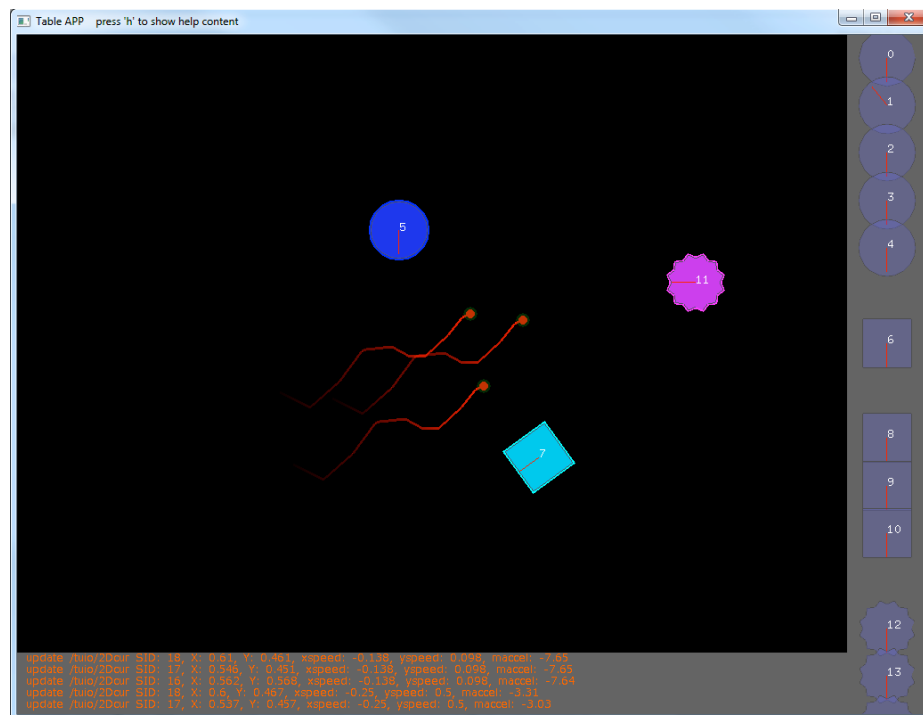


Figure 4.3: Embedded simulator

implementing this gesture required, however, a great amount of effort, which prevented the team from completing the rest of the planned application.

Although this new version proved to provide a way to program complex gestures more easily, we identified a common problem many students fell into: the difficulty of programming multi-user gestures. Gestures from a single user perspective are quite straightforward to program, as no other type of input can interfere with the gesture. If the sequence of events that defines a double tap is receiving a tap and then receiving another tap near the first within a time window, we do not have to worry about a tap happening in another place of the surface by another user. By contrast, if expecting multi-user input, such as in our case in tabletop applications, the recognizer has to be prepared to handle multiple simultaneous gestures as well as undesired input data. See Listing 4.4 for an example.

Another problem that we identified was the difficulty of implementing gestures that involved virtual elements on the surface. As gesture code was totally separated from the application logic, it was very difficult to know if a gesture was entirely happening within the area of a virtual object, as the information from those objects would be separated from the gesture code and thus not available for the gesture recognizer. Although for very quick and local gestures such as a tap or a double tap the problem is limited, for gestures that take a non negligible amount of time or space it is difficult to know a posteriori if the gesture developed inside the target element.

4.1.3 Multi-user gestures (2011)

Several changes were introduced to the framework to address the observed problems. As a large amount of code was introduced and the code structure substantially changed, there was a last change of name; the framework would definitely be called *ofxTableGestures*⁷.

One of the efforts made in this version was the simplification of the template-based programming interface. Many tedious event definitions are now automatized in order to minimize the amount of C++ template jargon code to be written that was prone to illegible bugs. As an example see how a declaration of an event type in the previous version (Listings 4.5) is simplified in this new version (Listing 4.6).

Another big addition to the framework was the implementation of areas that virtual elements can use to filter the gestures made on the surface, and that gesture recognizers are aware of. When input events are received by the framework, they are matched to

⁷https://github.com/chaosct/ofxTableGestures/tree/0.X_series

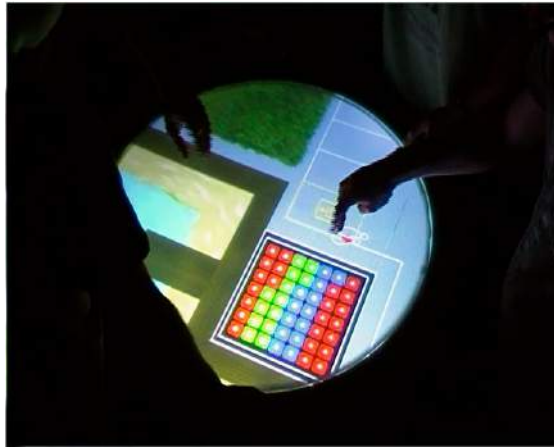


Figure 4.4: ProjectWalk

```
class TeventBasicFingersMoveFinger : public TTEvent <
    TeventBasicFingersMoveFinger >
{
    public:
        int32 s_id;
        float xpos, ypos, xspeed, yspeed, maccel;
};

template <class Base>
class CanBasicFingers : public Base
{
    //...
    TEventHandler(TeventBasicFingersMoveFinger)
    {
        TeventBasicFingersMoveFinger * e = static_cast<
            TeventBasicFingersMoveFinger *>(evt);
        updateTuioCursor(e->s_id, e->xpos, e->ypos, e->
            xspeed, e->yspeed, e->maccel);
    }
    //...
};
```

Listing 4.5: Declaring an event in ofxTuioGestures.

```
SimpleDeclareEvent(CanBasicFingers, updateTuioCursor, int32,
    float, float, float, float, float);
```

Listing 4.6: Declaring an event in *ofxTableGestures* (first version). see the difference with Listing 4.5.

```

class InputGestureSingleUserTap : public CanBasicFingers <
    tuioApp <InputGesture> >
{
    int32 _id;
    float _xpos, _ypos;
    float _tstamp;
    InputGestureBasicFingers * basicfingers;
public:
    InputGestureSingleUserTap()
    {
        basicfingers = Singleton< InputGestureBasicFingers
            >::get();
    }
    virtual void ReceiveCall(const char * addr, osc::
        ReceivedMessageArgumentStream & argList)
    {
        for (std::list<TEvent *>::iterator it =
            basicfingers->events.begin() ; it !=
            basicfingers->events.end() ; ++it)
        {
            processTevent(*it);
        }
    }
    //From CanBasicFingers
    void addTuioCursor(int32 id, float xpos,float ypos,
        float xspeed,float yspeed,float maccel)
    {
        _id = id;
        _xpos = xpos;
        _ypos = ypos;;
        _tstamp = ofGetElapsedTimef();
    }
    void removeTuioCursor(int32 id)
    {
        if (distance(xpos,ypos,_xpos,_ypos)<MAXDISTANCE
            and
            ofGetElapsedTimef() - _tstamp < MAXTIME)
        {
            TeventSingleUserTapTabed * evt = new
                TeventSingleUserTapTabed();
            evt->x = _xpos;
            evt->y = _ypos;
            events.push_back(evt);
        }
    }
};

```

Listing 4.4: An example of a gesture recognizer designed for single-user only.

```
class CircleButton: public tuio::CanTap < Graphic >
{
    public:
    CircularArea a;
    CircleButton()
    {
        this->Register(a);
        a.x = 0.5;
        a.y = 0.5;
        a.r = 0.03;
    }
    void Tap(float x, float y)
    {
        // Do something
    }
    virtual void draw()
    {
        ofSetColor(255,0,0);
        ofCircle(a.x,a.y,a.r);
    }
};
```

Listing 4.7: A simple button created using Graphic and Area.

the areas that contained those events. Then the gestures are recognized by the gesture recognizer and its resulting event delivered to the correct receiver, the area owner.

Taking advantage from the area introduction, the notion of virtual graphic objects (Graphic) with area was introduced. Those objects simplify the creation of graphical event receivers such as interactive elements, buttons, and any other element of interest (see Listing 4.7 for an example).

Another extra feature that was introduced is the ability to use a simplified XML-based global configuration system (GlobalConfig), to get rid of *magic numbers* within the application.

A very important difference with the previous year was how gesture recognition was taught. We put more emphasis in the multiuser setting, and proposed a strategy to simplify their programming. First we present a simple single-user gesture recognizer implemented as a *state machine* that describes some states (one being the *dead* state) and the input events that transition those states (see Figure 4.5 for the diagram, and Listing 4.8 for its implementation).

After successfully coding a single-user gesture recognition as a *state machine* a meta gesture recognizer is created that:

```

class SM_MyTap
{
    public:
    enum State {dead, inside} state;
    int cursor_in;
    float xi, yi;
    float arrival_time;
    std::list<TEvent *> & events;
    SM_MyTap(std::list<TEvent *> & _events): state(dead),
        events(_events){}
    void newCursor(DirectFinger * f)
    {
        if (state == dead)
        {
            cursor_in = f->s_id;
            xi = f->getX();
            yi = f->getY();
            arrival_time = ofGetElapsedTimef();
            state = inside;
        }
    }
    void removeCursor(DirectFinger *f)
    {
        if (state == inside and f->s_id == cursor_in)
        {
            if (f->getDistance(xi, yi) < MAX_DISTANCE and (
                ofGetElapsedTimef() - arrival_time) <
                MAX_TIME)
            {
                SimpleCallEvent (CanMyTap, Tap, (f->getX(), f
                    ->getY()));
            }
            state = dead;
        }
    }
};

```

Listing 4.8: A single user *tap* gesture recognizer implemented as a state machine.

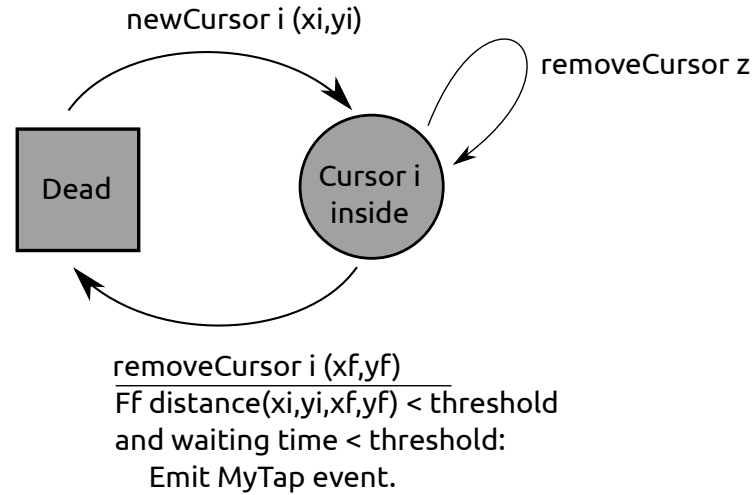


Figure 4.5: A simple single-user *tap* gesture recognizer state machine.

- Has a list of individual gesture recognizer *machines*.
- Every time an input event is received it is delivered to every machine. But before doing that, a new *pristine* machine is added to the list. After delivering the event, it removes the machines on the *dead* state (see Figure 4.6, Listing 4.9).

This strategy allows a much simpler programming, as gesture recognizers do not have to take into account multiple sources of events due to multi-user activity. Instead, the pattern itself manages this simultaneity. It is important to note that those single-user gesture recognizer machines should ignore any event unrelated to their gesture, and not take it as a proof of failure.

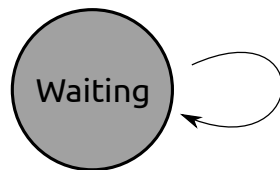
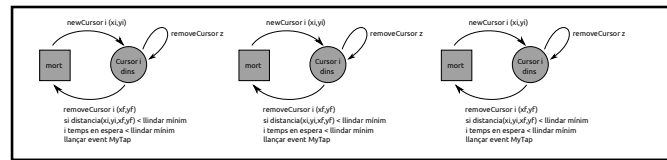
The resulting projects confirmed the good idea of centering the focus to autonomous objects and multi-user interaction. Some of them had an important number of interactive objects receiving complex gestures. As an example *Logic Gate Simulator* (by Leonardo Amico, Carlos Casanova, Álvaro Muñoz and Raul Nieves) provided with an arbitrarily large number of virtual elements representing logical gates that were manipulated independently with gestures such as on-finger drag, two-finger rotate, or X-crossing (see Figure 4.7).

4.1.4 Simplifying the API (2012)

The latest changes made to *ofxTable Gestures*⁸ were centered on a better API, and a simplification of the code and internal mechanisms. Some parts had then sufficient

⁸https://github.com/chaosct/ofxTableGestures/tree/1.X_series

State Machines list



Event

Add a State Machine.
For every State Machine:
Execute Event.
Delete dead State Machines.

Figure 4.6: Gesture recognizers are collections of simultaneous possible gestures.



Figure 4.7: Logic Gate Simulator

```

class InputGestueMyTap : public CanDirectFingers <
    CompositeGesture >
{
    public:
    std::list< SM_MyTap * > mes;
    InputGestueMyTap(){}
    void newCursor(DirectFinger * f)
    {

        mes.push_back(new SM_MyTap(events));
        for (std::list< SM_MyTap * >::iterator it = mes.
            begin(); it != mes.end(); ++it)
        {
            SM_MyTap * me = *it;
            me->newCursor(f);
            if (me->state == SM_MyTap::dead)
            {
                delete me;
                (*it)=NULL;
            }
        }
        mes.remove(NULL);
    }
    void removeCursor(DirectFinger *f)
    {
        mes.push_back(new SM_MyTap(events));
        for (std::list< SM_MyTap * >::iterator it = mes.
            begin(); it != mes.end(); ++it)
        {
            SM_MyTap * me = *it;
            me->removeCursor(f);
            if (me->state == SM_MyTap::dead)
            {
                (*it)=NULL;
            }
        }
        mes.remove(NULL);
    }
};

```

Listing 4.9: Meta-recognizer for a tap gestures. It has a list of single gesture recognizers to simultaneously handle several gesture instances.

entity for themselves to be separated as a distinct add-on. *ofx2DFigures*⁹ is used to create and display polygons and *ofxGlobalConfig*¹⁰ addresses the need of a simplified configuration system.

One of the relevant changes made in this version was eliminating the separation of gesture code and program logic in two different threads. After all the previous experiences with all the previous projects, it seemed clear that gesture recognition, at least when programmed with the mentioned strategies, was not as CPU consuming as we initially thought. Eliminating this separation has the consequence of not requiring anymore an event queue and thus the serialization of events, which was a key constraint in the implementation.

At its turn, without the requirement of serialization, we no longer require our own implementation of an event system. The standard OF event system can then be used instead, with the benefit of its much cleaner interface and implementation. See for instance how Listing 4.10 registers to *tap* events with a simple function call (in line 8).

Taking advantage from the new events interface, we introduced another event that is not related to user input, but very useful for gesture recognition: time alarms. With this event a recognizer can very easily dismiss failed gestures with timeouts, instead of waiting for the next event to transit to the dead state.

The resulting projects were similar to the previous year in terms of complexity and success. As a consequence of the API simplification, we received far less concerns about the confusing or *magic* API than previous years.

A problem that was identified in the last two years (2011, 2012), is related to the *state machine* strategy for multi-user gesture recognizers. As the single-user state machines are independent from each other, there is no easy way to share information between them. A typical problem will arise with, for instance the double tap gesture: three consecutive taps result in two independent double tap gestures (one using the 1st and 2nd tap, and the other using the 2nd and 3rd). A central register of used taps can be useful, and often is the solution that spontaneously emerges, but this only addresses the coordination between instances of the same gesture.

The overlapping gesture problem also happens between instances of different gestures: a tap recognizer and a double tap recognizer will not coordinate to get either one or the other. A class registered to both gesture events will receive two separate tap events along with every double tap.

⁹<https://github.com/bestsheep1/ofx2DFigures>

¹⁰<https://github.com/chaosct/ofxGlobalConfig>

```
1 class Button: public Graphic
2 {
3     public:
4         float x,y;
5         Button()
6         {
7             x = y = 0.5;
8             registerMyEvent (InputGestureTap::I().Tap, &Button::
              tap);
9         }
10        void draw()
11        {
12            ofSetColor(255,0,0);
13            ofCircle(x,y,0.1);
14        }
15        void tap(InputGestureTap::TapArgs & args)
16        {
17            // Do something
18        }
19        bool Collide(ofPoint const & point)
20        {
21            float distance = point.distance(ofPoint(x,y));
22            return distance < 0.1;
23        }
24    };
```

Listing 4.10: A simple button created using Graphic. Notice how registering events is no longer done with mixins (for instance Listing 4.7) but with the observer pattern.

year	abandoned course	total	abandon rate
2009	12	38	31,58%
2010	2	32	6,25%
2011	5	42	11,90%
2012	2	26	7,69%

Table 4.1: Abandon rate per year in TSI course

The alternative is to implement all the gestures in a single gesture recognizer, but this breaks the whole modularity and compositions of the gestures. The idea of creating a mechanism for automatically resolving this overlapping situation was the spark that started the development of *GestureAgents* (see Chapter 5), which is heavily influenced by this problem.

4.1.5 Discussion

After those years providing a framework for the use of students in a tabletop application creation course, the experiences lead to some reflections on the useful parts of such framework and its consequences.

It seems that the most important feature introduced to the framework was actually providing the needed libraries to load images, play sounds etc provided by OF. Apart from the experience related to the complains and encountered difficulties, a sign of that can be that a higher proportion of students abandoned the course the first year compared to the later years (see Table 4.1), although this data should be treated with caution as many other causes can explain this difference.

Although important, we think that forcing the separation of gesture code and program logic, the other important change made in the second year, did not have a relevant impact on abandon rates, in contrast with providing library. Instead we think that gesture infrastructure tend to impact mainly on the quality and complexity of the projects in a gesture interaction level, while the lack of very basic capabilities (such as image loading libraries) are perceived as essential parts of the projects and can be key to desisting.

On the impact of gesture infrastructure, we have seen that the gesture complexity and quality are tied to the provided basis. As this basis is developed and enhanced, the needed effort to reach such quality decreases. This is visible with the impact of multi-user gesture strategies, area capabilities, and autonomous interactive virtual objects facilities.

The fact that much of the effort on this framework was devoted to provide gesture recog-

dition infrastructure is not an arbitrary choice, the experience shows that programming and testing gesture recognizers is the most difficult and problematic task for the students. This has many causes, such as the abstract nature of event-based programming or the impossibility of accurately test gestures in the simulator due to the limitations of mouse input.

The problem of having independent, easy to write, composable, modular gesture recognizers is not trivial and has proven challenging. A continuation of this effort is done in *GestureAgents*, Chapter 5.

In conclusion, we think that this framework has effectively addressed the important issues when programming multi-user gesture-expressive tabletop applications, in the context of unexperienced programmers. Apart from the students of this course, the framework has been used in other external projects, such as master thesis or projects for other departments (such as (Vouloutsi et al., 2014)).

4.2 MTCF: a platform for sound and music tabletop creation

Apart from the need of frameworks by the application programmers, we identified that artists and other *non-programmers* also need some kind of facilitation to adapt their works into a tangible tabletop application. The fact that some tangible applications related to specific fields such as art and music already exist, such as the *Reactable*, also creates the need to modify them adding additional functions or objects by people from related fields, such as sound and music artists.

As we have seen earlier, creating a graphical interactive user interface for tangible tabletops is usually already a difficult task, even if using frameworks that facilitate this kind of programming. Additionally, sound artists work hard on the auditory part, as it is the core of their contribution, probably in another specialized programming language. Taking into account these considerations, it may be difficult to acquire the required skills for being capable of programming the visual interface and the audio component, or to find a single programming language or framework supporting well these two components.

A simple solution to this last problem, as presented by Hochenbaum et al. (2010) or Fyfe et al. (2010), is to divide the project into two different applications: one focused on the visual feedback and another focused on the audio and music processing, as this allows having different programming languages and frameworks. However, to divide the tasks will not itself eliminate the need of programming on both sides. Musical Tabletop Coding Framework (MTCF) (Julià et al., 2011) was designed to simplify these technical

difficulties.

MTCF began with a specific need. As we were mentoring master students of the *Sound and Music Computing* (SMC) and *Cognitive Science and Interactive Media* (CSIM) masters, some of their final projects were related to music; and, in particular, as we usually worked with the Reactable, related to music or sound effects that would be added to the current Reactable system.

While those projects were meant to be integrated to the current code of the Reactable, this integration was far from easy: the Reactable program itself was divided in several parts (graphical, messaging, audio) that shared some configuration files but not all, many of the parameters were hard-coded, and creating the code to add new functions and objects required very advanced techniques. At the end, the amount of work needed to integrate the finished prototype into the Reactable ecosystem was unassumable for a master student with no prior knowledge of the inner working of the Reactable.

This issue was the original drive to create MTCF. It consisted basically of a front-end graphical engine that mimicked the way Reactable presented controls and sound waves, and communicated with Pure Data, where the sound was computed. This first version is described in Section 4.2.1. After some time, MTCF evolved to provide more possibilities to creators, drifting away from the original Reactable-like interface and allowing more freedom. This second version is presented in Section 4.2.2.

4.2.1 A Reactable-like playground

MTCF is an open source platform¹¹ for the creation of musical tabletop applications that takes a step forward in simplifying the creation of tangible tabletop musical and audio applications, by allowing developers to focus mainly on the audio and music programming and on designing the interaction at a conceptual level, as all the interface implementation will be done automatically.

MTCF provides a standalone program covering the visual interface and the gesture recognition functions, which communicates directly with Pure Data (Puckette, 1996) (Pd), and which enables programmers to define the interactive objects and their control parameters, as well as the potential relations and interactions between different objects, by simply instantiating a series of Pure Data abstractions.

The choice of Pd as the programming language for sound artists is not arbitrary; apart from being open source, Pd is also usually taught in many sound, music and interaction

¹¹<https://github.com/chaosct/Musical-Tabletop-Coding-Framework/releases/tag/0.1b>

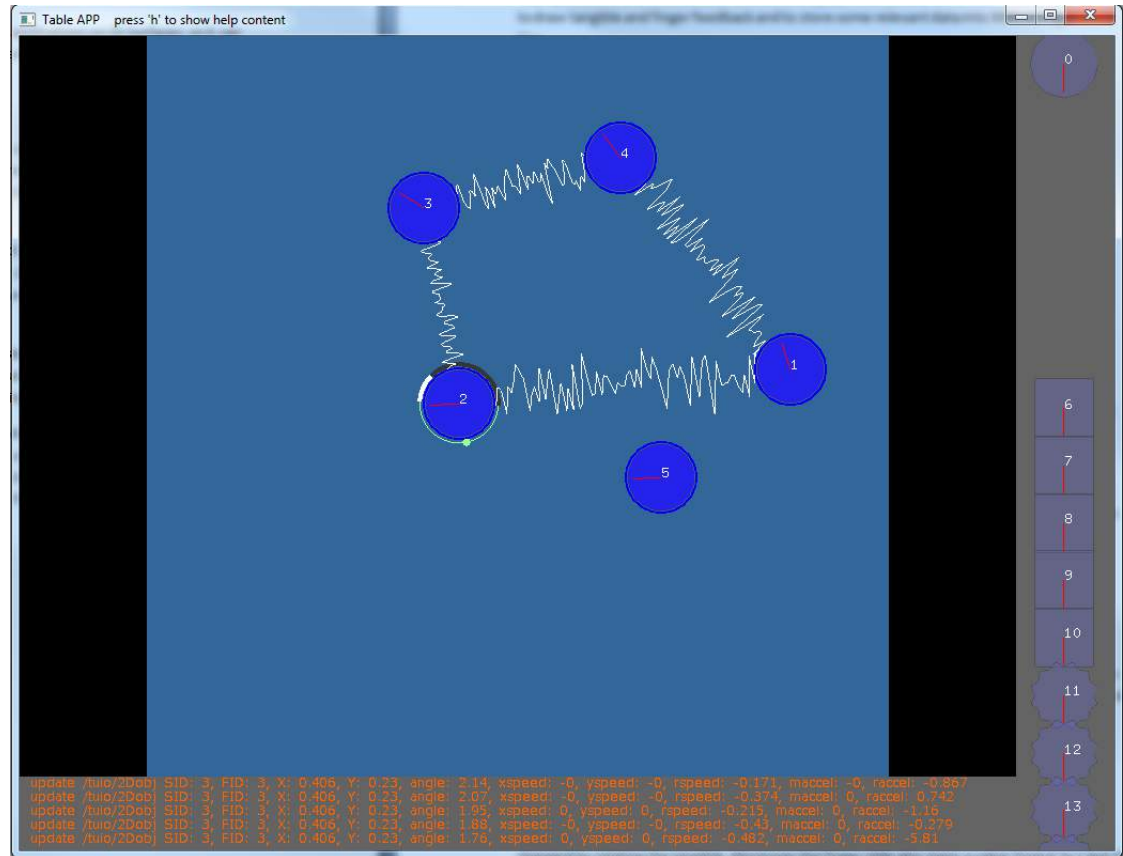


Figure 4.8: MTCF Simulator.

courses in various masters. It is broadly used by artists dealing with sound, music or image. It is also very similar to its very popular closed source equivalent, MAX/MSP, also from the same author, so artists and musicians knowing MAX/MSP can very easily adapt to it.

Dealing with input data and the GUI

As MTCF is implemented with *ofxTableGestures* (see Section 4.1) and provides means for creating tabletop applications, it can be seen as a specialized and simplified subset of *ofxTableGestures*; while it does not provide all of *ofxTableGestures*' functionalities, it simplifies enormously the programming tasks by putting everything on the Pd side. Of course, one of the advantages of using this add-on is its simulator that helps testing the programs without the need of accessing the tabletop hardware (see Figure 4.8).

MTCF receives data from the TUIO tracking application (i.e. reactIVision), processes

4.2 MTCF: a platform for sound and music tabletop creation

it, displays the graphic feedback and sends the filtered data to Pd via OSC messages. At this stage, MTCF only draws the figure shapes and the fingers' visual feedback, all in their correct positions. The remaining graphical elements (such as the waveforms and the relations between the figures) are drawn on demand, according to the additional information that is sent back via OSC messages from Pd to MTCF (see Figure 4.9).

By default, MTCF pucks only convey three basic parameters: X position, Y position and rotary angle. Two additional touch-enabled parameters can be enabled for any specific object in Pd, the object bar and the finger slider, which are displayed as two semicircular lines surrounding the puck, keeping the orientation towards the center of the table, as shown in Figure 4.10. Also, parameters resulting from the relations between pairs of pucks (distance and angle) can be activated in Pd. Coordinates and distances ranges relate to the coordinate system in *ofxTableGestures*, which considers the interface to be a circle of diameter equal to 1, centered in (0.5,0.5). The object bar conveys a value between 0 and 1 that can be changed by rotating the tangible. The finger slider, represented by a thinner line with a dot that can be moved using a finger, also ranges between 0 and 1.

Using MTCF from Pd

MTCF was designed to be used along with Pd, as it has become one of the most popular languages for real time audio processing and programming. The main idea of this framework was to allow expert Pd users to interface their patches using a tangible tabletop setup. For this, MTCF provides nine Pd abstractions that transparently communicate with MTCF graphical interface, and that will be used to define the objects, the relations between them, and the data the programmer wants to capture from the tabletop interface. Not all of these abstractions have to always be used, as this will depend on the characteristics of the designed musical application interface.

Only one abstraction is mandatory, the one responsible for all OSC communication between the Pd patch and MTCF: `[basicTangibleTabletop]`. Its single argument is the address of the computer running MTCF. This would typically be *localhost*, although changing this address can be useful in some situations, as when working with a shared tabletop. Because of implementation details, one and only one instance of this object must exist in the Pd program.

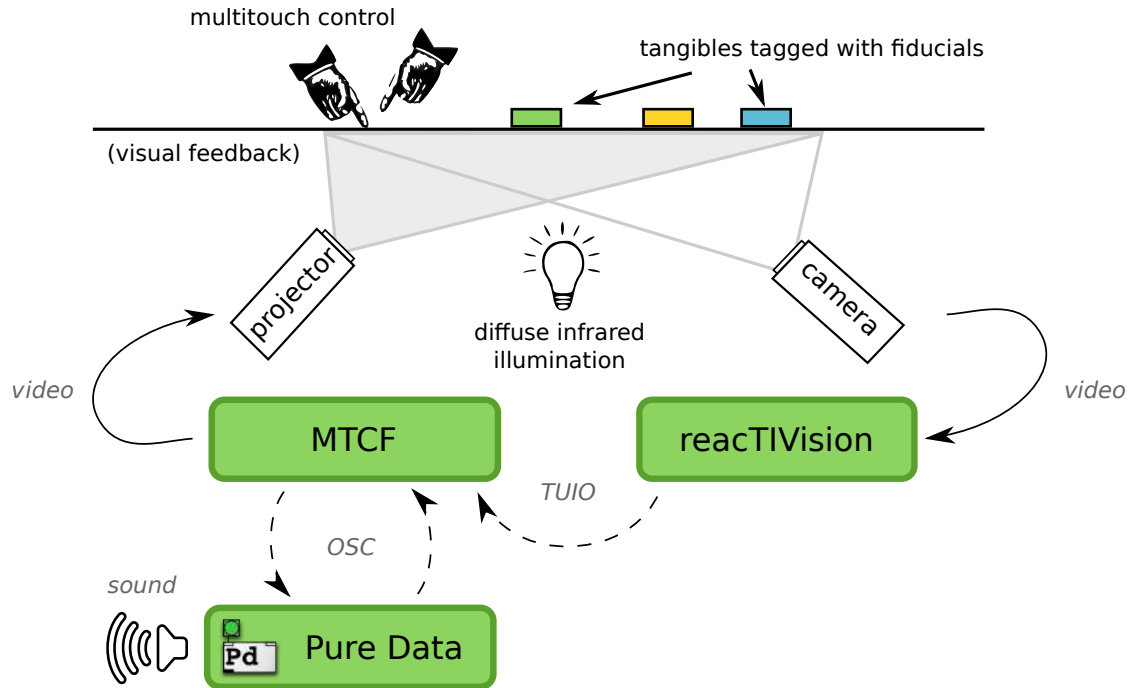


Figure 4.9: MTCF data flow.

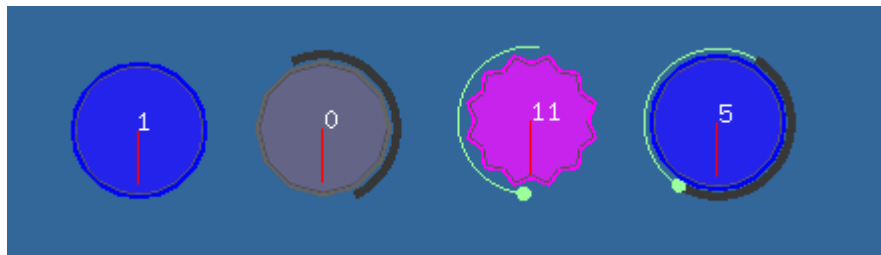


Figure 4.10: Tangibles with different feedbacks and controllers. From left to right, an object without extra controls, an object with a bar, an object with a slider, and an object with both a bar and a slider.

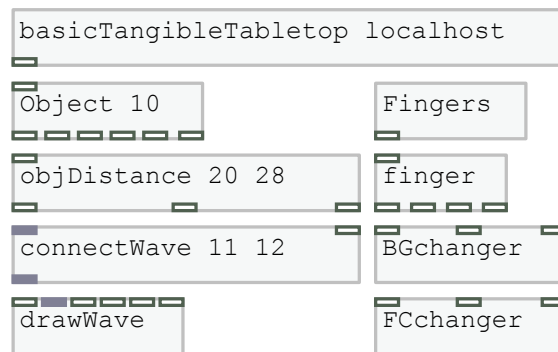


Figure 4.11: MTCF Pd Abstractions.

Defining objects and parameters

The programmer can use some additional abstractions to define what physical pucks will be allowed for use on the tabletop. By instantiating `[Object n]`, a programmer allows the puck with the fiducial id code n to be used. A slider and a $[0, 1]$ rotatory parameter can be (de)activated around it, if desired, by sending messages to it. Only when these elements are active Pd will receive this additional information. Outlets in `[Object]` output the presence of the puck (Boolean), its position, orientation, and if configured, its slider and rotary parameter values.

Inspired by the Reactable paradigm, which allows the creation of audio processing chains by connecting different objects (such as generators and filters), MTCF also allows to use the relations between different pucks and makes them explicit. However, unlike in the Reactable, MTCF is not limited to the creation of modular synthesis processing chains; any object can relate to any other object independently of their nature. This allows, for example, to easily create and fully control a tangible frequency modulation synthesiser (Chowning, 1973), by assigning each carrier or each modulator oscillator to a different physical object; or a Karplus-Strong plucked string synthesiser (Karplus and Strong, 1983) by controlling the extremes of a virtual string with two separate physical objects. On the counterpart, MTCF do not facilitate Pd dynamic patching (Kaltenbrunner et al., 2004), so it is not capable of producing a fully functional Reactable clone easily, neither was this its main objective. In MTCF, the connections between the pucks are made explicitly by the programmer in the Pd programming phase. This is attained by using `[objDistance m n]`, which continuously updates about the status of this connection, and (if existent) about the angle and distance between objects m and n . The programmer can also specify whether she wants this distance parameter to be drawn on the table by sending a Boolean value into the `[objDistance]` inlet.

Drawing Waves

Also inspired by the Reactable, MTCF can easily show the sound (in the form of waves) going from one object to another. This can be achieved by using the `[connectWave]` object. This abstraction takes two parameters that indicate the object numbers between which the wave should be drawn. As indicated before, this waveform does not necessarily indicate the sound coming from one object into the other, but can rather represent the sound resulting from the interaction between two combined objects, or any other sound thread from the Pd patch. An audio inlet and an outlet are used to take the waveform and to act as a gate, allowing the audio to pass only if the two pucks are on the surface

(and therefore the waveform exists). This ensures that no unintended sound will be processed neither shown, when its control objects are removed. Additionally, a control inlet lets the patch to activate and deactivate this connection.

This way of drawing waveforms has some consequences: first, waveforms are drawn by default between pucks, complicating drawing waveforms directly between 2 arbitrary points, such as from one object to the center, as the Reactable does. This can be overcome by using a simpler Pd abstraction, `[drawWave]` with this very purpose, drawing waves between 2 points. The second but very important consequence is that the audio connection between two physical pucks is a Pd object. Instead of having Pd audio connections between `[Object]` abstractions, the programmer must therefore use `[connectWave]` abstractions, which simply send the waveform information to MTCF for drawing. This can be confusing, specially when chaining multiple physical pucks imitating an audio processing chain: the programmer must then consider all the possible connection combinations (see Fig. 4.12 for an example).

Extra features

For more advanced interaction with fingers, additional abstractions are also provided. `[Fingers]` gives full information of the position of all fingers detected on the table, while `[finger]` can be used to extract individual fingers information (see Fig. 4.13). These abstractions can be used to control less obvious parameters.

Two additional abstractions can be used for visual purposes: `[BGchanger]` and `[FCchanger]` respectively allow changing the background color of the tabletop and the color of the fingers' trailing shadows. Changing colors, for example according to audio features, can create very compelling effects.

Example projects

Fig. 4.14 shows (a) a Karplus-Strong generator using two objects for controlling the length of the string and one additional object as the plucking pick, (b) a midi controller for Ableton Live¹², and (c) a speed-controlled wave player with a band-pass filter. All three examples, developed in a half-day workshop, achieved interesting sound and control results.

Another example is the implementation of a vowel synthesis engine in Pd made by a SMC master student. He used MTCF to test it in real time as an interactive instrument,

¹²<https://www.ableton.com/en/live/>

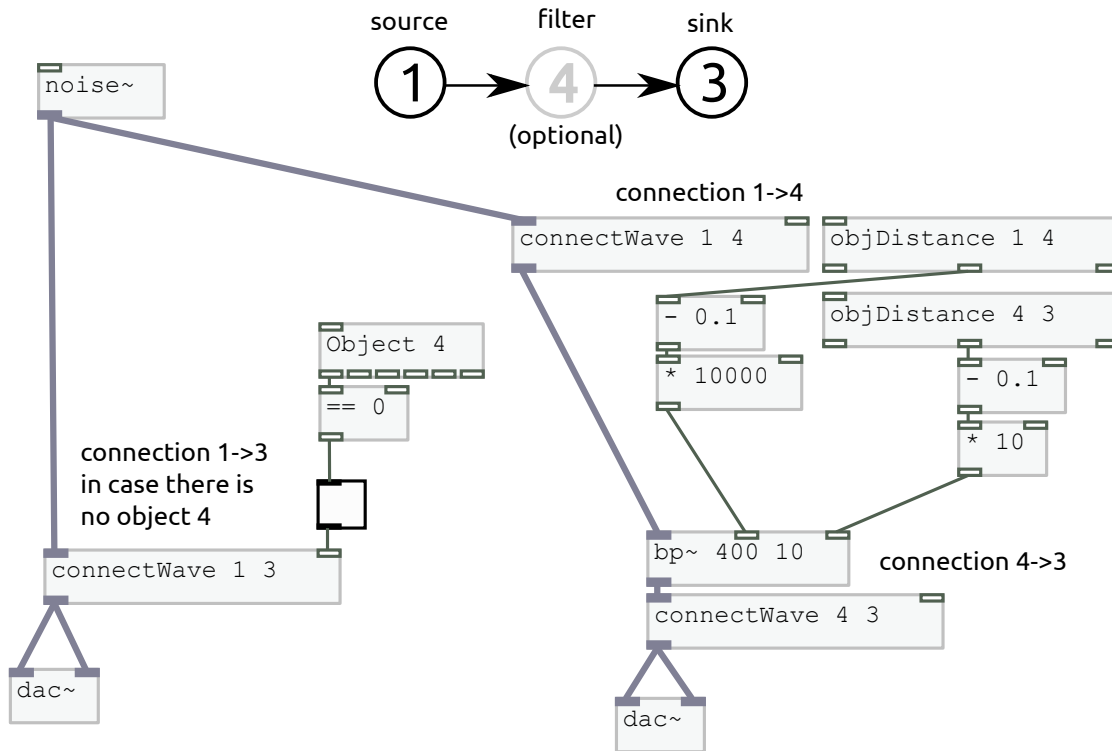


Figure 4.12: A processing chain example. A puck (1) is a noise generator, another (4) is a filter, and the remaining one (3) is an audio sink (i.e. the speakers). The programmer must consider the connections when puck number 4 is present ($1 \rightarrow 4 \rightarrow 3$) and when it is not ($1 \rightarrow 3$).

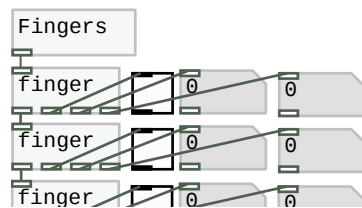


Figure 4.13: A Pd structure to receive information of the several fingers on the surface.



Figure 4.14: Free project results: (a) a Karplus-Strong generator, (b) a MIDI controller and (c) a band-pass filter.

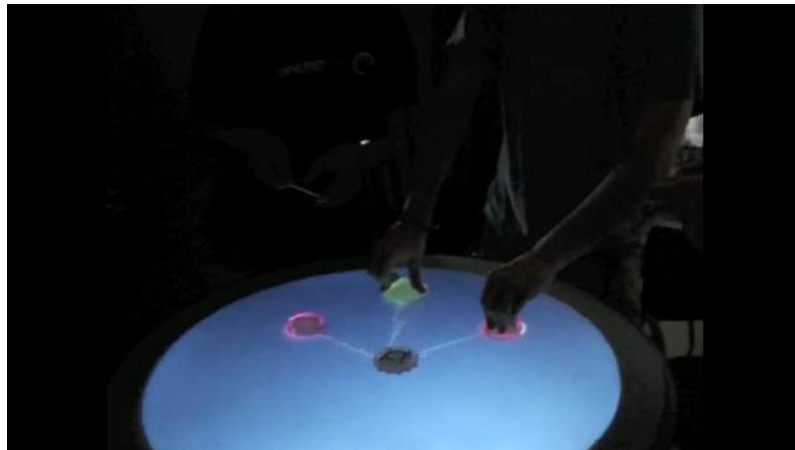


Figure 4.15: *Our little choir* vowel synthesis engine in MTCF.

called *Our little choir*¹³ (author: Alexandros Katsaprakakis) with excellent results (Figure 4.15).

Those and other previous experiences indicate that MTCF is not only a very valuable tool for the quick development and prototyping of musical tabletop applications, but also an interesting system for empowering discussion and brainstorming over some concepts of software synthesis control and interaction.

4.2.2 Allowing interface design

As useful as MTCF was, users often asked for features that were not present, such as drawing buttons and other GUI elements other than using objects as controls. To open up the user base, we decided to add functionality by adding the creation of virtual elements on the surface. This would also be useful for students of *Curs de Disseny de*

¹³Best seen in video: <https://vimeo.com/groups/main/videos/25966968>

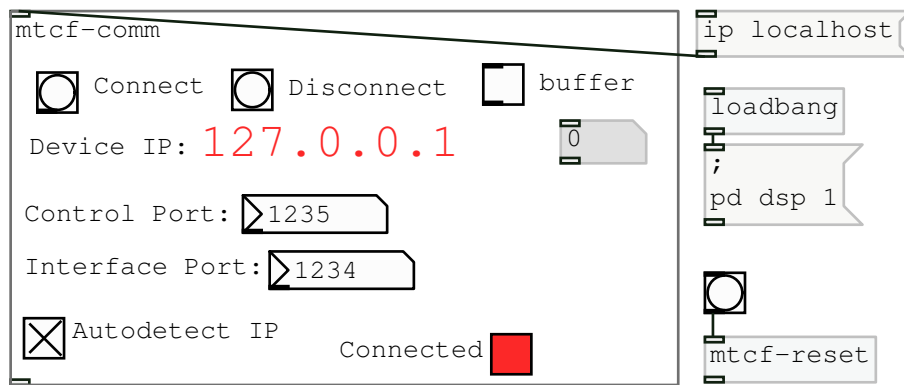


Figure 4.16: Common MTCF elements.

Sistemes Interactius Musicals (CDSIM) and *Master Universitari d'Art Digital* MUAD masters, with a less sound-processing-centered background. The core idea of this change consisted on providing functions to create virtual elements such as polygons and text, which would be displayed on the surface and be interacted with. The results of the interaction would be sent to Pd again to be used in the program.

Common elements, renamed

To homogenize the Pd abstractions' names and to prevent name collision with objects bundled with Pd, all the abstractions' names were named starting with the *mtcf-* prefix. This includes the original abstractions from the previous version, for instance [basicTangibleTabletop] would become [mtcf-comm]. Also, new functionalities were included to ease the connection between the MTCF front-end and the Pd client: buttons to connect and disconnect the client, *connection* status display, or a message buffer activation switch (see Figure 4.16).

Virtual elements: polygons and text

Two types of graphical elements were introduced in MTCF: polygons and text. Those can be declared and modified in Pd, and then automatically displayed in the front-end. The number of those elements is unlimited, and can be individually controlled within Pd.

Polygons can be created with [mtcf-polygon] and have arbitrary shapes, defined by the positions of their vertices with [addvertex (messages. Helpers exist to ease this

4 Empowering application developers

process in common cases such as squares, rectangles and circles (see Figure 4.17). Their appearance can also be defined by choosing their color (solid or translucent), their stroke presence and color, etc (see Figure 4.18). Additionally, textures can be used as an infill of the figures.

Polygons and Texts can be transformed through the surface: they can be translated, rotated, and scaled to the desire of the programmer (see Figure 4.19). These transformations can be chained, in a way similar to Gem (Danks, 1997) or OpenGL; the transformation operations are accumulated in a 2D transformation matrix, so translating and rotating is different from rotating and translating (see Figure 4.20).

Interaction inside polygons is captured by the front-end and exposed in Pd through the outlet of `[mtcf-polygon]`. Outgoing messages prepended with *finger* convey the finger information related to touches inside the polygon and can be unpacked with chained `[mtcf-finger]` objects. This is very useful to create interactive elements such as buttons.

Text elements are instantiated with `[mtcf-text]` and transformed in the same way as polygons. The color can also be defined likewise. A `[write Some text (message` changes the caption to *Some text*. Apart from normal scaling, the special Pd object `[mtcf-textsize]` is provided to uniformly scale all texts.

Projects

The ability to create virtual interactive elements and text empowered the users to create a greater variety of tabletop application interfaces better suited to the interaction needs of their applications. This was extensively used by CDSIM students, as they had a specific teaching module for tabletop interaction, where they had to create a tabletop musical instrument. The results ranged from static button-based interfaces, such as *Electrobichos* (Authors: Mauricio Iregui and Nicolás Villa), through zoom interfaces, such as *Zoom Interactivo* (Authors: Leon David Cobo and Patricia Sanz), to more experimental dynamic physical simulation interfaces such as *MTCF-Gravity* (Eloi Marín), all depicted in Figure 4.21.

Conclusions

What started being a way to cover a specific need, such as experimenting with new functionality in the context of the Reactable, was at the end enhanced to be useful for other collectives that wanted to create tabletop applications other than mere additions to the Reactable.

4.2 MTCF: a platform for sound and music tabletop creation

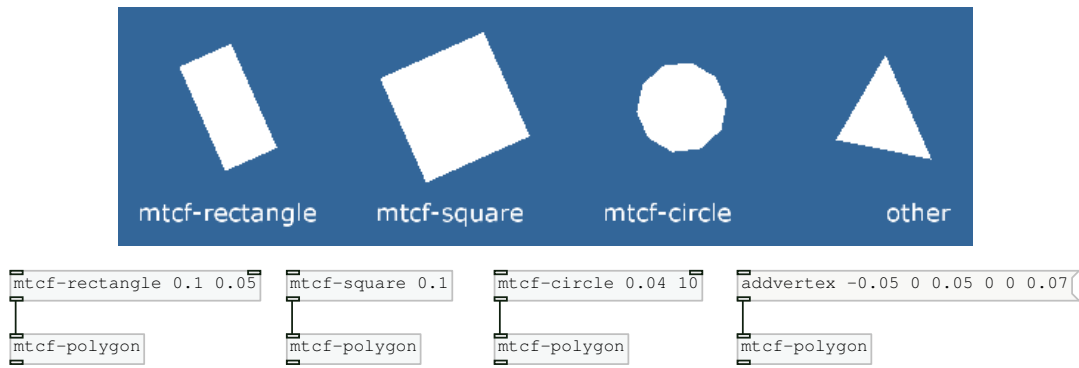


Figure 4.17: MTCF Polygon shapes.

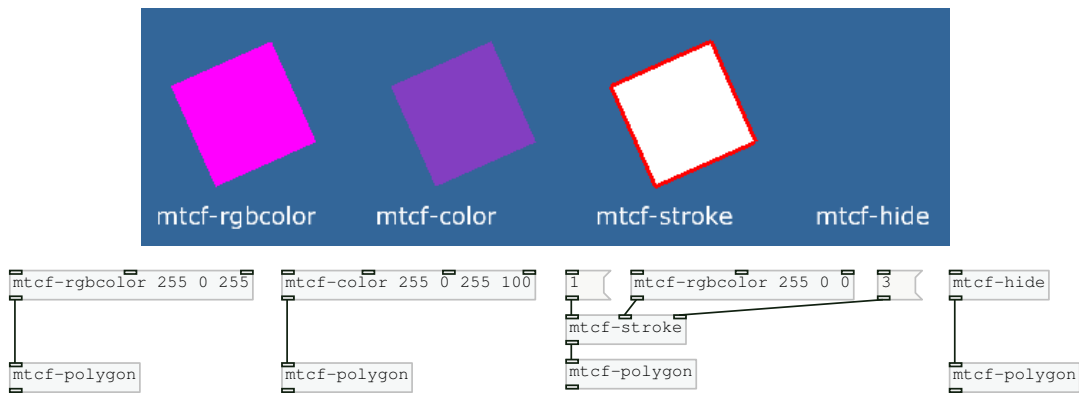


Figure 4.18: MTCF Polygon appearance options.

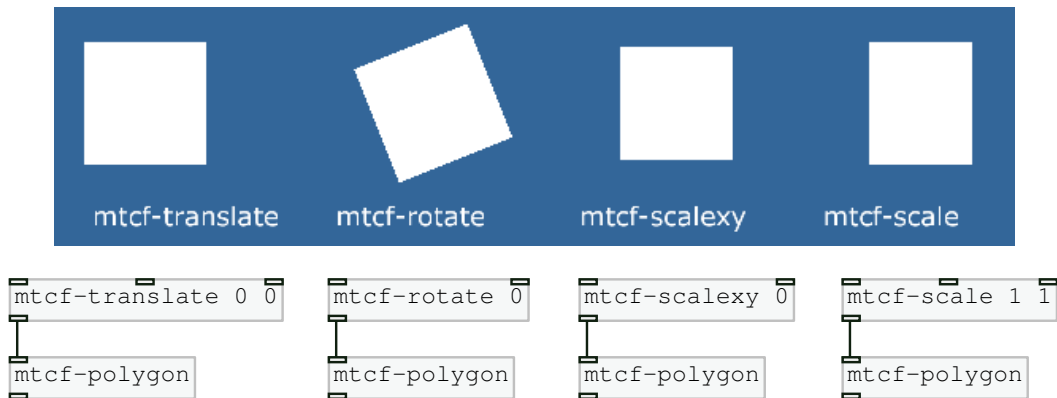


Figure 4.19: MTCF transformations.

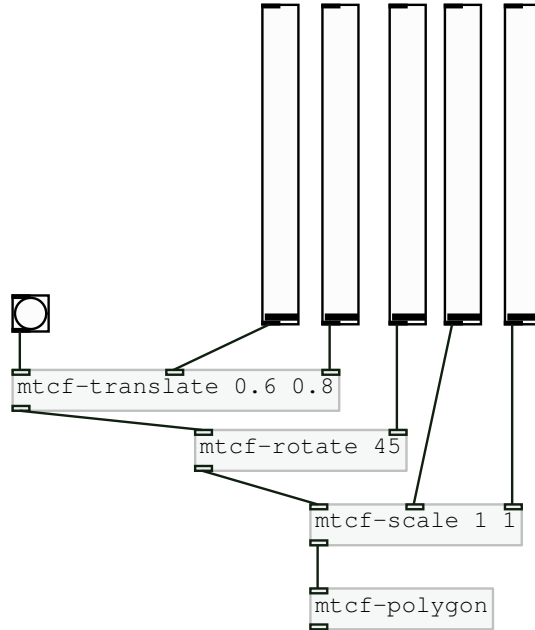


Figure 4.20: MTCF transformations can be chained.

It seems that the potential usefulness for tabletop interfaces for music-related users is specially high. The fact that many different collectives profited from this framework, even given that, because of the nature of the device, their development would probably not be used again, shows that there exists an interest.

We can extract the observation that the usefulness of tabletop applications is not necessarily linked to its gestures' complexity. Even though in section 4.1 we stress in this aspect and its benefits, the experience with MTCF shows us that some activities favor other types of gestures that do not require complex gesture recognizers. The difference between the two is whether the gestures are or are not eminently symbolic. An application that identifies commands and instructions performed by users, given the trajectory of their interactions (for instance if it tries to distinguish a drawn circle from a cross), will need specific code to recognize whether a sequence of events effectively matching a gesture is present. Otherwise, an application that creates a direct mapping between the trajectory and a particular parameter of the application (for instance in sound synthesis), in spite of the expressiveness of the gestures; does not need to identify any particular feature of the gesture to get the desired effect.

This type of interaction, direct mapping, is specially useful in expressive and artistic

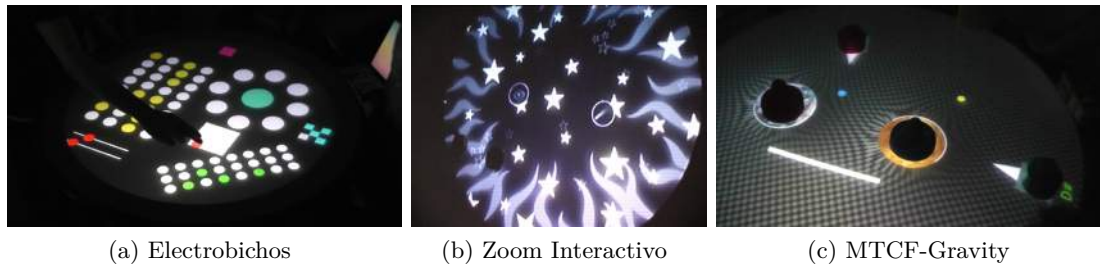


Figure 4.21: CDSIM example projects using MTCF

applications, as the control is maximum and direct, promoting an intimate relationship with its result. Sound and music specially benefit from this relationship. Unfortunately, direct mapping does not cover the needs of having a big action dictionary or multiplexing the interaction by the gesture type, and so other activities requiring a richer and formal interaction language cannot benefit from it directly.

So, regarding the two presented frameworks, we can say that they not only address different collectives, but also, they address different needs. MTCF will be more useful for applications with direct mapping, such as artistic and expressive, while *ofxTableGestures* will better address the problems from creating applications with formal complexity.

4.3 Conclusions

We developed two frameworks supporting the creation of tabletop applications, focused in different collectives with different needs: programmers and artists. Through the development we adopted an iterative developing strategy based on the feedback of our users: students. A continuous feed of new developers allowed us to enhance and fix our frameworks every year. This strategy proved useful.

The *ofxTableGestures* experience shows us the important role of third party libraries to speed up work on menial and uninteresting jobs and focus on the actual application. It also shows us how providing infrastructure that enforces good coding practices helps programmers to achieve good results, and how gesture programming is a difficult task that can benefit greatly from such good practices.

The MTCF experience shows us that there was a need for artists to have a low-entry point for tabletop programming. This need, although originally focused on enhancing existing applications, soon evolved to allow totally new, unrelated work.

The results of new developers making tabletop applications in both frameworks, show

a high preference for game and musical applications. This is expected, as the most evident tabletop affordances are direct manipulation, spatial multiplexing and real-time interaction. Something that is very useful in games and music.

We have approached gesture recognizer programming with *ofxTableGestures* and seen that, despite the several improvements to make it easy to code, some problems still persist. Those mentioned problems are related to the disambiguation between gestures, or, in other words, how to know if some sequence of input events relate to one gesture or another, but not both. This same problem is central when having multiple applications running at the same time in a tabletop: input events need to be distributed to the applications' gesture recognizers. Next Chapter, *Multi-Application systems: GestureAgents*, addresses this problem in both circumstances, disambiguation inside an application and between applications.

5 Multi-Application systems: GestureAgents

Until now, we have been working on tabletops that presented a single application to the users: both the applications presented in Chapter 3 and the frameworks from Chapter 4 deliberately take all the space on the tabletop surface and capture all the input data. In this chapter we deal with the problem of having multiple applications running simultaneously in a single computing device that presents a shareable interface. We explain why this problem is relevant, and how we attempt to solve it by creating the GestureAgents framework.

5.1 Introduction

Shareable interfaces are a common subject of study in the field of CSCW. Tabletops and vertical displays, for instance, are considered, in many ways, a good approach to promote collaboration, a circumstance that is valuable for solving complex tasks.

In the personal computer context (still the leading professional platform), complex task solving is often supported by the use of a combination of several unrelated software tools. However, the systems developed to study collaboration in shareable interfaces usually feature a single ad-hoc application that tries to cover all the aspects involved in the particular tested task. This approach is valuable for studying many mechanisms of collaboration, as it constitutes a controlled environment in which the interaction dynamics can be tested, however, it still does not really represent existing real-world practices.

Previous experiences in other kinds of interfaces, such as PCs or hand-held devices, suggest that the real world use of new general purpose computing devices will need some kind of multi-tasking capabilities if these aim to support general and potentially complex task solving features and if, in short, these aspire to become useful to the general

public. And yet, multi-tasking features in shareable interfaces may have deep differences even within single-user ones.

5.2 Collaboration in shareable interfaces

When designing collaborative computing appliances, we are in risk of losing essential elements previously present in personal computing and ending up with a useless system. Reviewing the features that made personal computing useful to people can help us to prevent the latter from happening. The qualities that contributed to make the personal computing platform successful and useful to its users are partially analyzed in Sections 2.1 and 2.2.3. We focused on multi-tasking as a key element to such systems, as we feel that it is often overlooked in current collaborative computing systems.

Collaboration has also been traditionally tied to complex task completion: group meetings are a common strategy to shed light into difficult problems. Big problems can be divided into smaller ones that can be redistributed (Strauss, 1985; Schmidt and Bannon, 1992), and points of view can be exchanged (Hornecker and Buur, 2006). Even in the computer era, the practice of physical meetings seems to be still (if not more than ever) prevailing. Empowering collaboration with computers is the primary goal of Computer Supported Collaborative Work (CSCW) field, and it relates directly to this group meeting problem. In this discipline, two different (but intersecting) problems are studied: In co-located CSCW all group individuals are present in the same workspace while in remote CSCW individuals are located in different places and all personal interaction is mediated by computers. Both problems deal with several users interacting with the same (local or distributed) system, leading to multi-user interaction.

Non co-located settings for multi-user interaction in a single virtual workspace, such as web-based collaborative systems (Bowie et al., 2011), or general cases of collective distributed work on single documents (groupware) (Ellis and Gibbs, 1989), are very common and widely studied. Co-located collaboration around computers, on its turn, already exists on a daily basis. Work meetings are often complemented with laptops, tablets, smartphones and other computing devices.

Needless to say, desktop and laptop computers have not been designed for co-located multi-user interaction, but for individual usage. Since they feature a single keyboard and a single pointing device, when used in multi-user setups computers inevitably lead to an interaction “bottleneck” with the users (Stanton et al., 2001; Shaer and Hornecker, 2010).

The use of computers in this context is thus still individual, lacking the social affordances that can be provided by “shareable” interfaces, or systems that have specifically been designed for co-located collaboration. Affordances, which according to scholars such as Hornecker and Buur, should particularly consider Spatial interaction and Embodied Facilitation (Hornecker and Buur, 2006).

Shareable interfaces, on their side, alleviate the interaction “bottleneck” by creating multiple interaction points, preventing individuals from taking over control of the computing device (Hornecker and Buur, 2006). Multiple interaction points do also promote user participation, lowering thresholds for shy people (Hornecker and Buur, 2006), and can provide means for bi-manual interaction, promoting a richer gesture vocabulary.

Typical types of interfaces developed for these collaborative scenarios are tabletop interfaces, which allow users to interact with horizontal displays using touch and/or pucks; vertical interactive displays (such as interactive whiteboards) in which users interact using pens or touch; tangibles which allow users to interact with physically-embedded artifacts and tokens (Rogers et al., 2009; Shaer and Hornecker, 2010); or body gestural interfaces, such as camera-based systems, which allow users to interact using their bodies (Shaer and Hornecker, 2010).

As a distinct characteristic, all these interfaces allow users a shared access to the same input and output physical interfaces, as opposed to typical groupware systems, where each user has its own interface device (Rogers et al., 2009). Besides collaboration, these shareable interfaces also show affordances more directly related with complex task completion. Epistemic actions, physical constraints, and tangible representations of a problem may contribute to problem solving and planning (Shaer and Hornecker, 2010). Spatial multiplexing allows a more direct and fast interaction (Fitzmaurice, 1996) while leveraging the cognitive load (Shaer and Hornecker, 2010); tangible objects facilitate creativity (Catalá et al., 2012); and rich gestures lighten cognitive load and help in the thinking process while taking advantage of kinesthetic memory (Shaer and Hornecker, 2010) (these strengths are discussed in Section 3.1).

This combination of social and personal affordances suggest that shareable interfaces are indeed well suited for complex task completion: apart from promoting collaboration, they provide individual and collective benefits that help completing these goals.

5.3 Multi-Tasking Shareable Interfaces: Current Situation and Related Research

Despite all the aforementioned affordances, and considering all the multi-task desirable properties, the majority of the currently available shareable interface systems created for research purposes consist of a single program that already includes all the necessary facilities to cover every subtask of the main activity. This is, however, consistent with the purpose of most research, because, in a collaboration co-located setting, CSCW researchers typically focus their investigations on the human factors in multi-user interaction, such as how input devices can be more effectively distributed between users in order to optimize group dynamics (Kim and Snow, 2013; Verma et al., 2013), or on studying different strategies to access digital and physical items from the perspective of digital content sharing (Verma et al., 2013), control sharing (Jordà et al., 2010; Kim and Snow, 2013), or proxemics (Ballendat et al., 2010).

A similar enclosing phenomenon happens with real-world products using shareable interfaces. While some of them focus on a very specific domain, avoiding to address more general problems (e.g. the Reactable (Jordà, 2008) approaches collaboration from the very specific and peculiar needs of musical collaboration (Xambó et al., 2013)), many others, such as interactive whiteboards, desist about using any particular multi-user interaction, thus directly presenting the PC graphical system (Beauchamp, 2004); and when addressing multi-tasking, they are single tasked or simply present methods to change the full-screen single active application (Ackad et al., 2010).

However, having multi-tasking capabilities in shareable interfaces seems to be in strict consonance with their goal of promoting and enabling collaborative work as, for instance, the recommendations by Scott et al. (2003) for collaborative tabletops are related: Multi-tasking provides a way to have simultaneous activities, allowing the transition between them (*support fluid transitions between activities*) and between personal and collective ones (*support transitions between personal and group work*). Also, several tasks can be done concurrently, by several users (*support simultaneous user actions*).

Therefore, it would seem clear that real world shareable interfaces should, at least, support some of the characteristics that have turned the personal computer into such a valuable tool, such as general purpose computing (and third party application support) and multi-tasking, which, sadly, are not yet typically found on most current research prototypes.

We argue that the lack of those features may not be an accident, neither an unconscious

5.3 Multi-Tasking Shareable Interfaces: Current Situation and Related Research

omission: the combination of multi-tasking -a feature so closely associated with single-user devices -with multi-user interaction, is not trivial; even less when combined with rich interfaces such as the ones provided by tabletops. And yet, we want to stress our vision that real world collaborative systems should allow third party applications (programs) to run and be interacted simultaneously. More precisely, every program should support multi-user input, and a single user should be able to interact with several applications at the same time.

An inspiration could be a table for group work. It is a shared space where users use tools; some share them while others do not. There can be several activities done at the same time, with tools used across them. A post-it note can be used on a shared map; but also on a private book. A map can be used for multiple purposes at the same time. In spite of all these potential concurrent activities, the designer or maker of the table should probably not explicitly provide support for any of them, not worrying, for example, whether a tea cup is compatible with a sheet of paper.

Imagine a Tabletop system where tangible objects and touches can be both used as input. On this Tabletop there is a music generating application like the Reactable that works by placing objects on the surface, manipulating virtual controls and drawing waveforms with fingers on the surface. On this same Tabletop we also have a typical application for photo sorting that can detect camera devices on the surface and displays digital pictures on the table. Both applications potentially use the whole tabletop surface. Imagine then two people wanting to sort pictures and play music on this same Tabletop. Our vision is that one user could be grouping pictures using a lasso gesture while another could be changing the loop of one cube and muting the output of an oscillator by crossing out the waveform with a finger, without things interfering with each other. On this concurrent multi-user multitasking interface, everything can occur at the same time in the same shared space.

This is not a novel or revolutionary idea and some works have, in fact, previously attempted at the creation of multi-user multi-task systems.

Dynamo (Izadi et al., 2003), proposes a shared surface for sharing digital data between several (remote) users, focusing on ownership and permissions over programs and documents in a shared multiuser WIMP system. Users may use pairs of mouse-keyboard to interact with a system that presents local and shared interfaces. In shared interfaces, it focuses the attention on methods for preserving and sharing control over applications and files. It does not, however, deal with co-located access to the interface, nor with third party applications in the shared space.

LACOME (Mackenzie et al., 2012) also depicts a common shared surface in which remote single-user PC systems are presented as manipulable windows. Third party applications are allowed, but those run in the logic of the former single-user systems. A similar concept is developed in TablePortal (AlAgha et al., 2010), where remote tabletop applications and activity is presented inside manipulable windows. In this case remote applications are multi-touch enabled, although its aim is to be used by a single user, the teacher of a classroom.

Ballendat et al. (2010) presents us with a series of devices, one of them a vertical shareable interface, which uses information such as the relative positions and orientations of the users, the devices, and other objects, and specifically their pairwise distances (proxemics), for affecting the interaction. As this information is shared between all the devices (as an Ubicomp ecology), and each device can run a different program, we could consider this example as a shared interface (based on the relative positions and orientations) with multi-tasking. The proposal does not describe, however, any strategy for coordinating the different programs, but rather assumes that they are created together as parts of the same system.

WebSurface (Tuddenham et al., 2009) presents a tabletop system with virtual windows that can be freely manipulated. These windows are web browsers presenting conventional web pages that can be interacted by the users. It could be argued that web pages are a form of third-party applications, although enclosed in a single-user paradigm. This is also the case of Xplane (Gaggi and Regazzo, 2013), a software layer presenting several tiled windows on the surface with a distinct focus to enable fast development of tabletop applications, although it does not provide window transformation abilities.

Multi Pointer X (MPX) (Hutterer and Thomas, 2007) tries to transform PCs into shared systems by allowing them to use several pairs of keyboard and mouse. As PCs are already multi-task and third-party application enabled, the result would be a shared multi-user multi-tasking system. Using a PC setting and applications, however, does not help to easily allow multi-user interaction inside the applications, neither collaboration dynamics related to the physical layout of the interfaces.

Julià and Gallardo's TDesktop (Julià and Gallardo, 2007) was a first unpublished attempt from this author of this thesis to create a tabletop operating system. It provided facilities for third-party tabletop applications to be developed, as well as an environment to run and manage multiple applications at the same time. Applications were multi-user by default, and they could ask the system for full-screen execution, when not designed as floating widgets. However, it did not enforce that input events were distributed to

one application at most, leaving the possibility to multiple interpretations.

In the next section we will study and try to overcome some of the technical and conceptual difficulties for designing a proper multi-tasking system on a shareable interface.

5.4 Approaches to Multi-Tasking

From an implementation point of view, interaction in multi-tasking can be narrowed down to two different problems: (i) allowing two or more processes to share the input and (ii) allowing two or more processes to share the output. In the PC, input would consist of mouse and keyboard events, whereas the output would take place in the monitor display (and in the speakers). In a tabletop system, the output would also be the visual and audible display, whereas the input would be provoked by the objects and the finger touches on its surface.

Although sharing input and sharing output may be superficially seen as two aspects of the same problem, they are fundamentally different. Sharing output is a relatively simple issue because it can be reduced to a mixing mechanism: many programs may require to output some data to a specific destination (the screen), and the task of such a system would simply consist on deciding how to (or rather whether to) mix these data. As the source and destination of the output events is known, the system can use simple rules to decide, for instance, if an app can draw into the display, occluding other programs, or if the sound that it is generating will be mixed with the sounds coming from other programs, and with which volume.

On the other hand, sharing input is a much more complicated de-mixing problem: data from one source (such as the data coming from the touch sensor on a multi-touch display) can potentially relate to several recipients, the programs. The task of the system on this case is more complex: the system must know the destination of every data element, that can be shared or not. On a PC, a *media play* keystroke, for instance, has to be distributed to the correct program that is waiting for these types of events, and not always necessarily to the “active” program, the one that is considered to be actually used by the user, with a privileged situation that makes it the default receiver of all input data.

5.4.1 Input sharing

As we envisage a system that allows for applications designed by third party developers, we must assume that applications developed by different people can be run and used at

the same time. This poses a problem to the traditional methods of disambiguation used inside the applications (see Section 5.5) as there may be ambiguity between gestures from different applications, which cannot be known at coding time. Even if each application uses an *orthogonal gestures* approach, the combination of the two gesture spaces from two applications can result into ambiguity.

As many programs can be potential receivers of this data, the system needs a set of rules and mechanisms to fully determine the correct recipient of every piece of input data. These rules will determine the way multi-tasking is presented to the user.

Several rules exist, for dealing with this uncertainty. Many of them will take into account the context. In interfaces where input and output happen at the same place (i.e. with Full or Nearby embodiment, according to Fishkin taxonomy (Fishkin, 2004)), such as in a touchscreen, input events can be tied to the output elements nearby. A touch can be tied to the visual element just underneath it, created by a particular program that will become its correct recipient.

Interfaces in which input and output are decoupled (Sharlin et al., 2004) may impose more difficulties. When input information is completely untied to the output elements of the processes, strategies other than using a simple distance criterion have to be used. In the case of a mouse device, for instance, the PC strategy is to create a virtual pointer that is controlled by it: as this pointer is coupled to the display, it can be treated as in the previous case (coupled). The mouse mediates between the user and the cursor; it is not a generic input device which is part of the interface, but a specific physical representation of the cursor.

The PC keyboard is another decoupled interface, and keyboard events can have several destinations, these being different programs or even different widgets inside a program. Some windowing systems simply send the keyboard events to the program under the pointer, while others create a default destination for the keystrokes (Scheifler and Gettys, 1990). . This destination is controlled by the input keyboard focus, so that only one widget (from one application- the active one) is the current receiver of all keyboard activity, and this destination can be changed using the pointer (interacting with another window/widget) or special key combinations (such as Alt-Tab in the PC) (see Section 2.2.4). The assumption of a single input keyboard focus by the PC interaction makes it difficult to adapt it into a multi-user setting, as the interaction would require multiple foci. Some approaches have been taken in this direction, such as the Multi Pointer X (MPX) (Hutterer and Thomas, 2007) extension, which allows having virtual input pairs of visual pointers and keyboards that can operate at the same time both with adapted

and with legacy X11 applications. It struggles with applications that assume that there is only one pointer and focus, enforcing single-user interaction with those applications as a partial solution. By pairing cursors and keyboards in pairs, MPX allows several foci (one per pair) to simultaneously exist (Hutterer and Thomas, 2008).

The approach to follow on shareable interfaces will depend on the type of interface and its purpose. Coupled input/output interfaces, such as tabletops or vertical displays have the possibility of tying input events to output entities. Gestural body interfaces may have to use other approaches, such as using a mediating virtual representation of the body (equivalent to the cursor) as the seminal work of Myron Krueger in Videoplace or Videodesk (Krueger et al., 1985) already suggested, or some other kind of focus mechanism.

Input sharing in tabletops is still a young question, as it seems that the problem of multi-tasking has still not arisen. Window-based application management is starting to be present on tables (Tuddenham et al., 2009; AlAgha et al., 2010; Mackenzie et al., 2012; Gaggi and Regazzo, 2013), but the preferred option continues to be full screen locking (Ackad et al., 2010).

In the next sections we will present the different approaches to an input sharing strategy between simultaneous programs: area-based, arbitrary shape area-based, and content/semantic-based.

5.4.2 Area-based interaction

In coupled interfaces it is common to find window-based multi-tasking, so that different programs obtain independent rectangular areas. They can draw and get all the interaction performed inside. Those areas can usually be transformed and manipulated by the user, making it possible for multiple processes to be present in the display at the same time, thus promoting multi-tasking. In these cases, all the programs inputs and outputs are confined inside their respective (or multiple) windows, and a simple coordinate test helps input events to be assigned to the correct program.

Rectangular windows particularly fit the PC setting. They have the same shape as the screen and, as they cannot be rotated, they can occupy the full screen if necessary, occluding other windows (rotation of windows is not desirable, as the display is vertical and has a well-defined orientation, similarly to what would happen to a painting in a wall).

Using windows on other non-PC situations can have some caveats. In non-rectangular interfaces, such as in round tabletops like the Reactable (Jordà, 2008), the rectangular

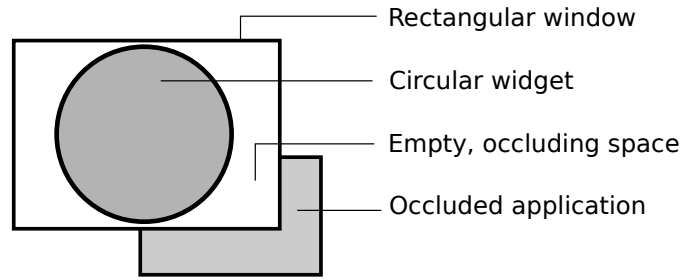


Figure 5.1: A window with empty space occluding the interaction for another

shape seems to perform poorly. The Reactable’s circular surface was designed to avoid dominant positions (Vernier et al., 2002; Jordà et al., 2005). While, perhaps for this same reason, the original Reactable avoided the use of windows or rectangular areas, its more recent commercial incarnations make use of them, and allows users to reorient them ¹, suggesting that when no predefined orientation exists, the potential rotation of windows seems necessary. Even within rectangular tabletops, at least two (or even four) predominant points of view could exist, making the rotation of windows a desired feature.

On top of these orientation issues, forcing a fixed shape for all applications may not always be a convenient solution: some programs may need less restricted areas, leaving most of its window space empty (for instance a circular program, such as a clock, would have considerable empty space at the edges of the window). This empty space would prevent input events to reach other occluded applications, making them unreachable (see Figure 5.1).

5.4.3 Arbitrary Shape Area-Based Interaction

An alternative to window-based interaction is area-based interaction. In this case, instead of windows, the system will have to maintain a list of active arbitrary-shape areas of the processes. The input events distribution mechanism should be equivalent as when using windows: a collision test will find the correct program that holds the target area for one particular event. By using arbitrary shapes instead of rectangular windows, processes no longer have the problem of empty occlusion, as all the unused application space does not have to be covered by an area. Using arbitrary-shape areas is already a popular approach when distributing events through different objects inside an application. Inside a program window, the different presented elements define areas where the

¹<https://www.youtube.com/watch?v=kYyg-wVYvbo>

forwarded input event can be assigned to. Buttons, sliders and many kinds of controls are examples of this strategy.

However, this approach is not perfect. Apart from the case of decoupled interfaces, where area-based interaction is not possible, this strategy may not be desirable in other additional situations, at least as the only discriminating mechanism.

Recent history of interaction in touch-enabled devices has shown that there is room for improvement beyond the simple gesture primitives that were associated with pointing devices, and a variety of touch-based gestures have been developed and even patented since the first portable multi-touch devices appeared (e.g. pinch zoom, swipe, swipe from outside of the screen, etc)(Hotelling et al., 2004; Elias et al., 2007).

The fact that portable devices tend to have full-screen applications, which can therefore trivially manage all the multi-touch input, has boosted the development of complementary and often idiosyncratic gestures, able to handle more complex and richer interaction. If areas were used to know the destination of every input event, the gestures of every application should start, continue and end inside of the process' areas, rendering many gestures that used to temporarily transit outside the target area, impossible to recognize. Even a strategy where only the starting event is used to check the colliding area may have problems with gestures starting outside of it. Let's imagine and study some examples of gestures that would be problematic when using areas. An application is responsible for displaying notes through the surface of a tabletop. Those notes can be translated and transformed by standard direct manipulation gestures such as pinch zoom or dragging. Imagine that the programmer wants to implement a gesture to save this note: circling the note.

Note that for circling a widget with one finger, we do not need to enter in contact with the widget itself (see Figure 5.2). If the area of the widget is defined by the surface of the note, the needed input events will never reach its right destination. Having a larger gesture area covering the places where gestures are likely to occur, may help to receive such events, but at the cost of occluding the interaction with other event recipients, such as other potential applications underneath this note's area.

In this other example, let's imagine a gesture (e.g. a cross) that instantiates a new widget (e.g. a new note in our note-taking application), anywhere on the interactive surface. As there is no predefined existing area listening for events, the note-taking application cannot know when and where to invoke a new note, and, if the whole-surface area was used for catching all potential crosses, other applications would be occluded and being unable to receive any input event. Although this particular example could be solved

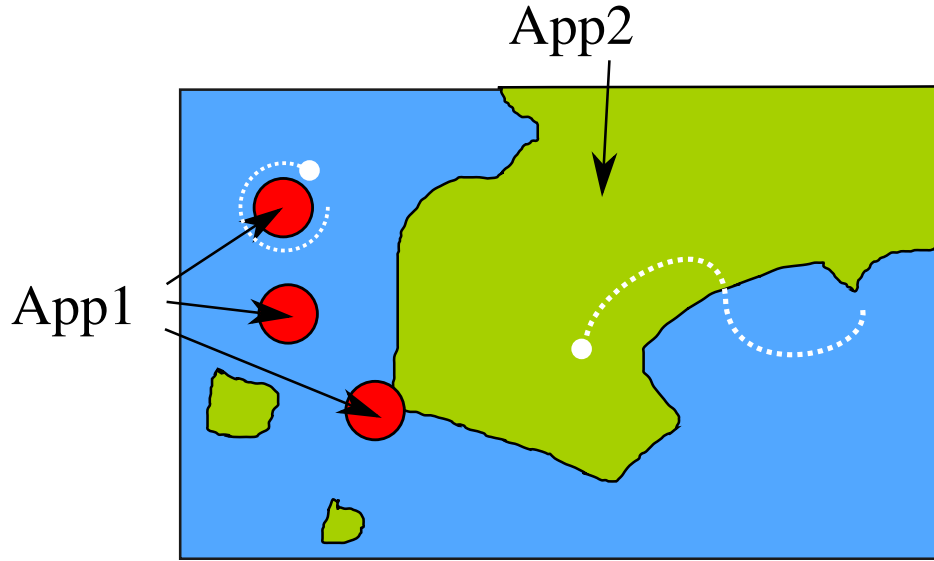


Figure 5.2: A note-taking application that allows the user to create new notes, by drawing circles with a finger over a map browsing application that can be dragged with the finger. Notice that in the case of using area-based interaction, the note taking program will not receive circling events.

by showing a button widget to create new notes, it would have to always be visible, cluttering the space. Global system gestures could be another example of gestures made outside areas; a gesture defined by the system to show a global menu, such as a wave gesture, can be performed anywhere on the surface, regardless of whatever is underneath. Julià and Gallardo’s TDesktop (Julià and Gallardo, 2007) tabletop operating system solved this problem by allowing the several applications that could run simultaneously to receive the raw stream of input events as an addition from its standard area-based input event filtering, thus receiving also input data that originated elsewhere of their areas. This solution, although effective, rises the problem of how to distribute events through applications, so to avoid the problem of having several subscriber programs receiving the same events, and each of them simultaneously assuming being the intended addressee of the interaction.

Finally, the area-based strategy to multi-task interaction is not possible with decoupled interfaces such as full-body sensors and camera-based interfaces (e.g. Kinect), motion sensors (e.g. Wii remote), voice and sound interfaces (e.g. Speech recognition); these could still benefit from multi-tasking abilities as they are already used in multi-user contexts. If multi-tasking with decoupled interfaces may still seem like a fringe problem, an example can quickly reveal its need. When multiple home appliances in the same

room, such as a hi-fi sound system and an air conditioner, can accept body gestures as commands, they are in fact sharing the same input interface (the body). Some mechanism has to ensure that the same body movement cannot be interpreted as commands for both appliances simultaneously.

In short, shareable interfaces (as we have seen in the example of TDesktop) trying to process area-less gestures, but also decoupled interfaces, would benefit from a mechanism different than using areas or windows, for distributing input data to its correct destination, and thus preventing various programs to process the same events.

5.4.4 Content/Semantics -based input sharing

For decoupled systems that cannot use window(or area)-based input sharing, as well as for coupled interfaces that for some reason would opt for not using it, an alternative can be using content-based input sharing.

In a content-based input sharing mechanism, the algorithm, instead of distributing the input events to their destinations based on the position of the event, would try to know which events are expected by every application, and would then distribute these events by deducing their right destinations. This approach would not necessarily treat input data as separated events, but rather as streams of events that may convey meaning within them. The destination of an input event, for instance, may not only depend on its own information, but also on the gesture it is part of, on the types and characteristics of the possible recipients, on the context, etc. Generally speaking, when a series of input events that have a global meaning/semantics as a gesture is defined, the system's function is to successfully recognize the performed gesture and subsequently distribute it into the processes, given their current expectations and their contexts. A very simple example implementing this idea could be a system which has the code to recognize a set of gestures from the input, and when it fully recognizes a gesture, this is distributed to the application that has requested it. In the possible case that applications A and B request respectively the stick and pinch gestures, when the system recognizes a stick gesture it handles it to A. Instead, when a *pinch* gesture is recognized this one is sent to B.

Some existing systems already use this last kind of approach. There are special cases in mobile device systems where two applications can share the same interaction space (if we consider the operating system to be an application). For instance, in the Apple's iPad² there are some system gestures that can be recognized even when running applications.

²<https://www.apple.com/ipad/>

In these cases the disambiguation technique used could be described as *disambiguation by cascade*: first the system tries to recognize its gestures, and then the application recognizes its own. This approach, however, is only valid between apps when we can set priorities between applications (focus), something that is justifiable for global system gestures, but not between different applications as the notion of multi-user multi-task denies interaction focus.

An issue arises when implementing a system that uses content-based input sharing: does the system incorporate all the code needed to recognize all the defined gestures? Should the full set of gestures be defined within the system or should they be defined within the addressees programs themselves? Depending on how we choose to distribute the role of defining and recognizing these gestures, three different strategies can be employed:

- *A centralized gesture recognition engine, with a fixed set of gestures.*
- *A centralized gesture recognition engine, with an application-defined set of gestures.*
- *A decentralized application-centered gesture recognition, with a coordination protocol.*

A centralized gesture recognition engine, with a fixed set of gestures

As in our *stick* and *pinch gestures* example, the system could define a fixed set of gestures the applications could register to. Based on the preferences of the applications at the time a gesture is recognized, the system just notifies the correct program when an individual gesture is recognized. Unfortunately, this strategy has a clear drawback, since it prevents programs to define their own gestures, the ones that the application programmer(s) felt were best suited. Rich interaction, understood as the possibility for individual applications to define their own optimal gestures independently of the existing system gestures, is thus dangerously limited.

A centralized gesture recognition engine, with an application-defined set of gestures

In this type of systems, common recognizing mechanism needs to be implemented, for which the application programmers will define their own respective recognizable gestures. Many recent advances have been attained in the direction of language-based gesture definitions, especially in the context of multi-touch applications, which in our case could allow arbitrary gesture definitions to be added to the system at runtime:

Proton (Kin et al., 2012), Midas (Scholliers et al., 2011), GeForMT (Kammer et al., 2010b) and GISpL (Echtler and Butz, 2012) all allow the programmer to describe gestures in specially crafted languages that simplify the programming of gesture recognizers, and therefore the code dedicated to detect gestures from the input event streams. From those, Midas, GeForMT and GISpL are interpreted (GISpL only partially) and could theoretically be used as the basis for more general systems, on which the applications carry their own gesture definitions and transfer such specifications to the system, which would use them to recognize the gestures.

The choice of the gesture definition language is also a non-trivial issue. Such a language should ideally be as complete as possible in order not to become an obstacle for the programmers, thus making some gestures impossible to define. For instance, for allowing gestures to be related to the application context data, such as virtual objects inside the application, the definition language should provide ways to access it. Proton, GeForMT and GISpL explicitly integrate areas (as parameters to be accessed in the language or as a previous filtering) as part of their languages, easing area-based gestures to be programmed, but making area-less gestures difficult to describe, as this is a fundamental part of these languages. Midas allows instead for a sort of generic user-defined code and object access from inside the gesture definition, thus enabling not only areas, but also other types of constraints to be used, showing its potential to be useful in many gesture recognition styles. However, it is unclear how such relationship would work when applied on a server-client schema, which would need to interpret the definitions within the system while the needed code and data resides on the program. Apart from these language issues, a gesture recognizer system should also meet some additional requirements. None of the aforementioned languages allow multiple instances of gestures being performed at the same time, treating instead all the input events as part of the same gesture, thus making them unsuitable for multi-user contexts.

Although a variation of the previous projects would probably fit the requirements for building this type of system, forcing all the programs to describe their gestures in a common language would also have the side effect of preventing other kinds of gesture-recognition approaches from being used. For instance, machine learning based approaches (such as (Wobbrock et al., 2007) or (Caramiaux and Tanaka, 2013)) would not be possible, since within this strategy, gestures are not formally described, but learned instead from examples.

A decentralized application-centered gesture recognition, with a coordination protocol

With this third strategy, the system does not participate directly on the recognition of the gestures, but helps instead in coordinating the set of programs interested in these gestures. The recognition process takes therefore place inside the applications, allowing nearly total freedom to the programmer, while a common protocol between the system and the programs is used to guarantee that no single event is mistakenly delivered to two different processes.

By running the gesture recognition inside the application, it can take into account its context (e.g. position of the application elements, and other internal logic) without having to rely on a good gesture language definition, as in the previous case. This approach also allows programmers to code the recognizers using their favorite techniques or frameworks, instead of having to rely on the system's choice of language or libraries. Furthermore, as the system is in charge of preventing double interpretations of gestures across different applications, the different recognizing mechanisms will not need to provide multi-tasking facilities. The aforementioned gesture description languages could be easily adapted to support the coordination protocol with the system, and they could be deployed inside the application. Other programs could for example use a machine learning approach, provided that they respect the protocol, and thus train their gesture recognizers with examples.

The framework we are presenting, *GestureAgents* (Julià et al., 2013), tries to create this common protocol and infrastructure. In *GestureAgents*, instead of relying on the use of a particular declarative language, the recognizing mechanism is conditioned by a series of coordination messages that the system and the processes need to exchange.

Our approach simply requires developers to code their recognizers in a way that allows the system to have information about the context and the input events involved in the gestures being recognized. The system decides, during the interaction, whether a recognizer can or cannot identify each input event as part of a gesture. It basically defines a set of sensible rules that all recognizers must meet, but it does not force a particular coding style or technique.

In particular, *GestureAgents* was created with these features in mind:

- Device-agnostic: e.g. it does not matter if we are using multi-touch interfaces or depth-perceptive cameras.
- Not enforcing a specific programming technique or library for gesture recognizing.
- Allowing concurrent gestures.

- Allowing both discrete and continuous gestures.
- Allowing multiple applications to share sensors and interfaces without the need of sub-surfaces or windows.
- Freeing the developer from knowing the details of any other gesture that can occur at the same time and device.
- Trying to take into account real time interaction needs.

5.5 Approaches for Multi-Gesture Applications

The presented strategy for enabling cross-application gesture disambiguation can also be useful inside a single program. Every application can have many possible gestures to recognize and many possible targets for these gestures, and their recognizers could be interacting with each other as if they were from different applications in order to coordinate their disambiguation.

In fact, *GestureAgents* started with this very purpose, enabling applications to use multiple simultaneous gestures in a safe way, relating to the problems identified in *ofxTabletGestures*, described in Section Simplifying the API (2012), where several gesture recognizers would consume the same input event, resulting in multiple simultaneous interpretations of the user behavior.

We approach the existing approaches to provide disambiguation inside a single application and their relevant properties in the next sections.

5.5.1 Gesture recognition and disambiguation in a single application

An interface or application that allows complex gesture interaction must decide the meaning of every single input event. When it finds that one same sequence of input events has several gesture candidates, it has to disambiguate them.

This process can be eluded by avoiding ambiguous situations altogether, for instance by designing the application to have orthogonal gestures and inputs, which would imply that different gestures do not share interaction spaces or sources. *TurTan* (Gallardo et al., 2008) is an example of a tangible tabletop that uses pucks and touches for its interaction, separating object-related gestures and touch-related gestures into two totally unrelated sets of gestures.

Another factor used to limit the possible gesture ambiguities is context. For instance, a given application can accept one type of gesture at one state and another in another

state. The same can happen amongst areas of interaction: if a button is only *tappable* and a canvas is only *drawable*, the application can use spatial information to rule out gestures. In this case we can think of multiplexing the interaction by time or space (Fitzmaurice et al., 1995).

Finally, more complex multi-gesture (multi-modal) interactions can be considered that do not use the limiting strategies defined above. There are many ways to handle the disambiguation process. Typical approaches are machine learning techniques (Schlömer et al., 2008; Wobbrock et al., 2007) or custom analytic disambiguation code, coded ad-hoc or generated by a static analysis of the possible gestures (Kin et al., 2012). Note here that, to be able to use these techniques, the programmers must know at coding time all of the possible gestures that can happen at run-time.

Event-driven gesture recognition frameworks

Many frameworks created to support touch-enabled applications in tabletops use the most low level approach to deal with gesture recognition, event-driven programming. With it, the code created to recognize a gesture will process the different input events directly as they arrive.

Dealing with events is considered difficult (Hoste and Signer, 2014), as the program flow is defined by the events instead of the program and it poses a challenge to deduce the current context in which events are generated, as the meaning of two events of the same type can vary greatly with the internal state of the gesture or the application. For instance, a *finger move* event can mean different things for a tap recognizer, depending on whether it is from the finger it is tracking or not.

This difficulty was greatly experienced by the TSI students when programming their gestures recognizers, even when assisted to first create a state machine on paper to understand all the possible interactions with events, as explained in Section Multi-user gestures (2011).

Microsoft Surface SDK (now PixelSense SDK)³ is the canonical way to create apps with gesture capabilities for the PixelSense tabletop (see Section Existing tabletop applications). It is based on Windows Presentation Foundation (WPF) and many other Microsoft Windows technologies. It gives access to raw events and provides a family of WPF multi-touch controls.

Multitouch for Java (Mt4j) (Laufs et al., 2010) provides a cross-platform framework to

³<http://msdn.microsoft.com/en-US/windows/desktop/hh241326.aspx>

access to the input events from several sources based on Java. A set of custom widgets is also provided.

Graffiti (De Nardi, 2008) is a library written in C# that focuses on gestures centered to tangible objects on the interactive surface.

PyMT (Hansen et al., 2009) is a framework for Python applications that provides both raw data access and specific functions for recognition. It provides with a library of custom widgets.

Kivy (Virbel et al., 2011) provides a multiplatform framework designed to create multi-touch applications for a variety of computing platforms. Coded in Python, it provides a strong and guided separation between the presentation and underlying program logic. Many custom widgets and gestures are provided, supporting multi-trace gesture recognition only experimentally.

GestureAgents follows this strategy in its part of gesture recognition. Although other approaches, such as the ones to be presented next, tend to be easier to use, the many limitations they have and that will be exposed, made us believe that this was the best option.

Machine Learning Techniques

Machine learning techniques are very present in gesture recognition across various interfaces. In general, many problems related to extracting the underlying model from the sensor data, can be a good candidates to being solved using machine learning. In particular, extracting useful data from images to reconstruct the body position and parts, is a typical field of application, from face tracking to human pose reconstruction (Popescu-Belis et al., 2008; Shotton et al., 2013).

Even with its usefulness, creating machine learning systems to solve these kind of problems is a complex task and requires an expert knowledge on the used methods to not fall in its various typical pitfalls (Domingos, 2012), being problematic to non-expert users.

Machine learning can also be used to classify gestures from provided examples. A very generic system is provided by The Wekinator (Fiebrink, 2011), allowing the user to configure many types of machine learning algorithms to recognize any type of data, often coming from camera or sensors. The resulted classification or output parameters are then sent to the application.

Recognizing gestures from sensor data is indeed a popular field for machine learning techniques, because of the difficulty of reconstruction of the original 3D movement. As

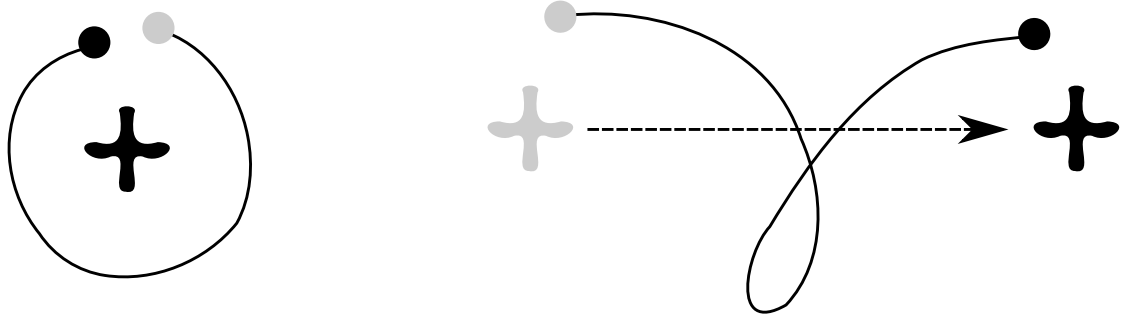


Figure 5.3: Two circling-a-target gestures, one with a still target and another with a moving target. Notice the difference between the resulting trajectories.

an example, Schlömer et al. (2008) use this approach to classify gestures using a hand-held Nintendo Wii controller.

More specialized systems being used in tabletops include the classic 1\$ recognizer (Wobbrock et al., 2007), that can be trained with a single example of every gesture, and performs an angle and scale-invariant match of the whole finished gesture.

A more complex system that also uses a one-example philosophy, but provides real-time recognition, is the Gesture Variation Follower (Caramiaux and Tanaka, 2013). It not only classifies the gesture, but also identifies the position inside it, allowing its progression to be tracked.

A very common problem of these approaches is the lack of context: the algorithm only takes into account the input data and not the relevant state information from the application. This means that a gesture which trajectory depends on external components of the interaction, as an application defined target, will not be necessarily well recognized afterwards. Take for instance a gesture circling a target. The trajectories of two of these gestures, one for a still target and another for a moving one, will be completely different, and the algorithm may have some trouble identifying them (see Picture 5.3). Even if context data is feed to the algorithm, it is impossible to know if irrelevant data will be learned to be used to recognize unrelated gestures, as the results of those algorithms are often treated as a black box (Hoste and Signer, 2014).

Another very important problem that usually face those approaches is the one of lacking multi-gesture support; the applications have to make a decision a priori of whether all of the input data belongs to a single gesture or not. In 1\$ recognizer users, for instance, it is very typical to assume that every trace belongs to solely a gesture, thus making multi-trace gestures impossible.

Domain specific languages

Another approach for gesture recognizing, trying to ease the coding of recognizers, is the definition of a domain specific language specifically designed to describe gestures. Those definitions can then be translated to recognizers compilable code, or be interpreted later by a central engine.

GeForMT (Kammer et al., 2010b) defines a grammar with atomic operations (POINT, SEMICIRCLE, ...), that will be interpreted by a centralized engine, to check if the input events meet the definitions.

GDL (Khandkar and Maurer, 2010) presents another grammar that allows the programmer to list a series of preconditions to verify on the input events, in order to recognize a gesture. Sequential traces can be used in a single gesture by saving partial results with an alias for later use.

Midas (Scholliers et al., 2011) provides an extensive language used to define rules that represent gestures. Those rules have prerequisites that if evaluated to true, execute the consequences. Midas provides a more low-level approach to gesture description by not focusing the language to the physical description, but to the programmatic requirements and actions to be performed in a functional style. It also allows a generic mechanism to use the application context inside the gesture definition, not explicitly limiting it to areas or targets.

Proton (Kin et al., 2012) takes a more formal approach by describing gestures as regular expressions to be matched on input data. By doing this, it allows a static analysis to discard conflicting gestures. It also provides a graphical editor for the language intended to leverage the creation of new gestures by novices.

GISpL (Echtler and Butz, 2012) (formerly GDL (Echtler et al., 2010)) focuses its work to provide a device-generic way of describing gestures, by defining gestures as a series of recognized features. By separating the description of the actual movement, in features, from the meaning-bearing gesture, it is easier to create gestures that are independent of the actual input device, also allowing a sort of code reuse.

As discussed earlier in Section 5.4.4, the lack of generic operations (due to the restricted language) and access to the application context are important limitations that we have to take into account, as they potentially limit the expressivity of the created gestures.

But, undoubtedly, the biggest limitation of the aforementioned languages is that none of them allow multiple instances of gestures being performed at the same time, treating instead all the input events as part of the same gesture, thus making them unsuitable

for multi-user contexts. Only limited support for concurrent gestures is provided using area filtering.

5.5.2 Gesture Composition

Gesture Composition is the ability to describe a gesture in terms of a combination of previously defined simpler ones. For instance, a *double tap* can be described in terms of *tap* (a *double tap* is the sequence of two *taps* close in time and space) instead of primitive finger ones.

Gesture composition allows programmers to better reuse and test the code, lowering the gesture recognizers verbosity and simplifying the code overall.

Many of the presented domain specific languages (such as Proton) define gestures as a combination of more or less primitive touch event types. Although this could be considered gesture composition, it does not allow using custom gesture types as building blocks.

Midas framework allows to develop complex gestures by combining multiple basic ones. This is achieved by asserting gesture-specific facts on gesture detection. GISpL introduces *features*, conditions to be met to match a *gesture*, which can be defined as a combination of other, possibly custom, *features*.

In GestureAgents the composition mechanism is provided by considering recognition as a multi-layer process: every gesture recognizer is also an event generator (*agent*) that can be used by other recognizers: there is no explicit concept of “sub-gesture”, in the sense that all recognizers are programmed the same way. The disambiguation process ensures that no overlapping gestures are recognized (the implementation is better described in Section 5.7.1).

5.5.3 Single-Gesture Certainty and Real Time Disambiguation Strategies

When faced with ambiguous gestures we can find that different frameworks differ on how they handle them. Some choose which gesture will “win” based on a probabilistic approach where each possible gesture is assigned a probability in a given situation, here the system will favor the most likely one. This probability could be computed using positional information of the input related to possible targets as well as completeness of the gesture (Schwarz and Hudson, 2010; Caramiaux et al., 2010).

Other frameworks rely on a list of priorities that the developer can define, gestures of low priority are “blocked” until gestures considered more important have already failed,

this logic is present in Midas (Scholliers et al., 2011), Mt4j (Laufs et al., 2010), Grafiti (De Nardi, 2008) and in Proton (Kin et al., 2012) frameworks.

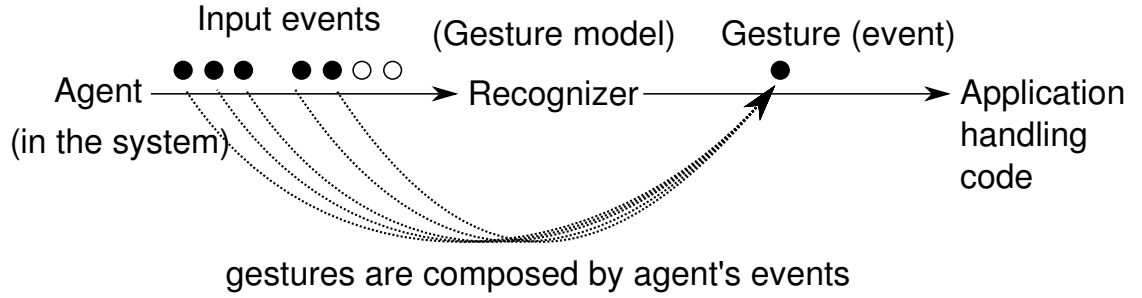
Some frameworks will attempt to take a decision as soon as possible to reduce lag, while others will prioritize certainty and will not resolve the conflict until there is merely one possible alternative left (Scholliers et al., 2011).

The choice between reducing lag (and prioritizing Real-Time) and maximizing certainty has deep implications. While the first allows instantaneous feedback to the user, regardless of the conflicting gestures, if new events favor a different gesture from the one being currently assign, the interpretation of the actions of the users can change abruptly. If the erroneously assigned gesture has made any actions on the program, a rollback mechanism can be necessary to maintain the integrity of the application. Independently of the unexpected consequences these actions may have, the overall process may confuse and deter users to confidently engaging with the system.

On the other hand, prioritizing certainty can mean delaying all consequences of a performed action while there is uncertainty. Although this is the safest strategy, actions requiring real-time feedback for the user may be impossible because of the delay of the gestures consequences.

This is intimately related to the two different types of gestures that exist when we consider the the Real-Time dimension of interaction: discrete (or symbolic) gestures and continuous gestures (also known as online and offline gestures (Kammer et al., 2010a; Scholliers et al., 2011)). Discrete gestures are gestures that do not trigger any reaction until they are finished (Kammer et al., 2010a). A typical example could be a hand writing recognition system where the user must finish the stroke before it is recognized. On the other hand, continuous gestures may trigger reactions while the gesture is still performed. For instance, a pinch gesture can be applied constantly to a map while it is performed. This distinction is important not only at its implementation level, but also in a conceptual one: discrete gestures do not acquire meaning (and do not change the system state or trigger any kind of reaction) until they are completed. For instance, in a CLI (where only discrete gestures exist) nothing happens until the *Enter* key is pressed. On the other hand, continuous gestures do already convey meaning before they are completed. This means that at some point, part of their meaning is defined (typically the Type of the Gesture) while other parameters can still change, ideally in real time. In such a case we can call these parameters Controls.

GestureAgents uses the certainty maximization approach, by not allowing gesture events to reach the application until there is only one valid explanation. It however tries to

Figure 5.4: Conceptual elements of *GestureAgents*.

reduce the disambiguation lag by requesting the recognizers to make a decision as fast as possible (see Section 5.6.3) or by creating specific policies allowing latent gestures (see Section 5.6.4). This way it tries to support both discrete and continuous gestures.

5.6 Implementation of *GestureAgents* Framework

In this section, we describe how the *GestureAgents* framework implements the proposed protocol strategy to manage input events to be consumed by recognizers implemented in several applications. We first introduce the basic elements, then the protocol between the applications and the system, the restrictions of the gesture recognizers' behavior, and give details about the functioning of the system and the particular implementation of *GestureAgents*.

5.6.1 Elements of *GestureAgents*

GestureAgents is a framework that aims to provide a generic and flexible solution for multi-user interaction in shareable interfaces, both inside a single application as in a multi-tasking system. As schematized in Figure 5.4, *GestureAgents* relies upon the concepts of “agent”, “gestures” and “gesture recognizers” and on the idea of “agent exclusivity”.

An **agent** is the source component of part of the interface input events, such as an object in contact with a tangible tabletop interface or a finger touching the surface on a touch-based interface. For example, in the case of a multi-touch interface, an agent would be created for every sequence of touches, considering that a sequence starts with the detection of a finger hitting the surface and concludes when the finger is removed from the surface. By default, agents will represent the minimal set of identifiable event types (such as the finger touch already described), while more high level agents, those

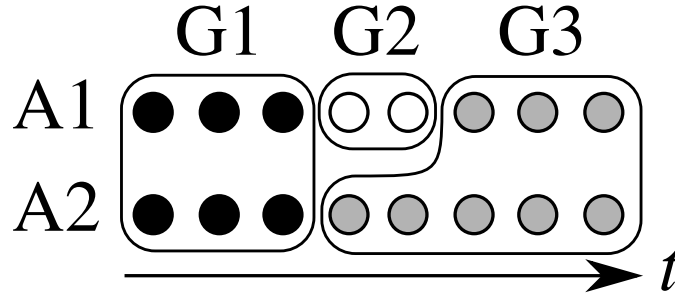


Figure 5.5: In this example the events, emitted by two agents (A1,A2) and represented by circles, are part of three different gestures (G1,G2,G3) that can occur simultaneously, as in the case of G2 and G3.

composed by other agents, such as a hand agent composed by finger ones, can be also provided, which are best suited for full body interfaces, where the interaction can have different “resolutions”.

Gestures are sequences of agents’ events, which convey meaning expressed by the user and defined by the program. A gesture can relate to a single agent or to multiple ones, both simultaneously and distributed in time. Gestures can be discrete (or symbolic), in which case they will not trigger any reaction until they are finished; or continuous, in which case already convey meaning before they are completed, and can therefore trigger reactions before finished (Kammer et al., 2010a).

A **gesture recognizer**, a piece of code that checks that the pattern that defines the gesture corresponds to the received events, is used by the program to identify a gesture coming from the agents’ events. By using agents as the basis for its gestures, recognizers do not have to receive all the events from the interface, but only the agents they are interested in. This allows the recognition of multiple gestures at the same time (see Figure 5.5), as opposed to the majority of gesture frameworks, where all the input events are part of the same gesture.

The fundamental idea in GestureAgents is based upon agent exclusivity. An agent, at one given time, can only be part of **one** gesture (see Figure 5.6). The system presents the input data, in the form of agents, to the gesture recognizers inside the applications, and, if they want to use them as a part of their associated gestures, they will have to compete between them to earn the exclusivity over the agent’s use before recognizing their gesture. By locking different agents, several recognizers can simultaneously recognize gestures, preventing double interpretation of the same input events, and allowing multi-tasking and multi-user gesture interaction.

Following we describe a concrete example. One mouse button press (one input event) can

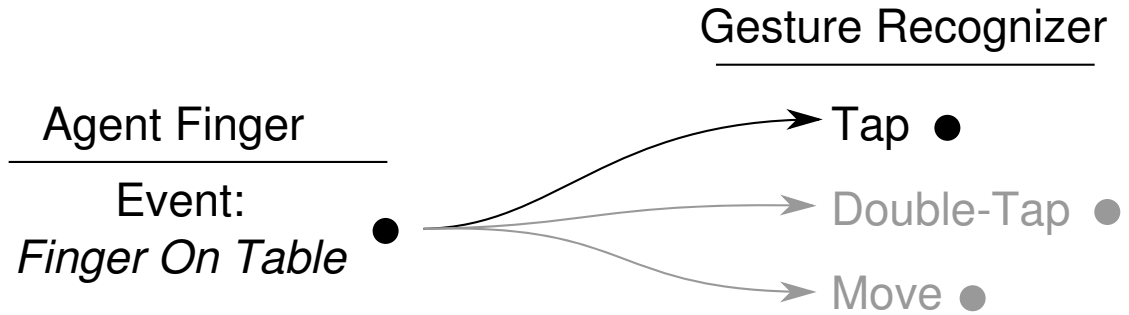


Figure 5.6: Agent exclusivity enforces that an agent at a given time can only be part of one single gesture.

only carry (or be part of) one meaning (Gesture = Agent \rightarrow meaning). It can not mean both a click and a double-click. Things have been this way most of the time in existing systems like the WIMP, where the concept of Focus forces every input event to have an obvious target that identifies its meaning/gesture. For instance, Agents that have no spatial context, like keystrokes, have a target widget defined by a focus point so a single keystroke always goes to a single target. This restriction seems also reasonable, as it is difficult to imagine attaching more than one gesture/meaning to one atomic operation in a given system. This is, however, a limitation. Systems that are not deterministic or that for artistic purposes want to mix incompatible gesture sets without defining the priorities or policies required by this restriction, must be built in a different way and cannot benefit from this framework.

As *GestureAgents* does not use a special gestural description language, problems concerning the limits of this framework in terms of completeness or design assumptions do not apply. Definition of gestures is done solely on the applications. Areas, if present, are also implemented at the application level, and tested by the applications' own gesture recognizers, using their own settings. It is thus up to the programmer to use any existing library to recognize gestures or to code a recognizer from scratch.

5.6.2 *GestureAgents* Protocol

The coordination protocol is defined by communication between recognizers (inside applications) and agents (in the system), relating to the process of soliciting agents, getting their exclusivity and releasing them.

The communication regarding the recognition of gestures, happens between recognizers (inside the applications) and the system (holding the agents), as shown in Figure 5.7. The *GestureAgents*' protocol defines various types of relationships between recognizers

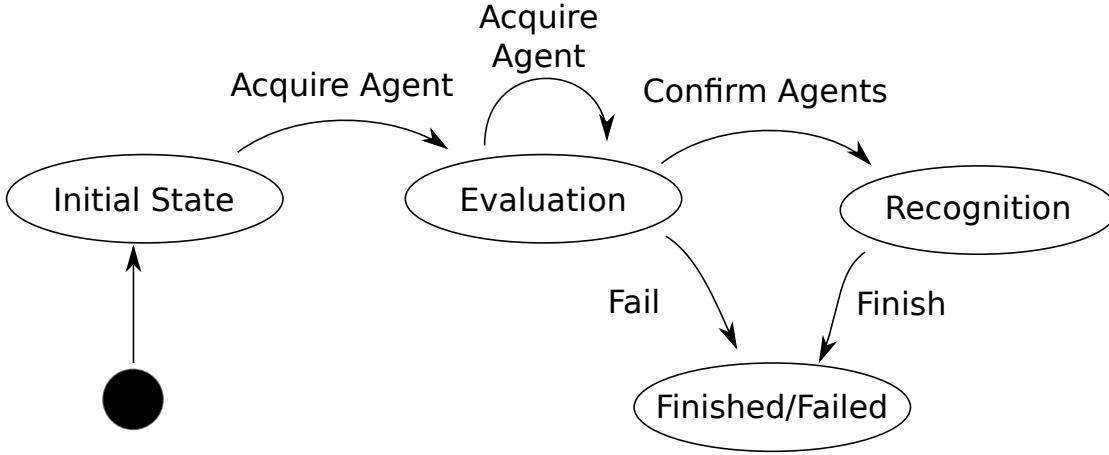


Figure 5.7: States of a recognizer

and agents, depending on their internal state. Specifically, a recognizer is considered to follow a process of four distinct steps:

- Initial state

The recognizer is waiting for an agent (of one specific type) to be announced by the system. While in this situation, the recognizer can be considered *dormant* (it is not related to any active agent or gesture).

- Evaluation state

The recognizer, which has communicated to the system an interest on one or several agents, is evaluating if their events match a possible gesture, which may or may not be recognized at the end. In this state, the confidence of the recognizer for the hypothesized gesture is not high enough for considering it to be correct or incorrect.

Depending on the type of gesture being evaluated, this state can be more or less extended in time. Discrete (or symbolic) gestures will be processed mostly in this state, because their correctness is not fully set until the end of the gesture (Kammer et al., 2010a). Continuous gestures, however, can be recognized way before the gesture has ended. In this former case, this state will last as long as the type of the gesture is not confirmed.

- Recognition state

In this phase the recognizer is confident that the tracked events of the agents match its associated gesture pattern. The transition to this state occurs after two subsequent factors: (i) the recognizer no longer considers the gesture a hypothesis (and

so it abandons its evaluation state), and (ii) the system grants the recognizer the exclusivity on the requested agents. In this state the recognizer simply processes the agents' events to extract control events from the gesture, until the recognizer considers it to have ended.

- Failed/finished state

In this state the recognizer is no longer active; this can be due to the nonrecognition of the gesture, or to the successful conclusion of the recognized gesture.

With this behavior in mind, the protocol is composed of a series of messages that can be exchanged between the recognizer and the system. From the recognizer perspective, these would be the messages sendable to the system:

- Register (or unregister) to a type of agent

If a recognizer is registered to a type of agent (for instance a “touch agent”), when a new agent of this kind appears in the system, the recognizer is notified. This message will typically happen in the recognizer's initial state.

- Register (or unregister) to an agent's event type

Given an agent, the recognizer subscribes to its events. For instance, given a touch agent, it could be possible to register to its update events (movement, or pressure). In the evaluation state, the recognizer will subscribe or unsubscribe to different type of the agent's events, depending of the pattern of the associated gesture.

- Acquire an Agent (preventing other recognizers from getting its exclusivity)

By acquiring an agent, the recognizer expresses its interest in it, communicating the system that it is currently evaluating if the agent is part of a given gesture. This message will be responded by the system with the result of the operation: *true* for success acquiring the agent; *false* for failure acquiring it. This prevents the agent from being assigned to other recognizers (from another program, for instance), until this recognizer dismisses it (due to conclusion or to nonrecognition). The recognizer will typically acquire agents in the evaluation state.

- Confirm an Agent (requesting the Agent exclusivity)

After successfully acquiring an agent and checking for its events, the recognizer may conclude that it is part of the expected gesture. It then proceeds to confirm it. This message will only be issued by the recognizer when attempting to transition from the evaluation state to the recognition state. The response from the system may not be immediate (we will address disambiguation delay later in Section 5.6.3), and until then the recognizer remains in the evaluation state. If the exclusivity

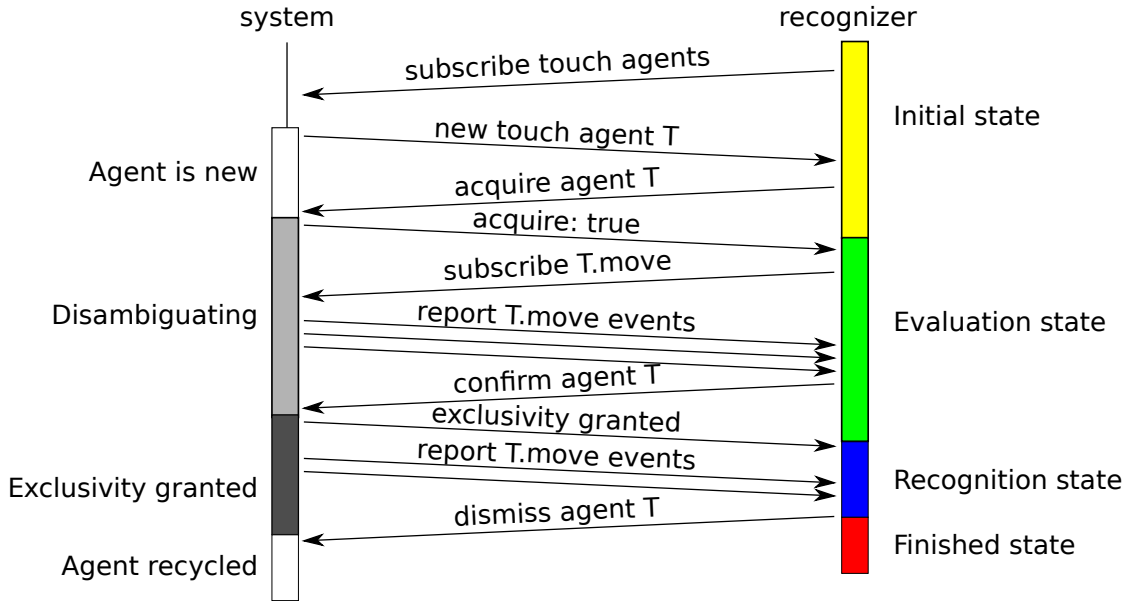


Figure 5.8: Protocol representation of the “straight line over a widget” gesture recognition example.

is finally granted by the system, the recognizer will receive a message from the system notifying so. If the system does not grant the exclusivity, it will send a message forcing the recognizer to fail.

- Dismiss an Agent

An agent can be dismissed in order to be reclaimed by the system, for being assigned to other recognizers. This may happen when a recognizer voluntarily considers that an acquired agent is not part of the expected gesture, or when confirmed agents are part of a gesture which the recognizer considers finalized. Also, when a recognizer fails, all the acquired and confirmed agents are forcefully dismissed.

The system will send signals to the recognizer, both (i) in response to its requests, (ii) in the case of acquiring an agent and, on its own prerogative, (iii) for notifying the presence of new agents, (iv) for transmitting agents’ events, (v) for granting the exclusivity over an agent, or (vi) for forcing the recognizer to fail.

To illustrate how this protocol works, we will detail a possible example of a recognizer’s life-cycle, based on a recognizer that implements the recognition of the gesture “straight line over a widget” in a tabletop system, as represented in Figure 5.8.

In this example, when the application starts, the recognizer is instantiated by the ap-

plication and it starts in its initial dormant state. It then subscribes to the touch agent type to receive new agents' announcements. Each time the system notifies the recognizer of the presence of a new touch agent, the recognizer checks that this agent is near a widget, as its gesture should be related to one of them. If the touch agent happens to be near a widget, the recognizer declares its interest in the agent by acquiring it, and entering into its evaluation state.

If this agent is not yet assigned in exclusivity to any other recognizer, the system accepts the query and communicates it to the recognizer, which subscribes to this agent's movement events, in order to track the trajectory of the touch. While the touch agent slides through the surface, the system sends the corresponding agent events related to this movement. With every update, the recognizer keeps checking if the overall movement is indeed a straight line, and if it is crossing the widget nearby.

When the touch crosses the widget, the recognizer notices that the events definitively do match its expected gesture pattern, and it confirms the already acquired touch agent, thus requesting its exclusivity. If, at this moment, no other recognizer is acquiring it, the system confirms the exclusivity to the recognizer. With this confirmation, the recognizer moves to the recognition state, and starts receiving the events from the touch agent. When the recognizer decides that the gesture is completed, it finishes by dismissing the agent in the process.

5.6.3 Restrictions on the Behaviors of Recognizers

The good functioning of the described protocol depends on the recognizers implementing the protocol correctly, but also on respecting some good practices. In particular, during all the time one recognizer stays in its evaluating state, it is preventing other (possibly correct) recognizers to get the agents exclusivity and enter their own recognition states. An ill-coded recognizer, for instance, could just acquire all the agents in the system, and never fail or confirm them. This would indefinitely prevent all other recognizers to successfully earn the agents' exclusivity and thus no recognizer would ever actually recognize their corresponding gestures.

Even when using correctly implemented recognizers, a delay between a recognizer confirming an agent and getting its exclusivity can be caused by another recognizer blocking the agents. This *disambiguation delay* is specially problematic with continuous gestures (Figure 5.9).

To minimize the disambiguation delay between the recognizer confirming the agents and getting their exclusivity (pictured in Fig. 8), recognizers must decide as soon as possible

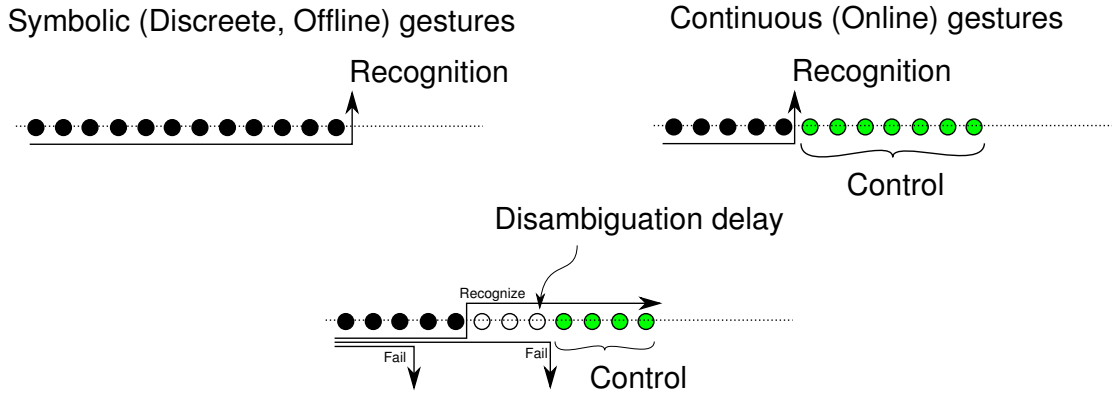


Figure 5.9: Disambiguation delay occurs between the evaluation state (black) and the recognition state (green).

whether a stream of input events can be or not be assigned to a gesture, thus minimizing their stay in the evaluation state.

Another consequence of this recognition process protocol is that confirming agents is a final decision. Once a recognizer enters the recognition state, the gesture should always be valid, and if the agent's events are no longer considered part of the gesture, the recognizer should finish and release all the agents' exclusivity. The agents can then be used again by other recognizers, in the condition of *recycled* agents, as their appearance is caused by the release from a previous recognizer, instead of being new.

When a continuous gesture is already identified and begins its control, switching to identifying this same agent as performing another gesture in the middle of its action could cause confusion (explicit exceptions can be defined though using policies). Imagine a user starting to move a virtual element and suddenly painting over it with the same gesture because in the middle of performing the gesture the system decided that *painting* was more appropriate than *moving*.

5.6.4 The GestureAgents System

The rationale behind these messages is embedded in the functioning of the system, while protecting the agent exclusivity. For each agent, the system manages a list of acquiring recognizers (those that are interested in the agent) and a slot for only one completing recognizer (that considers this agent as part of its gesture). When a recognizer acquires the agent, the system simply adds it to this list, unless this agent's exclusivity is already given.

When a recognizer confirms an agent requesting its exclusivity, the system removes the

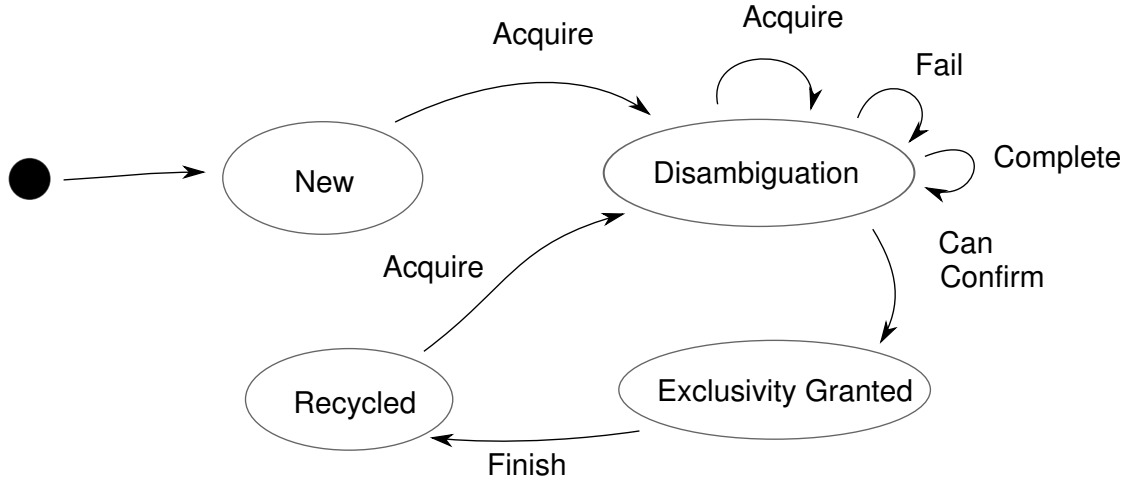


Figure 5.10: Life cycle of an agent

recognizer from the acquired list and puts it into the completing slot. If the slot is not empty, the system decides (via the consultation of several policies) whether or not the new candidate should replace the old one, and the loser (whichever it is) is forced to fail. In general, exclusivity is granted only when the list of acquiring recognizers is empty, which usually happens when alternative acquiring recognizers fail recognizing the gesture, and thus dismiss the agent, removing them from the agent’s list of acquiring recognizers.

When a recognizer dismisses an agent of which it had its exclusivity, this agent can be used again by other recognizers; the system sets a flag marking it as “recycled” and notifies other interested recognizers as if it was a brand new agent (an overall picture of the states and transitions of an agent is shown in Figure 5.10).

This mechanism actually prevents agents from being used as a part of a gesture, until no other recognizers are interested. When two competing recognizers are sure that an agent is part of their gesture, a decision has to be made. Policies, an ordered list of specific rules that apply to specific situations, will deal with cases of conflict, defining priorities and compatibilities between recognizers.

The decisions to be taken by the system can be defined by using two sets of policies, *completion_policies* and *compatibility_policies*:

- The *completion_policies* are consulted when confirming an agent. They decide whether the new recognizer candidate for exclusivity can replace the old one in the completing slot, defining a priority between two competing recognizers. For instance, a system could decide that recognizers from applications with a given

priority, will win over non-prioritized ones, or an application could enforce that a pinch zoom recognizer always wins a drag move recognizer.

- The *compatibility_policies* are used to decide whether a recognizer can be given the exclusivity over one agent, while another one is still acquiring it. Although, at a first glance, this may seem as if we were breaking the exclusivity rule, we are in fact only affecting the disambiguation mechanism, as we will still only allow one of the recognizer to use the events from this gesture. What this mechanism is in fact allowing is having recognizers in a “latent” state, which will allow other recognizers to use the agent until they can confirm it, thus finally provoking the recognized gesture to end. *Compatibility_policies* thus permit defining a priority between a confirmed recognizer and “latent” aspirants.

A common generic policy set that could be added to a system using GestureAgents, would be one prioritizing complex gestures over simple gestures. For instance, in a tabletop, prioritizing gestures involving multiple fingers over gestures involving one single finger. If we measure this complexity by the number of acquired agents, it would be simple to define a *completion_policy*, guaranteeing that complex gesture recognizers will be granted the agents’ exclusivity whenever they successfully recognize a gesture, in spite of the less complex gesture recognizers acquiring them:

```
@Agent.completion_policy.rule(0)
def complex_beat_simple(r1, r2):
    if len(r1._agentsAcquired) < len(r2._agentsAcquired):
        return True
```

By defining a similar *compatibility_policy*, we would allow simpler gestures to be recognized until a more complex gesture gets the exclusivity. This pair of policies would also solve the previously mentioned pinch-zoom versus drag-move gesture problem.

```
@Agent.compatibility_policy.rule(0)
def simple_can_recognize_until_complex(r1, r2):
    if len(r1._agentsAcquired) < len(r2._agentsAcquired):
        return True
```

At this point, it has to be noted that the current implementation is not using yet a real, portable network protocol, but is instead prototyped as a relationship between Python objects inside the system and the application. However, it follows this pattern closely. In the current prototype implementation, policies can be defined at many levels, and can be introduced by applications, recognizers or the system itself. In a more conservative implementation, with a network-based coordination protocol, it could be

more interesting that system-wide policies would only be defined inside the system, thus preventing arbitrary code from injected application-defined policies to be executed by the system. Application-based policies could be instead enforced at the application level. At the moment, two different systems are provided with GestureAgents source: one using pygame⁴ as a rendering engine and another using OpenGL.

5.7 The GestureAgents Recognition Framework

Apart from the agent exclusivity coordination protocol for multi-user and multi-touch interaction, GestureAgents provides a gesture recognition framework based on the same agent exclusivity concept. It provides gesture composition (i.e. describing a gesture in terms of a combination of previously defined simpler ones) by stacking layers of agent-recognizer relations, and by considering that recognized gestures can also be agents (such as double-tap agents). The framework also takes advantage of the agent exclusivity competition between recognizers for solving internal disambiguation for simultaneous instance of the same gesture recognizer, by treating them as different gestures that have to compete for the agent's exclusivity. It also provides a mechanism to quickly dismiss recognizers based on the application context.

5.7.1 Recognizer composition

We call *recognizer composition* the ability to express a gesture in terms of a combination of previously defined simpler ones. The way GestureAgents allows it, is by presenting simpler gestures as agents whose events may be used by higher-level recognizers. This is done by associating agents to Gestures in a one to one relation: A *gesture recognizer* instance, representing an ongoing gesture being performed, is associated to an agent that manages the events issued by the gesture itself. For instance, a *tap recognizer* consumes *finger* events, and is associated to the *tap agent*, which issues *tap* events that can be consumed by a *double tap recognizer* (see figure 5.11 and listing 5.1 for an example of a gesture recognizer implementation using composition).

This strategy creates a *tree of recognizers* for each top-level gesture, connected to simpler gestures via their associated agents. The ability to *complete* the top gesture will depend on receiving the exclusivity of the agents of the more simple gestures, which in their turn will compete for the exclusivity of the agents they are interested in: the *agent exclusivity* rule is enforced inside every agent individually, and the disambiguation

⁴<http://www.pygame.org/>

```

class DoubleTapAgent(Agent):
    eventnames = ("newDoubleTap",)

class RecognizerDoubleTap(Recognizer):
    def __init__(self, system):
        Recognizer.__init__(self, system)
        self.agent = None
        self.firsttap = None
        self.seconddtap = None
        self.register_event(
            self.system.newAgent(RecognizerTap), RecognizerDoubleTap.EventNewAgent)
        self.time = 0.3
        self.maxd = 10

    @newHypothesis
    def EventNewAgent(self, Tap):
        self.agent = DoubleTapAgent(self)
        self.agent.pos = Tap.pos
        self.announce()
        self.unregister_event(self.system.newAgent(RecognizerTap))
        self.register_event(Tap.newTap, RecognizerDoubleTap.FirstTap)

    def FirstTap(self, Tap):
        self.firsttap = Tap
        self.unregister_event(Tap.newTap)
        self.register_event(
            self.system.newAgent(RecognizerTap), RecognizerDoubleTap.EventNewAgent2)
        self.expire_in(self.time)
        self.acquire(Tap)

    @newHypothesis
    def EventNewAgent2(self, Tap):
        if self.dist(Tap.pos, self.firsttap.pos) > self.maxd:
            self.fail(cause="Max_distance")
        else:
            self.unregister_event(self.system.newAgent(RecognizerTap))
            self.register_event(Tap.newTap, RecognizerDoubleTap.SecondTap)

    def SecondTap(self, Tap):
        if self.dist(Tap.pos, self.firsttap.pos) > self.maxd:
            self.fail(cause="Max_distance")
        else:
            self.seconddtap = Tap
            self.unregister_event(Tap.newTap)
            self.cancel_expire()
            self.acquire(Tap)
            self.complete()
            self.fail_all_others()

    def execute(self):
        self.agent.pos = self.seconddtap.pos
        self.agent.newDoubleTap(self.agent)
        self.finish()

    def duplicate(self):
        d = self.get_copy(self.system)
        d.firsttap = self.firsttap
        d.seconddtap = self.seconddtap
        return d

    @staticmethod
    def dist(a, b):
        dx, dy = (a[0] - b[0], a[1] - b[1])
        return math.sqrt(dx ** 2 + dy ** 2)

```

Listing 5.1: A simple *double-tap* recognizer

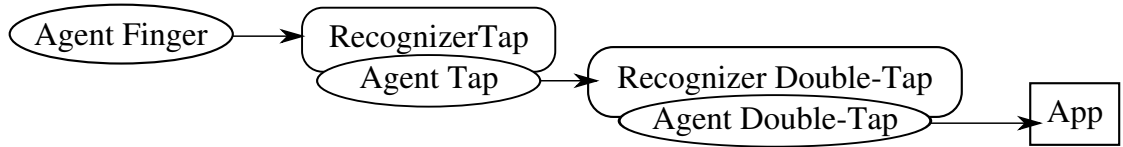


Figure 5.11: Example of composition: *double-tap recognizer* consumes *tap* events.

process spreads into different layers that are more manageable and simple (but that lack a global perspective, as we will see in Section 5.8).

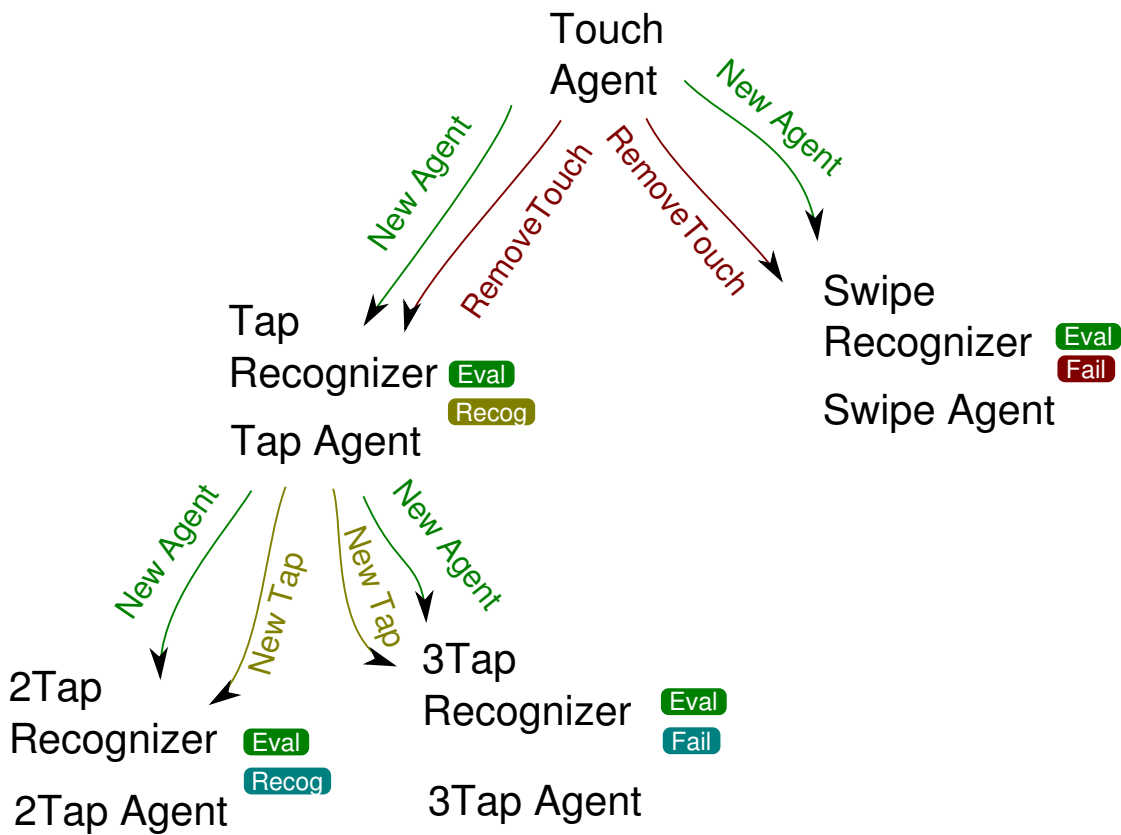
As the original Agent keeps sending events, recognizers start to match these to their gesture patterns and some will start to fail. Eventually only one branch of the original tree will remain active, only then can we consider this gesture successfully disambiguated (see Figure 5.12).

5.7.2 Recognizer instances as Hypotheses

The previous mechanism is useful not only to disambiguate between different types of gestures (swipe gesture vs tap gestures), but also to disambiguate between different possible gestures of the same type (double tap gesture 1 vs double tap gesture 2).

Allowing concurrent multi-gesture interaction allows for having the same gesture performed in two places at the same time, which can bring difficulties. A hypothetical recognizer that takes into account more than one of these gestures as input Agents must face the decision of whether a new Agent is part of the tracked gesture or not and acquire it in consequence. This can be problematic as the type of the new Agent may not be known a priori. Imagine a double-tap recognizer that has already recognized the first of its taps and is waiting for a second one, consider that before the second tap is performed another user performs a tap somewhere else on the table as part of a completely unrelated double tap (see Figure 5.13). In this case our double-tap recognizer would fail after realizing this new tap is too far away to match its gesture pattern and that would be the end of it, even though the first user might have been just about to perform a second tap.

We can use the technique of competition for Agent Exclusivity to solve this type of problems. This can be done by creating a duplicate of the recognizer instance before taking the decision of considering the new agent as part of the gesture or not. Both decisions are then evaluated, one on each instance. Eventually, these two instances must compete for the exclusivity of Agents as any other pair of recognizers. At some point, the instance that took the wrong decision will fail and let the good instance win. We



- ① New Touch on Surface
- ② Remove finger off Surface
- ③ Exclusivity granted to Tap Recognizer
- ④ After second Tap, time passes

Figure 5.12: Recognizer tree created by three recognizers registered by the application: A Swipe recognizer, a double-tap recognizer and a triple-tap recognizer. Both double and triple tap recognizers use composition with a tap recognizer. The actions regarding the user performing a double top on the surface are depicted.

consider every recognizer instance as a gesture hypothesis; every irreversible decision made generates a new hypothesis.

On our double-tap recognizer example, when the new tap is detected, a duplicate of the recognizer will be created, and only one of them will take this new agent into consideration while the other will wait for other taps. When the position of the new tap is evaluated and identified as being too far away from the first, only one of the recognizers shall fail. The same will happen when the other double-tap is completed, as seen in Figure 5.14.

As in this case the competition is not between two unrelated recognizers programmed by potentially different people, but between two instances of the same recognizer, the programmer can preview the obvious conflicts between the multiple instances generated by this method. For this reason, an instance can ask the acquired Agent to force-fail all of the competing instances of its same type when it is sure that they are wrong.

This technique can also allow for a simple *factory-like* strategy for the use of recognizers. At the beginning, one instance of every recognizer is created, this instance is in its initial state. As any time a new Agent is introduced we create a new hypothesis-instance, there will always be one (and only one) recognizer instance in the initial state, ready for new Agents to be tracked while all other instances are coupled with existing Agents. It is important to mention that none of the Agents associated to recognizers are duplicated when the presence of a new input creates a new hypothesis: An agent is not fed with events until the recognizer confirmed its source Agents, and this does not happen until only one instance is left.

5.7.3 Context polling

We mentioned before that context could be used for disambiguation. Instead of forcing the application to declare interaction areas or to manually activate and deactivate gestures at certain moments, we use a simple technique that can be called context polling. Every time a gesture to which an application is subscribed issues a new Agent, the application is asked whether it is interested in it or not. At this moment the agent only has some initial information, but nothing forbids the recognizer from asking again at any time when it has more complete data. When asked, the application can dismiss the Agent, and then the recognizer shall fail. As this is done recursively, all the intermediate recognizers over which this final gesture was edified fail as well.

An example of the benefits of this technique can be explained with a Tap and a double Tap: in contexts (areas, states...) where only a Tap is possible, the system does not

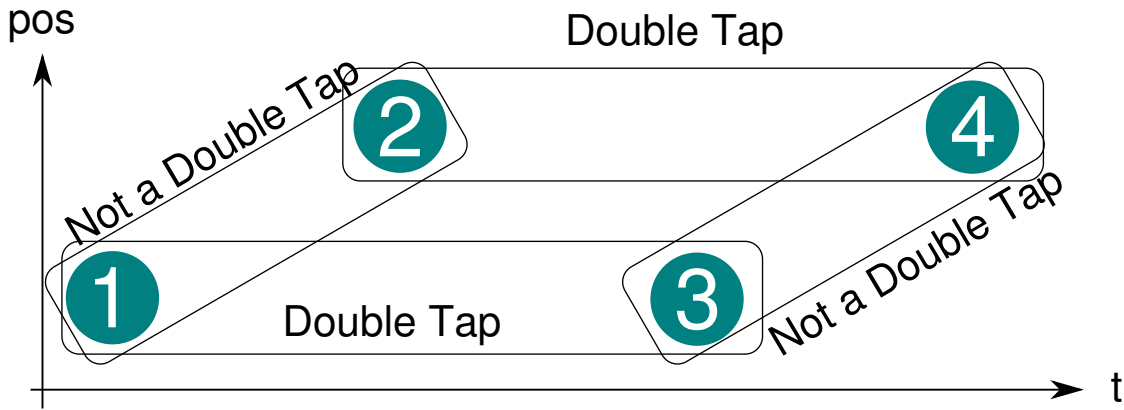


Figure 5.13: Two concurrent double-taps in different places can confuse gesture recognizers.

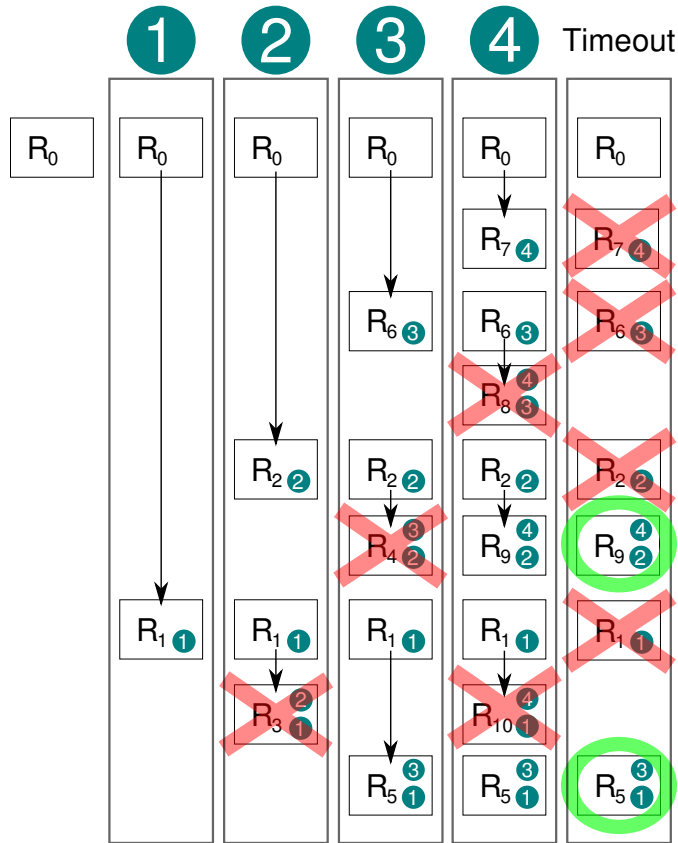


Figure 5.14: Recognizer instances as hypotheses strategy for the recognition of the two concurrent double-taps of Figure 5.13. Every irreversible decision creates a new hypothesis. At the end only the correct hypotheses are left.

have to wait until the double Tap fails in order to report the Tap; in this way the Tap is recognized faster.

An advantage of using context in this way, instead of directly coding it into the system, is that it makes the system much more flexible and independent, allowing multiple possible application management schemes: we allow using areas to delimit apps or widgets, but do not enforce doing it, as the system is agnostic. It also places the code for context evaluation where the context actually is: in the application code and not into an intermediate layer, where we should foresee all the possible aspects to take into consideration.

5.7.4 Link with the application

The application cannot directly subscribe to events from Agents. In order to enter into the competition for agent exclusivity, it would need to acquire and confirm gestures, which it cannot do because it is not a recognizer. The application needs a way to indirectly subscribe to the related Agents, which is done via a Fake Agent created by a special recognizer named `AppRecognizer` that simply accepts any kind of input. It provides the missing piece in order for an application to enter the recognizer competition. It also offers a special Agent type that mimics the original Agent but does not need to be acquired nor confirmed.

If we observe the full development of the Agent-Recognizer tree from the sensors to the application, we will always find an `AppRecognizer` just before reaching the application. Then the application only has to comply with the context polling mechanism, but not with the specific mechanics of the recognizer/Agent relationship.

5.7.5 Provided Gestures

A library of TUIO-based gesture recognizers is provided with *GestureAgents*: Tap, stick, double-tap, move and zoomrotate. Their implementation is also a guide to the good practices of creating gesture recognizers with the framework.

5.8 A second iteration: Composition with 3rd Party Apps

Recent developments in the framework have simplified the first layer of agent-recognizer relation, the one of the system-recognizer communication. By encapsulating every recognizer relationship tree inside an isolating proxy, the protocol becomes much clear and

eliminates possible incompatibility issues due to the use of the compositing feature of the gesture-recognition framework. In the previous structure, there was no distinction between end-user gestures and sub-gestures.

5.8.1 A working gesture composition example

The *agent exclusivity* mechanism is designed to prevent unconfirmed gesture recognizers to send any type of event via their associated agents and thus spread false information. For example, a *circle recognizer* will not send any circle-related event until it gets the exclusivity over its *acquired* agents. This has an important effect on gestures that take advantage of the *gesture composition* capability and split its recognition into sub-gestures: top-level gestures will not receive any event from them until they have *completed* (and so got the exclusivity over the original agents). The disambiguation process is then computed in a bottom-up approach: first sub-gestures recognize and compete for exclusivity, then top-level gestures start recognizing. Because the problem we are fixing in this section arises from this very property, it is crucial to understand this effect, although an in-depth explanation of the GestureAgents algorithm is beyond the scope of the current paper (we advise to refer to the original publication for a more detailed description).

Let us imagine a setting where there is a system that works with the GestureAgents framework, running two applications: AppA (accepting *double-tap* gestures) and AppB (accepting *triple-tap* gestures). The two applications are made by the same developer, thus it seems logical to reuse some code. In our case we end up having 3 gesture recognizers (see figure 5.15):

- A *tap recognizer* (RT), identifies *taps* from touch events issued by (raw) *touch agents*.
- A *double-tap recognizer* (RDT), identifies *double-taps* from events issued by *tap agents*.
- A *triple-tap recognizer* (RTT), identifies *triple-taps* from events issued by *tap agents*.

When the user starts performing a *tap* with her finger, a *touch agent* begins sending events (touch in, touch out). RT *acquires* the *touch agent*, blocking any other possible recognizer from getting its exclusivity. When the input events fully match the gesture (touch in + touch out = *tap*) it *completes* the gesture, requesting the agent's exclusivity. As there is no other gesture recognizer interested in it, the exclusivity is granted to RT.

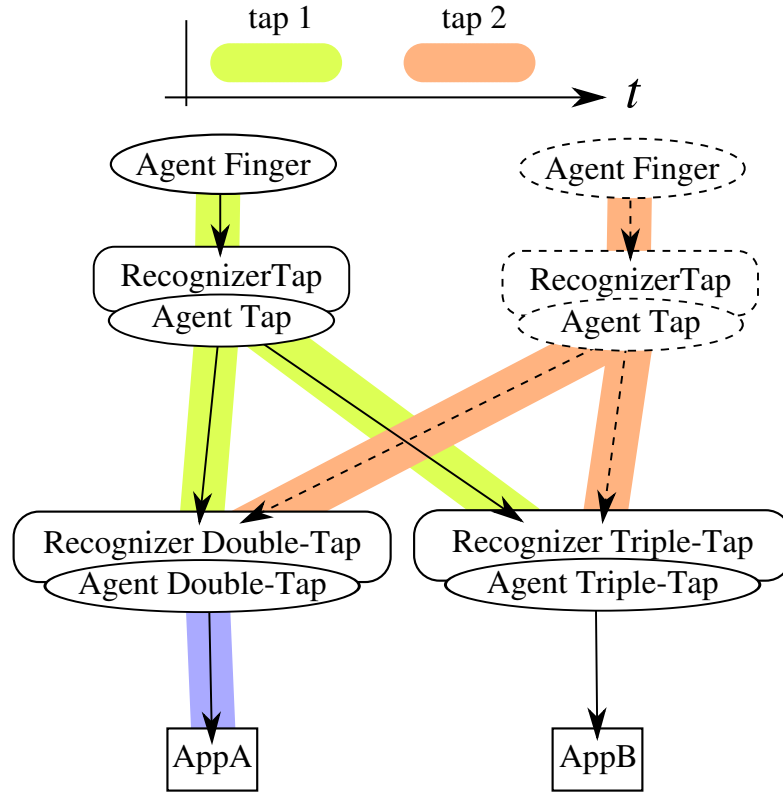


Figure 5.15: Above, a representation of a *double-tap* gesture over time. Bellow, the recognizer tree created by two applications with two gestures (*double-tap* and *triple-tap*) that share a sub-gesture (*tap*). The dashed components appear at the moment that the second *tap* is performed by the user. The colored areas highlight the paths where events actually flow at some point of time.

It is only at this point that the *tap agent* associated to RT, starts sending events - basically a *tap* event- to the RDT and RTT. Those see the agent as a possible part of their gestures (both gestures start with one *tap*) and acquire it.

When this process happens again with a second *tap*, the sequence matches with the *double-tap* gesture, so the RDT *completes* requesting the exclusivity over the acquired *tap agents*. But this is still not granted, because RTT is still blocking them.

Eventually, after some small time period, RTT realizes that there is no third *tap*, and as it does not match a *triple-tap* pattern, it *fails*. When *failing*, RTT stops blocking the two *tap agents*, allowing them to give its exclusivity to RDT. Only at this time, *double-tap agent* can send its event, a *double-tap* event to AppA.

5.8.2 Two apps, two developers and composition: Not working

The above example shows a setting that was successfully tested with the original implementation of the algorithm. Even when used by different applications, the gestures were programmed and composed using common components. This works perfectly, but what happens when the gestures are created by different developers, in the worst case scenario, composing their gestures using equivalent but different implementations of the same sub-gestures? Since the whole point of creating the GestureAgents framework was to allow multiple applications *with multiple gesture definitions* by unrelated developers to successfully share the interaction (input) interfaces, this is not a simple and isolated bug, but a major conceptual problem.

We will study this in detail with a variation of the previous problematic context, looking at the inner working of the algorithm so that we can see why this condition may lead to an error: In our same system working with GestureAgents framework, there are two different running applications by two developers, AppA (by Developer1, accepting *double-tap* gestures) and AppB (by Developer2, accepting *triple-tap* gestures), both using custom recognizers. The recognizer list would look like this (see figure 5.16):

- A *tap recognizer* (RT1), identifies *taps* from touch events issued by (raw) *touch agents* developed by Developer1.
- A *double-tap recognizer* (RDT), identifies *double-taps* from events issued by *RT1* agents developed by Developer1.
- A *tap recognizer* (RT2), identifies *taps* from touch events issued by (raw) *touch agents* developed by Developer2.
- A *triple-tap recognizer* (RTT), identifies *triple-taps* from events issued by *RT2* agents developed by Developer2.

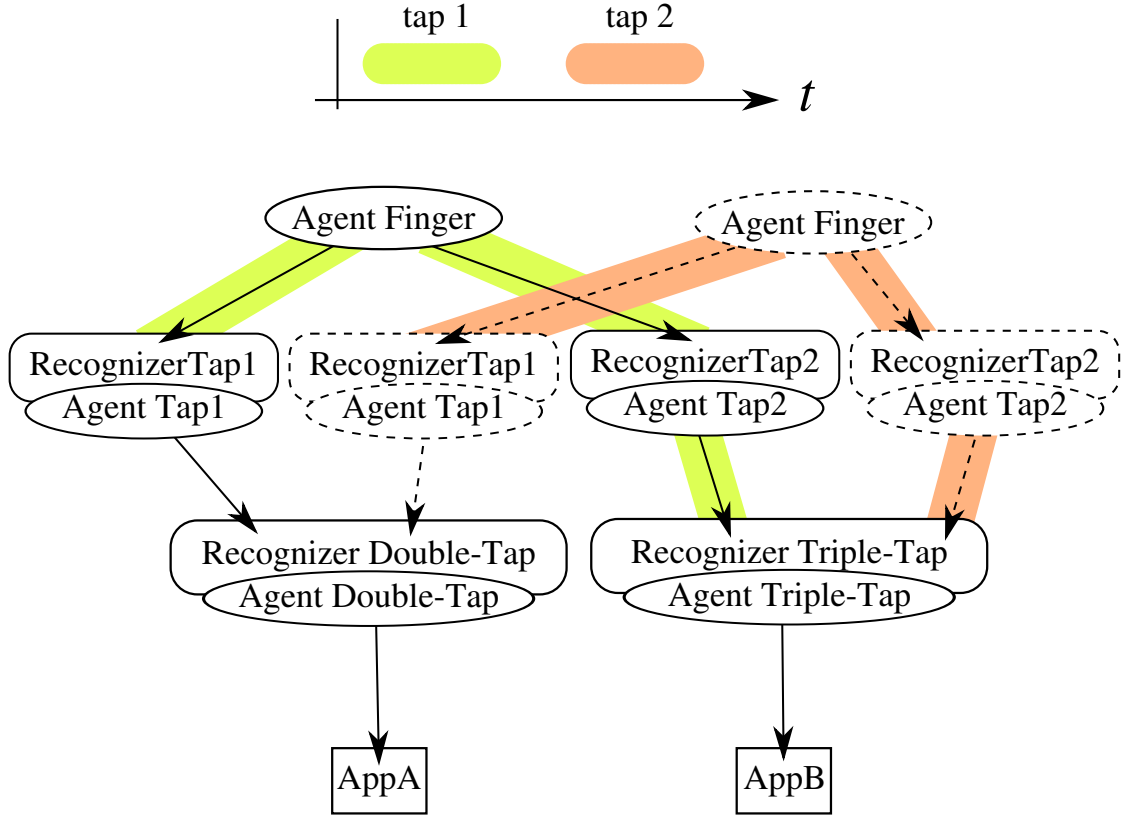


Figure 5.16: Above, a representation of a *double-tap* gesture over time. Bellow, the recognizer tree created by two applications with two gestures (*double-tap* and *triple-tap*) with different implementations of the same sub-gesture (*tap*). The dashed components appear at the moment that the second *tap* is performed by the user and colored areas highlight the paths where events actually flow at some point of time.

The same input used in the previous example, consisting of 2 valid *taps*, should theoretically match the pattern of *double-tap* and send a *double-tap* event to AppA. But this is not the case, as we can see in the actual development of the algorithm.

When the user starts performing a *tap* with her finger, a *touch agent* begins sending events (touch in, touch out). The two *tap recognizers* (RT1 and RT2) acquire the *touch agent*, preventing each other from getting its exclusivity. When the input events fully match the gesture (touch in + touch out = *tap*) they both try to complete the gesture, requesting the agent's exclusivity. The first *tap recognizer* to ask for the exclusivity is RT1, but it gets on hold, as RT2 is still blocking the agent. When RT2 asks for exclusivity to the *touch agent*, the agent has to make a decision: whether to *fail* RT1 or RT2. The existing policy "last recognizer replaces previous" requires it to fail RT1 as it

was the first requesting the exclusivity, and so is considered the less complex gesture.

Notice that, as RT1 fails, it will never send events to RDT, preventing it to recognize any type of input, even a correct one.

When RT2 gets the exclusivity it sends a *tap* event through its RT2 agent. RTT will then acquire the agent as it seems to be part of a *triple-tap*.

The previous sequence is repeated for the second *tap*, leading to a RTT with two RT2 agents acquired, waiting for a third one. Eventually, after some small time period, RTT realizes that there is no third *tap*, and as it does not match a *triple-tap* pattern, it *fails*.

The correct gesture recognizer that should have received the input -RDT- does not have the chance to even notice that there is some interesting input coming to the system, because any *tap*-shaped input will be disputed over by RT1 and RT2, and the second one will always win. The competition between two different implementations of the same gesture recognizer, which is not even directly used by the application, is preventing the competition between more high-level (valid) recognizers and thus preventing some gestures from being detected.

As we can see, the disambiguation algorithm fails because it assumes that the lower-level components of the recognizers will be shared, and that may not be the case if we allow total freedom for app developers to compose gestures when implementing gesture recognizers (as the objective of the system is to allow expressive, rich, custom gestures).

Although the mechanisms of the GestureAgents algorithm, individually, are reasonable solutions dealing with the complex problems that arise with the setting described in the motivation, combining them the way it is proposed, leads to problems like the aforementioned. Solving this problem may not be trivial, and removing any of the main features may not be desirable, as it may have an impact on several other internal mechanisms. So we must revisit the overall organization of the algorithm to deal with this.

5.8.3 Revisiting gesture composition

Agent exclusivity is the main contribution of GestureAgents, and it works perfectly when used individually. It seems though that the problem lies in the way the composition strategy splits the disambiguation problem into several local layers, which have no knowledge of the overall top-level gestures, and thus prevents information to flow to those top-level gestures without making a final decision first. It is clear that *gesture recognizer composition*, in the way defined by GestureAgents, is a valuable tool to leverage the effort put onto defining new gestures. So the obvious solution, removing composition altogether, is not desirable.

This composition strategy is not the only one possible. Other approaches exist that allow gesture composition without the creation of new gestures: GISpL (Echtler and Butz, 2012), for instance, makes the distinction between gestures and features, features being conditions that have to be met in order for the gesture to be fully recognized. Composition is allowed at a feature level, but not at gesture level. As a consequence of that, the algorithm is much more clear, as it separates the recognition from the disambiguation at all levels. This means, however, that many techniques for allowing complex gestures are not directly applicable here, as they are based in inter-gesture disambiguation and would fail with features.

Another possible solution is that disambiguation could come in two different steps: inside the tree created by a top-level gesture, and between different trees. Inside a given tree, all the components are known at development time by the developer, and can be tested and debugged. We can then maintain the original algorithm, with all its caveats as any problem can be detected. In the case of dealing with different trees of recognizers, we must change the way this is disambiguated, by adopting the philosophy of GISpL: disambiguation between complete gestures, not between features (or components). We can make this distinction by creating virtual recognizer universes and isolating the gesture tree inside them, so that, in fact, from its perspective, it is alone. The containers of this virtual universes behave like one-layer recognizers with no composition, letting *agent exclusivity* to successfully disambiguate.

We next present the modified algorithm implementing this last solution: we enable disambiguation without the burden of recognizer composition, between totally different top-level gestures, while still enabling composition in a nested gesture universe inside every top-level gesture, where no runtime incompatibilities can arise.

5.8.4 Implementation

To remove the problem of combining composition and different top-level gestures, we fake removing composition by nesting it inside a top-level fake recognizer that emulates the behavior of both the lowest-level recognizers that would normally directly receive raw input events, and the highest-level recognizer that defines the gesture being recognized. The idea is to encapsulate recognizer trees, composed by top-level gestures and sub-gestures, into a meta-recognizer that behaves like a single (not composed) gesture recognizer instead. From the program and system perspectives, it would appear as if composition is not used, and thus we avoid the problems that it introduces in the *agent exclusivity* algorithm.

We found that the best way of implementing this on top of GestureAgents, was reimplementing AppRecognizer, a dummy recognizer that already acted as a proxy between the top-level recognizer and the application, so that it would enclose all the former tree inside itself. To do this we needed to create a *proxy recognizer* (SensorProxy) that would mediate between original agents (sensors) in the global recognizer tree, and the rest of the tree, inside of AppRecognizer. Its behavior from both the global and local perspectives would be modeled to allow the local tree to progress matching gestures, without having to get the exclusivity of the sensors by the sub-gesture recognizers. From the point of view of the sensors, the proxies represent top-level gestures without composition, and from the one of the local tree, they represent the sensors, as if there was not any other top-level recognizer competing for them (see figure 5.17 to see how the failing example would look like with this variation)

As soon as the AppRecognizer receives events from the top-level recognizer, it knows that the input effectively matches the gesture and it then forces the involved proxies to ask for the exclusivity over the original sensors. Meanwhile AppRecognizer caches the events from the top-level recognizer, until all the proxies get confirmed. It then forwards the events to the application.

The first step achieved in this correction process was adapting the framework for making it work with independent recognizer trees. The original version assumes that there is only one source of agents for every recognizer type by using singletons for the NewAgent events. We thus encapsulated all the NewAgent functionalities into a class that would be the manager of the tree -the *system*-, so that recognizers did not directly subscribe to absolute NewAgent event sources, but rather request them to their *system*.

Implementation details get complex when allowing one of the features of GestureAgents: *Recognizer instances as hypotheses*. In the original framework, when a recognizer has to take an uninformed decision, such as *acquiring* an *agent* as a part of a gesture, it duplicates itself to cover both possibilities. The mechanism underlying these duplications was specially designed to work as independently as possible, making it very difficult to move this functionality to the *system* managing the tree. This means that the entire trees do not duplicate when any recognizer (top-level or not) takes this kind of decision, as it would be desirable if translating this philosophy to the meta-gestures, leaving instead a single tree (with multiple top-level recognizers, from individual duplications) shared by many AppRecognizers, each one focusing on one top-level gesture recognizer agent.

This has an impact on the way the list of proxies required to be confirmed before Ap-

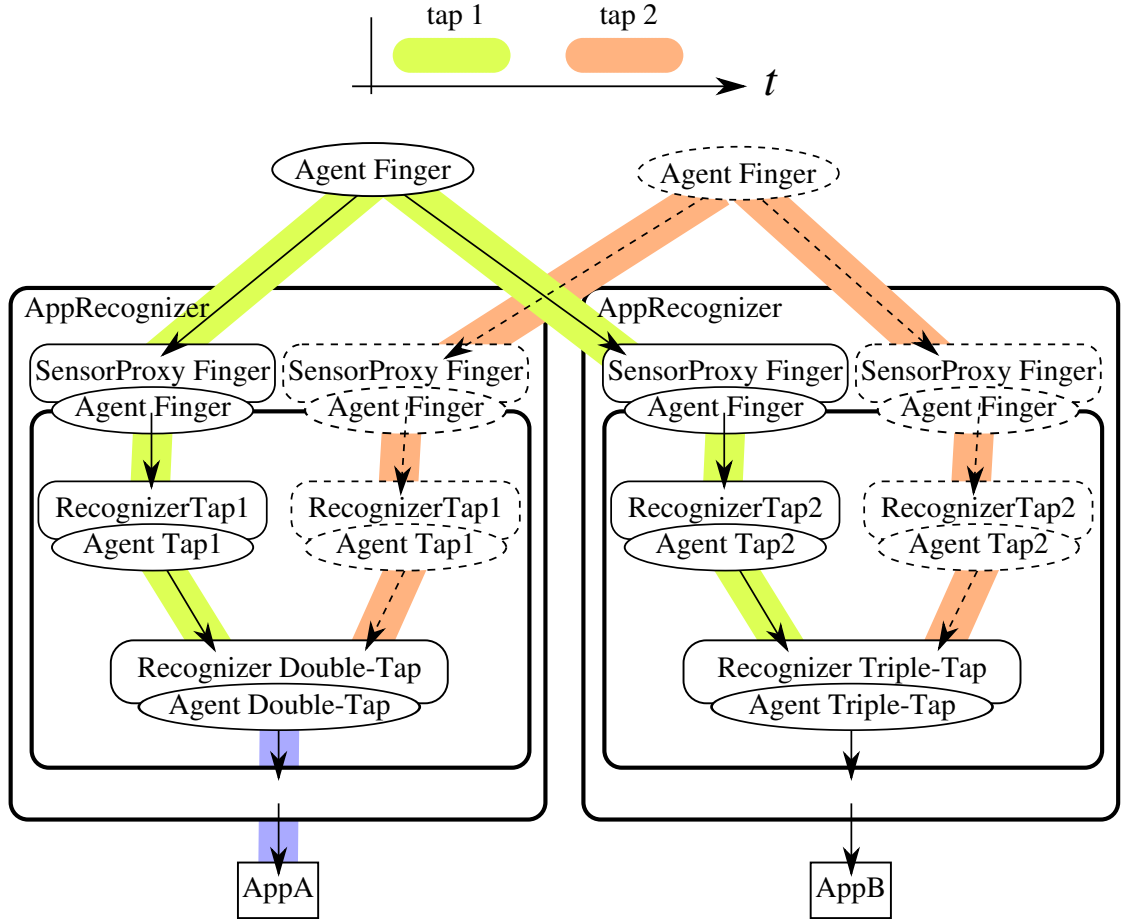


Figure 5.17: Above, a representation of a *double-tap* gesture over time. Bellow, the recognizer tree created by two applications with two gestures (*double-tap* and *triple-tap*) with different implementations of the same sub-gesture (*tap*) with the modified algorithm. The dashed components appear at the moment that the second *tap* is performed by the user and colored areas highlight the paths where events actually flow at some point of time.

pRecognizer can complete is computed: when a sub-gesture completes a sensor proxy, it is difficult to know which top-level recognizer will be finally affected by this action. The best we can do is to assume that all the subscribed top-level gestures (searched by browsing the tree) will benefit from that action. This is technically not necessarily true as, theoretically, a recognizer would be able to withdraw its interest in a sub-gesture without failing, but this capability is not used when using the *recognizer instances as hypotheses* technique, so it should not be a problem with correctly-implemented recognizers.

5.8.5 Effects on Policies

In GestureAgents, the rules solving situations of clashing gesture definitions are defined by *policies* inserted by the developer at any level. Those rules are evaluated by the agents and thus only have the information available for its own position on the recognizer tree. Similarly to the problem described in the present paper, the lack of global information at the agent level had some impacts in this aspect: policies dealing with conflicting gestures would only work for gestures that were directly competing for the same agent, but not when competing through their sub-gestures.

Also, the idea of having policies related to the applications, such as having priorities between apps, was before very difficult to implement, as many apps could perfectly share gesture recognizers.

With the modified algorithm, the recognizer trees created by top-level gestures are confined inside blocks in a per gesture and per application basis. This means that at the global level the owner of every gesture can be determined, allowing easy application-oriented policies, and top-level gestures are not masked behind sub-gestures, easing the creation of robust priorities between gestures.

5.8.6 Portability

The change we made to the algorithm has a very limited impact over the way gestures, apps and systems using GestureAgents are programmed, rendering the adaptation of previous code an easy task. Most of the changes that previous recognizers' and applications' code has to suffer relates to enabling recognizers to accept a system (a manager of a recognizer tree) as a creation parameter and to use it to subscribe to NewAgent events. As a system implementor (creating systems using GestureAgents framework), the developer will only have to provide a *system*-like class implementing the new NewAgent infrastructure. This can easily be achieved by using the framework's *System* reference

implementation as a base class. Otherwise the change should be transparent to developers and users, being able to benefit from the changes on the algorithm seamlessly.

5.8.7 Testing

Dealing with a problem related to the design of an algorithm, rather than solving an interaction issue, may be tricky. It is not a matter of how the users perceive or use the resulting work, but instead, whether the algorithm itself works or not. In our experience, solving the central issue of the present paper, it was a valuable resource having a way to programatically test the output of the algorithm given a series of known inputs.

We created a testing infrastructure to programatically test hand-crafted examples of input streams against recognizers using the framework. By defining a series of input events, the gesture recognizer to be tested, and the expected result, the testing program created an application that subscribed to the gesture recognizer and then simulated the stream of events just to check if the output was the expected.

We created several tests, covering the complex cases involving multiple recognizers that reproduced the problem described before, and also other more simple ones, to check for regressions of the framework. We consider that this kind of testing is particularly useful in complex interactive systems that pose a challenge in testing the full implementation as a whole, more than subject testing. Only after passing all the tests (including the ones that originally were failing, described before) we proceeded to informally test the changes in a live environment.

5.9 Applications and systems created with *GestureAgents*

GestureAgents has been used in several systems and applications, testing several aspects of the framework: a concurrency test application, a painting system demo, a map-browsing demo and an orchestra conducting simulator. Unless stated, the examples have been implemented in a *Reactable Experience* tabletop device⁵.

5.9.1 Original tests

In order to test the implementation of the framework, we programed an application over it to run on a *Reactable Experience* tabletop and put it to the stress of multi-user conditions (Earnshaw, 2012). In this simple application, a point is awarded whenever a

⁵http://www.reactable.com/products/reactable_experience/

gesture is performed and recognized, and users need to collect as many points as possible within a constrained time period.

The gestures used include a Tap, Double Tap, Tap Tempo (4 taps) and a variety of waveforms with different shapes and orientations (see Figure 5.18). The score system is an incentive for users to perform the gestures, and the time constraint is an incentive for these gestures to be performed fast and in an overlapping fashion. As a result we expect to have concurrent gestures occurring.

We performed experiments on it doing repeated measures over a single group of subjects, users worked both alone and in pairs. We have analyzed application's event logs measuring the rate of concurrent interaction (*CI*) defined as:

$$CI = \frac{\sum_i^{gestures} duration(i)}{InteractionTime}$$

where *InteractionTime* is the total time on where there is at least one gesture being performed.

We have found that the multi-user condition had a 11% higher concurrency ratio than the single user condition with a significance of .037, this implies that there was a meaningful difference in CI between conditions. We then conducted a questionnaire about the user's perception of how well the gestures were being identified but found no significant difference in answers between conditions, implying that the framework performed just as well on the high CI condition as on the low CI one.

5.9.2 Example Applications

A painting system constituted by two separate applications has also been created to test both the agent exclusivity competition by recognizers, and the effects of the recognition delay (see Figure 5.19, left). One application has recognizers for the tap, stick (straight line) and paint (free movement) gestures, while another uses a double-tap recognizer in a circular area. The results of the gestures of the first application are reflected in visual elements (lines, dots and traces), while the second application erases the display when a double tap is detected. As the double tap is only valid in a circular area, performing a single tap inside the area would, at first, activate also the double-tap recognizer, to end failing after a timeout call. This setting allowed to observe that the recognition delay introduced by the double-tap happened only inside the area.

A map application, featuring typical pinch zoom and drag move gestures for manipulating a world map, as well as tap and stick gestures for annotating geographical locations

Tap:



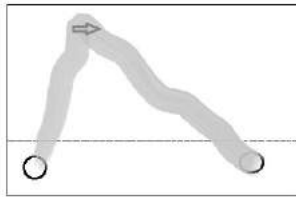
Double Tap:



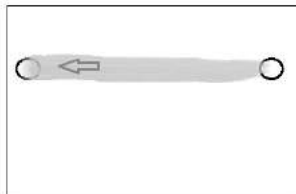
Tap tempo:



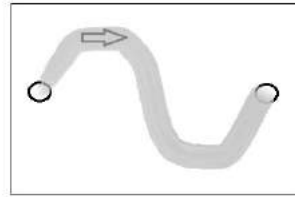
Envelope:



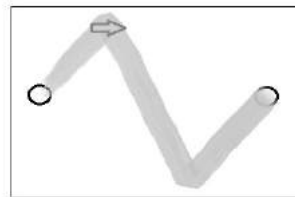
Equalizer:



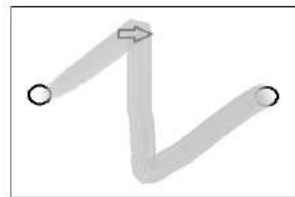
Sine wave:



Triangle wave:



Saw wave:



Three finger
swipe:

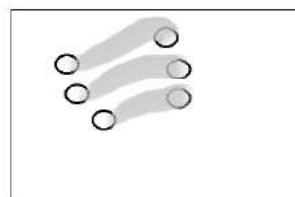


Figure 5.18: Gestures tested



Figure 5.19: A painting system (left) and a map browsing application (right) implemented in GestureAgents



Figure 5.20: Program recognizing valid gestures from Kinect recordings.

and resetting the view respectively, has been created to test the different policies (see Figure 5.19, right). The relationship between the pinch zoom and the drag move recognizers requires the first to be able to overcome the agents completed by the second, thus defining both a *compatibility_policy* and a *completion_policy* to achieve the effect.

5.9.3 Orchestra Conductor Gesture Identification

Finally, a fully “decoupled interface” application, consisting of an orchestra conductor simulator for the detection of conductor movements using a depth camera, has been developed in the context of PHENICX project (Gómez et al., 2013) (see Figure 5.20).

With the goal of creating a musical performance reinterpretation game-like experience,

we identified two different gestural information levels that could be identified both in a professional conductor level and in a nonprofessional player level. The first one addresses how the conductor's generic movement features are related to high-level and perceptive features of the music, such as loudness, complexity or tempo variation, and how they are perceived and replayed by the users. The other one is symbolic gesture identification: the actual symbols that are drawn in mid-air by the conductor that have a specific meaning to be transmitted to the orchestra. After having covered the first in the previous sections, we address the second in this one.

Hand gestures (symbols) used by conductors in real performances vary greatly. Apart from some isolated instances through the piece, most of the time they are a translation of the high-level perceptive musical properties into a movement that can be understood by the orchestra. This language is not entirely defined, but a construction that the orchestra must learn through the rehearsals with the conductor. In fact, every conductor has his own language (or conducting style), which may differ from other conductors, and dependent to the social and musical context of each concert (Konttinen, 2008).

In contrast, a set of formal, well-defined gestures for conducting exist, mainly used in teaching and rehearsal contexts. These are used to transmit the tempo and beats and have defined rules of how gestures must be performed. We think as them as good candidates for our game-like context, as they would not have to change (and be learned again) with every concert or conductor.

The recognition of the individual beat patterns from a well defined dictionary will allow us not only to achieve the game-like experience in the non-professional setting, but also to create methods to evaluate the quality of gesture performing in the context of conducting classes. We take the objective of building a game-like directing gesture rehearsal program as the motivation to build such recognition methods and testing them.

Approach

Many previous approaches to identify beat patterns (and other conducting symbols) use machine learning techniques: from the most used methods are neural networks (Ilmonen, 1999) and Hidden Markov Models (Kolesnik and Wanderley, 2004). These techniques use a set of annotated samples to train a model that will be used to classify the incoming live data. This is very appropriate to detect which of the trained symbols is more likely to represent the captured data, but totally unable to estimate the correctness of the performed symbol according to a formal definition.

To identify the gesture symbol type from the data according to a formal pattern we must

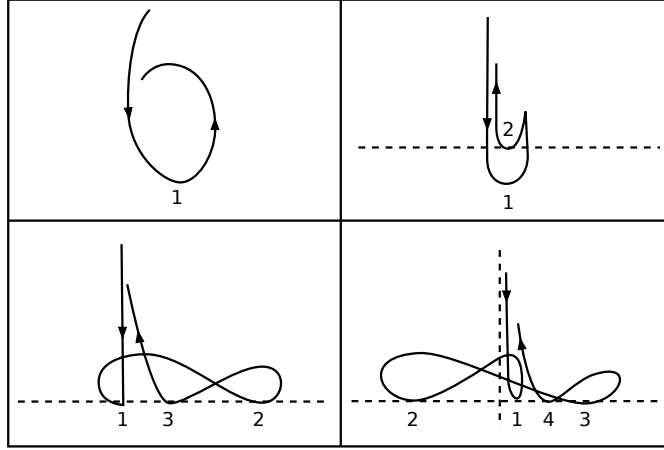


Figure 5.21: Graphical description of the gestures.

use an analytic approach that checks the user hand position and movement against this pattern. First, we study the selected gestures set to identify their mandatory components, and build a recognizer that checks if those components are present.

We acquired a corpus of performed gestures to be used as ground truth. Two sets of recordings were acquired: one of an expert performing the beat pattern gestures and another one of a real conducting teaching session. Examining those recordings we asserted that in the class setting, this particular session was focused on practicing real concert gestures and not the canonical beat patterns, rendering the recording as not useful for our case. The expert recording was then selected as the only source of ground truth, leaving out the teaching session set.

A gesture recognizer for every beat pattern was coded to match the formal definition of each gesture and then tested against the recorded data.

Implementation

We chose *GestureAgents* as our developing framework. By allowing us to program each gesture separately, we can optionally add any control gesture afterwards with minimal effort.

For the initial set of gestures we choose these beat patterns (Figure 5.21), which we will label 1T, 2T, 3T and 4T.

Those gestures are mainly characterized by a sequential vertical movement in two spaces: the upper space, where the gesture begins and ends, and the lower space, where the intermediate parts of the symbol are performed (see Figure 5.22). We started by charac-

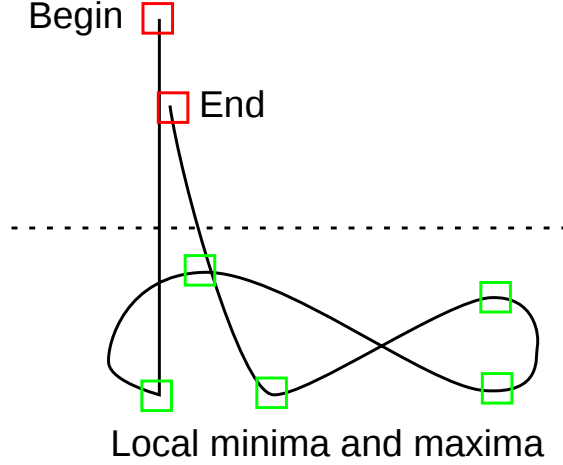


Figure 5.22: Key points, consisting of vertical local minima and maxima, in a beat pattern gesture (3T).

$k[1].y < k[0].y$	$k[2].y > k[1].y$	$k[2].y < k[0].y$
$k[3].x > k[2].x$	$k[3].x > k[1].x$	$k[3].x > k[0].x$
$k[3].y < k[2].y$	$k[4].y > k[3].y$	$k[4].y > k[1].y$
$k[4].y < k[0].y$	$k[4].x > k[2].x$	$k[4].x > k[1].x$
$k[4].x > k[0].x$	$k[5].x > k[1].x$	$k[5].x < k[3].x$
$k[5].x < k[4].x$	$k[5].y < k[2].y$	$k[5].y < k[4].y$
$k[6].y > k[2].y$	$k[6].y > k[4].y$	$k[6].x < k[4].x$
$k[6].x < k[3].x$		

Listing 5.2: Conditions for 3T recognizer.

terizing each gesture by the relative positions (not distances) of their alternating vertical local minima and maxima, obtained with a simple sliding window of size w .

$$\text{minima} = \{i \mid y_i \leq y_n \forall n \in \{i - w, i + w\}\}$$

$$\text{maxima} = \{i \mid y_i \geq y_n \forall n \in \{i - w, i + w\}\}$$

$$\text{keypoints} = \text{maxima}_0, \text{minima}_0, \dots, \text{maxima}_m, \text{minima}_m, \text{maxima}_{m+1}$$

As an example, 3T beat pattern gesture consists of 7 key points, characterized by the conditions of Listing 5.2. Note that by using relative positions instead of distances the recognizer becomes scale invariant.

Another possible mechanism would be using a temporal pattern to match the gesture instead of its spatial trajectory (Bergen, 2012). This strategy, however, requires to use

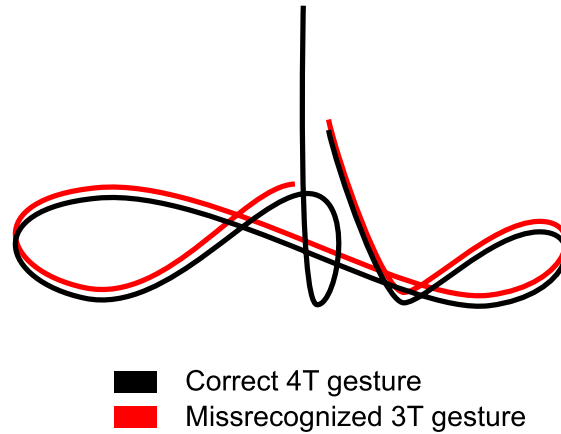


Figure 5.23: Missrecognition of a partial 4T gesture as a 3T gesture.

the actual score to predict the expected key points in time, and does not take into account the actual hand position, something essential when evaluating if the shape of a performed gesture also fits the ideal pattern.

After programming the gesture recognizers and testing them separately, we started testing them at the same time within the framework, which would have to automatically manage the disambiguation between the recognizers. We found compatibility problems between our recognizer strategy and the framework's own design, preventing successful disambiguation from happening, requiring some additional recognizing steps or a slight change on the strategy:

- In some cases the final part of one gesture could be recognized as the beginning of another. This is the case, for instance for 4T and 3T, shown in Figure 5.23. Because GestureAgents recognizers compete to have the exclusivity over the input events as a mechanism of disambiguation, a recognizer cannot successfully recognize a gesture until all other recognizers give up. This confusion, 3T recognizer trying to identify a 3T gesture in the middle of a 4T gesture, delayed the the recognition of the correct 4T gesture, preventing then the following gesture to be recognized at all (because of missed input events caused by the delay). This string of half failed recognitions has shown a flaw of GestureAgents when dealing with gestures that appear in mid-interaction, in opposition to the gestures in Tabletops, usually starting with the introduction of a new finger on the surface. We solved this problem by segmenting the hand trace using a common start/end gesture feature present on all the recognized gestures: the different vertical position.
- A related problem appeared with the concept of this type of gestures in Ges-

tureAgents: with our conducting gestures the last point of one gesture is also the first point of another. This, sharing input events between recognizers, is strictly forbidden by *GestureAgents*. We changed the segmentation code to repeat the same event at the end and beginning of the following segments to be able to feed different recognizers.

- Several other bugs of *GestureAgents* were fixed during the implementation. Most of them were simply errors never caught in a tabletop setup, as most of its gestures always start with the introduction of a new interaction Agent, instead of appearing inside an already started interaction trace.
- Also, using recognizer composition (defining a gesture in terms of another one) for segmentation prevented the gestures to compete due to the gesture isolation strategy created by *GestureAgents* to allow gestures from different programs to compete regardless of its composition (see Section 5.8). It seems clear that this strategy is problematic in this context, and does not always allow having composition inside the gesture isolation. We faced this problem by extracting the segmentation from the gesture recognizer category and treating it like a sensor (a recognizer that is used as a primary means of interaction events). Another possible solution would have been implementing the segmentation inside the recognizer avoiding composition altogether.

After resolving the aforementioned issues, the final implementation works well and identifies all the gestures that adhere to the definition from the recording.

Testing

For the ease of testing we implemented an alternative input method based on Leap Motion⁶, a depth sensor designed to capture hand and tool movement, instead of full body. This implementation allows rapid testing without having to set up a full-body Kinect scenario.

Informal testing shows that it is possible to recognize correct gestures in real time.

5.10 *GestureAgents* Code

The *GestureAgents* framework is open source and available to anyone for use and improve. The code can be found in the following repository: <https://bitbucket.org/chaosct/gesture->

⁶<https://www.leapmotion.com/product>

agents, and videos of some of the examples can be found at <http://carles.fjulia.name/gestureagentsvideos>. The resulting algorithm for the second iteration can be found at the original GestureAgents git repository, in the branch *composition2*.

5.11 Discussion on the framework

The GestureAgents approach to provide multi-tasking to shareable interfaces is still in a prototype stage and can primarily serve as a starting point to explore this type of application-centric distributed gesture recognition strategy. This means that many aspects regarding the real world usage of such mechanism are still to be explored and discussed in depth.

5.11.1 Accessory agents

Sometimes an agent can be related to a gesture, but only as a reference. For instance, while doing a circle around a puck, the object can be accessory but not core to the gesture, therefore it will not be governed by the rule of gesture exclusivity and can be shared by several gestures at the same time: imagine that a gesture that links two objects is performed at the same time as the object-circling gesture referenced above. Both gestures can have a single object as a reference point and yet not be exclusive to each other.

How to solve this kind of gesture conflicts is an open question. One possible approach could be implementing the gesture recognizer in a way that the accessory agent is acquired but never confirmed; thus temporarily blocking its use as a core part of another gesture but being able to be shared with other gestures as an accessory agent. This approach could surely work in the first version of GestureAgents, but the changes created in the second iteration isolating the gestures by the use of proxies (see Section 5.8) makes this strategy impossible. So an explicit solution in the foundations of GestureAgents can be needed.

5.11.2 Temporary Feedback

As the framework has no standard set of gestures and the applications do not receive events until the disambiguation is complete, it is unclear how to implement temporary feedback to the user while a gesture is still not defined. Several approaches are possible,

but we must find the most flexible one. As the application is responsible for the feedback, yet the information about the hypothetical gesture is only in the recognizer (and potentially duplicated across hypotheses in several recognizers), a way to bridge it to the application is needed until a single gesture can be clearly identified and executed.

Defining a new kind of event that can be sent from the gesture recognizer to the application (through the AppRecognizer), representing only temporary unconfirmed data to display, could be a way of solving this problem, although how to present such amount of possible gestures being recognized is unclear.

5.11.3 Supporting Other Gesture Recognizing Techniques

One of the key aspects of GestureAgents is the separation of gesture recognition from disambiguation, in a fashion that they are theoretically independent, as long as some rules are complied. It would be interesting to implement other gesture recognition algorithms that complied with the protocol.

Domain specific languages and machine learning techniques such as the ones presented in Section 5.5.1 could be a perfect fit, as their intention is to ease the gesture programming. Also, gesture reusability between those systems and GestureAgents could be very useful.

5.11.4 Security

A typical concern of such system could be its resilience against ill-behaved programs. An application that unintentionally grabs input events without releasing them, could effectively block all other programs from receiving the exclusivity over the agents to fully recognize their gestures, unless specific policies preventing or limiting this type of behavior were implemented.

Even malware could register similar or identical gestures to the ones from legit programs, in order to *steal* those to insert its malicious content. Again, careful policies would have to be designed to limit this kind of attacks, such as using proximity to areas to prioritize conflicting gestures. However, the experience with PC malware tells us that it is very difficult to be protected from malicious applications.

Another security-related concern is whether an application could steal secrets from our interaction with other programs. As in GestureAgents every process can receive all input information to check if it fits a particular gesture, it is sensible to think that keylogger-like applications could be effectively developed. Being the situation similar to the PC's in this case, we can learn from its implemented strategies to solve that particular

problem. Some operating systems implement a way to interact with a specific dialog that is isolated from all the other processes, in order to enter a password or to confirm an action that requires specific privileges. A possible solution in GestureAgents could be based on this same idea.

5.11.5 Efficiency

Other issues related to efficiency could be relevant. The GestureAgents strategy simply distributes the events to the applications, leaving the recognition to them. In this perspective, it does not pose any relevant computing burden. Additionally, the restrictions imposed on the recognizers' behavior favors incremental gesture recognition approaches, which are computationally cheap. In fact, the informal experience through the different exposed tests and demos, does not clearly reveal any perceptually relevant impact by GestureAgents.

That said, in current systems, input events are either processed in a central engine before distributing them to the applications, or are filtered by area (or by focus point) before being processed inside the application. In GestureAgents many applications can be processing the same events at the same time, multiplying the needed processing power. At least with the current implementation, this effect seems inevitable.

Existing centralized gesture recognition engines that recognize several hypothetical gestures simultaneously are making efforts to parallelize this processing while guaranteeing soft real-time (Marr et al., 2014). In GestureAgents the processing of gestures in different applications would be done in parallel by definition, although without real-time guarantees.

Overall, we think that the identification of the problem of the lack and need of multi-user concurrent multitasking, and our approach to the solution contribute to the current state of the art. By proposing a content-based disambiguation instead of an area-based one, GestureAgents approach can be a valid solution for multi-tasking, in both coupled and decoupled shareable interfaces, revealing itself as a generic solution. This can be increasingly relevant for new upcoming decoupled interfaces such as hand tracking sensors or depth cameras, which could benefit from policies and strategies developed for other more popular interfaces.

5.11.6 Debugging and Testing

The event-driven nature of GestureAgents as a whole, and particularly the diversity of places where the decision of failing a particular recognizer can be failed, make debugging it very difficult: The program flow is not linear, as some events are buffered for later use, expecting some of the consequences of actions to be effective before others.

When something was not working properly, the only way to inspect what was happening was saving long logs to be inspected later showing the execution flow, which made no particular sense. We would say that this is the biggest challenge of GestureAgents: making it debugging-friendly, clarifying the execution so it can be inspected.

A visual display of the running gesture recognizers around the related agents on the surfaces was actually implemented, but the great volatility of new recognizers instances as hypotheses made it of little use to visually understand what was happening.

To implement the second iteration, we felt that only structured testing could be of any help to understand if we were actually breaking something. We found that programmatic gesture testing can be a very valuable resource for testing the completeness of this kind of solution, more than user testing, as the main reason for testing in this case is the correctness of the algorithm.

A public repository of interaction recordings for testing gesture recognizers could be very useful for testing implementations between systems. This could be a very useful work to be done by and for the community.

5.11.7 Considerations on the second iteration

The original GestureAgents framework had some defects that affected its main goal. We analyzed these in order to identify their cause, finally tuning the algorithm to solve them (Section 5.8).

It seems clear that *agent exclusivity* algorithm can only work with single-level recognizers, independently of them having sub-trees, as we implemented as a solution. We think that this is an important finding, as it sets the limits of this method.

Although we introduced a solution that solves that case, it is not a major change on the philosophical sense, neither it fundamentally changes the components of the algorithm, and thus we can still benefit from its original properties. The top-level algorithm is now clearer, easier to understand and manage, while we keep the more complex interaction between recognizers as an option hidden to it.

However, it has made the internal organization of classes and program flow of the implementation very difficult to understand. By making it backwards-compatible, the new version has generated multiple classes and functions whose sole purpose is hiding the real complexity of the new system and pretending to be the old one.

5.11.8 Considerations on decoupled interfaces

The experience of using *GestureAgents* to build gesture recognizers for a decoupled interface, described in Section 5.9.3 revealed us some problems.

Despite being created as generic as possible, *GestureAgents* was developed and tested basically in tabletops. This means that some assumptions were remained hidden until a different approach was used.

The main assumption that was inserted into *GestureAgents* that unconsciously limits its potential use with decoupled interfaces, is that gestures usually start with the appearance of an agent. In a tabletop that would be, for instance, touching the interface. It is totally reasonable to think that, in interfaces where the users have an implicit or explicit mechanism to signal the beginning of a gesture, they will use it; and the appearance of a new agent detected by the sensors is a very evident way to signal it.

However, interfaces like the ones using full body interaction do not provide this capability of distinguishing whether the sensors start to detect something. Instead, gestures may occur at any time. The fact that gesture recognizers are typically interested by new agents, and because of that acquiring an agent and generating a new hypothesis is typically done then, rises the question of what to do when the starting point of a gesture is not known immediately. If we create a new hypothesis for every new movement event, would it be possible to handle that amount of recognizers?

The fact is that *GestureAgents* was designed so the only gesture generation event could be the introduction of a new (or recycled) event. If that is not the case, the continuous appearing of new recognizers interested on the agent will prevent the right recognizer to ever get its exclusivity. In the presented example that was mitigated with the segmentation recognizer that detected key points of the movement where new recognizers could appear. However, as exposed, the segmentation had to be placed outside the *AppRecognizer* infrastructure created for the second iteration (Section 5.8), because of the same assumptions.

But the problem is still there: disambiguation lag could affect future gesture recognizers trying to get exclusivity to a legitimate gesture. As in this case that the gestures would be chained following one to another, not successfully recognizing the gesture at the time

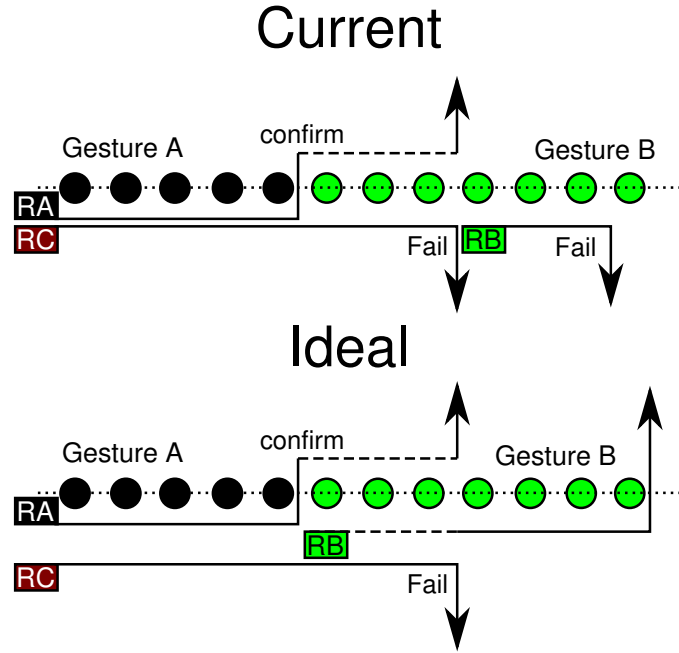


Figure 5.24: Disambiguation delay can prevent consequent gestures from being recognized. In a stream of events, two consecutive gestures exist (A and B). Gestures recognizers RA and RB try to recognize their correspondent gestures (A,B), while recognizer RC is still deciding if its gesture matches the events. Notice how the competition between RA and RC prevents RB from recognizing gesture B (Current situation, above). A an ideal implementation should allow RB to use the events from gesture B that were temporarily blocked by RC.

of the gesture ending would prevent the next recognizers from finding an agent in a clean state, and this would prevent the previous recognizer to ever get its exclusivity, which at its turn would do the same to the next one.

The fact is that disambiguation delay happening at the end of the gesture prevents the next ones to use the agent, even if the original recognizers are no longer interested on its events (see Figure 5.24 for an illustration).

Taking this case into account would require a major reformulation of *GestureAgents*: instead of acquiring agents for a time period, recognizers would have to acquire sequences of events, where they can confirm their gesture while leaving the next events free to use by other recognizers.

In short, focusing on sequences of events instead of on agents.

5.11.9 Future steps

When creating GestureAgents, we started with an idea that changed and grew over time, and its implementation reflects that. The lessons learned in the process, the final design of the elements and their interaction are now sufficiently complete to think about reimplementing it from scratch.

Given the discovered limitations, we think that a major rewrite of GestureAgents from the learned lessons, focusing on the protocol first, with a real network protocol in place, could be a good start. The mentioned problems could be addressed in the design before starting the implementation so they do not complicate things by trying to maintain a backwards-compatible system.

5.12 Conclusions

What started with the attempted solution to the problems identified in *ofxTableGestures* regarding simultaneous gestures and composition, developed into a much broader solution to a problem that shared some characteristics with the original one.

We have identified and exposed the need of multi-tasking capabilities in shareable multi-user interfaces. We have argued about the utility of multi-tasking when solving complex tasks with computers, and showed that multi-tasking features are currently missing in actual shareable interfaces, despite the fact that one of their main goals is complex task solving through collaboration between users.

We argue that this lack is not unintentional but a consequence of the difficulty of adapting current multi-tasking-capable systems into shareable interfaces.

An analysis of the complexities of implementing such a system together with a discussion of possible strategies has been carried, revealing that “area-based input events distribution” or “gesture language definition-based” approaches may pose problems in the context of rich interaction and decoupled interfaces. A third approach, using a protocol to control input event distribution but leaving gesture recognition to the application has been described and considered as the best choice.

An implementation of this approach, GestureAgents, has been presented as a possible solution, which implements the third of these strategies.

Examples of use of the framework have been finally presented, showing some of the possibilities of multi-user multi-tasking interaction and the potential of the framework itself, as well as its limitations, also analyzed.

5 Multi-Application systems: GestureAgents

We still feel that this exposed problem is relevant, and will be increasingly so. The appearance of new decoupled interfaces, such as Leap Motion, will require to distribute their events to the applications, not only based in their focus, but also in the gesture itself. We think this approach will be valuable for them.

6 Conclusions

We began this thesis with an example of a computing device, the tablet PC, the widespread adoption of which was delayed by many years because of a bad start. Despite the many capacities and advantages of these devices, a series of wrong decisions (or a precipitation on the commercialization) prevented them from becoming useful to widespread users.

Tangible tabletop technology has been ready for some years now, but the predicted widespread adoption is still pending. As researchers, we should also analyze the case of this poor adoption, to prevent it from having a worst fate to that of the first tablet PCs (by not having a second chance).

We have seen that tangible tabletops can be extremely useful. Their distinctive characteristics can actually contribute to making some tasks easier and other tasks possible; their collaboration facilities can ease problem-solving processes requiring many points of view, task parallelization or control sharing; they can help the thinking process with rich gestures, kinesthetic memory, epistemic actions, physical constraints or tangible representations of a problem; and their vast screen promotes screen multiplexing, complex visualization and monitoring, by enhancing the output bandwidth.

We presented and described two examples of tabletop applications that tried to take advantage of these capabilities, TurTan and SongExplorer, showing the potential of such devices. TurTan shows how a structured activity such as programming can be easier when it is made tangible, and with real-time visualization, and how this same tangibility can impact the learning process in education. SongExplorer shows how collection browsing and map navigation abilities can be used by tabletops to help solving problems involving massive data.

Using our own experiences from building tangible tabletop systems, and from teaching and supporting students in building such systems, we realized the need of programming frameworks adapted to these devices, helping programmers with difficult or repetitive tasks, but also enforcing good design and programming strategies. To cover this need we created two frameworks, *ofxTableGestures* and MTCF, addressing two different collectives: programmers and art and music students, respectively.

The process of building and enhancing these frameworks, driven by the needs of new students every year, allowed us to identify the activities they struggled with the most and addressing them. As seen in the current literature, creating gesture recognizers is one of the parts that definitely profits the most from framework infrastructure.

A broader analysis of the tabletop application scene gives us the hint that multitasking, an important factor for complex task solving, is missing in tabletop systems. We analyzed how the combination of multitasking and multi-user interaction poses a challenge on the distribution of the input events of the system. This problem, which in our opinion caused the lack of multitasking in tabletops, is also relevant in other kinds of shareable interfaces. To solve it, we proposed *GestureAgents*, a context-based but application centric framework for gesture disambiguation in concurrent multitasking multi-user systems. We do not propose it as a definitive software solution, but mainly as a proof of concept in order to design a blueprint. We tested it in several contexts and we showed its flaws, which need to be addressed in order to build an uncoupled interface compatible framework.

6.1 Contributions

Along the path of *Making Tabletops Useful* we have made several contributions. Some of them are original in this document, others have already been published in the form of academic papers (see Appendix A) and others in the form of software. We briefly list some of the main contributions below.

Published Contributions

- With *TurTan* we contributed by exploring how to adapt programming (and specifically tangible programming) for a tangible tabletop device so it could exploit the alleged creativity-stimulator traits.
- *SongExplorer*'s contribution was to adapt a general problem; interesting music finding in large databases, by exploiting the tabletops' capacity of exploratory interaction.
- *Musical Tabletop Coding Framework* contributed a novel way to have direct prototyping for sound and music tabletop applications.
- *GestureAgents* framework contributed to explore the novel problem of gesture disambiguation in multi-user concurrent multi-tasking systems.

Original Contributions

- We contributed to understand how learning activities could be affected by using tangible versions of tools such as Logo (TurTan).
- We analyzed the impact of collection ordering in the interesting music finding process of SongExplorer.
- We explored the typical interaction design strategies which emerged in a tabletop application creation course (TSI).
- We exposed how a second iteration of GestureAgents solved an incompatibility between the composition strategy and the disambiguation mechanism.

Software Contributions

- *ofxTableGestures* framework was developed to help tabletop application programmers to create applications that could exploit the distinctive tabletop characteristics. It is open source, available and currently used.
- Musical Tabletop Coding Framework had a goal to support sound and music artists to create tabletop versions of their projects. It is also open source, available and currently used.
- GestureAgents framework, which is in a more experimental stage, addressing the problem of gesture disambiguation in a distributed application-centric fashion is also open source and available.
- Neither TurTan nor SongExplorer are currently available to the public, due to the lack of efficient distribution mechanisms for tabletop applications.

6.2 Future Work

We have just opened the discussion on multi-user concurrent multitasking in shareable interfaces, and the community has to join this discussion and contribute with different solutions and strategies, before settling for a final approach. Even if the identified flaws in GestureAgents are addressed, it is probably still one of the many possible approaches.

The GestureAgents framework itself needs to be re-built. We can profit from the experience, the developed strategies and the architecture, but a clear implementation, using the final design, with a real-world focus, is still needed.

6 Conclusions

The presented application frameworks are in continuous evolution, and in spite of being single-task oriented, are still necessary at this moment. No major changes are planned, yet.

We can only ask that we (the research community) solve those issues in future, because all the research done on tabletops will only be useful at the end when people use them. May be the next breakthroughs in science and technology are made with the mediation of tabletops, thanks to their unique paradigm-changing capabilities. It would be a shame if we deprive humanity of these tools.

Bibliography

- Harold Abelson and Andrea A Di Sessa. *Turtle geometry: The computer as a medium for exploring mathematics*. the MIT Press, 1986. 3.5
- Christopher James Ackad, Anthony Collins, and Judy Kay. Switch: exploring the design of application and configuration switching at tabletops. In *ACM International Conference on Interactive Tabletops and Surfaces - ITS '10*, page 95, New York, New York, USA, nov 2010. ACM Press. ISBN 9781450303996. doi: 10.1145/1936652.1936670. 5.3, 5.4.1
- Edith K Ackermann. Constructing knowledge and transforming the world. In *A learning zone of one's own: Sharing representations and flow in collaborative learning environments*, chapter 2. IOS Press, 2004. 3.5.1
- Iyad AlAgha, Andrew Hatch, Linxiao Ma, and Liz Burd. Towards a teacher-centric approach for multi-touch surfaces in classrooms. In *ACM International Conference on Interactive Tabletops and Surfaces - ITS '10*, page 187, New York, New York, USA, nov 2010. ACM Press. ISBN 9781450303996. doi: 10.1145/1936652.1936688. 5.3, 5.4.1
- William Albert and Thomas Tullis. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics*. Morgan Kaufmann Publishers Inc., 2008. ISBN 0124157920. 3.5.5, 3.5.5
- Daniel Arfib and Loïc Kessous. Gestural control of sound synthesis and processing algorithms. In *Gesture and Sign Language in Human-Computer Interaction*, pages 285–295. Springer, 2002. ISBN 3540436782. 2.3
- Till Ballendat, Nicolai Marquardt, and Saul Greenberg. Proxemic interaction. In *ACM International Conference on Interactive Tabletops and Surfaces - ITS '10*, page 121, New York, New York, USA, nov 2010. ACM Press. ISBN 9781450303996. doi: 10.1145/1936652.1936676. 5.3
- Liam J Bannon. Cscw: An initial exploration¹. *Scandinavian Journal of Information Systems*, 5:3–24, 1993. 3.1.2
- Gary Beauchamp. Teacher use of the interactive whiteboard in primary schools: towards

Bibliography

- an effective transition framework. *Technology, Pedagogy and Education*, 13(3):327–348, oct 2004. ISSN 1475-939X. doi: 10.1080/14759390400200186. 5.3
- James Begole, Mary Beth Rosson, and Clifford A. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer-Human Interaction*, 6(2):95–132, jun 1999. ISSN 10730516. doi: 10.1145/319091.319096. 3.1.2
- E. Ben-Joseph, Hiroshi Ishii, John Underkoffler, B. Piper, and L. Yeung. Urban simulation and the luminous planning table: Bridging the gap between the digital and the tangible. *Journal of Planning Education and Research*, 21(2):196–203, dec 2001. ISSN 0739-456X. doi: 10.1177/0739456X0102100207. 2.4.1
- Ross Bencina, Martin Kaltenbrunner, and Sergi Jorda. Improved topological fiducial tracking in the reactivision system. 2005. 3.2, 3.4, 4.1.1
- Sakari Bergen. *Conductor Follower: Controlling sample-based synthesis with expressive gestural input*. PhD thesis, 2012. 5.9.3
- Eric A. Bier, Maureen C. Stone, K. Pier, and William A. S. Buxton. Toolglass and magic lenses: The see-through interface. In *Siggraph '93*, 1993. 2.3, 2.4
- Durrell Bishop. Visualising and physicalising the intangible product. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction - TEI '09*, page 1, New York, New York, USA, feb 2009. ACM Press. ISBN 9781605584935. doi: 10.1145/1517664.1517667. 2.4
- Alan F Blackwell. The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction*, 13(4):490–530, dec 2006. ISSN 10730516. doi: 10.1145/1188816.1188820. 2.2, 2.2.1
- Muriel Bowie, Oliver Schmid, Agnes Lisowska Masson, and Béat Hirsbrunner. Web-based multipointer interaction on shared displays. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work - CSCW '11*, page 609, New York, New York, USA, 2011. ACM Press. ISBN 9781450305563. doi: 10.1145/1958824.1958926. 5.2
- N. Brosterman and K. Togashi. *Inventing kindergarten*. Harry N. Abrams, 1997. 3.5.1
- J Bruner. *Theory of Instruction*. Harvard University Press, Cambridge, Mass., 1966. 3.5.1
- William A. S. Buxton. Living in augmented reality: Ubiquitous media and reactive environments. *Video Mediated Communication*, pages 363–384, 1997. 2.4.1

- William A. S. Buxton and B. Myers. A study in two-handed input. *ACM SIGCHI Bulletin*, 17(4):321–326, apr 1986. ISSN 07366906. doi: 10.1145/22339.22390. 2.3
- Baptiste Caramiaux and Atau Tanaka. Machine learning of musical gestures. In *Proc. of NIME 2013*, 2013. 5.4.4, 5.5.1
- Baptiste Caramiaux, Frédéric Bevilacqua, Bruno Zamborlin, and Norbert Schnell. Mimicking sound with gesture as interaction paradigm. Technical report, IRCAM - Centre Pompidou, Paris, France, 2010. 5.5.3
- Alejandro Catalá, Javier Jaen, Betsy van Dijk, and Sergi Jordà. Exploring tabletops as an effective tool to foster creativity traits. In *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction - TEI '12*, volume 1, page 143, New York, New York, USA, feb 2012. ACM Press. ISBN 9781450311748. 2.5, 3.1.3, 5.2
- Òscar Celma. *Music Recommendation and Discovery: The Long Tail, Long Fail, and Long Play in the Digital Music Space*. Springer, 2010. ISBN 3642132871. 3.6
- J Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 21(7):526–534, 1973. 4.2.1
- Andy Crabtree, Tom Rodden, and John Mariani. Collaborating around collections. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work - CSCW '04*, page 396, New York, New York, USA, nov 2004. ACM Press. ISBN 1581138105. doi: 10.1145/1031607.1031673. 3.1.2
- Mark Danks. Real-time image and video processing in gem. In *ICMC97*, 1997. 4.2.2
- Alessandro De Nardi. Graffiti-gesture recognition management framework for interactive tabletop interfaces, 2008. 5.5.1, 5.5.3
- Paul Dietz and Darren Leigh. Diamondtouch: A multi-user touch technology. In *Annual ACM Symposium on User Interface Software and Technology UIST*, pages 219–226, Orlando, 2001. 3.2
- Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78, oct 2012. ISSN 00010782. doi: 10.1145/2347736.2347755. 5.5.1
- Nicolas Earnshaw. *Evaluating an agent-based gesture recognizing framework*. PhD thesis, Universitat Pompeu Fabra, 2012. 5.9.1
- Florian Echtler and Andreas Butz. Gispl. In *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction - TEI '12*, page 233, New York, New York, USA, feb 2012. ACM Press. ISBN 9781450311748. doi: 10.1145/2148131.2148181. 5.4.4, 5.5.1, 5.8.3

Bibliography

- Florian Echtler, Gudrun Klinker, and Andreas Butz. Towards a unified gesture description language. In *HC'10*, pages 177–182, Aizu-Wakamatsu, Japan, dec 2010. University of Aizu Press. 5.5.1
- John Greer Elias, Wayne Carl Westerman, and Myra Mary Haggerty. Multi-touch gesture dictionary, nov 2007. 5.4.3
- Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Record*, volume 18, pages 399–407, jun 1989. doi: 10.1145/66926.66963. 5.2
- Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*, page 395, New York, New York, USA, dec 1968. ACM Press. doi: 10.1145/1476589.1476645. 2.3
- Ylva Fernaeus and Jakob Tholander. Finding design qualities in a tangible programming space. In *Proceedings of the SIGCHI conference on Human Factors in computing systems - CHI '06*, page 447, New York, New York, USA, apr 2006. ACM Press. ISBN 1595933727. doi: 10.1145/1124772.1124839. 3.1.2
- Ylva Fernaeus, Jakob Tholander, and Martin Jonsson. Beyond representations: towards an action-centric perspective on tangible interaction. *International Journal of Arts and Technology*, 1(3/4):249–267, 2008. ISSN 17548853. 3.1.2
- Wally Feurzeig, Seymour Papert, M Bloom, R Grant, and Cynthia J. Solomon. Programming-languages as a conceptual framework for teaching mathematics. final report on the first fifteen months of the logo project. nov 1969. 3.5, 3.5.1
- Rebecca Anne Fiebrink. *Real-time human interaction with supervised learning algorithms for music composition and performance*. PhD thesis, Princeton University, 2011. 5.5.1
- Kenneth P. Fishkin. A taxonomy for and analysis of tangible interfaces. *Personal and Ubiquitous Computing*, 8(5):347–358, 2004. ISSN 1617-4909. doi: 10.1007/s00779-004-0297-4. 5.4.1
- George W. Fitzmaurice. Graspable user interfaces. jan 1996. 3.1.1, 3.1.3, 5.2
- George W. Fitzmaurice, Hiroshi Ishii, and William A. S. Buxton. Bricks: laying the foundations for graspable user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 442. ACM Press/Addison-Wesley Publishing Co., 1995. 2.4, 2.1, 3.1.1, 5.5.1
- Lawrence Fyfe, Sean Lynch, Carmen Hull, and Sheelagh Carpendale. Surfacemusic: Mapping virtual touch-based instruments to physical models. In *Proceedings of the*

- 2010 conference on New interfaces for musical expression*, pages 360–363. Sydney, Australia, 2010. 4.2
- Ombretta Gaggi and Marco Regazzo. An environment for fast development of tabletop applications. In *Proceedings of the 2013 ACM international conference on Interactive tabletops and surfaces - ITS '13*, pages 413–416, New York, New York, USA, oct 2013. ACM Press. ISBN 9781450322713. doi: 10.1145/2512349.2514917. 5.3, 5.4.1
- Daniel Gallardo, Carles F. Julià, and Sergi Jordà. Turtan: A tangible programming language for creative exploration. In *3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems, 2008. TABLETOP 2008*, pages 89–92. Ieee, IEEE, oct 2008. ISBN 978-1-4244-2897-7. doi: 10.1109/TABLETOP.2008.4660189. 3.5, 5.5.1
- William W. Gaver, John Bowers, Andrew Boucher, Hans Gellerson, Sarah Pennington, Albrecht Schmidt, Anthony Steed, Nicholas Villars, and Brendan Walker. The drift table. In *Extended abstracts of the 2004 conference on Human factors and computing systems - CHI '04*, page 885, New York, New York, USA, apr 2004. ACM Press. ISBN 1581137036. doi: 10.1145/985921.985947. 3.1.2
- G Geiger and N Alber. The reactable: A collaborative musical instrument for playing and understanding music. *Her&Mus: heritage & ...*, 2010. 3.5.4
- James J. Gibson. The theory of affordances. In R. E. Shaw & J. Bransford, editor, *Perceiving, Acting, and Knowing*. Lawrence Erlbaum Associates, Hilldale, USA, 1977. 2.1
- Emilia Gómez, Maarten Grachten, Alan Hanjalic, Jordi Janer, Sergi Jorda, Carles F. Julià, Cynthia Liem, Agustin Martorell, Markus Schedl, and Gerhard Widmer. Phenix: Performances as highly enriched and interactive concert experiences. *Open access*, 2013. 5.9.3
- Masataka Goto and Takayuki Goto. Musicream: New music playback interface for streaming, sticking, sorting, and recalling musical pieces. In *ISMIR'05*, pages 404–411, 2005. 3.6.1, 3.22
- Jefferson Y. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proceedings of the 18th annual ACM symposium on User interface software and technology - UIST '05*, page 115, New York, New York, USA, oct 2005. ACM Press. ISBN 1595932712. doi: 10.1145/1095034.1095054. 2.4.1
- Jefferson Y. Han. Multi-touch interaction wall. *ACM SIGGRAPH 2006 Emerging technologies on - SIGGRAPH '06*, page 25, 2006. doi: 10.1145/1179133.1179159. 3.6
- Thomas E. TE Hansen, JP Juan Pablo Hourcade, Mathieu Virbel, Sharath Patali, and

Bibliography

- Tiago Serra. Pymt: a post-wimp multi-touch user interface toolkit. *Proceedings of the ACM ...*, 2009. 5.5.1
- Don Hatfield. The coming world of "what you see is what you get". In *CHI'81*, volume 13, page 138, New York, NY, USA, jan 1981. ACM. ISBN 0-89791-064-8. doi: 10.1145/1015579.810978. 2.2.2
- Ken Hinckley, Randy Pausch, John C. Goble, and Neal F. Kassell. Passive real-world interface props for neurosurgical visualization. *Conference on Human Factors in Computing Systems*, page 452, 1994. 2.3
- Ken Hinckley, Randy Pausch, Dennis Proffitt, and Neal F. Kassell. Two-handed virtual manipulation. *ACM Transactions on Computer-Human Interaction*, 5(3):260–302, sep 1998. ISSN 10730516. doi: 10.1145/292834.292849. 2.3
- Stephen Hitchner, J. Murdoch, and G. Tzanetakis. Music browsing using a tabletop display. In *8th International Conference on Music Information Retrieval*. Citeseer, 2007. 3.6.1
- Jordan Hochenbaum, Owen Vallis, Dimitri Diakopoulos, Jim Murphy, and Ajay Kapuy. Designing expressive musical interfaces for tabletop surfaces. In *Proceedings of the 2010 conference on New interfaces for musical expression*, pages 315–318. Sydney, Australia, 2010. 4.2
- Steve Hodges, Shahram Izadi, Alex Butler, Alban Rustemi, and Bill Buxton. Thinsight: versatile multi-touch sensing for thin form-factor displays. In *Proceedings of the 20th annual ACM symposium on User interface software and technology - UIST '07*, page 259, New York, New York, USA, oct 2007. ACM Press. ISBN 9781595936792. doi: 10.1145/1294211.1294258. 2.5
- Lars Erik Holmquist, Johan Redström, and Peter Ljungstrand. Token-based acces to digital information. *Lecture Notes In Computer Science*, 1707:234–245, 1999. doi: 10.1007/3-540-48157-5_22. 3.1.3
- Michael S Horn and Robert J K Jacob. Designing tangible programming languages for classroom use. *Proceedings of the 1st international conference on Tangible and embedded interaction TEI 07*, page 159, 2007. doi: 10.1145/1226969.1227003. 3.5.2
- Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. Jacob. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, page 975, New York, New York, USA, apr 2009. ACM Press. ISBN 9781605582467. doi: 10.1145/1518701.1518851. 3.5.2
- Eva Hornecker and Jacob Buur. Getting a grip on tangible interaction: a framework

- on physical space and social interaction. *Proceedings of the SIGCHI conference on Human*, 2006. 2.5, 3.1.2, 5.2
- Lode Hoste and Beat Signer. Criteria, challenges and opportunities for gesture programming languages. In *EGMI 2014*, volume i, 2014. 5.5.1, 5.5.1
- Steve Hotelling, Joshua A. Strickon, Brian Q. Huppi, Imran Chaudhri, Greg Christie, Bas Ording, Duncan Robert Kerr, and Jonathan P. Ive. Gestures for touch sensitive input devices, jul 2004. 5.4.3
- Jörn Hurtienne, Christian Stößel, and Katharina Weber. Sad is heavy and happy is light. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction - TEI '09*, page 61, New York, New York, USA, feb 2009. ACM Press. ISBN 9781605584935. doi: 10.1145/1517664.1517686. 3.1.3
- Peter Hutterer and Bruce H. Thomas. Groupware support in the windowing system. In *AUIC2007*, 2007. 2.2.4, 5.3, 5.4.1
- Peter Hutterer and Bruce H. Thomas. Enabling co-located ad-hoc collaboration on shared displays. In *9th Australasian User Interface Conference (AUIC2008)*, pages 43–50, Wollongong, NSW, Australia, jan 2008. Australian Computer Society, Inc. ISBN 978-1-920682-57-6. 5.4.1
- Tommi Ilmonen. Tracking conductor of an orchestra using artificial neural networks. *Master's thesis, Helsinki University of Technology*, 1999. 5.9.3
- Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. *Conference on Human Factors in Computing Systems*, page 234, 1997. 2.4, 3.1
- Shahram Izadi, Harry Brignull, Tom Rodden, Yvonne Rogers, and Mia Underwood. Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media. In *Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 159–168, 2003. ISBN 1581136366. 2.2.4, 5.3
- Sergi Jordà. Improvising with computers: A personal survey (1989-2001). *Journal of New Music Research*, 31(1):1–10, mar 2002. ISSN 0929-8215. doi: 10.1076/jnmr.31.1.1.8105. 2.3
- Sergi Jordà. Sonigraphical instruments: from finol to the reactable. In *Proceedings of the 2003 conference on New interfaces for musical expression*, NIME '03, pages 70–76, Montreal, Canada, 2003. National University of Singapore. 3.1.3
- Sergi Jordà. On stage: the reactable and other musical tangibles go real. *International Journal of Arts and Technology*, 1(3/4):268–287, 2008. ISSN 17548853. 3.1.3, 3.4, 3.6, 5.3, 5.4.2

Bibliography

- Sergi Jordà, Martin Kaltenbrunner, Günter Geiger, and Ross Bencina. The reactable*. In *Proceedings of the International Computer Music Conference (ICMC 2005)*, Barcelona, Spain, pages 579–582, 2005. 3.2, 3.4, 5.4.2
- Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reactable: exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 139–146. ACM, 2007. 3.1.2, 3.1.2
- Sergi Jordà, Carles F. Julià, and Daniel Gallardo. Interactive surfaces and tangibles. *XRDS: Crossroads, The ACM Magazine for Students*, 16(4):21–28, 2010. ISSN 1528-4972. 5.3
- Carles F. Julià and Daniel Gallardo. Tdesktop : Disseny i implementació d’un sistema gràfic tangible, 2007. 5.3, 5.4.3
- Carles F. Julià and Sergi Jordà. Songexplorer: A tabletop application for exploring large collections of songs. In *ISMIR 2009*, 2009. 3.6
- Carles F. Julià, Daniel Gallardo, and Sergi Jordà. Mtcf: A framework for designing and coding musical tabletop applications directly in pure data. In *New Interfaces for Musical Expression*, 2011. 4.2
- Carles F. Julià, Nicolas Earnshaw, and Sergi Jorda. Gestureagents: an agent-based framework for concurrent multi-task multi-user interaction. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction*, pages 207–214. ACM, 2013. 5.4.4
- Paul Kabbash, William A. S. Buxton, and Abigail Sellen. Two-handed input in a compound task. In *CHI’94*, 1994. 2.3, 2.4
- Martin Kaltenbrunner, Günter Geiger, and Sergi Jordà. Dynamic patches for live musical performance, 2004. 4.2.1
- Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. Tuio - a protocol for table based tangible user interfaces. In *Proceedings of the 6th International Workshop on Gesture in HumanComputer Interaction and Simulation GW 2005*, pages 1–5, 2005. 3.4
- Dietrich Kammer, Georg Freitag, Mandy Keck, and Markus Wacker. Taxonomy and overview of multi-touch frameworks: Architecture, scope and features. *Patterns for Multi-Touch*, 2010a. 5.5.3, 5.6.1, 5.6.2
- Dietrich Kammer, Jan Wojdziak, Mandy Keck, Rainer Groh, and Severin Taranko. Towards a formalization of multi-touch gestures. *ACM International Conference on*

- Interactive Tabletops and Surfaces - ITS '10*, page 49, 2010b. doi: 10.1145/1936652.1936662. 5.4.4, 5.5.1
- K Karplus and A Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983. ISSN 0148-9267. 4.2.1
- Loïc Kessous and Daniel Arfib. Bimanuality in alternate musical instruments. pages 140–145, may 2003. 2.3
- Shahedul Huq Khandkar and Frank Maurer. A domain specific language to define gestures for multi-touch applications. In *Proceedings of the 10th Workshop on Domain-Specific Modeling - DSM '10*, page 1, New York, New York, USA, oct 2010. ACM Press. ISBN 9781450305495. doi: 10.1145/2060329.2060339. 5.5.1
- Henna Kim and Sara Snow. Collaboration on a large-scale, multi-touch display: asynchronous interaction and multiple-input use. In *CSCW'13*, pages 165–168, 2013. ISBN 9781450313322. 5.3
- J Kim, J Park, H K Kim, and C Lee. Hci (human computer interaction) using multi-touch tabletop display. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2007. PacRim 2007*, pages 391–394, 2007. 3.6.3
- Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. Proton: Multi-touch gestures as regular expressions. In *CHI 2012*, 2012. ISBN 9781450310154. 5.4.4, 5.5.1, 5.5.1, 5.5.3
- David Kirsh and Paul Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science*, 18(4):513–549, oct 1994. ISSN 03640213. doi: 10.1207/s15516709cog1804_1. 3.1.3
- Peter Knees, Markus Schedl, Tim Pohle, and Gerhard Widmer. Exploring music collections in virtual landscapes. *MultiMedia, IEEE*, 14(3):46–54, 2007. ISSN 1070-986X. 3.6.1
- T Kohonen. *Self-Organizing Maps*. Springer, 2001. 3.6.1, 3.6.3
- Paul Kolesnik and Marcelo Wanderley. Recognition, analysis and performance with expressive conducting gestures. In *ICMC*, 2004. 5.9.3
- Anu Konttinen. *Conducting Gestures: Institutional and Educational Construction of Conductorship in Finland, 1973-1993*. PhD thesis, 2008. 5.9.3
- Myron W. Krueger, Thomas Gionfriddo, and Katrin Hinrichsen. Videoplace an artificial reality. In *CHI'85*, volume 16, pages 35–40. ACM, apr 1985. ISBN 0-89791-149-0. doi: 10.1145/1165385.317463. 2.3, 5.4.1

Bibliography

- Uwe Laufs, Christopher Ruff, and Jan Zibuschka. Mt4j-a cross-platform multi-touch development framework. *arXiv preprint arXiv:1012.0467*, 2010. 5.5.1, 5.5.3
- C Laurier, O Meyers, J Serrà, M Blech, and P Herrera. Music mood annotator design and integration. In *7th International Workshop on Content-Based Multimedia Indexing*, Chania, Crete, Greece, 2009. 3.6.2, 3.6.3
- Miguel Lechón. *SongExplorer Studies*. PhD thesis, Universitat Pompeu Fabra, 2010. 3.6.5
- Henry F. Ledgard. Ten mini-languages: A study of topical issues in programming languages. *ACM Computing Surveys*, 3(3):115–146, sep 1971. ISSN 03600300. doi: 10.1145/356589.356592. 3.5
- Hyun-Jean Lee, Hyungsin Kim, Gaurav Gupta, and Ali Mazalek. Wiiarts: Creating collaborative art experience with wiireMOTE interaction. In *Proceedings of the 2nd international conference on Tangible and embedded interaction - TEI '08*, page 33, New York, New York, USA, feb 2008. ACM Press. ISBN 9781605580043. doi: 10.1145/1347390.1347400. 2.3
- SK Lee, William A. S. Buxton, and K C Smith. A multi-touch three dimensional touch-sensitive tablet. *ACM SIGCHI Bulletin*, 16(4):21–25, apr 1985. ISSN 07366906. doi: 10.1145/1165385.317461. 2.3
- S. Leitich and M. Topf. Globe of music: Music library visualization using geosom. pages 167–170, 2007. 3.6.1, 3.6.3
- Russell Mackenzie, Kirstie Hawkey, Kellogg S Booth, Zhangbo Liu, Presley Perswain, and Sukhveer S Dhillon. Lacome: a multi-user collaboration system for shared large displays. In *CSCW'12*, pages 267–268, 2012. ISBN 9781450310512. 5.3, 5.4.1
- Milena Markova. *TurTan Studies: Evaluation of the Impact of Tangible Interfaces on Learning*. PhD thesis, Universitat Pompeu Fabra, 2010. 3.5.5
- Stefan Marr, Thierry Renaux, Lode Hoste, and Wolfgang De Meuter. Parallel gesture recognition with soft real-time guarantees. *Science of Computer Programming*, feb 2014. ISSN 01676423. doi: 10.1016/j.scico.2014.02.012. 5.11.5
- Paul Marshall. Do tangible interfaces enhance learning? In *Proceedings of the 1st international conference on Tangible and embedded interaction - TEI '07*, pages 163–170, New York, New York, USA, 2007. ACM Press. ISBN 9781595936196. doi: 10.1145/1226969.1227004. 3.3, 3.5.1
- Paul Marshall, Yvonne Rogers, and Eva Hornecker. Are tangible interfaces really any better than other kinds of interfaces? In *CHI'07 workshop on Tangible User Interfaces in Context and Theory*, San Jose, California, USA, 2007. 3.1, 3.1.2

- Maria Montessori. *The montessori method*. Frederick A. Stokes Co., 1912. 3.5.1
- Meredith Ringel Morris, A.J. Bernheim Brush, and Brian R. Meyers. A field study of knowledge workers' use of interactive horizontal displays. In *2008 3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems*, pages 105–112. IEEE, oct 2008. ISBN 978-1-4244-2897-7. doi: 10.1109/TABLETOP.2008.4660192. 2.4.1
- Christian Muller-Tomfelde, Anja Wessels, and Claudia Schremmer. Tilted tabletops: In between horizontal and vertical workspaces. In *2008 3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems*, pages 49–56. IEEE, oct 2008. ISBN 978-1-4244-2897-7. doi: 10.1109/TABLETOP.2008.4660183. 2.4.1
- J Noble. *Programming Interactivity: A Designer's Guide to Processing, Arduino, and OpenFrameworks*. O'Reilly Media, 2009. ISBN 0596154143. 4.1.2
- Donald Norman. *The psychology of everyday things*. Basic Books, New York, NY, USA, 1988. 2.1
- Judith S. Olson, Gary M. Olson, Marianne Storrøsten, and Mark Carter. How a group-editor changes the character of a design meeting as well as its outcome. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work - CSCW '92*, pages 91–98, New York, New York, USA, dec 1992. ACM Press. ISBN 0897915429. doi: 10.1145/143457.143466. 3.1.2
- A Pabst and R Walk. Augmenting a rugged standard dj turntable with a tangible interface for music browsing and playback manipulation. In *Intelligent Environments, 2007. IE 07. 3rd IET International Conference on*, pages 533–535, 2007. 3.6.3
- E Pampalk. Islands of music analysis, organization, and visualization of music archives. *Journal of the Austrian Soc. for Artificial Intelligence*, 22(4):20–23, 2003. 3.6.1, 3.21
- E Pampalk, S Dixon, and G Widmer. Exploring music collections by browsing different views. *Computer Music Journal*, 28(2):49–62, 2004. 3.6.1
- Seymour Papert. *The children's machine: Rethinking school in the age of the computer*. Basic Books, 1993. ISBN 0465010636. 3.5.1
- Seymour Papert and Idit Harel. Situating constructionism. *Constructionism*, pages 1–11, 1991. 3.5.1
- Simon Penny, Jeffrey Smith, and Andre Bernhardt. Traces: Wireless full body tracking in the cave. In *ICAT'99*, 1999. 2.3
- Piaget and Jean. *The Construction Of Reality In The Child*. Routledge, 1999. ISBN 1136316949. 3.5.1

Bibliography

- Jean Piaget. *The origins of intelligence in the child*. Routledge and Kegan Paul, London, 1953. 3.5.1
- Andrei Popescu-Belis, Steve Renals, and Hervé Bourlard, editors. *Machine Learning for Multimodal Interaction*, volume 4892 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78154-7. doi: 10.1007/978-3-540-78155-4. 5.5.1
- Timothy Poston and Luis Serra. The virtual workbench: dextrous vr. pages 111–121, aug 1994. 2.3
- M Puckette. Pure data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996. 4.2.1
- H.S. Raffle, A.J. Parkes, and H. Ishii. Topobo: a constructive assembly system with kinetic memory. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 647–654. ACM, 2004. 2.4
- M. Resnick, F. Martin, R. Berg, R. Borovoy, V. Colella, K. Kramer, and B. Silverman. Digital manipulatives: new toys to think with. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 281–287. ACM Press/Addison-Wesley Publishing Co., 1998. 3.5.1
- Mitchel Resnick. Technologies for lifelong kindergarten. *Educational Technology Research and Development*, 46(4):43–55, 1998. ISSN 1042-1629. doi: 10.1007/BF02299672. 3.5.1
- Yvonne Rogers and Tom Rodden. Configuring spaces and surfaces to support collaborative interactions. In *Public and Situated Displays*, pages 45–79. Kluwer Publishers, 2004. 3.1.2
- Yvonne Rogers, Youn-kyung Lim, William Hazlewood, and Paul Marshall. Equal opportunities: Do shareable interfaces promote more group participation than single user displays? *Human-Computer Interaction*, 24(1):79–116, jan 2009. ISSN 0737-0024. doi: 10.1080/07370020902739379. 5.2
- David Sachs. Sensor fusion on android devices: A revolution in motion processing, 2010. 2.3
- E. Sachs, A. Roberts, and D. Stoops. 3-draw: a tool for designing 3d shapes. *IEEE Computer Graphics and Applications*, 11(6):18–26, nov 1991. ISSN 0272-1716. doi: 10.1109/38.103389. 2.3
- Lori Scarlatos. An application of tangible interfaces in collaborative learning environments. In *ACM SIGGRAPH 2002 conference abstracts and applications on - SIGGRAPH '02*, page 125, New York, New York, USA, jul 2002. ACM Press. ISBN 1581135254. doi: 10.1145/1242073.1242141. 3.5.1, 3.5.5

- Robert W. Scheifler and Jim Gettys. The x window system. *Software: Practice and Experience*, 20(S2):S5–S34, oct 1990. ISSN 00380644. doi: 10.1002/spe.4380201403. 5.4.1
- Thomas Schlömer, Benjamin Poppinga, Niels Henze, and Susanne Boll. Gesture recognition with a wii controller. *Proceedings of the 2nd international conference on Tangible and embedded interaction - TEI '08*, page 11, 2008. doi: 10.1145/1347390.1347395. 2.3, 5.5.1, 5.5.1
- Kjeld Schmidt and Liam Bannon. Taking cscw seriously. *Computer Supported Cooperative Work (CSCW)*, 1(1-2):7–40, mar 1992. ISSN 0925-9724. doi: 10.1007/BF00752449. 5.2
- Christophe Scholliers, Lode Hoste, Beat Signer, and Wolfgang De Meuter. Midas: a declarative multi-touch interaction framework. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, 2011. 5.4.4, 5.5.1, 5.5.3
- Johannes Schöning, Peter Brandl, Florian Daiber, Florian Echtler, Otmar Hilliges, Jonathan Hook, Markus Löchtefeld, Nima Motamedi, Laurence Muller, Patrick Olivier, Tim Roth, and Ulrich von Zadow. Multi-touch surfaces: A technical guide. Technical report, 2008. 2.4.1, 3.2, 3.4
- J Schwarz and S Hudson. A framework for robust and flexible handling of inputs with uncertainty. *Proceedings of the 23rd annual ACM symposium on User interface software and technology*, 2010. 5.5.3
- E Schweikardt and MD Gross. roblocks: a robotic construction kit for mathematics and science education. ... of the 8th international conference on ..., 2006. 2.4
- Stacey D. Scott, Karen D. Grant, and Regan L. Mandryk. System guidelines for co-located, collaborative work on a tabletop display. *ECSCW 2003*, 2003. 5.3
- Orit Shaer and Eva Hornecker. Tangible user interfaces: Past, present, and future directions. *Foundations and Trends in Human-Computer Interaction*, 3(1-2):1–137, jan 2010. ISSN 1551-3955. doi: 10.1561/11000000026. 2.1, 3.1.1, 3.1.3, 3.5, 5.2
- C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3, jan 2001. ISSN 15591662. doi: 10.1145/584091.584093. 3.1.2
- Ehud Sharlin, Benjamin Watson, Yoshifumi Kitamura, Fumio Kishino, and Yuichi Itoh. On tangible user interfaces, humans and spatiality. *Personal and Ubiquitous Computing*, 8(5):338–346, jul 2004. ISSN 1617-4909. doi: 10.1007/s00779-004-0296-5. 5.4.1
- Chia Shen, Neal Lesh, and Frédéric Vernier. Personal digital historian. *interactions*, 10(2):15, mar 2003. ISSN 10725520. doi: 10.1145/637848.637856. 3.1.2, 3.1.2

Bibliography

- B Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983. ISSN 00189162. doi: 10.1109/MC.1983.1654471. 2.2.2
- Jamie Shotton, Toby Sharp, Alex Kipman, Andrew Fitzgibbon, Mark Finocchio, Andrew Blake, Mat Cook, and Richard Moore. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, 56(1):116, jan 2013. ISSN 00010782. doi: 10.1145/2398356.2398381. 5.5.1
- Danae Stanton, Tony Pridmore, Victor Bayon, Helen Neale, Ahmed Ghali, Steve Benford, Sue Cobb, Rob Ingram, Claire O'Malley, and John Wilson. Classroom collaboration in the design of tangible interfaces for storytelling. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '01*, pages 482–489, New York, New York, USA, mar 2001. ACM Press. ISBN 1581133278. doi: 10.1145/365024.365322. 5.2
- I Stavness, J Gluck, L Vilhan, and S Fels. The musictable: A map-based ubiquitous system for social interaction with a digital music collection. *Entertainment Computing-ICEC 2005*, pages 291–302, 2005. 3.6.1
- M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. Wysiwiw revised: early experiences with multiuser interfaces. *ACM Transactions on Information Systems*, 5(2):147–167, apr 1987. ISSN 10468188. doi: 10.1145/27636.28056. 3.1.2
- Anselm Strauss. Work and the division of labor. *The Sociological Quarterly*, 26(1):1–19, mar 1985. ISSN 0038-0253. doi: 10.1111/j.1533-8525.1985.tb00212.x. 5.2
- Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4):531–582, dec 2006. ISSN 10730516. doi: 10.1145/1188816.1188821. 3.1.2
- Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop on - DAC '64*, pages 6.329–6.346, New York, New York, USA, jan 1964. ACM Press. doi: 10.1145/800265.810742. 2.3
- Ivan E. Sutherland. A head-mounted three dimensional display. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*, page 757, New York, New York, USA, dec 1968. ACM Press. doi: 10.1145/1476589.1476686. 2.3
- H Suzuki and Hiroshi Kato. Algoblock: a tangible programming language, a tool for collaborative learning. *Proceedings of 4th European Logo Conference*, 1993. 2.4, 3.5.2
- Michael Terry. Task blocks: tangible interfaces for creative exploration. In *CHI '01*

- extended abstracts on Human factors in computing systems - CHI '01*, page 463, New York, New York, USA, mar 2001. ACM Press. ISBN 1581133405. doi: 10.1145/634067.634334. 3.5.2
- Philip Tuddenham, Ian Davies, and Peter Robinson. Websurface. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces - ITS '09*, page 181, New York, New York, USA, nov 2009. ACM Press. ISBN 9781605587332. doi: 10.1145/1731903.1731938. 5.3, 5.4.1
- Brygg Ullmer and Hiroshi Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3):915–931, 2000. ISSN 0018-8670. doi: 10.1147/sj.393.0915. 2.2, 2.4
- Brygg Ullmer, Hiroshi Ishii, and Dylan Glas. mediablocks: physical containers, transports, and controls for online media. *International Conference on Computer Graphics and Interactive Techniques*, page 379, 1998. 3.5.4
- Brygg Ullmer, Hiroshi Ishii, and Robert J. K. Jacob. Token+constraint systems for tangible interaction with digital information. *ACM Transactions on Computer-Human Interaction*, 12(1):81–118, mar 2005. ISSN 10730516. doi: 10.1145/1057237.1057242. 3.1.3
- John Underkoffler and Hiroshi Ishii. Urp: a luminous-tangible workbench for urban planning and design. *Conference on Human Factors in Computing Systems*, page 386, 1999. 2.4, 2.3, 3.1.3
- Himanshu Verma, Flaviu Roman, Silvia Magrelli, Patrick Jermann, and Pierre Dillenbourg. Complementarity of input devices to achieve knowledge sharing in meetings. In *Proceedings of the 2013 conference on Computer supported cooperative work - CSCW '13*, page 701, New York, New York, USA, 2013. ACM Press. ISBN 9781450313315. doi: 10.1145/2441776.2441855. 5.3
- Frédéric Vernier, Neal Lesh, and Chia Shen. Visualization techniques for circular tabletop interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces - AVI '02*, page 257, New York, New York, USA, may 2002. ACM Press. ISBN 1581135378. doi: 10.1145/1556262.1556305. 5.4.2
- Mathieu Virbel, Thomas Hansen, and Oleksandr Lobunets. Kivy - a framework for rapid creation of innovative user interfaces. In Marc Eibl, Maximilian AND Ritter, editor, *Workshop-Proceedings der Tagung Mensch & Computer 2011*, pages 69–73, Chemnitz, 2011. Universitätsverlag Chemnitz. ISBN 978-3-941003-38-5. 5.5.1
- Luc Vlaming, Jasper Smit, and Tobias Isenberg. Presenting using two-handed interaction in open space. In *2008 3rd IEEE International Workshop on Horizontal Interactive*

Bibliography

- Human Computer Systems*, pages 29–32. IEEE, oct 2008. ISBN 978-1-4244-2897-7. doi: 10.1109/TABLETOP.2008.4660180. 2.3
- Vasiliki Vouloutsi, Klaudia Grechuta, Stéphane Lallée, and Paul F M J Verschure. The influence of behavioral complexity on robot perception. In *Biomimetic and Biohybrid Systems*, pages 332–343. Springer, 2014. ISBN 3319094343. 4.1.5
- Michel Waisvisz. *The Hands: A Set of Remote MIDI-Controllers*. Ann Arbor, MI: MPublishing, University of Michigan Library, 1985. 2.3
- I Ivo Weevers, RJW Wouter Sluis, van CHGJ Claudia Schijndel, S Siska Fitrianie, L Lyuba Kolos-Mazuryk, and JBOS Jean-Bernard Martens. Read-it: A multi-modal tangible interface for children who learn to read. In *Entertainment Computing - ICEC 2004*, chapter Springer, pages 226–234. Springer Press, 2004. 3.5.1, 3.5.5
- Mark Weiser and Jonh Seely Brown. Designing calm technology. *PowerGrid Journal*, pages 1–5, 1996. 2.4
- Joel West and Michael Mace. Browsing as the killer app: Explaining the rapid success of apple’s iphone. *Telecommunications Policy*, 34(5-6):270–286, jun 2010. ISSN 03085961. doi: 10.1016/j.telpol.2009.12.002. 2.1, 2.2.3
- Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. Gestures without libraries, toolkits or training: a \$1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology - UIST ’07*, page 159, New York, New York, USA, oct 2007. ACM Press. ISBN 9781595936792. doi: 10.1145/1294211.1294238. 5.4.4, 5.5.1, 5.5.1
- Anna Xambó, Eva Hornecker, Paul Marshall, Sergi Jordà, Chris Dobbyn, and Robin Laney. Let’s jam the reactable. *ACM Transactions on Computer-Human Interaction*, 20(6):1–34, dec 2013. ISSN 10730516. doi: 10.1145/2530541. 5.3
- Oren Zuckerman, Saeed Arida, and Mitchel Resnick. Extending tangible interfaces for education. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI ’05*, page 859, New York, New York, USA, apr 2005. ACM Press. ISBN 1581139985. doi: 10.1145/1054972.1055093. 2.4, 3.5.1

A List of Publications

References

- [1] Daniel Gallardo, Carles F. Julià, and Sergi Jordà. Turtan: A tangible programming language for creative exploration. In *3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems, 2008. TABLETOP 2008*, pages 89–92. Ieee, IEEE, oct 2008.
- [2] Daniel Gallardo, Carles F. Julià, and Sergi Jordà. Using mtcf for live prototyping on tablet and tangible tabletop devices. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction*, pages 443–446. ACM, 2013.
- [3] Emilia Gómez, Maarten Grachten, Alan Hanjalic, Jordi Janer, Sergi Jorda, Carles F. Julià, Cynthia Liem, Agustin Martorell, Markus Schedl, and Gerhard Widmer. Phenix: Performances as highly enriched and interactive concert experiences. *Open access*, 2013.
- [4] Sergi Jordà, Seth E. Hunter, Pol Pla i Conesa, Daniel Gallardo, Daniel Leithinger, Henry Kaufman, Carles F. Julià, and Martin Kaltenbrunner. Development strategies for tangible interaction on horizontal surfaces. In *Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction - TEI '10*, pages 369–372, New York, New York, USA, 2010. ACM Press.
- [5] Sergi Jordà, Carles F. Julià, and Daniel Gallardo. Interactive surfaces and tangibles. *XRDS: Crossroads, The ACM Magazine for Students*, 16(4):21–28, 2010.
- [6] Carles F. Julià. Towards concurrent multi-tasking in shareable interfaces (in revision). *Journal of Computer Supported Collaborative Work*, (Special Issue "Collaboration meets Interactive Surfaces - Walls, Tables, Tablets and Phones"), 2015.
- [7] Carles F. Julià, Nicolas Earnshaw, and Sergi Jorda. Gestureagents: an agent-based framework for concurrent multi-task multi-user interaction. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction*, pages 207–214. ACM, 2013.
- [8] Carles F. Julià, Daniel Gallardo, and Sergi Jordà. Turtan: Un lenguaje de programación tangible para el aprendizaje. In AIPO, editor, *Interacción 2009*, Barcelona, 2009.
- [9] Carles F. Julià, Daniel Gallardo, and Sergi Jordà. Mtcf: A framework for designing

and coding musical tabletop applications directly in pure data. In *New Interfaces for Musical Expression*, 2011.

- [10] Carles F. Julià and Sergi Jordà. Songexplorer: A tabletop application for exploring large collections of songs. In *ISMIR 2009*, 2009.

B Tabletop applications in TEI, ITS, and tabletop conferences

To assess the current situation of tabletop applications in research, we identified the ones presented in conferences that usually focus on tangible and tabletop interaction: TANGIBLE AND EMBEDDED INTERACTION (TEI, 2007-2013), TABLETOP (tabletop, 2008), and INTERACTIVE TABLES AND SURFACES (ITS, 2009-2013). Only papers presenting an application (and not only a test case for a new interaction technique or detection system) are taken into account.

Conference	Authors	Application type	Device or Technology
ITS09	Helmes et al.	Interactive storytelling	Microsoft surface
ITS09	Battocchi et al.	puzzle game for collaboration/autist	DiamondTouch
ITS09	Seifried et al.	media controller	DiamondTouch
ITS10	Sultanum et al.	topological/reservoir visualization	Microsoft surface
ITS10	Dang and André	game implicit interaction	Microsoft surface
ITS10	Selim and Maurer	control center/utility companies	Microsoft surface
ITS10	Correia et al.	museum/collection browsing	CCV
ITS11	Conardi et al.	Electronics simulator	Microsoft surface
ITS11	Freeman and Balakrishnan	interaction design/ scripting	Microsoft surface
ITS11	Mikulecky et al.	Cloth-based map navigation	SMART
ITS11	Chaboissier et al.	multiuser game	DiamondTouch
ITS11	Kim et al.	think aloud/education	CCV
ITS12	Chang et al.	science divulgation	PixelSense
ITS12	Schneider et al.	Simulation of Climate Change	Microsoft surface
ITS12	Valdes et al.	scientific learning/children	Microsoft surface
ITS12	Chua et al.	Simulation of Evolutionary Processes	PixelSense
ITS12	Shneider	learning/collaboration/education	Microsoft surface
ITS12	Tozser et al.	collision reconstruction/police investigation	Microsoft surface
ITS12	Bertrand et al.	education/neuroscience	reactIVision
ITS12	Shneider et al.	education/mathematics	reactIVision
ITS13	Domova et al.	control/command/coordination	Microsoft surface
ITS13	Matulic and Norrie	pen and touch document editing	DiamondTouch
ITS13	Augstein et al.	Neuro-rehabilitation	PixelSense
ITS13	Lee and Lee	control center/nuclear plant	PixelSense

Table B.1: Tabletop application papers appeared in TEI, ITS, and TABLETOP conferences

B Tabletop applications in TEI, ITS, and tabletop conferences

Conference	Authors	Application type	Device or Technology
ITS13	Döweling et al	map/crisis management	PixelSense
ITS13	Wozniak et al.	Maritime operations	PixelSense
ITS13	Deb	task assignment	PixelSense
ITS13	Seyed et al.	oil,gas exploration	PixelSense
tabletop08	Wang and Maurer	Agile meetings	SMART
tabletop08	Rick and Rogers	education	DiamondTouch
tabletop08	Jiang et al.	Multi-surface sharing (like dynamo)	DiamondTouch
tabletop08	Gallardo et al.	education/programming	reactIVision
tabletop08	Rick and Rogers	education	reactIVision
TEI07	Jorda et al.	musical instrument	reactIVision
TEI08	Jo	sound performance instrument	DiamondTouch
TEI08	Seifried et al.	document organization (virtual/real)	DiamondTouch
TEI08	Couture et al.	science/geoscience	reactIVision
TEI09	Oppl and Stary	concept mapping	reactIVision
TEI09	Pedersen and Hornbaek	musical instrument	reactIVision
TEI09	Bartindale et al.	media production	reactIVision
TEI09	Mazalek et al.	storytelling	reactIVision
TEI10	Gallardo and Jordà	music player/organizer	reactIVision
TEI10	Allison et al.	visual game tessellation	reactIVision
TEI11	Xambó et al.	musical instrument	reactIVision
TEI11	Clifton et al.	sketching	reactIVision
TEI11	Canton	telepresence	CCV
TEI12	Salehi et al.	think aloud/education	CCV
TEI13	Dang and André	game	Microsoft surface
TEI13	Xu et al.	science divulgation	Microsoft surface
TEI13	Marco et al.	Board-game creator	reactIVision
TEI13	Oh et al.	education/programming	reactIVision

Table B.1: Tabletop application papers appeared in TEI, ITS, and TABLETOP conferences

C TSI Applications

Year	Name	Game	Music	Tool	Demo	Fingers	Objects	Player ident. obj.
2009	TUIGoal	✓				✓		
2009	PlanetWar - NanoWar	✓				✓	✓	
2009	Punk-o-table		✓			✓	✓	
2009	Puckr	✓	✓			✓	✓	✓
2009	Oracle's maze	✓					✓	
2009	TanQue			✓		✓	✓	
2009	ReaCTAnoid	✓				✓	✓	
2009	EBR - Beat the agents	✓	✓				✓	✓
2009	effectable		✓			✓	✓	
2010	80-table	✓					✓	✓
2010	Smash Table	✓					✓	✓
2010	projectwalk				✓	✓		
2010	autobahn	✓				✓	✓	✓
2010	blockout	✓					✓	
2010	el meu gat favi	✓		✓		✓	✓	
2010	audioplayer		✓	✓		✓	✓	
2011	Tower Defense	✓				✓	✓	✓
2011	Pulse Cubes		✓			✓	✓	
2011	reactanks	✓				✓	✓	✓
2011	logic gate simulator			✓		✓	✓	
2011	formigues	✓				✓	✓	✓
2011	daus mentiders	✓				✓	✓	
2011	poquer	✓				✓	✓	
2011	Rubik Musical		✓			✓	✓	
2011	tablewars	✓					✓	✓
2011	menu				✓	✓	✓	
2012	Ranas	✓					✓	
2012	Naus	✓				✓	✓	✓
2012	Invasors del desktop	✓				✓	✓	
2012	ninjafruit	✓					✓	✓
2012	Scrabble	✓					✓	✓

C TSI Applications

Year	Name	Game	Music	Tool	Demo	Fingers	Objects	Player ident. obj.
2012	pixels-laser	✓					✓	
2012	worms	✓				✓	✓	✓
2012	reactcurling	✓				✓	✓	
2012	interiorisme			✓		✓	✓	