T

D

AND D SPI

UMR-CRISAL

//

T R

A

'

P **M. L D**

R **M. S D-C**

M. Yeah L

AND **M. S C**

M. E P

D **M. S H**

P, U

P, U

M

FOOT

D

D

L

G A

(HDR), T PT

N 'A C

S

L

In memory of
Janet, to her kindness,
and his love of letters

Thanks

The document you have before you is the culmination of four years of research.
Four years that put my nerves to the test, alternating between painful learning, periods of deep melancholy, and moments of epiphany. Now that the result is here, I hope it will satisfy you. Programming interactive graphical systems (desktop interfaces, smartphone applications, video games) is a field that has always been particularly opaque to me, and it was this frustration that pushed me to dig deep into a thesis. So I wanted to make this manuscript as useful as possible, by trying to clarify everything I had learned about the field, and sometimes giving my opinion on what could be added to it.

This work would certainly not have been possible without the help and support of many people, whom I would like to take the time to mention. I would first like to thank my thesis director, Stéphane Huot, who showed great patience towards me. When I doubted everything, questioned his advice, and did what I wanted, he was unconditionally supportive and a great listener. I really enjoyed our discussions, where he often saw “further” and made key comments that would advance my work. The innovations of this thesis were thus truly co-constructed, through iterations and fruitful exchanges. I thank Stéphane for guiding me where I wanted to go, rather than where he wanted me to go.

I would then like to thank my rapporteurs, Sophie Dupuy-Chessa and Éric Lecolinet, for taking the time to reread my manuscript, and all the members of the jury (including Stéphane Conversy, Laurence Duchien and Emmanuel Pietriga) for agreeing to evaluate my work. I am honored to have been able to collect the comments and advice of those I consider to be the best French experts on my thesis subject!

The four years in Team Loki (ex-Mjolnir) were particularly pleasant, thanks to the kindness of all the members (past and present). Thanks to my senior doctors, Jonathan, Alix, Justin, Amira and Hakim, for their wisdom and advice (administrative, but not only). Thanks to Sébastien and Izzat, among others as teammates during Hash Code 2016. Thanks to Nicole, Axel, Marc, Damien, Raiza, Philippe, Grégoire, Gabriel and Pierrick, for making Starter ***the place to be,*** to complain, eat, play, or even work. Thanks to Julien for welcoming me into his clan, and for a memorable festival. Thank you to all the permanent members of the team, Thomas, Sylvain, Mathieu, Nicolas, Ed, Marcello, Fanny, Géry, for being super accessible, and for all supporting me when you had the opportunity. It may be insignificant to you, but every helping hand has been a huge help to me. And thank you to the team assistants, Karine and Julie, always smiling when I arrived with the most incongruous requests.

During the first two years of the thesis I shared my time with a second team, RMoD, and I would like to thank all the members who supported me in my work in interaction with the field of Software Engineering. Thanks to Damien for introducing me to ECS, and for fueling numerous discussions around original object models. Thanks to Clément, Vincent and Brice,

helped me relieve the pressure with the afterworks. Thank you to Olivier, Marcus, Anne, Christophe and Santiago for your kindness throughout my stay in the team, it was beneficial. Thanks to Stéphane for the relevant criticisms of ECS, which encouraged me to solidify his defense in relation to the object model.

Throughout my thesis I admit to having had an immoderate taste for secondary projects, to the despair of my thesis director. However, these projects have brought me a lot humanly and professionally, and I thank the people who participated in them. Thanks first of all to Yoann Dufresne, Yann Secq and Sophie Tison, who formed the core of the promotion of algorithmic competitions at the university, thanks to which we organized weekly training workshops for almost three years, and who were then supported to carry out teaching functions. Thanks to Lucien Mousin for his fantastic online coding platform which allowed us to boost the algorithmic tutorials (and the training workshops). Thanks to Benjamin Torres also in organizing the Catalysts competition hub. I would also like to thank the members of the bidouille space, a hyperactive community which welcomed me on Saturdays to code and download: Yoann (again), Matthieu Falce, Pierre Marijon, Pierre Pericard, Camille Lihouck and Sophie Kaleba, in particular for the Zoo Machines festival which was a truly creative moment!

Outside of the thesis, I was able to count on an outstanding friend and roommate, Léonard , whom I thank for introducing me to music festivals (... Boom-lay, BOOM!), introducing me to lots of great groups, and for the community of M's who gave me confidence at a turning point in my work .
Thanks to Pierre for the Rock sessions, and his taste for blind test team names, failing to win them (Emile Louise Attaque FTW). Thanks to Sandra and Amédée, Marie and Tupak, Charlotte and Guille, and everyone whose path I crossed during these four years in Lille. Thanks to Aurélien, Florian, Magda, Arnaud and Lela for the trips to the Great North (Norway), and to Antoine Tran, Alexandre Kohen and Flora Weissgerber for the trips to the Great South (Paris).

At a higher level, I would like to thank all the teachers who passed on their passion for Science to me throughout my studies, and often supported me in pursuing further. I owe them a lot, and hope to do the same one day. Thanks to Olivier Guillaumin for the very informative discussions on the future of programming languages, and for supporting my thesis application, even though I was resigning from his company. Thanks also to Wendy Mackay for taking the gamble of taking me on a predoctoral internship without prior experience, and for supporting my thesis application in Lille afterwards.

Finally, I couldn't go back in time without mentioning the tools that made my life easier during the writing of this manuscript, and which I am happy that they existed in time for my work. I am thinking in particular of Paged.js (HTML to book conversion), which allowed me to write this manuscript entirely in HTML, to manage the typography with CSS, and to generate dynamic elements (table of contents, figure numbers, reference links, etc.) with JavaScript. Thanks to Zapf and Frutiger for their work on the Palatino and Avenir fonts. And a big thank you to Hal Elrod for his ***Miracle Morning*** method which made the writing period particularly pleasant and productive.

Finally, I would like to thank my parents for being the island of stability that I could always cling to when things were not going well. And I would especially like to thank Audrey, who has brightened my life since we met, and showed great lucidity when it was necessary to unravel inextricable situations. With her I learned to desire a simple life, with kindness, hiking, and ecology. To summarize our epic I will end with the final line of ***Fight Club :*** “ You ***put me at a very strange time in my life. »***

Résumé

The objective of this thesis is to study and develop programming models for interactive systems, to promote prototyping and the development of new interaction techniques.
In this context, developers mainly use generic graphical interface frameworks, which they “tinker with” to integrate new ideas. However, these frameworks are poorly suited to such uses, and restrict the freedom of developers to deviate from established standards. A first study based on interviews identifies the problems, needs, utilities, and “DIY” techniques encountered in this context. A second study based on an online questionnaire assesses the prevalence of the problems and techniques identified in the first study, as well as the reasons why frameworks from research are rarely used to design new interaction techniques. The results of these studies lead to the definition of three Interaction Essentials, practical recommendations intended to improve interaction support in frameworks and programming languages: (i) explicit and flexible orchestration of interactive behaviors, (ii)) a minimal interaction environment initialized at the start of any application, and (iii) standardized mechanisms and conventions supported by a flexible language. These recommendations are illustrated for the first time by an extension to the Smalltalk language, which allows you to express animations by adding a duration to function calls.

The application of Interaction Essentials is illustrated a second time by the creation of the Polyphony framework, which is based on the Entity-Component-System model from Video Games, and adapts it for the creation of graphical interfaces and techniques. of interaction.

Abstract

The aim of this thesis is to study and develop programming models for interactive systems, to promote the prototyping and development of new interaction techniques. In this context, developers mainly use generic interaction frameworks, which they “hack” to integrate new ideas. However, these frameworks are poorly adapted to such uses, and restrict the freedom of developers to deviate from established standards. A first study based on interviews identifies the problems, needs, utilities, and “hacking” techniques encountered in this context. A second study based on an online questionnaire assesses the prevalence of the problems and techniques identified in the first study, as well as the reasons why research frameworks are rarely used to develop new interaction techniques. The results of these studies lead to the definition of three Interaction Essentials, practical recommendations to improve the support of interaction in frameworks and programming languages: (i) an explicit and flexible orchestration of interactive behaviours, (ii) a minimal and initialized interaction environment at the start of any application, and (iii) standardized mechanisms and conventions supported by a flexible language. These recommendations are illustrated initially by an extension to the Smalltalk language, which allows animations to be expressed by adding a duration to function calls. The application of the Interaction Essentials is illustrated once more with the creation of the Polyphony framework, which is based on the Entity-Component-System model originating from Video Games, and adapted for the creation of graphical user interfaces and interaction techniques.

Contents

Introduction

Research in Human-Computer Interaction is constantly exploring new ways of interacting with digital systems. For example, it combines the use of mobile devices, interaction with the finger and with the rest of the body, or even immersive navigation in 3D virtual universes. These developments follow and accompany regular technical progress. They thus ensure the regular development of new forms of interaction, the emergence of previously impossible innovations, and the renewal of modern uses.

However, the tools we use have progressed little since the appearance of the first graphical interfaces. They are often poorly suited to the search for new forms of interaction, directing developments towards stereotypical interfaces - such as selecting buttons, entering text with a keyboard, or separating applications with windows. Research is active in this area, and numerous tools regularly emerge to support the development of new interactions [Led18], but they are not sufficiently used in practice [Mar17]. Over time, it seems that a gap has widened between tools demonstrating innovative software architectures, and those more comprehensive benefiting from decades of development (frameworks).

Developing research prototypes is a difficult activity, requiring time and expertise. Researchers often have to spend a lot of effort integrating their ideas and prototypes into complex, and otherwise inflexible, computer systems in order to demonstrate their relevance and validity. Research demonstrators struggle to match the usability of existing interfaces, to which end users are accustomed. As a result, the often rudimentary aspect of research prototypes hinders their dissemination to targeted audiences, even though they are often practical contributions, intended to be integrated into modern interfaces as soon as possible.

Research questions

We are facing a complex situation, with many causes and many attempts at solutions. The first objective of this thesis is therefore to understand this situation, and to expose each of its aspects, in order to lay the foundations for discussions and arguments. Throughout this research, we focus less on the causes that led to current tools, and more on what we can contribute to them today. Thus, we first try to answer the question **What are the needs of researchers for the prototyping of new interfaces and interaction techniques?**

The subject of this thesis is the study of tools and languages for designing interactive programs (interacting with people), in particular for research. Our objective here is to study existing tools and what researchers do with them. We also analyze the tools from the

research that has explored alternative interface programming paradigms, with varying success. Furthermore, “tools” are not just software libraries, and we consider in our study the work of producing and promoting knowledge. It is then a question of understanding the impact of different types of contributions, to propose new tools adapted to the needs of researchers. Thus, we then answer the question **In what forms can we contribute to the HCI prototyping activity?**

Finally, in this thesis we choose to focus on software libraries and architectures for prototyping. These types of contributions being very widespread in HMI, we can base ourselves on a vast body of existing work, and thus analyze their advantages and disadvantages. It’s about determining which needs they meet, and which ones still lack support today. We therefore wish to offer innovative work, in line with contemporary technologies. In particular, the influence of programming languages on the expression of interactive programs is explored further in this thesis. Thus, we finally answer the question **How to design a framework or a programming language that meets part of the needs of HCI researchers?**

Methodology

Our subject of study in the broad sense is ***interaction programming.*** Like any human activity, it is difficult to quantify, that is to say it will be difficult to move forward and validate objective truths (e.g. ***more than 50% of researchers have difficulty with tool*** X) . In addition, many factors can influence programming activity (e.g. the user's physical comfort, their current mood, their office colleagues), not all of which can be controlled. Thus, the complexity of the situation we are studying is such that we have not sought to isolate each of its causes.
We based ourselves on systematic observations of researchers who programmed interactions, and from a sufficient number of observations sought to formulate ***probable hypotheses.*** Our method is inspired by grounded theory , by which we formulate hypotheses by induction, based on our observations [Gla17]. These hypotheses are limited by the variety of our observations, therefore generalize to a very small population of people. In this sense, we do not seek absolute truths and certain answers to our research questions, but to argue for probable solutions. This manuscript is therefore structured as a discussion support, in order to leave it to readers to nuance our conclusions, and if possible to contribute to seeing them more clearly.

We therefore began this thesis work with a series of interviews with HMI researchers, about their past interaction development projects. A systematic analysis of these interviews then allows us to formulate observations, which we then tested using an online questionnaire. Based on the results of the interviews and the questionnaire, we identify high-level needs for the prototyping of innovative interactions, and propose different avenues of contribution to respond to them. Secondly, we brought interaction programming closer to the use of a programming language, and considered the close relationship between an interaction library and the language on which it is based. We then formulate three ***Interaction Essentials,*** practical recommendations aimed at guiding the design of interactive systems, whether using a programming language or a software library.

These recommendations are intended to justify the choices of contributions in this manuscript, as well as our future work in the field. Finally, we explored both types of contributions, by developing on the one hand an extension of the Smalltalk language dedicated to animations, and on the other hand an interaction framework dedicated to the prototyping of ad hoc interfaces.

Contributions

Two contributions mainly emerge from this thesis. The first is an in-depth study of the interaction programming activity in a research context, based on interviews and the online questionnaire. In particular, we present the main problems that the researchers we interviewed faced, as well as the resolution strategies they employed. These observations have led us to consider that areas requiring future investment are low-level interaction support, and the rapprochement of web technologies with desktop applications.

The second main contribution is the application of an original programming model, the Entity-Component-System, to the programming of graphical interfaces. This model, based on the principle of ***Composition rather than Inheritance,*** facilitates the explicit manipulation of interactive behaviors in an interface, and promotes their reuse. Furthermore, it materializes input/output devices into virtual entities, which facilitate the implementation of interaction techniques using these devices. We implemented this contribution in a software library, Polyphony.

Plan of the manuscript

In **Chapter 1. Programming interaction,** we detail the context and research issues of this thesis, and present the current state of knowledge on interaction programming. We then introduce the interviews and the online questionnaire, which allow us to portray a clearer vision of programming activity in a HCI research context, and finish with a set of recommendations for future contributions to computer programming. interactions.

In **Chapter 2. Interaction Essentials,** we develop a set of three principles for engineering interactive systems. These principles guided our work during this thesis, and justify our choices of contributions. Each Interaction Essential is presented independently, and discussed using observations and reflections on the current and future state of computer systems.
The last point is developed from exploratory work on the expression of animations, with an extension of the Smalltalk language.

In **Chapter 3. The Polyphony Toolbox,** we present our work on designing a software library for creating graphical interfaces. We detail the Entity-Component-System model, which we adapted from the field of Video Games to Human-Computer Interactions. We demonstrate a functional implementation of this work, and detail its software architecture. The entire chapter is structured to make this work easy to reproduce, for anyone who would like to adapt it again.

Publications

We present here the list of publications produced during these four years of thesis. For articles whose content is partly presented in this manuscript, we indicate the corresponding chapter or section in parentheses.

1. **Raffaillac, T.** (2017). Language and System Support for Interaction. ***Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (Doctoral Consortium),*** 149–152. [Raf17]
2. **Raffaillac, T.,** Huot, S., & Ducasse, S. (2017). Turning Function Calls into Animations. ***Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 81–86. (section 2.4) [Raf17]
3. **Raffaillac, T.,** & Huot, S. (2018). Application of the Entity-Component-System model to interaction programming. ***Proceedings of the 30th Conference on Human-Computer Interaction,*** 42–51. [Raf18]
4. **Raffaillac, T.,** & Huot, S. (2019). Polyphony: Programming Interfaces and Interactions with the Best Paper. ***Proc. ACM Hum.-Comput. Interact.,*** 3(EICS), 8:1–8:22. (chapitre 3) [Raf19]
Entity-Component-System Model.

Chapter 1. Program the interaction

The way we interact with computers has evolved significantly over the past few decades. Their technical capabilities have progressed greatly, and continue to improve. They are capable of calculating faster, and carrying out more complex operations, while consuming less energy. They are smaller, lighter, and are now able to fit in a pocket or in the palm of the hand. They are more versatile, and multiply the sensors interacting with humans and their environment (mice, touch screens, physical buttons, microphones, accelerometers, etc.).
Finally, they are no longer confined to a dedicated box, but are integrated into everyday objects, in our environment, or even accessible via the Internet.

At the same time, uses related to computers evolved as they became more powerful. While from the 1950s there was one machine for several users, from the 1970s computers for personal use developed, from the 1990s we could find several machines for one user [Wei97], and nowadays it is common for several users to share several machines. Likewise, the contexts in which computers are used have greatly multiplied (scientific computing, office automation, industrial production, telecommunications, entertainment, etc.), and have contributed to the evolution of interfaces between humans and computers.

1.1 Context

Today, as the contexts of computer use continue to change, we still need to develop new types of interfaces, and improve the ways we interact with computers. Thus, the context of this thesis work is the **research and development of new interactions with computers,** in order to study and propose adequate tools for this activity. But then, ***what is interaction?*** And what do we mean by ***programming the interaction ?***

1.1.1 Characterization of the interaction

The definition of interaction has long been a topic of discussion in the Human-Computer Interaction (HCI) community. This definition is important because it should ideally encompass all human production related to the interaction between people and machines — past, present, and prospective. It delimits the HMI domain, characterizing what it ***is*** and what it ***is not.*** We do not have the ambition to provide a definition here, however we must characterize it in order to clarify the object of this thesis — interaction ***programming .*** Thus the Dictionary of the French Academy defines interaction as:

INTERACTION nf 19th century.

> PHYS. Reciprocal action of two or more bodies. ***Gravitation is a phenomenon of interaction between two bodies. Electromagnetic interaction. Strong interaction,*** attractive action between particles, which ensures the cohesion of atomic nuclei. ***Weak interaction,*** which manifests itself by forces of attraction or repulsion between particles, responsible in particular for beta radioactivity. • By ext. Influence exerted on each other by phenomena, facts, objects, people. ***The interaction of economic and political facts.***

This definition, taken from physics, applies by extension to Human-Computer Interaction, and we can summarize it by “reciprocal influence”. However, it does not specify in any way what types of influences man and machine can have on each other. It is therefore insufficient to characterize HMI work and its needs.

Dans leur article ***What is Interaction?*** paru en 2017, Hornbæk et al. s'attachent à présenter les principaux “courants de pensée” qui donnent des définitions de l'interaction [Hor17]. Ils en tirent la tentative de définition suivante : « Following ***Bunge, interaction concerns two entities that*** determine ***each other’s behavior over time. In HCI, the entities are computers (ranging from input devices to systems) and humans (ranging from end ef ectors to tool users). Their mutual determination can be of many types, including statistical, mechanical, and structural. But their causal relationship is teleologically determined: Users, with their goals and pursuits, are the ultimate metric of interaction ».***

This definition abandons the notion of action or influence, in favor of that of determination — or observation to acquire understanding. In this sense it seems to us to complement the definition given by the French Academy, rather than replacing it. From these definitions, we can specify what research into “new interactions” with computers consists of, and thus refine the subject of study of this thesis. This research is characterized by:

- new ways for Men to act on Machines — or in IT terminology, to transmit information to them; new ways for Machines to transmit information to People;
- new ways of linking reception and transmission of information (in humans or machines),
- in order to help others interpret coherent behavior based on these exchanges.

The first two points mainly refer to the development of new ***interaction devices,*** as well as the processing linked to them (e.g. transfer functions for pointing devices). Interaction peripherals are devices (physical or virtual), which capture actions (physical or virtual), and translate them so that they can be perceived by the recipient. For example, the mouse is a physical device, which captures hand movements, and transmits them to the computer in the form of relative movements. The smartphone keyboard is a virtual device, which captures touchscreen presses and transmits them to the computer as text input. Finally, the screen is a physical device, which captures color matrices (pixels), and transmits them to the user in the form of light points.

The last point — coherence between reception and transmission — refers to ***interaction techniques.*** These are generally computer programs, which express complex reactions based on simple rules ***“if I receive A, then I emit*** B”. Interaction techniques are ideally ***consistent,*** that is, users can anticipate how they will work before

to have interacted, and understand what is happening during the interaction. For example, if we click on a drop-down menu, we expect a list of options to be displayed, and the visually highlighted option to be the one selected. When the user's intention does not correspond very well to what the technique offers, we speak of “Gulf of Execution”, and when the technique does not communicate its current state very well, we speak of “Gulf of Evaluation” [Nor88]. Finally, note that we mainly talk about the behavior of the Machine seen by Man, and rarely the reverse, which is what Hornbæk et al. synthesized by “ There ***is little “C” in HCI ”*** [Hor17].

Because they are generally expressed in programs, interaction techniques are the most representative of interaction programming: they are programs that interact with users. As for interaction peripherals, they are often physical objects rather than programs, and in our characterization above their role is to transmit rather than interact — hence their quality as “peripherals”. In practice when they are ***the object*** of interaction rather than the ***means,*** we consider them as interaction techniques. For example, we can say that the virtual keyboard is an interaction device, but the study of using gestures to enter words on it is an interaction technique [Zha12]. So in this thesis we mainly focus on interaction techniques.

Finally, we often talk about programming ***interfaces*** in conjunction with interactions. The distinction between the two terms is historical: we used to speak of Human-Machine Interfaces, then the meaning became generalized with Human-Machine Interactions (although the first meaning is still often used). Indeed, the interface materializes the object of interconnection between human and machine, while the interaction more broadly designates the surrounding situation. In our case, programming always results in concrete code. Consequently, when we program interaction, it will necessarily be materialized by an interface. When it is visible on a screen, it is called a graphical ***user interface.*** In common sense, the difference between an interface and an interaction technique is that the former is the site of a greater number of interactions — we often develop an interaction technique ***for*** an interface. We therefore choose to focus on this basic building block, which we now need to describe more precisely, to understand the prototyping activity that interests us.

1.1.2 Characterization of interaction techniques

Tucker [Tuc04] defines interaction techniques as follows: “ An ***interaction technique is the fusion of input and output, consisting of all software and hardware elements, that provides a way for the user to accomplish a task ”.*** This definition is a pragmatic and technology-related characterization, which clearly delimits the initial object of our study. There are many works that can be qualified as interaction techniques, which have given rise to numerous classifications in the scientific literature [Jai07, Jan13, Bai16]. In addition, there are distinct development contexts for each interaction technique, with distinct interaction modalities (e.g. desktop applications with keyboard/mouse/screen, mobile applications with finger/screen, Virtual Reality with controller/headset). To give an overview of interaction techniques without going into the details of different contexts, we associate them by the ***types of tasks*** they accomplish, which are represented throughout. So we can distinguish:

- menu techniques (see figure 1) pointing
- techniques (see figure 2) navigation
- techniques (see figure 3) selection
- techniques (see figure 4) learning/
- feedforward techniques (see figure 5) techniques editing
- (see figure 6)

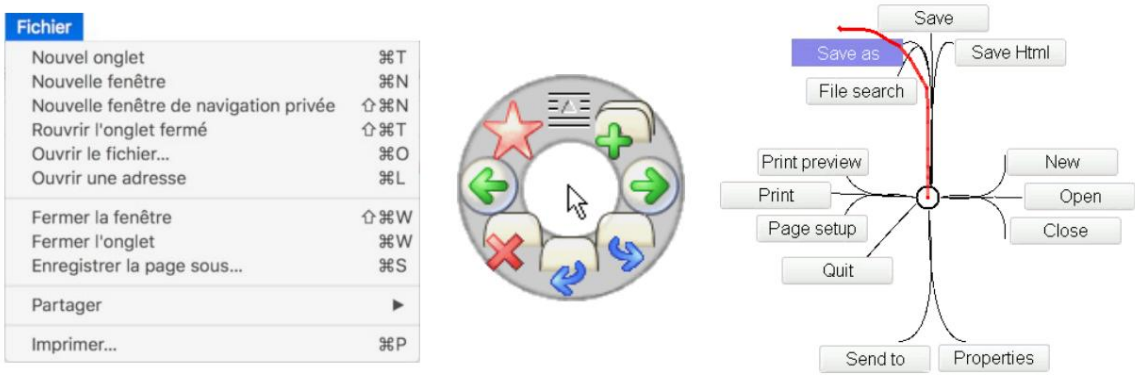

Figure 1: Illustration of three menu techniques, from left to right: macOS drop-down menu, circular menu, and *Flower Menu* [Bai08]

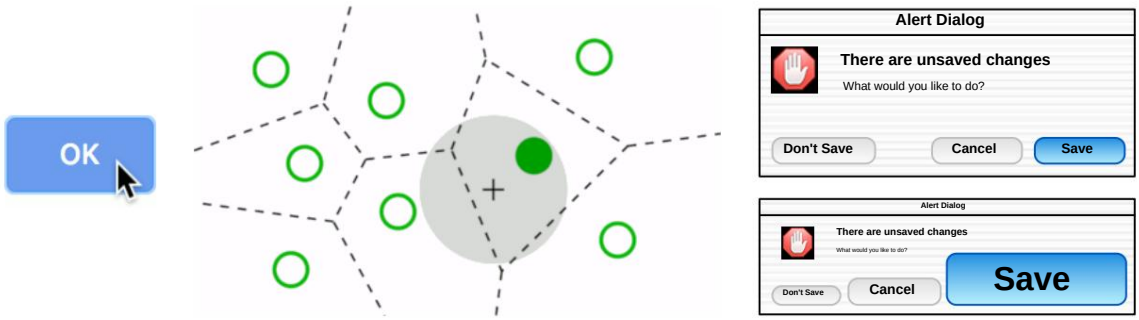

Figure 2: Illustration of three pointing techniques, from left to right: hover pointing, *Bubble Cursor* [Gro05], et *Semantic Pointing* [Bla04]

Figure 3: Illustration of two navigation techniques, from left to right: browser tabs Web, and avatar teleportation in Virtual Reality in Unity

Figure 4: Illustration of three selection techniques, from left to right: text selection, Photoshop lasso selection, and *query relaxation* element selection [Hee08]

Figure 5: Illustration of three learning/feedforward techniques, from left to right: tooltip when hovering the cursor over an interactive element, *ExposeHK* [Mal13], And *Octopocus* [Bau08]

Figure 6: Illustration of three editing techniques, from left to right: writing text on the keyboard, drawing with a brush in Gimp, and *Toolglass* [Bie93]

Finally, when we talk about interaction techniques today we must mention the major influence of the WIMP paradigm ***(Windows, Icons, Menus, Pointer).*** This paradigm, developed in 1973 with the Xerox Alto system, and popularized by the Macintosh in 1984, laid the foundations for our ways of interacting with computers. It defines a virtual environment inspired by the work office, and originally intended for office workers. This is made up of movable and resizable ***windows ,*** interactive ***icons*** representing the elements to be manipulated (files, programs), menus ***allowing*** you to list, select and execute commands, as well as a ***cursor*** controlled by the mouse, which allows you to interact directly with the virtual environment. Thanks to its simplicity of learning and use, as well as its adaptability to many contexts, the paradigm

WIMP has widely established itself on modern interactive systems, becoming a ***de facto standard.*** Most interaction techniques therefore ***integrate*** with this paradigm, by proposing artifacts that can coexist within WIMP systems.

The success of the WIMP paradigm has enabled the standardization of the user experience on different systems (Windows, MacOS, Linux, etc.). On the other hand, it was also followed by a crystallization of interaction techniques around well-established concepts, giving rise to stereotypical graphical interfaces, and to innovations apparently less ambitious than in the past. In reaction to this phenomenon, so-called “post-WIMP” interaction techniques were developed [Dam97], with the ambition of more profoundly redefining the way we interact with computers. This work has also led to a reconsideration of the ways of ***programming*** interaction techniques, and is the point of origin of the subject of this thesis [Huo06].

1.1.3 Interaction programming

We come to the development of new interactions, and more particularly to their programming. In a computer system, the majority of reactions to user actions are expressed by programs. Some reactions are specified mechanically (the sensation of a mouse “click”), and others electrically (the lighting of an LED when the computer is powered on), but once designed it is very difficult to modify them. It is therefore mainly through programs that researchers and designers develop new interactions with computer systems. We will also talk about HMI programming, or when specific to graphical interfaces, interface programming.

A program is a sequence of instructions, which sent one by one to the central processor (CPU), allows it to be controlled by making it perform simple tasks (read/write to memory, perform operations on numbers, communicate with devices connected).
Each program is expressed in a ***programming language,*** which defines a syntax by which the different instructions to be executed are specified. There are many programming languages, which are distinguished by the ways of reasoning with them, as well as their contexts of use. However, we can cite the languages mainly used today to express interaction techniques: C/C++, Java, Python, C#, JavaScript, and Objective-C.

The design of a new interaction technique involves first describing its operation, or even modeling it using mathematical tools. We are only interested in programming, that is to say once the technique is modeled, its expression in code. Programming a new interaction technique involves a certain number of steps:

- We create “visible” elements (most often on the screen), which suggest by their presence that interaction with them or the environment is possible.
- We list the actions on the visible elements to react to, and for each we define a subprogram to execute when it is triggered.
- After execution of the subroutine of an action, we update the visible elements and possibly create new ones.

To carry out these steps, a language is used, as well as a software library making it possible to detect user actions and record programs to be executed on actions. This type of libraries provides services related to the management of inputs and outputs with interaction devices, which make them essential for programming interactions. We commonly distinguish between ***frameworks*** and ***toolboxes.***

1.1.4 Frameworks and toolboxes

“Frameworks” and “toolboxes” are relatively vague terms, used to designate software libraries of large and small sizes respectively. To simplify in this manuscript, we call a toolbox what is not a framework, and we try to clarify what a framework is. Broadly speaking, a framework is a software library that makes it easier to create interfaces, declaring them as data structures rather than raw code.

It provides reusable elements allowing assembly or composition with minimal effort. In common sense, a framework is a library that:

- cannot be used in conjunction with other frameworks or versions of itself retains
- control of execution when the machine does nothing (“does not give back”) provides a
- development framework ***(framework)*** rather that a simple collection of tools or functions requires adapting
- other libraries to its operation (by extension mechanisms) rather than the reverse

We sometimes associate frameworks with the contribution of programming concepts and ***styles*** (e.g. ***widgets, listeners,*** signals/slots in Qt), but these characteristics are not systematically present. Likewise, libraries qualified as frameworks are often large-scale projects, equipped with numerous and varied functionalities, but without being able to rely on these observations to characterize them. Only one feature seems to link the points listed above, and that is ***Inversion of Control,*** which is generally recognized as distinctive of frameworks [Fow05, Joh88]. With Inversion of Control, an application entrusts its code to the framework for execution, rather than executing it itself. We illustrate it by the Hollywood principle [Swe85] : “ Don’t ***call us, we’ll call you ”.***
The control is thus reversed, since it is the framework which is responsible for organizing the different functions under its responsibility.

However, today several forms of Inversion of Control exist, which complicate the classification of different libraries. Thus, in the case of a library like JavaFX, it involves providing objects whose methods are executed by the framework (widgets), and also providing blocks of code to be executed when events are triggered (callbacks). The library responds to the points stated above, we can talk about a framework. In the case of a library like SDL [Lan98], we can also provide callbacks ***to*** react to input events, however the library does not retain control of execution when the machine is doing nothing. Indeed, we must call the **SDL_PollEvent()** function regularly, which executes the recorded ***callbacks*** then immediately returns control. Under these conditions, we are not talking about a framework here. We can therefore clarify our definition of a framework, which is **a software library which permanently takes control of the execution of the application, and allows the execution of third-party code by receiving *callback functions and methods.***

When researchers prototype new interaction techniques, they necessarily rely on one or more software libraries, which allow them to access interaction devices. HCI prototyping is therefore closely linked to frameworks and toolboxes. They are associated with different types of work facilitating interaction programming:

- the creation of new interaction toolboxes, which can ultimately evolve into ***frameworks***
- the invention of high-level modeling tools for interaction techniques, which are then possibly ***transpiled*** to lower-level languages development of new
- software architectures for interactive systems, which can facilitate their development and use; development or modification of
- programming languages, which facilitate the use of existing or new frameworks

In this context, researchers have at their disposal numerous tools to carry out the exploration and development of new interaction techniques. Why then is this situation not satisfactory?

1.2 Problem

Today, we can consider that interaction with computer systems has reached a point of stability. Desktop environments (Windows, macOS, Linux) share many common points linked to the WIMP model (movable and resizable windows, icons on the desktop, drop-down menus at the top, use of the mouse). Likewise, finger interaction on smartphones and tablets has become standardized between systems (use of finger gestures, application icons on virtual “screens”, execution of only one application at a time). The interfaces have become “standardized” [Bea00, Poo16], facilitating access to technology for users, and allowing them to reuse the same acquired knowledge between different systems. In addition, the programming of interfaces has been made faster, by expressing them through assemblies of standard controls (e.g. buttons, check boxes, menus, tabs), and thanks to a standardization of software architectures around the object model [Kra88 , Neck87, Con08].

However, this standardization has given rise to stereotypical interfaces, which are little different from each other, and whose developments (both functional and cosmetic) are weak over time. As a result, it is difficult to adapt current interfaces to new uses, and they lack the flexibility to introduce new modes of interaction. The crystallization of interface programming has generated a complex ecosystem, from which it is difficult to escape today. Several obstacles stand in the way of researchers' prototyping work: the ***horizontal*** multiplicity of tools with which to work, the ***vertical*** multiplicity of layers in interactive systems, and the limited extensibility of libraries.

1.2.1 Choice of programming tools

The first obstacle is the state of the systems' basic tools, for programming inputs/outputs with users. The operating systems, which manage these inputs/outputs, provide complex and/or very low-level programming interfaces (or APIs, for ***Application Programming Interface) ,*** that is to say which require taking care of many details. For example, in Linux at

At least two major APIs are available for drawing to screen, framebuffer and X11. The first does not provide any drawing primitive (line, polygon, image), and only gives access to a matrix of pixels. The second provides an API of 239 functions, described in a 476-page document [Get02].

Faced with this underlying complexity, programmers must resort to intermediate libraries, which interface with low-level APIs, while providing higher-level APIs allowing more complex behaviors to be expressed with less code. But there are a ***very large number of them.*** Indeed, multiple contexts require the programming of interactions, each with particularities that require specific tools: office interfaces, interactive data visualization, video games, interfaces for critical environments, smartphones, or even interfaces on microcontrollers. Each context benefits from dedicated libraries, however contexts often share needs (e.g. mouse management between desktop interfaces and interactive visualizations). It is therefore common for a library to be used outside its original context, and certain “generalist” libraries highlight their usability in many contexts. This is the case for example of Qt [Qt19], which cites on its main page the contexts ***Embedded Devices, Application UIs and Software, Internet of Things, Mobile, Automotive, Automation, Set-top-box & DTV,*** and ***Medical.***

Programmers are therefore confronted between the choice of specialized libraries, capable of responding in detail to a context of use but potentially limited outside it, and general libraries, capable of responding to numerous contexts, but potentially in a superficial manner. Additionally, when multiple libraries are used together, an additional interoperability issue arises. As Huot [Huo13] writes , “ As ***discussed before, we now have many different tools that address specific problems, but they are difficult or impossible to use together. Dif erent toolkits are based on dif erent abstractions and models, requiring to deal with several programming paradigms, abstractions, data structures and even sometimes programming languages within the same prototype ”.*** In practice it takes a lot of time to search and compare such libraries. When the choice falls on one of them, one must invest significant time and effort in learning how to use it, with no guarantee that it will meet all needs. This is particularly tricky in the case of developing new interactions, which frequently hit the limits of interaction libraries.

1.2.2 Structure of layered APIs

While the first obstacle to programming new interaction techniques involves a ***horizontal*** multiplicity of available libraries, the second obstacle concerns a ***vertical multiplicity.*** Managing input and output with users, in an interactive system, is generally organized in successive layers (see Figure 7), forming a hierarchy of peer-to-peer dependencies.

At the lowest level, device drivers ***communicate*** with the chips connected to the motherboard which manage external communication protocols (USB, audio, HDMI), and transmit this information to the operating system. The latter matches the information received into ***virtual*** devices (like the cursor on the screen), which allows different types of devices (mouse, finger, ***eye-tracking) to be used uniformly.*** Then, some languages

programming (rare) intercept the input/output information of the operating system, to make it available to programmers. Above, the number of layers is variable and can represent complex dependency graphs. However, we generally distinguish the existence of low-level libraries, which each specialize in a type of physical/virtual device (e.g. OpenGL for display, TUIO for multitouch, OpenAL for audio).
Finally, at the high level are application frameworks, which combine and provide access to all devices.

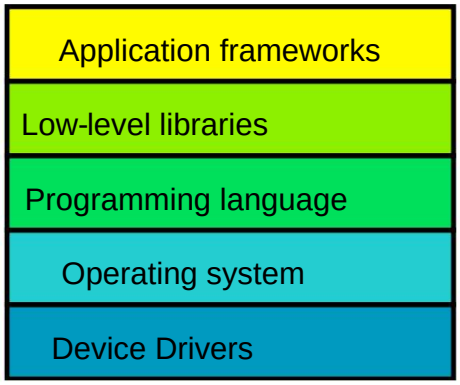

Figure 7: Layered structure of different types of software libraries for managing user interaction.

During a user action on the machine, the data generated by the peripheral sensors pass through each of the layers of this hierarchy. Each receives information from a lower layer, possibly interprets it into more complex actions (e.g. a mouse click is a press followed by a release), and makes it available to the higher layers. The same phenomenon occurs for machine actions towards the user, although we are preferentially talking about ***inputs.*** With this structure, the highest level data (which has passed through the most layers) is also the most abstract. Programmers query a single layer, as high as possible, and do not normally interact with other layers.

However, this layered hierarchy makes programming interactions complex. Indeed, each transmission from one layer to another can see information being ignored/lost (e.g. the resolution of the mouse sensor), converted into other units (e.g. a movement in millimeters of the mouse towards a displacement in pixels), or even corrupted during translation. It also happens that a layer merges information, and loses track of their cause. This is the case for example when a button is triggered at the application framework level, it is sometimes impossible to know whether it was a mouse click or a keyboard shortcut that triggered it.

From the point of view of the programmer seeking to prototype a new interaction technique, one must already choose between all available layers, where to intercept the interaction data. Each layer has an API that must be learned beforehand, as well as a programming model potentially different from the others ***(callbacks,*** events, ***listeners).*** When data is missing, you have to go to a lower layer, learn how to use your API, and risk having multiple contacts. In this case, we are exposed to consistency problems between layers: nothing guarantees that information propagated by one layer is also propagated by another.

high. For example, if a lower layer reports a mouse button press followed by a release of the same button, we cannot guarantee that a higher layer will interpret a mouse click. The programmer must therefore consider that the information intercepted at several levels may not be coherent, and compare them each time to “restore” their consistency. Overall, this layered organization tends to increase the programmer's task, both in the time spent understanding the topology of the layers, and in the efforts to ensure the consistency of the intercepted data.

1.2.3 Extensibility of libraries

The third obstacle to prototyping new interactions is the low extensibility of software libraries, and more precisely of application frameworks. First of all, in the field of HMI, graphical interfaces are mainly coded using the ***object model.***
In this model, all program elements (integers, strings, buttons, windows, etc.) are objects. An object is a collection of variables and functions (methods), which only reacts to its environment through method calls, and manages its state by manipulating its own variables. In frameworks, all interactive graphic elements (buttons, labels, ***sliders,*** etc.) are represented by objects, which are then called widgets . Thus any interface is an assembly of widgets, which communicate with each other and with their environment through method calls.

For a researcher, creating a new form of interaction normally means creating new widgets. However, these objects are inherently complex. They must manage numerous interactions with their environment (being displayable, resizable, reacting to the keyboard, being compatible for accessibility to disabilities, etc.), which must necessarily be implemented when creating a new widget. We tend to describe these frameworks which are based on heavy and inflexible objects as “monolithic” [Bed04]. Faced with this situation, adding a new functionality is a difficult operation, which requires understanding (at least in part) the behaviors managed by the widgets, the dynamics between objects in the program, and the data structures used to store widgets.

In practice, in addition to their structural complexity, frameworks are often not very extensible, mainly because it is a secondary need for their user base. The APIs proposed to integrate extensions are often internal functions, which have not necessarily been designed to be exposed, nor tested extensively in advanced use cases. Bugs may remain for such advanced uses, and receive little attention given the low interest to other potential users. The documentation of such portions of APIs is often insufficient, especially since for internal functions the API is more often subject to evolution, so their documentation must be regularly corrected. As the number of users with similar needs is low, online help is mainly limited, and we walk “off the beaten track”. Finally, it happens that frameworks integrate presumptions into their code, which seriously compromise the flexibility and replacement of certain functionalities. This is the case, for example, for changing the pointer transfer function, which matches the movements of

the mouse with the movement of the cursor on the screen [Case11]. This modification is impossible in a framework like Qt, and requires calling at the lowest level, compromising the flexibility of the framework which should be able to express this type of changes.

Finally, in this section we have highlighted the ***horizontal*** multiplicity of libraries to choose from, the ***vertical*** multiplicity of interdependent layers in interactive systems, and the difficulties of extending software frameworks. Now that we have presented the ecosystem of interaction programming and the issues related to the prototyping of new interaction techniques, it is appropriate to focus on the needs of the ***researchers*** themselves.

1.3 State of knowledge on interaction prototyping

As part of this thesis, it is important to take stock of the knowledge on the prototyping of interaction techniques in a research context. Indeed, interface and interaction programming tools have been mainly designed for users wishing to reuse standard and proven interaction methods (e.g. buttons, tabs, keyboard shortcuts), and accepting stereotyped interfaces. Their needs are well known, and have helped shape the
current tools.

For a context of research and prototyping of new interaction techniques, the needs are less well known, which leads to question the relevance of classic creation tools for these uses. Furthermore, the notion of “prototyping” is vague, it applies in many fields, and is not confined to IT. We therefore seek here to clarify **the activity of programming functional prototypes demonstrating advanced interaction techniques.** The state of knowledge presented here should allow us to describe this activity today, in order to form a basis to be completed by the work of this thesis. We analyzed existing work according to a set of questions:

- What do they actually teach us about the activity of programming advanced interaction techniques? What contexts are they
- interested in?
- Do they raise other issues than those we have stated?
- What solutions do they propose to achieve?

This state of the art is divided into three parts. In the first part, we present an overview of the types of work aimed at improving the usability of programming tools. This work is mostly presented for non-research contexts, however the fact that they are often tested and validated by researchers encourages us to mention them here. In the second part, we present the field of Opportunistic Programming [Bra09], which had as its object a programming activity similar to ours. In the third part, we describe the work mentioning or specifically targeting the context of HMI research.

1.3.1 Usability of software libraries

The field of study of the usability of programming tools is very vast. The ISO 9241-11 [ISO18] standard defines it as " the ***degree to which a product can be used by identified users to achieve defined goals, with effectiveness, efficiency and satisfaction in a specified context of use ".*** Tool usability directly influences programmer productivity, and has motivated active research in this area. We don't aim to list them all here, but studying the factors affecting usability can help us better understand the programming activity that causes these factors.

1.3.1.1 Quality of APIs

An API ***(Application Programming Interface)*** is the set of functions and data types offered by a software library for its users (programmers). It is the interface between the library functionalities and the programmers. Designing an API involves choosing which functions are exposed (some may remain internal to the library), their names, and the order in which they will be used. It is the result of a compromise between power of use (controlling every aspect of the library), and simplicity (using fewer functions).
The API is the second artifact that users of a library encounter (after documentation). It determines the amount of code they will write, and the time they will spend understanding its logic. It is therefore a major element of ***the user experience*** of any software library.

The characteristic contributions linked to the APIs aim to:

- determine what a ***good*** API is [Blo06, McL98]
- determine what a ***bad*** API is [Pic13, Gri12, Zib11, Hen09]
- measure the success or usability of an API
- [Bor05] facilitate their learning by users [Dua12, Rob11, Ko11,
- Rob09] promote their "correct" use, that is to say as anticipated by their designers
- [Sai15] allow the evolution of existing APIs to support that of libraries [Sty08]

Through the knowledge they give us on the use of APIs, this work underlines that the programming activity is based on the formation of a mental model of the library used, and that we mainly observe the symptoms of its inconsistency [Ko11, Gri12, Dua12]. With this in mind, learning is an important part of the effort spent using a library [Rob11], and API designers recognize that their design choices have important consequences that must be anticipated. Finally, the recommendations made by this work reveal hypotheses about programmers, which without being explicitly demonstrated, can be inferred from observations. Thus programmers would be resistant to change [Blo06], devote little effort to reading documentation [Dua12], never do what is expected of them [Blo06], and do not tolerate ambiguity in documentation [Bor05].

The issues raised here are mainly the complexity of the APIs, and the time spent using them. Furthermore, the works we studied describe little of the contexts in which they apply, focusing mainly on the programming experience of their participants, and describing generic tasks that can be applied in several contexts. This work mainly falls within an industrial framework, for which the optimization of an API must have a measurable impact [Bor05], which can justify in return what we invest in it.

In light of the importance of design choices, proposed solutions are often to carefully study usage scenarios before designing an API [Sty08]. Some authors also suggest providing examples and coherent documentation to facilitate the construction of a consistent mental model [Rob11, Ko11], and to design new tools for exploring and using an API [Dua12].

1.3.1.2 Programming by end users

End-User Programming refers to tools allowing people who do not consider themselves professional developers to program. These tools generally replace programming languages, letting users describe behaviors and data structures, without requiring mastery of a language. Spreadsheets like Excel are an example of end-user programming: manipulating formulas in the cells of a spreadsheet allows, like a language, to specify complex data transformation behaviors.

End-user programming can be found in:

- spreadsheets (e.g. Excel, Google Docs, Open/Libre
- Office) visual programming, with data flow paradigms [App09, Dra01], state machines [App06, Bux90], or Petri nets [Nav01, Bea00] programming
- by demonstration (or by example), in which the system observes the user's actions to reproduce them by translating them into a program [Mye86, Cyp93, Lie01] scripting

- languages, to a lesser extent, when intended for users with little programming experience (e.g. Arduino, Processing, PHP)

This type of programming is particularly useful in contexts where users who are not trained developers meet (e.g. home automation, audiovisual production, arts, administration). This is also the case in the HCI field, where people from the IT, Design, and Social Sciences fields coexist. In addition, it is a way to improve the adaptability of interactive systems, by allowing users of these systems to configure and control them as closely as possible to their needs.

Just like the work on the usability of APIs, we note here that learning and expertise play a crucial role, since the work discussed in this section aims to reduce them. Visual programming stands out from other fields because it has been used extensively to program interaction techniques. Indeed, most interaction techniques are well represented with graphs, where the edges are the actions that users must carry out. The number of actions to be taken into account is often reduced, and the tasks

mainly sequential (because of the human difficulty in managing simultaneous tasks), it is possible to represent these graphs visually. Finally, we note that a lot of programming work by end users is linked to ***incremental*** construction of programs.
This method of programming consists of maintaining a functional (but incomplete) program at all times, and gradually adding missing features, rather than obtaining a functional program only at the end of development. Incremental construction is particularly suited to programming human interactions, when these are segmented into tasks (themselves subdivided into actions).

In terms of the issues raised here, end-user programming primarily addresses the high “entry ticket” of interface and interaction programming.
As we mentioned at the beginning of this thesis, computer systems are very complex ecosystems, which discourage novices from contributing to proposing new forms of interaction.
The platforms proposed here therefore hide part of this complexity, improving accessibility to interaction programming. In addition, they respond to the difficulty for novice programmers ***to abstract*** the functioning of an interaction technique, when the actions to be carried out seem very concrete. An interaction device could, for example, be represented with an object that can be manipulated directly, to which the programmer can attach behaviors [Dra01]. Finally, another problem raised by this work is that it is difficult with “classic” frameworks to control that a program (or a portion of a program) behaves as desired. This is particularly important in operational safety contexts, where it is essential to have a complete and unambiguous view of the program.

For this problem, richer representations (such as in visual programming) will make it possible to include more details, to exhaustively describe the state of the program.

The contexts for which end-user programming is addressed are very varied and specific, each with specific needs:

- Spreadsheets are designed for automating office tasks.
- Visual programming is used for operational safety, but also in the arts and teaching.

- Programming by demonstration is used in home automation, which lends itself well to defining simple rules (eg. ***If someone enters the room, turn on the light),*** but also in robotics, and in teaching.
- “Accessible” scripting languages are used in all areas where many programmers are novices or self-taught (e.g. Web, embedded computing, arts).

Finally, the majority of solutions proposed in the work studied here are complete and autonomous platforms. They thus largely control the user experience, and save users from interacting with other tools. This autonomy allows them to reduce the knowledge needed before starting to program, at the cost of more limited freedom to design advanced techniques.
Furthermore, in the case of using visual programming, a much debated question is that of the choice compared to a textual programming model [Con13]. Historically they were clearly distinguished, by the use of a text editor on one side, and a graphics editor on the other. However today the two worlds tend to get closer, with editors capable of managing both text and graphic forms. There

The question then becomes ***how*** to combine the best of both worlds [Con14]. A notable example is the exploration of an interface for learning programming by Bret Victor, which enriches the interactivity of the text, and presents graphical views of the code [Vic12]. Possible solutions are therefore possible and to be explored, which would integrate elements of visual programming with textual languages.

1.3.1.3 Integrated development environments

An Integrated Development Environment (IDE) is a set of tools, generally provided in a single program, which form a complete programming environment with the aim of optimizing user productivity.
Among the known IDEs, we can cite Eclipse, Visual Studio, Xcode, or even Qt Creator. They always contain a text editor, and a compiler/interpreter for each supported programming language.

In addition to these basic features, each IDE is distinguished by advanced features, which can offer significant time savings to their users:

- syntax highlighting, which uses color coding to distinguish the different syntactic elements of the program (language key words, variable names, function names, constants) automatic code
- completion, which allows at the press of a key insert a function name or a block of code without having to type it entirely on the keyboard navigation in files
- and classes of the same ***project*** (or between projects), in order to quickly move from one to the other code folding, which
- allows you to hide blocks of source code (comments, class/function body) in order to concentrate on the rest the integration of
- testing, debugging, or version management tools, accessible using keyboard shortcuts , and in interfaces directly integrated into the IDE the export of a project
- in an archive ready to install and use, in order to facilitate its distribution

Development environments are an active area of research, particularly to improve the contextuality and relevance of code suggestions [Oma12, Moo10, Bru09], the visualization of the code and the dependencies between blocks of code [Con12, Ase16], and access to documentation integrated into development tools [One12, Dua11, Zho09].

The knowledge that this work brings us about the activity of programming with an IDE is first of all that programmers ***make use*** of code facilitation functionalities, which justify the interest and success of tools like Eclipse, or Sublime Text. Then, the integration of the different tools in the same interface reminds us that the programming activity (including interaction techniques) uses numerous tools: text editor, compiler/interpreter, saving and synchronizing the code in line, unit testing, debugging, and generation of an installer for deployment. When each tool requires separate learning, more accessible approaches like those presented in the previous section are possible. Another knowledge to be gained from using IDEs is the coexistence of detailed views (the application code) with overview views (e.g. the list of project files, or diagrams

UML [Dur18]). The programming activity therefore requires different levels of inspection, depending on whether we wish to think about the entire project or concentrate on a specific portion. Finally, the success of development environments highlights the importance of extensive use of the human visual spectrum, with the common example of syntax highlighting, and work exploring richer differentiations [Ase16, Con14].

A major problem that IDEs attempt to address is the excessive amount of information accessible to programmers. It includes documentation, tutorials and online support forums, APIs, but also the visual overload due to interface modules sharing the screen. This remark supports the problem of complexity that we raised at the beginning of this thesis. The work presented here seeks less to reduce this quantity of information than to ***prioritize it,*** that is to say highlight the most important elements. They help programmers deal with complexity when it seems inevitable.

Finally, as with work on the quality of APIs, the programming contexts targeted here are poorly defined, with the authors often addressing programmers with at least some experience. Many of the works we have studied argue in favor of integrated contributions to existing IDEs, rather than as independent tools [Moo10, Sty09]. As Omar writes about these two articles [Oma12], “ Empirical ***evidence presented in these studies, however, suggests that directly integrating these kinds of tools into the editor is particularly effective ”.*** Another useful characteristic of tools contributing to programming activity is observation and adaptation to the context of use. Bruch et al. illustrate this with a code completion tool that sorts suggestions by relevance in the current context of use estimated dynamically [Bru09].

1.3.1.4 Knowledge from this work

Most of the work we have studied here has in common a close relationship with complexity . They are part of a complex ecosystem, which they seek to make more accessible for programmers. In the case of work on the quality of APIs, the complexity is that of the interfaces with software libraries, which reflects their power and expressiveness. In the case of end-user programming tools, it is the programming languages that are to blame. Finally for IDEs, the complexity lies essentially in the documentation sources, and the visual accumulation of information. In all cases, the work we have studied does not seek to reduce complexity, but rather to make it acceptable. As Niedoba [Nie19] writes , “ Simplification ***is necessary. But there is a point, simplification reaches a limit. The limit is an unavoidable complexity .*** We can thus distinguish the complexity of the task (which depends on that of the computer system) from the complexity of the tool to carry out the task. Assuming that the task is as simple as it can be, it is the usability of the tool that is in question [Nor10].

Thus, much of the programming tool usability work we have studied focuses on ***the interface*** between the software library and its users — that is, the tools and documents that allow programmers to interact with the library. By this choice, few works call into question the library, its internal functioning, or its position in the library ecosystem faced by users. In this sense, it is implicitly up to the user to adapt, and the interface is there to help them do so. We are still far away

support for co-adaptation (adaptation of technology by users, for their context) as highlighted by Mackay [Mac00], nor even an adaptation initiated by the machine.
Work such as programming by demonstration makes the interface a translator (between the user's intention and the capabilities of the tool) which partly spares this adaptation, however they are limited in power of expression and generally little used today.

Thus, simply observing work on the usability of programming tools can lead us to believe that programming tasks are pre-established, immutable, and must be correctly taught. On the contrary, we consider that it is appropriate to question the nature of certain tasks, in an effective approach to simplification, which we address in this thesis work.

1.3.2 Opportunistic development and *mashups*

Opportunistic development [Bra08] and composite applications ***(mashups)*** [Cao10, Wei10] refer to development practices studied over the past ten years, which focus on rapid application development, regardless of the quality of the result. This type of practice is generally linked to communities of “tinkerers”, to the ***Do It Yourself movement,*** as well as to the prototyping stage of a project. Indeed, since the relatively recent development of the Web, the tools for developing and distributing “computer creations” have become more accessible, attracting many non-professional developers. As the number of developers has increased, communities have formed around computer creation (e.g.
Arduino, Processing, Raspberry Pi), distinguishing itself from other communities by the inclusion of amateur members. These developers have particular uses, distinct from more professional communities, and have therefore motivated research work in order to understand them, and to better support their uses.

Among this research, the field of opportunistic programming has developed around one-off development practices, marked by short-term projects, which are not part of more ambitious projects. The November 2008 edition (volume 25, number 6) of IEEE Software provides an overview, which interested readers can refer to [IEE08].
Brandt [Bra08] defines Opportunistic Programming as follows: “ It ***is an activity where non-trivial software systems are constructed with little to no upfront planning about implementation details, and ease and speed of development are prioritized over code robustness and maintainability ”.*** Mashups are a practice related to opportunistic programming, which is akin to the reuse and combination of artifacts that were not necessarily designed to be reused. Yu et al. [Yu08] define them in the context of the Web by: “ Web ***mashups are Web applications generated by combining content, presentation, or application functionality from disparate Web sources ”.***

Programming new interaction techniques in a research context is similar to opportunistic programming, in that researchers focus on the result, often to the detriment of its robustness. Work in this area therefore helps us clarify how HCI programming activities are “opportunistic”. According to Brandt [Bra09], they consist
in :

- occasional learning of new skills and methods clarification
- and extension of existing knowledge
- reobtaining knowledge deemed useless to retain

This type of practice responds to the lack of horizontal interoperability that we raised as a problem. Mashup designers are faced with artifacts with complementary functionalities, but which were not necessarily designed to coexist. Some are also not designed to be reused. These problems can be caused by insufficient documentation, or inaccessibility to source code, which transform them into “black boxes” for programmers [Obr08]. Finally, as Brandt's definition above suggests, opportunistic programming is linked to needs that are little known upstream, or even changing. This problem limits the robustness of the prototypes, in that their architecture is subject to change during the project.

The contexts linked to opportunistic programming and mashups are not very detailed compared to the areas of use of the created artifacts. However, all the articles we studied are related at least in part to Web technologies. This medium today has a central role in the management of knowledge, for its provision, navigation, and referencing. The proposed solutions mainly integrate with the Web, and promote its closer integration with development tools [Bra09]. We agree with and support this recommendation in the discussions at the end of this chapter.

1.3.3 Studies of HMI programming needs

Understanding the needs of developers of interactive systems is a problem studied for a long time in the scientific literature. Yet it is also one of the least understood today. Indeed, as technologies evolve, new uses emerge or become possible, for which new tools must be developed [Nor10]. In return, the evolution of uses creates a tension (needs) on the technologies, which pushes them to evolve, in a path from ***discoveries*** to ***maturity*** illustrated by the BRETAM model [Gai91]. The cycle of interdependencies between technological developments and uses is illustrated by the concept of ***Designeering Interaction*** proposed by Huot [Huo13]. Thus, both interactive systems and their users are moving targets, on which knowledge must be regularly reassessed.

1.3.3.1 Studies of designers’ needs

Among the categories of developers, ***designers*** have often been at the heart of needs studies. They are associated with expertise in the design of graphical interfaces, but also less mastery of programming, which makes them more demanding on this point. They are therefore often considered extreme users, because their difficulties in programming are exacerbated compared to programmers of interactive systems. Some work therefore focuses on these users, with the hypothesis that the contributions they draw can be generalized to other contexts.

Thus in 2008, Myers et al. observed in a study of 259 designers, that 86% consider that behaviors are more difficult to prototype than appearance, and that 78% of the time they must collaborate with developers to design interactive behaviors [Mye08]. They suggest the development of tools to explore more complex behaviors than by selecting them from simple menus, and to test several simultaneously. They also call for better integration of coding tools with drawing tools.

In 2009, Grigoreanu et al. observed that little is known about the needs of designers, and little is known about their difficulties with tools such as Adobe Dreamweaver, Adobe Flash, and Microsoft Expression Blend [Gri09]. From a comprehensive study including discussions on mailing lists and user forums, as well as 10 interviews with designers, they extract 20 needs regularly expressed by designers, and classified by perceived importance. Two needs emerge from the others: (i) represent how data, events, and other resources flow in the application, and (ii) ensure that the interactivity ***(feel)*** of the application conforms to the intention of the author. The study is notable for the pragmatism of the needs formulated. All of them can directly be used to suggest new tools, and the authors even include a discussion of converting different requirements into tools.

As part of an industrial maritime surveillance project, Letondal et al. studied the needs of designers related to the use of model-driven architectures [Let14]. They thus seek to adapt and improve the models they use, and their interactive application engineering process in general. Using in-situ interviews and participatory design, they conclude on the usefulness of models as a support for exchanges between designers and engineers, although they are not used to generate interfaces. They also suggest better sharing of representations, in order to improve collaboration between teams.

Among this work, we note the importance given to the transparency of applications. This point ties in with Norman's "Gulf of Evaluation" mentioned above [Nor88]. The lack of information about the internal state of applications forces designers to guess from what they observe. Finally, all the work suggests the development of new tools to facilitate programming activity by designers. In this sense, they are in line with work on the usability of software libraries.

1.3.3.2 Needs for programming tools

It is well known that programming interactions with human users is different from programming algorithms [Bea08]. Since computing was born with scientific computing, the first programming languages were designed for computing. Thus, the basic concepts of languages used today come from calculation (functions, arithmetic expressions, data structures).
These languages often relegate interaction programming to a secondary level, and naturally make it difficult [Mye94] Research initiatives have therefore targeted programming languages and tools, in order to improve interaction support.

Among these, in 2010 Letondal et al. focused on the usability needs of programming tools [Let10]. They recognize that the numerous tools, languages, architectural patterns and models proposed to support interaction have not had significant influences. They therefore drew up a list of 12 requirements for this work, constructed from the study of around fifty

previous work. These requirements are formulated as high-level discussions of the goals that notable works have shared. They aim to guide future work to better match the needs of interaction programmers.

In the context of the design of the ***djnn framework,*** Chatty and Conversy sought to characterize future languages for designers of interactive systems [Cha14]. Based on three research themes in systems engineering, they formulate six research directions, to guide the evolution of languages adapted to interaction: extend functionalities, unify concepts, formalize concepts, extend language concepts (their semantics), extend language notations (their representation), and consolidate past results. This work accompanied the evolution of djnn, and led to the creation of the Smala programming language [Mag18].
Among these research directions, functionality extension refers to contributions by tools that many other works have proposed. The study of programming concepts, and in particular their links to programming languages, is an original approach, which is very well illustrated by the ***reification*** of the concept of binding [Cha07] into Smala language operators (ÿ and ÿ).
We provide another illustration of this approach in this thesis, for animations. Finally, the consolidation of existing results is an interesting approach, which deserves to be supported.
As the IT ecosystem evolves rapidly, in-depth work may struggle to gain traction when it takes time to mature. This phenomenon can explain the relative inertia of interface architectures, which have evolved little in recent decades.

1.3.3.3 Limits and opportunities for research into programming needs

Most of the work studying the programming needs of HMIs is based on populations external to the research teams. They are designers, engineers, or even end users. Few specifically observe researchers designing new forms of interaction. However, this category of programmers also struggles with the complexity of programming tools. Furthermore, they are extreme users, like designers, since they use interaction frameworks beyond the intended use cases. The research context is distinguished from other usage contexts by the following points:

- short iteration times, time to assess whether a project has a chance of
- succeeding a population trained in IT, with a good level of programming
- the frequent combination of multiple software libraries, not necessarily designed to coexist

- “advanced” use of frameworks outside the scope of the intended and recommended use

The last point is characteristic of the research approach, and is an important point that we still need to clarify in this chapter. These are the **needs of researchers, in the specific context of programming new interaction techniques.** Many works presented here have sought to understand the prototyping activity as a whole, in the sequence of its stages, and where difficulties are encountered. However, few studies look at the specific problems encountered by users of interaction frameworks at a lower level.
The consequence is that we can today offer tools to ***support*** the development of research prototypes, but not call into question the tools already used, due to lack of knowledge.

how they should be improved. This results in an accumulation of tools, which in spite of themselves contribute to the complexity of HMI programming activity. The aim of the following studies is therefore to fill this gap.

Many previous works have recognized the difficulty of programming interaction, and have proposed research directions to improve the situation. However, these proposals are often high-level, and it is difficult to capitalize on them into pragmatic changes to development tools. For example, in the six lines of research supported by Chatty and Conversy, the “concepts” mentioned are difficult to link to examples of possible achievements. They are mainly aimed at an audience of HCI researchers, who can understand the scope of these concepts in the context of interaction programming. They also allow them to explain and justify their work in progress (see section 2.1.5.1). This problem echoes work on the usability of software libraries, which contributes a lot through new tools, and often recommends these types of contributions. We have come to wonder if there are **other ways to contribute to the programming of interaction techniques,** other than through new development tools. This question mainly fuels the discussions at the end of this chapter, as well as the conclusion of this manuscript.

In the following sections, we present two studies which aimed to acquire a more detailed understanding of the practical needs of programmers, for the development of research prototypes in HMI.

1.4 Interviews with researchers in the field

The first study during this thesis work consisted of observing and questioning researchers on the programming of interaction techniques. Firstly, we needed an overview of what the development of interaction techniques involves in a research context, to identify areas for improvement. Second, we needed to list the problems that researchers encounter most, which would help us explain why implementing interaction techniques is so difficult. Indeed, few studies have specifically studied the population of HMI researchers. Additionally, while many have focused on the activity of prototyping in general, we are specifically targeting the frameworks used by researchers. This type of study is new to our knowledge, and we hope to contribute to the improvement of frameworks in research contexts. Thus, we first chose to conduct interviews ***with*** HMI researchers. We first detail the interview study protocol, then present the results, and finally analyze them.

1.4.1 Study protocol

This study is based on the principles of grounded theory ***[Gla17],*** which consists of collecting phenomenological data, without a priori, and looking for recurring patterns and “meaning”. We therefore conducted interviews with designers of interaction techniques, seeking to understand the design activity as a whole.

1.4.1.1 Selection of participants

Our study is placed in the context of prototyping new interaction techniques. We therefore looked for participants developing or having developed new interaction techniques. As a selection criterion we asked potential participants if they had felt limited by their programming tools. This criterion was not strictly necessary but favored the selection of candidates, because we expected them to have needs to state.
We recruited researchers from among the members of our research team, with priority given to those with the most experience. This proximity may have created a “proximity bias”, where participants would exaggerate certain problems to satisfy us. However, the majority of interview participants being international HMI experts, they are familiar with this type of problem, and we consider that they were able to remain objective so as not to bias the results of the study. In total, 9 interviews were carried out (6 researchers, an engineer, a doctoral student, and a Master's student), for an average programming experience of 14 years.

1.4.1.2 Interview plan

During each interview, we reviewed 2~3 projects of the participant. To facilitate the choice, we offered a priori a selection of projects, chosen from those referenced on the personal page of each participant. We particularly asked for projects for which they had felt like they were “hacking”, or had been limited by their tools. The interview plan was designed as a guide through the design cycle of each project and the different activities implemented, focusing on the problems encountered related to the use of interaction libraries. This plan accompanied an exploratory analysis, intended to abstract the problems from the framework of a particular interaction library. However, we also sought to understand how programmers perceive their activity, as part of the design of new interaction techniques. The plan therefore also included asking each participant to define the notions of ***hacking*** (or tinkering) and ***low level*** in their projects (see the plan in the appendix).

The interviews were semi-structured [Wen01], to leave it to the interrogator to explore a relevant aspect in more depth if necessary. The plan addressed the following points: choice of framework, nature and number of code rewrites, nature of the initial prototype, initial ambitions and actual implementation, perception of ***hacking*** and low level, perception of code cleanliness, resources for learning of the framework, and sharing code for a community. This plan was designed as a guide through the different “stages” of a project, starting with the choice of tools, then the first prototypes, and ending with the distribution of the work. It thus supported an exploratory study of programming activity, and more particularly the use of frameworks in a research context. We used the critical incident method ***(Critical Incident Technique)*** [Che98] to help participants remember each project as well as their main difficulties.

In addition, we had a number of hypotheses to test, derived from personal experience, for which we were careful to avoid encouraging positive or negative responses from participants:

- that initial ambitions are often lowered due to framework limitations that participants would
- consider their code dirtier if they thought they had hacked ***that*** participants would prefer
- better documentation to a better API that participants would prefer to share their
- code as an independent project rather than integrating it into an existing project

In practice, participants often spontaneously shared with us the ***techniques*** (or ***strategies)*** they had used to overcome each problem. Although the plan was not initially designed to collect these techniques, we found them interesting, and considered that their knowledge could contribute to the improvement of programming tools. We therefore studied and classified them during this work.

1.4.1.3 Conduct of the study

For each interview, we asked the participant to use their work computer (laptop or desktop), to help recall the project, and show code when appropriate. Interviews took place in participants' offices when they were unoccupied, otherwise we moved to a quiet room to reduce interference to the audio recording.
Each interview was conducted by a single speaker and a single participant.

All interviews were conducted in French (mother tongue of 8 participants), then transcribed and analyzed in this language, so as not to introduce bias linked to a possible intermediate translation into English. In addition, the use of colloquial language allowed us to observe the ***nuances*** in the participants' comments, and to identify with what importance the different problems were perceived.

1.4.1.4 Data collection

All interviews were recorded, using a smartphone microphone application.
The interviews lasted more than an hour on average, for a total of 9.6 hours of audio files. We then transcribed them in full, to facilitate analysis, and to ensure transparency by precisely citing the origin of each observation. These transcriptions had to be done manually due to a lack of adequate transcription tools. Indeed, the recorded voices lacked clarity and were disturbed by background noise, some participants spoke quickly, and their sentences were sometimes fragmentary. Notes taken on paper during each interview facilitated the transcription work, to remove ambiguities when the audio was difficult to interpret.

1.4.2 Data analysis

We focused our analysis on the use of programming tools to prototype new interaction techniques. Our goal was to generally understand ***how*** participants used the tools available to them. We sought in particular to identify the difficulties they had experienced, and the limiting aspects in their tools. Although the interviews are oriented towards the problems encountered, in practice the participants often described the

means implemented to resolve them. In doing so, they detailed their “tips”, as well as the tools they had found useful to implement them. It seemed to us that these means could be useful knowledge for the HCI community.

We therefore divided the analyzes into four types of observations: problems, needs, utilities, and strategies. In addition to these types, there are three themes that we sought to clarify with the help of the participants: low level, hacking , and code cleanliness. The data analysis was carried out firstly by extracting observations of each type from the transcriptions, and secondly by synthesizing them.

1.4.2.1 Extraction of observations

During this first step, we went through the transcripts of each interview, and extracted observations of each type, into seven files. Each observation is preceded by a participant number, and a line number in the corresponding transcription — for example P2:L1. Participant 1 (pilot) is not included in this study, as the audio file was not correctly recorded and subsequently lost.

Problems were selected by noting what participants described that they could not do, or perceived limitations . For example, P10:L80 describes " ***The interaction mode works, except I didn't have control over the scrolling speed ."*** As important problems have often been formulated several times with different terms, we have ignored multiple occurrences.

The **requirements** were mainly selected from explicit requests from participants, for example P4:L206 “ there ***would be a thing where the system knows for each method what is the percentage of use by developers in general ”.*** We also interpreted them based on what the participants were trying to achieve, before being confronted with problems. For example, P3:L126 “ And ***there are frequencies that I struggled to do. Slow frequencies are difficult, because [...] »*** gave rise to the need ***“Control a hardware frequency counter”.*** Although the above observation could also be interpreted as ***“Set a slower hardware counter”,*** we still chose the most general need to make it easier to summarize afterwards.

Utilities are the resources and libraries used by participants ***in addition to*** interaction libraries, for the prototyping of new interaction techniques . We noted libraries, such as P2:L34 “ Accessibility ***API to intercept command activations ”.*** We also noted design patterns, like P6:L47 “ And ***then we also used decorator, it wasn't there, it was me who added it [...] ”.*** Later in this analysis, we classify the different types of useful resources listed in this study.

Strategies are the actions taken after each stated problem or limitation . As with utilities, this type of observation is subject to interpretation. We generally noted the actions implementing the frameworks, which went against recommended practices and
documented. Further we identify recurring patterns and classify them.

The points on **low level,** hacking , and **cleanliness of the code,** were noted from the answers to the dedicated questions. Due to the semi-directive nature of the interviews, these questions were secondary and were not systematically asked. They mainly serve us to clarify the different aspects of the activity of designing interaction techniques.

1.4.2.2 Summary of observations

At the end of the observation extraction phase, we went from 2519 lines of transcriptions in 9 files, to 276 lines of observations in 7 files. At this level the observations are factual, it was necessary to group them into categories, in order to then be able to synthesize them. We therefore looked for recurring patterns between observations, and classified the four types of observations in tables inspired by [Dua12], focusing on the relative importance between items rather than their prevalence in the group of researchers studied. In each table, the number of participants who were affected by each point is indicated, as well as the total number of observations relating to it. Each point is illustrated by one of the observations it synthesizes.
Finally, it must be remembered that the number of observations is not the number of times each type of observation has been mentioned, but rather the number of ***unique observations,*** hence the low numbers in appearance.

For the **issues** (Table 1), we formed a first level of categories from the different engineering stages of a software library: feature definition, API, documentation, scalability, bug fixing, and internal architecture. Within these categories, we classified the observations into 25 subcategories, forming the ***types of problems*** encountered by the people we interviewed.

For our **purposes,** after extracting the observations it appeared to us that the problems were strongly correlated. For example, to the problem ***"P2: The tooltip is a static object in the framework, which can only be instantiated once, and therefore cannot be used simultaneously in several places"*** corresponds to the need ***"P2: Display a text next to it of several buttons, in the style of tooltips".*** As we identified more problems than needs (mainly thanks to the critical incident method), we chose to focus on the former.

For **utilities** (Table 2), we classified the observations according to what the programming tools allow, using the first level of problem type categories. Given that only 28 observations were collected, this classification is not intended to be exhaustive, but to provide suggestions for tools to help with the prototyping of interaction techniques.

For **strategies** (Table 3), we formulated initial categories based on action verbs (e.g., intercept, reproduce, divert, or combine). We then refined them and grouped them into three groups (Obtaining data, Interacting with other libraries/tools, and Opportunistic use), to make them easier to read in the table.

Problems	participants	observations
Features		
Insufficiently deterministic/specified behavior — ***P3: Sometimes the framework lets presses pass without release, sometimes releases without press***	4	6
Absence of a useful functionality for prototyping — ***P7: diff does not support displacements***	4	5
Bad engineering choice complicating the work — ***P3: The clock on Arduino is managed in software, the making it sensitive to user slowdowns***	4	5
Mismatch between supply and demand — ***P6: Apple trackpads give relative displacement rather that absolute***	3	3
API		
Inconsistent/illogical API — ***P7: A long press on a keyboard arrow results in a sequence of events, rather than an event with duration***	3	4
API insufficiently controllable — ***P10: Inability to control scroll speed during a drag***	3	6
Hidden/prohibited functions/data for the programmer — ***P2: Some system events are not observable by applications (e.g. active corners)***	5	6
API providing too much data — ***P4: Windows API data structures are "a bit barbaric", it it's difficult to know how to use them***	1	1
API too complex to use — ***P4: The GridBagLayout is too complex to be usable***	1	2
Documentation		
Lack of context and examples — ***P5: Lack of clear documentation to create a custom event for Qt***	3	6
Fragmented/parceled documentation — ***P4: The information is "fragmented", we collect it gradually from different sources***	2	2
Insufficient/passive documentation tools — ***P4: From the name of a method, "ok, how do I do it"***	2	3
Undocumented Significant Behavior — ***P9: Vicon does not document all of its payload data formats***	3	4
Research requiring too much investment — ***P5: Extend signals/ slots to custom events a lot of work***	3	5
Limitations and inconsistencies		
Scale change problem — ***P2: The tooltip is a static object in the framework, which cannot be instantiated only once, and therefore cannot be used simultaneously in several places***	4	4
Artificial (coded) limitation of the framework — ***P2: The design of a button is clamped to the terminals of the button***	2	2
Inconsistent behavior over time (versions) — ***P8: Apple abandoning backwards compatibility with l'API Carbon***	2	4
Inconsistent functionality between Systems — ***P7: Using scrolling (wheel) in Java is difficult between Windows and Mac***	2	2
bugs and slowdowns		
Documented function giving a bad result — ***P4: Under Windows, the pixel density given by the system functions is wrong***	2	2
Non-deterministic error/crash — ***P5: Qt network socket callbacks don't work all the time***	1	1
Unnecessarily/abnormally slow behavior — ***P5: Using a dedicated thread to read a serial port fait lager Qt***	4	5
Performance-intensive instrumentation — ***P5: For heavy interactive applications, the debugger and Valgrind requires too much effort and machine resources***	1	1
Internal aspects		
Unanticipated interference of internal behavior — ***P3: Arduino preconfigured timers interfere with user timers***	2	3
Lack of storage of execution traces — ***P2: A command trigger does not retain history activations that led to its occurrence***	1	1
Internal complexity complicating introspection — ***P2: Two elements activating the same command do so sometimes with indirections***	1	1

Table 1: Different types of problems observed during the interviews

Utilities	participants	observations
Tool to access private data — ***P2: Accessibility API to know which command is executed***	3	3
Extensible class or protocol of the framework — ***P9: The OSC protocol, to add custom events to an existing flow***	6	8
Useful implicit behavior — ***P2: Automatic animation between states in CALayer***	3	3
Debugging/visualization tool — ***P3: printf to debug your text-based communication protocol***	1	1
Design/engineering pattern — ***P8: Naming conventions, to save time converging on its own rules***	5	6
Active or well-designed documentation — ***P4: Method name completion to help discover an API***	3	4

Table 2: Different types of utilities noted during the interviews

Strategies	participants	observations
Obtaining data		
Instrumentation/Intrusion into an existing object/application — ***P2: Code injection to instrument the order executions and determine their triggers***	5	5
Access to private or non-exposed data/functions — ***P3: Writing assembly code to access data alternative modes of timers on Arduino***	5	5
Reconstruction of data from previous/raw states — ***P4: Decoding the HID descriptor by hand***	5	7
Retrieval/combination of information from different sources — ***P4: Reading the different system APIs to find the one that returns the low-level information we need***	1	1
Interaction with other libraries/tools		
Functional reimplementation of an existing widget/mechanism — ***P5: Design your own widgets to be able to make "weird" shapes***	7	9
Inhibiting/replacing existing behavior — ***P6: Filtering system events to block and move the cursor manually***	3	3
Augmentation/modification of an existing mechanism — ***P7: Modification of the progress bar widget for refresh more often***	4	4
Overlaying an overlay on an existing application — ***P2: Using a Canvas overlay on the home screen of an Android tablet to intercept touch presses***	3	3
Extension using dedicated framework tools — ***P5: Creation of custom events for Qt, with new metadata***	1	1
Black box reverse engineering (knowledge extraction) — ***P8: Determining the network MTU by adaptive decrease and statistics***	5	8
Modifying a tool's environment to alter it — ***P7: Modify your PC's clock to continue use the framework***	4	4
Opportunistic use		
Dummy reproduction of existing application — ***P5: Reproduction of a dummy Google Maps application***	2	3
Duplication to bypass a number limit — ***P9: Connect 4 mice to 4 computers and send the events to a main program, to manage 4 cursors in a Swing application***	2	3
Diverting a tool/parameter outside its intended scope — ***P2: Adding spaces in menu names to be able to unroll them all without them overlapping***	4	5
Introduction of a new API paradigm — ***P3: Development of a parser + generic protocol flexible text-based communication (for debugging), with "who is connected to this port" query and commands expandable***	3	3
Complementary use of a low-level tool/source — ***P4: Recovery of raw information on the screen in the registry***	4	5
Combination of tools to perform a more complex task — ***P4: Combination of several tool managers placement in nested widgets to achieve impossible placement with each***	2	2

Table 3: Different types of strategies observed during the interviews

1.4.3 Study results

Now, it is a question of discussing and interpreting these results and observations in order to draw possible causes and implications. We address issues, utilities, and strategies separately, then focus on clarifying low-level, hacking , and its connection to code cleanliness.

1.4.3.1 Causes of problems encountered when using a framework

In the classification of Table 1, we notice that many problems can be explained by a poor mutual understanding between users and framework designers. First, the issues ***Insufficiently Deterministic/ Specified Behavior, Insufficiently Controllable API,*** and ***Inconsistent Functionality Across Systems*** show that participants used features in conditions that were not anticipated (by the framework designers), and probably not tested. Additionally, with the ***Poor Engineering Choice Complicating Work*** and ***Scaling Problem issues,*** participants used more framework resources than expected, which was not anticipated. There is therefore a gap between the needs met by the frameworks and the needs of the participants. However, we cannot conclude that this discrepancy concerns the ***perception*** of needs by framework designers, because we can think that they simply do not take them into account.

Then, the problems ***Research requiring too much investment, Insufficient/ passive documentation tools,*** and ***API too complex to use*** show us that some participants are not ready to devote as much time and effort as necessary, to use a framework as its developers intended. There is thus a gap between developers' assumptions about users (as well as their user experience) and reality.

Finally, through the problems ***Mismatch between supply and demand*** and ***inconsistent/ illogical API*** we notice a gap between what the participants wanted to do, and what the libraries could do. In this case, it's not just a matter of considering the number of features each library has, as participants' needs may have been influenced by what they thought they could do with it. It is also a lack of consistency in the interface of the frameworks, which misleads users in what they want to do there. In addition, we consider that the problem ***Lack of context and examples*** shows that the conceptual model of the framework is not sufficiently explained for advanced use in our study context. It is possible that developers consider their documentation clearer than it really is because they have not asked affected users. This point thus relates to Norman’s “Gulf of Execution” [Nor88].

1.4.3.2 Artifacts contributing to the prototyping activity

Among the programming tools that have been classified as utilities (Table 2), we note that most relate to unifying conventions and principles, which users can appropriate. For example, in the ***framework's extensible class or protocol case,*** participants benefit from an explicitly specified interface. Such an interface is useful for framework developers, who know precisely what types of subclasses or data will go there.

attach, unforeseen cases being excluded. It is also useful for users of the framework, who discover how to communicate with the framework, in a complete and unambiguous way. In the ***Implicitly Useful Behavior case,*** these are conventions of the framework, explicitly defined or not, but which are normally applied consistently throughout the framework. For example, for ***P2: Automatic animation between states in CALayer,*** the participant was able to consider that any desired animation could be expressed by a simple variable assignment, rather than having to check case by case how to animate each property.

Such conventions are essential in a framework. They form the elements of the ***language*** with which users express interfaces and interactions. When they are sufficiently well specified, it is possible to extrapolate them into new situations, knowing how they will apply there. They therefore allow researchers to appropriate the framework, or even divert it to express new uses. We illustrate this point in section 2.4, with the naming convention for widget properties.

1.4.3.3 Interpretations of the types of strategies observed

In Table 3, many strategies are similar to the practice of ***mashup*** development mentioned in the state of the art. In particular, ***Diverting a tool/ parameter outside its intended scope, Combining tools to perform a more complex task, Superimposing an overlay on an existing application,*** and ***Modifying the environment of a tool for to alter.*** Participants thus abstract the functioning of the different tools and functions, to combine and divert them. In doing so, they can possibly construct a simplified approximate mental model of a component, in order to facilitate its appropriation - cf. ***Reverse engineering (knowledge extraction) black box*** strategy . For example, for ***P8: Determination of the network MTU by adaptive reduction and statistics,*** the participant was able to reduce complex behavior (the maximum size of packets that can pass through the Internet varies depending on the routes taken), to simple data to memorize. (a maximum value of this size).

Furthermore, we observe that participants often took the opportunity to reuse their own knowledge, in particular with the strategies ***Introduction of a new API paradigm, Duplication to circumvent a number limit,*** and ***Functional reimplementation of a widget / existing mechanism.*** They thus work in familiar territory, often at the cost of deviating from the framework's recommended practices. With this observation, we consider that frameworks should facilitate their reappropriation by programmers, rather than seeking to constrain a single way of programming. Here we join the Beaudouin-Lafon Instrumental Interaction [Bea00], and in particular the principles of ***reification, polymorphism,*** and ***reuse,*** which are very relevant in our context [Bea00].

Finally, we note that with the strategies ***Inhibition/ replacement of an existing behavior, Functional reimplementation of an existing widget/ mechanism,*** and ***Dummy reproduction of an existing application,*** the participants brought the system into a controlled state before developing above. In this case, it involves ***resetting*** part of the framework, to eliminate any interference from existing behaviors. In the case of reimplementation of a widget, we start from a base

neutral, which does not display or react to input events. It therefore seems important to us to minimize existing behaviors when designing new widgets, which we illustrate with the composition mechanism used for Polyphony, in chapter 3 .

1.4.3.4 Low-level clarification

During the interviews, we asked participants to define what they considered to be ***low-level*** in their project — in their code or in the tools they used. Indeed, the term “low level” often comes up in the field of advanced application development, and sometimes has a negative connotation, synonymous with “source of difficulties”. Its clarification will therefore allow us to provide a new point of view on the difficulties encountered by developers of new interaction techniques, but also why they choose to turn to the low level during their projects.

Low level does not have a definition in the dictionary of the French academy (denoted by the absence of a hyphen between the two words). However, it is used as an adverbial phrase (working ***at low level),*** adjective phrase (a ***low-level library),*** and noun phrase (characterizing ***the low level).*** The low level is rarely used alone and is used in Computer Science to characterize types of software artifacts. We often talk about a ***low-level language,*** a ***low-level communication protocol,*** or a ***low-level library.***

We observe that the low level is almost systematically associated with a layered organization, in which high-level layers ***depend*** on low-level layers. Indeed, programming languages are organized in layers separated by compilation (translation) phases — for example Python to C, then C to Assembler, and Assembler to Machine Code. Low level here refers to languages **closer to the machine.** By extension, we say that a language is lower level than another if it allows operations closer to what machine code allows, even if the two languages do not belong to the layers of the same compilation chain (e.g. C++ is higher level than C but compiles directly to Assembler).

In the case of communication protocols, the layers are separated by relationships of including high-level data in low-level communication channels. We generally talk about the simplified OSI model in 5 layers — physical transmission (cables), link (MAC), network (IP), transport (TCP and UDP), and application (HTTP and FTP). Here, the low level refers to more fundamental protocols, which inherit few characteristics from other protocols. They are therefore generally **less reliable,** because they accumulate fewer error robustness and reception control features. But they have **better latency,** because it accumulates and increases with each new layer of communication.

Finally, in the case of software libraries, the layers are separated by dependency relationships. These relationships include the inclusion and compilation discussed for protocols and languages, so the characteristics listed above still apply here. Libraries exchange data between layers, and can optionally transform it. In the case of input data that interests us for interaction techniques, low-level libraries handle **more raw** data since they are closer to the hardware and less transformed.

These libraries are also **faster,** since they execute less code from other libraries. However, they are also **less powerful,** meaning that for an equivalent code size they perform less complex behavior.

During the analysis of the interviews we noted 25 observations linked to the low level, some of which provide new characteristics. The low level thus allows access to **more data,** because it has been less blocked or ignored between layers — ***P10: More information, more precise, not modified downstream.*** The functions and behaviors performed are **less biased/stereotypical** — ***P8: "When you want to do things that are a little off the beaten track ."***
Low-level libraries require more development **efforts** — ***P5: No widget or library available to meet a specific need, you have to reimplement it yourself.*** They are also sometimes **less documented** in the operating system, to compensate for security flaws or force the use of higher layers - ***P5: Hidden functions of the system, reserved for its developers, not intended for external developers.*** They are more prone to **technical limitations** since low-level libraries often target specific hardware and usage contexts — ***P8: Related to technical limitations.*** Finally, they are **more sensitive** to misuse, by a ***butterfly effect*** of dependencies between libraries - ***P7: More serious consequences if we do anything.***

From the points highlighted in bold, we conclude that programmers are turning to low-level libraries for:

- access minimally filtered, minimally transformed data, and quickly after their production
- use hidden functions of the system, less constrained towards a style of results, and faster

However, they are faced with:

- heavier code, and linked more closely to hardware or context of use a greater
- investment in time and energy to learn and develop with the low level

- less reliability in the event of unforeseen events, and more serious consequences of misuse

1.4.3.5 Hacking and code cleanliness

During the interviews we asked participants how clean they considered their code to be, on a scale of 1 (very dirty) to 5 (very clean), and to explain why. We also asked them to indicate whether they considered their work to be ***hacking,*** then to define this notion in their work. The action of "hacking" a system is sometimes used in the context of prototyping interactive systems. It is often associated with advanced use, against established practice, however the term has been used in many contexts, and is now vague. Just like the low level, clarifying hacking will allow us to better understand the needs of interaction programmers, as well as the problems they face. In addition, we wanted to evaluate the perception of dirty code, and its link to hacking activity.

As a rather recent anglicism, ***hacking*** does not have a definition in the dictionary of the French academy. We start with the definition given on the collaborative encyclopedia Wikipedia [Wik19] : " Hacking can be ***similar to computer hacking. In this case, it is a practice aimed at a "discreet" exchange of illegal or personal information. This practice, established by hackers, appeared with the first home computers. Hacking can also be defined as a set of techniques making it possible to exploit the flaws and vulnerabilities of an element or a group of material or human elements .*** This definition shows that hacking is primarily understood as computer hacking, but this is not our study context, although the end of the definition attempts to generalize the use of the word. It nevertheless brings the notions **of interaction with an existing system** and of **transgression.** The Merriam-Webster English Dictionary gives us more information on the word ***hack :***

hack

noun 1 : a tool for rough cutting or chopping : an implement for hacking 2 : nick, notch 3 : a short dry cough 4 : a rough or irregular cutting stroke : a hacking blow 5 : restriction to quarters as punishment for naval officers — usually used in the phrase under hack ***6a : a usually creatively improvised solution to a computer hardware or programming limitation 6b : an act or instance of gaining or attempting to gain illegal access to a computer or computer system 6c : a clever tip or technique for doing or improving something***

It is definitions 6a and 6c that interest us and shed light on the practice of hacking. They emphasize the notions of **creativity, improvisation, problem solving,** and tricks .

Then, during the analysis of the interviews we noted 12 observations on hacking activity. Thus, it can involve **taking advantage of resources** without being limited to a particular tool — ***P7: Use of existing resources as much as possible.*** It sometimes consists of a **change of environment** or context — ***P9: Using a library for something for which it is not made.*** Finally, it's a **difficult** activity — ***P6: Getting stuck.***

Using these characteristics of hacking activity, we can give its definition:

In the context of programming interaction techniques, hacking ***is the opportunistic and creative resolution of a problem linked to an existing system, by improvisation with any available resource, and implementing tricks deviating from the framework of permitted practices.***

In practice during the interviews, some of the participants told us that they did not consider themselves to be ***hacking*** - even though the descriptions of their activities corresponded very much to the definition that we gave. This can be explained by the less rich meaning in French, synonymous with computer hacking. It is also possible that some participants view their creative process as ***systematic,*** rather than ad hoc and opportunistic.

Regarding the cleanliness of the code perceived by the participants, over 10 projects we noted an average of 3.55 as well as a median of 3.75. The participants therefore consider their code rather

clean in practice. For lack of further observations, and with the low interest of the participants, we cannot conclude on the links between hacking and code cleanliness, and have chosen not to pursue this path.

1.4.3.6 Interpretation of results

At the end of the analysis of the interviews, we obtained two main classifications, of the ***problems*** encountered by the developers of new interaction techniques, and of the ***strategies*** used to solve them. In addition, we clarified the notions of low-level and hacking, which constitute an important part of the working context of these developers. To return to the question **“How can we best support the development of new interaction techniques ?”,** the type of contribution of this thesis chapter is oriented towards highlighting major problems to be resolved, as well as opportunities to innovations in development tools.

However at this level we are not able to assess how important different issues and strategies are relative to each other. The numbers of observations noted in Table 1 and Table 3 represent a small sample of people, selected near our research team. They are therefore not representative of all HCI researchers. Furthermore, the categories were inferred from interviews that did not have the initial objectives of formulating classifications. We did not design the interview plan to systematically explore all participants' problems and strategies. It was therefore necessary to complement this first study to complete and validate our classifications.

1.5 Online questionnaire

Interviews with HMI researchers allowed us to observe the lack of understanding between framework designers and their users, in the context of research. However, frameworks (Qt, Android, JavaFX, Cocoa, HTML, etc.) are well established in programming practices. As we have observed in the state of the art, little work has focused on ***improving*** interaction frameworks, due to lack of knowledge about their limits. We therefore seek here to complete this knowledge, to contribute to the adaptation of existing tools to the research context. We thus hope to contribute to generally reducing the difficulty of programming new interaction techniques. The interviews allowed us to have a first set of observations, which allowed us to identify trends to confirm. We therefore formulate the following research questions:

- **Q1 — What are the main selection criteria for software libraries used to program interactive applications in a research context?**
- **Q2 — What are the problems most limiting the work of researchers with the use of these libraries?**
- **Q3 — What development strategies are mainly used to circumvent the problems encountered in this context?**

1.5.1 Study protocol

To answer these questions, we opted for an online questionnaire in English, which allows us to collect more responses than with local participants, and to reduce the bias of a local research area.

1.5.1.1 Questionnaire outline

The questionnaire was designed to assess different possible answers to our three research questions. Preliminary questions collected the environment in which participants programmed interaction (profession, type of institution, number of collaborators), their expertise in programming interactive applications, and the proportion of their working time spent programming. . These questions were intended to identify the context of research and development of interactive systems, to ultimately guide proposals for future work.
Then, three series of questions evaluated the relevance of different criteria for each of the research questions: importance of the criteria for choosing interaction libraries (Q1), severity of the problems encountered (Q2), and frequency of the types of solutions provided (Q3). The criteria for choosing libraries were initially formulated by our experience in the field, then refined by pilot questionnaires. The selections of problems and solutions were formulated based on the classifications of problems and strategies from the interviews, and refined after the pilot questionnaires.

The questionnaire as presented online is included in the appendix. We opted for multiple choice questions, in order to limit any bias linked to the interpretation of the results, and to possibly manage a larger number of responses. The relevance of the criteria is assessed using 5-level Likert scales, with an additional level for problems making it possible to distinguish problems never encountered from problems already encountered but of zero severity.
For each library choice criterion (Q1) and each type of solution (Q3), we included a free text field so that participants could possibly detail and specify their answers. These fields also allowed us to reorganize the different categories following the pilot questionnaires. Finally, with the criteria for choosing libraries (Q1), we asked participants to indicate the frameworks or toolboxes they used the most, to be able to make a first estimate.

Our main hypothesis was that participants systematically use major frameworks rather than research-based toolkits. We therefore took care to adopt neutral formulations ***("interaction libraries", "frameworks/ toolkits")*** so as not to induce participants to respond with a given type of library, and that they consider any type of software library allowing program interactive applications. Concerning the library choice criteria (Q1), we also wanted to assess the prevalence of subjective criteria (e.g. reputation, personal experience) compared to more directly actionable objective criteria (e.g. performance, quality of documentation). ***The objectivity*** of the different criteria being subject to interpretation, we simply mixed the criteria that we considered objective and subjective.

A first pilot questionnaire was submitted to four participants. The responses allowed us to include new criteria for choosing libraries (Q1). We also removed overly specific problems (Q2) in order to limit the number of criteria to evaluate, as well as the time spent responding. Finally, for problem solution types (Q3), we removed all mentions of factual examples, which participants frequently cited in response details, and which risked biasing results based on participants' experience . Due to the change in library selection criteria, responses to the pilot questionnaire were excluded from the results.

We used the ***open source*** online software Framaforms [Fra19] to create and publish the questionnaire. Due to the presence of fields allowing details of the answers (Q1 and Q3), Framaforms did not allow us to mix the presentation of the different criteria to the participants. For these questions we therefore chose a logical order between the criteria (e.g.
Tool reputation ÿ ***Developer reputation*** ÿ ***User community*** ÿ ***Documentation quality),*** in order to reduce the memory effort to transition between each. We discuss below the probable influence of this limitation on the results.

1.5.1.2 Selection of participants

The participants of this questionnaire should ideally have prototyped new forms of interaction (interaction techniques, interfaces, interactive artifacts) in a research context.
As it is difficult to precisely characterize what qualifies individuals to have sufficient experience in this area, we recruited participants exclusively from the HCI community. The questionnaire was distributed for the first time via the ***chi-Announcements@acm.org*** and éditions@afihm.org mailing lists , in order to reach a maximum number of participants by reducing the number of duplicate emails received. These first lists were then supplemented by distributions to our colleagues' former teams.

L'introduction du questionnaire indiquait clairement le contexte de la recherche, afin de sélectionner spécifiquement des chercheurs : « The ***goal of this survey is to better understand the process of programming new interactive artefacts for research and innovation purposes, and the software libraries used to support this process. We are interested in projects where you reached the limits of libraries (e.g. undocumented needs, accessing private functionalities, interfacing with existing applications, combining with other libraries), for the prototyping and implementation of innovative interactive artefacts (e.g. non-standard interaction techniques, data visualisations, UI toolkits, interactive applicatio***

We collected 32 responses in total, over a period of 2 months. 25 of the participants have their main activity as a researcher, and 27 work for a university, school or public institution. Two thirds of participants say they are ***at least*** advanced. The participant profiles are therefore very relevant to the context of our study. However, we are unable to say whether they are representative of the general proportions of users who prototype interactive applications in a research context.

1.5.2 Analysis and interpretation of results

The aim of the questionnaire was to understand the choices of libraries and their uses to prototype new interaction techniques, then to formulate recommendations for tool designers as well as researchers. We therefore focused our analysis on the aspects of libraries that matter most to researchers when designing new interactions, as well as those aspects that matter less.

We can first note from preliminary questions that two thirds of the participants have less than 40% of their time to program, and that the participants collaborate on average with two other people. In short, the participants have little time and resources to carry out their projects, which is not surprising if we consider that the majority are researchers, for whom it is generally accepted that computer development is a fraction of their activity. This observation justifies the idea that improvements should mainly be sought in programming tools, rather than in user programming methods.

1.5.2.1 Interaction libraries used

Immediately after the preliminary questionnaire questions, we asked participants which frameworks or toolkits they used the most. Several answers could be given, separated by commas. The results are presented in Figure 8. We have represented on the abscissa the names of the libraries cited in the responses, and on the ordinate the numbers of participants having cited each of the libraries. With equal citations, the libraries are listed in alphabetical order. We excluded from the results responses that were clearly not interaction libraries (“Atom” and “Eclipse”).

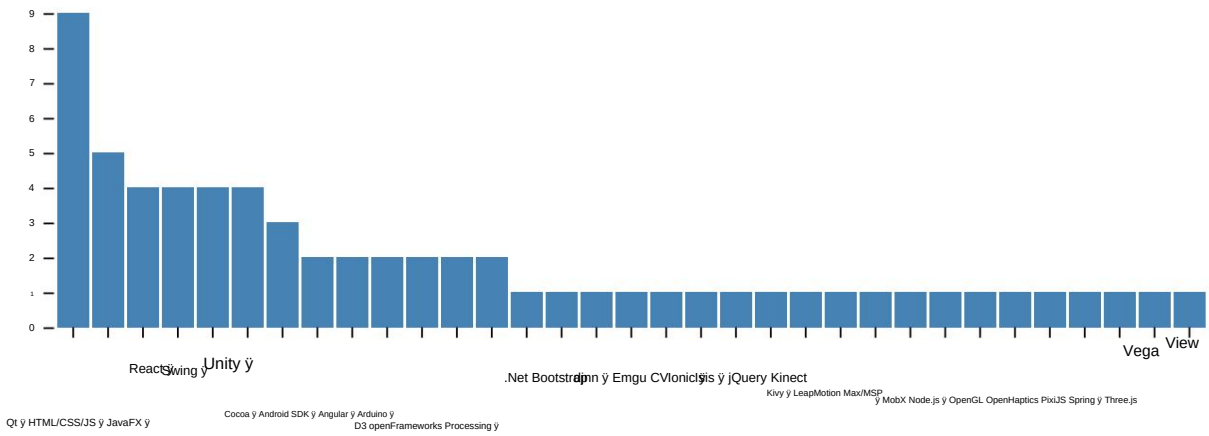

Figure 8: List of libraries cited in the responses, ordered by number of participants. Frameworks are indicated by a star.

We first observe the clear predominance of the Qt framework, used by more than a quarter of the participants. The Swing and JavaFX frameworks (the official successor to Swing) together account for a quarter of the participants, none having mentioned both simultaneously. Then, we can observe that Web frameworks (based on HTML, CSS, and JavaScript technologies) are very widely represented. They represent 14 of the libraries cited, and 37.5% of participants cited at least one. Finally, a large number of libraries were cited only once. The majority of them appeared less than 10 years ago, particularly among web tools. This observation supports the ***opportunistic*** nature of research and prototyping of interaction techniques, in that it relies to a significant extent on recent and emerging technologies.

We have indicated by a star (ÿ) in Figure 8 the libraries that we consider to be frameworks, according to the definition given in section 1.1.4. Thus, 48 of the participants' cumulative citations are frameworks, while 17 are not. We also notice that the most used libraries (on the left of the figure) are mainly frameworks. Finally, among the 33 libraries cited, 4 come from academic research (D3 [Bos11], djnn [Cha16], Max/MSP, and Vega [Sat14]), for a total of 7.6% citations. This observation may be moderated by the fact that D3 and Max/MSP are widely used in specific communities (Computer Aided Visualization and Music), which have not been specifically targeted by our mailing lists.
Additionally, djnn and Vega are relatively new libraries, which may explain their low proportions in citations. However, the low number of research toolkits is a concern in terms of linking researchers' needs to their tools, since it is mainly developers outside the research context who must respond to this context. . Furthermore, this means that libraries made ***by*** researchers ***for*** researchers are not sufficiently successful. The selection criteria can explain this phenomenon.

1.5.2.2 Criteria for choosing interaction libraries

The results of evaluating the importance of the library choice criteria are summarized in Figure 9. We classified them according to the number of voters who answered ***Very important*** or ***Absolutely essential,*** to distinguish the criteria more important than the average. For each, we displayed a 95% confidence interval obtained by the bootstrapping method [Efr79], which corresponds to the probability that a resampling with replacement from the data gives a number of voters included in the interval. When the upper limit of this interval is less than half of the voters, we consider it **very likely** that more than half of the HMI researchers from the community represented by our sample consider the criterion to be at least ***Very important.***

Three criteria emerged: ***Documentation quality*** (24 voters), ***Quality of the API*** (23 voters), and ***User community*** (22 voters). They suggest to us that the main factor influencing the choice of a library is its ease of learning and use by programmers. The ***Compatibility with other tools/ libraries*** criterion is also highlighted by the large number of ***Absolutely essential votes,*** and suggests to us that frameworks are rarely used alone in the context of HMI research.

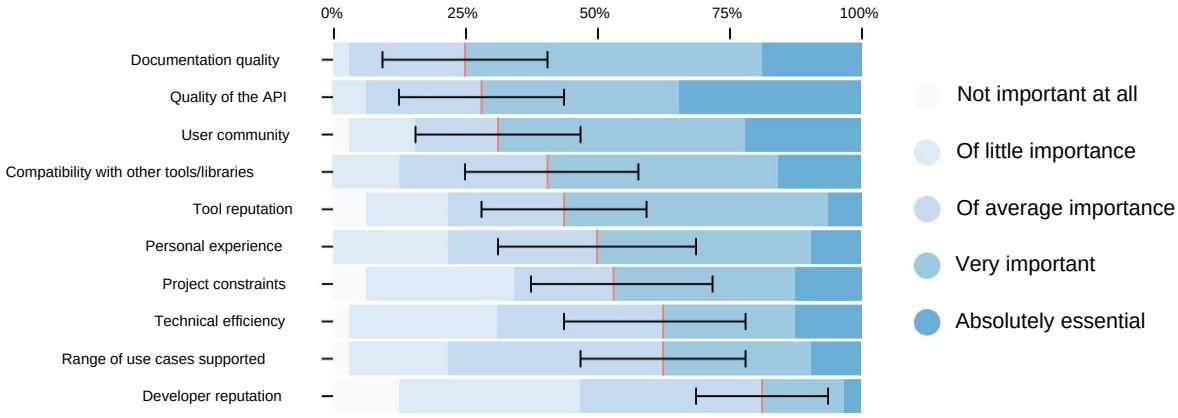

Figure 9: Participants' evaluation of library choice criteria, ranked according to the number of responses at least *Very important,* with 95% confidence intervals for these numbers.

Finally, to assess the possibility for framework designers to influence these criteria, we have divided them into three groups:

- mesurables ***(Documentation quality, Range of use cases, Quality of the API, Technical efficiency)*** mixtes ***(User community, Compatibility with other tools/ libraries)*** subjectifs ***(Tool reputation, Developer reputation, Project constraints, Personal experience)***

Measurable criteria are those for which there are tools to assess their quality, and methods to improve it. For mixed criteria, certain metrics and methods exist but are only partially sufficient. For example, for ***Compatibility with other tools/ libraries,*** we can measure the number of ***plugins*** available for a framework, but it is difficult to evaluate their usability (absence of bugs, implemented features, etc.). For subjective criteria, there are no or few objective measures, and framework designers have no direct control to improve their evaluation on these criteria.

Measurable criteria have on average 55% of votes above ***Of average importance.*** Mixed criteria are at 64%. The subjective criteria are at 43%. The participants in our study therefore tended to favor objective criteria, although a significant difference could not be inferred for the entire population. These results also suggest to us that the two mixed criteria ***User community*** and ***Compatibility with other tools/ libraries*** would benefit from measurement tools and improvement methods in the literature.

1.5.2.3 Problems using interaction libraries

The criticality results of the different problems linked to the use of interaction libraries are summarized in Figure 10. They are classified according to the number of voters having responded at least ***Met, medium trouble,*** threshold from which we consider that the problems took up a sufficient amount of participants' time/ energy. Below each type of problem, thinner bars represent the same results but only including participants calling themselves ***Advanced*** or ***Expert*** (20 of the 32 participants).

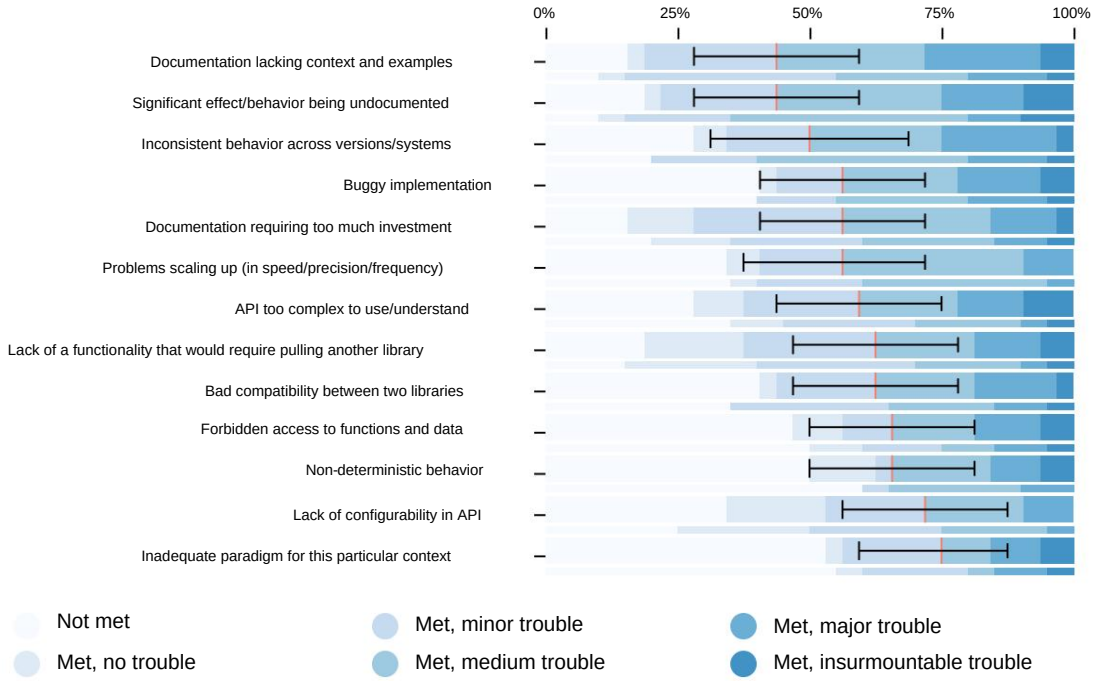

Figure 10: Rating of problems related to the use of interaction libraries, ranked according to the number of responses at minimum *Met, medium trouble,* with 95% confidence intervals for these numbers, and secondary bars representing the same results for participants with at least *Advanced expertise.*

Among the most important issues, those related to **documentation** occupy the 1st, 2nd and 5th positions, which is consistent with the observations on library selection criteria. We do not observe strong variations between the full results and those including only advanced and expert participants. This may be due to the low number of participants, which does not allow us to divide the population in two and observe significant differences. However, we observe that many participants answered ***Not met,*** including among advanced participants.

We have therefore represented in Figure 11 the criticality results of the different problems for the participants who did not answer ***Not met.*** We consider this to be a more reliable estimate of the criticality of the problems for the population of HCI researchers. Indeed, the fact of encountering a particular type of problem depends a lot on the projects on which the participants worked. If a group of participants from the same team responded to our questionnaire, they may thus bias the evaluation in Figure 10, in favor of the problems they encountered as a group. By representing criticality only for people who have encountered the different problems, we limit this bias, at the cost of a lower number of responses per problem. We therefore discarded 33% of the data, which explains the widening of the 95% confidence intervals, shown in the figure.

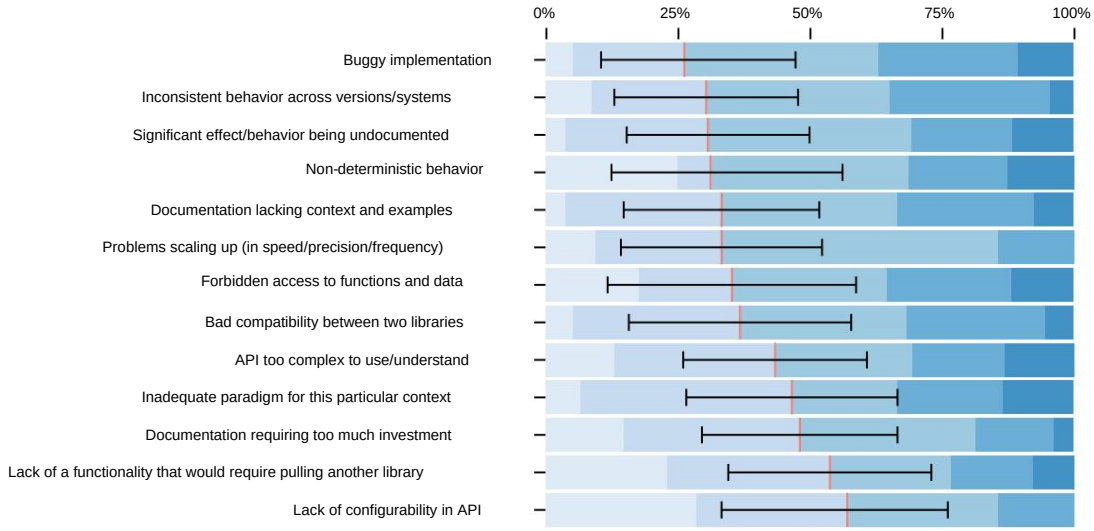

Figure 11: Assessment of the criticality of problems by participants who have already encountered them, with 95% confidence intervals.

Two problems emerge compared to the previous figure, ***Buggy implementation*** and ***Non-deterministic behavior.*** With the ***Significant effect/ behavior being undocumented problem,*** they suggest to us that the most important characteristic of a framework is its **reliability.** That is, a framework should always ***document what it does and do what it documents.*** This point as well as the previous one bring us back to the importance of documentation raised by numerous state-of-the-art works. They encourage us to consider documentation work as important contributions, in that it could have a major impact in reducing the problems encountered.

A certain number of secondary problems were of major criticality for more than a quarter of the participants who encountered them. Thus, ***Inconsistent behavior across versions/ systems*** and ***API too complex to use/ understand*** lead us to consider **consistency** as a secondary characteristic of frameworks. ***Forbidden access to functions and data*** make us consider **transparency** as another important feature. Finally, ***Bad compatibility between two libraries*** and ***Inadequate paradigm for this particular context*** lead us to add **adaptability.** While we can bring consistency and transparency back to documentation work, adaptability is a concept that seems interesting to explore further. It could be associated with the flexibility and dynamic nature of programming frameworks and languages. In practice we explored these points in the prototypes presented in chapters 2 and 3.

1.5.2.4 Strategies for prototyping interaction techniques

In the last part of the questionnaire, we asked participants to rate the prevalence of different programming strategies in their work. The results are presented in Figure 12. One strategy, ***Reimplementing an existing widget/ mechanism [...],*** clearly stands out from the others. In a framework, this programming practice is made possible by the **extension** and **reuse mechanisms.** Then, for the 2nd and 3rd strategies, ***Using accessible raw data to reconstruct/ reinterpret a state[...]*** and ***Using an external mechanism to obtain and process data that is not exposed by an application,*** it is access to hidden data which is important, therefore the **transparency** of

frameworks. Finally, among the following strategies, 4ÿ (Aggregating ***multiple sources of interaction data [...]),*** 5ÿ ***(Using a visual overlay [...])*** and 7ÿ ***(Introducing a dif erent programming model [...])*** relate to the extensibility of frameworks, and the 6ÿ ***(Reverse-engineering a closed tool or library)*** supports their transparency.

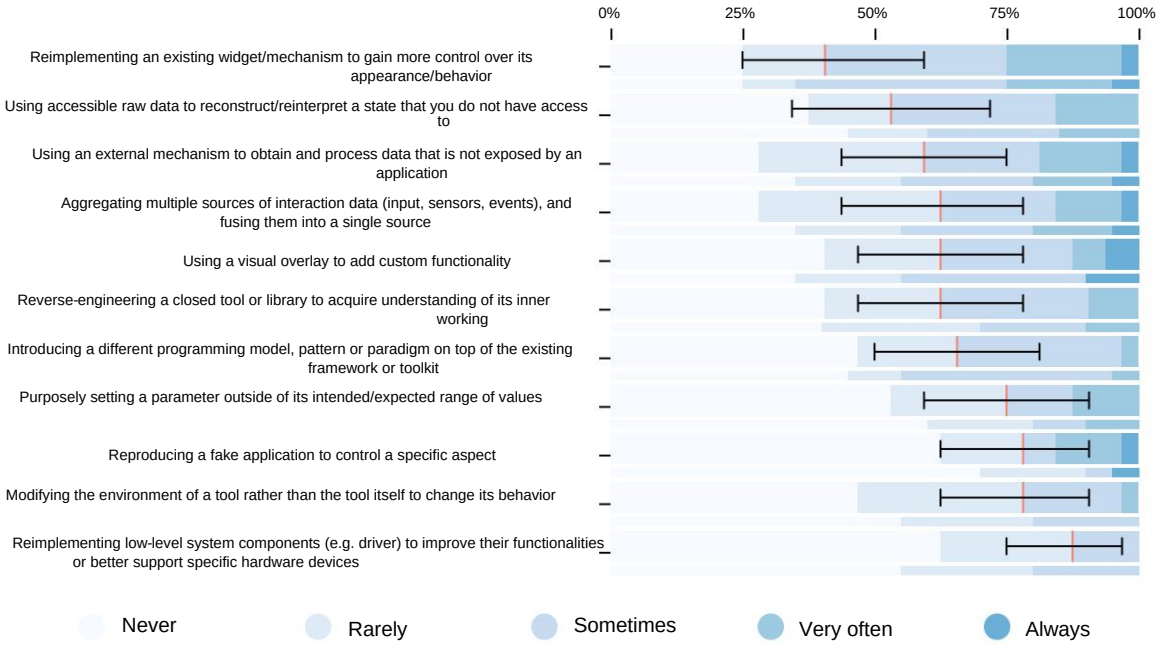

Figure 12: Rating the frequency of use of different programming strategies, ranked according to the number of responses at least *Sometimes,* with 95% confidence intervals for these numbers, and secondary bars representing the same results for participants advances.

We noticed that many participants responded that they had never used each of the strategies. As with the issues assessed in the previous section, it is possible that bias related to subgroups of participants favored certain strategies over others. If two participants who worked on the same projects responded to our questionnaire, their assessments of the prevalence of each strategy could thus be correlated. However here we cannot eliminate this bias by ignoring the ***Never responses,*** because prevalence is a rather objective measure (therefore it can be similar between two participants of the same team), while criticality was mainly subjective (therefore specific to each participant).

1.6 Discussions et implications

Now that we have analyzed the results of the interviews and questionnaires, it is necessary to answer the question: ***What can we do?*** We therefore discuss here the solutions that can be brought to the problems raised in the two studies, and what future work seems most relevant to us.

1.6.1 Suitability of frameworks with HCI research

Are the most widely used interaction frameworks really suitable for prototyping new HMI interaction techniques? This question is worth raising, because the interviews revealed to us that interaction frameworks posed many problems, and the questionnaires confirmed that these problems were encountered by a significant proportion of developers. We noted three contradictions between the ***nature*** of frameworks and that of research.

First of all, supporting advanced uses involves a greater risk of bugs. Indeed, such uses involve writing more complex code, which uses concepts and technologies that are less mastered or not very robust, or even linked to the low level (and all the difficulties that this brings). Here, bugs concern the code written by the researchers, and relate to undesirable behaviors, which thus include problems of lack of precision or latency, although they do not necessarily compromise the execution of the program. On the framework side, supporting advanced features (such as the ability to create new types of widgets) often means writing ***more code,*** either because new functions need to be offered, or to expose an interface to internal functions. . However, many authors evaluate the ratio of bugs per line of code, implicitly defending the hypothesis that ***larger projects contain more bugs*** [May12, McC04]. Therefore, we consider that support for advanced features contributes, at least indirectly, to a greater risk of bugs. This hypothesis balances frameworks between two contradictory objectives: supporting new uses linked to research, and reducing their number of bugs.

Then, the low number of users does not favor the exhaustiveness of the documentation. For the Qt framework most cited in the questionnaire responses, the research domain is not mentioned at all on the main Qt page [Qt19], nor the extension, reuse, and transparency functionalities noted in the questionnaires. These absences are a symptom of the low share of uses of Qt for the purposes of developing research projects, and of the relatively less interest that the developer has in this context. Thus it is likely that developers will devote less energy to providing documentation tailored to the needs of developers conducting research projects. We mainly observe the consequences, namely insufficient documentation regarding the extension, reuse, adaptability and transparency capabilities of the framework.

Finally, the robustness of a framework tends to limit its flexibility to extensions. As new uses develop, data needs evolve. This is what we observe with the data ***transparency*** raised by the questionnaire strategies. However, making previously hidden data available always presents a security risk. For example, allowing keyboard events to be intercepted at the system level also allows passwords to be intercepted. It's a game of cat and mouse, in which any opening of data is often counterbalanced by

a restrictive measure.

Under these conditions, we argue that the use of a framework will be ***inherently*** hindered in a research context. Without being prohibitive, the constraints mentioned have no reason to evolve favorably in the future. The frameworks are therefore not inadequate, but let us not

do not seem to be the most promising solutions. Better solutions would instead be:

- less complex projects, reducing the number of bugs
- dedicated to research, and documented
- to have great flexibility, rather than robustness

1.6.2 Facilitate the development of *ad hoc* applications

Faced with the difficulties encountered with interaction frameworks, one solution would be to facilitate the development of ***ad hoc*** applications (or ***from scratch),*** that is to say without the use of reusable design patterns. It would then be a matter of directly using low-level libraries, for the inputs (SDL, TUIO [Kal05, libpointing [Cas11], etc.) and outputs (OpenGL, Cairo, etc.).

Such an approach would have the advantage of offering greater freedom to adhere to contemporary uses, in terms of appearance of interactive elements and their interactivity. It promotes the evolution of interaction architectures and the questioning of the functionalities to be included, not being subject to historical constraints (lack of memory, speed of graphics rendering on CPU). In addition, the amount of work to produce is much less than that of a framework, because it is not necessary to include functionalities outside of those needed, nor extension mechanisms, or even document for other developers. Finally, due to the reduced scale of such projects, project management tools (CMake, Gradle, Maven) can be spared, helping to reduce learning and programming efforts.

However, the design of an ***ad hoc interactive application,*** even partial, is a complex project, the efforts of which increase with the size of the project. The use of opportunistic design methods adapted to small projects can hit a wall of complexity, if its ambition has not been properly planned in advance. Additionally, the code produced will likely not be reusable for future projects, nor portable to other systems. Finally, such an approach requires a lot of knowledge or experience to be advantageous in terms of time compared to using a proven framework.

In practice during the interviews, three participants explicitly told us that they had freed themselves from the reusable elements of a framework: P2:L114 “ It's ***fast, it's super fast. It depends how you do it, but what I did was I made an application where I draw rectangles, it loads an image, draws rectangles, [...]*** ", P5:L88 “ A ***lot of times yes, I redesign my own widgets, because I need widgets with shapes that are quite... weird, or... particular ”,*** P7:L149 “ But ***I would have just done the simple buttons . I wouldn't have looked for a code, I know how to do that, by hand, from scratch .”*** Furthermore, P4 described his difficulties to us when his ***ad hoc*** project began to take on excessive proportions: P4:L46 “ There ***it is like that, suddenly my application started to grow, grow, grow. And then you realize that you're missing things, so I modified, and it went through the same pipes, but sometimes there were hacks becau***

What could we do? According to our interpretation of the questionnaire results, two problems mainly limit the development of ad hoc applications. First, there is a **lack of documentation on programming ad hoc interfaces.** Indeed, the main references on the subject are interaction architectures (such as MVC, MVP, and PAC), which are designed to be robust, that is to say which operate as expected regardless of the events occurring. 'entrance. He

There are few architectures or documentation dedicated to ***quickly*** implementing an interactive application. To our knowledge, only ImGui [Mur05] meets this need by avoiding the construction of a scene tree, but it is only popular in the field of Video Games.
The second problem is a **lack of documentation and examples on using low-level devices.** Today, it is difficult to use low-level APIs directly, without the use of a framework. These often provide their own internal APIs — for example allowing you to initialize a viewport for OpenGL. Otherwise, we must confront the complexity of the system's functions, which we have detailed in the problem of this chapter. Moreover, the three participants cited who programmed ad hoc applications all did so ***within the framework*** of a framework. Libraries exist to provide direct and simple access to input/output devices, however they are rarely sufficient on their own and do not always coexist very well with others. For example, when using libpointing for the mouse [Cas11], we were not able to capture keyboard inputs with SDL. The multiplication of low-level access libraries therefore does not seem to us to be the most sustainable solution, because they contribute to the layered structure of the APIs, which we also seek to avoid. It would seem preferable to ***document*** the use of the low-level APIs of the different operating systems, and to provide code examples that are quick to implement.

1.6.3 Bringing the Web closer to desktop applications

During our research, we observed that web technologies are increasingly used in the field of desktop applications. For example, this could involve creating a prototype of a web-based desktop application, and using it to carry out a series of controlled experiments. This shift in usage deserves to be underlined. When the Internet initially spread to individuals, HTML and then CSS technologies were used to design websites . A website is a vertically scrolling page of fixed width and variable height, with content (mainly blocks of text and images) automatically arranged from top to bottom and left to right. This operation is inspired by word processing software, but without the separation into pages. Websites are accessed through a browser, which converts HTML/CSS pages into visible, interactive interfaces. On the other hand, ***desktop applications*** are windows of variable width and height, the content of which is freely determined by each application. These are executable programs that do not depend on a browser, but integrate directly into the windowing system of the operating system. Each application therefore manages one or more windows, in which it can draw and capture events

entry.

Historically, websites and desktop applications were considered separate domains, with separate frameworks. However, the emergence of ***Web applications*** (webmails, content management systems, wikis, online games, etc.) has brought the two worlds closer together. On the web side, widgets have emerged to replicate some of the desktop user experience — like buttons, tabs, or menus. On the desktop side, frameworks have reused technologies underlying the Web to benefit from their robustness inherited from numerous developments — such as JavaScript in QML, or CSS in JavaFX. Today, while Web technologies increasingly make it possible to venture into the field of office automation, we argue that future work should support this trend.

Indeed, the movement initiated by the ***World Wide Web Consortium*** (W3C) is an attempt to standardize the internal structure of a framework. Beyond using an architecture like MVC, it involves defining the available classes ***(Document Object Model*** — DOM), style attributes (CSS), drawing instructions **(canvas** and SVG), and much more. others. These standards are thus implemented by several competing browsers, which help to detect ambiguities in specifications due to different behavioral details. This was also the case thanks to the Acid1, Acid2 and Acid3 tests [Hic99], which contributed to standardizing the observable behaviors of browsers. Internally, the DOM specification allows access to part of browsers' private structures — some of which are not standardized and specific to each browser. We thus have a certain ***transparency,*** an essential criterion for the programming strategies classified in the results of the questionnaire. Additionally, web technologies are designed to be portable between operating systems, and it is the role of browsers to handle any differences. Finally, W3C standards are constantly evolving, pushed by browser designers and all volunteering people. They can thus correct any “youthful errors” and adapt to contemporary uses. This is the case for example with HTML tags deprecated with each new version.

Web technologies, however, suffer from some major drawbacks. First of all, it is difficult for browsers to achieve an execution speed and memory footprint comparable to those of desktop frameworks. This is due in part to the use of the dynamic JavaScript language, and also to ***the immense*** complexity of the specifications needed to implement a modern web browser, which complicate optimization efforts. The number and complexity of standards is linked to the significant pressure from many players to add various functionalities [W3C19]. This complexity is further accentuated by the permissiveness of browsers, which historically had to interpret the code as best they could, despite possible errors, in order to facilitate the creation of websites by novices. The complexity of Web technologies nevertheless tends to self-regulate thanks to depreciation mechanisms, and standards that do not achieve success in use are eventually abandoned or replaced. Finally, just like desktop frameworks, the lifespan of an extension accessing internal APIs is limited in time, even when the APIs are standardized. Indeed, with the appearance of overly flexible extension mechanisms, new ***malware*** is developing using them to harm users. Additional security then appears, which ultimately invalidates the previously developed extensions.

This is the case for example with Ajax technology, which made it possible to issue HTTP requests from JavaScript code, but which was then constrained by ***Cross-Origin Resource Sharing technology.***

What could we do? The use of Web technologies to build desktop applications, and prototype interaction techniques linked to this type of applications, requires bridges ***between*** the two domains. The Electron framework, for example, allows you to distribute a web application as a desktop application, without a browser interface [Git13]. However, the API for accessing operating system services is partly specific to Electron, and is not standardized as W3C technology. Standardization of widgets in HTML tags has been attempted in the past, some having been adopted **(button, progress,** or **textarea),** others having been abandoned **(menu, command).** With the recent development of Web Components, it is possible to create new HTML tags to prototype extensions to HTML. It seems to us

therefore timely and crucial to **participate in the ongoing standardization efforts of the W3C,** to facilitate the creation of desktop applications with Web technologies. To inform and guide these developments, it is also necessary to **demonstrate the implementation of Web interaction techniques** (see for example [Roy19]).

In this thesis work, as we will see later, we used Web technologies for their flexibility, which allowed us to explore innovative code syntaxes. Contribution to W3C working groups is a matter for future work.

1.6.4 Links with this thesis work

The two studies presented in this chapter allowed us to observe the low use of HMI toolboxes in practice, therefore the limited interest of such a contribution in this thesis work.
The important thing is the ***knowledge*** you gain from it. So that our work contributes to knowledge on interaction programming, we have always attached it to existing projects or communities, in order to contribute to their development. We have focused in particular on the development of ad hoc interfaces, and the use of Web technologies directly linked to the low level (without browser).

Chapter 2. Interaction Essentials

As we have observed, researchers mainly use proven interaction frameworks to develop new interaction techniques. However, these frameworks are excessively complex, and restrict the freedom of researchers to explore new ideas and deviate from established standards. The main problem is that their developers do not sufficiently take into account the needs of researchers, because these are not their priorities. As these needs are expressed by a relatively small community of users, framework developers would be reluctant to make major architectural changes (as suggested in section 1.5.2). We therefore propose to contribute with work that recognizes the preponderance of frameworks, and integrates in a complementary way into this ecosystem. In connection with the lack of support for low-level interaction (mentioned in section 1.6.2), and inspired by the work on Amulet [Mye97], Smalltalk [Kra88], and Smala [Mag18], we considered the possibility of extending programming languages, as well as frameworks. Through our work, we also sought to study the lack of dissemination of research tools, and to propose answers. Finally, we chose to move towards software contributions (rather than documents as discussed in section 1.6). Indeed, it seemed essential to us, as part of a doctoral thesis, to benefit from practical experience in interaction programming tools, before subsequently contributing to efforts to standardize these tools.

Thus, starting from the problem that the expression and implementation of interaction is often made complex with current tools, in particular because of a multitude of levels of abstraction, we propose and defend the thesis according to which it is **necessary to make more accessible low-level interaction programming, using concepts applicable to programming languages and framework architectures.** We illustrate this thesis with an extension to the Smalltalk language (in section 2.4), as well as a new framework dedicated to the development of ad hoc interfaces (in chapter 3).

We start by formulating ***Interaction Essentials,*** practical recommendations aimed at guiding our accessibility approach. These recommendations were defined based on regular mentions in the state of the art, recurring observations in major interaction frameworks, as well as our preliminary studies. The Interaction Essentials are:

- ***An explicit and flexible orchestration of interactive behaviors,*** where the rules for triggering blocks of code are clearly highlighted, and make it possible to give the initiative for their execution directly to the environment
- ***A minimal interaction environment initialized at the start of any application,*** to reduce initial developer efforts when accessing user inputs and outputs.
- ***Standardized mechanisms and conventions, as well as a language with flexible syntax,*** to ensure the compatibility and extensibility of software libraries to express interaction.

The first section of this chapter is devoted to a state of the art of recommendations for improvements to frameworks and programming languages. The next two sections are dedicated to the first two Interaction Essentials. In the fourth section, we illustrate our simplification approach with the integration of animations into a programming language, and let us deduce a third Essential Interaction.

2.1 State of the art of recommendations for interaction languages and frameworks

The syntax and semantics of interaction programming have been relatively little studied in the fields of Human-Computer Interaction and Software Engineering. They are dependent on the expressiveness given by programming languages, but these are often considered immutable, and interaction is expressed by extension. For example, to connect a ***signal*** to a ***slot*** in Qt, we would write **QObject::connect(&objA, &ClassA::fctA, &objB, &ClassB::fctB).** This syntax is largely based on that of the C++ language: the creation of a connection is done by a **QObject::connect function call,** we refer to the objects by their pointers **&objA** and **&objB,** and we refer to the signals and slots on these objects by function pointers **&ClassA::fctA** and **&ClassB::fctB.** On the other hand, to express a binding (relatively similar concept) in Smala [Mag18], we would write **objA -> objB.** Here, binding is ***integrated*** into the language syntax, which makes it easier for Smala users to express interactive applications, and results in shorter code.

By definition, ***syntax*** is the way words fit together to form sentences. In a programming language, these words are lexemes (or ***tokens),*** which are generally of five types: identifiers (names of variables and functions), language keywords, punctuation (operators and parentheses), literals (numeric constants and character strings), and comments. The syntax of a language is accompanied by grammar rules , which make it possible to verify whether a sequence of lexemes is a valid program or not (without defining what it will produce).

The ***semantics*** of a language is the meaning given to a program (sequences of lexemes), that is to say the effect it will produce at execution. For example, an identifier followed by an opening parenthesis and a closing parenthesis **(f())** forms a function call, which will correspond to the execution in a branching instruction from the execution flow, to the function code.

In the case of a programming framework, the syntax and semantics are always defined within the limits allowed by the base language. It will therefore be impossible to add grammar rules that violate the rules of the language (for example allowing an opening parenthesis without a closing parenthesis). Furthermore, the semantics of a framework will generally be defined by ***extending*** those of the language, because it is often impossible to inhibit basic behavior of the language (such as intercepting all function calls of the program to change its parameters).).

The study and prototyping of new syntaxes and semantics for interaction programming is a difficult activity, which amounts either to modifying an existing language (and accepting its constraints) or to creating a new language. This explains the low number of works having operated at

the interface between frameworks and languages. We present here significant work that has proposed or suggested new forms of interaction programming. They differ from the toolboxes offered in HMI by questioning the concepts underlying programming languages, which requires operating at both levels (language and framework).

2.1.1 The three services of the semantic core essential to the HMI

Fekete proposed three services of the semantic core, which he describes as essential to the HCI, for the HMI conference [Fek96], then during the days of the CNRS Programming Research Group [Fek96]. These points address the “semantic core” of interactive systems, which can be compared to programming languages. They designate functionalities that are difficult to implement without explicit support from the semantic core, and detail each of the common mechanisms to implement them. The services presented by Fekete are:

notification

This service relates to the need to be able to observe all types of events in the system (writing a variable, modifying a file, etc.). The problem is that some types of events are not observable, and registering callbacks ***is*** not a consistent mechanism throughout the system, which calls for unified support at the programming language level. . **error prevention** This service means knowing if a function will
trigger an error before
even executing it

(e.g. Can I read the contents of a file like an image?), or even list the valid functions in a particular context. In the absence of this type of functionality, interactive systems must store validity information upstream (e.g. the extension of a file before it is read by an application), which represents redundancy and a risk of correspondences. erroneous. **cancellation**

This service amounts to saving the state of the application at given times, and being able to return to it at any time, in order to implement the ***undo*** and ***redo commands.*** It is present in all applications today, and it is among the three Fekete services the one that has been the most studied in the HMI field [Nan14, Hee08]. However, it also remains very difficult to implement, due to the irremediable nature of many commands in computer systems (variable assignment, file deletion, etc.). Implementing cancellation requires materializing and manipulating the current state of the system, which could be facilitated by the language.

This work is notable for having been presented to both the Human-Computer Interaction and Software Engineering communities. The need therefore arises from the experience of a practitioner having implemented interaction techniques and an interactive application programming tool, and is addressed to a community which can provide solutions. In addition, the services illustrate specific needs, and are described technically as well as theoretically. However, the work was not followed by research to resolve the problems raised — which still remain relevant today.
The article having mainly adopted a HMI point of view, it remains little detailed on the implementation options on the programming language side, which was probably insufficient for

convince researchers in the GL community. Nevertheless, the point corresponding to global notification was the subject of active research in HMI shortly after Fekete's article, with in particular the creation of the Ivy software bus [Bui02], the TUIO ***multi-touch*** input protocol [Kal05], and the concept of publicly activatable components of djnn [Cha15].

2.1.2 *Usability requirements for interaction-oriented development tools*

"Usability requirements for interaction-oriented development tools" [Let10] were studied by Letondal et al. at ENAC, as part of the development of critical systems for civil aviation. In this context, the authors observe a gap between what common programming languages offer (mostly designed for calculation), and the practices and needs identified in HCI. They therefore list the differences between calculation programs and interactive programs, in order to draw recommendations to guide the design of programming tools. Thus, their work is similar to dimensions of comparison of programming tools with respect to the interaction support, in the manner of Green's ***Cognitive Dimensions of Notations*** for languages [Gre96]. They draw high-level recommendations from each dimension, which we summarize here:

- minimize the complexity of information, the complexity of access (which is similar to reducing the fragmentation of code and data structures), and unpredictability
- provide concepts of graphics, adaptation to the environment, different models of interaction (state machines, data flow, parallelism, reactivity), and applications distributed
- support the production of code, and visualization as well as debugging at runtime
- manage the life of the project (in its different stages), the reuse of code and knowledge, and the development of several

To justify the low impact of such recommendations outside the HMI field, the authors recognize that interfaces are a moving target, and that the solutions proposed in research are probably too far from industrial practices. Their study is remarkable for having been followed by the development of an architectural model using "interactive components" [Cha12], as well as the formalization of causal relationships with djnn/Smala ***bindings*** [Mag18]. This work, both theoretical and practical, resulted in concrete technical achievements, which ecologically validate the feasibility of the concepts produced. However, the target audience for the theoretical work remained linked to HMI rather than Software Engineering, and high level rather than based on technical descriptions. This lack of dissemination to the communities concerned by the type of innovations required has probably, and for the moment at least, limited the adoption of their usability requirements to adapt programming tools to interaction.

2.1.3 Let's factor in the management of interactive application events!

In their article presented at the IHM'98 [Con98] conference, Conversy et al. present a position based on a single recommendation, very explicit and presented in detail. As part of their work, the authors have developed numerous interactive applications and interaction programming tools. They relate the use of several types of software libraries,

which they qualify as ***components,*** and which they must make coexist in interactive applications. The problem they raise in this context is that each component defines event sources, and offers its own event management mechanism, often partially incompatible with the other components. The authors therefore suggest factoring the management, and present three common approaches to event propagation in interactive systems:

- the recording of a callback at each event source, like the ***listener*** design pattern , the notification of each event to a
- single intermediary (“switcher”), to which the event drains subscribe to listen (like the Ivy communication bus [Bui02]) the use of several intermediate switchers, to possibly allow
- the existence of several interaction contexts

Their work is notable for having highlighted a proven, well-identified problem, and for having detailed the sources of this problem. The article gives us a good vision of the context of event management, and lets us imagine potential solutions to its factorization. However, the authors are primarily aimed at an audience of HCI researchers, who are likely to contribute to the creation of toolkits rather than work on programming languages. Furthermore, in the absence of a position on the type of solution to be provided, the article cannot show us how such a solution would resolve the problem posed. In particular, a toolbox factoring in event management could be considered as a new “component”, and accentuate cohabitation problems rather than solving them.

2.1.4 *The Natural Programming Project*

The ***Natural Programming Project*** has been running for over two decades at Carnegie Mellon University's HCII [Mye08]. It aims to produce “natural” programming tools, that is to say closer to the way people think about algorithms and solving tasks. The project is aimed at the many people who interact professionally with computers, or even who call themselves programmers, without being “professionals”. The team working on this research theme has produced a very large number of studies and tools, summarized in [Car19].
One of these projects was the creation of a programming language and its environment, HANDS, based on studies on natural reasoning linked to solving programming problems [Mye04]. The project focuses on the creation of the tool rather than on providing explicit recommendations. However, four points listed at the end of the preliminary studies seem to us to be relevant recommendations. These studies aimed to observe people solving programming problems, without computers, and without influencing the type of reasoning to adopt. We translate the points and reproduce their examples:

- A structure based on events or rules is often used (" ***When PacMan loses all his lives, it's game over.*** ").
- Operations are specified on groups of elements rather than iterating over each one (" ***Move everyone below the 5th place down by one.*** ").
- Boolean logical operations are rarely used, and do not correspond to usual mathematical logic.

- Behaviors and actions are specified with text rather than drawings.

These points are interesting because they provide clear and oriented answers to debated design choices (e.g. the choice of a textual or visual language). They can therefore be used directly and clearly to guide the design of programming tools. Furthermore, they were applied in practice for the design of the HANDS programming environment, the use of which was subsequently tested experimentally. However, we can notice that this work has drifted towards a target of young and novice users (figure 13), while the project did not initially target only young people. Their work could therefore be applied to intermediate to advanced languages, which better correspond to the area of study of this thesis.

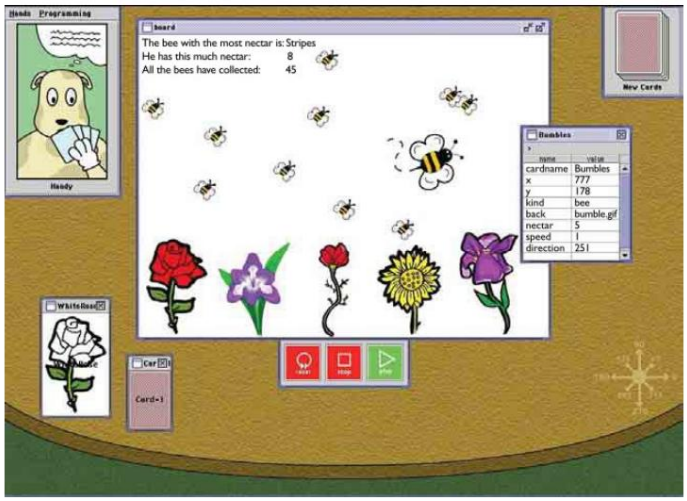

Figure 13: HANDS environment interface (figure taken from [Mye04]).

2.1.5 Limitations of the state of the art and lessons to be learned

Despite the numerous works pointing out the lack of suitability of programming languages for the expression of interactions, and the works presented offering avenues for improvement, the state of programming tools has evolved little in the desired direction. We attribute two main reasons to this observation: the too great distance between the needs identified and the technical improvements required, and the lack of ecological scope of the recommendations.

2.1.5.1 From recommendations to achievements

The work we have studied focuses on formulating recommendations for the implementation of languages or frameworks dedicated to interaction. Unfortunately, most of these recommendations have not resulted in tangible achievements. Several reasons can explain this observation. First, implementing these recommendations may prove much more difficult than expected. When you start by modifying an existing language or framework, it often has limited extensibility, which can impose unexpected constraints on the solutions. If you want to develop a language or framework from scratch, you will need to develop a large number of features to obtain a system that can be tested in real (or even controlled) conditions. Then, the engineering of languages and frameworks requires specific skills, distinct from the engineering of interactive applications. It will require calling on people

not necessarily having participated in the definition of the recommendations. If these people were not involved at the start of the project, they may have different opinions, which could hinder and slow down the continuation of the project. Finally, we observe that once presented, the recommendations linked to the evolution of interactive systems are often insufficiently valued by their authors. It is likely that an effective “wall” between ideas and their realizations discourages authors, who have already invested a lot of effort in identifying relevant problems, and give up investing more effort for lack of clear prospects of results .

So what is the point of such recommendations? Since few languages have responded to the recommendations set out in HMI, we must justify their interest, in particular for the Interaction Essentials formulated in this manuscript. The primary function of this type of recommendation is not their use by other researchers to guide the design of interactive systems. Indeed, these points are always debatable, particularly when they do not come down to experimentally verifiable hypotheses. They can possibly inform the decisions of other researchers, or even support them in their decisions. However, we consider that these are mainly ***declarations of intentions.*** They are formulated in order to spark discussions and gather the opinions of other researchers. In this sense, the Interaction Essentials are a stage of our work and not its end. They are designed to guide the achievements presented in this manuscript, and have no further scope. Thus, it is the technical achievements that will convey our real recommendations on the evolution of programming tools for interactive systems, and these are the ones that we will strive to promote in the future.

2.1.5.2 Achievements for users

Among the works we studied, we noticed that they rarely answer the question ***Why should we improve interaction programming tools?*** This question must justify the enumeration of recommendations, and then the work to modify programming languages and frameworks. It is by responding to them that we can convince developers of programming tools to adopt our recommendations. Without this, it will be difficult for them to imagine the benefits of such contributions. However, we observed that certain justifications are endogenous, in that they relate to the difficulty researchers have in implementing their recommendations, rather than to needs external to the authors. The solutions provided then tend to move away from the needs expressed by users of interaction languages and frameworks, at the risk of no longer influencing them. Faced with this problem, it seems important to us to plan in advance ***the ecological integration*** of technical achievements into a community of users.

This community should be clearly identified, as Fekete did with the Programming Working Group. It is also preferable to work hand in hand with it, in order to ensure the successful completion of technical achievements and regularly collect feedback on their adoption.
From our observations of work that has proposed recommendations related to interaction, we consider rapprochement with a community of users as an essential criterion for success. In addition, the success or failure of a project should be able to be measured, using previously stated criteria. In our case, we evaluate the success of our work against one of these criteria:

- the use of our tools by developers outside our direct collaborators the adoption
- of one of the proposed concepts, in a software library other than ours the publication
- of work inspired by (or calling into question) our results

2.2 Explicit and flexible orchestration of interactive behaviors

In a programming language, the behavior of the different elements of the language during execution is called an ***execution model .*** It is accompanied by a set of syntax elements and grammar rules, and makes it possible to predict the observable behavior of an application from its source code. For example, the C language specifies that statements (separated by semicolons) execute in sequence, each waiting for the previous one to complete before beginning. It models the execution of a program as a continuous sequence of instructions, divided into blocks of code, which are followed by branch instructions such as **goto** or function calls. The execution model is an essential characteristic of each programming language, and several execution models can coexist in the same language.

Sometimes a software library uses a different execution model than the programming language on which it runs—for example, to execute instructions in parallel rather than in sequence. Without modifying the language execution model, it superimposes its own model on it using function calls, which form the API of this library. Its execution model is then called a ***programming model.*** When we designate the programming style (the ways of reasoning with a given programming model), we will rather speak of a ***programming paradigm.*** For example, the “object model” consists of representing the elements of the program by autonomous containers (the objects), having a set of data and functions (the methods) to operate on this data and other objects. Here, the “object-oriented programming” paradigm consists of organizing the operation of a program through exchanges between objects, where each has a defined responsibility, and possibly delegates certain tasks to other objects. In object-oriented programming, it is common to encounter short methods, of a few lines of code, which is part of the programming style here.

In the context of HMI programming, we call ***interactive behaviors*** observable behaviors in reaction to user actions. In this area, we observe that many frameworks apply a programming model different from the execution model of the language on which they are based. For example, Qt implements its signals and slots model on top of the C++ procedural model [Qt19]. Similarly, Amulet implements a prototype object model on top of the C++ class model [Mye97]. These observations support the idea that the native execution models of current languages are not sufficiently adapted to express the needs of interactive applications [Bea08]. We therefore seek to answer the question ***How to design an execution model adapted to interaction?***

In the following subsections, we present different ***programming practices*** common in the development of interactive applications, and put into practice in many frameworks. Our goal is to transpose them into concepts generalizable to languages, and to discuss the forms they should take in the future.

2.2.1 The different interaction programming models

Many programming models have been used to express interaction. We draw up a list in this subsection, in order to position ourselves on the models to favor in our Interaction Essentials. For the sake of readability, we separate them into two groups, those which define the ***individual*** behaviors of instructions, and those which define the behaviors of ***groups*** of instructions. The first models focus on operations of a language (or framework) that cannot be subdivided. They include arithmetic operations, memory accesses, and even higher level operations such as file accesses. They also define how these different operations follow one another (sequencing, simultaneity). The main execution models used to program individual interactive behaviors today are: imperative programming (used in imperative languages), which models instructions as point orders given to the machine, ordered in sequence and executed without overlapping

-

- declarative programming, in which the application is a data structure that specifies its behavior (e.g. visual appearance with HTML), but does not describe ***how to*** execute it dataflow programming , in which the instructions
- continuously update output values based on input values, and which is implemented in the VHDL and Max/MSP languages, or the ICON framework [Dra01] state machines and Petri nets, in which the system is at any time in one state among a finite set, and evolves by atomic
- transitions between these states, with frameworks like CPN2000 [Bea00] as representatives. and ICO [Nav09]

At the instruction ***group*** level , we designate the definition of the code blocks of a language (or framework). This is about answering ***when*** and ***how*** they are executed, and how the overall behavior of the program is expressed by assembling these blocks. The main execution models used to program interactive group behaviors today are:

- procedural programming, in which blocks of code are functions, executed by static or dynamic function calls (function pointers), and which return execution to the calling function at the end of the block, possibly with a value of return functional programming, which models program execution as the evaluation
- of mathematical functions, prohibiting side effects authorized by assignments of

  variables
- object-oriented programming, which models blocks of code as functions belonging to objects (methods), and organizes the execution flow by messages sent between objects

- reactive programming, which links the triggering of code blocks to the activation of signals (events internal or external to the program), and models the execution of the program by cascades of signal activations
- parallel programming, usable in conjunction of the other four, and which allows several execution flows to be executed simultaneously, by proposing various programming paradigms allowing concurrent access to common resources

Imperative programming is now omnipresent for programming graphical interfaces and interactions. Indeed, this model corresponds to the sequential operation of the most widespread processors, and makes it possible to generate fast code. Additionally, interface programming is largely based on object-oriented programming languages (C++, Java, JavaScript, Swift, etc.). In practice, these models are used in almost all of the interaction libraries that we noted in the online questionnaire (see section 1.5). However, we also observe that many frameworks support ***several*** programming models, allowing several ways to define the same interface. For example, Qt allows you to build an interface using function calls in C++, but also defines the dedicated language QML to express it in a declarative manner. Likewise, the React framework introduces a ***dataflow*** model without replacing the imperative model of JavaScript, as well as the dedicated JSX language. These ***multi-paradigm*** approaches seem essential to us for programming interactions. They leave developers free to choose their way of reasoning, and to change depending on the type of interface to be created. This is particularly useful for associating a type of need with a suitable model (e.g. ***dataflow*** for constraint propagation, reactive for recording to interaction devices, declarative for interface construction) We have therefore chosen to favor subsequently a library that includes or supports the use of several programming models.

2.2.2 Event propagation by active and passive waiting

In interaction programming, the active ***(polling)*** and passive ***(pushing)*** types designate two ways of propagating user input information in a computer program. While actively waiting, an input source (mouse, keyboard, joystick, etc.) is repeatedly interrogated.
When a new entry is detected, code is executed which will take care of the processing following the entry (command, visual feedback, etc.). Then, whether an entry has been detected or not, we possibly delay before interrogating the entry source again, so as not to monopolize the machine's resources. Active waiting is commonly illustrated in low-level management of input devices, and synchronization between parallel processes on multi-core machines, where it is essential to react as quickly as the machine allows:

```
volatile int condition;          // force systematic accesses to memory
while (condition == 0)
    { wait(10);                  // 10ms timeout
}
```

In the absence of a delay, active waiting guarantees an instantaneous reaction to any event, at the cost of continuous consumption of the processor, which makes it unavailable for other tasks. By extension, it is also illustrated when we interrogate all the input events occurring during a period of time, in order to manage the inputs in a fixed frequency loop:

```
while (!end) { while
    (pending_input()) {
        event = poll_input(); // non-blocking request process_input(event); //
        treatment

    } render_scene();                    // drawing
    wait(10);                            // 10ms timeout
}
```

This type of looping and obtaining events is particularly useful when the interactive application must react to several competing event sources (display, mouse, keyboard, network). The fixed frequency is generally synchronized with that of the screen, to minimize the delay between the visual rendering of the application and its perception by the user's eye. It is also useful for controlling the determinism of the application, in particular as a ***tickrate*** in multiplayer games which synchronize calculations between all connected machines. However, due to the delay, the use of an active wait introduces a latency here between the occurrence of an event and its processing, which is on average worth half the timeout period (5ms in the example above).

In passive waiting ***(pushing),*** it is the transmitter of an event who "pushes" it to consumers, rather than the latter requesting it. To do this, a callback function ***is*** registered beforehand, which contains the code to be executed when an input is detected. It is the input source which is responsible for waiting for an event to occur, and which then executes the callback function instantly. With this type of event propagation, the example above would become:

```
set_on_event(e ->
    { process_event(e);
})
set_on_display(() ->
    { render_scene();
})
```

Passive waiting is useful when events occur sporadically — the interaction is not continuous. This is the case for the keyboard and mouse, for which it is normal to observe periods of time without activity, during which repeated requests to input sources would be superfluous. This type of wait is rarely used at low level, due to the cost of executing callback functions. On the other hand, it is commonly observed in frameworks, which propagate input events to user code using ***callbacks.*** Compared to a fixed frequency loop, passive waiting does not introduce any latency in the processing of user inputs, which is desirable in areas where this latency is harmful (real-time systems). However, while with active waiting it is the application which remains in control of execution, with callback functions it cedes control to the environment which is responsible for calling it at an undetermined time. Passive waiting therefore introduces non-determinism ***from the point of view of the called party,*** which explains why event management loops are still used internally in many frameworks.

Because of the latency induced by active waiting in a fixed frequency loop, passive waiting will always be preferred. This is particularly true in HMI, where the harmful effect of latency on user performance has been extensively studied [Pav11, Ng14, Deb15]. However, non-determinism due to delegation of control to the environment is problematic. Indeed, without knowing the processing carried out by the framework, it is impossible to know if a delay separates the occurrence of an event from the moment when control is received. Likewise, in the case of an application carrying out complex calculations (e.g. video game), it is impossible to guarantee that we will have enough calculation time each second, because the framework's processing is unpredictable (e.g. setting up graphics rendering cache, garbage collection). Finally, as we pointed out in the problem of this thesis, with the layered organization of interaction libraries, we cannot guarantee that the framework forwards all input events, and a developer who interacts with several layers will have to ensure the consistency of the data received. Under these conditions, we have chosen to promote passive waiting for events, but where the processing carried out by the framework is specified and observable.

2.2.3 Blocking and asynchronous processing

Blocking and asynchronous processing models refer to two ways of predicting the execution of a block of code after executing a function that can take a long time. The blocking model consists of pausing the execution of the program until the function has finished executing. Once it finishes, the program resumes execution. This type of processing is widely used for input/output with the hard drive or the network. In the context of interaction programming, it is used when we expect user input in response to a system request (for example a form with a “Continue” button). We illustrate it for example in C:

```
print("What is your name?") name
= read_string()                    // blocking treatment
print("Hello %s, how old are you?", name) age =
read_number()                      // blocking treatment
print("You are %d.", age)
```

The advantage of this model is that we can represent each “thread of execution” by a continuous block of code, in particular when it stretches over time. Indeed, the **print** instructions above belong to the same block of sequential code, yet they will execute at very different times. The synchronous blocking model therefore makes it possible to model a ***conversation*** between the user and the machine very well, in which they communicate by questions and answers. On the other hand, we cannot wait for several questions simultaneously with this model (unless we allow several simultaneous threads of execution).

With asynchronous processing, the execution of a time-consuming function no longer blocks the current execution thread. Instead, costly processing of the function is delayed, waiting until the machine has no more calculations in progress. It is the function itself which determines the processing operations to be carried out instantly and those which are put on hold, and it is the “system” (framework, language,

or operating system) which manages pending processing and chooses when to execute them. When code must be executed ***after*** the function has been executed, a callback function is often passed as an argument, which will be called after pending processing. The previous example becomes:

```
print("What is your name?")
read_string_async(name -> {
      print("Hello %s, how old are you?", name)
      read_number_async(age ->
            { print("You are %d.", age)
      })
})
```

Asynchronous execution allows you to manage multiple tasks simultaneously with a single thread. As each asynchronous function does not block execution, we could for example ask several questions at the same time, and react to the answers in the order in which they are given. In the case of graphical interfaces, asynchronous programming is particularly used for inputs/outputs whose results are ***secondary*** to the interface, for example to wait for the loading of images which are inserted into the interface as they are received. To model direct interactions with the user, the major problem is that we do not control when the callback function is executed. We therefore introduce non-determinism, which can be a source of bugs when the result of the different tasks depends on their order of execution. Additionally, asynchronous programming tends to produce cascading callback functions, which harm the readability of the program (see example above).

When we extend the conversation between user and machine to a graphical interface, we can distinguish two main threads of execution. The ***interface*** thread contains the processing carried out in sequence at regular intervals, to refresh the interface (e.g. input processing, updating of positioning constraints, visual rendering). It is an actual thread, which generally corresponds to one or more processes on the machine. The ***interaction*** thread represents all the processing carried out (in sequence or in parallel) in reaction to the user's actions (e.g. displaying visual feedback, changing interaction mode, triggering a command). It represents the conversation (or interactions) between the user and the machine. However, it rarely corresponds to a dedicated process on the machine. Indeed, managing multiple processes is complex for a framework, because their possible synchronizations must be managed. While some frameworks separate the updating of the interface and the rest of the processing into processes (e.g. the ***Event Dispatching Thread*** of AWT), the management of the interaction is closely linked to the management of the interface, which would require many synchronizations.

So we have two opportunities at the end of this discussion. The first is to represent the interaction thread by a real process (overcoming the above-mentioned challenges), like Huot's interaction graphs [Huo06], but in an imperative rather than declarative execution model. The second is to integrate the interaction thread more closely with the interface thread, in order to maintain only a single process. The conceptual simplicity of this approach led us to explore it in this thesis. Because of the existence of a single thread of execution, we excluded synchronous blocking processing. To avoid introducing unwanted non-determinism, we also excluded asynchronous processing. We therefore sought to integrate the interaction thread

in interface processing. In practice this principle is illustrated in chapter 3, where changes of state and possible timeouts are ***reified*** into global data, on which blocks of code inserted in a single thread of execution can react.

2.2.4 Callbacks and distributed logic

In procedural programming languages, any function must be called explicitly to execute. The initiative for a function call is usually internal to the application, it allows you to “jump” from one portion of the program to another, and then return to it. When the initiative comes from outside (for example following a user input event), callback functions are used . These are functions materialized by values, which can be passed as arguments to other functions, and saved in variables. This materialization takes the form of a function pointer (address of the first instruction in memory) in an imperative language like C, or of an object containing a function pointer (lambda ***function)*** in an object language like Python or Java. The use of callbacks allows the principle of Inversion of Control, in which we delegate the triggering of a function to the framework (or language). We illustrate it with the Hollywood Principle: “ Don’t ***call us, we’ll call you ”*** . They are used to specify behaviors to be executed after user events, for which the initiative necessarily comes from outside the application (by the framework, language, or operating system).

The use of callbacks promotes decentralized code distribution. Indeed, each ***type*** of processing in response to the user (triggering of each command, propagation of each constraint) is contained in a separate callback. We then speak of ***distributed logic*** to mean that the observable behavior of the application is caused by sequences of instructions fragmented in several places in the code. From the application programmer's point of view, these fragments can be distributed across several files, or in several places within the same file.
Distributed logic also relates to the storage of callbacks during application execution.
These are usually attached to the widgets that trigger them — for example a command is attached to the button that triggers it. Each widget then stores the values (pointers, or lambda functions) of the callbacks that it triggers, and the storage of all the callbacks is therefore distributed between all the widgets.

Realistic interfaces can contain hundreds or even thousands of callbacks — at least as many as the number of clickable and interactive elements in the interface. Under these conditions, the creation, understanding, and maintenance of the interface are made difficult by the multiplicity and fragmentation of interactive behaviors. The problem is called ***“spaghetti of callbacks”,*** named after the article by Myers which highlighted it [Mye91]. It has given rise to numerous works aimed at:

- separate the interface and the code to reduce fragmentation in files (HTML/JavaScript, QML/Qt, FXML/JavaFX)
- group the storage of callbacks linked to a particular interaction technique (state machines) merge
- callbacks carrying out similar processing (see for example **ToggleGroup** in JavaFX, or **NSSegmentedControl** in Cocoa)

The distribution of logic in an interactive application is a question of point of view, depending on the object of interest whose cohesion is being evaluated. When looking at the logic of a program from the machine's point of view, unfragmented distribution is equivalent to the sequentiality of instructions executed, that is, the reduction of function calls and conditional branches. When we look at the logic from a widget point of view, it is equivalent to grouping all the actions done on a widget into a single object. Finally, when we observe the logic from the point of view of an interaction modality (e.g. the mouse), it is equivalent to grouping all the actions that can be executed by a modality in a single object. In practice we observe a compromise between these three points of view in interaction frameworks, and the widget point of view is often favored over the others in architectures based on widgets.

The problem raised by the fragmentation of logic in interactive applications is that interfaces are best expressed in one point of view, often to the detriment of others. For example, the drag-and-drop technique is notoriously more difficult to implement when expressed by callbacks on widgets [Wag95], than by callbacks linked to the interaction technique [App08]. The lack of centralization of logic from the machine's perspective makes it difficult to understand and visualize the flow of program execution, making it particularly difficult to debug GUIs. We therefore wish to improve the centralization of the logic for each of the three points of view, and improve the transition between the three so as not to favor one to the detriment of the others. With this in mind, we have chosen a minimally fragmented execution orchestration, in which the actions relating to an object or an interaction technique are sequential in code as much as possible.

2.2.5 Support for explicit and flexible orchestration of interaction

In light of the aspects of the execution paradigms listed above in the context of interaction, we want to formulate an Interaction Essential with the following characteristics:

- includes or supports the use of several programming models
- propagates events passively ***(pushing),*** but where the processing linked to event management is specified and observable
- integrates the “interface thread” and the “interface thread” interaction” coherently in the
- same procedure brings together similar processing (by type, membership widget, or interaction modality) in sequential blocks of code

What these features have in common is that they are intended to give more control to the programmer seeking to express interactive behaviors. This may involve clearly expressing which user actions can trigger a block of code, or even bringing together all the processing linked to the same interaction technique in the same block of code. We compare this extended control to that of an orchestra conductor. ***Orchestration*** refers to the arrangement of different actions in relation to each other. It is easily compared with music orchestras.
Each musician must play in rhythm, in the same key (note frequencies), and with the same intensity as the other musicians, to bring coherence to the entire composition. The conductor, who seeks this harmony, communicates collectively and individually with the musicians, through specific vocabulary, to unify their rhythm, tone, and intensity. We

Let's talk about orchestration to designate the tools available to the conductor (and in our context, programmers), to arrange, order, and synchronize musicians (or interactive processes).

Orchestration occurs at a macroscopic level, as Berry underlines in the development of Hop and HipHop [Ber14]. It is not a question of orchestrating atomic instructions between them, but of complex processes. The needs related to orchestration, in the context of interactive applications, have been studied with “reactive” execution models, with languages like Esterel [Ber87], Chandelier [Cas87], and Signal [LeG91]. Work has also proposed extensions to imperative languages, which facilitate their integration into existing ecosystems. Thus, HipHop.js integrates into the JavaScript language to produce Web applications distributed between client and server [Vid18]. ReactiveC is integrated into the C language using a preprocessor [Bou91].
Finally, ReactiveML is integrated into the OCaml language, and has been notably applied to interactive simulations [Man09] and musical production [Bau13].

However, work based on the reactive model presupposes that ***all*** execution takes place within the program, and does not allow the initiative to be given to the environment — with the exception of ICon which extended this model to external events. to apply it to the inputs of interaction devices [Dra04]. Processes are always triggered from the application itself, and it is from there that we wait for events, or synchronize processes. This operation is different from that of common interaction frameworks, in which the application leaves control to the system, to be called in return. The problem of initiative influences the interaction model that is adopted, but as we have presented, the conservation of initiative in the application favors a conversational model. Furthermore, they do not allow the expression of causal relationships between processes, as proposed by the djnn ***bindings*** [Cha12], or Aspect Oriented Programming [Kic97]. Without overly directing the type of solutions to adopt in our work, we therefore formulate the first Interaction Essential:

Interaction Essential #1

Orchestrate procedural interactive behaviors, expressing their sequencing, their causality, and their reaction to the initiative of external events

2.3 The interaction environment of any application

The interaction environment of an application designates the set of functions and data structures allowing it to interact with interaction devices (keyboard, mouse, screen, etc.), and with users via peripherals. It designates the software libraries which make it possible to intercept mouse and keyboard events, to interrogate their physical characteristics, or to draw on the screen. The ease of use of these tools determines the effort that programmers will invest in designing interaction techniques. Unfortunately, we can consider today that these tools are generally complex and not very accessible, and that they hinder the work of HCI researchers. The problems are multiple and have been highlighted in the first chapter of this manuscript. First, programmers are faced with a vast choice of software libraries, and must spend time choosing them correctly based on partial or incomplete information. Second, libraries and interactive systems in general are arranged in interdependent layers, which pose problems of information consistency, and complicate the choice of which layers to address. Finally, most interaction libraries lack extensibility, and are difficult to lend themselves to research work that explores — by definition — new solutions.

In this section, we argue in favor of a definition of functions and data structures dedicated to interaction with users, which can be integrated into a programming language or framework, in order to respond to the problems raised. The following subsections are devoted to the study of three aspects of programming inputs and outputs with users, which lead us to define as Essential Interaction **a minimal interaction environment initialized at the start of any application .**

2.3.1 Support for interaction in programming languages

The programming language consists a priori of a syntax as well as grammar rules, which make it possible to express orders to be given to a machine. The complexity of the orders given depends on the power of expression of the language, and is conditioned by the types of problems targeted by the language. Most programming languages today primarily target scientific computing and data communication (with storage devices, or between machines). In practice this means that their basic concepts are very suitable for calculation (arithmetic operations, mathematical functions, MapReduce data processing paradigm, etc.), and communications (reading/writing in data streams, abstraction of file descriptors , asynchronous wait, etc.). Besides these concepts, ***interaction*** does not seem to be explicitly supported in common programming languages.

What do we mean by supporting interaction in a programming language? This is to facilitate the programming of low-level interactive applications — that is, the creation of programs that receive data from input devices, and emit data to output devices, in order to carry out a ***perception-action*** loop from the user's point of view. Input, computers are mainly controlled by keyboard and mouse for desktop computers, and by finger for smartphones. As output, the majority of interactive applications use a screen

matrix (a matrix of colored dots). Other types of peripherals are used in specific contexts (controllers, voice, gaze, audio, haptics, etc.), however the limited support for basic peripherals encourages us to focus on them first. An ***essential*** support for interaction programming therefore consists of the keyboard/mouse/screen triplet.

To explain our idea of better support for interaction devices, it is first necessary to describe their support in current computer systems. Let's start by taking the example of the C language, because it is a clear example, and the second most used language today (September 2019) according to the TIOBE index [Tio19]. The core of the language mainly supports: variables, types, pointers, **struct records,** arrays, enumerations, lexical scope, functions, conditional blocks, loops, **goto branches,** arithmetic/binary operations/ logic, integers, floating point numbers, complex numbers, character strings, import/export of external symbols, and a preprocessor [ISO18]. Then, every language has a “standard” library, which implements functionalities that do not require syntactic support from the language, and are simply expressed by functions. The C language provides the following modules: **assert.h, complex.h, ctype.h, errno.h, fenv.h, float.h, inttypes.h, iso646.h, limits.h, locale.h, math. h, setjmp.h, signal.h, stdalign.h, stdarg.h, stdatomic.h, stdbool.h, stddef.h, stdint.h, stdio.h, stdlib.h, stdnoreturn.h, string.h, tgmath.h, threads.h, time.h, uchar.h, wchar.h, and wctype.h.**

Among these two levels (syntax and standard library), the C language does not specify ***any*** support for the keyboard/mouse/screen triplet. Only the UNIX and POSIX conventions make it possible to integrate rudimentary support with the use of files: text entry on the keyboard and visual feedback in a text terminal. It is from the operating system that we have sufficient support for interaction peripherals, however many alternatives exist and complicate the choice of programmers: GNU Readline, OpenGL, conio.h (MS-DOS) , ncurses (UNIX), Newt (Linux), , GDI (Windows), Direct2D (Windows), DirectDraw (Windows), DirectWrite (Windows), QuickDraw GX (macOS), Carbon Event Manager (macOS), Quartz 2D (macOS), Metal (macOS). Here we have only listed ***native*** libraries of the different systems. Additionally there are a large number of libraries that build on previous ones, provide similar services, and help complicate the topology of interaction programming tools in computer systems.

This situation can be explained by historical changes in input and output interaction modalities. Although the use of the keyboard and mouse have been standard for more than 40 years now, their characteristics have evolved considerably. Keyboards have evolved mainly in the key layouts and “special” keys (commands, multimedia control). Mice have evolved in the types of sensors, the precision of the wheel, the buttons present, as well as the transfer functions between physical movement and position of the cursor on the screen. In addition, many peripherals have been developed that have attempted to evolve these standard interaction modalities, such as the 3D mouse, the trackpad, or the “jellyfish” keyboard. Faced with these developments, interaction functionalities could be considered unstable over time, and dissuade language designers from integrating them into standard libraries. Additionally, it is common for some interaction devices to work partially (muted audio, no acceleration

hardware, multimedia keys not recognized), without preventing the operating system from functioning correctly. However, the specification of ***any*** functionality of a language implies that it must work on all systems, therefore that it is the responsibility of the compiler designers. Under these conditions, support for interaction in languages would require compiler designers to follow developments in interaction devices, and would require significant maintenance work, which may explain why interaction is ultimately not included.
not.

How can we formulate a realistic and achievable recommendation in this situation? We propose three possible solutions, without choosing one in particular at the moment. The first possibility is **to include functionalities that have been stable for several decades in the languages :**

- for the keyboard, key press and release events, with the key position code and the corresponding printable character for the mouse, relative
- physical movement, button press and release, and physical movement events of the wheel for the screen, the synchronization event with the
- refresh of the display, with the color matrix to edit

These features are basic, but would facilitate the development of robust, cross-platform software libraries. Indeed, they would not require the use of a dedicated library for each device and each system, and would thus reduce the learning required. Then, by their inclusion in the language they would not require the installation of a third-party library, nor specific actions to activate them, which would improve their accessibility. Finally, we conjecture that their inclusion in a language (alongside basic features such as text manipulation or file access) would force them to adopt a simple interface, which could reasonably be supported by compilers while retaining some power. of expression.

A second possibility is to **recognize the longevity of certain software libraries linked to interaction, and to facilitate the use of their API.** We could thus imagine that the compiler automatically initializes these libraries, if it detects that their functions are used. Furthermore it would not be necessary to install these libraries in addition to a compiler, to have to use them.
Finally, standardizing these libraries would ease the dilemma of choice when alternatives exist, and help strengthen them and their online communities. Among these libraries, we can cite:

- OpenGL (1994), for 3D rendering
- FreeType (1996), for rendering fonts
- SDL (1998), for managing windows and inputs

Finally, a third possibility is **to integrate support for interaction into the syntax of languages, rather than using functions.** These mechanisms would extend programming languages to express interaction in a more “native” way. An example of such an extension is mouse support in the Processing language [Rea14], which consists of global variables like **mouseX** and **mouseY,** and function names like **mousePressed()** that are called automatically when the mouse is pressed.

The third avenue seems the most interesting to us, because it recognizes the essential nature of interaction in programming. After all, it would be impossible for us to observe the operation of a program, nor to act on it, without minimal support for the keyboard/mouse/screen triplet. We therefore focused on proposing concepts that could be integrated naturally into programming languages, not as function calls but as native syntaxes. In this thesis work, we propose an extension of the Smalltalk language dedicated to animation, which we present in section 2.4. Finally, in the framework presented in Chapter 3, we represent interaction devices as global objects, which can be interrogated at any time to observe input events.

2.3.2 Alternatives to event programming

Event-driven programming is a popular concept in HMI, but which is also very poorly defined. It is mainly characterized by the manipulation of events , which can represent user actions (e.g. mouse movement), messages between processes or machines, or all types of state changes. For example, to model mouse movements in event-driven programming, we would define a **MouseMoveEvent structure,** containing the relative movement of the mouse, as well as possible information such as the name of the device or its precision.

```
struct MouseMoveEvent
    { int dx;
    int dy;
    String productID;
    int dpi;
    ...
}
```

Event-driven programming is often used to qualify systems that listen to their environment, however the ***way*** of managing events varies. When used to propagate events with passive waiting, it is associated with the listener pattern ***to*** register callback functions ***(event handler)*** handling each type of event. With this mechanism, each widget stores a callback function for each type of event it can receive (mouse click in its area, key press when active, etc.). When the framework detects the occurrence of an event, it fills the fields of an event structure, finds the widget on which the event occurred, and executes the callback function(s) registered there for the type of event in question. Functions receive the structure as an argument

initialized.

When event programming is used in propagation with active waiting, it is associated with a queue of events ***(event queue),*** which accumulates them in order to process them periodically in packets. The queue is then traveled in the order of arrival of the events, these are processed one by one, then the queue is emptied to accommodate the next events. This type of storage of events is common in video games, when they are synchronized with a particular tickrate — that of a server in the case of multiplayer games.

The modeling of events in an interactive application is a compromise between the size of event structures, and the specialization of processing, which forms a continuum of granularity of solutions ranging from ***monolithic*** (with a complex and unique type of event) to ***fine*** (with a wide variety of event types with a simple structure). This continuum is represented in Figure 14. At one end, all events are represented by the same structure, and each manager must filter events that do not interest him. In return, we only keep a single list of event handlers, or a single event queue. Examples of application of this principle are the SDL framework [Lan98] as well as the TUIO protocol [Kal05]. At the other end, each type of event is represented by its own structure, and an event manager never has to filter out an event that it is not interested in. In return, it is necessary to maintain as many lists of managers, event queues, and event transmission channels as the number of possible events. Examples of application of this principle are the AWT and Swing frameworks, as well as the GLFW framework [GLF02] (this does not declare any structure but passes the variables directly as arguments to the callback functions).

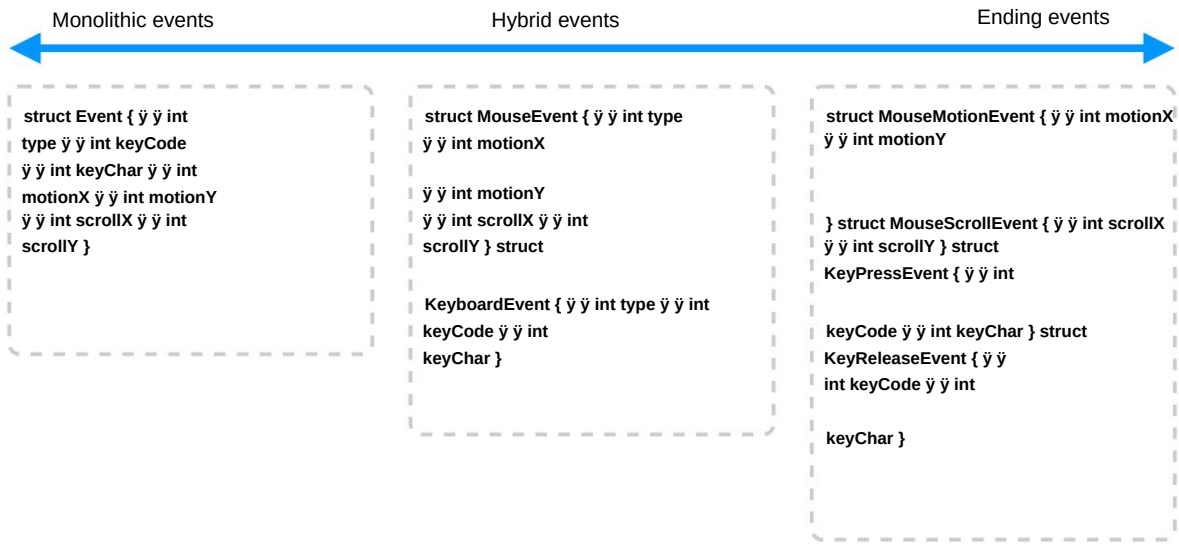

Figure 14: The granularity continuum of event types, illustrated by an example of code supporting mouse movements and keyboard key presses/releases.

In the case of fine events, the type of each event is not stored explicitly but corresponds to the name of the structure. It saves the program from using conditional structures **(if/else** and **switch)** to filter events, and leaves this responsibility to the framework and the language. In practice, most object-oriented frameworks define event hierarchies (figure 15), which allow them to offer managers at all levels of the continuum, at the cost of more complex code.

However, in practice the use of fine events favors the fragmentation of the logic discussed in section 2.2.4. Indeed, the different types of processing are separated into different functions (e.g. **onMouseMotion(...), onMouseScroll, onKeyPress, onKeyRelease).** These functions help to separate all the behaviors corresponding to the possible actions in the interface, into a multitude of individual functions which complicate the overall maintenance of

the application. In addition, event-driven programming abstracts the characteristics of each device, reducing it to an interaction ***modality .*** Although it has made it possible, for example, to easily adapt desktop interfaces to finger use (by generating mouse events), this type of abstraction reduces the richness of finger interaction (precision , pressure intensity, orientation) by reducing it to a pointer. Finally, event structures by definition communicate data about the observed action, and often omit data about input devices. They thus make it more difficult to adapt the interaction to the types of devices that generate events.

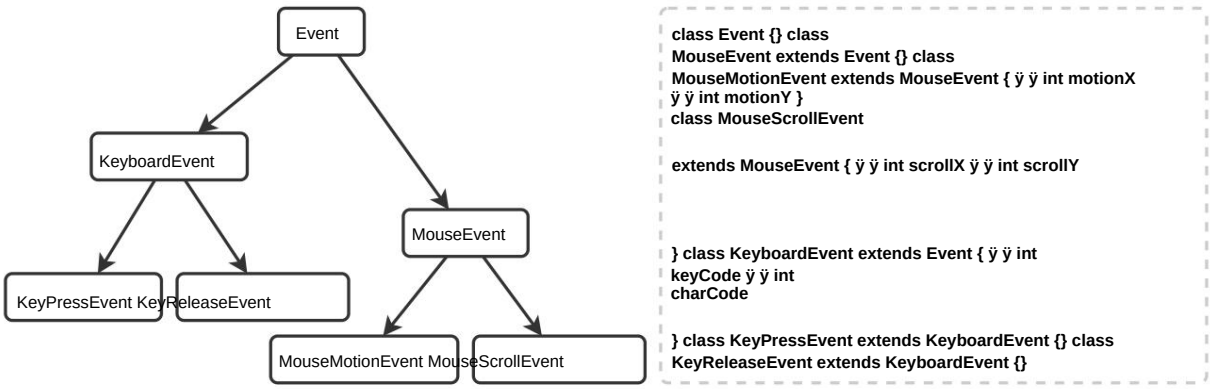

Figure 15: Example of a hierarchy of events supporting mouse movements and keyboard key presses/releases, and the corresponding code

We therefore consider it essential to **propagate the input data while preserving the characteristics of the devices that generated them,** and looked for alternatives to event-driven programming during this thesis work. In an interactive system, any change of state is accompanied by data describing this change (which button is pressed, how many units the mouse moves, etc.). This data must be made available to the program. Alternatives to event programming therefore consist of their storage somewhere other than event structures:

- on the object receiving the interaction (clicked button, active
- text field) on the device originating the interaction (mouse,
- keyboard) as arguments to event handlers in
- global variables, accessible from any handler 'event

In chapter 3, we present an alternative in which this data is stored in objects representing the devices at the origin of the interaction.

2.3.3 Immediate and deferred rendering

To draw geometric shapes, text and images on the screen, there are two drawing methods adopted by various graphics libraries, immediate mode ***and*** retained mode . In immediate drawing, we execute drawing instructions as we would give orders to the graphics system. This one paints the instructions on the screen immediately

as they run, or can add them to a queue to run them as a group later.
The immediate drawing libraries provide a “canvas” into which the instructions will paint, and which will then be displayed on all or part of the screen.

In practice, low-level drawing libraries (OpenGL, Skia, Cairo) are controlled with immediate instructions. Most interaction frameworks also provide immediate drawing modes **(canvas** for HTML/JS, **QPainter** for Qt, or **Canvas** for JavaFX). Finally, the ImGui framework is notable for having extended the immediate mode to the definition of graphical interfaces, and clarifying the distinction between the two modes [Mur05]. Immediate mode is generally seen as providing the best control because instructions are executed at a specific time and in a specific order. However, it does not provide any persistence to the drawing instructions, they will have to be returned again for rendering of each new frame.

In deferred drawing, we provide a description of the elements to be drawn to the system (a ***model),*** and it is responsible for displaying them without explicit control from the programmer. Descriptions include the background color of the elements, their geometric shape, or even the text to display. The system decides the order of their display, the refresh rate on the screen, as well as visual details like sub-pixel smoothing or the use of motion blur. The model is generally structured beyond a simple list, in a ***scene tree*** which is not limited to visual rendering, but also serves for the distribution of input events and the resolution of positioning constraints.
The appearance of elements is also sometimes specified in (immediate rendering) code, to precisely control their appearance. This code is then specified in a callback method (e.g. **paintComponent** with Swing), delegated to the framework by Inversion of Control.

Our objective in this part would be to reconcile the fine control of immediate rendering, with the persistence of deferred rendering. Indeed, the indeterminate order in which the framework executes drawing instructions in deferred rendering limits the control given to developers of interaction techniques, because they must explicitly specify the order relationships between drawn elements (which requires more coded). We therefore seek to **support a deferred rendering model, but in which the rendering steps are explicit and not reserved to the discretion of the framework.**

2.3.4 Support for a minimal and initialized interaction environment

From the discussions developed in this section, we want to formulate an Essential Interaction with the following characteristics:

- makes simple support for the keyboard/mouse/screen triplet available in the
- language integrates concepts linked to interaction into the very syntax
- of a language propagates data from input devices while retaining their
- characteristics supports deferred graphics rendering, with explicit (even flexible) rendering steps

We argue that these points could facilitate the development of interactive applications, and more particularly that of new interaction techniques in a research context. First of all, by removing the initialization step of the different device libraries, we reduce the code of an application (which the majority of frameworks already do). In addition, by integrating the devices as close as possible to the language, we intend to reduce the learning required for each additional library. It is important then that this integration is based on

concepts that make it clearer than the classic use of API functions. We conjecture that the definition of concepts integrating interaction into the syntax of the language would promote their appropriation by developers, which is essential to opportunistic development practices. We illustrate these principles with the concept ***of function animation*** presented in the following section, as well as that of ***reification of peripherals into global objects*** presented in chapter 3. Thus, our second Interaction Essential is the following:

Interaction Essential #2

Provide minimal input and output support in any application environment, supported by integrated concepts consistent with the programming language

2.4 Transforming function calls into animations

Motivated by the predominance of frameworks observed during the first interviews, we undertook the realization of a project improving them, without investing energy to explicitly modify one. Furthermore, based on the Interaction Essentials, we wanted to demonstrate our vision of a syntax integrated into a programming language. We have therefore developed, from the concept ***of animating a function,*** an extension of the Smalltalk language which converts an instantaneous transition by method call, into an animated transition over time. With this work, we also focused on including support for long- ***term*** processing in the application interaction environment (i.e. processing that takes place over a period of time rather than instantaneously). In accordance with the first Interaction Essential, the orchestration of animations is made explicit by a simple rule: any animated function is executed at a fixed frequency during the animation duration.

This work was possible thanks to collaboration with a Software Engineering research team, RMoD, which is developing its own interpreter and development environment for Smalltalk, Pharo [Duc17]. With the help of the researchers and engineers of this team, we were able to create an extension of the Smalltalk language dedicated to the expression of animations in graphical interfaces. This work was presented in ***Late Breaking Result*** at the EICS'17 conference [Raf17], and demonstrated live at the Pharo Days 2017 conference [Raf17]. We detail this work here.

2.4.1 Introduction

Since the development of computers, animations have been used in an increasingly wide range of scenarios such as: teaching programming with visual environments [Sta93, Res09, Dan11], transitions in graphical interfaces and visualizations to resize windows or alternate between views of data [Sha07, Kle05, Dra11], or the animation of virtual characters in video games by interpolation between keyframes [Bro88, Wil97]. Animations are considered useful in user interfaces to help track changes [Sch07], and in visualizations to build a mental map of spatial information [Bed99]. They can also give meaning to data visualization [Gon96], to narration [Ken02], as well as many other uses in user interfaces [Che16].

Interaction frameworks have evolved over the years to support a wider variety of uses, by offering more flexible ways to animate user interface elements.
While systems of the past animated a few properties (position, color) with dedicated functions for each, modern systems contain too many to continue this way. For example, CSS has 44 animable properties [Bar18], and Core Animation has 29 animable properties [App06]. To handle this growing number of animable properties, most frameworks define generic types that can be animated (like **IntProperty,** or **DoubleProperty),** instead of having a specific function for each. They thus improve the flexibility of choosing properties to animate at runtime, and reduce the size of their API. They also implicitly state that ***any*** property can be animated, or at least those that would make sense to the programmer.

However, this flexibility comes at a price. Animating user-defined types requires providing an advanced API, which exposes the low-level details of the frameworks' animation systems, particularly timers and threads. This results in broader animation APIs and heavy syntaxes due to additional indirections to access animable types. There is also a steep learning curve between the basic API and the advanced API, which is likely to force programmers to stick to existing animable properties as much as possible.

In this section we introduce a ***duration operator,*** to express animations by transforming ***setter*** calls into animated transitions. We illustrate it with the following pseudo-code:

object.setProperty(target) during 2s

This syntax extends very simply to function calls (excluding object methods), however in the context of this work we have implemented it in an object language. In the rest of this section, we will talk exclusively about methods, and specify the processing of functions when they differ. We begin by listing the attributes characterizing an animation, as well as the steps necessary to construct it from a method call with duration. Then, we

Let's describe the implementation of a working prototype for the Pharo platform. Finally, we compare six modern interaction frameworks, and discuss the limitations of our system and the implication of this work for the rest of this thesis work.

2.4.2 Characterization of an animation

In modern interactive systems, the transition from an initial state to a final state is instantaneous. Changing the position of an object on the screen makes it disappear from its current position, and at the same time appear in its new position. This abrupt change can break our perception that a single object has moved, rather than a new object having appeared in another position. When we want to emphasize the movement of such an object, or maintain the perception of its uniqueness, we use animations to soften the transition in time, so that it seems continuous. The definition given by Betrancourt and Tversky is: " any ***application which generates a series of frames, so that each frame appears as an alteration of the previous one, and where the sequence of frames is determined either by the designer or the user "*** [bet00].

Animating a property in an interaction framework involves replacing an instantaneous transition with several minor changes to the same property, in rapid sequence. The frequency of these intermediate changes is as high as possible, to create an illusion of continuity in the main transition. It is usually set to the maximum of the screen refresh rate, which is the number of generated frames that can be painted on the screen each second. On most screens, this frequency is 60 Hz, and is sometimes slowed down when rendering a frame is too long.

Each modification begins by calculating a relative time between the start and end of the animation: ***t = f(now),*** where ***t = 0*** at the start, and ***t = 1*** at the end. The time ***t*** does not necessarily increase uniformly in [0, 1]: it can accelerate at start-up, bounce before stopping, and even oscillate around the finish (see [Sit19] for a complete list). We call ***f*** a transition function (in English ***easing function).***

Then, the value calculated for each modification is obtained with an interpolation function, ***interpolate(start, target, t),*** which returns ***start*** when ***t*** is 0 and ***target*** when ***t*** is 1. This function is specific for each type of values, for example colors and positions would be interpolated with different algorithms.

To describe the concept of an animated property, we based ourselves on the 5 ***high-level aspects of animations*** defined by Mirlacher et al. [Mir12], and which we have extended. We therefore define an ***animated transition object*** as containing:

- **receiver** — the object to animate (absent in the case of a function call)
- **mutator signature** — the name of the method modifying the targeted property, and the types of its arguments
- **key values** — the start and finish values for each argument
- **duration** — in seconds
- **relaxation function** — which controls speed effects during the transition

- **interpolation function** — for each argument, specific to each **execution**
- **state** type — allows you to pause, restart, loop, or reverse the animated transition

2.4.3 The duration operator

Our operator associates a duration with a function call, creating an animated transition object **<functioncall> during <duration>.** Its interpretation consists of four steps, as illustrated in Figure 16.

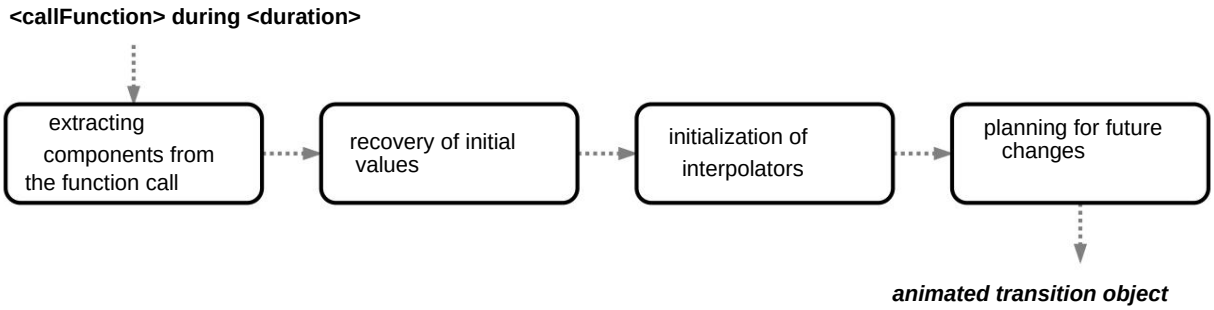

Figure 16: The four steps to transform a function call into an animated transition object

The first step, ***extracting the components of the function call,*** retrieves four elements of the function call: its recipient (if it is an object method), its name, the values of its arguments, and their types . It also cancels the immediate execution of the function. This step actually ***reifies*** the call, as we inspect its data, and extract its features. It requires that the programming language supports code introspection, that is to say the possibility of examining its properties instead of executing it (more details are given in the Implementation section).

For ***retrieving initial values,*** we need a mechanism to obtain the current values of a property targeted by a mutator method or function. Fortunately, many frameworks adopt naming conventions for getters ***and*** setters.
Qt [Qt19], for example, matches most **setProperty(...)** setters with getProperty() accessors . On the Smalltalk platform, the convention is to give them the same name, with one simply taking an argument and the other not — such as **object property** and **object property: <value>.** Naming conventions are important: without them, each mutator would have to be explicitly associated with the corresponding accessor, which would require editing each framework and would invalidate the point of this work. For functions with multiple arguments, we can require accessors to return multiple values if the language allows it (e.g. Python), or return them in references passed as parameters. Once the accessor is obtained from its name, it is called dynamically to retrieve the current values, which will be the initial values of the animated transition. For languages without dynamic function calls ***(dynamic dispatch)*** like C, a smart naming convention (e.g. **property()** ÿ **get_property())** would allow substitution by the preprocessor. Otherwise, this would require explicit support from the compiler, which we have not explored in this thesis work.

The third step, ***initializing interpolators,*** assigns a default interpolation function to each argument, based on their type. This function can be replaced later on the transition object. For integers and real numbers, we use the formula:

interpolate(start, target, t) = start × (1 ÿ t) + target × t

Composite types such as positions and colors have their fields interpolated separately as numbers. When the programming language supports polymorphism, the formula above can be used for these composite types (provided they implement addition and multiplication to a real), and thus support a large number of default types.
However, there are types for which this type of interpolation would not make sense, such as arrays or strings, for example in Figure 17. In these cases, programmers must be able to provide their own interpolation function.

Figure 17: Two alternative interpolators for character strings. On the left, the new string is inserted from the right. On the right, each modification is made on random indices.

For ***planning future changes,*** it involves asking the framework or operating system to run the animation update code at regular intervals in the future. In practice, interaction with users requires the lowest possible feedback latency, that is to say the delay between each modification and its result on the screen. This is important in many contexts, such as interacting with touchscreens [Deb15], online games [Cla06], or even digital musical instruments for which we would animate the audio signal [Jac16]. Therefore, each animation update should be scheduled as late as possible, and before any function that depends on it, as shown in Figure 18.

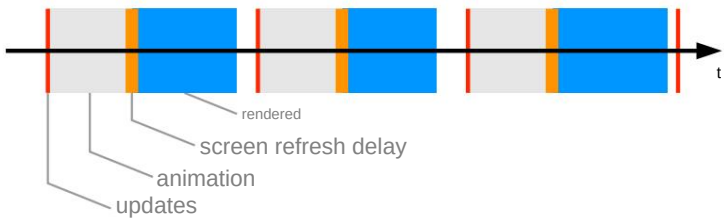

Figure 18: Simplified representation of when to insert animation updates. The operating system receives a signal from the screen when the previous frame is painted, then times out to minimize the time remaining between rendering and the next refresh. The animation code is inserted just before each rendering run.

Here, the ideal support would take the form of a ***callback*** mechanism to execute the code just before the rendering phase, such as the **requestAnimationFrame** function of the DOM in HTML [Fau16]. Finally, once planning is done, the duration operator initializes the execution state to immediately start the animation.

2.4.4 Implementation

Our functional prototype and proof of concept of the duration operator was created with the Smalltalk language, and tested with three interaction frameworks. Smalltalk is an object-oriented language, in which method calls are replaced by ***message sendings :*** the code **object x: 10 y: 20** sends to **object** the message **x:y:,** with arguments **10** and **20.** The messages are distributed dynamically, that is, the method executed for the **x:y:** message is selected at runtime. Our prototype expresses the animations with:

[object property: target] during: 2 seconds

Our extension is embedded in the **during:** message. The brackets around the mutator call create a closure , which is an object containing the code. So we can inspect the code inside without running it, and solve the four steps described in the previous part as follows:

- The extraction of message components is done by analyzing the ***bytecode*** of the closure, using the platform's introspection functionalities; To retrieve an
- initial value, we remove the colon from **property:** to get the name of the corresponding accessor. This message is then sent to **object,** and the value we get back is the initial value. Note that this mechanism does not allow functions with several arguments to be animated, because an accessor only returns a single value; Our default interpolator uses polymorphism, directly executing **(start * (1 - t)) +**
- **(target * t),** which sends the ***** and **+** messages to the start and end values. Developers can also override this interpolation code for incompatible types. Otherwise, the Debugger (a standard error mechanism in Smalltalk) is invoked during the first update of the animation; Since we do not have access to screen refresh events within the Pharo platform, future updates are recorded using the callbacks mechanism of one of the host frameworks. This limitation is discussed in more
- detail in this section.

The extension returns an **Animation** type object , which contains all the animation parameters and allows them to be modified. It is thus possible to replace the default interpolator, to specify a transition function, or even to modify the target value of the animation:

```
animation := [object position: 100@100] during: 2 seconds.
animation interpolator: [:v :t | (v first * t) + (v last * (1 - t))]. animation easing:
animation backOut. [animation to: 0@0]
 during: 2 seconds.                                    “animation of animation!”
```

2.4.5 Preliminary tests

We tested this animation extension with the three most popular interaction frameworks for the Pharo platform: (i) Morphic [Mal95], a graphical interface initially developed for the Self language, then ported for Smalltalk with Squeak, and now available in Pharo; (ii) Block [Pla15], which aims to be the successor to Morphic by supporting more input devices, and using vector rendering to support high pixel densities; and (iii) Roassal [Deh13], a visualization framework with a large selection of models and a dedicated language for expressing interactive visualizations with little code.

Our biggest challenge was planning the entertainment. To satisfy the objective shown in Figure 18, we needed to execute the animation updates before the rendering code of each of the frameworks, ideally without having to modify them explicitly. Morphic(i) provides a callback, **World defer: <closure>,** to execute a block of code before the next rendering step. We used it to plan our animation updates. Indeed, Morphic is so tied to the system that all other frameworks depend on it, which allows our extension to work for everyone. Although Bloc (ii) provides an equivalent callback, **BlUniverse defer: <closure>,** we did not want to patch our animation system for each new framework, and relied exclusively on Morphic. This raises the problem of real framework independence: ***our system is not really independent,*** it works with other frameworks because they also depend on Morphic. Ideally, this should not be the case, and the insertion of code before the rendering phase should be the responsibility of the ***programming language,*** or its ***standard library.*** This would allow third-party libraries like ours to make improvements to frameworks, without explicitly depending on them. We discuss this point at the conclusion of this section.

Using the duration operator with Morphic, we were able to animate changes in position, background color, border style, and title of a window (see Figure 19). We were also able to modify these same attributes on buttons inside the window. To allow default interpolation of character strings as on the left of Figure 17, we implemented **+** as a concatenation of two strings, and ***** string from the left. as extraction of a fraction of the

However, in practice some properties could not be animated with our system. For example, the fill text inside an input field was not editable once displayed (its accessors being designed to be used only before inserting the widget into the scene tree). The animation therefore had no effect, our system only cared about calling methods, without knowing if they worked.

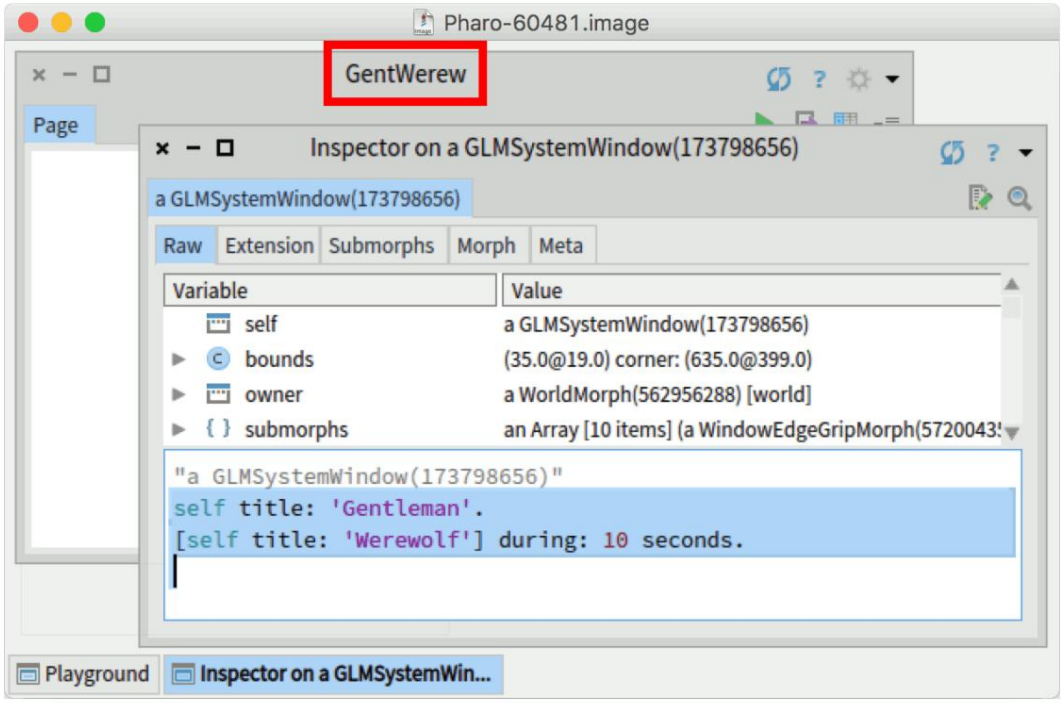

Figure 19: Introspection window open on a background window. The highlighted code is running and changes the title of the focused window from 'Gentleman' to 'Werewolf'.

Our system also works with Bloc (ii) and Roassal (iii), although Roassal sometimes violates the naming convention between accessors and mutators. For position, the widgets have a **translateTo:** mutator, and a position accessor , whose names do not match. In such a case, we had to patch Roassal with a new **position:** message referring to **translateTo:.** For color, the **color** and **color:** messages match, but the accessor does not return the same **Color** object that is passed to the mutator. In this case, we had to fix our system to always ***convert*** the **start** and **target** values to the type returned the first time by the accessor. These problems show the importance and usefulness of a system-wide naming convention: it allows our system to operate without storing a lookup table between

mutators and accessors.

A more serious problem in Roassal was that updating a widget did not automatically refresh its containing view for display. So all animations were invisible until we moved or resized the view. Although we consider this behavior to be a framework bug, we resolved it by ***manually refreshing the view*** with framework-specific syntax. However, with more work, we would fix Roassal's widgets, to automatically refresh containers in each mutator.

These problems reveal that the extensibility of frameworks is influenced by the existence and respect of common conventions. When a framework lacks consistency with respect to platform conventions (method naming, accessor parameters) and widespread practices with other frameworks (automatic refreshing of views), it is more difficult to

adapt for new uses. Highlighting and respecting these conventions therefore seems essential to us to support the development of new frameworks, and thus improve the programming of new interactions.

Finally, we added support for the duration operator on message groups:

```
[object position: 100@100. object color: Color blue] during:
    2 seconds
```

This example returns two animated transition objects in an array, which we modified to pass control messages **(start, stop, pause)** to its elements. To properly implement this functionality, we have integrated into Pharo the ability to extract all message sends from a closure, into an array of **MessageSend objects.** Our animation system then simply iterates over these reified message sendings, and for each one initializes a
animation.

The objective of this work being to illustrate a proof of concept linked to interaction in a programming language, we did not pursue the exploration of more advanced uses of animations. For example, it would have been possible to study the expression of “choreographies” of multiple animations, using sequencing and simultaneity relationships, which would have had to be integrated coherently into the language. Furthermore, although we suggest the interest of this work for managing transitions of non-visual signals (e.g. audio, haptics), we based ourselves in practice on callbacks ***linked*** to the display (see figure 18). The extension to generic signals should therefore be reserved for future work.

2.4.6 State of the art

For this work, we studied the animation APIs of six popular interaction frameworks.
Qt [Qt19] is a common interaction framework for C++, available on desktop and mobile platforms. Core Animation [App06] is Apple's standard framework recommended for OSX and iOS platforms. JavaFX 8 [Ora08] is the official successor to Swing for Java, and Android [Goo08] is another common Java framework for phones of the same name. We also chose D3.js [Bos11] for its popularity as a web visualization tool, and GSAP [Gre14] for its most flexible choice of animable properties. We were interested in the ***flexibility*** and ***compactness*** of each of these frameworks for expressing animations. Our goal was to identify their different techniques for animating properties without resorting to static functions.

The comparison is presented in Table 4. The dimensions considered were:

- ***animable properties*** — the properties supported, highlighting the techniques used to support new ones; ***animable types***
- — the types supported, with the techniques used to support others; ***syntax example*** — code
- snippets implementing a minimal example of animating **property to target** for 2 seconds.

Framework	Animable Properties	Animable types	Syntax Example
Duration operator (Smalltalk)	any property with a mutator and a accessor ***(which must have the same name)***	types supporting ***** and **+** operations ***(the others can provide interpolators)***	**[object property: target] during: 2 seconds**
Qt 5 (C++)	any integer, real, **QLine property,** providing a mutator ***(the object must inherit from*** additional ones must be ***QObject)*** supported by **QMetaType)**	**QPoint, QSize, QRect,** and **QColor** (owner types	**QPropertyAnimation a(object, "property");** **a.setDuration(2000);** **a.setEndValue(target);** **a.start();**
Core Animation (Swift)	29 default properties ***(objects must inherit from CALayer and new properties must offer an accessor and a mutator)***	integers, real, **CGRect, CGPoint, CGSize, CGAffineTransform, CATransform3D, CGColor,** and **CGImage** ***(no support for other animable***	**let a = CABasicAnimation(keyPath:"property")** **a.toValue = target** **a.duration = 2.0f** **object.addAnimation(a, forKey:"property")**
D3.js (JavaScript	**attr** properties and **style** of elements of the DOM ***(limited to DOM objects)***	***types)*** numbers, colors (various spaces), dates, numbers in text, tables, dictionaries, 2D transformations ***(other types are animable by providing interpolators)***	**object.transition() .duration(2000) .attr("property", target);**
JavaFX 8 (Java)	all properties implementing the interface **WritableValue<T>** ***(objects only need to store these properties)***	integers, reals , **Color** (additional types must implement the interface **Interpolatable)**	**Timeline t = new Timeline();** **t.getKeyFrames().add(new KeyFrame(Duration.seconds(2), new KeyValue(object.property(), target)));** **t.play();**
Android Property Animation (Java)	any property with a mutator **set<Property>()** ***(no restriction on the container object)***	integers, reals, colors ***(other types can be animated by providing interpolators)***	**ObjectAnimator a = ObjectAnimator .ofInt(object, "property", target);** **a.setDuration(2000);** **a.start();**
GSAP (JavaScript)	any property except a set of names reserved for passing options ***(no restrictions on items)***	numbers, numbers in text ***(other types are animable by providing interpolators)***	**TweenLite.to(object, 2, {property: target});**

Table 4: Comparison with animation features of six other frameworks

We observe three types of restrictions on animable properties: the need to inherit from a particular class or interface (Qt 5, Core Animation, JavaFX 8), a naming convention for property mutators (Android), and Special Keyword Reservation (GSAP). For types, we identify two groups of frameworks: those with a fixed set (Qt 5, Basic Animation), and those requiring a custom interpolation function. It is often provided in the form of a ***functor object,*** which is an object with a single method. In addition, D3 stands out for the automatic interpolation of unknown object fields, thanks to JavaScript's ability to

enumerate.

As for the syntax, some frameworks have other variations, the most complete being presented here. Almost all construct animation objects from a **"property" string.**
For GSAP, **TweenLite** retrieves the property name by enumerating the fields of its 3rd argument. D3's approach is to insert .transition() between the mutator and the receiver object, a proxy object intercepting **style** and **attr** function calls to animate them automatically.
With the concept of function animation, we achieve a syntax as concise as that of GSAP, with the only restriction that the framework must specify and respect a naming convention. The major drawback of our approach, however, is that we depend on advanced features of the programming language. Rather than simple function calls, our syntax relies on a dynamic language, with strong introspection and polymorphism capabilities.

The purpose of this work was to provide a concept that developers could ***appropriate,*** in order to encourage the use of animations when they are appropriate, and to generate original solutions diverting their use to respond to other problems. This hypothesis is difficult to confront, because the feeling of appropriation of a concept is difficult to measure objectively. We therefore limited ourselves to this proof of concept, which constitutes a building block to be assembled in a possible future language ***adapted*** to interaction.

2.4.7 Conclusion and formulation of an Interaction Essential

In Object-Oriented Programming, objects encapsulate their own state and expose it with interfaces. If they have the coordinates of a position, then they will certainly expose **getPosition()** and setPosition(Point) functions . Dynamic call animation uses this principle, obtaining a **start** value with the first function, and sending small variations between **start** and **target** to the second. Objects don't need to know ***how*** to animate a position: with a series of small modifications we simulate smooth movement towards the final value.

From this observation, we explored the concept of animation for ***all*** objects in the system, independent of a framework. In addition, we created a syntax reusing existing lexical elements as much as possible. The result is a duration operator attached to the calls
function:

object.setProperty(target) during 2s

We have presented a process and a working implementation to produce an animated transition object from this syntax. This work gives rise to two recommendations for interactive systems: (i) ***provide a mechanism for notifying the global display to the system,*** and (ii) ***encourage naming conventions for accessors and mutators.*** From these points, and our experience in exploring alternative syntaxes to express interaction, we formulate as a final Interaction Essential **to ensure the compatibility and extensibility of tools (languages and libraries) for design and prototyping of interactions.** We discuss this point in the following sections.

2.4.7.1 Using metaprogramming

Metaprogramming refers to the use of data structures that describe programs. This may involve generating code which is then interpreted or compiled, manipulating the behavior of objects with a meta-object protocol ***(Meta-Object Protocol),*** or even using the macros of certain languages (e.g. preprocessor of C). In our work, metaprogramming allowed us to integrate the expression of animations coherently with the rest of the language, as if it were part of the base language. In practice, this involved retrieving the ***bytecode*** of a compiled code block, traversing it using the platform's functions, and generating a **MessageSend** object for each enumerated message send. In a first version of this work, we expressed the animations with the syntax **object property: (target during: 2 seconds).**
We then also used metaprogramming, invoking the built-in Debugger when executing **during:,** to retrieve the call to **property:** on the message send stack.

Without the use of metaprogramming, our syntax for expressing animations would have looked like **object animate: 'property' to: target during: 2 seconds.** However, this syntax is not consistent with the method call without animation, which motivated us to extend the syntax of the language. Thus, in the context of research, metaprogramming makes it possible to ***modify the meaning of existing syntax elements,*** in order to propose new coherent forms of expression. In our case, the syntax **[object property: target] during: 2 seconds** should have meant immediate or delayed execution of the code block, which we changed to converting it to an animated transition object.

More generally, our metaprogramming needs were to ***decorrelate the syntax and semantics of the code,*** that is to say to be able to inspect the code as a data structure (not just a character string), and to ***replace certain rules semantics*** while preserving the others. Many dynamic languages today (e.g. Smalltalk, JavaScript, Python, Lua) define a meta-object protocol, that is, it is possible to intercept and change the meaning of expressions like **object.property = value,** or **object.func().** However, in our case it is the inspection of code blocks which would have been useful, and this is still poorly supported in the majority of languages.

Indeed, metaprogramming often exposes structures internal to the language, which are documented ***by*** the language developers, and ***for*** their own use. They are therefore not always accessible to third-party users, especially since they are not aware of possible side effects. In addition, certain internal details can become more complex as the language evolves, making the use of metaprogramming more difficult. This is the case for example with the introduction of new types of functions (generative, asynchronous). These new features brought new primitive types to the JavaScript and Python languages, with the result that a function can be represented internally by multiple types. For our work, we had the chance to work with a developer of the Sista virtual machine for Smalltalk, Clément Béra [Ber17]. It allowed us to implement our extension in perfect integration with the platform, in particular to identify all types of ***bytecodes*** which translated message sendings. Since this opportunity is rare, it is more likely in the future that we will have to make do with the resources available in the language. We therefore consider that it

It is essential **to use a programming language with the most specified and accessible metaprogramming possible.** Smalltalk stands out favorably from other languages on this point, and our work on expressing animations would have been more difficult (if not impossible) with most other programming languages.

2.4.7.2 Interaction with the low level

During our use of the Pharo Smalltalk platform, we designed other prototypes related to interaction programming. The first consisted of a causal syntax **[<message>] afterDo: <block>.** It was based on the work carried out for animations, and made it possible to execute a block of code systematically after a message is sent, in the manner of programming by Aspects [Kic97], and Djnn ***bindings*** [Mag17]. We demonstrated this prototype in conjunction with the animations at the Pharo Days 2017 conference [Raf17].
The second prototype was a preliminary version of the work presented in the next chapter, and consisted of a rudimentary toolbox for drawing graphic shapes on the screen. We presented it at the ESUG 2016 conference [Raf16].

For all the prototypes developed with Pharo, we often worked closely with the low-level. In practice, we wanted to access mouse and keyboard events with very low latency, therefore with the minimum of intermediaries, and above all to be certain that no queue was used. Additionally, we needed to draw simple geometric shapes and text in a window as quickly as possible. However, on this platform the native Morphic framework [Mal95] was noticeably slow, to the point of running at a frequency well below that of the screen. As for the recovery of input events, it was based on an old version of SDL [Lan98], and did not support continuous scrolling with trackpads.

We therefore undertook to use the OpenGL, FreeType, GLFW (alternative to SDL) and libpointing libraries [Cas11] directly with Pharo, because we had already used them in the past. To access these libraries, the virtual machine must call functions in shared libraries (.dll files under Windows, .so under Linux, .dylib under macOS). You must then write bindings , which allow Smalltalk code to execute these functions. For example, the native **glfwCreateWindow** function maps to the following Smalltalk method:

```
createWindow: title width: width height: height monitor: monitor share: window
    ^self ffiCall: #(GLFWwindow glfwCreateWindow(int width, int height,
        String title, GLFWmonitor monitor, GLFWwindow window))
```

We designed scripts to automatically generate the bindings for Smalltalk from the function definitions for the C language of each of the libraries. Once the bindings were made, we could open an additional window with GLFW, and draw directly in it with OpenGL and FreeType. However, because of the tight integration of Morphic into Pharo, our window should coexist with a main window. Like the operating system

sent a single stream of events for both windows, the execution of our system ***asphyxiated*** the main window which no longer received input events. So we were able to experiment using Pharo, but at the cost of ***a lot*** of low-level effort.

In the context of our work, a major drawback of the Pharo platform was its insufficient support for low-level interaction, and that it does not intend this support for any use other than internal to the platform. Obsolete bindings required significant engineering work to use newer libraries. So we invested a lot of time and learning, which we invested less in high-level research. Additionally, we found little interest in the community for contributions to low-level interaction, probably because we were improving what already existed rather than bringing major features.

The exploration of high-level concepts having led us to depend on advanced low-level functionalities, it seems essential today **to use a programming language with the most complete support for low-level interaction. possible,** so that we don't have to worry about it. This is the case for example with Java, which provides advanced functionalities through the Swing framework, such as explicit mouse control with the **Robot class.**

2.4.7.3 Compatibility and extensibility of interaction programming tools

In light of the preceding discussions, we therefore state an Interaction Essential with the following characteristics:

- provides a mechanism for notifying the global display to the
- system encourages naming conventions for software library accessors and mutators promotes
- a programming language with highly specified and accessible metaprogramming promotes
- a programming language with full low-level support

These characteristics would firstly facilitate the development of interactive applications thanks to less learning of the rules specific to each framework. With common conventions, compatibility between libraries would be facilitated, and programmers could reuse their knowledge among them. Second, developers would have to make less effort to access low-level features if they were included in the language from the outset. Finally, we conjecture that the possibility of modifying the meaning of language elements and adding new ones would allow more fine control of ***the user experience*** of the development of new interactions (compared to the use of functions of an API) . The aim here would be to free oneself from programming concepts inherited from calculation, to imagine a programming experience dedicated to interaction. We are still far from that today, because creating a programming language is a very heavy task, which requires us to modify existing languages rather than working from a blank copy. However, we hope that the accumulation of programming concepts dedicated to interaction will facilitate the eventual creation of such a language. Our third Interaction Essential is therefore:

Interaction Essential #3

Promote the compatibility and extensibility of software libraries, using low-level standard mechanisms and conventions, and a language with flexible syntax

Chapter 3. The Polyphony Toolbox

In previous chapters, we discussed the adequacy of current tools with the needs expressed by programmers of new interaction techniques, and noted that the documentation and flexibility of frameworks were crucial to promote their use in non-priority contexts. . We also wanted to experiment with the use of Web technology (here the JavaScript language) in the design of an interaction architecture, to enable a possible transfer of knowledge to the Web. We therefore chose to contribute with the creation of a new framework, ***Polyphony,*** designed for ad hoc interfaces (simple, not very robust, and created for a specific use) and flexible (whose code and appearance are easy to modify).

This work illustrates our Interaction Essentials with an ***original programming model,*** in which all interface processing (their order, their triggers) are clearly explained in the program, and can be manipulated. It is possible to observe them, reorder them, replace them, and insert new ones, in order to modify the behavior of an existing interface. We thus support an incremental development practice, which allows needs to be incorporated and refined as they emerge in a project. We also present a ***reification of interaction devices into programmable and extensible entities,*** whose introspection is facilitated (to discover the available variables), and the variables are dynamic (to accumulate higher level data).
Thanks to a mechanism of temporary variables (existing for a short period of time), we provide an alternative to event-driven programming, using entities reifying devices as data carriers for interaction techniques. This representation in turn allows us to integrate the processing of interaction techniques into those of the interface, and thus ***to unify the interface thread with the interaction thread.*** We thus achieve a simple representation of the overall execution flow, which can be represented graphically in its entirety, making it easier to understand. Finally, experimenting with a dynamic language like JavaScript consists of taking advantage of the dynamic nature of objects (their variables are not constrained to the definition of a class). We use this functionality to symbolize the addition of behavior to interface elements, and thus constitute new elements by Composition rather than Inheritance. This work therefore consists of ***translating the dynamicity of the language into the dynamicity of the framework,*** which leads us to propose new needs and developments for this language.

During our bibliographic and technological research, we considered the study and application of the Entity-Component-System (ECS) model to the programming of graphical interfaces and interactions. This architectural model comes from the field of video games, has been used there for more than a decade, and to our knowledge has not been used to model inferfaces. It seemed to us to be relevant for the prototyping of new interaction techniques. Indeed, ECS is based on the principle of Composition, which is a recurring need in HMI, for which numerous contributions have been proposed, such as Web Components [Coo14], UBit [Lec03], or Jazz [Bed04]. Additionally, ECS places significant priority on modeling the execution flow, which is designed as a ***pipeline*** — a common model in video games, which seek to maximize

the use of machine resources. It therefore provides a solution to the logic fragmentation problem stated in section 2.2.4. Finally, ECS has been successfully applied in commercial games such as Thief: The Dark Project [Leo99], Operation Flashpoint 2 [Mar07], or Overwatch [For17]. The community of developers around this model is very active today, both as users of ECS-based libraries and as developers of these libraries. Prior to our work on ECS, we identified a need, expressed mainly on online discussion forums, to see this model applied to the construction of interfaces in games. These conditions therefore encouraged us to adapt the ECS model to interaction programming in a research context.

In this chapter, we present the first step of our work, which was to apply ECS to HMI programming, to provide a clear and unambiguous architecture, and to identify the main contributions to programming. graphical interfaces and interactions. The first section of this chapter presents a state of the art on software libraries dedicated to the prototyping and programming of new interaction techniques. In the next section, we present Polyphony, an ECS-based toolkit, and introduce its use from the perspective of application programmers. Next, we detail the architecture of Polyphony, for replicability purposes. Finally we list the implementation choices characteristic of any variant of ECS, and explain those of Polyphony.

3.1 State of the art of programming tools for the search for new HMIs

This section is dedicated to presenting the state of the art and positioning our work on the Polyphony toolbox. We are particularly targeting the context of programming interaction techniques, the central theme of this thesis work, although we consider that Polyphony could be used in many application contexts. In Chapter 1, we discussed the problems encountered by HCI researchers in using interaction frameworks. They mainly relate that their uses are not the priority targets of framework developers, and that their needs are poorly recognized and supported. They therefore lack relevant documentation and examples to illustrate the use of frameworks in a research context. Major frameworks make it difficult to create non-stereotypical interfaces, both in terms of appearance and interactivity. To this end, we observed that researchers expressed the need to appropriate their tools, or even to divert them to create artifacts for which they were not intended. Finally, to integrate contributions to existing interfaces, such as new interaction techniques, researchers encountered a lack of flexibility in the frameworks. The state of the art presented here should allow us to identify the solutions that have been proposed in response to the problems above. We analyzed the work according to the following questions:

- What new uses have they introduced/allowed?
- What concepts and new ways of programming did they propose?
- How flexible are they in changing predefined behaviors?
- How do they integrate with their programming language or interaction library?

HMI research has led to a large number of toolboxes, covering many uses. We do not pretend to list them all here, especially since our subject of study may lead us to consider many of them. Readers interested in a recent and very complete state of the art can refer to the thesis manuscript of Vincent Lecrubier [Lec16]. We present here the most significant works, which allow us to situate the contributions of Polyphony to the programming of interaction techniques.

3.1.1 djnn et Smala

djnn is a complete framework for building graphical interfaces, developed at LII of ENAC since 2014 [Mag14, Cha15, Rey15, Cha16]. Its initial context of application is the specification of interfaces in the field of aeronautics, for which verification and certification of programs are common and necessary. In addition, it is characterized by multidisciplinary teams, in which it is common for the appearance of interfaces, their logic, and their interaction techniques to be developed by separate teams. djnn is built around iterative interface design, using a rigorous development paradigm: each program is a tree of interactive components, whose interactions define the execution of the program. ***All*** interactive elements are modeled by these components (widgets, input/output devices, or even calculation formulas), and are included in a hierarchy of components. The XML format representation of this tree is used as an interface between the different teams.

djnn is based on the theoretical model I* developed by Stéphane Chatty a few years earlier [Cha07, Cha08, Cha12]. This model is centered around the definition of ***processes,*** and the causal relationships between them. It defines the concept of ***binding,*** which connects a source process to a dependent process, propagating any activation from one to the other. This concept is taken up and developed in Smala, a programming language based on the djnn framework, and transpiled to C [Mag18]. It introduces two causality operators, process coupling and ***data-flow*** link (see figure 20), supported by a concise syntax. In addition, it introduces a way of expressing interfaces halfway between immediate code execution and alternation of declarative models.

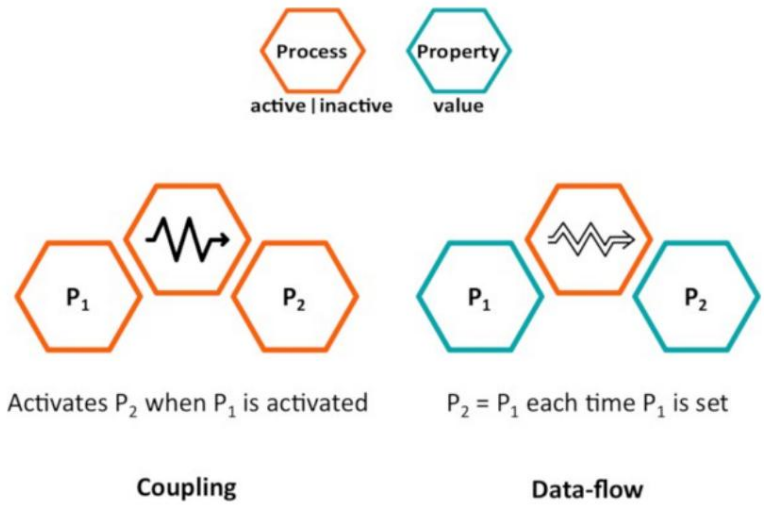

Figure 20: The causality operators of djnn/Smala (figure taken from [Mag18]).

djnn provides a reduced set of unifying concepts, with which its authors manage to express complex interfaces. This framework invites programmers to think about an interface as a set of interconnected processes, and causing the activation or deactivation of nodes in the scene graph. By defining causal relationships, it is easy to propagate changes of state, which is made even easier by Smala's syntax. The distinction as well as the close relationship between djnn and Smala perfectly illustrates our idea of integration between framework and language. Here, djnn is the framework, which is accessed by an API in C. Smala is the language (here specially designed for djnn), whose interest is to be a syntax for djnn. The work on Smala made it possible to associate concise and clear syntax elements with concepts formerly expressed by function calls (in particular bindings). However, although the Smala language is particularly suited to defining a new interface, it does not demonstrate the modification of an existing interface. This type of flexibility is part of the needs linked to the prototyping of new interaction techniques, and requires introspection of all elements of the interface. However, it is to be feared that modeling all the causal relationships of the interface into binding objects will generate a large number of them, and make their introspection difficult.

3.1.2 Murder

UBit ***(Ubiquitous Brick Interaction Toolkit)*** is a toolbox based on the C++ language, implementing a “molecular” architecture to build HMIs [Lec99, Lec03]. It was designed from a simple model offering a lot of flexibility, in order to support the development of new interaction techniques, in varied contexts such as distributed or multi-user interaction. This model responds to the “monolithic” nature of interfaces based on widget classes, for which the behaviors of each widget are predefined in its class.

UBit uses a scene tree, in which interface widgets compose their appearance and interactivity using bricks , small semantic units acting like words in a sentence (see figure 21). It advantageously uses sibling relationships in the scene tree, which are often little highlighted, and also allows “recursive replication” by which a node has several parents for which it is duplicated. Additionally, UBit integrates closely with C++ by redefining the arithmetic operators **+** and /, which allow it to initialize a scene tree recursively, without having to construct the nodes in intermediate variables.

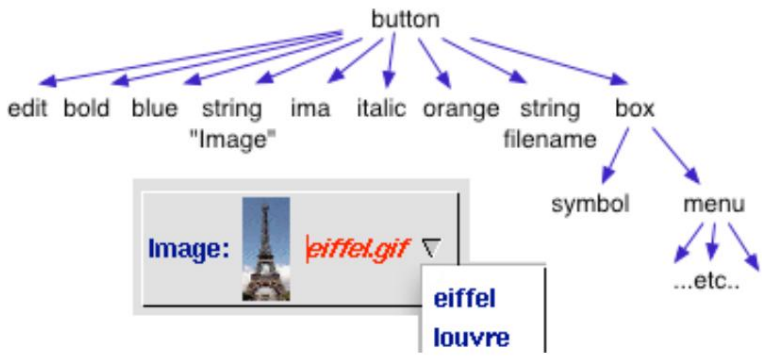

Figure 21: Illustration of an assembly of bricks in UBit (figure taken from [Lec03]).

However, the fine granularity of the concept of “atomic” bricks means that any interface of modest size contains a large number of bricks. It can then be difficult to visually represent the scene graph, and to manipulate it directly, for example to extend an existing application or modify parts of it. In addition, UBit having been designed to reproduce WIMP interfaces, the choice of different types of bricks and their possible combinations were designed to reproduce standard controls (buttons, check boxes, menus, etc.). Adding a new type of brick must take into account possible interactions with existing bricks, which makes it difficult to extend UBit to new types of controls. In the case of new types of interfaces (e.g. Virtual Reality, ubiquitous interfaces), it would potentially be necessary to redesign all the building blocks.

3.1.3 Amulet/Garnet

Amulet is a framework for building graphical interfaces and interactions based on the C++ language, which was a pioneer in HMI programming innovation [Mye97].
Developed over more than a decade, it was characterized by widgets based on a prototype object model, the integration of a system of constraints that could be attached to all variables, the management of user input by objects Reusable ***“Interactors” ,*** and advanced possibilities for inspecting and modifying the interface at runtime. These characteristics are still ahead of current HMI programming frameworks today. Amulet is the successor to Garnet (shown in Figure 22), a framework and development environment based on Common Lisp and X11 [Mye90].

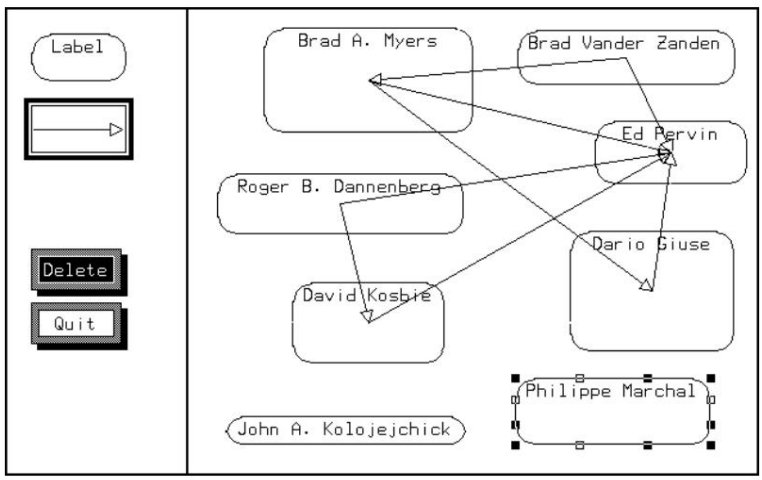

Figure 22: Example of interface built with Garnet (figure taken from [Mye90])

Amulet was specifically developed to support GUI research. Flexibility and extensibility were therefore central themes, integrated into its innovative model of interface construction. It was used to explore topics such as distributed interaction on the Internet network, support and expression of animations, support for the ***undo action,*** graphical debugging of interactive applications, creation of a graphical tool for interface definition (Gilt), instantiation of widgets by freehand drawing (SILK), and programming by demonstration (Gamut). The Amulet prototype model consists of each object having a parent (its

prototype), which implements the object's methods (just like a class would). However, unlike classes it is possible to add or remove fields from each object (independently of the others), and to change prototype. We can therefore consider prototype-oriented programming as a more flexible alternative to classes. Amulet is one of the rare works to have explored the use of a dynamic language (in the modern sense, we mean the typing and topology of objects), for the construction of graphical interfaces. The prototype model was implemented in C++ using functions, without using metaprogramming features, which explains why it suffered from poor performance.

Today, some of the innovative features that Amulet helped to develop have become more widely available (Cassiowary constraint management [Bad01], introspection of interfaces with web browsers), while the others remained in the domain of research. It is possible that the rise of dynamic languages such as Python and JavaScript will allow the development of frameworks exploiting prototype objects, which we also drew inspiration from in this thesis work.

3.1.4 SwingStates et HsmTk

SwingStates is a toolbox based on the Java language, integrating with the native Swing framework to apply the state machine formalism [App06]. It was designed to structure the control flow of graphic applications, the complexity of which has long been recognized as problematic, for the maintenance and introspection of interfaces [Mye91]. It is also notable for the development of the notions of ***tags,*** inspired by Tcl/Tk, and popularized by CSS1 classes [W3C96]. State machines (or ***finite automata)*** are a mathematical model used to describe the operation of a system as a sequence of states, and the execution of transitions between these states. Applied to HMI programming, they make it possible to describe interaction techniques in a rigorous and systematic way — in particular to identify and clarify unexpected interactions (e.g. simultaneous pressing of two buttons in Figure 23).

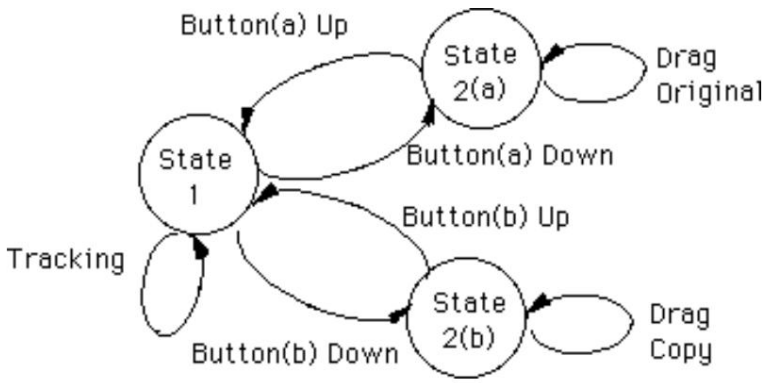

Figure 23: Illustration of a two-button interaction technique modeled by a state machine (figure taken from [Bux90]).

HsmTk is another toolbox, based on C++, this time applying the formalism of hierarchical state machines [Bla06] (see figure 24). It is integrated both as an extension of the C++ language (to define the logic of automata), and as an extension of the SVG format (to

integration of logic into interface specification). HsmTk is notable for seeking to escape the widget model, reusing existing, proven, and well-optimized technologies (SVG and C++ syntax), to design innovative graphical interfaces.

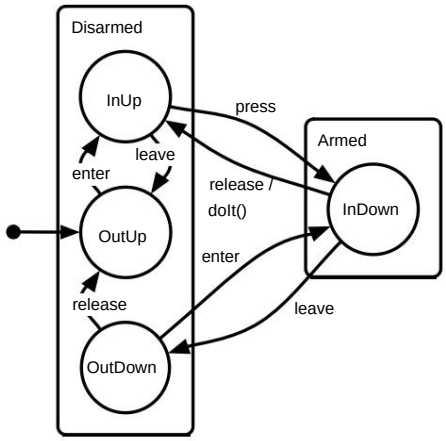

Figure 24: Example of a hierarchical state machine (figure taken from [Bla06]).

SwingStates and HsmTk have the common points of having both been developed — independently — at LRI (Paris-Sud University), and of applying the principle ***of interaction as a first-class object*** of Beaudoin-Lafon [Bea04]. State machines are a robust and proven paradigm for modeling interaction techniques, yet they have rarely been used for prototyping — including academic research. Indeed, automata are best expressed through visual diagrams, and are often transposed into text in a verbose manner. In addition, they are quickly limited in the complexity of the interactions that they can model, in particular when it is necessary to consider the combination of several states, continuous transitions between states (animations), or the expression of ***feedback*** and ***feedforward.*** These needs involve adding states, and therefore increase the complexity exponentially. For these same reasons, state formalisms are generally very limited in their capacity to change the number of states and transitions. State paradigms are however widely used in areas where the interaction must be formally specified and deterministic, such as critical systems, with ICO/Petshop which is based on Petri nets [Nav09]. Finally, InterState is notable for having provided a unified graphical representation between state machines and object data (see figure 25), in order to specify reusable behaviors in the form of state machines [One14].

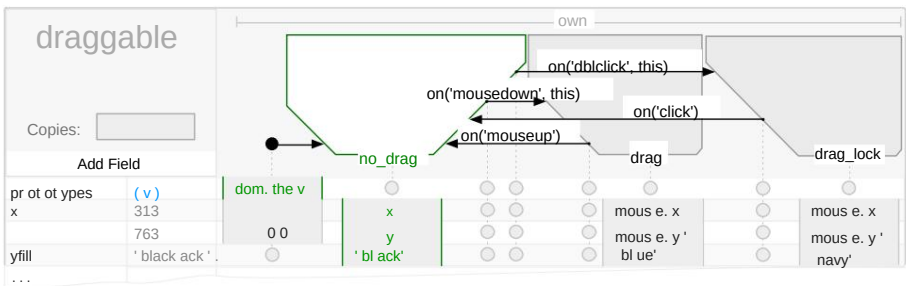

Figure 25: Example of drag-and-drop behavior implemented in InterState (figure taken from [One14]).

3.1.5 ICON and MaggLite

ICON is a toolbox dedicated to manipulating an application's input devices, and specifying interaction techniques [Dra01, Dra04]. It is centered around an interactive graphical representation of data flows , in which input and output slots are connected to establish permanent flows (see figure 26). ICON was created to ***adapt*** the richness of input devices to the limited interaction modalities of office software, in order to enable the effective use of peripherals deviating from the stereotypical framework of the mouse/keyboard pair. It is built primarily for the Swing framework, but also partially works for external applications.

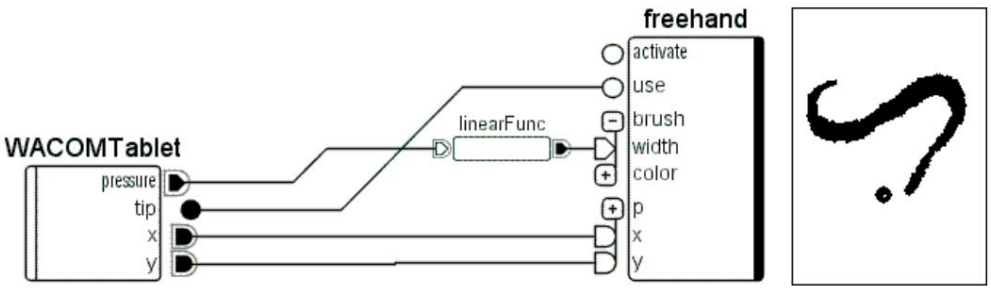

Figure 26: Example of an input configuration in ICON (figure taken from [Dra04]).

ICON is notable for having been used for over a decade to manage entries on the LRI's WILD screen wall [Bea12]. In addition, it demonstrated the applicability of defining relatively complex techniques, through a visual programming language based on flow diagrams (boxes, slots and connections). The very structured visual appearance allows it to display a significant density of information while remaining readable, something generally difficult with systems like Max/MSP or Pure Data. However, although the data flow approach is suitable for continuous aspects of interaction, it is less suitable for describing states and transitions between states. Thus, the hybrid approach FlowStates was proposed, which combines data flows from ICON with state machines from SwingStates [App09]. Interfaces built with ICON stand out for the flexibility of connections between boxes. It is possible to link connectors on the fly, and disconnect them, without disrupting the operation of the system. Developers can thus plug in a device and make the necessary connections to adapt it to an existing application, all while the application is running. Finally, ICON being a visual programming system, there is no question of integration into a language, since it is not used from the functions of an API.

MaggLite is a freehand interface building toolkit (shown in Figure 27), developed in conjunction with ICON and using it as a basis for handling input events [Huo04]. It is explicitly dedicated to the exploration and definition of new interaction techniques. By a ***combined graph*** model [Huo06] bringing together the interaction graph (from ICON) and the scene graph, it clearly separates the appearance and logic of an application, giving increased importance to the management of interactions (compared to a system based on

propagation of events). MaggLite is also notable for highlighting the formulaic nature of WIMP interfaces, demonstrating the use of unconventional peripherals, research-based interaction techniques, and unusual visuals.

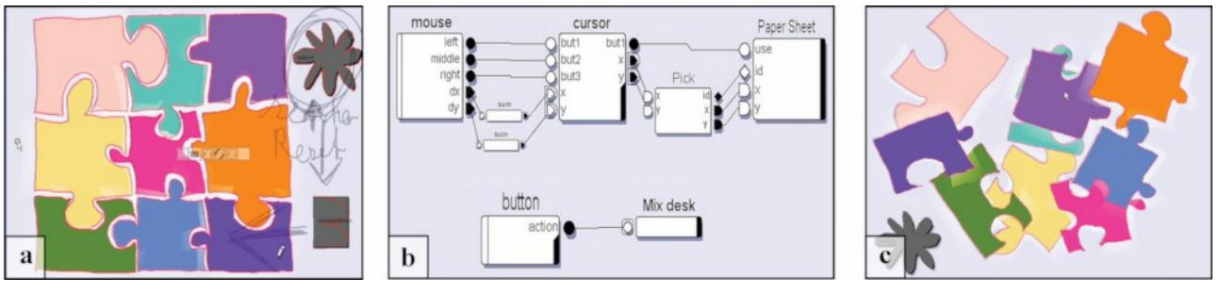

Figure 27 : Illustration du principe *"Draw it, Connect it and Run it"* de MaggLite (figure extraite de [Huo04]).

Although ICON and MaggLite helped to undo the inflexible and conventional nature of interfaces, the principle of interaction graphs was little replicated subsequently. However, interaction peripherals and techniques have continued to diversify, but also to gain in degrees of freedom and parameters to control. Interaction graphs show their limits in the expression of complex techniques which are difficult to read, as well as the lack of integration in the languages commonly used to prototype graphical interfaces.

3.1.6 Proton and Proton++

Proton is a toolbox facilitating the expression of ***multi-touch*** interaction techniques on touch screens [Kin12]. It addresses the difficulty of quickly prototyping new finger interactions, particularly when they involve sequences of gestures and combinations with several fingers. Proton models finger techniques using regular expressions, which are formed by assembling three types of actions (press, move, release), to which finger number and targeted object attributes are added (see figure 28). It also provides a graphical ***tablature editor,*** inspired by musical notation. Proton++ is a continuation of the work done on Proton, which allows new types of parametric actions to be defined, and their parameters to be specified by an extension to Proton regular expressions [Kin12].

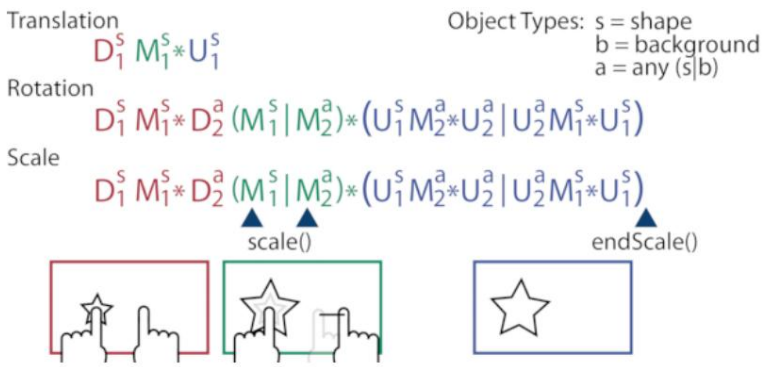

Figure 28: Illustration of Proton regular expressions (figure taken from [Kin12]).

Proton and Proton++ are notable for bringing the expression of interaction techniques closer to the definition of regular expressions, providing new ways for developers to reason about these artifacts. In addition, thanks to their definition of a concise and unambiguous syntax to describe complex gestures, they allow reasoning at a higher level to prototype gesture control, without having to get lost in often verbose code.

However, both toolkits rely heavily on the absence of ambiguities in atomic gestures. In fact, it is easy to differentiate a finger press from a finger movement, both for the program and for the user. The gestures added by Proton++ respect these same properties: they are highly differentiable from each other. However, ambiguities are common in gesture interfaces, for which classic recognition systems quantify a recognition probability. In general, we argue that each interaction technique may require fine control, particularly when it evolves with current uses.

For example, it is possible that users performing a pinch gesture ***on*** an object to enlarge it, start the gesture in the air and thus position the first points of contact outside the object. For an interaction technique that is robust to this type of use, it is appropriate to detect ambiguous gestures, but the reduction to regular expressions of Proton and Proton++ limits this control.

3.1.7 subArctic/Artkit

subArctic is a complete framework for programming graphical interfaces and interactions, mainly built around the notion ***of extensibility*** [Hud05]. Developed as Garnet over over a decade, and to support research, it has been used to explore topics such as the abstraction and implementation of animations, the implementation of locally propagated positioning constraints (µConstraints), the interactive visual debugging of graphical applications, and more generally the implementation of non-standard interfaces. This work has contributed to improving the functionality and appearance of graphical interfaces, and is present today, in different forms, in many frameworks. subArctic is the successor to Artkit, an interaction framework based on C++ [Hen90].

When it comes to programming new interaction techniques, subArctic is mainly characterized by a comprehensive model of extensible inputs (see Figure 29). This model allows the definition and addition of new ***interactors,*** at both compilation and execution, which interpret higher-level behaviors in sequence. It is thus possible to integrate new interaction techniques into existing interfaces, or to replace existing interaction methods. This model is analogous to the one we implement in Polyphony, and which we present in the following sections. Finally, since the subArctic toolbox does not use language integration techniques, it is used via a standard API for Java.

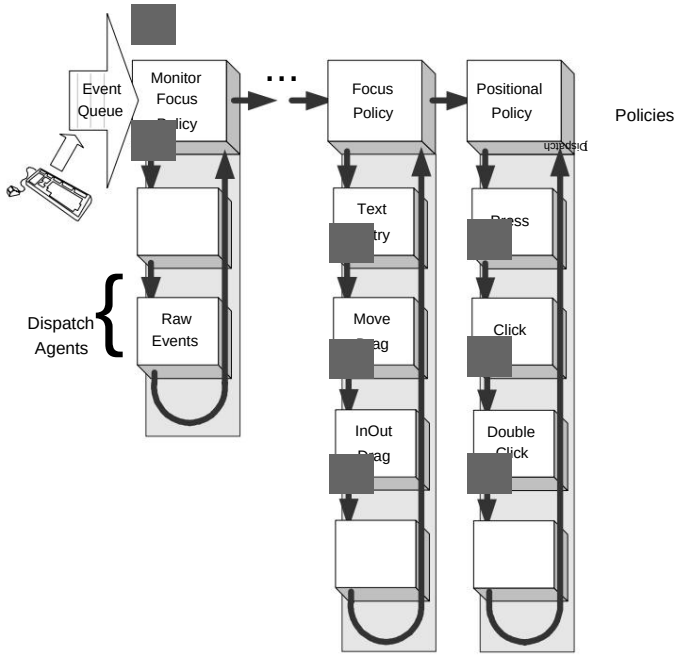

Figure 29: Illustration of the subArctic event model (figure taken from [Hud05]).

3.1.8 D3/Protovis

D3 is a library designed for creating interactive visualizations on the web, which provides a dedicated JavaScript-based language for manipulating SVG included in the DOM of a web page [Bos11]. It is based on a pragmatic approach to creating visualizations: being the most ***efficient*** (less effort to produce a visualization), ***accessible*** (easier to learn), ***expressive*** (variety of possible visualizations, see figure 30), and ***efficient*** (execution time). D3 is notable for its deliberate integration with existing technologies (SVG, DOM), without introducing a new representation or extending an existing format. The authors of this library openly criticize the use of intermediate representations for contravening the previous objectives. They justify their choice for: the low learning required for users familiar with the Web, the large base of Web users who can quickly adopt D3, the already numerous and complete documentation on DOM and SVG, the benefit of DOM debugging tools present in web browsers, and the absence of reduction in expressiveness due to translation from a different format.

D3 was a success in use, as it was widely adopted to produce interactive online visualizations, and helped to undo the stereotypical nature of visualizations, reducing their designs to drawings of simple shapes in SVG. The concept of data ***linked*** to graphics connects the flexibility of data with that of visualizations. However, D3 does not expressly seek to extend the capabilities of interaction devices, other than by proposing new ways of manipulating the data present in the DOM. Its interest in programming new interaction techniques is therefore less, although its pragmatic approach to validating contributions by the community is an example to follow. D3 succeeds Protovis, a complete visualization framework based on JavaScript [Bos09].

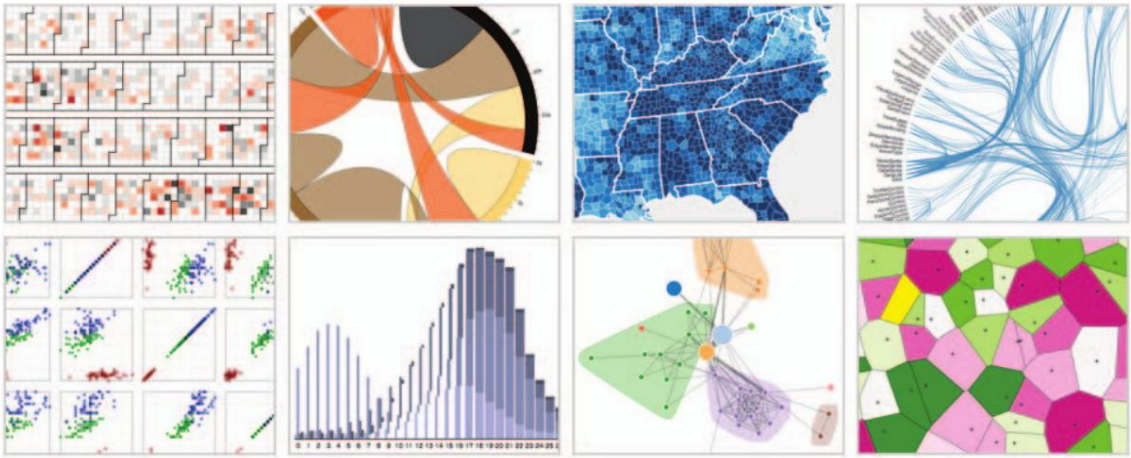

Figure 30: Examples of visualizations built with D3 (figure taken from [Bos11]).

3.1.9 Limitations of the state of the art and opportunities for contributions

From the work presented above, we observe a number of recurring patterns as we focus on the questions posed at the beginning of this section. First of all, most meet the **needs of information propagation.** This information can be data, or more abstractly “occurrences of events”. This may involve propagating mouse click information towards triggering the command of a button, or propagating mouse movement information on a positioning constraint during a press-release. These needs do not exactly correspond to the paradigm of updating states in memory put forward in hardware architectures and imperative programming, because information must pass between processing stages, rather than being archived in memory. This is why many toolboxes have introduced information propagation paradigms ***(dataflow, bindings,*** constraint systems). Since the toolboxes are mostly implemented with imperative languages, these paradigms must coexist with different types of reasoning.

We consider that this situation is a source of confusion for developers, and **have chosen instead to adapt the propagation of information to the imperative model** (in particular with the storage of state changes in “temporary variables” presented in section 3.2. 5).

Next, the majority of jobs we studied require the **construction of a scene tree.** This type of structure is very widespread in interaction frameworks, and very useful for providing coherent global behaviors (e.g. flow rules for positioning elements in their containers in HTML), for sharing the modeling of complex relationships (e.g. . the order of display and the order of reception of mouse events), and more generally to master the complexity of realistic interfaces. The scene tree requires reasoning about the construction of interfaces by recursive nesting of rectangular elements, and in return allows the reuse of standard controls (widgets), implemented as nodes of the scene tree. However, the scene tree is not strictly necessary for producing small applications. For example, to draw an object following the cursor on the screen, most toolboxes would need to provide a drawing canvas at the size of the screen, and execute drawing instructions on it. The need for a scene tree thus hinders the ***incremental*** nature of the prototyping activity, mentioned in section 1.3.1.2.

According to the Cognitive Dimensions of Notations [Gre96], This point represents a significant cost of ***Premature Commitment,*** in that developers must first build a minimal scene tree (along with the base code to initialize the library) before they can build with it. This cost is accentuated by the introduction of notions specific to scene trees, of which a certain understanding must be acquired beforehand. Finally, we conjecture that the use of a scene tree, as well as concepts aimed at simplifying the different types of nodes that exist, results in the instantiation of a greater number of nodes. Without being able to support this assertion, **we nevertheless sought to question the need for a scene tree, and studied the different types of relationships that benefit from it.**

Many works offer **close integration with a programming language** (djnn with Smala, UBit with C++, D3 with JavaScript, even Proton with regular expressions), beyond functions accessible via an API. They often make use of metaprogramming features, to ***twist*** the rules linking syntax and semantics, and superimpose new ones. For example, UBit redefines the operators **+** and /, which respectively express arithmetic addition and division, to make them respectively express the addition of a sibling in a tree, and the attachment relationship of a ***callback*** to a type of trigger. Other toolboxes such as InterState, ICON, or Proton provide graphical interfaces for visually programming interaction. They thus have finer control of the syntax elements that developers will use to express interactive behaviors, without depending on the pre-existing rules of a language. During the design of Polyphony, we sought **to support integration with the JavaScript language, making the notion of Entity coincide with that of objects, so that the syntax of Polyphony seems "native" to JavaScript,** rather than "superimposed" like can be for example UBit.

We will now discuss salient points, for which we have voluntarily distinguished ourselves from the state of the art presented here, and more generally from the interaction frameworks used for HCI prototyping. We identify opportunities for contributions, which form the origin of our work on Polyphony.

3.1.9.1 Pooling and reuse of behaviors

One way to differentiate HMI construction libraries is to analyze their management of interactive ***behaviors .*** We consider this notion of behavior at two interdependent levels. At a high level, they refer to the observable reactions of Entities to different stimuli.
These stimuli can be external events (mouse click, rising clock edge), or internal events (the mouse hovers over an element). At a low level, behaviors designate the variables and code triggered by these events, and which give rise to the observed reactions. For example, the behavior "allow text editing" refers at high level to the display of characters in sequence during each keyboard press, and at low level to the code which after each keyboard entry obtains the targeted object by the keyboard, and adds each character at the end of a character string whose display is updated.

Most object-based libraries use the notion ***of an inheritance tree*** to organize individual and shared behaviors, and can be described as "monolithic" [Bed04]. The inheritance tree establishes a hierarchy of types between objects, according to

which any object descending from another inherits the definitions of methods and variables (therefore the behaviors). Thanks to this property, the descendant type can be used where the parent type is expected — this is ***polymorphism.*** In HMI applications, the behaviors shared by this tree are mainly: the appearance of widgets, their propagation of sizing constraints to neighbors/children, and their reactions to mouse/keyboard events. This reuse is particularly useful for creating new types of widgets, which can build on existing widgets (inherit their class), and add functionality rather than reimplement everything.

However, hierarchies between ancestors are generally predefined, making it difficult to assign behaviors to objects that have not been specifically designed and implemented to receive them [Bed04, Lec03, Mye97], both statically (at compile time) and dynamically (at runtime). For example, it is often difficult to add text editing behavior to interface ***labels*** (e.g. **JLabel).** Indeed, the functions for ***receiving keyboard presses, displaying a flashing text cursor*** and ***modifying a character string*** often belong to another family of components, dedicated to text editing.

For a programmer with access to the source code of the interface, this change can be done in two ways: either with a subclass of a ***text field*** modified to resemble a label, or with a subclass of a ***label*** modified to allow text editing. In both cases one of the two behaviors is recreated because it cannot be inherited. This problem has given rise to object architectures favoring ***Composition*** rather than ***Inheritance,*** such as Traits [Cur82] or mixins [Bra90]. However, these architectures have been little used to develop HMI libraries, which we attribute to the constraints they impose to maintain their consistency. For a programmer who does not have access to the source code of the interface, adding behavior to an element amounts to changing its ***nature,*** which is sometimes impossible. As Lecolinet [Lec03] points out , “ behaviors ***and other features are not seen as general services that could be used by any widget ”.***

Many so-called “polylithic” libraries [Bed04] have been proposed, linked to the Composition paradigm, to share behaviors without an inheritance tree, statically or dynamically. UBit [Lec03] does so by synthesizing composable behaviors and properties in the “bricks” of a scene tree, whose recursive interpretation materializes the interface. Likewise, Jazz [Bed00] models behaviors by nodes inserted between the elements of the scene tree. The behaviors are thus inherited by all the children of the tree.
MaggLite [Huo04] takes a similar principle, adding a ***combined graph*** model which links the ICON interaction graph to that of the scene graph [Huo06].

The majority of these approaches are based on the existence of a ***scene tree*** to supplant the inheritance tree proposed by object-based languages. The scene tree, which is already used to structure elements visually, lends itself well to inheriting visual styles — graphical properties from UBit, nodes from Jazz. However, it is not suitable for ***all*** properties. CSS, for example, indicates for each type of property whether it is inherited by default or not [Moz19]. Indeed, the text color **(color)** can be shared with the children of an element, but not its margins **(margin/padding).** The scene tree is therefore not a universal solution, especially since its exclusive use requires

many knots, which make its visual inspection difficult. As we stated previously, our objective is **to explore approaches that are not based exclusively on a scene tree, in particular using composition.**

3.1.9.2 Dynamicity of behavior

Most HMI libraries have limited support for adding new behaviors to objects at runtime. It is also difficult to alter behaviors on the fly (e.g. ***enable/ disable),*** unless they were intended to do so. In a graphical interface, for example, drop-down menus do not allow their elements to be reordered with the mouse. Making the elements of a menu orderable during program execution is an example of dynamic behavior.

It is important to note that this dynamicity is independent of the choice of Composition or Inheritance to share behaviors. It's actually about being able to change the components or parents of a given object ***at runtime .***

A common solution is for each element to have the targeted behavior, and to disable it by default. This is the case for example in Qt for the example cited: the base class **QAbstractItemView** contains a **setDragEnabled** method which allows each list to activate the possibility of moving the elements with the mouse. This approach is however limited by the fact that the addition of new behaviors must be done by the base class.

This limitation has led researchers to develop various techniques to attribute behaviors on the fly. Scotty [Eag11] allows for example to analyze the structure of the interaction components of an existing Cocoa application and to inject code to modify their behavior. Although spectacular and effective for the rapid prototyping of new interactive behaviors in existing applications, this approach presents robustness and persistence defects which do not allow it to be used on a larger scale than for temporary “hacking”. Ivy [Bui02] is a messaging bus on which agents can send text and register to listen to messages (selected by regular expressions). In this way, new behaviors are added through new agents, and the agents replace each other by sending compatible messages. Amulet [Mye97] is based on a model of objects with dynamic ***slots*** (variables and methods), which seem to allow the addition of behaviors on the fly.

However, specifying a new method does not guarantee that it is executed automatically, the correct prototype must be inherited, and the prototype hierarchy is not editable.

Most work managing the attribution of new (non-predetermined) behaviors on the fly comes up against the ***need for specification.*** Indeed, objects are passive by nature, they only execute if they are “called”, by a function or by sending a message. Programs are passive too, the operating system launches them by executing an expected function (usually **main).** To execute, a behavior/object/program must therefore provide a specification (or interface), which indicates how to execute it, and with what arguments. When it is defined ***before*** the code that is to call it, it can impose its specification, and it will be up to the caller to comply by calling the correct functions. When defined ***after*** the calling code, the specifications necessarily have

been already fixed. For example, for an object to be displayed in JavaFX it must implement the **onDraw method,** which will be executed by the framework for any object in the scene graph. Any other drawing method will be ignored, because the system cannot otherwise know that it is used for drawing.

The works studied are limited by the extensibility of the specifications they provide — Ivy by the format of messages exchanged on the bus, and Scotty by the unchangeable architecture of Cocoa applications. **The ad-hoc definition of new specifications, and the adaptability of these specifications to pre-existing objects,** are functionalities rarely present in object-based languages (apart from the notion of ***structural typing*** implemented in the Go language). They have been little explored in HCI libraries, and provide opportunities for contributions to this thesis work.

3.1.9.3 Behavior orchestration

The triggering and order of execution of functions on interactive elements is important. This may involve executing a function systematically before/after another, after a given change of state, or upon receipt of events from one or more input devices. HMI design is full of such complex use cases. For example, the graphical rendering of elements of a Web interface involves the drawing of backgrounds, borders, text, or even drop shadows. Using the painter's algorithm, the rendering steps for each element must run in a specific order: the background ***before*** the border and text, and the drop shadow ***before*** the background. Another example, certain interaction techniques rely on sequences of actions, potentially coming from several input devices, and which can implement pause delays (such as ***CTRL + mouse click + wait of 500ms).*** Again, languages and libraries have limited support for these needs. Function calls provide sequential and static orchestration of different blocks of code, and callback mechanisms record hundreds or even thousands of links for “realistic” interfaces [Mye91].

To overcome these limitations, different models have been proposed. Models based on the state-transition formalism (state machines) of SwingStates [App06] and HsmTk [Bla06] make it possible to limit the “burst” of the code defining interactive behaviors, while offering a representation of the interaction close to that of designers and programmers. Dataflow models have also been proposed, which execute blocks of code when all the data on which they depend is available. Among these, ICon [Dra01] reifies dependency links into graphically represented arcs, which users can manipulate directly, for example to add or replace an input device. The two paradigms have even been united in FlowStates [App09] in order to take advantage of the two formalisms at the levels where they are most suitable.

Model-Driven Engineering approaches like MARIA [Pat09] and UsiXML [Lim05] were also used to abstract the definition of interfaces in relation to the interaction modalities available on each machine (devices, operating system), thanks to transformations between models of different levels of abstraction. They allow the interface to be adapted to the context in which it is used, whether it is the size of the screen (e.g. ***responsive*** websites), or the physical and emotional state of the user. [Gal17, Gaj10], or simply to explore an interface design space [Lei15]. In the context of gestures and sequences

stock, Proton [Kin12] is notable for its use of regular expressions to represent sequences of gestures, and GestIT [Spa13] is notable for its modeling of complex action sequences using composition operators.

What most model-based approaches have in common is that they represent micro ***-dependencies*** within interaction techniques — they make “do B after A” relationships explicit, rather than with implicit code sequencing. In doing so, they tend to require a lot of code, and are difficult to scale to handle complex interfaces and interactions. In ECS, the orchestration of behaviors is done at the ***macroscopic level,*** not on data but on processes. In this sense, we consider it complementary to model-based engineering approaches, which could use it as a target of lower level of abstraction. The ECS model is in fact aligned with the execution of algorithms linked to video games (3D rendering, physical simulation), which quickly and regularly process large volumes of data. It encourages us to consider behaviors as processes that transform input events into output events. As Chatty et al. write it to present djnn, “ Like ***computations can be described as the evaluation of lambda expressions, interactions can be described as the activation of interconnected processes ”.*** Polyphony stands out by adding attributes (Components) to processes (Systems), so as to explicitly deal with the dependencies between them, without constraining the choice of attributes to be given. The orchestration of an application can thus be extended and improved by the introduction of new components.

Finally, few works have succeeded in overcoming the “fragmentation” of logic in interactive applications, highlighted by Myers [Mye91]. Many of them implement ***small*** functions, which carry out minimal behaviors reduced to a few lines of code (e.g. state machine transitions, or propagation of constraints between variables). The use of these small functions is encouraged by the practice of ***callbacks,*** which consists of registering a function to be called when an event occurs. The programs then make numerous “jumps” between functions, which make their operation difficult to observe, in addition to being inefficient on modern processor architectures. These problems justify **the study of models based on macroscopic processes, and ruling out the use of callbacks to reduce the fragmentation of logic in interactive applications.**

3.2 Programming interactions with Polyphony

In this section, we present the Polyphony toolkit, based on the ECS programming model, and suitable for prototyping graphical interfaces and interaction techniques.
We first describe the basic concepts of ECS and then illustrate the use of Polyphony from the perspective of an application programmer.

3.2.1 The Entity-Component-System model

ECS (sometimes called CES) is a paradigm designed to structure code and data in a program, which appeared in the field of video game development [Leo99, Bil02, Mar07]. In this context, teams are typically separated between programmers designing logic and tools, and designers producing script or multimedia content. ECS designs the

Components as the interface between these two worlds: programmers define the available Components and create the Systems that will iterate on them; designers instantiate content with Entities and associate behaviors with them by assembling selected Components. The elements constituting the model are therefore:

Entities

These are unique identifiers for each element of the program (often simple integers). They are similar to objects, but do not have ***data or code.***

Components

They represent program data (like **bounds** or **children),** and are dynamically associated with Entities. Depending on the interpretations, a single Component can designate the ***type*** of data attachable to all Entities, or its ***instance*** attached to an Entity.

Systems

They execute continuously following a predefined order, and each represents a reusable behavior of the program. Entities do not register with the Systems, but rather acquire the Components necessary to be “seen” by them.

The general operation of an ECS-based application is illustrated in Figure 31. Components are the heart of the program, they define both the attributes **(shape, backgroundColor)** and the capabilities of the Entities **(clickable, draggable).** Entities acquire behavior through the acquisition of Components. Finally, it is the Systems that execute each type of behavior, selecting Entities from their Components. For example, a Background Drawing System would select all Entities that have the Components **bounds, shape** and **backgroundColor,** and for each one would draw a background shape on the screen. We represent the Systems in an order of execution common to most interaction frameworks: management of input devices, interpretation of interaction techniques, processing specific to the different widgets, processing specific to the application, and rendering of outputs (especially on screen).

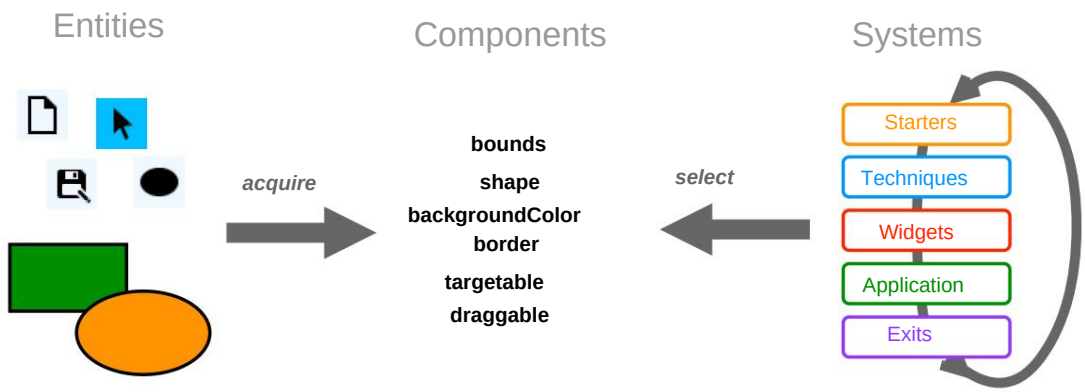

Figure 31: Illustration of high-level exchanges between Entities, Components, and Systems.

In addition to the three basic elements of the model, there are four recurring elements in the majority of ECS implementations, although they are not part of the basic model and do not have fixed terminologies:

Descriptors

These are conditions based on the Components owned by each Entity. They are analogous to dynamically evaluated ***interfaces ,*** and are the fundamental mechanism by which Systems know which Entities to operate on.

Selections

These are containers based on Descriptors, and dynamically updated as Entities acquire or lose Components.

Context

It represents the current ***world*** to which all Entities belong, stores Components and global variables, saves and executes Systems, and provides an interface for obtaining Selections.

Entity Factories

These are predetermined templates, used to instantiate Entities with a set of Components and default values.

3.2.2 Presentation of Polyphony and the development prototype

Polyphony is an experimental ECS-based software toolkit dedicated to building graphical interfaces and interactions (see Figure 32). It consists of a core implementing the notions of Entities, Components, Systems, Descriptors and Selections. It also provides ***bindings*** to one or the other of two low-level SDL libraries [Lan98] and libpointing [Cas11]. Our goal was to be able to use the advanced libpointing mouse support (multiple mice, transfer function replacement, ***subpixel*** interaction [Rou12]), with SDL for display. However, libpointing prevented SDL from receiving keyboard events, which forced us to use ***one or the other*** bindings. At a higher level dependent on the first, Polyphony provides predefined Components, Systems, Selections and factories, which can be used by applications to quickly build interfaces.

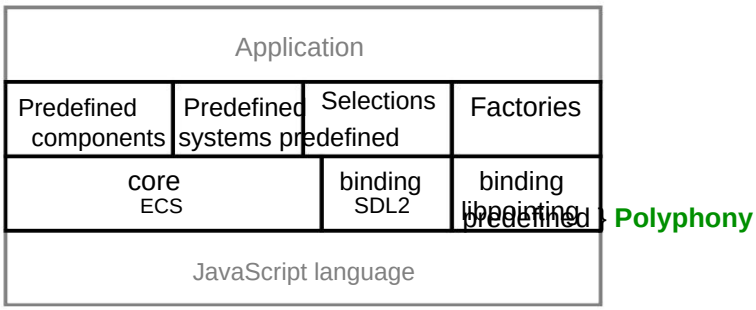

Figure 32: Polyphony module structure

The vector drawing application is a common and well-suited example to illustrate the design of interaction techniques [Gog14, Api04]. It combines the implementation of graphic objects, tools (e.g. drawing shapes, color pipette), commands (e.g. copy/paste, undo), and offers a wide range of possible tasks with numerous improvements and combinations between elements.

Our basic drawing application (see figure 33) thus allows you to:

- draw rectangles and ovals; move,
- delete and change the type of created shapes; save
- the result in SVG format; reset the
- workspace.

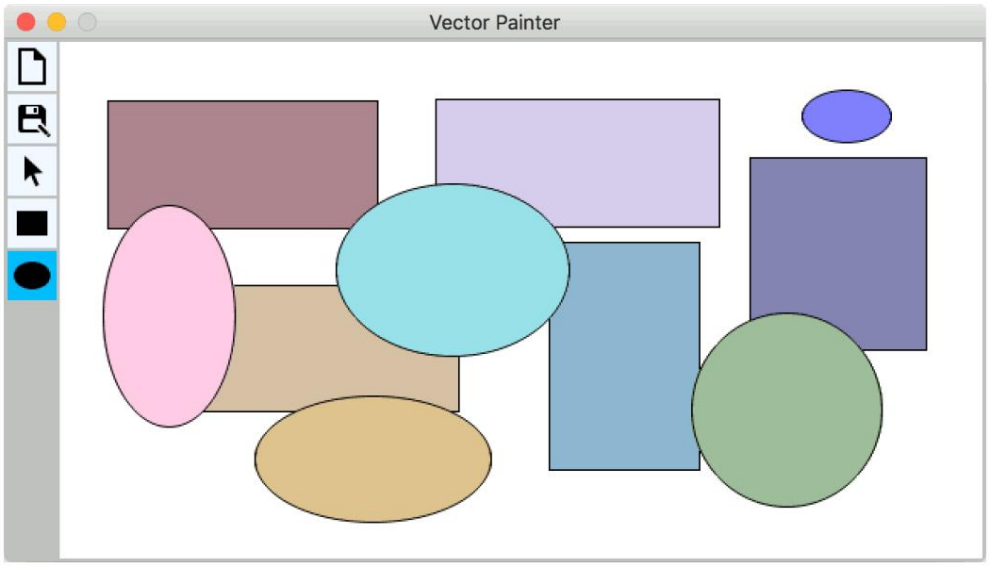

Figure 33: Workspace of our example vector drawing application

3.2.3 Illustration with application code

All interface elements are Entities: buttons, drawn shapes, and background drawing area. Input devices are also represented by Entities, as is the display area.

3.2.3.1 Creation of Entities

Entities are essentially identifiers that can be created on the fly and without Components with the **Entity** function : **let e = Entity().** In practice, it may be desirable to create Entities with initial Components. This is why the **Entity** function can receive as an optional parameter a JavaScript object to add these initial Components. For example, in our drawing application, an Entity for a simple rectangle will be created with:

```
let e =
    Entity({ bounds: new Bounds(0, 0, 100,
    50), shape: SHAPE_RECTANGLE,
})
```

At this point, the Entity is visible to all Systems that select Entities with **bounds** or **shape** (e.g. “ Layout ***System”),*** without needing to save it somewhere. In our case, the “Background Renderer” will select and display Entities that have at least the

Components **bounds, shape,** and **backgroundColor.** Thus, adding the appropriate Component to our Entity will make it appear on the screen:

```
e.backgroundColor = rgba(0, 0, 255)
```

The Entity then becomes visible to the “Background Rendering System”, which draws a blue rectangle on the screen. In practice, Polyphony provides Entity factories which allow standard widgets to be instantiated with predefined Components. Thus, in the drawing application, the toolbar buttons as well as the canvas are created from the **Button** and **Canvas** factories :

```
let y = 2
let resetButton = Button(new Icon('icons/reset.bmp'), 2, y) let
saveButton = Button(new Icon('icons/save.bmp'), 2, y += 34) let
moveButton = Button(new Icon('icons/move.bmp'), 2, y += 34, {toggleGroup: 1}) let rectButton
= Button(new Icon('icons/rect.bmp'), 2, y += 34, {toggleGroup: 1}) let ovalButton = Button(new
Icon('icons/oval.bmp'), 2, y += 34, {toggleGroup: 1}) let canvas = Canvas(36, 2, 602, 476)
```

Each factory takes as arguments a set of values necessary for the construction of the base Entity. As a last optional argument, we can pass a dictionary containing a set of Additional Components to initialize the new Entity. If this dictionary is already an Entity, it is directly completed rather than creating a new one. For example, the **Button** factory is defined by:

```
function Button(imgOrTxt, x, y, e = {}) { e.depth
    = e.depth || 0 e.bounds =
    e.bounds || new Bounds(x, y, imgOrTxt.w + 8, imgOrTxt.h + 8) e.shape = e.shape ||
    SHAPE_RECTANGLE e.backgroundColor
    = e.backgroundColor || rgba(240, 248, 255) e[imgOrTxt instanceof
    Image ? 'image' : 'richText'] = imgOrTxt e.targetable = true e.actionable
    = true return Entity(e)


}
```

When the factory receives an Entity to complete, it only adds the Components if they have not already been defined. This is what the operation **e.component = e.component ||** achieves. **value** — when a Component is not defined it evaluates to **undefined,** which is equivalent to **false** for Boolean operations.

Finally, in ECS the Entities are deleted manually. As they are globally accessible through Selections, they can never be described as “out of scope” for local scope or garbage collection mechanisms. Deleting an Entity is done with **e.delete(),** its Components then being automatically removed, as well as any references pointing to **e** in the system.

3.2.3.2 Creation of Components

In Polyphony, Components are JavaScript object constructors:

```
function Bounds(x, y, w, h) { this.x
    = x
    this.y = y
    this.w = w
    this.h = h
}
```

In accordance with the ECS model, they do not normally contain code. An exception to this rule is defining accessors, for which we then use instance methods — implemented by JavaScript's ***prototypical*** inheritance . A ***setter,*** for example, is created with:

```
Bounds.prototype =
    { setX(x)
        { this.x = x
        return this
    }
}
```

By convention, ***setters*** in Polyphony always return the targeted object **(this),** in order to be able to chain several calls in a single instruction and thus make the code more concise: **(e.setX(10).setY(20)).**

3.2.3.3 Creation of Systems

As with lambda functions, which are reified as language objects, we implemented Systems as Entities. Their dependencies are therefore represented by data stored in the form of Components. Systems are instantiated once with the same **Entity function,** which takes as parameters a function followed by its Components:

```
let ResetSystem = Entity(function ResetSystem() { if
    (resetButton.tmpAction) {
        for (let c of canvas.children)
            c.delete()
        canvas.children = []
    }
    ...
}, { runOn: POINTER_INPUT, order: 60 })
```

In our application example, this simple System will check if the resetButton **defined** above has been activated (clicked) by the mouse. When it does, it deletes all the children of the Canvas Entity. The **runOn** Component of this System indicates that it will be triggered during each event

pointing (mouse). The **order** Component distinguishes and orders the classes of Systems (represented in Figure 31) : Inputs (0 to 19), Interaction Techniques (20 to 39), Widgets (40 to 59), Application (60 to 79), and Outputs (80 to 99). **ResetSystem** is therefore part of Application Systems.

3.2.3.4 Creating Selections

When we want to iterate on groups of Entities having Components in common, we generally use Polyphony Selections. For example, to highlight all Entities that can be mouse-targeted:

```
for (let t of Targetables)
    t.border = new Border(1, rgba(0, 255, 0))
```

Here, we iterate over the **Targetables Selection,** which contains all the Entities with the Components **bounds, depth** and **targetable.** For each of these Entities, we replace the border with a green border one pixel thick. A Selection is defined with the **Selection** function , which takes an ***accept function as a parameter: Entity ÿ boolean*** to filter entities by programmable condition. A second optional argument provides an order criterion ***to compare: Entity × Entity ÿ number*** when iterating over the Selection. For example, the **Targetables** Selection is defined with:

```
let Targetables = Selection(e => 'bounds' in e && 'depth' in e && e.targetable, (a, b) =>
    b.depth - a.depth) // sort by decreasing depth
```

Selections in Polyphony are implemented with simple arrays. When an Entity undergoes a Component modification (addition, replacement, or deletion), it is added to an internal list of Entities to be “submitted for Selections”. At the end of the execution of any System (and before the execution of the next), each Entity in this list is submitted to each Selection (it is passed as an argument to the ***accept function)*** which includes it or not. Thanks to this mechanism, we do not run the risk of modifying a Selection while we iterate over it, which is a common source of bugs.
Finally, sorting a Selection is done before iterating over it, using a **dirty** boolean indicating whether the Selection has been modified and needs to be sorted again.

3.2.4 Modeling an interactive application with ECS

Figure 34 shows all the Entities in our example drawing application, each identified by the factory that instantiated it. Unlike the most common frameworks (e.g. Qt, JavaFX, HTML), graphical objects do not necessarily belong to a scene graph. From their creation, Entities are accessible to Systems, therefore already visible and interactive. Tree relationships, represented in this figure by arrows between Entities, are only defined when it is necessary to establish order between Entities — such as display depth with the **depth Component.** Interaction devices are also materialized by Entities **(Pointer, Keyboard,** and **View),** to provide persistent and flexible storage for interaction techniques. Finally, the Systems are represented in order of execution, their color differentiating the type of processing they are executing.

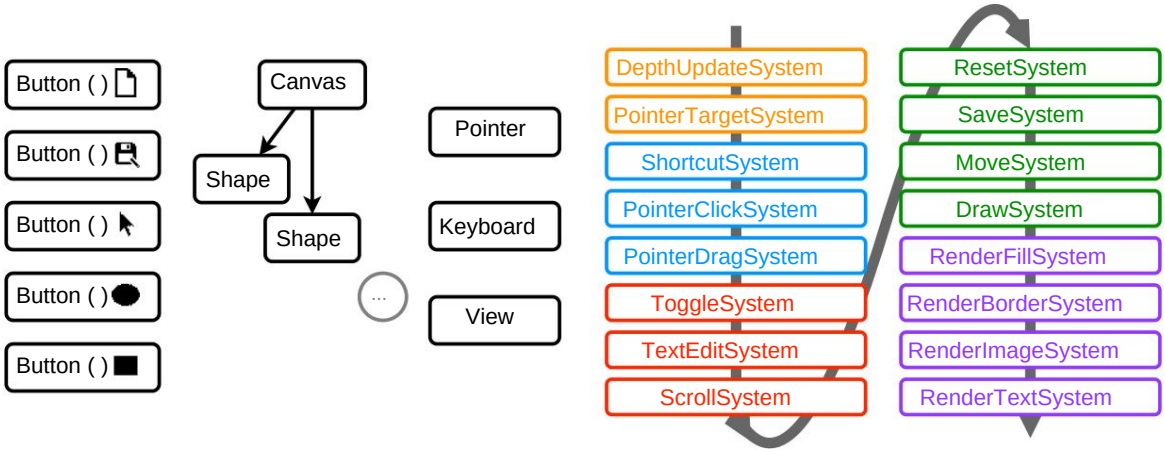

Figure 34: List of Entities (including Systems) in the drawing application. The arrows on the left represent tree relationships. The Systems on the right are presented in their order of execution, their border color indicating their type of processing: Inputs, Interaction Techniques, Widgets, Application, Outputs.

The execution flow in Polyphony works like a reactive machine [Dra01]. Every ***25ms (40Hz),*** Polyphony checks all pending operating system events and instantly updates the keyboard and mouse Entities. When at least one keyboard or mouse event occurs, an ordered list of Systems is collected, through a **Systems Selection.**
Polyphony then orders them according to the Component **order,** and executes each System whose **runOn** Component corresponds to one of the types of events produced. We chose to synchronize the triggering of all Systems (including the display) with input events (mouse movement, keyboard press, or any other input device), due to the lack of accessibility of events Low-level ***VSync .*** In the future we plan to improve Polyphony's reactive execution model, with a more generic and extensible reactive machine, independent of input events.

The Components determine the behaviors that can be acquired by each Entity. They are read by each System implementing these behaviors. Table 5 lists the Components of the drawing application, as well as the factories that assign them. These Components were inspired by CSS1 [W3C96], which synthesizes most of the behaviors common to different HTML tags, allowing for example to mimic a **button** with a div container , only by acquiring CSS properties.

Components	**Button**	**Canvas**	**Shape**	**Pointer**	**Keyboard**	**View**	***Systems***
children		×					
depth	×	×	×				
bounds	×	×	×			×	
shape	×	×	×				
backgroundColor	×	×	×				
border			×				
image	~						
richText	~		~				
targetable	×	×	×				
actionable	×						
toggleGroup	~						
draggable			×				
textEditable			×				
cursorPosition				×			
buttons				×			
keyStates					×		
focus					×		
origin						×	
scrollable						~	
runOn							×
order							×

Table 5: Components and Entity factories assigning them. The Entities of each factory receives all the Components indicated by ×, and according to their specialization then receive those indicated by ~.

Polyphony allows the implementation of standard user interface controls, with Systems and Components. Three types of controls are illustrated in our example application:

- ***Toggle buttons*** are Entities with the components **bounds, shape, backgroundColor, image/richText, targetable,** and **actionable** (like buttons ordinary). The enable/disable behavior requires another Component, **toggleGroup.** A **ToggleSystem** selects all Entities with that Component, and searches for a pointer click on one of them. When this is the case, the activated Entity receives a new Component **toggled** value **true,** and its Component **backgroundColor** is modified, at the same time as all other Entities in the same **toggleGroup** are disabled.
- Text fields require a **Focus** Component on each Keyboard Entity, and a Component **richText** to display formatted text within the boundaries of an Entity. This last Component contains a string, internal margins, and font information characters. A **TextEditSystem** observes each character typed on the keyboard and, depending on the **focus,** updates the **richText** Component of the targeted Entity, while managing a flashing cursor.

- Dropdown Views extend **View** Entities with **viewport** and scrollable Components . A **ScrollSystem** adds and manages a child Entity for each view with these Components, in order to display a scrollbar. The system detects drag and drop actions of the bar, and movements of the wheel inside the view, to update the **origin** component within the bounds allowed by **viewport.**

More generally, the implementation of new types of widgets requires the addition of new Components describing their capabilities, and the insertion of new Systems in the range of order widgets **(between** 40 and 59). These Systems form reusable behaviors, which can in turn be composed for the design of future widgets.

3.2.5 Reification of devices into Entities

Polyphony interfaces with operating system resources via an extensible set of ***Peripheral*** Entities **(Pointer, Keyboard, View).** Their Components store the current state of each device. **Pointer,** for example, has the **buttons** Component which stores the state of the buttons at any time, as well as the **cursorPosition** Component which stores the coordinates of the system cursor. These coordinates are obviously accessible, but also modifiable in order to control the position of the cursor in a simple way. In our drawing application, this mechanism is used to force the cursor to stay within the bounds of the canvas when drawing a shape using drag and drop.

Modeling devices as Entities provides flexible and persistent representation between different layers of the program. The structure of an Entity being extensible, new data can be introduced without creating a new object, but by adding new Components which will allow the Entity to be processed by the corresponding Systems. This mechanism is a form of event propagation, which does not create temporary event objects since the data is centralized in the Entity Components.

The reification of the pointer into an Entity is illustrated in Figure 35. The Entity representing the system cursor is initialized by the **Pointer** factory with two Components, **cursorPosition** and **buttons.** As long as it is the only pointer available, we call it the **Pointer Entity.** Next, Polyphony's I/O ***binding*** adds temporary Components **tmpPressed, tmpReleased, tmpMotion** or **tmpWheel** — which store the relative or absolute coordinates of the pointing event. Downstream Systems that depend on a mouse action will observe the **Pointer** Entity through the **Pointers** Selection — in case multiple pointers have been instantiated. These Systems can then add or remove Components for subsequent Systems. For example, **PointerClickSystem** uses **tmpPressed/ Released** Components to detect mouse clicks, and then adds a **tmpClick** Component with the value of **target** to the Entity when this is the case. Finally, temporary Components (whose name begins with **tmp)** are automatically deleted from the Entities at the end of each System execution chain.

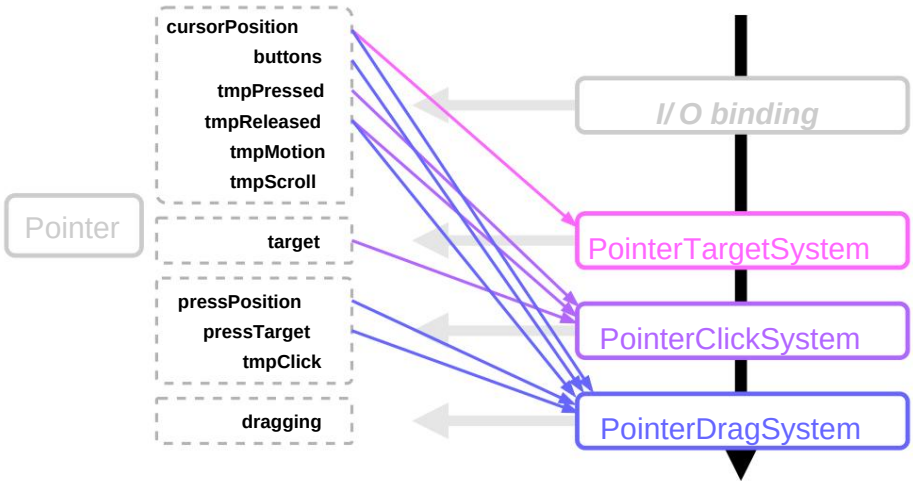

Figure 35: Evolution of the Components of the Pointer Entity, through the Systems reacting to pointing events.

3.2.6 A practical example: drag and drop implementation

Our application example is mainly based on direct manipulation techniques (drawing, moving, modifying shapes), and in particular on drag and drop. This interaction technique is notoriously difficult to implement in most current interaction libraries, and illustrates well the benefits of Polyphony for implementing interaction techniques. Drag and drop involves moving an object and dropping it onto an empty space or another object. In the second case, if the two objects are compatible, a command is executed, which depends on the two objects and the location of the repository.

In the drawing application, graphic objects can be dragged and dropped onto the toolbar buttons, in order to modify their shape or delete them: The canvas reset button/tool deletes the dropped shape; The rectangle button/tool transforms it into a rectangle (keeping its dimensions); The ellipses button/tool turns it into an ellipse. The possible combinations are summarized in Table 6 .

Displaced shape	Deposit button		
			
Rectangle	suppression	***without ef and*** ÿ Ellipse	
Ellipse	suppression ÿ Rectangle		***without ef and***

Table 6: Matrix of drag and drop combinations and commands executed in the application

The sequence of actions for performing and modeling drag and drop is:

- the cursor hovers over the object to be moved (a ***feedforward*** mechanism can indicate that the object can be
- moved); a button is pressed above the object to be
- moved; the cursor moves and the
- object follows it; the cursor hovers over a drop object (a ***feedforward*** may suggest that the drop
- will have an effect); the button is released (a command depending on the nature of the two objects is executed if they are compatible).

The difficulties associated with this technique are of several types. First, moving an object involves removing it from its current location, which can break local positioning constraints. Then, the different steps require waiting for user actions, and can therefore hardly be expressed by a continuous block of code (they are generally implemented in several ***callbacks*** at different places in the code). Finally, the commands and their ***feedforward*** do not depend on one or the other object, but on the ***combination*** of the two. The number of possibilities can therefore be very important since it depends on the product of the number of objects that can be moved by the number of recipient objects. Depending on the case, the resulting behaviors may belong to the moved objects or to the drop objects.

In our example drawing application, a **PointerDragSystem** is dedicated to detecting and managing drag and drop. It depends on the **cursorPosition, buttons** and **tmpReleased Components,** as well as **pressPosition** and **pressTarget** added by **PointerClickSystem.**
Its full code is shown below:

```
let PointerDragSystem = Entity(function PointerDragSystem() {
    for (let pointer of Pointers) { let
        position = pointer.cursorPosition let target
        = pointer.pressTarget let dragging =
        pointer.dragging // Entity being dragged if (!dragging && pointer.buttons[0]
        && target && target.draggable && position.distance(pointer.pressPosition) >
          10) { pointer.dragging = dragging = target
            dragging.draggedBy = pointer delete
            dragging.targetable // can no
            longer be pointed

        } if (dragging)
            { dragging.bounds.setX(position.x).setY(position.y) if
            (pointer.tmpReleased == BUTTON_PRIMARY)
                { delete pointer.dragging
                pointer.tmpDrop = dragging
                delete dragging.draggedBy
                dragging.targetable = true
            }
        }
    }
}, { runOn: POINTER_INPUT, order: 22 })
```

At the start of a valid drag action, this System adds the **dragging** and **draggedBy** Components to the pointer and dragged element, respectively. As long as the pointer has the **dragging attribute,** it updates the position of the dragged object with the coordinates of the cursor. When the System detects a button release, it deletes the previous Components and adds **tmpDrop** to the pointer (its Component **target** already contains the drop target). Systems inserted after **PointerDragSystem** will detect the drag and drop technique by observing the Pointer Entity Components. In our vector drawing application, the **ResetSystem, MoveSystem ,** and **DrawSystem** detect and execute the actions specified in Table 6. Additionally, moving shapes on the canvas uses the Components added by **PointerDragSystem** to handle this action.

This Systems sequencing illustrates the composition of behaviors with ECS. The shape dragging tool is based on the **PointerDragSystem,** which itself depends on **PointerClickSystem** to detect the clicked target. Dependencies between Systems are modeled by their order in the execution chain.

Each System inserts Components onto Device Entities, based on the Components inserted by previous Systems. Thus, building higher-level interaction techniques involves positioning them downstream in the execution chain.

3.3 Architecture de Polyphony

This section is dedicated to building a software architecture for Polyphony. The goal of this work is to facilitate the ***replication*** of Polyphony for research purposes. Indeed, we consider that the ECS model is intended to evolve, as more complex applications are built with it. Polyphony being a relatively young work, we still lack perspective to judge certain design choices. Polyphony being relatively preliminary and exploratory work, some of our design choices will still need to be evaluated and tested in more ecological situations, in order to better understand the advantages and disadvantages. It is therefore important for us to document how it works as clearly as possible, so that other researchers can reproduce it and develop it.

We begin by detailing the “philosophy” of programming with ECS, in light of its differences with Object Oriented Programming (OOP). Then we present ***the architecture*** of Polyphony, that is to say all of the structures that make it up and their interactions, which make the applications interactive with end users. Finally, because there are many ways to design Systems and Components in a GUI with ECS, we highlight these choices and explain our decisions for Polyphony.

3.3.1 Foundations of a software architecture based on ECS

The origins of ECS are closely related to the object model. Indeed, the field of video game engine development (excluding ***scripting)*** is notoriously dominated by C++ and object-based programming. ECS initially emerged as an alternative to OOP, with the starting point of using data (Components) as interfaces between programmers and designers/artists.
Comme décrit par Leonard [Leo99], « Programmers ***specified the available properties and relations, and the***

interface used for editing, using a set of straightforward classes and structures. Using GUI tools, the designers specified the hierarchy and composition of game objects independent of the programming staf . In Thief there was no code-based game object hierarchy of any kind ».

Then, the 7th generation consoles (Playstation 3 and Xbox 360) contributed to the development and popularization of ***data -oriented design*** for game optimization [Act14]. These consoles being very sensitive to the fragmentation of memory access, it was necessary to be able to precisely control the storage of the different data, for which OOP was perceived as limiting [Llo09]. With its central use of data, ECS has grown alongside data-driven design, subsequently gaining a sharper distinction between the abstraction of Components and their storage in practice.

Due to its origin as an alternative to object-based programming, ECS is best understood from its differences with OOP. We summarize these in Table 7, and use them as starting points to describe the Polyphony architecture below. In the descriptions that follow, “elements” refer to both objects and Entities.

	POO	ECS
Links between elements and data	the object ***stores*** its data	data is ***attached*** to Entities
Data structure of elements	structure/record (static languages), dictionary (dynamic languages)	database (Components)
Code localization	global (functions), local (object methods)	global (Systems)
Execution flow model	messages (method calls) between objects	sequence of Systems
Data access control	private/public variables (encapsulation)	Public components
Code reuse and variables	inheritance (classes or prototypes), composition (interfaces, mixins, ...)	composition (Systems and Components)
“Nature” of an element	types of classes/prototypes in its inheritance chain while	set of Descriptors evaluating it positively at any time
Visibility of an element non-global	retaining a reference to the object	by maintaining a reference to the Entity or by obtaining it by an explicit
Deleting an item	implicit (lexical scope, garbage collection), explicit (C++)	Selection

Table 7: Comparison of discriminating characteristics between OOP and ECS.

3.3.1.1 Links between elements and data, and data structure of elements

In ECS, we clearly separate the attachment of data to Entities from their storage in practice. Thus, if an Entity “owns” **bounds** and backgroundColor Components , there is no reason to think that these two data will be stored jointly in the same structure or dictionary. However, this data will be easily accessible with an accessor syntax common to objects, **e.bounds** and **e.backgroundColor** — the point then symbolizes the “Entity ***Component ”*** membership relationship . In Polyphony we have chosen to use dictionaries, to focus on the use of high level ECS rather than its

implementation. However, many implementations use sophisticated databases in an effort to improve memory access performance, particularly from a Systems perspective (see section 3.4 for a discussion of existing implementations).

It should therefore be noted that modifying ***a*** Component on ***an*** Entity is a non-trivial operation, particularly in implementations with databases. The choice of Component storage is a compromise between frequency of Entity modifications and frequency of access by Systems, the latter typically being favored because they are normally more frequent. This point explains that the description of the Polyphony architecture represents Entities and Components separately, while their connection is on the surface analogous to that of objects and their data.

3.3.1.2 Code localization

Unlike objects, Entities do not have their own code. The application code is entirely contained in the Systems, and their execution “brings to life” the interaction with users. In an interaction architecture based on objects like MVC [Kra88], the operation of an interface is modeled by exchanges of messages (method calls) between objects. For example, any object visible on the screen would have an **onDraw method,** refreshing the display would consist of going through all the visible objects in a predetermined order, and for each one calling the **onDraw method.** We therefore say that an object “knows how to display itself”.

With ECS, any element visible on the screen has attributes (Components) of background color, border color and thickness, displayable image, or text. Thus, it does not “know” how to display itself, but composes its display using its attributes. Several Systems are responsible for executing the display instructions:

- A “Background Renderer” obtains all Entities that have a background color and shape, and for each one draws a solid shape on the screen.
- An “Outline Renderer” obtains all Entities that have a border and a shape, and for each one draws a shape outline on the screen.
- An “Image Rendering System” obtains all Entities that have an image to display, and for each one draws their image on the screen.
- Finally, a “Text Rendering System” obtains all Entities that have a text attribute, a display font, and a bounding shape, and for each draws the text on the screen inside the bounding shape.

In some cases a behavior may not be composable using existing Systems.
For example, an Entity may require drawing a 3-dimensional figure. Whether this behavior is specific to ***an*** Entity, or reusable by several Entities, it will always be implemented with a new System (rather than a method attached to each Entity). With this mechanism, each “type” of behavior is normally contained within a System. Indeed, when a System already implements a given behavior, the fact that it is publicly accessible promotes its use. It is entirely possible that developers do not design a System to be reused, but we conjecture that the ECS model encourages reusability in the program (but does not mandate it).

3.3.1.3 Execution flow model

The data-oriented design approach means that with ECS we imagine an interactive application as a ***pipeline*** continually transforming interaction data into user feedback (see figure 36). Systems are the stages of this pipeline. Their order is normally fixed, or rarely changes over time. The data they process are the Components of visible and interactive Entities, as well as Device Entities. As each System generally performs an operation specific to a set of Entities, it often lends itself well to parallelization optimizations (e.g. ***multithreading*** on CPU, or rendering on the graphics card).

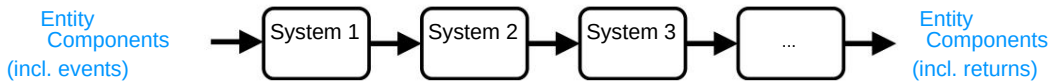

Figure 36: Schematic of the Systems execution ***pipeline*** **in Polyphony, which transforms input interaction data into output user feedback data.**

Some implementations like Unity [Uni19] make this ***pipeline*** a graph rather than a linear chain, in order to allow the parallelization of certain processing operations. However, the model of a chain remains common to the majority of ECS implementations, and we have made do with it.
in this work.

3.3.1.4 Data access control

In OOP, many languages support object data ***encapsulation*** — that is, only the owning object can access its data. Encapsulation can be an integral part of the language (e.g. private slots in Smalltalk), be supported optionally (e.g. the **protected/private** keywords in Java), or be a recommended practice (e.g. the __ prefix in Python). Its main consequence is that you must "request" an object for access to one of its data, and that it is ***responsible*** for any processing done with its data. In a model like PAC [Cou87] or MVC [Kra88], the objects containing the data of interest (respectively the Abstraction and the Model) thus occupy a central position in the architectural descriptions, the other facets having the role of transmitting the data to them.

In Polyphony, Components are the data storage media, and occupy a secondary position. Likewise, Entities are only used to find common Components. These are the Systems which occupy the central position, and interact with all the other actors of the architecture.

3.3.1.5 Code reuse and variables

In ECS, Components are simple structures, whose fields are predefined and indivisible. For example, if an Entity has a red background intensity value, then it will generally also have green and blue intensity values, forming the **backgroundColor component.** Components are thus the model for reusing ECS variables (just like C structures). They allow you to reason at a higher level than variables

atomic Entities — which can also be very numerous. In most implementations, each Component is attached to a single Entity, however some implementations allow a Component to be attached to multiple Entities. By their dynamic attachment to Entities which implements the addition of behaviors, they are analogous to interfaces and ***mixins*** in OOP, although being constrained to **only define data.**

The provision of code that can be reused is done by the Systems. Each System is a block of code, which ***materializes*** a behavior of the program. So, detecting mouse clicks is a System, as is displaying images on the screen, or serializing a group of Entities into text. Not all Systems are necessarily behaviors intended to be reused within the program. For example, in a toolbar, the triggering of commands for each button click will typically be implemented in a System specific to the toolbar. ECS therefore discourages the use of ***callbacks*** in the application, which are known to fragment the program logic, and complicate maintenance [Mye91]. This grouping of behaviors comes at the cost of a fragmentation of the point of view of the Entities. Indeed, to list the observable behaviors of an Entity, it is necessary to list its Components, and for each one identify the Systems which have an influence on it. The choice of available Components is therefore important, because they must make ***implicit*** the behaviors that will be observed for the Entities that acquire them.

They thus form a ***coherent*** set of composable attributes, which if well chosen will allow developers to anticipate the observable behaviors of Entities from their Components.

3.3.1.6 Nature of the elements

In OOP the object which was used to initialize an object instance (its class or its prototype) defines its ***type.*** This type characterizes the very nature of the object, for example a car will have the type **Car,** and the latter is itself an object, which defines the intrinsic properties of a car like **height** or **nbDoors.** The type is used to check (statically or at runtime) that an object has expected variables and methods. For example, a function defined in Java as **int open_door(Car)** expects to operate on a car, so a call with an integer like **open_door(42)** will trigger a type error during compilation.

With ECS the Entities do not have any intrinsic property, that is to say which is always present. The nature of an Entity is a time-varying notion, which changes as Components are added to and removed from the Entity. The factory that instantiated an Entity cannot predict its nature later, and therefore cannot be used as a type. The association of one or more types with an Entity is therefore necessarily dynamic, that is to say that they must be evaluated each time at execution, and that their validity lasts as long as no modification of Components occurs. 'intervenes. It is through Descriptors that we test the ***validity*** of an Entity for a given Boolean condition.

3.3.1.7 Visibility of a non-global element, and deletion of an element

As in OOP, we access an Entity from the moment we have a reference to it, and the fact of having a reference allows free access to the Entity, without restriction on the origin of the access . With ECS, an additional mechanism (Selection) makes it possible to obtain the

references of Entities responding to a given Descriptor. This mechanism is analogous to CSS selectors [W3C96]. In Polyphony it is used at the heart of the interaction architecture, to allow Systems to enumerate the Entities on which to operate.

In practice, this principle favors a less strong structuring of the application. Most object-oriented frameworks are based on the construction of a ***scene tree*** [Bed00, Lec03, Huo04]. This data structure contains all the interactive elements of the application. It is used both to enumerate interactive elements (by recursive traversal), and to represent parentage relationships, useful for positioning constraints and property propagation [Bed00]. In Polyphony, the scene tree is used only for parentage relationships, and to propagate style information (opacity, text alignment, etc.). Not all Entities are necessarily part of it (including visible and interactive ones), hence the absence of a scene tree in the description of the architecture.

The counterpart of this principle is that unlike OOP, we can no longer infer that an object is out of scope if no reference points to it. With ECS it is normal that an Entity is not referenced by any other, but that it is nevertheless always visible and interactive. Entities must therefore be deleted explicitly. Invalidating references (by assigning them to **null,** or by removing Components from other Entities that reference them) is the responsibility of the programmer in Polyphony, however it is possible in the future to automate this operation.

3.3.2 Description of the architecture

In this part, we present the constituent elements of Polyphony, as well as the relationships they have with each other. This description of the architecture is intended to give an overview, as well as to highlight the sequence of execution stages of the program.

The basic principle of Polyphony is summarized in Figure 37. When a System is activated during the Systems execution chain, it performs several types of operations:

- It retrieves the lists of Entities and devices on which to operate, via Selections (not shown in the figure).
- It reads and combines Entity Components with devices. For example, combining the terminals of an Entity with the position of a mouse creates targeting information from the mouse to the Entity.
- Finally, it optionally modifies the Components of the Entities and devices listed, in order to propagate the data for the following Systems.

In the descriptions that follow, we formulate the ***services*** necessary for Polyphony to work, and illustrate them with their syntax in JavaScript code. The descriptions in this section should allow anyone to reimplement the first low-level half of Polyphony, in JavaScript or any other language.

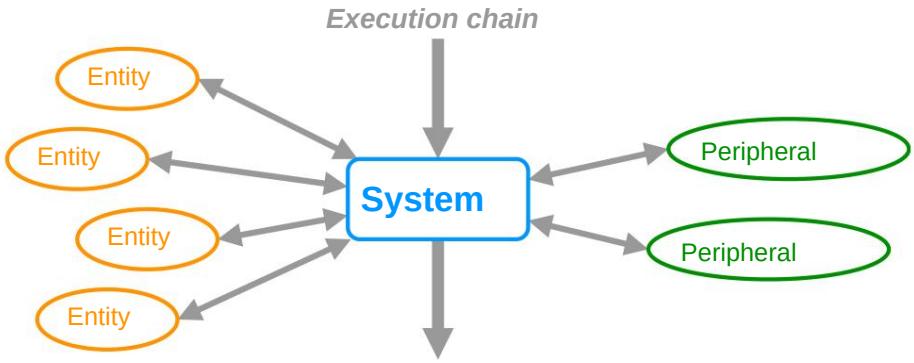

Figure 37: Illustration of the operation of a System in the Polyphony execution chain.

3.3.2.1 Entities and Systems

Entities are the heart of Polyphony. They represent all active and interactive elements of the application, from the users' point of view. In Polyphony, each Entity behaves like a dictionary associating names with Component references. An Entity can be created from a function, in which case it is executable and represents a System. This function can exist in the form of a special Component, or using a language-specific object type (in our case the native **function type).** The services relating to Entities and Systems are:

Creating an Entity — let e = Entity(obj, fct_params)

An Entity is instantiated, and its reference is stored in the list of active Entities. An object can optionally be passed as the first parameter, in which case it is used to provide initial Components to the Entity. This object can also be a function, in which case a second parameter provides the initial Components for this Entity. The reference of the Entity created is also added to an internal **set** of modified Entities, which will be used to update the Selections before the execution of Each System. Selections will be notified of these modifications upon completion of this System.

Association/Replacement of a Component to an Entity — e.comp = reference

A link is created between a given Entity and Component. Each Entity can be associated with several Components, but a Component is only associated with a single Entity. Thus, when one of the variables of a Component is modified, the Entity to which it is attached is automatically added to the list of modified Entities. An exception to this rule concerns ***immutable*** Components (whose internal variables cannot be modified), which can be shared between several Entities. This is the case for example for colors, whose underlying structures are thus reused. Each Component is in fact an association between a name and a location (untyped), any modification of which (addition, replacement, internal modification, deletion) results in the Entity being added to the list of modified Entities.

When the name of the Component is prefixed with **tmp,** the link is ***temporary,*** that is to say it will be destroyed at the end of the current execution chain of the Systems. This type of Component allows information to be passed from upstream Systems to downstream Systems, by storing this information on Entities (such as **Pointer** and **Keyboard).** Finally the reference of the associated Entity is added to the internal set of modified Entities.

Reading a Component of an Entity — let reference = e.comp The link between an Entity and a Component is found, and the reference to the second is returned. If there is no such link defined, an invalid value is returned **(undefined** in JavaScript).

Checking the association of a Component with an Entity — if ('comp' in e)

The expression returns a Boolean indicating whether the Entity is attached to a Component with that name. This operation is not strictly necessary in a Polyphony implementation, as it can be replaced by **if (e.comp != undefined) .**

Iterating over Entity Components — for (let [c, r] of Object.entries(a))

A block of code is executed as many times as there are Components attached to an Entity, two variables **(c** and r) informing at each iteration the name of the Component iterated as well as its value. This operation is particularly useful for serializing an Entity into text, when it is necessary to debug the program or save its current state in a file.

Deleting a Component from an Entity — delete e.comp

The link between an Entity and a Component is destroyed, and the data structure of the second is erased from the program memory (if no other Entity refers to it, in the case of immutable Components). If no Component of this name was defined for the Entity, the operation has no effect and does not return an error. The Entity reference is added to the internal set of modified Entities.

Deleting an Entity — e.delete()

All Components linked to an Entity are deleted, and this Entity is removed from the list of active Entities. It is also added to an internal set of Deleted Entities. Selections will be notified of this change when the next System is run. If references to the Entity to be deleted still exist in the application, they are kept (the Entity pointed to is then an empty shell). It would also be possible to automatically invalidate these references, as described by Kedia [Ked17].

Execution of a System — e(...)

All Selections are first updated with the sets of Modified and Deleted Entities. These sets are then reset to their empty state. The Entity that is attempted to be executed is normally associated with a function (stored as a hidden Component), which is executed by relaying any parameters of the function call. The absence of such a function triggers an error during program execution.

Execution of the Meta-System — MetaSystem(eventType)

The Meta-System is a particular System, whose function is to order (according to the Component **order)** and execute the other Systems. It uses a Selection allowing it to iterate over all Systems (except itself). This Selection being updated each time the System is executed, a copy is made beforehand so as not to risk modifying it while iterating over it. The Meta-System receives as an argument the type of trigger event allowing it to select the Systems to include in the execution chain (based on their **runOn Components).** At the end of its execution, the Temporary Components are automatically deleted.

3.3.2.2 Components and environment

Components describe the attributes that Entities can acquire (like **shape** or **backgroundColor),** as well as their abilities (like **actionable** or **draggable).** In Polyphony, the Component designates a structure attached to an Entity, rather than the data type from which it is instantiated. The environment, through ***bindings*** with the operating system, manipulates the Components of the Entities materializing the interaction devices (mouse, keyboard, and screen), at the beginning and at the end of each execution chain of the Systems. The services relating to Components and the environment are:

Creating a Component type — function Comp(i) { this.i = i }
: A global and permanent type is created, which is similar to a ***structure*** declaration of the host language. This structure defines a fixed number of variables, in a determined order. Each variable has a name, a type if the host language allows it, and is accessible by everyone without restriction. The definition of a new type of Component implies the existence of a syntax allowing new Components to be instantiated.

Instantiation of a Component for an Entity — e.comp = new Comp(...)
: A Component is created, and its memory is initialized with the values provided as arguments (plus any variables calculated from the arguments). A link is immediately created between this Component and the Entity.

Modification of a value of a Component — e.comp.i = val
: The Component sees one of its variables in memory receive a new value. By the link which binds it to an Entity, this one is automatically added to the internal set of modified Entities, so that the Selections can react to this change.

Serializing a Component into code — serialize(reference)
: This involves transforming an existing Component into a character string, such that the interpretation of this string as source code generates the same initial Component. This operation is not strictly necessary for the implementation of Polyphony, however it allows us to load and save the Entities of a scene in JavaScript code, without resorting to an external format like XML.

Updating Peripheral Entity Components
: This operation depends on the software functions used to retrieve operating system events. When an external event occurs (action on the mouse, keyboard, program interruption), the type of the event is first tested to update the correct Components of each Peripheral Entity (e.g. **pointer.cursorPosition** for a movement mouse). Then, the Meta-System is launched, passing as a parameter the type of event that triggered the update.

Initialization of environment elements
: This operation corresponds to starting the application. Bindings with the operating ***system*** are first initialized if necessary. Then a graphical window is created, as well as the predefined entities **pointer, keyboard,** and **view.** It is also possible that these Entities are created by ***hot-plug,*** that is to say after their connection while the application has already started.
This is particularly the case with the use of libpointing [Cas11]. Finally, the Meta-System is launched for the first time to generate a first display on the screen of the Entities already present.

3.3.2.3 Descriptors and Selections

Selections are the primary interface between Systems and Entities. They allow Systems to retrieve the identifiers of the Entities on which to operate, before using these identifiers to retrieve the Components. A Selection is a set of Entities based on a Boolean condition (Descriptor), whose content is regularly updated to include ***all*** Entities validating the condition, and ***only*** these Entities. The services relating to Descriptors and Selections are:

Creating a Descriptor — let desc = (e) => 'comp' in e && e.comp < extVal

A reference to a function is returned, which takes an Entity as argument and returns a boolean. It is therefore a programmable condition, which can be based on the attached Components, their values, or a combination including external values.

Querying a Descriptor — if (desc(e)) {...}

Execution of the function referenced by the Descriptor returns a Boolean, which allows you to check at a given time whether or not an Entity belongs to a certain category. This operation is the basis for constructing and updating Selections.

Creating a Selector — let sel = new Selection(desc, comp)

A global and permanent Selection is created, from a Descriptor provided as an argument.
A binary order relation can be provided as a second argument (Entity **× *Entity ÿ number function),*** in which case the Selection is guaranteed to always be sorted according to this relation. The Descriptor cannot be modified after creation of the Selection, however it is publicly accessible for reading.

Updating Entities for a Selection — sel.update(modified, deleted)

A Selection is notified of lists of Entities that have been modified and deleted since the last execution of a System. It inserts the Entities validating its Descriptor for the first time, keeps the Entities having already validated it, removes the Entities no longer validating it, and ignores the Entities still not validating it. Entities in the **deleted** list are systematically removed, regardless of the result of the Descriptor.

Iterating over the Entities of a Selection — for (let e of sel) {...}

A block of code is executed as many times as there are Entities in a Selection, a variable **(e)** informing at each iteration the reference of this Entity. If an order relationship exists for the Selection, it is sorted before the loop is executed. This operation allows a System to execute a given behavior on each Entity, independently of the others.

Random access to any Entity in a Selection — let e = sel[n]

The Entity in nth position in the Selection is returned. More generally, any Selection can be used as an array, with the exception of array modification operations. This access allows a System to execute transverse behavior across several Entities, for which iteration in sequence would be tedious. The global resolution of linear positioning constraints is an example, since it involves considering all of the constraints as a Linear Optimization problem, and solving it globally [Bor97].

Many ECS implementations offer combination algebras for Selections, for example “iterate over entities that are in S1 and S2, but not in S3”. These algebras generally correspond to the Boolean operators ***AND, OR,*** and ***NOT.*** However, in practice, they are already implemented in the JavaScript language, with the operators **&&, ||,** and !. We therefore designed the descriptors as functions (rather than lists of Components), in order to reuse the integrated composition algebra, and to be consistent with the language. Additionally, we wanted to have the most flexibility in expressing descriptors, because we could not anticipate which algebraic operations would be most used in practice. Indeed, the majority of ECS implementations express the basic descriptors (excluding algebraic combinations) as lists of Components, which must all be possessed by an Entity to include it. These descriptors actually implement the basic ***AND operation,*** and therefore imply that this is the operation that will be used the most. As we wanted to observe before taking such an orientation, we chose the most flexible option, with descriptors specified in code.

3.3.3 Design choice of Systems, Components and peripherals

This part illustrates the second level of Polyphony (shown in Figure 32), built on top of the ECS implementation presented in the previous section. This level is made up of reusable elements provided by Polyphony, to help programmers quickly build complete interfaces. The choice of the proposed Systems, the available Components as well as the use of the Selections, result from compromises which it is important to explain in order to support future iterations of HMI toolboxes based on ECS.

3.3.3.1 Choice of Systems

The design of Systems in Polyphony results from a compromise between simple and complex behaviors. Take for example drawing widget borders (rectangular or polygonal) in the application. We have four alternatives, ranked from simplest to most complex:

- Create a System drawing segments, with which a border is formed from several Entities positioned in a polygon.
- Create a System drawing rectangles, which will be suitable for the majority of needs.
- Create a System drawing arbitrary geometric shapes, for which drawing a border amounts to providing a table of points.
- Create a System drawing basic widgets (background, border, image, and text), for which drawing a border will be a subset of its capabilities.

With simple types of behaviors, it's easy to put them together in creative ways. With a System drawing segments, we could change the color of one of the edges, independently of the others, which is not possible with more complex alternative Systems (if they are based on a single color). On the other hand, with complex types of behavior, fewer Systems are defined overall, so the System execution chain is shorter, and it is easier to have an overview. In general, we want to facilitate the programming of complex applications, and make the structure of Systems as simple as possible, so we favor complex Systems and few in number.

In the example above, the first System drawing one segment per Entity is too simple. To draw a complete rectangular border (without “holes”), it requires creating four Entities, and positioning them relative to the Entity of the widget that you wish to draw. Additionally, this System does not provide “high value” behavior, as it can be recreated quickly if a developer needed to draw lines.

The second System is more advanced, but stereotypical. Indeed, each widget that requires a border will only have to acquire a few Components (without requiring new Entities), so it will be quick and easy to add rectangular borders to widgets. However, researchers experimenting with the design of non-rectangular controls will not be able to use this System, and will have to create another one.

The third System is more advanced and less stereotypical, but complex for simple uses. Borders for non-rectangular controls would be simple to create, and their coordinates would be included in a Component, so would not require multiplying Entities. However, they make it more complex to draw borders for rectangular controls, which would then have to specify four points.

Finally, the fourth System is too complex. Indeed, although it is particularly suited to designing standard controls, this System is “monolithic” and makes no use of the composition capabilities of ECS. The creation of new types of controls would be done by multiplying the Entities, each using a subset of the System's capabilities. In addition, this type of System is the most difficult to replace, and therefore contributes to the crystallization of the types of controls available, and more generally of the user experience.

The example we have chosen is deliberately simple, so as to clearly illustrate the choices available to programmers to design Systems according to their needs. In practice it is possible to create a System which covers several uses simultaneously (e.g. which allows a complete border to be specified, or each side separately). When developing a System, we might be tempted to multiply these options, and include hypothetical future needs, to facilitate the prototyping of new interactions. However, the extrapolation of needs creates more complex Systems, because there are numerous development avenues, some of which will never be developed. In the example of borders above, such tracks would be: polygonal borders, parametric curved borders, borders of non-uniform thickness/colors, or even borders animated over time. From this remark, our first recommendation is to **support only contemporary uses.** The ECS model is designed to facilitate the creation of new Components and Systems, which is preferable to complex Systems supporting unusual needs. Finally, in the same way that we quantify the number of elements that working memory can retain ***(memory span)*** around 7 [Mil56], we prefer to reduce the number of Systems to allow developers to have an “overview” of them (or even to represent them graphically). It is then necessary to identify independent behaviors (e.g. drawing borders, backgrounds, text, or images), and distribute them into a reasonable number.

The final choice is a compromise that depends on the context, and does not have a single solution. We therefore recommend **dividing Systems into a small number of independent behaviors.**

3.3.3.2 Choice of Components

The primary function of Components is to aggregate data. This data can always be divided into atomic values — for example, an x coordinate , a y coordinate , a **width width,** a **height height.** On the other hand, this data can also be grouped into sets that share a common meaning — for example, x, y, **width** and **height** form a rectangle, or bounds. At the extreme, data can be grouped into large sets with broad meaning — like the ***basic widget*** classes defined in most frameworks. There is therefore a balance to be found between light (or even atomic) components and heavy components.

Programmers interact directly with Components, to manage the behaviors added to the various Entities. The challenge of this design choice is therefore the ease that programmers will have in abstracting themselves from the low level, and in handling a few Components at a time. If the Components are too heavy (like complete widgets), then the ECS composition mechanism cannot be exploited to its maximum, since we would add behaviors by activating options in a Component, rather than by acquiring new Components. If Components are too lightweight, then different behaviors require a lot of Components, and programmers have to write more code to add them to Entities.

The performance of the application also influences this design choice. In favor of heavy Components, access to data from a Component (reading and writing) is faster than access to a Component from an Entity, because of the flexibility of storing the Components of an Entity . In favor of lightweight Components, the grouping in memory of all the data used by the same System favors the use of processor caches, so the unused data of a Component generally reduces the performance of the application.

For the choice of Polyphony Components, we therefore applied these recommendations:

- **data required by *a* System should belong to *a* single Component data shared**
- **by *several* Systems should be distributed so that each System dependent on a Component uses *all* the variables**

As an example of the application of these recommendations, we observed that all displayable elements of the interface necessarily had the variables x, y, **width** and **height** — including non-rectangular ones which then have an enclosing rectangle. Only the system pointer only has **x** and y coordinates , however its behaviors are not intended to be composed with those of the interface widgets. As a result, Polyphony offers a **Bounds** Component including all four variables.

3.3.3.3 Choices related to Selections

In object (and Entity) programming, objects “know” each other by storing references (generally addresses in memory):

- one object “knows” another by storing a reference to it one object
- “knows” several others by storing an array of references to them

With the use of Selections, new ways of referring to objects appear:

- we can refer to an Entity that we know to be the first element of a Selection (eg: the first pointer, **Pointers[0])** we can refer
- to several Entities by assigning them a Component differentiating them from the others Entities, and recovering them by a Selection

In Polyphony, all these ways of referring to Entities coexist, and often offer several ways of doing things. In the example of our drawing application, to refer to the mutually exclusive activatable buttons (the movement tools, rectangles, and ellipses), should we keep a table of references, or assign the buttons the same **toggleGroup** Component ? Similarly, to define parentage relationships in a scene tree, does each node store an array of children, or do the children carry a Parent **Component ?**

To answer these questions, we must remember that Selections are costly structures in performance, because Polyphony keeps them up to date by presenting ***each*** Entity that is modified. We must therefore avoid misusing them. In the first case, we will use a **ToggleGroups** selection which refers to all the Entities having a **toggleGroup Component,** and we will filter during their enumeration the Entities with the same **toggleGroup** (because we know that there are few of them). This choice also allows you to easily add a new button to a group, by assigning it the same **toggleGroup Component.** In the second case, it is assumed that large scene trees are likely to be used. With a Selection of all Entities having a **parent Component,** one would have many Entities to ignore to enumerate those with a given **parent .** In addition, the order of a node's children is important (which would not be guaranteed in a Selection without sorting criteria).

We will therefore instead use an array of children on each parent node of the tree. So, to differentiate the two cases, we recommend **using a Selection when the number of Entities to discard is small,** because each Entity ignored in a Selection is much more expensive than an Entity ignored in a table.

Finally, when detecting interaction techniques on devices, the question arises of the Entities to which to attribute the generated Components. Take for example the drag-and-drop technique: when dropping one object onto another, a temporary drop ***event*** is generated, which must be communicated to downstream Systems in the execution chain. The Components to be created can be stored:

- on the receiving object, which indicates which object it received **(tmpDropOf),** and possibly which pointer dropped it
- **(tmpDropBy)** on the dropped object, which indicates which object it is released on **(tmpDropOn),** and possibly which pointer dropped it
- **(tmpDropBy)** on the pointer at the origin of the drop, which indicates which object was moved **(tmpDropOf),** and on which object he dropped it **(tmpDropOn)**
- on a combination of the three

The interaction between devices and Entities leads to the possibility of storing information on one side or the other (or on both sides). In the case of drag and drop, it is assumed that ***many*** Entities will be movable, ***many*** will be able to receive drops, but that there will be ***few*** pointers (usually only one). It will therefore be wise to add the Components to the pointer, because

Downstream Systems will only need to enumerate the Selection **Pointers** to detect the deposit action (and will discard few Entities). Thus, we recommend **storing temporary Components on the least numerous Entities** (i.e. accessible from the smallest Selections).

3.4 Implementation of the Entity-Component-System model

In the previous section, we described the essential principles of ECS, as well as the high-level architecture of Polyphony. We have not ***described*** our low-level design choices (Are Entities objects? Where are Components stored? etc.), in order to facilitate the application of Polyphony to any type of programming language (oriented -object, imperative, static or dynamic). This section is dedicated to the implementation of Polyphony. We begin by studying three major implementations of ECS, in order to identify a design space in which to position ourselves. Then we study the differences between the fields of Video Games and HMI, which justify our deviations from existing implementations. Finally, we present in detail the implementation choices of Polyphony.

3.4.1 Analysis of existing implementations

There are many ECS-based frameworks, each with variations of the same concepts. This multiplicity makes it difficult to choose one implementation over another for anyone needing to use this programming model. To better inform, document and inform the use of ECS as a programming model, and in particular in the case of HMI and interaction techniques, we have built a ***design space*** for ECS implementations. It is based on the study of 3 of the most referenced frameworks [Mar14] and better documented, as well as our experience in the development of a new variant dedicated to interaction programming, ***Polyphony.*** All the frameworks we found target video game development, and we didn't find any dedicated to GUIs.

We selected :

- **Artemis** (Java), one of the first historical implementations of ECS. As the original version is no longer documented and no longer maintained, we have selected its most popular successor, Artemis-odb [Pap18].
- **Entitas** (C#), a popular interpretation of ECS for Unity, which includes a meta-language and preprocessor for C# [Sch18]. Its user base as well as its comprehensive documentation make it a major ECS implementation.
- **GameplayKit** (Swift), one of the many ECS-inspired frameworks that implement behaviors directly on Components, rather than in separate Systems — with for example CORGI (C++) and Nez (C#) [Goo15, Pri18]. They are sometimes referred to ***as Entity-Component systems ,*** which fuels confusion with the original ECS model. Unity (C#) also implemented this approach initially, and later converted to the original model, but is not yet sufficiently documented [Uni17].

The analysis presented in Table 8 is not exhaustive, but highlights the design choices on which ECS interpretations primarily differ. We have also included our own choices for implementing Polyphony, and discuss them at the end of the section. THE

criteria of our analysis are:

- **Entity Representation** — how Entities are materialized in the framework
- **Component Representation** — how Components are materialized, and defined by programmers

- **Systems Representation** — how Systems are materialized, and defined by programmers
- **Context Structuring** — how the environment is materialized in the framework, to instantiate new Systems/Components/Entities, and to obtain Selections
- **Entity Selection** — how Entities are grouped from their Components
- **State changes** — what mechanisms are available to react to state changes
- **Component Storage** — how and where Components are stored
- **Systems Scheduling** — what mechanism allows you to order Systems in the execution chain

- **Entity Factories** — how do you define Entity models for reuse
- **Language extensions** — does the framework implement ***meta-language extensions,*** i.e. new syntax not permitted in the host language

Framework (Language)	Artemis-odb (Java)	Entity (C#)	GameplayKit (Swift)	Polyphony (JavaScript)
Entity Representation	integer, or integer in an object	object	object	object
Representation of Components	subclasses of **Component** with default values	subclasses of **IComponent**	subclasses of **GKComponent** with default values	any object
Systems Representation	subclasses of **BaseSystem,** attached to a Selection, periodic execution	subclasses of **ISystem,** attached to a Selection, periodic or unit execution	Component method **updateWithDeltaTime:**	Entity initialized with a function
Context Structuring	objet **World**	objet **Context**	scene tree	***global***
Entity Selection	by list of Components, with ***all/ one/ none*** algebra	by list of Components, with ***all/ one/ none*** algebra	by list of Components, or condition programmable	par condition programmable
State changes	***listeners*** on Selections	***listeners*** on Selections	***not explicitly supported***	Temporary components
Storage of Components	by Component, with EntityÿValue table	by Entity, with ***slots*** dynamically	by Entity, with table ComposantÿValeur	as native object properties
Systems Scheduling	dynamically by ***factory object***	predetermined by insertion into a list	statically	dynamically by priority
Entity Factories	***priority,*** post-processing file loading of	file loading ***not explicitly supported***		***factory*** function
Language extensions	the compiled executable	pre-compilation of a language overlay	***none***	Entity syntax by ***Proxy*** objects

Table 8: Analysis of three variants of ECS and Polyphony according to our design space.

3.4.2 Polyphony: design choice

The main design dimensions for implementing ECS having been presented, we must now apply them to HMI programming. It is first necessary to study the use of a composition paradigm in video games, and to justify its application to the construction of graphical interfaces and interactions. We then highlight the differences between the development of video games and graphical interfaces, then discuss our choices with regard to these differences.

3.4.2.1 Adaptation of ECS to the HMI context

All the toolboxes we have studied are dedicated exclusively to video game development. However, this type of application has very specific needs, sometimes similar, but often different from interaction programming, which we detail here.

Multiple System Triggers

Game state is typically updated by a single System chain, with a fixed ***tickrate*** (often 60Hz, a common screen refresh rate). As GUIs are typically updated in response to multiple and varied event sources, we introduced multiple triggers to the System chain, and filtering of Systems to execute based on the triggering event.

Entity Scene Tree

The elements of a game have loosely structured relationships between them. They move, appear, disappear, and require few relationships between them — apart from the use of partitioning structures to optimize the display [Bis98]. On the other hand, elements of a GUI have relationships of display order, relative positioning, and inherited styles, which are best expressed by scene trees.

Entities materializing interaction peripherals

Many games obey the principle of ***one player per machine,*** and support a single keyboard/ mouse pair (but sometimes several controllers). As part of the research and prototyping of interaction techniques, we want to support a greater variety of devices, in multiple copies, and support their addition/removal at runtime. We introduced ***Peripheral*** Entities to allow advanced input management.

Definition of Temporary Components

In a game, most observable state changes are linked either to the player's actions or to the environment. In the first case, a global event mechanism is generally used to signal changes of state. In the second case, most of the implementations that we have studied integrate ***listeners*** (therefore callbacks ***in addition*** to Systems) on the Selections, in order to observe changes in groups of Entities. In graphical interfaces, state changes are common, and many types of actions are associated with them — interaction techniques, commands, positioning. To materialize and manipulate many types of state changes ***without introducing listeners,*** we introduced the use of Temporary Components, primarily for Peripheral Entities.

Representation of Systems in

Entities The execution ***pipeline*** of the different Systems is generally fixed, which is why ECS implementations order them in simple lists. To facilitate prototyping and manipulation of Systems at runtime, we have adopted a recursive approach and represent Systems by Entities. The triggering of Systems is thus made more flexible, thanks to the management of triggers by Components, and the establishment of a Meta-System which orders and executes the Systems.

Abandoning Context Objects

Games are often organized into “levels”, independent of each other, and trigger loading sequences between them. They store the Entities, Components, and Systems relating to each level in “Context objects”, which allow the levels to be clearly separated. In contrast, navigation in GUIs involves moving back and forth between views, tabs, or modes, with many elements in common, and avoiding loading sequences that could interrupt the interaction. To this end, we have discarded the use of Context objects, preferring the flexibility of Components to share elements between views.

3.4.2.2 Implementation of Entities and Components

In Polyphony we represent Entities by native objects, encapsulated in JavaScript ***Proxy*** objects. Proxy objects allow us to intercept all the native operations they receive, which allows us to offer Entities accessible with the native JavaScript syntax [Ecm15], but behave differently. These operations are:

- **ec** returns the value of Component **c** for Entity **e** (or **undefined) ;**
- **ec = v** associates Component **c** of value **v** with Entity
- **e ; 'c' in e** returns **true** if and only if **e** is associated with
- **c ; delete ec** dissociates **c** from Entity e.

The reasons for reusing an object syntax are mainly because HMI developers are very familiar with the object model, and therefore with the **obj.property** syntax for accessing a field of an object. We are therefore reusing a known and well-mastered concept. The other reason for this choice is that the representation of Entities by integers (in Artemis) presumes that we will store them in tables, which breaks the abstraction between the Entities and their effective storage. Finally, as we could not anticipate the uses that would be made of the Entities, we chose to allow maximum flexibility in the evolution of our model, by making the Entities simple dictionaries. Likewise, we did not impose any constraints on the Components, to experiment with different design choices: use of ***setter*** methods on the Components, distinction of the types and names of the Components (e.g. **backgroundColor** for a **Color type,** while implementations often do coincide the two), reuse of immutable components (e.g.
Color), or even dynamic storage in Components so that Systems add private variables.

As for the actual storage of Components, the implementations that we studied implement sophisticated databases, in order to specifically optimize certain types of processing. Thus, Artemis optimizes the processing of the Systems, because from each

Component,Systems can obtain all the values that interest them. In contrast, Entitas stores the Components together within the same Entity, and thus optimizes the processing carried out consecutively on the same Entity. In our case, we considered that optimizing object storage is already a major problem for JavaScript, so we chose to reuse objects
natives.

As for Entity factories, in the first versions of Polyphony we initialized each Entity with hand-written code. When we were able to draw shapes in our sample application, we implemented saving and loading Canvas Entities. Rather than importing an external format and complicating learning, we chose to ***serialize*** the Entities in JavaScript code, and “load” such a file by simply executing it.

3.4.2.3 Implementation of Systems

In Polyphony, Systems are materialized by JavaScript functions, encapsulated as Entities in ***Proxy objects.*** JavaScript allows you to assign properties to functions, while giving them a distinct native type **(function** rather than **object).** We can therefore distinguish an executable Entity from an Entity which is not, while supporting the management of Components in a common way.

We initially chose to implement Systems with simple functions (rather than object methods) to get closer to a concept of “functions with attributes” (which would store variables). Our idea was then ***to augment*** the declaration of a function with attribute initializations, which would inform in particular the external triggers of the function.
We could thus do without explicitly registering these functions as callbacks , since the system would automatically take care of redirecting external events to functions with the corresponding attributes.

Unlike Entitas, Polyphony only supports Systems running periodically, and unlike Artemis and Entitas, Systems are not explicitly attached to a Selection.
In these frameworks, it is common for each System to operate on a Selection of Entities. They therefore promote these uses by offering Systems which already implement iteration on a Selection, and only require one function which will be executed for each iterated Entity. In our case, the combination of widgets (Entities) and devices (Entities) causes many Systems to iterate over several Selection. In addition, we wanted not to limit the execution of a System to a function executed by Entity, but to allow group processing if necessary. We have therefore not explicitly attached the Systems to the Selections they process. This point may change in the future, as it is currently difficult for an Entity to list the Systems that are active for it, which may make it difficult for developers to use them.

For the scheduling of Systems, we wanted to be able to visually ***represent*** the Chain of Systems, or even ultimately offer an editing interface. This point encouraged us to limit the number of Systems, and to opt for a linear chain rather than a dependency graph which would have been more complex to represent. Given the small number of Systems, it was possible to order them by hand, rather than expressing explicit dependencies between Systems, and the

order with a traversal in a dependency graph. The Systems therefore have an **order** Component to manually order their list. However, supporting Components on Systems does not exclude the addition of dependencies between Systems later.

3.4.2.4 Implementing Context and Selections

Artemis-odb and Entitas each define a class to store the active Entities and materialize the Context. They create new Entities, link Components to them, and register new Systems by calling methods on the Context object. They also make it possible to instantiate several of these Contexts to represent several ***worlds,*** for example to load the next level or manage a second player on the same machine. Polyphony on the other hand does not provide explicit Context, and makes everything global, without the need to maintain a contextual reference. Instead, we can manage multiple ***worlds*** by adding a **world** Component to each Entity, in order to manage the parent Contexts of each Entity.

As explained in section 3.3.2.3, we chose to implement each Descriptor by a function (rather than a list of Components to have). Furthermore, we do not provide composition algebra for Selections, in order to consistently reuse the Boolean algebra of JavaScript, and not to presuppose the use that developers will have. Until now, our needs have mainly come down to lists of Components, with a few rare exceptions (conditions on the internal values of Components) being able to be reduced to Components. It is therefore possible in the future to integrate Polyphony with an existing ECS implementation, to benefit from more optimized execution.

3.4.2.5 Integration with JavaScript language

As discussed in Interaction Essentials, we wanted to emphasize the close relationship between a framework and the programming language we use. In Polyphony, this relationship is manifested by the reuse of object access syntax, to access Entities. In particular, the initialization of Entities reuses and extends that of JavaScript objects:

```
let e = Entity({ depth:
      0, bounds:
      new Bounds(0, 0, 100, 50), shape:
      SHAPE_RECTANGLE,
      backgroundColor: rgba(0, 0, 255),
})
```

The **Entity** function creates and returns a JavaScript **Proxy** object , which intercepts all read and write access, and allows us to make it behave ***like*** an Entity. By extension, the definition of an Entity tree is done with a recursive declaration, which the JavaScript object declaration syntax already allows:

```
let root =
    Entity({ children: [

        Entity({ depth:
            0, bounds: new Bounds(0, 0, 100, 50),
            shape: SHAPE_RECTANGLE,
            backgroundColor: rgba(0, 0, 255),
        }),

        Entity({ depth:
            1, bounds: new Bounds(50, 20, 80, 50),
            shape: SHAPE_RECTANGLE,
            backgroundColor: rgba(255, 0, 0),
        }),
    ],
}
```

As an illustration, we have summarized in Table 9 the differences in Polyphony syntax, between JavaScript and a previous version in Java [Raf18]. Each of these differences is enabled by the **Proxy object's use of metaprogramming.** The major difference between the two versions is in the readability of the syntax, shorter and less punctuated. Thanks to this change in language, we were able to reduce the size of applications using Polyphony, and thus build more complex case studies more easily.

	Java	JavaScript
Entity Initialization	`Entity e = new Entity() .add("property", value) ...`	`let e = Entity({ property: value, ...})`
Reading property	`(Type)e.get("property")`	`e.property`
Writing property	`e.set("property", value)`	`e.property = value`
Property test	`e.has("property")`	`'property' in e`
Adding property	`e.add("property", value)`	`e.property = value`
Property deletion	`e.remove("property")`	`delete e.property`

Table 9: Comparison of Polyphony usage syntaxes between Java et JavaScript

To benefit from a syntax “integrated” with that of the JavaScript language (that is to say which reuse and divert elements of syntax, rather than offering an API by functions), we have made compromises in the choices of Polyphony design. Thus, our framework was influenced in part by certain characteristics of JavaScript:

- Implementing objects as dictionaries naturally prompted us to use this data structure for Entities, rather than a database.
- The absence of types for object fields encouraged us not to impose them a priori in Polyphony, although in the future we do not exclude experimenting with their use.

- An Entity does not “carry” its identifier, unlike frameworks like djnn [Mag14] where the DOM [WHA15]. Identifiers are carried by object properties as well as variables language, which ***reference*** Entities. Thus each Entity can be referenced under several names. This characteristic is integrated into JavaScript, and shared by the majority of languages programming, so for consistency we adopted it in Polyphony.

In practice, Polyphony can be implemented in many languages. Although the choices design have been ***oriented*** to be highly compatible with object-oriented languages dynamic, implementation in a less compatible language is possible. The level of A language's compatibility with Polyphony will depend on support for different features. For clarify this last point, we list in table 10 the characteristics defining this compatibility for five imperative languages.

JavaScript and Python have the best compatibility, you can implement Polyphony based on on the native objects of the language. Smalltalk does not allow you to expand instance variables by particular. However, it allows you to intercept access to properties (which are simple sendings of messages), therefore implementing Entities as objects, but with dictionaries. Java and C do do not allow us to intercept property reads/writes, so we cannot use the native object access syntax (see Table 9). Finally, for languages that do not support simultaneous declaration and initialization of objects, we will not be able to initialize Entities like objects recursively.

	JavaScript	Python	Java	Smalltalk	C
Dictionaries	Yes	Yes	Yes	Yes	non
Concurrent object declaration/initialization	Yes	Yes	non	non	non
Recursive object initialization	Yes	paintings	non	Yes	Yes
Typing	nominative, dynamic	nominative, dynamic	nominative, static	nominative, dynamic	nominative, static
References to functions	objects	objects	objects	objects	pointers
Functions with properties	Yes	Yes	non	non	non
Add/remove variable	Yes	Yes	non	non	non
Add/remove method	Yes	verbose	non	non	non
Extensible Iterators	Yes	Yes	Yes	n/a	non
Intercepting access to a property	Yes	Yes	non	n/a	non
Enumeration of properties	Yes	Yes	Yes	Yes	non

Table 10: Compatibilities of five languages for implementing Polyphony

3.5 Conclusions

In this chapter we discussed the design of a HMI library based on the Entity-Component-System model, and oriented towards the prototyping of new techniques of interaction. We presented the state of the art of libraries related to interface prototyping, and identified the research opportunities that motivated this thesis work. We presented the box to Polyphony tools, and illustrated its use using a vector drawing application. We have then detailed the architecture of Polyphony, focusing on high-level descriptions

allowing it to be replicated in other contexts. Finally, we analyzed the design choices for the ECS implementation using three major variations, and explained our own choices for designing an HMI toolbox.

3.5.1 Contributions and limits

We now discuss the contributions and disadvantages of ECS and Polyphony to the programming of graphical interfaces and interactions, based on our experience in their design and implementation. When designing Polyphony, we were faced with numerous implementation problems for which the chosen solutions could strongly influence the usability of the toolbox. In particular, we could have used common design patterns, such as ***listeners*** or ***delegates.*** Instead, we tried to keep the ECS model as ***pure*** as possible by not mixing it with other paradigms, in order to emphasize its strengths and
its weaknesses in the context of HMIs.

3.5.1.1 Application of a composition model to interaction

Interaction frameworks commonly rely on two tree hierarchies to structure the code and data in an application: a ***type tree*** (classes) to share and reuse the code, and a ***scene tree*** to structure the interface. Type trees propagate the methods and variables of any object type to its children, and allow any child to be used where a parent is expected. This is type polymorphism, and it is the mechanism by which behaviors are generally shared between different widget types. Scene trees are used to structure the interface, and allow the coupling of many aspects of graphical interfaces with a single tree: display order, relative positioning, and style propagation. Some works also introduce a third ***interaction graph,*** to express the dependencies between the interaction techniques [Hud05, Huo04, Mye97].

Video games often use type trees, while game elements generally require less structuring than a scene tree. In this area, the application of a composition model like ECS has proven useful, in part because many behaviors (visibility, physicality, controllability) can be decorrelated and composed at will.
For example, we can instantiate a flowerpot which is visible, which does not physically block the player, and which is inert, but we can also give the player control of this plant, and assign it a mass and a collision box .

For GUIs, many aspects can be disconnected from type and scene trees, and therefore benefit from a composition model. In our example application, the tree traversal typically performed during mouse picking ***and*** graphical rendering of widgets is synthesized into a single DepthUpdateSystem . Likewise, composing the graphic appearance of widgets using simple shapes (filled polygons, lines, images, and text) allows for faster rendering by sending them in groups to the graphics card. Another example is the specification of elastic positioning constraints between widgets in the interface [Bad01], instead of relative positioning of widgets to parents in the scene tree. Finally U.S

We showed with the example of drag and drop how each interaction technique can be composed from the results of simpler interaction techniques (widget targeting, clicking on widget, etc.).

With Polyphony, the ECS composition model completely replaces the type tree, while partially leaning on a stage tree. This is still useful to us for:

- explicitly refer to children rather than using a Selection (e.g. canvas shapes) express display
- order relationships between Entities propagate visual
- styles between entities (e.g. font), although we do not have not yet implemented this functionality

In the future, we plan to experiment with different alternatives, and we are always looking for a good compromise between these structuring principles.

3.5.1.2 Non-hierarchical reuse

ECS implements a powerful reuse mechanism: it avoids ***code duplication*** by allowing Entities to delegate their behaviors to Systems, and it also limits ***execution duplication*** by executing each System once for all Entities. Execution duplication is common for widgets performing periodic processing in instance methods. Indeed, when several widgets of the same type share the same method, it is executed as many times as there are widgets. Systems, on the other hand, carry out the same processing on a whole set of Entities. They can duplicate execution by processing each entity at once, but they also have the option to run them in parallel when possible.

The enumeration of Entities integrated into ECS allows you to not depend on the recursive traversal of a scene tree, and to choose the order in which you traverse them. For example, one can solve layout constraints using a linear optimization solver such as Cassowary [Bad01], rather than propagating constraints with messages between entities [San93]. In ECS implementations, Systems are also useful for implicitly optimizing the rendering of graphics components by sending them in batches ***to*** the graphics API. Such optimizations exist in most interaction frameworks, but they require complex display managers, which manage display caches and compile the multiple drawing instructions into packages for the graphics API.

With Systems, all behaviors are separated from the elements to which they relate (unlike instance methods or callbacks). While this is useful for shared behaviors, which are written and executed once for all Entities, it provides no benefit for individual behaviors — written for an Entity and never reused. More realistic use cases and interfaces need to be designed and implemented with our ECS approach before we can draw firm conclusions, but we could consider a mixed approach using systems for shared behaviors and existing mechanisms for individual behaviors .

3.5.1.3 Dynamicity of interfaces

With Descriptors, we can verify that an Entity respects a given protocol. In this sense they are analogous to interfaces , with the difference that they are dynamically defined, dynamically evaluated (since Entities can change during execution), and are not limited to owned Components (these are programmable conditions) . With Selectors, we can obtain a list of all Entities checking a Descriptor. This mechanism makes it possible to discover Entities according to their capabilities, and to filter them finely in order to select those on which to act.

We have therefore designed interfaces as flexible as possible, in order to allow the addition of new behaviors, both at compilation and at runtime. These behaviors can apply to future Entities as well as to Entities already initialized. Through this mechanism, we want to allow the modification of any existing application for research and prototyping purposes. We are thinking for example of the integration of ExposeHK [Mal13] to an existing application, which consists of displaying a ***tooltip*** with its keyboard shortcut under each button, when pressing the Cmd/Ctrl key. Implementing such an interaction technique would first require enumerating all the buttons, which is achievable using a Selection on the Entities having the **clickable Component.** We thus list the widgets which are buttons by their ***characteristic,*** rather than by the class names which are clickable.

Then, you would have to associate each button with the command it triggers, and likewise each keyboard shortcut with the command it triggers, in order to then associate the buttons with the keyboard shortcuts. If the execution of a command after a click or a keyboard shortcut is managed directly in code, then it will be necessary to inspect the Systems code to find this association. In our drawing application, it would take the following form:

```
let saveButton = Button(new Image('icons/save.png'), 2, y += 34)
...
let SaveSystem = Entity(function() { if
      (saveButton.tmpClick || Keyboards[0].tmpShortcut === SDLK_s) { //
            sauvegarde du canevas
            ...
      }
}, { runOn: POINTER_INPUT | KEYBOARD_INPUT, order: 61 })
```

Alternatively, the **if** statement could be split into two **if** statements calling the same saveCanvas() function , or the save code could be duplicated (which is unlikely if the command is complex). Here, ECS does not easily make it possible to find the association between buttons and keyboard, because the ***causal*** relationship is not explained in an object that can be enumerated, as can be the djnn ***bindings*** [Mag17], or the **Action** class of Swing. In their absence, we will have to inspect the ***bytecode*** of each System to detect **if** statements , or like the authors of ExposeHK, to instrument function calls at the system level. To avoid having to resort to these very complex techniques (and not portable between systems), it is important to include a level ***of indirection*** in the triggering of any

command, which allows you to find the association between a command and its triggers. In our case, this would involve adding attributes to **SaveSystem,** and delegating its activation to the Meta-System (or to a third-party System):

```
let SaveSystem = Entity(function() { //
    save the canvas
    ...
}, { runOn: SHORTCUT_S, order: 61 })
```

More generally, we argue that the use of indirections in the links between elements (objects or processes) contributes to the ***flexibility*** and ***observability*** of applications. Indirection could therefore be a design principle, applicable to all interactive applications, and which we plan to study and promote in the future.

3.5.1.4 Centralization of program logic

The systems help centralize the definition of interactions and behaviors in a manner similar to the state machines in SwingStates [App06]. The tool selector in our drawing application illustrates this principle well. The **ToggleSystem** detects clicks on any **toggleable Entity,** and deactivates all other Entities with the same **toggleGroup.** So none of the tool selection buttons implement an **onClick** callback to activate one tool and deactivate the others. This common behavior is managed in a dedicated, higher-level system.
Additionally, adding a new tool only requires adding the **toggleable** and **toggleGroup** Components to the new button, for **ToggleSystem** to take it into account. It is therefore not necessary to add new callbacks or modify the application code in several places.
For each tool, it is common to have code to execute upon activation or deactivation.
In our application, this code is contained in the System of each activatable tool **(MoveTool** and **DrawTool),** which detects activation with the Temporary Component **tmpToggled == true.**

Another example is the drag and drop implementation we talked about earlier.
The logic of this interaction technique is contained in a single System, despite the many actions required to perform a complete drag and drop. Multimodal interaction techniques could also be implemented, by triggering the same System on several types of events, for example with **runOn: MOUSE_INPUT | KEYBOARD_INPUT.**

The grouping of all the actions of a type of behavior in a single System also implies that the processing of several Systems cannot be intertwined. However, certain operations require ordering processing by Entities, rather than by type of behavior, such as graphic rendering. In this case, drawing background shapes, borders, and images are separate operations performed in order for each Entity. These operations must also be ordered between Entities, that is, drawing the image of a background Entity before the border of a foreground Entity. This is illustrated in Figure 38, where grouping processes into Systems reorganizes their execution.

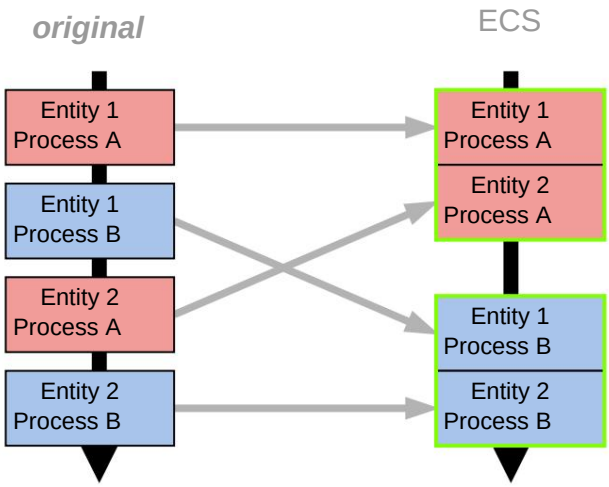

Figure 38: Scheduling of two processes with two Entities. Grouping executions by process type alters the order of executions.

Programming with Systems requires us to make reordered processes insensitive to execution order. In the case of graphic rendering, the use of a ***depth-buffer*** allows all opaque elements to be drawn in any order. For elements with transparency, you should normally draw the background elements before the foreground elements. Our solution is to redirect the drawing instructions for each transparent element to a private texture, and draw them in order in a final System. The application of ECS principles therefore requires minimizing order relationships between entities.

3.5.1.5 Persistent device hardware

In Polyphony, Device Entities are the storage media for data relating to interaction techniques. The different Systems interpreting the techniques in sequence do not have to pass the data as arguments, at the risk of losing data from previous Systems. All data is stored on Peripheral Entities, and therefore allows Systems to access both low-level and high-level data.

Additionally with ECS, input devices are characterized by their Components: **cursorPosition** and **buttons** for mice, **bounds** and **origin** for views, and **keyStates** for keyboards. This principle makes it possible to abstract devices while retaining their individual characteristics. Data such as whether there are extra buttons on a mouse is retained, but simply ignored by any system dealing with two-button pointers.

Through the systematic use of Selections to bring Entities together, interaction techniques are implicitly exposed to the possibility of using multiple devices. Although we have not demonstrated it in our prototype, the use of multiple instances of the same input or output devices (physical or virtual), as well as their replacement, is facilitated because it is implicit, as long as they present the correct Components. For example, a System

implementing a pointing technique could work with any Entity that provides the **cursorPosition** and **buttons components,** whether a mouse, a tablet, or a virtual device producing this data (for example, a “robot” that replays input).

In practice, we observed that the Components were divided into two categories:

- **attributes** — which are only properties of Entities, are often shared by several Systems, and do not provide any specific behavior (e.g. **bounds, keyStates, order) activators**
- — which are often required by ***a*** System, therefore relating to a specific behavior (e.g. **richText, targetable, focus).**

When designing all available Components and corresponding Systems, we must make the activation of each System for each Entity ***predictable.*** To this end, the activation of a System by a single Component is obviously the most predictable (than the combination of several Components), hence the emergence of activators. In our drawing application, for example, **View** shares the **bounds** Component with other Entity types (it's an attribute). Its Activator Component here is **origin,** and it is therefore this which grants the ability to “represent the Entities contained in a given rectangle”. The association of an activator with each System is what allows developers to choose the Systems that will influence an Entity. If an activator is shared by two Systems, it will be impossible to operate one without the other for any Entity.

Generally speaking, the basic ECS model does not have clear links between Systems and the Components they use. Their separation is however important, because it allows the Systems to be replaced at any time, whether to optimize their implementation or increase it. Following the description made by Leonard [Leo99], Components would actually be a ***specification of behaviors*** (especially activators), which are then implemented in Systems. Their naming is therefore essential, as it should inform the behavior learned with each, and spares the need for additional documentation to explain their roles (although this is useful for complex behaviors). In the future, we therefore plan to specify two types of Components, ***attributes*** and ***behaviors,*** in order to encourage developers to name the latter as behaviors (e.g. **clickable, draggable, drawableAs).** They could also benefit from declaration keywords, and syntax highlighting to distinguish them in code.

Conclusion

During this thesis work, we were interested in the programming of research prototypes in HCI, which we sought to make less difficult and less stereotypical. We began by studying programming activity in this context, and to do so carried out a series of interviews with researchers who had developed new interaction techniques. The observations led to three classifications, of the problems participants encountered, the tools they found useful, and the techniques they used to overcome their difficulties.
We then proposed an online questionnaire to evaluate the relevance of the problem classes and techniques with a larger number of HCI participants, and to study the criteria according to which they choose libraries. We deduced that (i) researchers mainly use frameworks in the context of their work, (ii) they prioritize the most documented and used ones, (iii) documentation problems are the most frequent and problems most critical reliability, and (iv) the characteristics best supporting their advanced techniques are the extensibility, reusability, and transparency of the frameworks.

We chose to explore software contributions to improve the programming of HMI research prototypes, and first studied design principles to guide our work. Their synthesis gave rise to three ***Interaction Essentials,*** which clarify and differentiate our research approach compared to the state of the art. We thus seek to (i) **increase the orchestration capabilities of interactive behaviors,** (ii) **provide a complete user input/output environment for any application,** and (iii) **take advantage of programming conventions and integration possibilities. to languages.**

We illustrated the application of the Interaction Essentials with two realizations. The first is an extension of the Smalltalk language to express animated transitions by adding a duration to function calls. It reduces the code necessary to express an animation compared to the state of the art, and brings together in the language the management of animations formerly specific to each framework. This work allowed us to formulate the third Interaction Essential, by analyzing our work in close connection with the programming language. The second achievement is an interaction framework, which applies the Entity-Component-System model from video games, to the programming of graphical interfaces. It mainly meets the needs of reusability (thanks to the composition mechanism integrated into the model), and extensibility (by facilitating the creation of new interaction techniques and types of widgets).

With this work, we sought to improve the programming models of frameworks and languages, for the prototyping of new interactions. We proposed low-level concepts applicable to languages, and facilitating the implementation of interaction frameworks (animation of functions, typing and dynamic enumeration of objects from their fields). State-of-the-art work has mainly proposed reuse models based on type trees (e.g. HTML, Qt, JavaFX), or scenes (e.g. Jazz, UBit), but which fragment the execution of the code (the flow

execution frequently “jumps” in memory, making it difficult to follow and visualize). We propose a composition-based reuse model, which organizes the execution flow in a linear way, and makes the internal workings of the application transparent, for developers wishing to manipulate it. Furthermore, the state of the art has mainly relied on layered user input propagation, which communicates data from layer to layer, without necessarily retaining the raw data. We propose a representation of interaction devices as extensible and persistent objects, to which the different interaction techniques add higher-level data (even temporary data for state transitions). This representation centralizes interaction data, and prevents developers from having to aggregate it from different layers. Finally, numerous state-of-the-art works have proposed reduced programming syntaxes (compared to the use of API functions), thanks to partial integration with programming languages. We supported and deepened this approach, through the development of concepts that can be integrated into languages, and the use of metaprogramming to implement them.

Limits

We have favored the development of new concepts that can facilitate the development of less stereotypical interfaces, rather than tools that facilitate the use of existing concepts. Additionally, we spent a lot of effort refining their design in order to best explore and demonstrate the relevance of our Interaction Essentials. On the other hand, the low maturity of these concepts did not allow us to test them in real conditions. In the case of function animation, it would be appropriate to adapt this concept to other languages, to encourage its more coherent integration with their syntax (e.g. **object.setProperty(target) during** 2s).
In the case of Polyphony, it would be appropriate to validate our work with more complex interfaces, implementing the management of positioning, visual styles, multiple and dynamically changing devices, or even optimized rendering on a graphics card. We could then distribute Polyphony to HMI researchers, and collect their feedback, as well as the strengths and weaknesses of the model for prototyping new interactions.

Furthermore, in this work we have not addressed the documentation problems raised at the end of the interviews and questionnaires. Although this problem is most common for users of interaction frameworks, it benefits from abundant research work. These lead us to believe that the solution cannot be reduced to a new documentation tool, and that it could very well go beyond the scope of a thesis work. Finally, although we observed the low use of frameworks from research to prototype new interactions, we have not explored the avenue of their promotion among researchers. Indeed, the low-level complexity of computer systems with respect to interaction has encouraged us to focus our efforts on reducing it, rather than promoting tools that hide it. We thus hope to accelerate the development of future toolboxes, and thereby contribute to the development of new interactions with machines.

Promote Polyphony for ECS users

Prior to our work on Polyphony, we noted discussions on video game developer forums, which showed an interest in the application of ECS to graphical interface programming. Indeed, while they already apply ECS to the architecture of their games, many developers wish to apply it to the interfaces of the same games, mainly so as not to mix it with an object-oriented model within the same program. On these same forums, we observed many responses recommending against the use of ECS for interfaces, mainly arguing that existing tools are very well suited. So we worked on a topic with an explicit request, and demonstrated that ECS can be applied to GUI design. We further propose to introduce support for Temporary Components, to enable the reification of devices into Entities.

Prospects for future work on ECS: At this stage of our work, we consider the basic model of Polyphony sufficiently stable, however the choices of Components and Systems are still subject to change. Additionally, we consider other languages for implementing Polyphony. The first version was made in Java, which turned out to be verbose in use because of the incompatibility between its object model and ECS (requiring us to use function calls).
The current version is made in JavaScript (with Node.js), but its support for low-level interaction has proven to be very limited, due to the low number of bindings available, very rapid obsolescence of third-party libraries, and general instability of the platform. We are currently considering development in Python, which allows equivalent metaprogramming, while providing sufficient low-level support (with PyGame). It is also possible that we work with an existing framework for ECS, such as Unity. Finally, we are currently in contact with the ECS community, to promote our interaction architecture based on it, and to confront it with discussions with potential users. In the future, we would like to see ECS evolve to explicitly support interaction programming, and in the longer term contribute to the evolution of object/entity languages, and the future design of a dedicated and optimized language for ECS.

Improve low-level support for interaction

As suggested above, we had a lot of difficulty developing the two projects presented in this manuscript, mainly at low level. We partly agree with the problems encountered by the participants of our studies, in that we used libraries (Node.js, OpenGL) for rarely expressed needs, and lacked documentation and contextual examples to support our work. Unlike our participants, our difficulties are not linked to the use of frameworks, but of low-level libraries. These difficulties consisted of:

- the choice of software libraries to use for keyboard/mouse input, and for drawing on screen (because of their number, the uncertainty of the existence of functional bindings for a particular language, the uncertainty as to their performance, the late discovery of missing features, or the late discovery of new libraries)

Conclusion

- choosing an appropriate binding for a given library (most redefine function names, alter API usage concepts, omit certain features, or simply no longer work) implementing a binding for a given library (each language defining its own
- syntax for defining bindings, with often unofficial support and not very stable over time, and not always supporting the entire Binary-Program Interface of the system) the lack of power of the libraries low-level, which requires more development than expected (e.g. writing multi-line text in FreeType,
- drawing lines with thickness in OpenGL, or entering text with SDL) the lack of flexibility of the libraries high level, when you want to use them for simple needs (because of complex installation procedures, incompatibilities between libraries, complex
- models requiring significant learning, or even poor performance)

Low-level tools are therefore not very stable, which is consistent with the idea that few programmers are supposed to access them. However, this situation does not favor the development of new toolboxes, and contributes to the sustainability of major frameworks, which can invest development efforts to test and support numerous alternatives on each operating system. Following on from the second Interaction Essential ***(a minimal interaction environment initialized at the start of any application),*** we wish to study low-level interaction support in the future, to facilitate its access in toolboxes. interaction tools, and in direct use by programmers. This support would take the form of an ***essential*** interaction API , which can be easily used in any language, using a subset of the Binary-Program Interface, and providing a C interface facilitating the automatic generation of bindings . Libraries like SDL already exist to provide full interaction support. However, they suffer from the lack of power mentioned above, which makes it impossible to use them alone.

Perspectives for future work on low-level tools: We believe that it would be necessary to first clarify ***the design space*** of a low-level library, based on the study of the many available libraries. This space could be represented as a dependency graph, where the nodes are the functions of the API, and the edges are the dependencies between them. For example, the **drawingTextMultiLine** function would depend on the drawingWord function , which itself would depend on **drawingLetter.** Each function would weigh its implementation complexity if it were not included in an API. This would then involve maximizing the sum of weights included in an API, while minimizing the number of functions to include. We hope to help stabilize low-level APIs, or even eventually bring them closer to programming languages.

Towards semi-structured interaction programming

Almost all of the frameworks we studied (with the notable exception of ImGUI) require the initialization of a scene tree to create any interactive application.
Whether you want to create a complex interface comprising multiple widgets and various interaction methods, or on the contrary draw a few shapes on the screen, it is always necessary to ***structure*** the interface. Such a structure implies for users of a framework:

- to learn the concepts of scene tree, display order, relative positioning, event propagation ***(event bubbling),*** or even recursive traversal to express their
- application using the reusable widgets of the interface (e.g. a canvas to draw, a transparent view to intercept input data) to initialize a scene tree, with at
- least one view and one container, and place their code in ***callbacks***

These efforts are balanced by the simplicity with which complex interfaces can be expressed. However, they are a major obstacle when it is not a question of creating an interface that is an assembly of widgets. There seems to be a gap between these “assembled” interfaces, and minimal (or less stereotypical) interfaces, for which a scene tree would be superfluous.
We argue that there should be a happy medium, which would allow the use of recognized frameworks, to develop differently structured interfaces. In Polyphony, we have reduced the use of the scene tree, making its use non-systematic. We can describe this model as ***semi-structured,*** because the application has a structure between certain Entities, but not between all.

Perspectives for future work on semi-structured programming: Throughout this thesis manuscript we have sought to understand the nature of “interaction programming”. We characterized it based on the work of HMI practitioners, observed during interviews with researchers, and practiced in the development of our software contributions. However, we mainly observed ***graphical interfaces.*** The idea of interaction programming that we have distilled in this manuscript is in fact biased by the environment where we studied it. This thesis is therefore ***really*** about programming graphical interfaces, although it is the interactions that motivate us. Based on this observation, we wish in the future to acquire a more complete knowledge of the areas where interaction is programmed, and where it is given an original structure, or even no structure. Between all these areas, semi-structured programming could be a unifying concept, which would help to bridge their possible differences, and provide a more complete vision of interaction programming.

Bibliography

[Act14] Acton, M. (2014). ***Data-Oriented Design and C++.*** Accessed at https://www.youtube.com/watch?v=rX0ItVEVjHc

[Api04] Apitz, G., & Guimbretière, F. (2004). CrossY: A Crossing-based Drawing Application. ***Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology,*** 3–12. https://doi.org/10.1145/1029632.1029635

[App06] Apple Inc. (2006). About Core Animation. Accessed April 26, 2017, at http://developer.apple.com/documentation/Cocoa/Conceptual/CoreAnimation_guide/ [App06]

Appert, C., & Beaudouin-Lafon, M. (2006). SwingStates: Adding State Machines to the Swing Toolkit. ***Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology,*** 319–322. https://doi.org/

10.1145/1166253.1166302 [App08] Appert, C., & Beaudouin-Lafon, M. (2008). SwingStates : Adding state machines to Java and the Swing toolkit. ***Software: Practice and Experience,*** 38(11), 1149–1182. https://doi.org/10.1002/spe.867

[App09] Appert, C., Huot, S., Dragicevic, P., & Beaudouin-Lafon, M. (2009). FlowStates: Application Prototyping Interactive With Data Streams And State Machines. ***Proceedings of the 21st International Conference on Association Francophone D'Interaction Homme-Machine,*** 119–128. https://doi.org/10.1145/1629826.1629845

[Ase16] Asenov, D., Hilliges, O., & Müller, P. (2016). The Effect of Richer Visualizations on Code Comprehension. ***Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems,*** 5040–5045. https://doi.org/10.1145/2858036.2858372

[Bad01] Badros, G. J., Borning, A., & Stuckey, P. J. (2001). The Cassowary Linear Arithmetic Constraint Solving Algorithm. ***ACM Trans. Comput.-Hum. Interact.,*** 8(4), 267–306. https://doi.org/10.1145/504704.504705

[Bai08] Bailly, G., Lecolinet, E., & Nigay, L. (2008). Flower Menus: A New Type of Marking Menu with Large Menu Breadth, Within Groups and Efficient Expert Mode Memorization. ***Proceedings of the Working Conference on Advanced Visual Interfaces,*** 15–22. https://doi.org/10.1145/1385569.1385575 [Bai16]

Bailly, G., Lecolinet, E., & Nigay, L. (2016). Visual Menu Techniques. ***ACM Comput. Surv.,*** 49(4), 60:1–60:41. https://doi.org/10.1145/3002171

[Bar18] Baron, D., Jackson, D., & Birtles, B. (2018, October 11). ***CSS Transitions.*** Accessed at https://www.w3.org/TR/css-transitions-1/#animatable-css

[Bau08] Bau, O., & Mackay, W. E. (2008). OctoPocus: A Dynamic Guide for Learning Gesture-based Command Sets. ***Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology,*** 37–46. https://doi.org/10.1145/1449715.1449724

[Bau13] Baudart, G., Mandel, L., & Pouzet, M. (2013). Programming Mixed Music in ReactiveML. ***Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design,*** 11–22. https://doi.org/10.1145/2505341.2505344

[Bea00] Beaudouin-Lafon, M., & Lassen, H. M. (2000). The Architecture and Implementation of CPN2000, a post-WIMP Graphical Application. ***Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology,*** 181–190. https://doi.org/10.1145/354401.354761

[Bea00] Beaudouin-Lafon, M. (2000). Instrumental Interaction: An Interaction Model for Designing post-WIMP User Interfaces. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 446–453. https://doi.org/

10.1145/332040.332473 [Bea00] Beaudouin-Lafon, M., & Mackay, W. E. (2000). Reification, Polymorphism and Reuse : Three Principles for Designing Visual Interfaces. ***Proceedings of the Working Conference on Advanced Visual Interfaces,*** 102–109. https://doi.org/

10.1145/345513.345267 [Bea04] Beaudouin-Lafon, M. (2004). Designing Interaction, Not Interfaces. ***Proceedings of the Working Conference on Advanced Visual Interfaces,*** 15–22. https://doi.org/10.1145/989863.989865

[Bea08] Beaudouin-Lafon, M. (2008). Interaction is the Future of Computing. In T. Erickson & D. McDonald (Éd.), ***HCI Remixed, Reflections on Works That Have Influenced the HCI Community*** (p. 263ÿ266). Consulté à l'adresse http://www.visi.com/~snowfall/HCIremixed.html

[Bea12] Beaudouin-Lafon, M., Huot, S., Nancel, M., Mackay, W., Pietriga, E., Primet, R., ... Klokmose, C. (2012). Multisurface Interaction in the WILD Room. ***Computer,*** 45(4), 48ÿ56. https://doi.org/10.1109/MC.2012.110

[Bed99] Bederson, B. B., & Boltman, A. (1999). Does animation help users build mental maps of spatial information? ***1999 IEEE Symposium on Information Visualization, 1999. (Info Vis '99) Proceedings,*** 28ÿ35. https://doi.org/10.1109/INFVIS.1999.801854

[Bed00] Bederson, B. B., Meyer, J., & Good, L. (2000). Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. ***Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology,*** 171–180. https://doi.org/10.1145/354401.354754

[Bed04] Bederson, B. B., Grosjean, J., & Meyer, J. (2004). Toolkit design for interactive structured graphics. ***IEEE Transactions on Software Engineering,*** 30(8), 535–546. https://doi.org/10.1109/TSE.2004.44

[Ber87] Berry, G., Couronné, P., Gonthier, G., & BANATRE, J.-P. (1987). ***Synchronous Programming of Reactive Systems: The Esterel Language.*** 6(4), 305–316.

[Ber14] Berry, G., & Serrano, M. (2014). Hop and HipHop: Multitier Web Orchestration. In R. Natarajan (Éd.), ***Distributed Computing and Internet Technology*** (pp. 1ÿ13). Accessed at https://link.springer.com/chapter/10.1007/978-3-319-04483-5_1

[Ber17] Béra, C., Miranda, E., Felgentreff, T., Denker, M., & Ducasse, S. (2017). Sista: Saving Optimized Code in Snapshots for Fast Start-Up. ***Proceedings of the 14th International Conference on Managed Languages and Runtimes,*** 1– 11. https://doi.org/10.1145/3132190.3132201

[Bet00] Bétrancourt, M., & Tversky, B. (2000). Effect of computer animation on users' performance : A review. ***Le Travail Humain: A Bilingual and Multi-Disciplinary Journal in Human Factors,*** 63(4), 311ÿ329.

[Bie93] Bier, E. A., Stone, M. C., Pier, K., Buxton, W., & DeRose, T. D. (1993). Toolglass and Magic Lenses: The See-through Interface. ***Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques,*** 73–80. https://doi.org/10.1145/166117.166126

[Bil02] Bilas, S. (2002, mars). ***A Data-Driven Game Object System.*** Présenté à Programming, San Francisco, CA, USA. Consulté à l'adresse https://www.gdcvault.com/play/1022543/A-Data-Driven-Object [Bis98]

Bishop, L., Eberly, D., Whitted, T., Finch, M., & Shantz, M. (1998). Designing a PC game engine. ***IEEE Computer Graphics and Applications,*** 18(1), 46ÿ53. https://doi.org/10.1109/38.637270

[Bla04] Blanch, R., Guiard, Y., & Beaudouin-Lafon, M. (2004). Semantic Pointing: Improving Target Acquisition with Control-display Ratio Adaptation. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 519–526. https://doi.org/10.1145/985692.985758

[Bla06] Blanch, R., & Beaudouin-Lafon, M. (2006). Programming Rich Interactions Using the Hierarchical State Machine Toolkit. ***Proceedings of the Working Conference on Advanced Visual Interfaces,*** 51–58. https://doi.org/10.1145/1133265.1133275

[Blo06] Bloch, J. (2006). How to Design a Good API and Why It Matters. ***Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications,*** 506–507. https://doi.org/

10.1145/1176617.1176622 [Bor97] Borning, A., Marriott, K., Stuckey, P., & Xiao, Y. (1997). Solving Linear Arithmetic Constraints for User Interface Applications. ***Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology,*** 87–96. https://doi.org/

10.1145/263407.263518 [Bor05] Bore, C., & Bore, S. (2005). Profiling software API usability for consumer electronics. ***2005 Digest of Technical Papers. International Conference on Consumer Electronics, 2005. ICCE.,*** 155ÿ156. https://doi.org/10.1109/ICCE.2005.1429764

[Bos09] Bostock, M., & Heer, J. (2009). ***Protovis: A Graphical Toolkit for Visualization.*** 15(6), 1121–1128. https://doi.org/10.1109/TVCG.2009.174

[Bos11] Bostock, M., Ogievetsky, V., & Heer, J. (2011). D3 Data-Driven Documents. ***IEEE Transactions on Visualization and Computer Graphics,*** 17(12), 2301ÿ2309. https://doi.org/10.1109/TVCG.2011.185

[Bou91] Boussinot, F. (1991). Reactive C: An extension of C to program reactive systems. ***Software: Practice and Experience,*** 21(4), 401ÿ428. https://doi.org/10.1002/spe.4380210406

[Bra90] Bracha, G., & Cook, W. (1990). Mixin-based Inheritance. ***Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications,*** 303–311. https://doi.org/10.1145/97945.97982

[Bra08] Brandt, J., Guo, P. J., Lewenstein, J., & Klemmer, S. R. (2008). Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. ***Proceedings of the 4th International Workshop on End-user Software Engineering,*** 1–5. https://doi.org/10.1145/1370847.1370848

[Bra09] Brandt , J. , Guo , PJ , Lewenstein , J. , Dontcheva , M. , & Klemmer , SR (2009). Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 1589–1598. https://doi.org/10.1145/1518701.1518944

[Bra09] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Writing Code to Prototype, Ideate, and Discover. ***IEEE Software,*** 26(5), 18ÿ24. https://doi.org/10.1109/MS.2009.147

[Bro88] Brotman, L. S., & Netravali, A. N. (1988). Motion Interpolation by Optimal Control. ***Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques,*** 309–315. https://doi.org/

10.1145/54852.378531 [Bru09] Bruch, M., Monperrus, M., & Mezini, M. (2009). Learning from Examples to Improve Code Completion Systems. ***Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering,*** 213–222. https://doi.org/

10.1145/1595696.1595728 [Bui02] Buisson, M., Bustico, A., Chatty, S., Colin, F.-R., Jestin, Y., Maury, S., … Truillet, P. (2002). Ivy: Un Bus Logiciel Au Service Du Développement De Prototypes De Systèmes Interactifs. ***Proceedings of the 14th Conference on L'Interaction Homme-Machine,*** 223–226. https://doi.org/10.1145/777005.777040

[Bux90] Buxton, W. A. S. (1990). ***A Three-State Model of Graphical Input.***

[Cao10] Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M., & Grigoreanu, V. (2010). End-user Mashup Programming: Through the Design Lens. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 1009– 1018. https://doi.org/10.1145/1753326.1753477

[Car19] Carnegie Mellon University. (2019). Natural Programming. Accessed September 12, 2019, at https:// www.cs.cmu.edu/~NatProg/ [Cas87]

Caspi, P., Pilaud, D., Halbwachs, N., & Plaice, JA (1987). LUSTER: A Declarative Language for Real-time Programming. ***Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,*** 178–188. https://doi.org/

10.1145/41625.41641 [Cas11] Casiez, G., & Roussel, N. (2011). No More Bricolage!: Methods and Tools to Characterize, Replicate and Compare Pointing Transfer Functions. ***Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology,*** 603–614. https://doi.org/

10.1145/2047196.2047276 [Cha07] Chatty, S., Lemort, A., & Vales, S. (2007). Multiple Input Support in a Model-Based Interaction Framework. ***Second Annual IEEE International Workshop on Horizontal Interactive Human-Computer Systems, 2007. TABLETOP '07,*** 179ÿ186. https://doi.org/10.1109/TABLETOP.2007.27

[Cha08] Chatty, S. (2008). Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering. In ***Lecture Notes in Computer Science. Engineering Interactive Systems*** (p. 356ÿ373). https://doi.org/10.1007/978-3-540-92698-6_22

[Cha12] Chatty, S. (2012). Reconciling Interface Design and Software Design: Towards "Systems-Oriented Design". ***Proceedings of the 2012 Conference on Ergonomics and Human-Computer Interaction,*** 73:73–73:80. https://doi.org/10.1145/2652574.2653412

[Cha14] Chatty, S., & Conversy, S. (2014, June 17). ***What programming languages for interactive systems designers?*** pp. 47-51. Consulté à l'adresse https://hal-enac.archives-ouvertes.fr/hal-01024013/document

[Cha15] Chatty, S., Magnaudet, M., & Prun, D. (2015). Verification of Properties of Interactive Components from Their Executable Code. ***Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 276–285. https://doi.org/10.1145/2774225.2774848

[Cha16] Chatty, S., Magnaudet, M., Prun, D., Conversy, S., Rey, S., & Poirier, M. (2016). Designing, developing and verifying interactive components iteratively with djnn. ***8th European Congress on Embedded Real Time Software and Systems (ERTS 2016).*** Presented in TOULOUSE, France. Accessed at https://hal.archives-ouvertes.fr/ hal-01292291

[Che98] Chell, E. (1998). Critical incident technique. In ***Qualitative methods and analysis in organizational research: A practical guide*** (p. 51ÿ72). Thousand Oaks, CA: Sage Publications Ltd.

[Che16] Chevalier, F., Riche, N. H., Plaisant, C., Chalbi, A., & Hurter, C. (2016). Animations 25 Years Later: New Roles and Opportunities. ***Proceedings of the International Working Conference on Advanced Visual Interfaces,*** 280–287. https://doi.org/10.1145/2909132.2909255

[Cla06] Claypool, M., & Claypool, K. (2006). Latency and Player Actions in Online Games. ***Commun. ACM,*** 49(11), 40–45. https:// doi.org/10.1145/1167838.1167860 [Con98]

Conversy, S., Janecek, P., & Roussel, N. (1998). Factorisons La Gestion Des Événements Des Applications Interactive! ***Proceedings of the 10th Francophone Days on Human-Computer Interaction (IHM'98),*** 141–144. Accessed at http://insitu.lri.fr/roussel/publications/IHM98.pdf

[Con08] Conversy, S., Barboni, E., Navarre, D., & Palanque, P. (2008). Improving Modularity of Interactive Software with the MDPC Architecture. In J. Gulliksen, M. B. Harning, P. Palanque, G. C. van der Veer, & J. Wesson (Éd.), ***Engineering Interactive Systems*** (p. 321–338). https://doi.org/10.1007/978-3-540-92698-6_20

[Con12] Conversy, S. (2012, mars 23). ***A visual perception account of programming languages: Finding the natural science in the art.*** Accessed at https://hal.inria.fr/hal-00737414

[Con13] Conversy, S. (2013). Is there a difference between visual and textual languages in terms of perception? ***Proceedings of the 25th Conference on L'Interaction Homme-Machine,*** 53:53–53:58. https://doi.org/10.1145/2534903.2534911

[Con14] Conversy, S. (2014). Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. ***Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software,*** 201–212. https://doi.org/10.1145/2661136.2661138

[Coo14] Cooney, D. (2014, July 24). ***Introduction to Web Components*** (D. Glazkov & H. Ito, Eds.). Accessed at https://www.w3.org/TR/components-intro/

[Cou87] Coutaz, J. (1987). PAC: AN OBJECT ORIENTED MODEL FOR IMPLEMENTING USER INTERFACES. ***SIGCHI Bull.,*** 19(2), 37–41. https://doi.org/10.1145/36111.1045592

[Cur82] Curry, G., Baer, L., Lipkie, D., & Lee, B. (1982). Traits: An Approach to Multiple-inheritance Subclassing. ***Proceedings of the SIGOA Conference on Office Information Systems,*** 1–9. https://doi.org/10.1145/800210.806468

[Cyp93] Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., & Turransky, A. (Éd.). (1993). ***Watch What I Do : Programming by Demonstration.*** Cambridge, MA, USA: MIT Press.

[Dam97] van Dam, A. (1997). Post-WIMP User Interfaces. ***Commun. ACM,*** 40(2), 63–67. https://doi.org/10.1145/253671.253708

[Dan11] Dann, W. P., & Pausch, R. (2011). ***Learning to Program with Alice*** (3 edition). Boston: Pearson.

[Deb15] Deber, J., Jota, R., Forlines, C., & Wigdor, D. (2015). How Much Faster is Fast Enough? : User Perception of Latency & Latency Improvements in Direct and Indirect Touch. ***Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems,*** 1827–1836. https://doi.org/10.1145/2702123.2702300

[Deh13] Dehouck, M., Bhatti, M. U., Bergel, A., & Ducasse, S. (2013, septembre 10). ***Pragmatic Visualizations for Roassal: A Florilegium.*** Presented at International Workshop on Smalltalk Technologies. Accessed at https://hal.inria.fr/hal-00862065/document

[Dra01] Dragicevic, P., & Fekete, J.-D. (2001). Input Device Selection and Interaction Configuration with ICON. In ***People and Computers XV—Interaction without Frontiers*** (p. 543ÿ558). https://doi.org/10.1007/978-1-4471-0353-0_34

[Dra04] Dragicevic, P. (2004). ***An input interaction model for highly configurable multi-device interactive systems.*** Accessed at https://tel.archives-ouvertes.fr/tel-00426037 [Dra04] Dragicevic, P., &

Fekete, J.-D. (2004). Support for Input Adaptability in the ICON Toolkit. ***Proceedings of the 6th International Conference on Multimodal Interfaces,*** 212–219. https://doi.org/10.1145/1027933.1027969

[Dra11] Dragicevic, P., Huot, S., & Chevalier, F. (2011). Gliimpse : Animating from Markup Code to Rendered Documents and Vice Versa. ***Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology,*** 257–262. https://doi.org/10.1145/2047196.2047229

[Dua11] Duala-Ekoko, E., & Robillard, M. P. (2011). Using Structure-Based Recommendations to Facilitate Discoverability in APIs. ***ECOOP 2011 – Object-Oriented Programming,*** 79ÿ104. https://doi.org/10.1007/978-3-642-22655-7_5

[Dua12] Duala-Ekoko, E., & Robillard, M. P. (2012). Asking and answering questions about unfamiliar APIs: An exploratory study. ***2012 34th International Conference on Software Engineering (ICSE),*** 266ÿ276. https://doi.org/10.1109/ICSE.2012.6227187

[Duc17] Ducasse, S., Zagidulin, D., Hess, N., & Chloupis, D. (2017). ***Pharo by Example 5.0.*** Square Bracket Associates.

[Dur18] Duruisseau, M., Tarby, J.-C., Le Pallec, X., & Gérard, S. (2018). VisUML: A Live UML Visualization to Help Developers in Their Programming Task. In S. Yamamoto & H. Mori (Éd.), ***Human Interface and the Management of Information. Interaction, Visualization, and Analytics*** (p. 3ÿ22). Springer International Publishing.

[Eag11] Eagan, J. R., Beaudouin-Lafon, M., & Mackay, W. E. (2011). Cracking the Cocoa Nut: User Interface Programming at Runtime. ***Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology,*** 225–234. https://doi.org/10.1145/2047196.2047226

[Ecm15] Ecma International. (2015, June). ECMAScript 2015 Language Specification – ECMA-262 6th Edition. Consulted May 14, 2018, at http://www.ecma-international.org/ecma-262/6.0/index.html

[Efr79] Efron, B. (1979). Bootstrap Methods : Another Look at the Jackknife. ***The Annals of Statistics,*** 7(1), 1ÿ26. https://doi.org/10.1214/aos/1176344552

[Fau16] Faulkner, S., Eicholz, A., Width, T., & Danilo, A. (2016, novembre 1). HTML 5.1: 7.8 Animation Frames. Accessed April 6, 2017, at https://www.w3.org/TR/html51/webappapis.html#animation-frames [Fek96] Fekete, J.-D.

(1996, September). ***The three services of the semantic core essential to the HMI.*** 45-50. Accessed at https://hal.inria.fr/hal-00911555 [Fek96] Fekete, J.-D. (1996).

The three services of the semantic core essential to the HMI. ***"Full session of the days of the GDR Programming of the CNRS.*** Accessed at http://insitu.lri.fr/~fekete/ps/ihm96.pdf

[For17] Ford, T. (2017). ***Overwatch Gameplay Architecture and Netcode.*** Accessed at https://www.youtube.com/watch?v=W3aieHjyNvw [Fow05] Fowler,

M. (2005, June 26). InversionOfControl. Accessed August 1, 2019, from https://martinfowler.com/bliki/InversionOfControl.html

[Eng19] Framasoft. (2019). Create and distribute your forms easily | Framaforms.org. Accessed September 22, 2019, à l'adresse https://framaforms.org/

[Gai91] Gaines, B. R. (1991). Modeling and forecasting the information sciences. ***Information Sciences, 57ÿ58,*** 3ÿ22. https://doi.org/10.1016/0020-0255(91)90066-4

[Gaj10] Gajos, K. Z., Weld, D. S., & Wobbrock, J. O. (2010). Automatically generating personalized user interfaces with Supple. ***Artificial Intelligence,*** 174(12), 910ÿ950. https://doi.org/10.1016/j.artint.2010.05.005

[Gal17] Galindo , JA , Dupuy-Chessa , S. , & Ceret , ÿ. (2017). Towards a UI adaptation approach driven by user emotions. ***Proceedings of the tenth international conference on advances in computer-human interactions (ACHI'2017),*** 12ÿ17. JARIA.

[Get02] Gettys, J., Scheifler, RW, Adams, C., Joloboff, V., Hiura, H., McMahon, B., … Yamada, S. (2002). ***Xlib—C Language X Interface: X Window System Standard.*** 476.

[Git13] GitHub. (2013, July 15). Electron | Develop cross-platform desktop applications with JavaScript, HTML and CSS. Accessed August 12, 2019, at https://electronjs.org/ [Gla17]

Glaser, BG, & Strauss, AL (2017). ***Discovering grounded theory—2nd ed. - Strategies for qualitative research.*** Armand Colin.

[GLF02] The GLFW Development Team. (2002). GLFW—An OpenGL library. Accessed June 7, 2019, at https://www.glfw.org/

[Gog14] Goguey, A., Casiez, G., Pietrzak, T., Vogel, D., & Roussel, N. (2014). Adoiraccourcix: Multi-touch Command Selection Using Finger Identification. ***Proceedings of the 26th Conference on L'Interaction Homme-Machine,*** 28–37. https://doi.org/10.1145/2670444.2670446 [Gon96]

Gonzalez, C. (1996). Does Animation in User Interfaces Improve Decision Making? ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 27–34. https://doi.org/10.1145/238386.238396 [Goo08] Google.

(2008). Android. Consulté 26 avril 2017, à l'adresse Android website: https://www.android.com/ [Goo15] Google Inc. (2015).

CORGI: Main Page. Consulté 23 mars 2018, à l'adresse http://google.github.io/corgi/ [Gre96] Green, T. R. G., & Petre, M.

(1996). Usability Analysis of Visual Programming Environments : A 'Cognitive Dimensions' Framework. ***Journal of Visual Languages & Computing,*** 7(2), 131ÿ174. https://doi.org/10.1006/jvlc.1996.0009
[Gre14] GreenSock. (2014, août 3). GSAP,

the standard for JavaScript HTML5 animation. Consulté 26 avril 2017, à GreenSock website address: https://greensock.com/gsap

[Gri09] Grigoreanu, V., Fernandez, R., Inkpen, K., & Robertson, G. (2009). What designers want: Needs of interactive application designers. ***2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC),*** 139ÿ146. https://doi.org/10.1109/VLHCC.2009.5295277

[Gri12] Grill, T., Polacek, O., & Tscheligi, M. (2012). Methods towards API Usability: A Structural Analysis of Usability Problem Categories. ***Human-Centered Software Engineering,*** 164ÿ180. https://doi.org/10.1007/978-3-642-34347- 6_10

[Gro05] Grossman, T., & Balakrishnan, R. (2005). The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 281–290. https://doi.org/10.1145/1054972.1055012

[Hee08] Heer, J., Agrawala, M., & Willett, W. (2008). Generalized Selection via Interactive Query Relaxation. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 959–968. https://doi.org/10.1145/1357054.1357203

[Hee08] Heer, J., Mackinlay, J., Stolte, C., & Agrawala, M. (2008). Graphical Histories for Visualization: Supporting Analysis, Communication, and Evaluation. ***IEEE Transactions on Visualization and Computer Graphics,*** 14(6), 1189ÿ1196. https://doi.org/10.1109/TVCG.2008.137

[Hen90] Henry, T. R., Hudson, S. E., & Newell, G. L. (1990). Integrating Gesture and Snapping into a User Interface Toolkit. ***Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology,*** 112– 122. https://doi.org/

10.1145/97924.97938 [Hen09] Henning, M. (2009). API Design Matters. ***Commun. ACM,*** 52(5), 46–56. https://doi.org/10.1145/1506409.1506424

[Hic99] Hickson, I. (1999, janvier 26). Acid Tests. Consulté 12 août 2019, à l'adresse https://www.acidtests.org/ [Hor17]

Hornbæk, K., & Oulasvirta, A. (2017). What Is Interaction? ***Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems,*** 5040–5052. https://doi.org/10.1145/3025453.3025765 [Hud05]

Hudson, S. E., Mankoff, J., & Smith, I. (2005). Extensible Input Handling in the subArctic Toolkit. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 381–390. https://doi.org/

10.1145/1054972.1055025 [Huo04] Huot, S., Dumas, C., Dragicevic, P., Fekete, J.-D., & Hégron, G. (2004). The MaggLite post-WIMP Toolkit: Draw It, Connect It and Run It. ***Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology,*** 257–266. https://doi.org/10.1145/1029632.1029677

[Huo06] Huot, S., Dragicevic, P., & Dumas, C. (2006). Flexibility and Modularity for Interaction Design: The Software Architecture Model of Combined Graphs. ***Proceedings of the 18th Conference on Human-Computer Interaction,*** 43–50. https://doi.org/10.1145/1132736.1132742

[Huo13] Huot, S. (2013). ***'Designeering Interaction': A Missing Link in the Evolution of Human-Computer Interaction*** (Thesis). Accessed at https://tel.archives-ouvertes.fr/tel-00823763 [IEE08]

IEEE Computer Society. (2008). Opportunistic System Development. ***IEEE Software,*** 25(6).

[ISO18] ISO/IEC. (2018, June). ***Information technology—Programming languages—C.*** Accessed at http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/45/74528.html [ISO18] ISO/

TC 159/SC 4. (2018, mars). ISO 9241-11:2018. Consulté 4 septembre 2019, à l'adresse ISO website: http:// www.iso.org/cms/render/live/fr/sites/isoorg/contents/data/standard/06/35/63500.html [Jac16] Jack, R. H.,

Stockman, T., & McPherson, A. (2016). Effect of Latency on Performer Interaction and Subjective Quality Assessment of a Digital Musical Instrument. ***Proceedings of the Audio Mostly 2016,*** 116–123. https://doi.org/ 10.1145/2986416.2986428 [Jai07] Jaimes,

A., & Sebe, N. (2007). Multimodal human–computer interaction: A survey. ***Computer Vision and Image Understanding,*** 108(1), 116ÿ134. https://doi.org/10.1016/j.cviu.2006.10.019 [Jan13] Jankowski,

J., & Hachet, M. (2013, mai 6). ***A Survey of Interaction Techniques for Interactive 3D Environments.*** Presented at Eurographics 2013 - STAR. Accessed at https://hal.inria.fr/hal-00789413/document

[Joh88] Johnson, R., & Foote, B. (1988). Designing Reusable Classes. ***Journal of Object-Oriented Programming,*** 1(2), 22ÿ35.

[Kal05] Kaltenbrunner, M., Bovermann, T., Bencina, R., & Costanza, E. (2005). ***TUIO A Protocol for Table-Top Tangible User Interfaces.*** Accessed from https://doi.org/files/publications/07a830-GW2005-KaltenBoverBencinaConstanza.pdf

[Ked17] Kedia, P., Costa, M., Parkinson, M., Vaswani, K., Vytiniotis, D., & Blankstein, A. (2017). Simple, Fast, and Safe Manual Memory Management. ***Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation,*** 233–247. https://doi.org/10.1145/3062341.3062376

[Ken02] Kennedy, K., & Mercer, R. E. (2002). Planning Animation Cinematography and Shot Structure to Communicate Theme and Mood. ***Proceedings of the 2Nd International Symposium on Smart Graphics,*** 1–8. https://doi.org/10.1145/569005.569006

[Kic97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In M. Akÿit & S. Matsuoka (Éd.), ***ECOOP'97—Object-Oriented Programming*** (p. 220ÿ242). Springer Berlin Heidelberg.

[Kin12] Kin, K., Hartmann, B., DeRose, T., & Agrawala, M. (2012). Proton: Multitouch Gestures As Regular Expressions. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 2885–2894. https://doi.org/

10.1145/2207676.2208694 [Kin12] Kin, K., Hartmann, B., DeRose, T., & Agrawala, M. (2012). Proton++: A Customizable Declarative Multitouch Framework. ***Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology,*** 477–486.

https://doi.org/10.1145/2380116.2380176 [Kle05] Klein, C., & Bederson, B. B. (2005). Benefits of Animated Scrolling. ***CHI '05 Extended Abstracts on Human Factors in Computing Systems,*** 1965–1968.

https://doi.org/10.1145/1056808.1057068 [Ko11] Ko, A. J., & Riche, Y. (2011). The role of conceptual knowledge in API usability. ***2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/ HCC),*** 173ÿ176. https://doi.org/10.1109/VL

[Kra88] Krasner, G. E., & Pope, S. T. (1988). A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. ***J. Object Oriented Program.***

[Lan98] Lantinga, S. (1998). Simple DirectMedia Layer - Homepage. Accessed February 26, 2019, at http://libsdl.org/

[Lec99] Lecolinet, E. (1999). A Brick Construction Game Model for Creating Graphical User Interfaces: The Ubit Toolkit. ***Proc. INTERACT'99,*** 510–518.

[Lec03] Lecolinet, E. (2003). A Molecular Architecture for Creating Advanced GUIs. ***Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology,*** 135–144. https://doi.org/10.1145/964696.964711

[Lec16] Lecrubier, V. (2016). ***A formal language for designing, specifying and verifying critical embedded human machine interfaces*** (Theses, HIGHER INSTITUTE OF AERONAUTICS AND SPACE (ISAE); UNIVERSITY OF TOULOUSE). Accessed at https://hal.archives-ouvertes.fr/tel-01455466 [Led18] Ledo, D., Houben,

S., Vermeulen, J., Marquardt, N., Oehlberg, L., & Greenberg , S. (2018). Evaluation Strategies for HCI Toolkit Research. ***Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems,*** 36:1–36:17. https://doi.org/10.1145/3173574.3173610 [LeG91] LeGuernic, P.,

Gautier, T., Borgne, ML, & Maire, CL (1991). Programming real-time applications with SIGNAL. ***Proceedings of the IEEE,*** 79(9), 1321-1336. https://doi.org/10.1109/5.97301 [Lei15] Leite Neto, NM, Lenormand, J.,

du Bousquet, L., & Dupuy-Chessa, S. (2015). Toward testing multiple users interface versions. In T. G. Aho, Pekka; Vos Juan; Bøegh, Jørgen; Rennoch, Axel (Éd.), ***Joint research workshop 10th systems testing and validation (STV15) and 1st international workshop on user interface test automation (INTUITEST 2015)*** (p. 61ÿ72). Sophia-Antipolis, France: Institute Fraunhofer.

[Leo99] Leonard, T. (1999, July 9). Postmortem: Thief: The Dark Project. Accessed March 12, 2018, at https://www.gamasutra.com/view/feature/131762/postmortem_thief_the_dark_project.php [Let10] Letondal,

C., Chatty, S., Philips, G., André, F. , & Conversy, S. (2010, September 19). ***Usability requirements for interaction-oriented development tools.*** pp xxx. Accessed at https://hal-enac.archives-ouvertes.fr/hal-01022441/document

[Let14] Letondal, C., Pillain, P.-Y., Verdurand, E., Prun, D., & Grisvard, O. (2014, septembre 9). ***Of Models, Rationales and Prototypes: Studying Designer Needs in an Airborne Maritime Surveillance Drawing Tool to Support Audio Communication.*** pp 8. https://doi.org/10.14236/ewic/

hci2014.8 [Lie01] Lieberman, H. (2001). ***Your Wish is My Command.*** https://doi.org/10.1016/B978-1-55860-688-3.X5000-3

[Lim05] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., & López-Jaquero, V. (2005). USIXML: A Language Supporting Multi-path Development of User Interfaces. In R. Bastide, P. Palanque, & J. Roth (Éd.), ***Engineering Human Computer Interaction and Interactive Systems*** (p. 200ÿ220). Springer Berlin Heidelberg.

[Llo09] Llopis, N. (2009, décembre 4). Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within. Consulté 7 mars 2019, à l'adresse http://gamesfromwithin.com/data-oriented-design

[Mac00] Mackay, W. (2000). Responding to cognitive overload: Co-adaptation between users and technology. ***Intellectica,*** 30(1), 177ÿ193. https://doi.org/10.3406/intel.2000.1597

[Mag14] Magnaudet, M., & Chatty, S. (2014). What Should Adaptivity Mean to Interactive Software Programmers? ***Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 13–22. https://doi.org/10.1145/2607023.2607028

[Mag17] Magnaudet, M., Rey, S., & Conversy, S. (2017). Djnn: A Process Oriented Programming Language for Interactive Systems. ***Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 159–160. https://doi.org/10.1145/3102113.3102161

[Mag18] Magnaudet, M., Chatty, S., Conversy, S., Leriche, S., Picard, C., & Prun, D. (2018). Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. ***Proc. ACM Hum.-Comput. Interact.,*** 2(EICS), 12:1–12:27. https://doi.org/10.1145/3229094

[Mal95] Maloney, J. H., & Smith, R. B. (1995). Directness and Liveness in the Morphic User Interface Construction Environment. ***Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology,*** 21–28. https://doi.org/10.1145/215585.215636

[Mal13] Malacria, S., Bailly, G., Harrison, J., Cockburn, A., & Gutwin, C. (2013). Promoting Hotkey Use Through Rehearsal with ExposeHK. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 573–582. https://doi.org/10.1145/2470654.2470735

[Man09] Mandel, L., & Plateau, F. (2009). Interactive Programming of Reactive Systems. ***Electronic Notes in Theoretical Computer Science,*** 238(1), 21ÿ36. https://doi.org/10.1016/j.entcs.2008.01.004

[Mar07] Martin, A. (2007, September 3). Entity Systems are the future of MMOG development – Part 1. Accessed May 9, 2018, at t-machine.org website: http://t-machine.org/index.php/2007/09/03/entity- systems-are-the-future-of-mmog-development-part-1/

[Mar14] Martin, A. (2014, November 30). What's an Entity System? -Entity Systems Wiki. Accessed September 19, 2019, at http://entity-systems.wikidot.com/ [Mar17]

Marquardt, N., Houben, S., Beaudouin-Lafon, M., & Wilson, AD (2017). HCITools: Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits. ***Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems,*** 624–627. https://doi.org/10.1145/3027063.3027073

[May12] Mayer, D. (2012, November 11). Ratio of bugs per line of code. Accessed August 8, 2019, at https://www.mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio [McC04]

McConnell, S. (2004). ***Code Complete: A Practical Handbook of Software Construction, Second Edition*** (2nd edition). Redmond, Wash: Microsoft Press.

[McL98] McLellan, S. G., Roesler, A. W., Tempest, J. T., & Spinuzzi, C. I. (1998). Building more usable APIs. ***IEEE Software,*** 15(3), 78ÿ86. https://doi.org/10.1109/52.676963

[Mil56] Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. ***Psychological Review,*** 63(2), 81ÿ97. https://doi.org/10.1037/h0043158

[Mir12] Mirlacher, T., Palanque, P., & Bernhaupt, R. (2012). Engineering Animations in User Interfaces. ***Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 111–120. https://doi.org/10.1145/2305484.2305504

[Moo10] Mooty, M., Faulring, A., Stylos, J., & Myers, B. A. (2010). Calcite: Completing Code Completion for Constructors Using Crowds. ***2010 IEEE Symposium on Visual Languages and Human-Centric Computing,*** 15ÿ22. https://doi.org/10.1109/

VLHCC.2010.12 [Moz19] Mozilla. (2019, mars 23). CSS Inheritance. Consulté 18 septembre 2019, à l'adresse MDN Web Docs website: https://developer.mozilla.org/en-US/docs/Web/CSS/inheritance

[Mur05] Muratori, C. (2005). Immediate-Mode Graphical User Interfaces (2005). Consulté 12 février 2019, à l'adresse https:// caseymuratori.com/blog_0001 [Mye86]

Myers, B. A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 59–66. https://doi.org/

10.1145/22627.22349 [Mye90] Myers, B. A., Giuse, D. A., Dannenberg, R. B., Zanden, B. V., Kosbie, D. S., Pervin, E., … Marchal, P. (1990). Garnet: comprehensive support for graphical, highly interactive user interfaces. ***Computer,*** 23(11), 71ÿ85. https://doi.org/10.1109/2.60882

[Mye91] Myers, B. A. (1991). Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs. ***Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology,*** 211–220. https://doi.org/

10.1145/120782.120805 [Mye94] Myers, B. (1994). Challenges of HCI Design and Implementation. ***interactions,*** 1(1), 73–83. https://doi.org/

10.1145/174800.174808 [Mye97] Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., … Doane, P. (1997). The Amulet environment: new models for effective user interface software development. ***IEEE Transactions on Software Engineering,*** 23(6), 347ÿ365. https://doi.org/10.1109/32.601073

[Mye04] Myers, B. A., Pane, J. F., & Ko, A. (2004). Natural Programming Languages and Environments. ***Commun. ACM,*** 47(9), 47–52. https://doi.org/10.1145/1015864.1015888

[Mye08] Myers, B., Park, S. Y., Nakano, Y., Mueller, G., & Ko, A. (2008). How designers design and program interactive behaviors. ***2008 IEEE Symposium on Visual Languages and Human-Centric Computing,*** 177ÿ184. https://doi.org/10.1109/VLHCC.2008.4639081

[Mye08] Myers, B. A., Ko, A. J., Park, S. Y., Stylos, J., LaToza, T. D., & Beaton, J. (2008). More Natural End-user Software Engineering. ***Proceedings of the 4th International Workshop on End-user Software Engineering,*** 30–34. https://doi.org/10.1145/1370847.1370854

[Nan14] Nancel, M., & Cockburn, A. (2014). Causality: A Conceptual Model of Interaction History. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 1777–1786. https://doi.org/10.1145/2556288.2556990

[Nav01] Navarre, D., Palanque, P., Bastide, R., & Su, O. (2001). A model-based tool for interactive prototyping of highly interactive applications. ***Proceedings 12th International Workshop on Rapid System Prototyping. RSP 2001,*** 136ÿ141. https://doi.org/10.1109/IWRSP.2001.933851

[Nav09] Navarre, D., Palanque, P., Ladry, J.-F., & Barboni, E. (2009). ICOs : A Model-based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. ***ACM Trans. Comput.-Hum. Interact.,*** 16(4), 18:1–18:56. https://doi.org/10.1145/1614390.1614393 [Ng14]

Ng, A., Annett, M., Dietz, P., Gupta, A., & Bischof, W. F. (2014). In the Blink of an Eye : Investigating Latency Perception During Stylus Interaction. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 1103–1112. https://doi.org/10.1145/2556288.2557037

[Nie19] Niedoba, M. (2019, April 19). Managing complexity. Accessed September 6, 2019, at Medium website: https://uxdesign.cc/managing-complexity-8094c316a506

[Nor88] Norman, D. A. (1988). ***The Design of Everyday Things.*** New York: Basic Books.

[Nor10] Norman, D. A. (2010). ***Living with Complexity.*** Cambridge, Mass: MIT Press.

[Nor10] Norman, D. A. (2010). Technology First, Needs Last: The Research-product Gulf. ***Interactions,*** 17(2), 38–42. https://doi.org/10.1145/1699775.1699784

[Obr08] Obrenovic, Ž., Gaševic, D., & Eliëns, A. (2008). Stimulating Creativity through Opportunistic Software Development. ***IEEE Software,*** 25(6), 64ÿ70. https://doi.org/10.1109/MS.2008.162 [Oma12]

Omar, C., Yoon, Y. S., LaToza, T. D., & Myers, B. A. (2012). Active code completion. ***2012 34th International Conference on Software Engineering (ICSE),*** 859ÿ869. https://doi.org/10.1109/ICSE.2012.6227133 [One12] Oney, S., &

Brandt, J. (2012). Codelets: Linking Interactive Documentation and Example Code in the Editor. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 2697–2706. https://doi.org/10.1145/2207676.2208664

[One14] Oney, S., Myers, B., & Brandt, J. (2014). InterState : A Language and Environment for Expressing Interface Behavior. ***Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology,*** 263–272. https://doi.org/10.1145/2642918.2647358

[Ora08] Oracle Corp. (2008). Client Technologies: Java Platform, Standard Edition (Java SE) 8 Release 8. Accessed 26 April 2017, at http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

[Pap18] Papari, A. (2018). ***artemis-odb: A continuation of the popular Artemis ECS framework*** [Java]. Consulté à l'adresse https://github.com/junkdog/artemis-odb (Original work published 2012)

[Pat09] Paterno', F., Santoro, C., & Spano, L. D. (2009). MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments. ***ACM Trans. Comput.-Hum. Interact.,*** 16(4), 19:1–19:30. https://doi.org/10.1145/1614390.1614394

[Pav11] Pavlovych, A., & Stuerzlinger, W. (2011). Target Following Performance in the Presence of Latency, Jitter, and Signal Dropouts. ***Proceedings of Graphics Interface 2011,*** 33–40. Consulté à l'adresse http://dl.acm.org/citation.cfm?id=1992917.1992924 Piccioni, M., Furia, C. A.,

[Pic13] & Meyer, B. (2013). An Empirical Study of API Usability. ***2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement,*** 5ÿ14. https://doi.org/10.1109/ESEM.2013.14

[Pla15] Plantec, A. (2015). Block: A new Morphic framework. Accessed April 7, 2017, at http://sdmeta.gforge.inria.fr/YouTubeVideos/PharoPreview/BlocSlides-ESUG2015.pdf

[Poo16] Poor, GM, Jaffee, SD, Leventhal, LM, Ringenberg, J., Klopfer, DS, Zimmerman, G., & Klein, BA (2016). Applying the Norman 1986 User-Centered Model to Post-WIMP UIs : Theoretical Predictions and Empirical Outcomes. ***ACM Trans. Comput.-Hum. Interact.,*** 23(5), 30:1–30:33. https://doi.org/10.1145/2983531

[Pri18] prime31. (2018). ***Nez is a free 2D focused framework that works with MonoGame and FNA*** [C#]. Consulté à l'adresse https://github.com/prime31/Nez (Original work published 2016)

[Qt19] The Qt Company. (2019). Qt | Cross-platform software development for embedded & desktop. Accessed April 18, 2019, at https://www.qt.io [Raf16] Raffaillac, T. (2016). ***Show***

us your Project—Session 2. Accessed at https://youtu.be/ltXFtdhKuM8?t=3m25s

[Raf17] Raffaillac, T. (2017). Language and System Support for Interaction. ***Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 149–152. https://doi.org/10.1145/3102113.3102155

[Raf17] Raffaillac, T. (2017). ***PharoDays17: Show us your Projects.*** Accessed at https://youtu.be/1yzc-EKFVs0?t=28m5s

[Raf17] Raffaillac, T., Huot, S., & Ducasse, S. (2017). Turning Function Calls into Animations. ***Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 81–86. https://doi.org/10.1145/3102113.3102134

Bibliography

[Raf18] Raffaillac, T., & Huot, S. (2018). Application of the Entity-Component-System model to programming d'interactions. ***Proceedings of the 30th Conference on L'Interaction Homme-Machine,*** 42–51. https://doi.org/10.1145/3286689.3286703

[Raf19] Raffaillac, T., & Huot, S. (2019). Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model. ***Proc. ACM Hum.-Comput. Interact.,*** 3(EICS), 8:1–8:22. https://doi.org/10.1145/3331150

[Rea14] Reas, C., & Fry, B. (2014). ***Processing: A Programming Handbook for Visual Designers and Artists*** (second edition edition). Cambridge, Massachusetts: The MIT Press.

[Res09] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. (2009). Scratch: Programming for All. ***Commun. ACM,*** 52(11), 60–67. https://doi.org/10.1145/1592761.1592779

[Rey15] Rey, S., Conversy, S., Magnaudet, M., Poirier, M., Prun, D., Vinot, J.-L., & Chatty, S. (2015). Using the Djnn Framework to Create and Validate Interactive Components Iteratively. ***Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 230–233. https://doi.org/10.1145/2774225.2775438

[Rob09] Robillard, M. P. (2009). What Makes APIs Hard to Learn? Answers from Developers. ***IEEE Software,*** 26(6), 27ÿ34. https://doi.org/10.1109/MS.2009.193

[Rob11] Robillard, M. P., & DeLine, R. (2011). A field study of API learning obstacles. ***Empirical Software Engineering,*** 16(6), 703ÿ732. https://doi.org/10.1007/s10664-010-9150-8

[Rou12] Roussel, N., Casiez, G., Aceituno, J., & Vogel, D. (2012). Giving a Hand to the Eyes : Leveraging Input Accuracy for Subpixel Interaction. ***Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology,*** 351–358. https://doi.org/10.1145/2380116.2380162

[Roy19] Roy, Q. (2019). ***Marking-Menu*** [JavaScript]. Accessed at https://github.com/QuentinRoy/Marking-Menu (Original work published 2017)

[Sai15] Saied, M. A., Sahraoui, H., & Dufour, B. (2015). An observational study on API usage constraints and their documentation. ***2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER),*** 33ÿ42. https://doi.org/10.1109/SANER.2015.7081813

[San93] Sannella, M., Maloney, J., Freeman-Benson, B., & Borning, A. (1993). Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. ***Software: Practice and Experience,*** 23(5), 529ÿ566. https://doi.org/10.1002/spe.4380230507

[Sat14] Satyanarayan, A., Wongsuphasawat, K., & Heer, J. (2014). Declarative Interaction Design for Data Visualization. ***Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology,*** 669–678. https://doi.org/

10.1145/2642918.2647360 [Sch07] Schlienger, C., Conversy, S., Chatty, S., Anquetil, M., & Mertz, C. (2007). Improving Users' Comprehension of Changes with Animation and Sound: An Empirical Assessment. ***Human-Computer Interaction – INTERACT 2007,*** 207ÿ220. https://doi.org/10.1007/978-3-540-74796-3_20

[Sch18] Schmid, S. (2018). ***Entitas-CSharp: Entitas is a super fast Entity Component System (ECS) Framework specifically made for C# and Unity*** [C#]. Consulté à l'adresse https://github.com/sschmid/Entitas-CSharp (Original work published 2014)

[Sha07] Shanmugasundaram, M., Irani, P., & Gutwin, C. (2007). Can Smooth View Transitions Facilitate Perceptual Constancy in Node-link Diagrams? ***Proceedings of Graphics Interface 2007,*** 71–78. https://doi.org/10.1145/1268517.1268531

[Sit19] Sitnik, A., & Solovev, I. (2019, August). Easing Functions Cheat Sheet. Accessed September 15, 2019, at http://easings.net/

[Spa13] Spano, L. D., Cisternino, A., Paternò, F., & Fenu, G. (2013). GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition. ***Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems,*** 187–196. https://doi.org/10.1145/2494603.2480307

[Sta93] Stasko, J., Badre, A., & Lewis, C. (1993). Do Algorithm Animations Assist Learning? : An Empirical Study and Analysis. ***Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems,*** 61–66. https://doi.org/10.1145/169059.169078

[Sty08] Stylos, J., Graf, B., Busse, D. K., Ziegler, C., Ehret, R., & Karstens, J. (2008). A case study of API redesign for improved usability. ***2008 IEEE Symposium on Visual Languages and Human-Centric Computing,*** 189ÿ192. https://doi.org/10.1109/VLHCC.2008.4639083

[Sty09] Stylos, J., Myers, B. A., & Yang, Z. (2009). Jadeite : Improving API Documentation Using Usage Information. ***CHI '09 Extended Abstracts on Human Factors in Computing Systems,*** 4429–4434. https://doi.org/10.1145/1520340.1520678

[Swe85] Sweet, R. E. (1985). The Mesa Programming Environment. ***Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments,*** 216–229. https://doi.org/10.1145/800225.806843 [Tio19] TIOBE.

(2019, septembre). TIOBE Index | TIOBE - The Software Quality Company. Consulté 15 septembre 2019, at https://www.tiobe.com/tiobe-index/ Tucker, A.B. (2004).

[Tuc04] ***Computer Science Handbook, Second Edition.*** Chapman & Hall/CRC.

[Uni17] Unity Technologies. (2017). Unity - Scripting API: MonoBehaviour. Consulté 16 avril 2018, à l'adresse https:// docs.unity3d.com/ScriptReference/MonoBehaviour.html [Uni19] Unity

Technologies. (2019). DOTS - Unity's new multithreaded Data-Oriented Technology Stack. Consulté 19 septembre 2019, à l'adresse Unity website: https://unity.com/dots Victor, B. (2012, septembre). Learnable

[Vic12] programming. Consulté 12 septembre 2019, à l'adresse Bret Victor, beast of burden website: http://worrydream.com/#!/ LearnableProgramming Vidal, C., Berry, G., & Serrano, M. (2018). Hiphop.Js : A Language

[Vid18] to Orchestrate Web Applications. ***Proceedings of the 33rd Annual ACM Symposium on Applied Computing,*** 2193–2195. https://doi.org/10.1145/3167132.3167440

[W3C96] W3C. (1996, December 17). Cascading Style Sheets, level 1. Accessed May 17, 2018, at https://www.w3.org/ TR/CSS1/

[W3C19] W3C. (2019). All Standards and Drafts—W3C. Accessed September 10, 2019, at https://www.w3.org/ TR/

[Wag95] Wagner, A., Curran, P., & O'Brien, R. (1995). Drag Me, Drop Me, Treat Me Like an Object. ***Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,*** 525–530. https://doi.org/10.1145/223904.223975 [Wei97] Weiser,

M., & Brown, J. S. (1997). The Coming Age of Calm Technology. In P. J. Denning & R. M. Metcalfe (Éd.), ***Beyond Calculation: The Next Fifty Years of Computing*** (p. 75ÿ85). https://doi.org/10.1007/978-1-4612-0685-9_6

[Wei10] Weiss, M., & Sari, S. (2010). Evolution of the Mashup Ecosystem by Copying. ***Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups,*** 11:1–11:7. https://doi.org/10.1145/1944999.1945010

[Wen01] Wengraf, T. (2001). ***Qualitative Research Interviewing: Biographic Narrative and Semi-Structured Methods.*** London; Thousand Oaks, Calif: SAGE Publications Ltd.

[WHA15] The WHATWG community. (2015, novembre 19). DOM Standard. Consulté 17 janvier 2019, à l'adresse Web Hypertext Application Technology Working Group website: https://dom.spec.whatwg.org/ [Wik19] Wikipedia. (2019).

Hacking. In ***Wikipédia.*** Consulté à l'adresse https://fr.wikipedia.org/w/index.php? title=Hacking&oldid=162099826 Wiley, D. J., & Hahn, J. K. (1997). Interpolation

[Wil97] synthesis of articulated figure motion. ***IEEE Computer Graphics and Applications,*** 17(6), 39ÿ45. https://doi.org/ 10.1109/38.626968 Yu, J., Benatallah, B., Casati, F., & Daniel, F. (2008).

[yu08] Understanding Mashup Development. ***IEEE Internet Computing,*** 12(5), 44ÿ52. https://doi.org/10.1109/MIC.2008.114

[Zha12] Zhai, S., & Kristensson, P. O. (2012). The Word-gesture Keyboard: Reimagining Keyboard Interaction. ***Commun. ACM,*** 55(9), 91–101. https://doi.org/10.1145/2330667.2330689

[Zho09] Zhong, H., Xie, T., Zhang, L., Pei, J., & Mei, H. (2009). MAPO: Mining and Recommending API Usage Patterns. ***ECOOP 2009 – Object-Oriented Programming,*** 318ÿ343. https://doi.org/10.1007/978-3-642-03013-0_15 Zibran,

[Zib11] M. F., Eishita, F. Z., & Roy, C. K. (2011). Useful, But Usable? Factors Affecting the Usability of APIs. ***2011 18th Working Conference on Reverse Engineering,*** 151ÿ155. https://doi.org/10.1109/WCRE.2011.26

Appendix A. Study plans

A.1 Plan des interviews

Introduction

This interview is part of my PhD, where I look at the limits of GUI libraries and frameworks for prototyping and building interactive applications, or advanced interaction techniques (Qt, Cocoa, Swing, SDL, …), and in particular how they are hacked around in actual projects, to get things done. I would like to backtrack with you a few of your past works, where the library could not do everything you intended, so you had to hack your way in. I selected some on the Internet already, but we can review another one if it is more relevant to you.

Questions

1. Age? Years of experience? In main language? Frequency of programming? Languages/IDEs/frameworks of choice?
2. Which platform/language/IDE/framework(s) did you use, and why these choices? Approximately how many lines of code is the project? How long did it take to code it? How many versions did you do, w.r.t. refactoring?
3. At which point in the design/prototyping process did you get a working software prototype? What did it implement already? What was left to implement?
4. What were your ambitions at the start of the project? Is there some of your ideas that you could not implement and test because of technological issues/limitations?
5. What was the most difficult thing you had to implement? Would you consider it ***hacking?*** What would you consider ***low-level*** there?
6. On a Likert scale (from 1 very dirty to 5 very clean), how “dirty” is it now? If you had the opportunity to recode it, how different would it be?
7. How did you learn the framework(s)? (official doc, book, tutorials, copy/paste examples) How much time did you dedicate to it? Did you have to learn some additional API over the course of the project?
8. With hindsight, what would have helped you best to complete the project? (excluding any library done after) A better framework? A better tutorial?
9. Now if you were to add this code to one of the libraries you used, which one would it be? (higher/lower level, new library) Why?
10. How do you think the framework(s) should have been designed to best suit your need? (may answer weeks later)
11. [Do you have any expectations about my work? :]

Final words

Thank you for your time!
I can send you the results of this study later if you want. Also, it would be nice if we can schedule a short meeting in about two weeks, in case you have some more feedback for this study.

A.2 Questionnaire outline

The goal of this survey is to better understand the process of programming new interactive artefacts for research and innovation purposes, and the software libraries used to support this process. We are interested in projects where you reached the limits of libraries (e.g. undocumented needs, accessing private functionalities, interfacing with existing applications, combining with other libraries), for the prototyping and implementation of innovative interactive artefacts (e.g. non-standard interaction techniques, data visualisations, UI toolkits, interactive applications).

This research is conducted by Thibault Raffaillac (thibault.raffaillac@inria.fr) and Stéphane Huot (stephane.huot@inria.fr), members of the Inria's team Loki (http://loki.lille.inria.fr/).

This survey takes about 20 minutes to complete. Please read this consent form carefully before proceeding.

Basis to take part in this survey

You must be over 18 years old.
You must have prior experience in developing interactive applications, preferably in a context of research and/or innovation.

Voluntary participation in the project

Your participation to this survey is entirely voluntary, and without any constraints or outside pressure. If you are lacking information needed to make your decision, or have any questions about the project or your rights as a participant, do not hesitate to ask for additional information from the contact person (thibault.raffaillac@inria.fr).

Withdrawal from the project at any time

You are free to terminate your participation to this survey at any time, without any justification, by closing this web page. In this event, the answers you already have responded will not be logged and their will be no trace of your participation in our data.

Anonymity and confidentiality

All the raw information collected from your participation is anonymous and only the members of the research project may have access to it. The data we are collecting is only your answers to the questions (checkboxes and text). We are not collecting any personal data (e.g. email, IP) and the survey system we are using does not allow to retrieve your identity while you are answering the survey or once you have submitted your answers.
Collected data will be used to illustrate the results of our work in scientific communications (articles, posters, oral presentations) or scientific mediation after it has been processed and analyzed. Thus, there will be no way to identify you as a participant. In any case, while processing the data, we will take care to anonymize any collected data that could be used to refer back to you as an individual in your answers.

Informed consent *

☐ I acknowledge that I have read and understood this consent form, and voluntarily consent to participate in this research project

Next page >

Main professional activity*

- Researcher
- Interaction designer
- Project manager
- Engineer
- Software developer
- Other

Other main professional activity

Which kind of institution/company do you work for? *

- University/School
- Public institution
- Startup
- Self-employed
- Small enterprise (< 50 employees)
- Medium enterprise (< 250 employees)
- Big enterprise (ÿ 250 employees)

How many people on average (excl. yourself) are working with you on a single project?*

- 0
- 1
- 2
- 3
- 4
- 5+

How do you evaluate your own expertise in writing interactive applications?*

- Basic knowledge
- Novice
- Intermediate
- Advanced
- Expert

How much of your time (professional or leisure) do you devote to programming? *

- 0%~20%
- 20%~40%
- 40%~60%
- 60%~80%
- 80%~100%

< Previous page Next page >

Appendix A. Study plans

In this section we ask you to evaluate the importance of various criteria when choosing a library for a project. Here we consider general-purpose frameworks (e.g. Qt, Cocoa, JavaFX, ReactJS) as well as research toolkits for specific needs (e.g. remote collaboration, wall-sized displays, low-power devices). Below each criterion you may give examples to illustrate your answer.

Which frameworks/toolkits do you use the most? (comma-separated)

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Tool reputation (timely and frequent bug fixes, online comparisons, recommendation from peers, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Developer reputation (number of developers, brand, other tools from the same developer(s)/company, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
User community (activity, size, companies using it, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Documentation quality (abundance, clarity, time to learn, examples, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Range of use cases supported (number, variety, relation to your context, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Quality of the API (coherency, simplicity, adequate paradigm for your needs, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Compatibility with other tools/libraries (bindings already available, extensibility, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Project constraints (already in use, expertise from colleagues, licensing terms, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Personal experience/familiarity with the library	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Technical efficiency (performance, latency, memory use, ...) *	○	○	○	○	○

Example criteria

Are there other important criteria we did not mention here? (comma-separated)

In this section we ask you to rate the severity of selected issues, that you might have encountered while using interaction libraries. In the table below you may select "Not met" if you never encountered a given issue. Otherwise, indicate how difficult it was to overcome.

Did you encounter any of these issues, and if so how much did they hinder your work at worst? *

	Not of	met, no trouble	Met, minor trouble	Met, medium trouble	Met, major trouble	Met, insurmountable trouble
Inadequate paradigm for this particular context *						
Problems scaling up (in speed/precision/frequency) *						
API too complex to use/understand *						
Documentation lacking context and examples *						
Significant effect/behaviour being undocumented *						
Bad compatibility between two libraries *						
Forbidden access to functions and data *						
Inconsistent behavior across versions/systems *						
Documentation requiring too much investment *						
Non-deterministic behavior*						
Lack of a functionality that would require pulling another library *						
Buggy implementation *						
Lack of configurability in API *						

< Previous page | Next page >

Appendix A. Study plans

In this section we consider how you addressed the various issues met in the previous section. We selected a number of coding techniques from previous interviews, and ask you to rate their prevalence in your work. For each technique there is an optional text entry, where you may give examples of the coding techniques you used.

	Never	Rarely	Sometimes	Very often	Always
Using an external mechanism to obtain and process data that is not exposed by an application					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Using accessible raw data to reconstruct/reinterpret a state that you do not have access to					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Aggregating multiple sources of interaction data (input, sensors, events), and fusing them into a single source					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reimplementing an existing widget/mechanism to gain more control over its appearance/behavior					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reimplementing low-level system components (e.g. driver) to improve their functionalities or better support specific hardware devices					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Introducing a different programming model, pattern or paradigm on top of the existing framework or toolkit					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Using a visual overlay to add custom functionality *					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reverse-engineering a closed tool or library to acquire understanding of its inner working*					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Modifying the environment of a tool rather than the tool itself to change its behavior*					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Purposely setting a parameter outside of its intended/expected range of values					

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reproducing a fake application to control a specific aspect					

Example techniques

< Previous page Submit