# Language and system support for interaction

Thibault Raffaillac

# Language and System Support for Interaction

**Thibault Raffaillac**
Inria, Lille, France
thibault.raffaillac@inria.fr

## Abstract

Interaction frameworks are the norm for prototyping, implementing and sharing user interfaces and interaction techniques. However, they often lack the flexibility to easily implement new kinds of interfaces and interaction techniques, since they were basically designed for implementing standard and normalized WIMP user interfaces. This forces programmers to rely on “hacking” in order to experiment with functional prototypes, and could drastically limit the range of scenarios where these prototypes will work. In my PhD, I study the interplay between people designing interaction techniques, and their software frameworks. My goal is to identify a number of fundamental features and requirements that programming languages and systems should support, in order to improve the flexibility of interaction frameworks for programming advanced interaction techniques.

## Author Keywords

User Interface Engineering; Software Prototyping; Hacking; Opportunistic Programming; Language Constructs

## ACM Classification Keywords

H.5.2 [User Interfaces]: Prototyping; D.2.2 [Design Tools and Techniques]: User interfaces; D.3.3 [Language Constructs and Features]

## Introduction

Prototyping new interaction techniques today is still a difficult task [6, 5]. Most user interfaces are programmed with common interaction frameworks and platforms (e.g. Cocoa, Qt, Android, HTML), which are thus the best candidates for introducing and popularizing new interaction techniques (e.g. menu systems, pointing methods, visualization techniques). Despite being often not designed for implementing non standard techniques, they benefit from a large base of users, are based on widely understood paradigms (MVC, listeners, events), and provide many useful facilities to work on a prototype and test it (templates, logging, build scripts).

However, these frameworks are made to design interfaces, rather than interaction per se [1]. Since they adopt interactive elements at their core, the widgets, new techniques are constrained to rely on widgets. These in turn have constraints, in their drawing area, and their interaction with mouse and keyboard. Moreover, software frameworks are large code bases, with steep learning curves to transition to their internal – sometimes ill-documented – APIs [8, 3]. In these conditions, "*hacking*" is a solution to make an idea become possible, although it could eventually reduce one's ambitions in order to get a satisfying result in time [4].

In my doctoral work, I defend the idea that the flexibility of interaction frameworks can be improved and their complexity reduced by: (i) identifying and extracting a set of fundamental features, or *bare essentials*, for expressing interaction instead of interfaces; and (ii) considering this set as part of the programming systems themselves, as $1^{st}$ class objects or constructs of the language. These are for example "receiving mouse input", "drawing things on screen", or "animating visual properties". This set should be simple enough to be reasonably acceptable by language designers, i.e. *no* APIs like OpenGL or SDL.

## Expressing animations

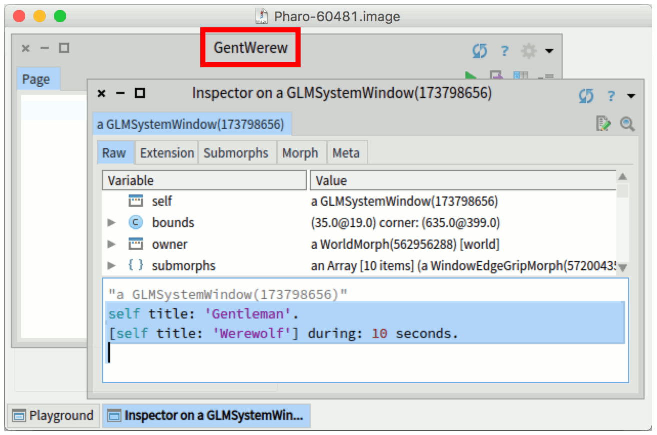
The first of these features I am addressing is to code animations, and more generally *things that occur in time.* Most frameworks with embedded animation capabilities restrict animations to the transitions of properties values in widgets, i.e. one has to create a widget to be able to use its animation capabilities. This is inconvenient when using multiple frameworks, like Qt and OpenGL, because the animation API of one framework may not work well with the others. This is also limiting when animating an arbitrary property or object that has not been designed for this purpose in the base framework.

During the first part of my PhD I introduced and prototyped a *delay operator* [9], which turns an instantaneous transition `myWidget.setProperty(value)` into a smooth one with animation, preserving the initial syntax for readability, and being integrated at the system level (i.e. for any framework):

```
myWidget.setProperty(value) during 3s
```

The prototype runs on the Pharo Smalltalk platform [2]. It works independently of any interaction frameworks on this platform, and does not have any set of predefined animatable properties. It relies on a getter-setter naming convention to retrieve the starting value for each animation. For the types of the properties, like numbers, colors and strings, it interpolates between initial and final values by relying on + and * operators, when available. Otherwise it expects an `interpolate` function to be provided, and fails otherwise.

I tested this system successfully with three GUI and visualization frameworks, for moving windows, changing background colors, replacing text labels in buttons and titles (see figure 1), and creating a particle effect.


Pharo-60481.image
GentWerew
Page
Inspector on a GLMSystemWindow(173798656)
a GLMSystemWindow(173798656)
Raw
Extension
Submorphs
Morph
Meta
Playground
Inspector on a GLMSystemWin...

| Variable | Value |
|---|---|
| self | a GLMSystemWindow(173798656) |
| bounds | (35.0@19.0) corner: (635.0@399.0) |
| owner | a WorldMorph(562956288) [world] |
| submorphs | an Array [10 items] (a WindowEdgeGripMorph(5720043 |

```
"a GLMSystemWindow(173798656)"
self title: 'Gentleman'.
[self title: 'Werewolf'] during: 10 seconds.
```

**Figure 1:** A Smalltalk Inspector open over a background window. The bottom code transitions its title from `'Gentleman'` to `'Werewolf'`. The snapshot was taken during this transition.

## Future work

The next step of my work is to validate the usefulness of the delay operator with a showcase of diverse examples, as well as to evaluate the gain for programmers (e.g. programming speed, legibility and understandability of the code) [7].

More generally, I am also focusing on interaction devices, exploring the raise of `Mouse`, `Keyboard`, `Display` objects to *first-class objects* [1]. While this is not a novel idea – Mozilla Firefox provides a `Screen` interface for interacting with the display, also Microsoft Visual Basic has a `Mouse` class – it has seen little use and deserves more attention given my overall objective in this thesis.

Finally, I am interested in how the design of frameworks can help their hacking, how people interact with the *low-level* functions, and how they relate code *dirtiness* to the amount of hacking they put into it. I already conducted a series of 8 interviews of people who are frequently prototyping very new and advanced interaction techniques, to gather the difficulties they faced while doing so. My goal is to classify the different aspects of software frameworks, into how they prevent or help programmers for exploring and prototyping alternative ideas. I am also interested in the “hacking” strategies people use to realize their ideas. As a knowledge contribution it should help the design of tools and languages to support the development of advanced interfaces and new interaction techniques.

## REFERENCES

1. Michel Beaudouin-Lafon. 2004. Designing Interaction, Not Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, USA, 15–22. `DOI:` `http://dx.doi.org/10.1145/989863.989865`

2. Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. `http://pharobyexample.org/,http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf`

3. Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 632–642. `DOI:` `http://dx.doi.org/10.1145/2568225.2568299`

4. B. Hartmann, S. Doorley, and S. R. Klemmer. 2008. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing* 7, 3 (July 2008), 46–54. `DOI:` `http://dx.doi.org/10.1109/MPRV.2008.54`

5. Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. DOI: http://dx.doi.org/10.1145/344949.344959

6. B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko. 2008. How Designers Design and Program Interactive Behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, Washington, DC, USA, 177–184. DOI: http://dx.doi.org/10.1109/VLHCC.2008.4639081

7. Dan R. Olsen, Jr. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 251–258. DOI: http://dx.doi.org/10.1145/1294211.1294256

8. Chris Parnin and Christoph Treude. 2011. Measuring API Documentation on the Web. In *Proceedings of the 2Nd International Workshop on Web 2.0 for Software Engineering (Web2SE '11)*. ACM, New York, NY, USA, 25–30. DOI: http://dx.doi.org/10.1145/1984701.1984706

9. Thibault Raffaillac, Stéphane Huot, and Stéphane Ducasse. 2017. Turning Function Calls into Animations. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '17)*. ACM, New York, NY, USA, 81–86. DOI: http://dx.doi.org/10.1145/3102113.3102134