Turning Function Calls Into Animations

Thibault Raffaillac, Stéphane Huot, Stéphane Ducasse

Turning Function Calls Into Animations

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

Stéphane Huot
Inria, Lille, France
stephane.huot@inria.fr

Stéphane Ducasse
Inria, Lille, France
stephane.ducasse@inria.fr

ABSTRACT

Animated transitions are an integral part of modern interaction frameworks. With the increasing number of animation scenarios, they have grown in range of animatable features. Yet not all transitions can be smoothed: programming systems limit the flexibility of frameworks for animating new things, and force them to expose low-level details to programmers. We present an ongoing work to provide system-wide animation of objects, by introducing a delay operator. This operator turns setter function calls into animations. It offers a coherent way to express animations across frameworks, and facilitates the animation of new properties.

CCS CONCEPTS

• **Software and its engineering** → **Graphical user interface languages**; • **Computing methodologies** → *Animation*; • **Human-centered computing** → *User interface programming*;

KEYWORDS

animation, programming constructs, expressing time, user interface design

ACM Reference format:
Thibault Raffaillac, Stéphane Huot, and Stéphane Ducasse. 2017. Turning Function Calls Into Animations. In *Proceedings of EICS '17, Lisbon, Portugal, June 26-29, 2017,* 6 pages.
https://doi.org/10.1145/3102113.3102134

1 INTRODUCTION

Since the development of computers, animations have been used in an increasingly wide range of scenarios such as: teaching of programming with visual environments [9, 25, 28]; transitions in Graphical User Interfaces and visualization systems for resizing windows or transitioning between views on data [12, 19, 27]; or the animation of virtual characters in video games by interpolation among motion-captured keyframes [6, 29]. Animations are considered useful in user interfaces to help following changes [26], and in visualizations to build a mental map of spatial information [2]. They can also convey meaning in data visualization [14], storytelling [18], as well as many other roles in user interfaces [7].

Interaction frameworks have been evolving over years to support a greater range of uses, by developing more flexible ways to animate elements in user interfaces. While early systems would animate a few properties, such as position or color, with different functions for each, modern systems contain too many to scale this way. In particular, CSS3 has 44 animatable properties[1], and Core Animation has 29[2]. To cope with this increasing numbers of animatable properties, most frameworks rely on naming strategies to refer to properties – like `"position"`, `"scale"`, `"color"` – instead of having one specific function for each. This improves flexibility for choosing animated properties at runtime, and reduces API size. It also gives an implicit contract that *any* property can animate, or at least one which would make sense to the programmer.

This flexibility has a price though. The animation of custom properties and types requires frameworks to provide an advanced API that exposes low-level details of their animation systems, including timers and threads. This results in larger animation APIs, and cumbersome syntaxes owing to the complex techniques often used to animate properties by names. It also creates a steep learning curve from basic to advanced API, which is likely to force programmers to stick to existing animatable properties whenever possible.

In this paper, we introduce a *delay operator* to express animations by turning setter functions calls into smooth transitions, as illustrated by the following pseudo-code:

```
object.setProperty(target) during 2s
```

We start by describing our structure of an animation, and the steps required to build it with a delay operator. Then we describe the implementation of a working prototype on the Pharo Smalltalk platform [4]. Finally, we review and compare six modern interaction frameworks, and discuss the limits of our system and future work on the topic.

EICS '17, June 26-29, 2017, Lisbon, Portugal

[1] https://www.w3.org/TR/css3-transitions/#animatable-css
[2] https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreAnimation_guide/AnimatableProperties/AnimatableProperties.html

2 DESCRIBING ANIMATIONS

In interactive computer systems, a transition from an initial to a final state is instantaneous: changing the on screen position of an object makes it disappear from its current position, and at the same time appear at its new one. This abrupt change can break our perception that one same object moved, instead of a new one appearing at another position. Animations smooth the transition in time, so that it looks continuous. Betrancourt and Tversky define it as "*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*" [3].

Animation of properties in interaction frameworks consists in replacing one instantaneous transition with many small updates of the same property, in quick sequence. These changes happen as often as possible but are bound by the display refresh rate, which is the frequency for painting each rendered frame on the screen. This frequency is usually of 60 Hz, and is occasionally slowed down when the rendering of a frame takes too long.

Each update starts by computing a relative time in the animation: `t = f(now)`, where `t = 0` at the beginning, and `t = 1` at the end. `t` does not necessarily grow uniformly in [0, 1]: it may accelerate at start, bounce back before stop, and even oscillate around target[3]. We call `f` a pacing function.

Then the value computed for each update is obtained with an interpolation function, `interpolate(origin,target,t)`, which returns `origin` at `t = 0` and `target` at `t = 1`. The function depends on the types of the values, i.e. *colors* and *positions* would not be interpolated with the same algorithm.

To describe the concept of an animated property, we rely on the 5 *high level aspects of animations* defined by Mirlacher et al. [21]. We define a *Smooth Transition* object as containing:

receiver which is the object to animate;

function signature is the name of the function setting the target property, and the types of its arguments;

key values defines the initial and final values for each of the arguments;

duration in seconds;

sampling object which fires events to notify *when* each update should happen, at the output frequency;

pacing function which adds temporal effects by varying animation speed;

interpolation functions for each argument, depending on their types;

running status allowing the transition to be paused, stopped, looped or reversed.

[3]For a complete list, see http://easings.net/

3 THE DELAY OPERATOR

Our delay operator binds a duration to a function call, creating a Smooth Transition object: `<functionCall> during <duration>`. It proceeds in four steps, as shown in Figure 1.

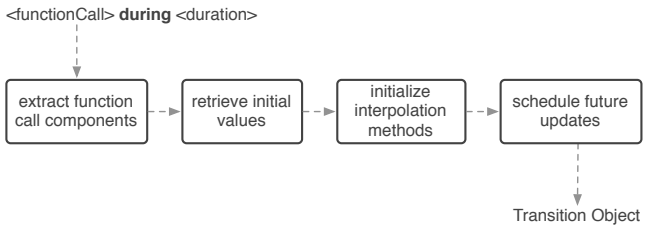

Figure 1: The four steps to translate a function call into a Transition Object

The first step, *extract function call components*, extracts four elements out of the function call: its receiver, its name, the ending values, and their types. It also cancels immediate execution of the function. This step effectively *reifies* the call, because we look at data, and we extract its characteristics. This requires support for introspection of code in the programming language, i.e. the ability to examine its properties instead of executing it (more details are given in the Implementation section).

To *retrieve the initial values*, we need a mechanism to obtain the current values of a property targetted by a setter function. Fortunately, many frameworks adopt conventions in how setter and getter functions are named. Qt [24], for example, matches most `setProperty(...)` setter functions with a `property()` getter function. On Smalltalk platforms, the convention is to name getters and setters the same, like `object property` and `object property: <value>`. Naming conventions are important: without them compilers would have to associate a stored value to each function flagged as a setter. For multiple-argument functions, getters can return in value holders passed as arguments. Once the getter function is inferred, it is called dynamically to retrieve the initial values. For languages without dynamic dispatch like C, a clever naming convention like `property() => get_property()` would allow substitution at preprocessing. Otherwise this requires explicit support from the compiler.

The third step, *initialize interpolation methods*, provides a default interpolation algorithm for each argument, depending on their type. For integers and real numbers we use:

$$interpolate(initial, final, t) = initial \times (1 - t) + final \times t$$

Composite types such as positions and colors interpolate their components separately as numbers. However, there are types for which this kind of interpolation would not make sense, such as arrays or strings, as in Figure 2. For these cases, programmers should be able to provide their own alternate

interpolator. This step also benefits from polymorphism support, to expect an `interpolate` method in each argument, if an alternate one has not been given.

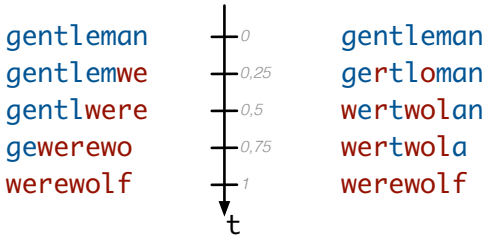

Figure 2: Two alternative string interpolators. On the left, the new string comes from the right; on the right, in-place interpolation is using random indices.

For *scheduling the future updates*, interaction with users requires the lowest latency in a feedback loop, i.e. the delay between each update and its result on screen. This matters in contexts such as interaction with touchscreens [10], performance on digital musical instruments [17], and online gaming [8]. Consequently, each animation update should be scheduled as late as possible, and before any depending events, as shown in Figure 3. Here, support from the system would take the form of a callback mechanism to execute code right before rendering, like W3C's `requestAnimationFrame` [13]. The delay operator automatically sets the running status to start the animation once it is created.

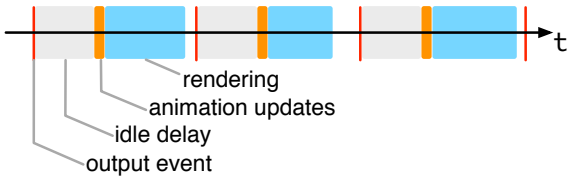

Figure 3: Representation of the moment to insert animation updates. An output event is generated by the operating system, then the rendering system pauses in order to minimize the time left after rendering and before the next output tick. Animation code is inserted before each rendering execution.

4 IMPLEMENTATION

Our proof-of-concept implementation of the delay operator has been done in the Smalltalk language and tested with three interaction frameworks. Smalltalk is object oriented at its core, and uses messages between objects instead of functions: the code `myWidget color: col1` sends the message `color:` with argument `col1` to `myWidget`. Messages are dispatched dynamically, i.e. the code to execute is selected at run-time. Our prototype expresses animations with:

```
[object property: target] during: 2 seconds
```

Our extension is embodied in the `during:` message. The square brackets around the setter message create a block closure, allowing us to inspect the code inside without executing it. We resolve the four steps described in the previous section as follows:

- Extraction of the message components is done by parsing its byte code at runtime, relying on the introspection features of the platform;
- To retrieve an initial value, we remove the colon from `property:` to infer a getter message. This message is then sent to `object`, and the return value is the initial value. Note that this mechanism does not allow animating functions with multiple arguments, because the getter message returns only one value;
- Our default interpolator relies on polymorphism, by executing `(origin * (1 - t)) + (target * t)`, which sends `*` and `+` messages to the starting and ending values. However, developpers can also provide alternate interpolation code per transition object. Failure to provide any of these invokes the debugger, which is a standard error mechanism in Smalltalk;
- Since we do not have access to system-wide events for display, future updates are registered using the callback mechanism of one of the host frameworks. This limitation is discussed further in this section.

Preliminary tests

We tested this animation kernel with the three most popular interaction frameworks for the Pharo platform: (i) Morphic [20], a graphical interface initially built for the Self language, then later ported to Smalltalk for Squeak, and now available in Pharo; (ii) Bloc [23], that aims at being the successor for Morphic by supporting more input devices, and using vector graphics to cover a wider range of pixel densities; and (iii) Roassal [11], a visualization framework with a large library of templates and a Domain Specific Language designed to express interactive visualizations with little code.

The biggest challenge we faced is related to *when* we update animations. In order to carry out the schedule in Figure 3 with existing interaction frameworks, we should insert the updates *before* their rendering code, if they allow so.

Morphic provides a callback, `World defer: <closure>`, to execute a block of code before the next rendering pass. We used it to schedule our animation updates: Morphic is so bound to the system, that every other framework depends on it thus allowing our extension to work for all of them. Although Bloc also provides a similar callback: `BlUniverse defer: <closure>`, we did not want to patch our animation system for every new framework and relied exclusively on

Morphic. This raises the problem of actual framework independence: *our system is not actually independent*, it works for other frameworks because they depend on Morphic too.

Ultimately this should not be the case, and a callback for animation should be the responsibility of the *language* or *standard library*. This would allow third-party systems like ours to function independently of any frameworks.

By using the delay operator with Morphic, we were able to animate changes in position, background color, border style, and title of a window (see Figure 4). We also managed to change these attributes on existing buttons inside a window. To allow a default interpolation of strings as on the left of Figure 2, we implemented + as concatenation on strings, and * as extraction of a portion of a string.

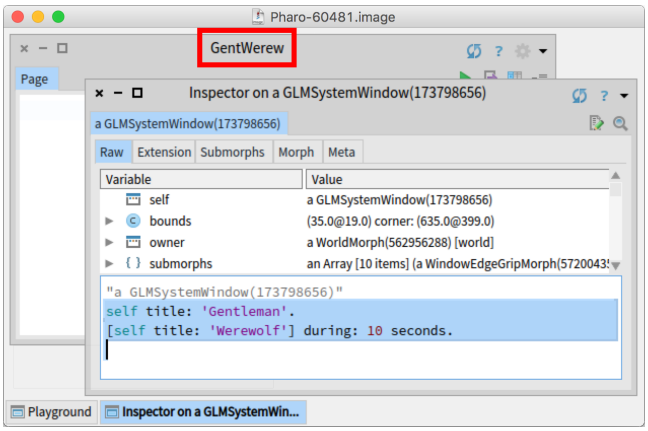

Figure 4: An Inspector open on a background window. The bottom code transitions its title from `'Gentleman'` to `'Werewolf'`. The image is taken during this transition.

However, not all properties would transition properly. The default text inside a text entry field did not allow edition once it was displayed, despite providing a getter and a setter. This was a limit of this widget: animation through dynamic calls cares only about calling methods, no matter their effect.

Our system worked with Bloc and Roassal too, although sometimes Roassal did not follow the getter-setter convention. For position, widgets would have a getter `position` and a setter `translateTo:`. In such a case, we had to fix Roassal with a new `position:` message. For color, the names `color` and `color:` matched, but the getter would not return a `Color` object. In this case, we had to fix our system to always convert the target value to the type returned by the getter function. These issues show the importance of a naming convention for getters and setters: it allows our system to work without storing a correspondence table.

Another practical issue appeared in Roassal, as updating a widget would not automatically flag its containing view for refresh. This made all animations invisible until we moved or resized the view. Although we considered it as a flaw in this framework, we resolved it by *manually refreshing the view* along each animation, with a second one in parallel. However, with further work we would fix the setters on widgets in Roassal, to call their parent view for refresh.

Finally, we added support for the `during:` message on groups of messages:

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

This example returns two Smooth Transition objects in an Array object, which we modified to forward running operations – `start`, `stop`, `pause` – to its elements. While this extension is out of the scope of this paper, it required additional support for reifying several function calls at once.

5 RELATED WORK

For this work we reviewed the animation APIs of six popular interaction frameworks. Qt [24] is a mainstream interaction framework for `C++` available on desktop and mobile platforms. Apple's Core Animation [1] is the standard recommended framework for the OSX and iOS platforms. JavaFX 8 [22] is the official successor of Swing, and Android [15] is another popular Java framework for mobile devices. We also selected D3.js [5] for its popularity as a web visualization tool, and GSAP [16] for having the most flexible choice of animatable properties. We were interested in how *flexible* and *compact* each of these frameworks were for expressing animations. Our intent is to observe their different strategies for animating properties dynamically.

The comparison is displayed in Table 1. The dimensions considered were:

- *animatable properties:* the supported properties, with emphasis on the techniques used to support new ones;
- *animatable types:* the supported types, and analogously the techniques used to support custom types;
- *sample syntax:* code snippets implementing a minimal example to assign `property` to `target` during 2 seconds.

Restrictions on properties come in three sorts: the need to inherit a particular class or interface (Qt 5, Core Animation, JavaFX 8), the naming of property accessors (Android), and the avoidance of special keywords (GSAP). For types we identify two groups of frameworks: those with a bounded set (Qt 5, Core Animation), and those requiring a custom interpolation function. It is often provided as a *functor* object, which is an object with a single method. In addition, D3 is notable for automatically interpolating the fields of unknown objects, thanks to Javascript's ability to list their properties.

As for the syntax, some frameworks have alternate forms, the more comprehensive being presented here. Almost all build animation objects using a `"property"` string. GSAP's

Framework	Animatable properties	Animatable types	Sample syntax

Qt 5 (C++)	every property providing a setter function (*the owner object must inherit* `QObject`)	integers, floats, `QLine`, `QPoint`, `QSize`, `QRect` and `QColor` (*additional types must be supported by* `QMetaType`)	`QPropertyAnimation a(object, "property");`
			`a.setDuration(2000);`
			`a.setEndValue(target);`
			`a.start();`

Core Animation (Swift)	29 default properties (*objects must inherit* `CALayer` *and custom properties must have getter and setter functions*)	integers, doubles, `CGRect`, `CGPoint`, `CGSize`, `CGAffineTransform`, `CATransform3D`, `CGColor` and `CGImage` (*no support for other animatable types*)	`let a = CABasicAnimation(keyPath:"property")`
			`a.toValue = target`
			`a.duration = 2.0f`
			`object.addAnimation(a, forKey:"property")`

D3.js (Javascript)	properties among `attr` and `style` from DOM elements (*restricted to objects from the DOM*)	numbers, colors (in multiple spaces), dates, numbers embedded in strings, arrays, dictionaries, 2D transforms (*other types can animate by providing functor objects*)	`object.transition()`
			`  .duration(2000)`
			`  .attr("property", target);`

JavaFX 8 (Java)	all properties implementing the `WritableValue<T>` interface (*objects need only store such properties*)	integers, floats, `Color` objects (*custom types must implement the* `Interpolatable` *interface*)	`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 `set<PropName>()` setter function (*no restriction on owner object*)	`int`, `float`, colors (*custom types must provide an interpolation functor*)	`ObjectAnimator a = ObjectAnimator`
			`  .ofInt(object, "property", target);`
			`a.setDuration(2000);`
			`a.start();`

GSAP (Javascript)	any property except a set of reserved names for passing options (*no restrictions on objects*)	numbers, numbers in strings (*other types can animate with a functor object*)	`TweenLite.to(object, 2, {property: target});`

Table 1: Comparison of animation features in six frameworks.

`TweenLite` retrieves the property name by parsing the fields of its 3rd argument. D3's approach is to append `.transition()` after the receiver, a proxy object intercepts `style` and `attr` function calls, to smooth them automatically. These many syntaxes pose the question of which representations are easier to read, faster to code with, and lead to less bugs. In particular, we are concerned about the effect of maintaining function call syntax, as with D3's proxy object.

6 CONCLUSION

In Object-Oriented Programming, objects encapsulate their own state and expose it with interfaces. If they have coordinates for a position, then they will likely expose functions `getPosition()` and `setPosition(Point)`. Animation through dynamic calls makes clever use of these, by obtaining an `origin` value with the former, and sending small variations between `origin` and `target` to the latter. Objects do not have to know *how* to animate their positions: with a series of small updates we fake a smooth movement towards its destination.

From this observation, we studied the concept of animation for all objects in the system, independently of any frameworks. In addition, we aimed for a syntax reusing as much as possible the existing lexical elements. This resulted in a delay operator appended to function calls:

```
object.setProperty(target) during 2s
```

We have presented a process and a working implementation to create and initialize a transition object from this syntax, resulting in two recommendations for interactive systems: (i) to provide a system-wide signaling mechanism for display refresh events; and (ii) to promote naming conventions for getter and setter functions.

In the future, this prototype system should contribute to measuring the effect of animation syntax on programmers' ease in prototyping. More generally, we are interested by the effect of maintaining the consistency of function call syntax.

We also aim to extend this approach to non-visual transitions such as *audio* and *haptic* feedback or microcontrollers. Indeed, they rely on smooth transitions too: the cross-fading

between two audio sources, the acceleration of a DC motor, or the smooth variation of light intensity on a connected LED. Eventually, this approach may contribute to the more general problem of programming time-based interactive systems.

REFERENCES

[1] Apple Inc. 2006. About Core Animation. (2006). http://developer.apple.com/documentation/Cocoa/Conceptual/CoreAnimation_guide/

[2] B. B. Bederson and A. Boltman. 1999. Does Animation Help Users Build Mental Maps of Spatial Information?. In 1999 IEEE Symposium on Information Visualization, 1999. (Info Vis '99) Proceedings. 28–35. https://doi.org/10.1109/INFVIS.1999.801854

[3] M. Bétrancourt and B. Tversky. 2000. Effect of Computer Animation on Users' Performance: A Review. Le Travail Humain: A Bilingual and Multi-Disciplinary Journal in Human Factors 63, 4 (2000), 311–329.

[4] 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

[5] M. Bostock, V. Ogievetsky, and J. Heer. 2011. D3 Data-Driven Documents. IEEE Transactions on Visualization and Computer Graphics 17, 12 (Dec. 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

[6] Lynne Shapiro Brotman and Arun N. Netravali. 1988. Motion Interpolation by Optimal Control. In Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88). ACM, New York, NY, USA, 309–315. https://doi.org/10.1145/54852.378531

[7] Fanny Chevalier, Nathalie Henry Riche, Catherine Plaisant, Amira Chalbi, and Christophe Hurter. 2016. Animations 25 Years Later: New Roles and Opportunities. In Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI '16). ACM, New York, NY, USA, 280–287. https://doi.org/10.1145/2909132.2909255

[8] Mark Claypool and Kajal Claypool. 2006. Latency and Player Actions in Online Games. Commun. ACM 49, 11 (Nov. 2006), 40–45. https://doi.org/10.1145/1167838.1167860

[9] Wanda P Dann, Stephen Cooper, and Randy Pausch. 2011. Learning to Program with Alice (w/CD ROM). Prentice Hall Press.

[10] Jonathan Deber, Ricardo Jota, Clifton Forlines, and Daniel Wigdor. 2015. How Much Faster Is Fast Enough?: User Perception of Latency & Latency Improvements in Direct and Indirect Touch. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15). ACM, New York, NY, USA, 1827–1836. https://doi.org/10.1145/2702123.2702300

[11] Mathieu Dehouck, Usman Bhatti, Alexandre Bergel, and Stéphane Ducasse. 2013. Pragmatic Visualizations for Roassal: a Florilegium. In International Workshop on Smalltalk Technologies. http://rmod.inria.fr/archives/papers/Deho13a-IWST2013-AlgoRoassal.pdf

[12] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. Gliimpse: Animating from Markup Code to Rendered Documents and Vice Versa. In Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11). ACM, New York, NY, USA, 257–262. https://doi.org/10.1145/2047196.2047229

[13] Steve Faulkner, Arron Eicholz, Travis Leithead, and Alex Danilo. 2016. HTML 5.1: 7.8 Animation Frames. (2016). https://www.w3.org/TR/html51/webappapis.html#animation-frames

[14] Cleotilde Gonzalez. 1996. Does Animation in User Interfaces Improve Decision Making?. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '96). ACM, New York, NY, USA, 27–34. https://doi.org/10.1145/238386.238396

[15] Google. 2008. Android. (2008). https://www.android.com/

[16] GreenSock. 2014. GSAP, the standard for JavaScript HTML5 animation. (2014). https://greensock.com/gsap

[17] Robert H. Jack, Tony Stockman, and Andrew McPherson. 2016. Effect of Latency on Performer Interaction and Subjective Quality Assessment of a Digital Musical Instrument. In Proceedings of the Audio Mostly 2016 (AM '16). ACM, New York, NY, USA, 116–123. https://doi.org/10.1145/2986416.2986428

[18] Kevin Kennedy and Robert E. Mercer. 2002. Planning Animation Cinematography and Shot Structure to Communicate Theme and Mood. In Proceedings of the 2Nd International Symposium on Smart Graphics (SMARTGRAPH '02). ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/569005.569006

[19] Christian Klein and Benjamin B. Bederson. 2005. Benefits of Animated Scrolling. In CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05). ACM, New York, NY, USA, 1965–1968. https://doi.org/10.1145/1056808.1057068

[20] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95). ACM, New York, NY, USA, 21–28. https://doi.org/10.1145/215585.215636

[21] Thomas Mirlacher, Philippe Palanque, and Regina Bernhaupt. 2012. Engineering Animations in User Interfaces. In Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12). ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/2305484.2305504

[22] Oracle Corp. 2008. Client Technologies: Java Platform, Standard Edition (Java SE) 8 Release 8. (2008). http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

[23] Alain Plantec. 2015. Bloc: a new Morphic framework. (2015). http://sdmeta.gforge.inria.fr/YouTubeVideos/PharoPreview/BlocSlides-ESUG2015.pdf

[24] Qt Development Frameworks/Digia. 2015. The future is written with Qt: Cross-platform software development for embedded & desktop. (2015). https://www.qt.io/

[25] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (2009), 60–67.

[26] Céline Schlienger, Stéphane Conversy, Stéphane Chatty, Magali Anquetil, and Christophe Mertz. 2007. Improving Users' Comprehension of Changes with Animation and Sound: An Empirical Assessment. Springer Berlin Heidelberg, Berlin, Heidelberg, 207–220. https://doi.org/10.1007/978-3-540-74796-3_20

[27] Maruthappan Shanmugasundaram, Pourang Irani, and Carl Gutwin. 2007. Can Smooth View Transitions Facilitate Perceptual Constancy in Node-Link Diagrams?. In Proceedings of Graphics Interface 2007 (GI '07). ACM, New York, NY, USA, 71–78. https://doi.org/10.1145/1268517.1268531

[28] John Stasko, Albert Badre, and Clayton Lewis. 1993. Do Algorithm Animations Assist Learning?: An Empirical Study and Analysis. In Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93). ACM, New York, NY, USA, 61–66. https://doi.org/10.1145/169059.169078

[29] D. J. Wiley and J. K. Hahn. 1997. Interpolation Synthesis of Articulated Figure Motion. IEEE Computer Graphics and Applications 17, 6 (Nov. 1997), 39–45. https://doi.org/10.1109/38.626968