An empirical study of programming paradigms for animation

Jan-Peter Krämer[1,2], Michael Hennings[1], Joel Brandt[2], Jan Borchers[1]
[1] RWTH Aachen University
{kraemer, hennings, borchers}@cs.rwth-aachen.de
[2] Creative Technologies Lab, Adobe Research
joel.brandt@adobe.com

ABSTRACT

Animations are an essential part of many modern user interfaces. They are often defined programmatically, which allows for parametrization and reuse. Two programming paradigms to define animations are common: Procedural animation programming allows the developer to make explicit updates to object properties at each frame, allowing maximum control over behavior. Declarative animation programming allows the developer to specify keyframes, i.e., the value of an object's property at a given point in time. All frames between two keyframes are automatically interpolated by the animation library.

In this paper, we investigate how these common programming paradigms differ in terms of developers' productivity. In a controlled laboratory study, we asked developers to implement a set of simple animations using both paradigms. We found that developers can implement a given behavior faster using declarative animation programming, but the abstraction introduced by automatically creating the animation through keyframe interpolation left participants with unexpected behavior for some tasks.

CCS Concepts

•Software and its engineering → Software development techniques; •Human-centered computing → *Empirical studies in HCI;*

Keywords

Animations; programming paradigms; empirical studies

1. INTRODUCTION

Animations are used in many contexts [4]. For example, they are added to user interfaces to provide feedback and convey a sense of plausability [3], and they are used in scientific visualizations to add another dimension to the display of data. More broadly, animations frequently appear in digital artifacts, hence, software developers with various backgrounds are often faced with the task of creating animations.

Many animations can be abstracted as follows [14]: A scene contains a number of objects and each object has a set of animatable properties, such as position or size. The animation defines how and when each of these properties changes over time.

Property changes can be obtained from various sources, e.g., from the real world through motion capturing or from a physics simulation. In this paper, we focus on *artistic* animations [14], where the animator defines all property changes as part of an artistic process.

Very broadly, two kinds of tools can be used to create animations: graphical tools, such as Adobe After Effects[1], and textual programming languages. Graphical tools usually allow users to animate a fixed set of objects in a single scene and excel at previsualizing how the finished animation will appear. In many scenarios, it makes sense to create animations programmatically, because this allows for modularization and parametrization of behavior and, hence, for reuse. This allows users to apply the same animation to multiple user interface elements or to provide a character with realistic behaviors, such as breathing, that happen continuously but adapt automatically to the character's acting.

Visual programming languages can combine the benefits of a graphical tool with the benefits of defining animations programmatically [2]. Resnick et al. [15] demonstrated that visual programming languages for animation programing are approachable enough to teach programming to children. Animations are often created as part of a bigger coding task that is performed using a textual language, e.g., implementing an application. In these cases, developers often continue to use the language they are already using to avoid bridging between different languages and context switching between different development environments. Even if both a graphical tool and a textual programming language are used together, the animation still needs to be controlled based on the application state, which is implemented in a textual langauge. Hence, creating and controlling animations using a textual language remains a common and relevant task.

In the existing libraries to support this task, we found two prevalent paradigms: In the *procedural* paradigm, a developer updates the properties of each object in an update method that is called for each frame of the animation. In the *declarative* paradigm, a developer declares the object's behavior using keyframes. Each keyframe defines the value of an object's parameter at a given point in time. An anima-

[1] http://www.adobe.com/products/aftereffects.html

CHASE'16, May 16 2016, Austin, TX, USA

ISBN 978-1-4503-4155-4/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897586.2897597

tion library is then responsible for interpolating the property values for all frames between two keyframes. The following code implements a ball that moves up and down using a procedural API (left) and using a declarative API (right):

```
function render(time) {
  var loc = time % 300;
  if (loc > 150) {
    loc = 150-(loc % 150);
  }
  ball.top = loc;
}

window
.requestAnimationFrame(render);
```

```
var actor = animator.addActor({
  context: ball
});

actor.keyframe(0, {top: 0});
actor.keyframe(150, {top: 150});
actor.keyframe(300, {top: 0});

animator.play();
```

Our long term goal is to provide better support for creating artistic animations using textual programing langauges. First, we want to understand how developers utilize current animation libraries, what problems occur, and whether or not one of the widespread programming paradigms is superior in allowing developers to create animations quickly. To this end, we present a between-groups laboratory study comparing a procedural and a declarative animation library.

2. EXPERIMENTAL SETUP

We performed a laboratory experiment with two conditions: *declarative* and *procedural*. Participants had to solve five tasks (Task A-E) once using a declarative API and once using a procedural API. Half of all participants started in the procedural condition, the other half started with the declarative condition. Assignment of participants to starting condition was randomized for each task.

In our study, developers had to use Javascript to animate HTML elements. In the procedural condition, developers used Javascript's native animation frames, which allow to provide a function that updates the properties of all animated objects in each frame. In the declarative condition developers used Rekapi[2], a keyframe-based animation library for JavaScript. We opted to use Rekapi instead of CSS animations, a widespread way to declaratively animate HTML elements, because Rekapi uses Javascript syntax, making it more comparable to the procedural condition. Both APIs we selected are only one example of a procedural and declarative animation libray. We designed the tasks in our study to use only the most fundamental features of each library. We expect these features to exist in virtually all other libraries following the same paradigms. The API documentation necessary to solve the study tasks was provided in the form of basic cheat-sheets [9] (following the guidelines by Mayer [12]).

The tasks [8] (see Figure 1) were designed in a set of pilot studies to make sure they could be solved in both conditions within two hours. When time ran out, later tasks were skipped.

Task A Animate a ball that continuously bounces up and down on a black bar. The ball should not deform when it hits the black bar and move at a constant speed between the upper and lower turing point.

Task B Add easing to the ball's movement to make the animation more realistic. In Rekapi, a keyframe can be configured to use a library-provided easing function. To allow a fair comparison, we provided helper functions for the procedural condition as well.

[2]http://rekapi.com/

Task C Duplicate the bouncing animation for a second ball that bounces next to the first one but with a delay.

Task D Implement a single bouncing ball that deforms when it hits the ground.

Task E Implement a pendulum-like movement in which a ball moves left and right while falling down. The ball should stop at the lowest position.

For all tasks we provided a scaffold containing all required HTML and CSS code and all JavaScript code except for the animation itself. For Task B and C this scaffold included a working solution of the previous task. The tasks we designed are representative of many common animations in user interfaces that are created by changing one or only few properties of an object over time. In the popular presentation software Keynote[3], 40% of all animations to reveal an object can be recreated by changing no more than three properties of the revealed object over time.

Participants had to work using the Brackets[4] editor. We added the following features to Brackets to support the study procedure: When starting a task, Brackets opened all required files and the task description as a PDF document. Participants had read-only access to the code written in previous tasks. A play button allowed participants to run their animation in the Google Chrome[5] browser that also provided a debugger. Participants could use Brackets to hand in their solution at any point. This caused all open files to be saved and closed and the next task to be loaded.

Participants used a 15-inch laptop computer, optionally with an external mouse and keyboard. We recorded the participant's screen, all key strokes and scroll events in the editor, and the timestamps of each manual code execution.

3. RESULTS AND DISCUSSION

A total of 14 developers participated in the study, all of whom were students from our university majoring in computer science or a similar field. The participants were on average 24.1 years old (SD = 3.4) and reported to develop software on average 14.3 (SD = 7.6) hours per week. Six participants reported to rarely or never have created animations before, while three participants reported to create animations regularly. Two participants reported to have rudimentary knowledge about Rekapi. Participants were informed that we wanted to compare two APIs for animation programming but they did not know our research hypotheses.

As tasks were skipped when time ran out, fewer participants worked on later tasks.

To compare both conditions, we analyzed three aspects: user preference, performance, and changes in strategies.

Only one participant reported to prefer the procedural condition, compared to seven developers prefering the declarative condition. The remaining participants would have liked a combination of both approaches, because they believed that the expressiveness of a declarative API was limited. Several participants suggested to use keyframes to define a timespan during which properties were not interpolated but changed manually in a callback for each frame, as

[3]http://www.apple.com/mac/keynote/
[4]http://brackets.io
[5]http://www.google.com/chrome

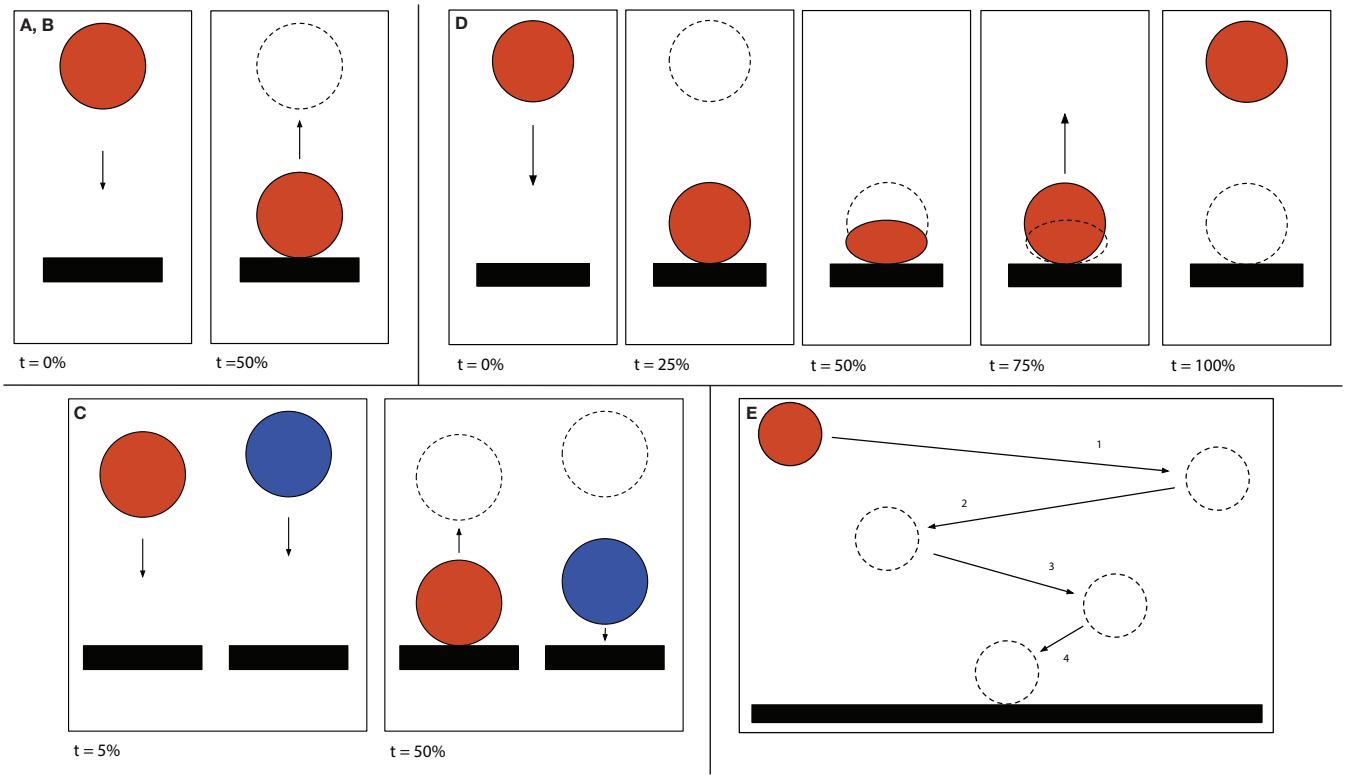

Figure 1: An illustration of the animation participants had to implement in each task. Tasks A and B only differ by the easing function used to transition betwen the upper and lower positions.

in the procedural condition. As the expressiveness of both libraries was sufficient to solve all tasks, this indicates that fully understanding how to utilize a declarative animation library can be difficult.

We analyzed task completion times only for Tasks A-D (only two participants completed Task E), and only for participants who solved a task in both conditions. Participants solved tasks 2.4 times faster on average in the declarative condition ($F(1, 42.4) = 14.2$, $p = 0.005$) [10]. We found no learning effects caused by the within-groups study design. Only in Task C, task completion times are comparable in both conditions. This task turned out to be challenging in the declarative condition: Because the animation for the added ball follows the first one with a delay, the last key frame for the new ball is added after the last keyframe for the first ball. When the complete animation loops, it restarts after both balls have moved back to the top. This causes both balls to pause for some time. This observation indicates a more general problem: The abstraction introduced in the declarative API can lead to unexpected behavior which, in turn, caused participants to want to go back to the procedural, less abstracted model.

We formed two hypothesis about how a change in developers' behavior can explain the differences in task completion times: (1) Animation creation tasks are known to require many manual edit-test-edit cycles [1], hence, the number of manual tests needed to arrive at a correct solution might be lower in the declarative condition. (2) A declarative animation library might provide a more suitable abstraction of the animation, leading to a smaller diversity in solutions.

We first tested hypothesis 1: In the procedural condition, participants performed 15.0 tests on average ($SD = 14.4$), while those in the declarative condition only performed 7.0 ($SD = 6.9$). This difference is significant ($F(1, 60.5) = 10.1$, $p = 0.002$) [10].

The observed difference in number of tests performed is caused by a large difference for Tasks A and B, while the number of tests while working on Tasks C and D is similar in both conditions. Task C has been identified to be difficult to solve in the declarative condition before, but Task D posed a different challenge: While compressing the ball by decreasing its height, the offset from the top needs to be adjusted accordingly to keep it fixed on the black bar. Hence, two different but dependent properties needed to be animated concurrently. The declarative programming model seemed to offer few helpful abstractions for this case. Overall, we found that developers can predict the behavior of declaratively defined animations more easily compared to the procedural conditon, but only if one or more properties change independently.

To analyze hypothesis 2 we compared the diversity of solutions in both conditions. We only found solutions in the declarative condition to be more consistent than those in the procedural condition for Task A: Nearly all participants in the Rekapi condition ended up with the same solution. Only one participant calculated individual keyframes for all intermediate steps, again indicating that understanding how to utilize a declarative animation library can be difficult for inexperienced users. In the procedural condition, solutions were more diverse: Half of the participants calculated the

ball's offset from the top as a function of elapsed time, and the other half used fixed offsets to move the ball from the previous position. Even though these solutions yield similar results visually, only the first guarantees a predefined speed of the animation, whereas for the latter the speed depends on the system refresh rate.

For the other tasks, solutions in the procedural condition were often more or similarly consistent compared to those in the declarative condition. For example, in Task B nearly all participants in both conditions ended up using the provided `easeInOut` function. While the solutions were similar, participants were more uncomfortable adding the easing methods into procedural code compared to configuring a keyframe to use easing in the declarative condition. Consequently, a lower number of participants found the correct solution at all in the procedural condtion. We suspect that developers did not fully comprehended the procedural implementation of Task A. This caused those participants to be hesitant to change the existing code, or to opportunistically perform and test changes, which is a common but ineffective strategy that aims to avoid understanding existing code [5].

In Task D, we again find surprisingly homogeneous solutions. In the procedural condition, most participants started with code similar to Task B and expanded it to move the ball further downwards while decreasing its height once the ball passed a certain threshold. Only one participant in the procedural condition diverged from this solution and implemented a state machine. This effectively implements a structure that was similar to the solution in the declarative condition. Once again, even though the diversity in successfull strategies is lower than expected, the percentage of participants finding a successful strategy is higher in the declarative condition.

In Task C, we expected developers to make use of the modularization and parametrization possible with programmatic definition of animations. Instead, all participants in both conditions duplicated the existing animation code to animate the second ball. Most of their time was spent altering the copied code, resulting in highly redundant and hard to debug code. We believed that the possibility to reuse code to be one of the key benefits of defining animations programmatically. Copying and modifying code instead of using modularization is a common behavior of developers who have not fully understood the existing code [5].

4. CONCLUSION AND FUTURE WORK

We found that developers can often create animations implemented declaratively faster than those implemented procedurally, and developers needed less manual tests to author declarativly implemented animations. This is likely caused by the declarative framework providing guidance through abstraction and a rigid structure. This abstraction, however, is only effective while multiple parameters animated in parallel are independent, and it can lead to side effects that are unexpected for inexperienced programmers. Both animation programming paradigms we studied do not promote modularization appropriately.

In the future, we want to explore two directions to support developers in creating animations: First, we want to study the effect of continuous execution [11, 7] on animation programming. In particular, we want to explore which visualizations can be used in such a tool to show side effects of changes and minimize the need to watch an animation repeatedly to fine-tune parameters. Live conding could also help visualizing the effect of animating multiple dependent parameters at the same time. Second, we want to propose a declarative library that in addition to keyframes allows to define temporal and spatial relationships between objects in the form of constraints [6, 13]. For example, Task C could be solved by defining that the second ball's x-coordinate is always 200px right of the first ball (spatial relationship), and the second ball's y-coordinate is always the first ball's y-coordinate 500ms ago (temporal relationship). We also want to explore if the benefits of declarative APIs we found for animation programing also apply to other use cases.

5. REFERENCES

[1] R. Brinkmann. *The Art and Science of Digital Compositing*. Techniques for Visual Effects, Animation and Motion Graphics. Morgan Kaufmann Publishers/Elsevier, Amsterdam, May 2008.

[2] P. Carlson, M. Burnett, and J. Cadiz. A seamless integration of algorithm animation into a visual programming language. In *Proc. AVI '96*, New York, NY, USA, 1996. ACM Press.

[3] B.-W. Chang and D. Ungar. Animation: from cartoons to the user interface. In *Proc. UIST '93*, New York, NY, USA, 1993. ACM Press.

[4] A. Dahotre, Y. Zhang, and C. Scaffidi. A qualitative study of animation programming in the wild. In *Proc. ESEM '10*, New York, NY, USA, 2010. ACM Press.

[5] F. Détienne. *Software Design—cognitive Aspects*. Springer-Verlag Inc., New York, NY, USA, 2002.

[6] R. A. Duisberg. Animation Using Temporal Constraints: An Overview of the Animus System. *Human–Computer Interaction*, 3(3):275–307, Nov. 2009.

[7] J. Edwards. Example centric programming. *SIGPLAN Notices*, 39(12):84, Dec. 2004.

[8] M. Hennings, J.-P. Krämer, J. Brandt, and J. Borchers. Programmatic animations - user study tasks, Jan. 2016, DOI: 10.5281/zenodo.45262.

[9] M. Hennings, J.-P. Krämer, J. Brandt, and J. Borchers. Programmatic animations - cheat sheets, Jan. 2016, DOI: 10.5281/zenodo.45263.

[10] J.-P. Krämer, M. Hennings, J. Brandt, and J. Borchers. Programmatic animations - performance data, Jan. 2016, DOI: 10.5281/zenodo.45273.

[11] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers. How live coding affects developers' coding behavior. In *Proc. VL/HCC '14*. IEEE.

[12] R. E. Mayer. Multimedia learning: Are we asking the right questions. *Educational Psychologist*, 32(1):1–19, 1997.

[13] S. Oney, B. Myers, and J. Brandt. ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States. In *Proc. UIST '12*, pages 229–238, New York, NY, USA, 2012. ACM.

[14] R. Parent. *Computer Animation*. Algorithms and Techniques. Morgan Kaufmann, 3rd edition edition.

[15] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Nov. 2009.