Euclase: A Live Development Environment with Constraints and FSMs

Stephen Oney, Brad A. Myers
Carnegie Mellon University
Pittsburgh, PA 15232 USA
{ soney, bam }@cs.cmu.edu

Joel Brandt
Adobe Research
San Francisco, CA 94103 USA
joel.brandt@adobe.com

***Abstract*—Euclase is a live development environment focused on creating interactive web applications. It uses a programming model that combines constraints and finite state machines to specify interactive behaviors. Euclase is "live" in the sense that while the user is developing code, their program is always executing. Changes made to the source of the program are reflected immediately in the running program. We identify some of the implementation and design challenges of making our development environment live, including performance issues, ensuring predictability, dealing with errors in the source, and handling edge cases such as the removal of code that is currently running. We also discuss how Euclase's use of finite state machines and constraints can help alleviate these difficulties.**

***Index Terms*—Live development, interactive applications, constraints, finite state machines, interaction design**

I. Motivation

Interaction designers often find it valuable to build high-fidelity interactive prototypes of interactive behaviors, both to fully explore an idea and to clearly communicate intended behaviors to developers [8]. Unfortunately, standard tools and languages for building interactive applications are too complex for rapid prototyping, and do not conceptually match how designers think about constructing interfaces [12]. Designers usually iterate through many designs, first sketching the behavior of the interactive application and then iteratively refining their designs through prototypes and mockups [2,8]. As a result, we believe that designers would benefit greatly from programming tools that are specifically designed to support rapid iteration on user interface behavior and with primitives made especially for creating interactive behaviors.

We consider designers to be a form of end-user programmers (EUPs) because they create these prototypes not as a primary goal, but as a step in designing a user interface [8]. To support these designers, we are creating a development tool for programming interactive applications. We want our development tool to be *live* (changes in the source are reflected immediately in the running program) in order to achieve beginner friendliness, quick evaluation, and quick experimentation. We will discuss these goals in more depth in the next section. Because we consider the development of interactive applications to be conceptually separate from the development of backend applications [4], we decided to start from scratch and develop new programming primitives especially for interactive applications. In this paper, we will refer to both the interactive editor and our underlying programming primitives as the "development environment", as the two are often inseparable from the users' perspective.

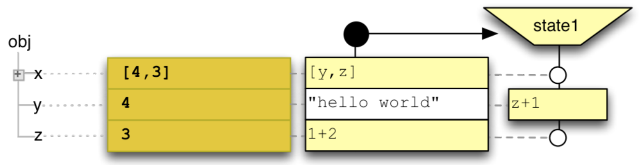

Fig. 1. A mockup for the display of the properties of an object in Euclase. Properties are represented in rows while different states are represented in each column. An entry for a particular property in a particular state represents a *constraint* that applies to that property in that state.

Our system, which is called *Euclase*, is a visual language with a live development environment. As changes are made to the source, the running program immediately updates to reflect these changes. In this paper, we will give a quick introduction to Euclase before focusing on the implementation and design challenges of making it a live environment. Previous research as part of the Euclase project identified the ways that designers think about programming tasks [8] and how a development environment might enhance creativity [11].

II. Live Development

One of the most important aspects of our development environment is that it is *live* – changes made to the program's source code are instantly reflected in the running application. We decided that it was important to have a live development environment for three primary reasons:

1) Beginner friendliness: Many interaction designers are not familiar with traditional programming languages. We hypothesize that by providing a live environment, we can help beginners better understand their programs. We believe that having a live development helps bridge the gulf of evaluation [9], which can be a significant barrier for new developers [3].

One reason is that beginning developers can become discouraged when a barrage of syntactic errors appears after they try to compile their programs. Even after they manage to fix all of those syntactic issues, by running their program, they will often find that they made semantic errors as well, which can lead to debugging and potentially another round of syntactic and semantic error fixing. A live development environment can help novice programmers in overcoming this barrier. Although syntactic errors can *sometimes* be made immediately visible in edit-compile-run environments, live programming allows both syntactic *and* semantic errors to become immediately apparent by enabling developers to immediately test their code.

Another important aspect of most live development environments is that the developer always has a working program[1]. One great aspect of spreadsheet programming, for instance, is that when the user makes a mistake in a particular cell's formula, the entire spreadsheet does not stop working [1,7]. Similarly, Euclase allows errors to be "localized": cells with errors only prevent the parts of the program from running that depend on those cells.

2) *Quick evaluation:* In addition to helping designers catch semantic errors, a live development environment allows designers to quickly evaluate their designs. This is particularly important for designers, as *reflection-in-action* – stepping back and evaluating their design as they are in the process of creating it – is a crucial part of the design process [13]. Previous research [8] has shown that designers today are more satisfied with their tools for designing an application's *look* than with those for designing an application's *feel*. While sketches and drawing applications allow designers to quickly evaluate the *look* of their application during the design process, Euclase is designed to be one of the first tools to allow them to quickly evaluate the *feel* of their application.

3) *Quick experimentation:* Experimentation is a crucial part of the design process and one that is not well supported by today's development environments [2]. Again, it is relatively easy to experiment with different application looks with sketches, drawing programs, etc. However, it is more difficult to change or experiment with the *feel* of the application. For example, imagine that the designer wants to tweak the scrolling "friction" to find a suitable value. With live development, this parameter can be iteratively modified to see the result, compared to a conventional environment where the user would have to re-run the entire program and re-enter the program state where this parameter is relevant.

Live development is particularly important in Euclase, as Euclase objects are intended to be highly stateful, with different behaviors in different states. Live development allows users to forgo the step of repeatedly putting their application in the desired state while iteratively testing it.

III. Euclase Design

Our development environment's primary goals are to be:

- *Succinct*. We want to be able to express the code for interactive applications in as few lines of "code" as possible. This means we must have powerful primitives.
- *Beginner friendly and approachable*. As previously mentioned, many interaction designers are not familiar with traditional programming languages [8]. This means that our programming primitives should be easy to understand and not too numerous, a common tension in new development tools [14]. Additionally, Euclase should be a "gentle slope" system; simple things should be simple and difficult things should scale linearly [7]. We also made our development environment live to help achieve this goal, as we will discuss later.
- *Expressive*. Interaction designers often have nuanced custom designs [12] and our development environment should support these designs. This means that while it is convenient to provide pre-made widgets (like buttons and scroll bars), designers should be able to understand and modify the code for these widgets if desired.

In deciding what primitives to use, we looked at *one-way constraints*, which have previously been shown to potentially simplify development of interactive behaviors [5,6], and there is evidence that designers think about relationships using constraint-based concepts [8]. A *one-way constraint* is a relationship that is declared once and maintained automatically. In pseudo code, we use `X::=Y` to express that the variable `X` is constrained to the value of the expression `Y`. This way, not only is `X` equal to the value of `Y` immediately after that expression is run (as in traditional assignment statements), if the value of the expression `Y` changes later, `X`'s value is automatically updated.

Previously, we built *ConstraintJS* [10], a JavaScript constraint library for Web environments. One insight we had in designing ConstraintJS is that because user interfaces are often stateful [4], it is useful to have constraints that *only* hold in particular application states. A simple example of this is a draggable icon. We want a constraint to make the icon follow the mouse, but *only* if that icon is in the dragging state. Thus, ConstraintJS allows finite state machines to dictate when one-way constraints are enabled or disabled.

In developing ConstraintJS, we found that many interactive behaviors could be specified entirely using constraints and finite state machines together, without extra imperative code. Thus, we chose to use these as two of the primary primitives for our development environment. However, using ConstraintJS requires advanced knowledge of JavaScript, CSS and HTML programming, so we wanted to take its ideas and make them more approachable to interaction designers.

The fundamental idea of Euclase is to use a spreadsheet-like presentation, since many people are familiar with writing formulas (which are a form of one-way constraints). Euclase lists properties in rows and states in columns. To specify that a property `y` should be constrained to the value `z + 1` in state `state1`, users would simply enter the text "`z + 1`" in the row of `y` and the column of `state1`. This is illustrated in Figure 1, which is a conceptual illustration of how we envision the system looking.

IV. Implementation

It is easy to make the case that reflecting changes in source code immediately is better than requiring an edit-compile-run loop. One reason there are not more live development environments is likely because of the difficulties of implementing them, especially for compiled languages.

[1] This is not necessarily inherent to live development environments but because of the implementation requirements of live development environments, it is common.

A. General Implementation Challenges

Implementation-wise, we believe declarative languages (where the source specifies *what goals* the program should accomplish) are more suitable for live development environments than imperative ones (where the source specifies *how* the program should accomplish those goals). Live development environments for imperative languages often need to re-execute code that the developer has changed. However, because imperative code can contain side effects, re-executing the code may have undesirable effects that may cause the running program to behave differently than it would in a standard edit-compile-run environment. For example, suppose a developer has the following imperative code in a live development environment:

```
var num_entries = 1;

function add_entry(e) {
      //CODE TO ADD ENTRY
      return num_entries++;
}
```

If the developer changes code in the `add_entry` function, should the development environment go back and re-execute that code? Should it save information every time the `add_entry` function was called and intelligently try to morph any side effects those calls may have had? Should it only affect future calls to `add_entry`? The answer is not clear and the fact that `add_entry` uses a side effect (`num_entries++`) means that each of these three possibilities would result in a different value for the variable `num_entries`.

Declarative languages, on the other hand, do not rely on side effects, meaning that any piece of code can be re-evaluated any number of times without changing the program's behavior. This is likely why many of the development environments for declarative languages are live, including Chrome's HTML editor and many dataflow language editors. Constraints are particularly well suited to live development environments. Since constraints are declarative[2], they can be re-evaluated without concern for undesirable side effects changing the meaning of the program. Additionally, the development environment can use the same constraint solver as used by application developers to make sure that the running program automatically updates in response to changes in the source program. The constraints expressed in Euclase are not entirely declarative, because they are enabled or disabled by finite state machines, where transitions are a form of side effect. However, this is accounted for in the Euclase implementation.

B. Euclase Implementation

Euclase is built in HTML and JavaScript using the ConstraintJS constraint solver [10]. ConstraintJS uses a *pull-based* constraint solver based on the algorithm described by Vander Zanden et al [15].

One implementation challenge of Euclase was dealing with changing variable references. In Euclase, not only may variable *values* change as the program is executing, but variable *references* may change as the user is editing the program source code. For example, if we have a cell for a variable `y` whose value is `x+1`, not only does the constraint for `y` need to be re-evaluated when `x` changes; it also has to be re-evaluated if the user *renames, moves, or removes* the property `x`, in order to achieve a live response to the edit. Also, if the user adds a variable named `x` closer in scope, then we need to use that `x` instead of the original `x`. Further, if the user deletes the property `x`, Euclase should handle that error without crashing the entire executing program.

Another challenge is handling finite state machine transitions because event listeners must be kept in sync with changing variable values. Suppose one finite state machine has a transition whose event is `on('dbl_click', selected_item)`, meaning to switch states when the selected item is double clicked, Euclase needs to update its underlying event listeners as `selected_item` changes (listening to every item and determining later on if it was `selected_item` would be too inefficient). Euclase also has mechanisms to ensure that constraint values are correctly timed with the transitions that occur in finite state machines.

The same issue applies with function calls: if a cell's value depends on the return value of a particular function and that function's code is edited, Euclase must update the value of the cell that relies on that function.

Performance is also a concern with Euclase. The fact that ConstraintJS used *pulled* constraints (which evaluate only when the constraint's value is requested) instead of *pushed* constraints (which evaluate as soon as a constraint's value may have changed) has important performance implications for Euclase. This helps insure that if the user is making a change to the program source code that will not be reflected in the running program, that change does not need to be re-evaluated. Further, if the user edits one cell which subsequently changes the value of 100 other cells that one particular constraint depends on, the last cell should only be re-evaluated at most one time, rather than 100 times.

V. Design Challenges of a Live Environment

In addition to the technical challenges of implementing a live development environment, there are also many human-centric questions about what the user would want the system to do in certain situations. For example, what if the program enters a state and the entire source specifying how the program should behave in that state is then deleted? For instance, suppose we create an icon with a "selected" state that highlights the icon after it has been clicked. What if we then delete our specification of what should happen in that "selected" state? How should our running application respond? Some viable possibilities are:

- To put the icon back in the last valid state it had before the selected state.
- To keep the icon "as-is" until the user resets the application.

[2] Some constraint evaluators allow constraints to have side effects [15]. However, constraints in the Euclase environment are more like spreadsheet formulas and cannot have side effects.

- To detect that an in-use state has been deleted and automatically reset either the whole application or reset the state of that particular icon.

All of these possibilities can be considered "valid" in some sense. In Euclase, we use the second option because it is conceptually the simplest and most predictable for users.

Also, there are some places where live development does not necessarily mean a quicker turnaround time for the development of the application. Suppose, for instance, that the developer has mostly written an application but now wants to make tweaks to its "loading" screen that should flash for about three seconds before the application loads. In a live development environment, how does the programmer specify that they want the loading screen to *stay visible* while they are making tweaks to it, instead of requiring repeatedly resetting the application (akin to the standard edit-run loop in most compiled languages)? Some possibilities for dealing with this would be to:

- Give the developer the option to temporarily disable some transitions while they are developing, to ensure that the program stays in a particular state.
- Allow developers to add breakpoints that "freeze" the current state of the program but still allow them to make changes to particular cells if necessary.

In our implementation, we are planning to use the latter option, to avoid the complications that may result from having to go back and mark transitions as "special" and because users with more development experience may have more familiarity with the ideas of breakpoints.

The importance of visual design to many designers may also make it difficult for our users to specify *everything* solely in code. For instance, when creating an interface layout, most designers will be familiar with direct manipulation interfaces where objects can be directly grabbed, dragged, and their properties changed in the running interface, rather than dealing with the underlying source code. For that reason, we plan on implementing a "design mode" where users specify that they no longer want to edit source code by hand and the code should not be live, but instead the users can drag and drop objects in a static version of the interface (the interface should not be interpreting clicks and events).

We also want to insure that there is no difference between a program executing live and a program that executes later. There are some questions about timing and when certain cells should be executed. For instance, suppose a cell has the value `random()`, which returns a random number. When should that random number be generated? Only when the user first enters the cell's value? Only when that value is used? Current spreadsheets reevaluate the `Rand()` formula unpredictably whenever the sheet is reevaluated. Currently, our implementation would use the latter option, so that a program executing live in the development environment behaves the same way as it would if it went through compile-edit-run loop.

Finally, the fact that the finite state machines used by Euclase are imperative presents another design challenge. Sometimes, it *is* important to be able to keep track of how an object got into a certain state. For example, if a property has the value `1` on a transition from the start state but the designer specifies that value *after* that transition has already happened, the property's value should be `1` and not `undefined`.

VI. Future Work

Euclase is currently under development. When development is complete, we plan on making the full source and a demo freely available. We also plan on improving the implementation and adding more features, tutorials, documentation, etc.

We also want to evaluate the effect that Euclase has on developers. We plan on first comparing Euclase to traditional languages and tools for creating interactive applications, like Adobe Flash Professional and JavaScript. We then want to investigate to what extent any differences are because Euclase is a *live* development environment. We are also interested in finding out the extent to which Euclase can encourage designers to test out more possibilities and be more creative. By enabling quick experimentation, we want to see if designers end up more satisfied with their designs.

Acknowledgements

Funding for this research comes from a grant from Adobe, and from NSF grant IIS-1116724. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect those of any of the sponsors.

References

[1] Burnett, M., Atwood, J., and Djang, R. W., "Forms / 3 : A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm," p 1–51, 2001

[2] Grigoreanu, V. et al., "What designers want: Needs of interactive application designers," *VL/HCC*, p 139–146, 2009

[3] Ko, A. J., Myers, B. A., and Aung, H. H., "Six Learning Barriers in End-User Programming," in *VL/HCC*, 2004, p 199–206

[4] Letondal, C., Chatty, S., Phillips, W. G., and André, F., "Usability requirements for interaction-oriented development tools," in *PPIG*, 2010, p 12–26

[5] Meyerovich, L. et al., "Flapjax: A Programming Language for Ajax Applications," *OOPSLA* 2009, p 1–20

[6] Myers, B., "Separating Application Code from Toolkits: Eliminating the Spaghetti of Callbacks," *UIST* 1991, p 211–220

[7] Myers, B., Hudson, S., & Pausch, R., "Past, Present, and Future of User Interface Software Tools," *TOCHI*, v. 7, p. 3–28, 2000.

[8] Myers, B., Park, S. Y., Nakano, Y., Mueller, G., and Ko, A., "How Designers Design and Program Interactive Behaviors," in *VL/HCC*, 2008, pp. 177–184

[9] Norman, D., *The Design of Everyday Things*. New York, New York, USA: Doubleday, 1988

[10] Oney, S., Myers, B., & Brandt, J., "ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States," in *UIST*, 2012, pp. 229–238

[11] Oney, S., Myers, B., & Zimmerman, J., "Visions for Euclase : Ideas for Supporting Creativity through Better Prototyping of Behaviors," 2009

[12] Park, S., Myers, B., & Ko, A., "Designers' Natural Descriptions of Interactive Behaviors," *VL/HCC* 2008, p 185–188

[13] Schön, *The Reflective Practitioner*. London, England: Temple Smith, 1983

[14] Travers, M., "Recursive Interfaces for Reactive Objects," *CHI*, 1994, p. 379–385

[15] Vander Zanden, et al., "Integrating Pointer Variables into One-Way Constraint Models," *TOCHI* vol. l no. 2, p 161–213, 1994.