A Programming Paradigm for Creating Interactive Behaviors

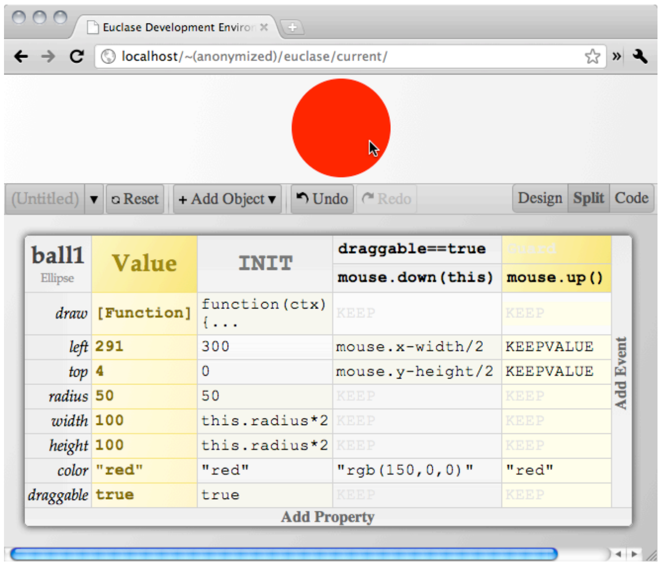
ABSTRACT
Interactive behaviors (the parts of an application that respond to user input) are difficult for designers to author. Euclase provides a new programming paradigm that supports the features of interactive behaviors that make them difficult to write in traditional programming languages. By combining the more appropriate features of spreadsheets, event-based programming, state diagrams, constraints, and live prototyping, Euclase concisely expresses the kinds of custom behaviors that designers want to create. A preliminary usability evaluation indicates that Euclase has promise.

Author Keywords
Interactivity programming, programming languages, programming environments

ACM Classification Keywords
D.2.6. Programming Environments D.2.2. Design Tools and Techniques

INTRODUCTION
Previous research has shown that while interaction designers are able to program the *look* of their applications, they have trouble programming the *feel* [4]. The feel of an application is specified by its *interactive behaviors*, which we define to be the changes of an application's user interface in response to user input. Whereas most tools for defining interactive behaviors have focused on adding widgets on top of existing languages, our goal is to improve the underlying programming paradigm, in order to allow designers to author their own unique behaviors, and to be able to customize the widgets in the library.

Design Process
In designing our paradigm, we first set out to find out what are the issues when creating interactive behaviors [4], which revealed that designers want to author custom behaviors, but are often stymied by existing tools, and the need to transfer their ideas to developers. We then performed two participatory design workshops with interaction designers and programmers [5], using several examples of interactive behaviors that are difficult to program in discovered that designers use gestures, context, and analogies when trying to express their designs to developers. Using these results, we formulated a new programming paradigm to support the kinds of behaviors that we observed designers needing. We also created a prototype of a programming environment using our paradigm to evaluate its feasibility and performed a preliminary evaluation of its learnability. We call our paradigm and environment Euclase, short for **E**nd **U**ser **C**entered **L**anguage, **A**PIs, **S**ystem, and **E**nvironment (see Figure 1).



ball1 Ellipse	Value	INIT	draggable==true	Guard
			mouse.down(this)	mouse.up()
draw	[Function]	function(ctx) {...	KEEP	KEEP
left	291	300	mouse.x-width/2	KEEPVALUE
top	4	0	mouse.y-height/2	KEEPVALUE
radius	50	50	KEEP	KEEP
width	100	this.radius*2	KEEP	KEEP
height	100	this.radius*2	KEEP	KEEP
color	"red"	"red"	"rgb(150,0,0)"	"red"
draggable	true	true	KEEP	KEEP

Figure 1 - A Euclase application with a single object, showing design and code views. The event `mouse.up()` has just happened.

INTERACTIVE BEHAVIORS
In designing Euclase, we identified four primary features of interactive behaviors, the first three of which tend to make them difficult to program in conventional languages:

1. Interactive behaviors are state dependent. A user action in one state often has a different meaning than that same user action in another state. Furthermore, there are often many different states in a program, with different parts of the program being in different states at the same time. For example, a drawing editor might be in drawing-rectangle mode, while the user is operating an independent button, which is in interim-selected state.

2. Interactive behaviors are also often constraint-heavy. There are constraints between the view of an application and its model, and constraints among graphical objects that define the layout of the application. For example, constraints might be used to ensure that the text of a button is centered or that two buttons are adjacent. Furthermore, the appropriate constraints change based on the current state. When the user is operating a color widget, for example, the color of a selected object is set to be the widget's color, but when changing which objects are selected, then the color widget is set to be the color of the newly selected object.
3. Animations, which are an excellent means of improving user understanding of program behavior, are often integrated with interactive behaviors. This can increase the complexity of the interactive behavior, as animation duration and interruption now has to be taken into account when writing the code for the interactive behavior.
4. Finally, interactive behaviors are usually event driven. This makes the event-action callback paradigm popular among languages that are intended for writing interactive applications, including Javascript and ActionScript.

A programming language made for interactive behaviors must be able to support these features effectively. In addition, we found from our participatory design workshops [5] that a tool to create interactive applications must also be sufficiently flexible to support nuanced design requirements, since designers want to have fine-grain control over many aspects of the timing, movement and behavior. Designers also need support for *reflection-in-action*, the ability to test and evaluate what one is creating while in the process of creating it.

LANGUAGE & ENVIRONMENT DESIGN

To support reflection-in-action, Euclase (shown in Figure 1) has two views, like Dreamweaver and other HTML editors. On the top is the "design" view, where users may see and interact with their application. Below this is the "code" view where users write their applications. In our prototype, there is no explicit compilation step – the design view dynamically changes its appearance and behavior as Euclase objects are created, deleted, and modified.

Objects

Euclase objects use a hierarchical prototype-instance model. `Object` is the main object type, `Graphical Object` is a type that extends `Object` and also creates a graphical object in the design view. `Graphical Object` is extended by `Rectangle`, `Ellipse`, `Image`, etc. Each of these objects is represented in the code view as an "object sheet." Every object sheet shows the object's type, optional name, properties, and events.

Properties

When an object is first created in Euclase, it looks like a standard property sheet, which has a row for each property and its current value. The default properties that are created depend on the type of object. A standard object (type `Object`), for example, has no properties by default, whereas a generic `Graphical Object` by default has fields for horizontal and vertical location (`left` and `top` respectively), `width`, `height`, and a `draw` function whose default value depends on the type of graphical object, but may be overridden, if necessary, to create a custom graphical object.

Constraints

The value for a property can be a constant, like `50` for `left`, or `"red"` for `color`. Alternatively, the value can be a *constraint*, which recalculates the value based on a formula that uses the properties of this or other objects. The constraint solver in Euclase is based on the Amulet solver [8], and supports one-way constraints with indirection (the target object can be itself be calculated by a constraint, as in `this.object-at-left.color`).

Currently, we use a Javascript-like language for the syntax of the constraints, to support full flexibility of expression. In the future, we may explore a new syntax based on investigations of what designers find more natural, guided by commonalities in how they describe program behavior [6].

Events

Whereas object properties are represented by rows in the object sheet, events are represented as columns. The first column, immediately to the right of the property names, lists the current value for every property. Event columns are to the right of the current value column.

Every event has three parts: an optional "guard", a "trigger," and a column of property values. The trigger is the event that occurs, such as a mouse up, a mouse down over a particular object, etc. Triggers can also be Boolean expressions like `this.x > 500`, that fire any time the value of the expression switches from `false` to `true`. A guard is a conditional expression that enables or disables the event. Guards are intended to allow the object to have different responses in different application or object states. An event is "activated" when the guard evaluates to `true` and the trigger is fired.

To begin, there is a special event called "`INIT`," short for initialization. Every object has an `INIT` event. The cells in its value column represent the default values for every property in the object sheet (the values before any event is activated). Once an event is activated, every property takes on the values of the cells in the event's value column. These cells might have static values (e.g. `3`, `"blue"`, `5+2`, etc.), constraints (e.g. `mouse.x`, `foo.a + bar.b`, etc.), or a special value, described next. After an event is activated, every property gets its value from that events's column until another event is activated. Euclase highlights the column from which the current values are being used, as shown in Figure 1. Euclase's "reset" button allows users to return to the default state (as specified by objects' `INIT` events) and undo any changes due to activated events.

As an example, to make an object follow the mouse after the mouse is pressed on the object, a column would have the trigger `mouse.down(this)`, a guard of `true` (the default), and constraints in the `left` and `top` properties that use the value `mouse.x` and `mouse.y`. In Figure 1, the circle will be dragged from its center, and will get dimmer while being dragged (because the `color` property is set to a dark red).

Special Values

As mentioned before, cells may have special values:

`KEEP` is the default value for empty cells outside of the `INIT` event (for which the default is `NULL`). When `KEEP` is assigned to a property in a column, that property keeps the value or constraint that it currently has. In Figure 1, none of the events change the initial value of the `radius` property.

`KEEPVALUE` is similar to `KEEP` but the property keeps only its current value and removes the constraint. For example, in Figure 1, the circle should stay at its last position when the mouse button is released, so the `mouse.up()` column uses `KEEPVALUE` in the `left` and `top` slots, so when this event happens, these properties will keep the current value, but discard the constraint.

`ONCE()` converts an expression from a constraint into a value by evaluating the constraint only once each time the event is activated. For instance, a cell with the expression `ONCE(mouse.x + this.foo)` in the value column for some event `E` takes the values of `mouse.x` and `this.foo` immediately when `E` is activated and creates a static value from their sum. `ONCE` is useful when a property's value needs to be calculated and stored before other constraints use it. For example, if we want event `E` to constrain an object's `left` property to be `mouse.x - draggingX` and `draggingX` to be `this.left - mouse.x` at the time of event `E` (so the object will be dragged from the place that it is clicked with the mouse), we could set `draggingX` within a `ONCE` statement: `ONCE(this.left - mouse.x)`. These values, which would form a cycle without the `ONCE` operator, instead constrain the horizontal position of the object to where it was relative to the mouse when `E` was activated.

Design Rationale

Our language and environment design combines features from property sheets, state transition diagrams, event languages, and spreadsheets, which have each proven successful as mechanisms for expressing certain kinds of computations, but none of which are adequate alone. Property sheets, which normally list object properties and their values, allow the developer to see what properties an object has and their current value. Euclase's guards and events allow state transition diagrams to be easily translated into working code without having to specify behavior in every state or state combination, avoiding the state explosion problem.

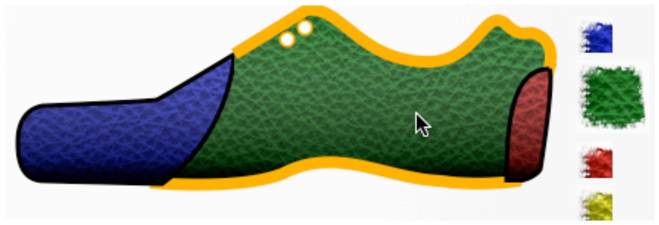


Figure 2 - Euclase can emulate two-way constraints by enabling and disabling constraints conditionally. In this implementation of a shoe customization site, a constraint sets the selected color to that of the currently selected shoe part (outlined in orange) of the currently selected shoe part or the currently selected shoe part's color to the last color picked, depending on what was selected last. Constraints also maintain the distance between the color palettes so that if one grows, the others move automatically. This example requires 6 constraints (3 to constrain shoe part colors and 3 for color positioning) in 21 event columns (excluding `INIT`s) across all 7 graphical objects and one non-graphical object. This will be significantly reduced in the future when we add support for lists of objects all with the same behavior.

Euclase also derives several features from spreadsheets, including the way users enter in constraints – by writing formulas among cells in a table rather than in the context of imperative code. Another trait Euclase shares with spreadsheets is that Euclase is a "live" system in that the designer can edit properties even while the system is running. This fosters exploration and the incremental building up of the desired program and enables more reflection-in-action than languages that require the edit-compile-run loop or even the edit-run loop of interpreted languages like Javascript.

Euclase augments, as well as combining, some of the features from these four paradigms. For example, constraints, which are always activated in spreadsheets, can easily be enabled or disabled in Euclase depending on different application states. This replaces most places where previous systems have needed two-way constraints, while giving the designer more control. For example, the shoe-color selection widget shown in Figure 2 constrains the selected color to be the color of the selected shoe part or alternatively the color of the selected shoe part becomes the color that was last clicked, depending on whether a shoe part or color was clicked last. In other words, states control whether the value comes from or goes to the selected object. Another example where this would be useful is when an object should either grow or move based on where it is clicked; in which case different states would activate different constraints.

PROTOTYPE AND USER STUDY

We created a prototype of Euclase in client-side Javascript that runs entirely in Firefox or any modern Webkit-based browser. The code in each cell is parsed and interpreted in Javascript with the help of the JSCC parser generator[1], so that the syntax may be changed in the future.

[1] http://jscc.jmksf.com/

Preliminary Usability Evaluation

As an initial evaluation of our proposed paradigm for creating interactive behaviors, we recruited three interaction designers to write interactive behaviors using our system. Two of our participants had less than one year of programming experience while the third had more than five years of programming experience. Participants were first shown an example of creating a draggable rectangle to guide them through the system. This took approximately 15 minutes for each participant. They were then asked to modify this example by changing the color of the rectangle during hover, which they were all able to do without difficulty.

Participants were then asked to implement a scrollbar trough and indicator where the user could drag the indicator within the trough. Participants were first asked to formulate their strategy for ensuring that the indicator remains in the range of the scroll bar. Two participants were able to immediately formulate a strategy that would fit into our described paradigm. The third participant needed help formulating a strategy, but had no difficulty completing the task after being given a strategy. Upon knowing their strategy, participants were given a utility function `range(val, low, high)` that returned `max(low, min(high, val))`. After being given a strategy, two participants were able to complete the task with no problems, while the third participant had conceptual difficulties primarily due to confusion about coordinate spaces. All participants were able to complete the task within 20 minutes (from problem description to working implementation).

Whereas the experienced programmer tended to write constraints for relative graphical object positioning, the novice programmers tended to hard-code numeric values for positioning, making many more incremental changes. Because participants worked in smaller increments, they tended to catch errors immediately – not only syntax errors that would be caught by static analysis, but also conceptual errors - programs that would run, but not have the desired behavior. Participants expressed that not only did they find the ability to iterate quickly very useful, two participants expressed that they wanted even more immediate feedback, in the form of the design view showing the effect of a particular change as the user is still typing in a cell, even though its event had not yet been activated.

RELATED WORK

Other researchers have investigated some of the factors of programming interactive systems from a psychological perspective [2]. We believe our design alleviates many of the issues presented by these researchers. There has also been research on how to enable spreadsheet-like programming of applications, including the Forms/3 [1] system, which proved that procedural & data abstractions and graphical output were viable in the spreadsheet paradigm. However, most of this research has been focused on general functional programming and does not take into account the particular challenges of programming interactive behaviors. There has also been a number of constraint programming systems, including Kaleidoscope [3], which enables constraints to be turned on and off. In addition, easy-to-use programming languages and environments, many of which make use of non-textual elements, such as Scratch [7], have long been a subject of research. However none of these languages have been designed to deal with the particular complexities of custom interactive behaviors.

CONCLUSION & FUTURE WORK

Our initial results from the usability evaluation shows promise for our paradigm and suggest many avenues for further development. The two participants with less programming experience expressed interested in the addition of a direct-manipulation style tool that would allow them to lay objects out directly on-screen, as is possible in Flash Catalyst and the design view of Dreamweaver. We also hope to implement a form of programming-by-demonstration where users can author interactive behaviors by demonstrating their desired behavior to the system.

One participant with years of imperative programming experience expressed that they wanted to be able to set the properties of object `B` from object `A`'s object sheet (pushing values) rather than having to use constraints to pull values. We are currently investigating ways of allowing users to write more imperative style code like this without adversely affecting the visibility of current object sheet.

We are also working on integrating animations into Euclase with the addition of a "timeline" cell view where users can specify animation paths, timing curves, and exceptional cases for when the animation is interrupted or cancelled while operating.

REFERENCES

1. Burnett, M. *et al.* Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.,* vol. 11, no. 2, (2001), 155-206.
2. Letondal, C., Chatty, S., Phillips, W. *et al.*, "Usability requirements for interaction-oriented development tools," *PPIG*, 2010.
3. Lopez, G., "The design and implementation of Kaleidoscope, a constraint imperative programming language," *University of Washington, Seattle, WA*, 1997.
4. Myers, B. A. *et al.* How Designers Design and Program Interactive Behaviors. *IEEE VL/HCC'*2008, 177-184.
5. Ozenc, F. *et al.*, "How to Support Designers in Getting Hold of the Immaterial Material of Software," 2010.
6. Park, S., Myers, B. & Ko, A. Designers' Natural Descriptions of Interactive Behaviors. *VL/HCC* 2008 185-188.
7. Resnick, M., Maloney, J., *et al.*, "Scratch: programming for all," *Communications of the ACM*, 11, 2009, pp. 60-67.
8. Vander Zanden, B. T. *et al.* Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Trans. Program. Lang. Syst.,* vol. 23, no. 6, (2001), 776-796.