Alternative Programming Interfaces for Alternative Programmers

Toby Schachman
New York University, Interactive Telecommunications Program
tqs@alum.mit.edu

Abstract

This paper seeks to broaden the view of what programming is, who programs, and how programming fits in to larger systems. With growing frequency, people are approaching programming from unlikely backgrounds such as the arts. Often these new programmers bring with them ways of working which are incompatible with mainstream programming practices, but which allow for new possibilities in programming interfaces. This paper makes suggestions for the design of these new programming interfaces. It presents as a case study and demonstration Recursive Drawing. Recursive Drawing is a reimplementation of the textual programming language Context Free as a graphical, directly manipulable interface. Instead of a compiler or interpreter, Recursive Drawing's programming interface is modeled as a constraint solver. This allows the programmer to modify the program's source code by manipulating the program's output. Additionally, the design of the interface focuses on program transformation, rather than program construction.

1. Introduction

This paper posits the emergence of a new generation of "alternative programmers." This new generation has no programming background, but has a need to program computers in order to realize their goals. Indeed they approach programming from unrelated backgrounds, from well-developed disciplines with their own paradigms for understanding the world.

This influx of new programmers provides an opportunity to radically shift the way programming is done. But to realize this opportunity, we will need to be sensitive to the backgrounds and goals of this new generation, and to reconsider the activity of programming itself.

This paper explores the notion of *alternative programming interfaces* for the next generation of *alternative programmers*.

Sections 2 and 3 define these terms, with the goal of expanding the scope of what is traditionally considered programming and who are traditionally considered programmers.

Section 4 introduces a case study, Recursive Drawing. Recursive Drawing is a reimplementation of the textual programming language Context Free as a graphical, directly manipulable programming interface. The following sections are illustrated with examples from Recursive Drawing.

Section 5 suggests an alternative to the linear nature of programming. Instead of compilers, programming interfaces are implemented as constraint solvers, which allows the programmer to modify the source code by modifying a running process.

Section 6 suggests shifting our focus from program construction to program transformation.

Section 7 discusses the strengths and weaknesses of Recursive Drawing and directly manipulable programming interfaces in general.

Section 8 concludes.

2. What are Alternative Programming Interfaces?

I will address programming interfaces from three perspectives: physical, conceptual, and social.

2.1 Physical Interfaces

Physical interfaces concern the human body and the computer's "body"–its hardware inputs and outputs–and the affordances allowed by each of these bodies.

The dominance of the text medium tends to dictate the form of physical interfaces. The human communicates to the computer through typing on the keyboard. The computer communicates back through whichever outputs the program addresses: the screen, speakers, or other peripherals. But even if a program is intended to create audio or graphical output, throughout much of the programming process the computer communicates to the programmer through text on the screen (through the console).

Communicating text back and forth through these interfaces has proven to be effective because this medium can be made largely unambiguous (through formal languages), it emulates how we communicate intellectually with other humans (talking and writing), and we have evolved conventions (syntax and semantics) for densely packing abstract information into this form.

Alternative physical interface possibilities for programming include:

1. Visual interfaces. These exploit the full graphical capabilities of the screen rather than just text, and often encourage human input through a spatial pointing device such as a mouse, trackpad, or stylus, in addition to or replacing keyboard input.
2. Touch screen interfaces. Building on visual interfaces, but with the human touching the screen directly. Two potential advantages over mouse-based visual interfaces are a more direct feeling of manipulation of the screen's output, and the expressive possibilities of multiple points of contact (multitouch).
3. Arbitrary interfaces. These include the human communicating to the computer using physical gestures in space, sounds, or the manipulation of peripheral sensors such as knobs and accelerometers. The computer communicates back visually, aurally, or haptically.

None of these alternative physical interfaces have produced widely adopted general purpose programming environments. However, they have had significant success in limited domains. Patching environments such as Pd [20], Max [19], vvvv [3], Quartz Composer [16], and Isadora [7] use visual interfaces with dataflow semantics to program interactive audio and visual works. Rebecca Fiebrink's Wekinator [11] uses arbitrary inputs (such as cameras and accelerometers) and arbitrary outputs (usually sound) to program novel musical instruments using a supervised learning workflow both on the part of the computer (recognizing human gestures) and the human (learning to "play" the instrument).

I see three reasons to continue pursuing these alternative physical interfaces, in growing order of importance:

1. We have the technology. The programming interfaces today are largely a result of the technological evolution of the computer going all the way back to mainframes and teletypes. This historical bias suggests that alternative physical interfaces may have dormant potentials.
2. Alternative physical interfaces allow new workflows for programming. Communicating text back and forth is a *turn-based* experience. Programmer talks, computer talks, etc. Alternative interfaces can allow for a *continuous* feedback loop between human and computer. For example, each of the patch-based visual environments allow the user to adjust parameters with sliders, and see the results of these changes in realtime. This *live coding* workflow is possible with text–syntax highlighting is a form of realtime feedback, for example–but the medium does not naturally support it. This is why textual interfaces supporting live coding are often augmented with visual inputs like sliders, as in Bret Victor's "Inventing on Principle" demonstrations [24] and OpenEnded Group's Field [4].
3. Alternative physical interfaces engage different parts of the human brain. Textual interfaces engage the "language center" of our brain. We have difficulty expressing concepts to the computer which we cannot translate through this part of our brain. Yet many of our most profound ideas we think of *visually* or *kinesthetically*. Alan Kay relates an anecdote about the mathematician Jacques Hadamard, who polled the great mathematicians and scientists of his day about how they "do their thing." Most replied that they did not think using mathematical symbols (the language center of mathematics) but rather imagined figures or even experienced sensations. Einstein replied, "I have sensations of a kinesthetic or muscular type." [15]

2.2 Conceptual Interfaces

Conceptual interfaces concern the the metaphors we use to think about our programs. Examples include objects, actors, structures, and streams. Conceptual interfaces are largely equivalent with the semantics of a programming language. They are the key mental structures that must exist solidly and isomorphically in the mind of the programmer and the mind of the computer (that is, in its implementation), in order for programmer and computer to collaborate effectively.

2.3 Social Interfaces

Social interfaces concern programming's relationship to society and how program creation interacts with social systems. It addresses the questions:

1. What is programming and how does it relate to the rest of the world?
2. How should we program together?
3. Who should program?

A traditional, now humorously out-dated view, is that programming is calculating. Computer operators feed problems into a (physically huge) computer which spits out an answer. Calculation (usually prefaced with "cold") is the antithesis of humane, creative activity. We have made inroads towards a new perspective, where programming is seen as a creative, collaborative process between human and machine. This is largely due to pioneering work such as Ivan Sutherland's Sketchpad [22] and Douglas Engelbart's NLS [10]. Yet the traditional view still maintains a hold on the collective (un)consciousness. Many people are intimidated by computers, or intimidated by programming. They see the

computer as *The Other*, a soulless entity with whom they cannot engage.

The next question concerns how we relate to our human collaborators in programming. People have always worked together in teams when appropriate, but the internet and platforms such as GitHub [2] have made the world of code more like an ecosystem than ever before.

The semantics of a language often reflect and reinforce the organizational structures that collaborate using the language. Social interactions are subtle–and I want to avoid making sweeping generalizations–but for the purposes of illustration I will provide a stereotyped example: Java's semantics reinforce an insulated hierarchical organization of programmers where one programmer cannot "step on the toes" of another. Contrast this with Ruby, whose semantics encourage substantial monkeying with the language internals. Ruby's semantics thus require more cross-communicative teams, necessarily smaller, or alternatively the top-down institution of conventions like Rails [13].

I am not implying that any one way of collaborating is better or worse, just that there is a relationship between programming interface design and the way we work with each other. I see substantial opportunities to research the sociological implications of human collaboration in programming.

Finally, the question of who should program I will address in the next section.

3. Who are Alternative Programmers?

Many profound advances in programming were the result of people reconsidering the question, *who are the programmers*? Engelbart's NLS expanded the view of programmers from business analysts and artificial intelligence researchers to any information worker [10]. Smalltalk originally focused on children as programmers [15]. Hypercard was developed and distributed at Bill Atkinson's insistence that "end users" need programming capabilities [6]. Even web programming, at least initially, promoted a culture where anybody could contribute their content or software to the web.[1]

I believe a new generation of programmers is emerging. These "alternative" programmers are people who do not self-identify as programmers, but who regularly program computers in order to achieve their goals. Alternative programmers can include for example musicians, performers, writers, visual artists, designers, scientists, architects, and activists.

Evidence of this emergence includes:

1. The growth of the DIY hacker and maker cultures, with hacker spaces, hackathons, workshops, and meetups. These serve as social support structures for alternative programmers.
2. Platforms and communities built around beginner-friendly, dive-right-in programming, such as Arduino [1] and Processing [12].
3. The ubiquitous use of computers as a means of creative expression, ranging from editing video for YouTube to using Max for live performances.

Some alternative programmers take well to the current ecosystem of programming interfaces. But certain creative processes–relied on by alternative programmers in their primary work–are inadequately accommodated by traditional programming interfaces. For example:

1. Non-linear workflows. Traditionally, programmers build software towards a *specification*. The traditional programming process works best when this specification is clear and unambiguous. But many alternative programmers work towards more ambiguous goals, driven for example by feelings, intuitions, or emotions. To support these goals, programming interfaces must support exploration and discovery.
2. Improvisation. Improvisation is the art of tuning the mind's rhythms and momentum to allow for the organic exploration of a conceptual space. Improvisation is usually brought up in the context of music, but it is often central to the process of writers, visual artists, and other creative explorers. To support improvisation, a programming interface must first provide continuous feedback, as turn-based feedback will impart its own rhythm on the improvisation. Second it must let the improviser work directly in the representation of concern. Any need to translate to a different representation, for example to translate a visual or musical idea to numerical values, can break the "flow" of improvisation.

The use of computers in general for creative expression prompts the question: Where do we draw the line between *programming* and *authoring*–the use of specialized computer tools to produce specialized results? I don't have a good answer but I encourage the reader to take a broad view of programming. For the purposes of this paper, I will take programming to mean any instance of *designing a system*. Bret Victor's distinction between static and dynamic pictures may also be helpful [23].

Like Smalltalk or Hypercard, I intend to blur the line between *programmer* and *user*, between programming and authoring. Consequently, throughout this paper the reader is encouraged to play with substituting the words "user" and "programmer."

4. Case Study: Recursive Drawing

To explore and demonstrate alternative programming interfaces, I created Recursive Drawing. Recursive Drawing is a reimplementation of the textual programming language

[1] There seems to be a pattern where an environment is developed for alternative programmers, then as a consequence of success is overtaken by "real" programmers. Adobe Flash, originally designed for animators who wanted to work with interaction, also follows this pattern.

Context Free [9] as a graphical, directly manipulable interface.

Context Free is similar in thrust to Logo's Turtle Graphics [18]. It is a small, elegant programming language for creating graphics. But the two languages diverge fundamentally in their semantics and the programming experience they induce. Logo has imperative semantics and induces in the programmer a "body syntonic" feeling. The programmer directs a drawing robot's movements through space. Context Free has pure (side-effect free) semantics and induces in the programmer a more abstract, disembodied feeling. The programmer declaratively nests spatial transformations. Each declaration is independent of any global context, hence the name Context Free.

In Context Free, the programmer specifies *rules*. A rule is simply a list of references to other rules, each with a spatial transformation (e.g. translation, rotation, scale) to apply to that rule. There are two primitive rules, circle and square, which simply draw the shape. Rules can reference themselves. See Figure 1 for an example. [2]

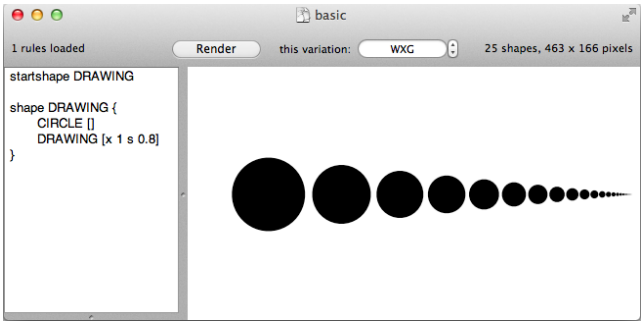

Figure 1. A basic example of Context Free. The rule DRAWING draws a circle, then draws itself translated one unit to the right and scaled by a factor of 0.8.

Through self-reference, Context Free encourages the exploration of self-similar shapes: fractals. These shapes are co-recursively generated infinite structures, the graphical equivalent to Lisp's streams [5] or Haskell's infinite data structures [14]. Execution of a Context Free program can be understood as the recursive substitution of rules with their definitions. Context Free features a form of lazy evaluation, in that when drawing to the screen, recursion halts when the shapes are too small to be seen (i.e., at a suitably small, sub-pixel size).

Context Free presents a paradox. On one hand, it features semantics which are considered advanced, even esoteric, by the mainstream programming community: referential transparency and co-recursive structures. On the other hand, it is visually intuitive and has been enthusiastically adopted by artists.

[2] Context Free has many more features, but these basic ones will be sufficient for the purposes of this paper.

Recursive Drawing was inspired by this paradox, along with Bret Victor's challenge to create directly manipulable programming interfaces [23]. Directly manipulable interfaces not only feature continuous feedback, but also allow the programmer to manipulate the program using a representation that resembles the final output of the program, in this case graphics. Note that graphical programming interfaces are not necessarily directly manipulable. For example, patch-based languages can produce graphics but in this case the programmer manipulates the patches, not the graphical output itself.

Context Free presented a comparatively easy target for a directly manipulable interface. Its output representation is graphical and its semantics are based around spatial transformations. Thus user interface conventions from graphical authoring tools such as Photoshop could be employed by the programming interface. Additionally, lazy evaluation is a powerful abstraction which allows very small programs, so I could mitigate the information density constraints that often plague graphical programming interfaces.

Dynamic interaction is central to the point of Recursive Drawing, so I encourage the reader to watch a short video demonstration of Recursive Drawing or to try it out (in the browser). These resources are available at:

`http://totem.cc/onward2012`

In the following sections I will contrast Context Free with Recursive Drawing in order to illustrate alternative programming interfaces. I am not claiming that Recursive Drawing is a better way to program than Context Free, or even that any of these alternative interfaces are better than traditional interfaces. I only wish to illustrate that alternative possibilities are available.

5. Rethinking Causality

Programming is traditionally a forward-progressive, linear activity. We think in procedures: one thing leads to another. We have a goal in mind, and in order to reach our goal we start at the foundation and build our software step by step. This section explores relaxing this notion of linear, forward progression.

5.1 Cause and Effect

Changing the source code of a program so as to effect a specifically desired change in its output is a very common activity in programming. Indeed, in the context of traditional programming interfaces, this activity can be seen as equivalent to debugging.

Traditionally, if a programmer wants to change the output of a program in a specific way, she must solve two problems:

1. Find the line(s) of code which resulted in the part of the output she is concerned with.
2. Understand the relationship between this code and the output, in order to adjust the code accordingly.

To deal with the first problem, the programmer must trace backwards in the causal chain ending at the output. That is, she must trace back to the line of source code which initially started the chain. Some tools keep track of this chain of causality, and attach this history to the output in some form. For example:

1. Stack traces show the chain of functions which were called to reach a given point in the execution of a program.
2. Console logging is often performed for the sole purpose of determining if certain code has been reached. Here, the programmer manually does a binary search through the source code, testing whether different parts of code are or are not part of the causal chain of concern.
3. A DOM inspector in a browser allows the programmer to point at an element on the screen and see what node of the DOM tree was responsible for drawing it. However, the programmer cannot look further back in the causal chain to see, for example, what line of Javascript was responsible for creating that DOM node.
4. Patch-based languages graphically show the flow of data through the system. Because there are no side-effects, the flow of data is equivalent to the flow of causality.

To deal with the second problem, understanding the relationship between code and output, a programmer usually uses some form of test-and-repeat. This activity can take the form of:

1. A purely mental procedure. The programmer simulates the computer's operation step-by-step in her head.
2. Turn-based feedback with the computer. The programmer changes the code and recompiles, or interacts with a REPL.
3. Continuous feedback with the computer: live coding.

A program is a collection of causal relationships, and to program effectively the programmer must understand these causal relationships. The tools and processes mentioned above help build this understanding, and help solve the practical need of effecting a desired output with a program. But I believe we can go further by rethinking the nature of our programming interfaces.

5.2 Constraints Generalize Procedures[3]

We currently think of a programming interface as a one-way causality flow: a procedure. This one-way arrow is embodied by a compiler, interpreter, or live coding environment. Compilers are functions. They take input (source code) and return output (a running process). REPLs and live coding environments are stream processors. They take a stream of input (source code modifications) and incrementally modify a running process.

The alternative is to think of a programming interface as a two-way causality flow: a constraint system. Instead of specifying a compiler, the programming interface specifies a constraint solver.

The traditional workflow proceeds as normal. When the programmer modifies the source code, these modifications propagate via the constraints to the running process. But alternatively, the programmer can manipulate the running process, and these modifications will back-propagate to modify the source code.

To illustrate, Context Free uses the one-way procedural programming model whereas Recursive Drawing uses the two-way constraint model.

For example, when working with a self-referential rule in Context Free, the programmer can modify the transformation under which the rule calls itself. Because the rule is self-referential, this initial transformation gets called on itself iteratively so as to produce different recursive effects. The only way to adjust the recursive effects is to adjust the initial transformation. However in Recursive Drawing, the programmer can modify a shape *at any depth in the recursion* (Figure 2). This modification then back-propagates to the rule definition which specifies the initial transformation. This feature is implemented as a constraint solver (in this case, with a numerical algorithm).

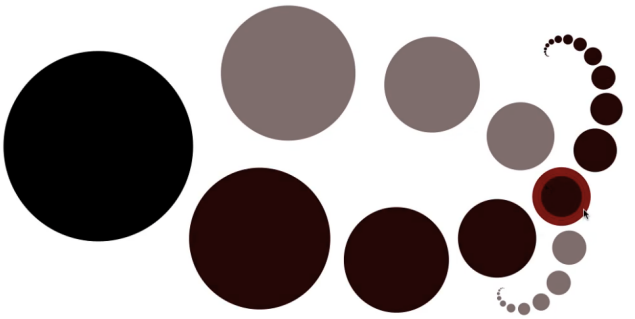

Figure 2. The programmer can modify a shape at any depth in the recursion. This modification back propagates to the rule definition.

It is important to note that users intuitively think of the common drag-and-drop convention as a constraint-based operation. When the user presses down the mouse button in preparation for dragging, she expects that the mouse pointer and the point she pressed on will remain constrained together. So when she moves the mouse, the object she is dragging moves with it. Recursive Drawing's constraint model is a generalization of this convention.

The major design challenge of programming interfaces as constraint solvers is providing the user with the power to

[3] "Constraints Generalize Procedures" is a section of "Building Robust Systems" by Gerald Sussman [21]. In this essay, Sussman discusses tracking the provenance of information (keeping track of the causality chain), and generalizing procedures to constraints so as to allow causality to flow in multiple directions.

specify what is constrained in the current context. With procedural programming interfaces, the programmer modifies a line of source code and expects every other line of source code to stay the same.[4] But if the programmer modifies the output, there may be multiple ways to change the source code in order to produce the new output. The programming interface must then either infer further, "natural" constraints, or the programmer must be able to manually specify further constraints so as to remain in control of her program.

Recursive Drawing currently solves this problem by only allowing the direct editing of the rule which is currently shown in the workspace. Thus it assumes that every other rule's definition is constrained to its current state. However I don't believe this is always the most "natural" constraint to impose. A further line of research would be to reimplement Recursive Drawing on a touch-screen interface. With this interface, the programmer could drag, or more accurately constrain, with multiple fingers at a time, allowing more expressive modifications to the program.

6. Programming: Construction or Transformation?

Much of the design process in programming is concerned with how information is represented: the model. The programmer carefully chooses the data structures with which the program is built. The programming language designer carefully chooses the primitive constructs with which programmers will build their programs. Both of these workflows reflect a reductionist mindset. Elements are defined by the smaller elements they are made up of. This forms an "abstraction pyramid", with primitives on the bottom and the finished piece of software on top.

The abstraction pyramid has served us well in the past. Reductionism allows us to reason at the relevant level of the pyramid. It can enable us to build quite complex software (tall pyramids) because we can build on previously laid foundations. But a reductionist mindset can also reduce our flexibility and produce cognitive dissonance when a piece of software is approached from a different perspective.

For example, users approach software differently than the creators of the software. Discrepancies inevitably arise between the model underlying the program and the model that forms in the user's mind. Many recognize these discrepancies as the root cause of usability issues [17]. Often it is argued that the model needs to be simplified–made more elegant and powerful–so that the user can more fully grasp it. This is often true but it misses a subtle issue. A creator of software is concerned with its reductionist nature–the pyramid of pieces it's made out of. But the user of software is concerned with what she can *do* with the software. That is, the user is only concerned with the aspects of the software which are *operationally relevant* in the context of a larger system [8].

The same applies to programmers approaching existing code. When we choose our representation for the program, we limit the ways in which we can easily modify the program. By "easily modify" I mean transforming the program without choosing new primitives–what programmers appropriately call "refactoring." This is why experienced developers think long and hard about the primitives they will use before they touch the keyboard.

Is there a way make refactoring cheaper? Is there an approach to program design that will not conceptually lock us in to the primitives we initially choose, so that we can open our minds to the various contexts in which our software might be used?

This is of course a deep challenge, but I believe we can better attack it with a shift in mindset.

I suggest we shift our focus from program construction to program transformation. Instead of concentrating on the primitives and what we can build from them, we concentrate on the transformations we might want to apply to our program in various contexts. In other words, I'm suggesting we think of programs *operationally* rather than *reductively*. We define a program by its relationships to other (potential) programs, not by the atoms which constitute it.

This is analogous to the shift in mindset from set theory to category theory. In set theory, we define a property of a set in terms of its elements. In category theory, we define a property of a set in terms of its relationships to other sets.

I'll provide two examples from Recursive Drawing. In each case, I will show how my initial design reflected a reductionist mindset, then how I rethought the issue from a transformation-centric mindset.

6.1 Relativity

The first example relates to how we traditionally use coordinate systems. In Context Free, the underlying representation consists of a hierarchy of coordinate systems. The nest-able coordinate system is a key primitive on which Context Free programs are built.

Now, recursively nested spatial transformations are intrinsic to the concept of Context Free, but their representation as coordinate systems is an implementation detail which is forced on the programmer. Indeed early versions of Recursive Drawing did the same thing. Every coordinate system was explicitly shown in the interface as arrows. Additionally, the user could only manipulate the drawing by manipulating the arrows. (Figure 3)

When we force an underlying representation on the programmer, *program transformations can only be performed with respect to that underlying representation*. Indeed this is the only way to tweak a Context Free Art program. The programmer must tweak a value which makes a change with respect to the coordinate system that the value lives in.

[4] This can be tedious for certain types of modifications, which is why modern IDEs relax this constraint by supporting advanced search-and-replace functionality

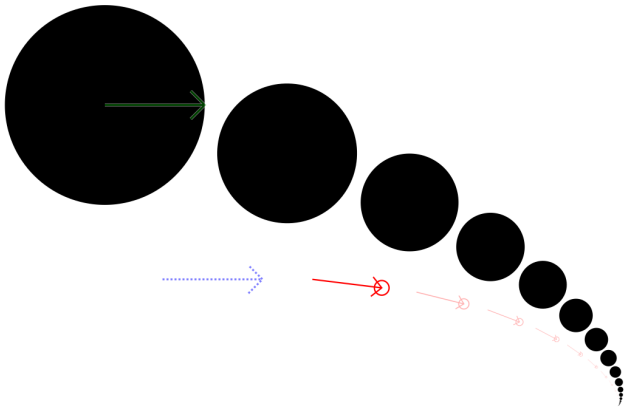

Figure 3. An early version of Recursive Drawing. Coordinate systems are explicitly shown as arrows in the interface.

But as Recursive Drawing's interface evolved, I found that it was more intuitive to tweak a shape by transforming it *with respect to the other shapes*. That is, it didn't matter where a shape was or how it was oriented with respect to the underlying coordinate system, it only mattered how it related to the other shapes. Thus Recursive Drawing's canvas–the model it presents to the programmer–has no center, orientation, or scale. Shapes are positioned, oriented, and sized with respect to each other.

6.2 Ontology

The second example relates to ontology: how we divide a program into separate objects, or equivalently, how we define identity. A reductionist mindset implies a fixed ontology. But in life, we can shift between contexts. Each context provides a different way to divide the world. Analogously, we would like our ontology to change depending on the context in which we're working with a program.

A heuristic we can use to produce an ontology in a given context is based on the transformations that the context supports. Given two things, A and B, if in our context every transformation we apply to A also uniformly applies the same transformation to B and vice versa, then A and B can be considered identical in that context.

To illustrate, say we have a rigid body like a coffee cup. It is unclear that this should be a single object if we're looking at it in the context of atoms or quantum clouds. However, in the context of everyday interactions, we can identify the coffee cup as a single entity. We determine this based on the transformations available in our everyday interaction context. If I transform the handle of the cup by lifting it two feet upwards, then the rest of the cup is also lifted two feet upwards. Rigidity by definition implies that a transformation on any given point of the object must apply uniformly to every other point on the object. In this way, an operational context–a set of allowable transformations–implies an ontology.

This principle was violated in early versions of Recursive Drawing. In an initial design, during editing, the highest level (the rule to be modified) and the lowest level (a primitive shape, circle or square) of the hierarchy were always highlighted when the programmer hovered her mouse over a shape (Figure 4, above). This was intended to show the relevant parts of the abstraction pyramid, to help the programmer comprehend the reductionist model. But in user testing, I found that users were confused about what shape would move when they performed a manipulation. An improvement was made when highlighting the lowest level was changed to highlighting *all* shapes which would transform uniformly if the user started dragging (Figure 4, below). If it moves the same, it is the same. This more closely mapped a user's intuition about what constituted a singular object.

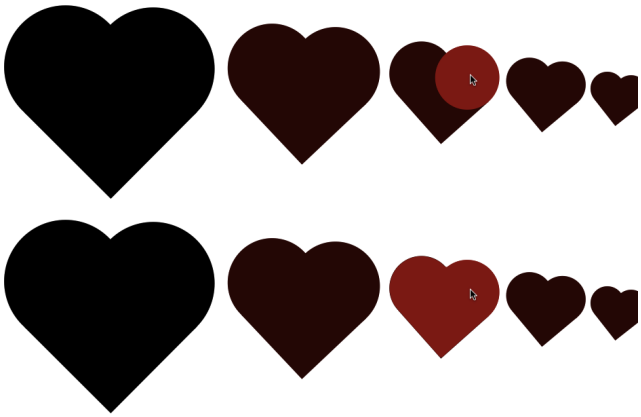

Figure 4. Above, highlighting based on the abstraction pyramid. Below, highlighting based on what will transform uniformly.

In each of these examples, when I started from a reductionist perspective, the primitives of the model determined the transformations that were available to the programmer. This could be seen as a pernicious form of representation exposure. In the alternate version, the transformations available in a given context were considered first. These transformations then implied an appropriate model to display to the programmer in that context.

7. Strengths and Weaknesses of Recursive Drawing

Context Free powerfully exploits recursion, allowing programmers to create complex graphics with surprisingly short and elegant code. Recursive Drawing extends this strength, encouraging experimentation with graphical co-recursive structures.

This experimentation can lead to unexpected insights which would be difficult to attain in a textual programming interface. For example, these are all insights I had into the mathematics of these shapes while playing with Recursive Drawing:

1. Seeing how a convergent transformation applied iteratively always converges to the same point despite its initial position. This is perhaps the graphical equivalent of saying if you keep dividing by 2 you will approach 0 no matter what number you start at. By dragging around the base case, I was able to gain a kinesthetic understanding of this principle. (Figure 5)

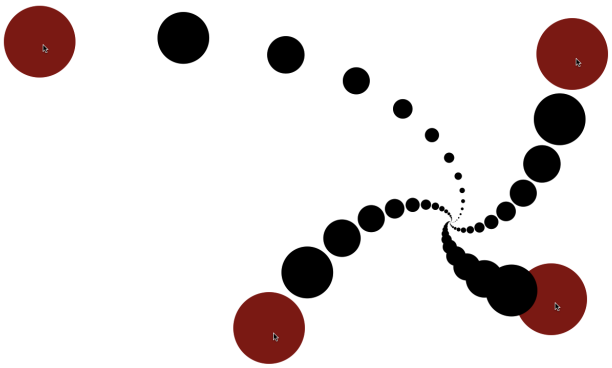

Figure 5. Convergent transformations always converge to the same point.

2. Seeing how spirals, with iterative rotation amounts close to exact divisors of 360 degrees, create second-order spirals. (Figure 6)

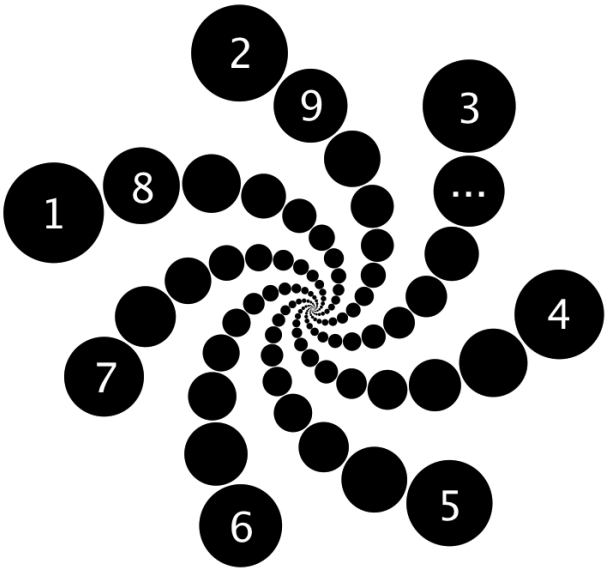

Figure 6. A single spiral creating second-order spirals.

3. Seeing how the Fibonacci series is exponential binary branching with one branch "carried up a level." (Figure 7)

Recursive Drawing has several weaknesses compared to Context Free.

The programmer loses some control over her program because Recursive Drawing does not expose the underlying

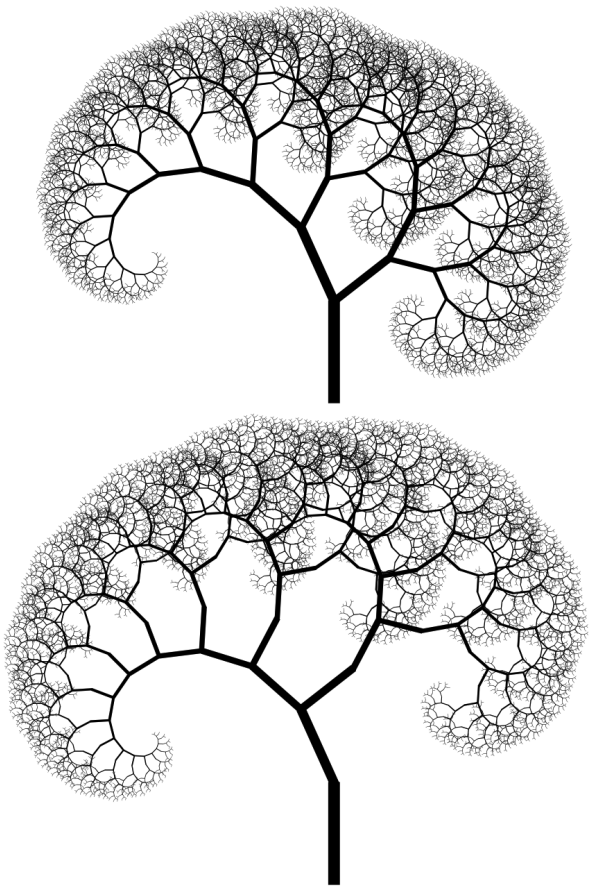

Figure 7. Binary tree (above) and Fibonacci tree (below).

representations of numbers and coordinate systems. For example, in Context Free, one can adjust a number manually, in an exact way so as to, for example, make one shape exactly 3 times larger than another. Further, by manually controlling the nesting of coordinate systems, it is possible in Context Free to specify an exact origin point around which a shape should rotate. By contrast, in Recursive Drawing in its current iteration, shapes can only be transformed approximately, and can only be rotated with respect to the origin of a primitive shape in the drawing. However, I am confident that this functionality can be adequately addressed in further iterations of Recursive Drawing by allowing the programmer to specify additional constraints, for example as is done in Sutherland's Sketchpad.

A more subtle problem is that the programmer loses control because she has less *conceptual distance* from her creation. This paper argues that textual interfaces may be inappropriate for certain types programming, but their weakness is also a strength in that they force the programmer to take a step back and consider her problem from a different perspective (in this case, from the traditional programming perspective). On the other hand, Recursive Drawing makes it easy to experiment wildly. This experimentation can cause

the creator to lose sight of her original vision. Directly manipulable interfaces of the future will need to find a balance in supporting both experimentation and meticulous control.

Directly manipulable interfaces thus share a problem with textual programming interfaces: they both make some things easier to do than others, and some ways of working easier than others. This can have repercussive influences on the programs we make and our conception of what programming is about. In order to more fully explore the universe of programming possibilities, we must always maintain vigilant awareness of this influence.

8. Conclusion

The physical, conceptual, and social dimensions of our programming interfaces reinforce our notions of *what programming is about*. The programming community self-selects for members who can think using the dominant paradigms of the community.

By reaching out to alternative programmers, who do not naturally think in these paradigms, we have the opportunity to transform the nature of programming. But to do so we will need to reconsider in a broad sense the nature of our programming interfaces.

In particular, our traditional programming interfaces are tightly coupled to the medium of text. They thus primarily engage the language center of our brain. While great thinkers and great programmers work on problems internally using their visual and kinesthetic intuition, these impulses must be filtered and processed through the symbolic manipulation part of the mind in order to communicate them to the computer.

Recursive Drawing is a reconsideration of the textual language Context Free using a graphical, directly manipulable interface. Direct manipulation means working with the program in a representation that closely resembles the output of program.

Two design principles were used in the creation of Recursive Drawing: the programming interface is thought of as a two-way constraint solver rather than a one-way compiler, and the interface focuses on the program transformations available in various contexts, rather than program creation from fixed foundational primitives.

Directly manipulable interfaces naturally encourage experimentation. This can lead to new insights and deeper understanding of the programming model. It can also distract from a programmer's original intention. Creators of directly manipulable interfaces will thus need to carefully balance experimentation and control, and maintain awareness of the influences a programming interface can have on the programs we create.

Acknowledgments

Thanks to my test users, whose discoveries and frustrations pointed the way in the development of Recursive Drawing. Thanks to my thesis advisor, Nancy Hechinger, for comments on drafts of this paper and encouragement.

References

[1] Arduino. URL `http://www.arduino.cc/`.

[2] Github. URL `https://github.com/`.

[3] vvvv - a multipurpose toolkit. URL `http://vvvv.org/`.

[4] Field, 2008. URL `http://openendedgroup.com/field`.

[5] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Mit Electrical Engineering and Computer Science Series. Mit Press, 1996. ISBN 9780262011532. URL `http://books.google.com/books?id=2p887sEWtAYC`.

[6] B. Atkinson. Hypercard, 1987.

[7] M. Coniglio. Isadora. URL `http://www.troikatronix.com/isadora.html`.

[8] A. Cooper. *The Inmates Are Running the Asylum*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1999. ISBN 0672316498.

[9] C. Coyne, M. Lentczner, and J. Horigan. Context free, 2005. URL `http://www.contextfreeart.org/`.

[10] D. C. Engelbart. Augmenting Human Intellect: A Conceptual Framework. Technical report, Air Force Office of Scientific Research, 1962.

[11] R. Fiebrink. *Real-time Human Interaction with Supervised Learning Algorithms for Music Composition and Performance*. PhD thesis, Princeton University, Princeton, NJ, USA, January 2011.

[12] B. Fry and C. Reas. http://processing.org/. URL `http://processing.org/`.

[13] D. H. Hansson. Ruby on rails. URL `http://rubyonrails.org/`.

[14] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. ISSN 0362-1340. doi: 10.1145/130697.130699. URL `http://doi.acm.org/10.1145/130697.130699`.

[15] A. Kay. Doing with images makes symbols: Communicating with computers, 1987. URL `http://archive.org/details/AlanKeyD1987`.

[16] P.-O. Latour. Quartz composer, 2005.

[17] D. Norman. *The Design of Everyday Things*. Basic Books. Basic Books, 2002. ISBN 9780465067107. URL `http://books.google.com/books?id=w8pM72p_dpoC`.

[18] S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980. ISBN 0-465-04627-4.

[19] M. S. Puckette. Max, . URL `http://cycling74.com/products/max/`.

[20] M. S. Puckette. Pure data, . URL `http://puredata.info/`.

[21] G. J. Sussman. Building robust systems an essay. Technical report, 2007.

[22] I. E. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA, 1963. ACM. doi: 10.1145/1461551.1461591. URL http://doi.acm.org/10.1145/1461551.1461591.

[23] B. Victor. Dynamic pictures, 2011. URL http://worrydream.com/DynamicPicturesMotivation/.

[24] B. Victor. Inventing on principle, 2012. URL http://vimeo.com/36579366.