Why Can't Programming Be Like Sketching?

Clayton Lewis
Coleman Institute for Cognitive Disabilities and Department of Computer Science
University of Colorado Boulder
Boulder, Colorado, USA
clayton.lewis@colorado.edu

ABSTRACT

Many people compare the labor of programming unfavorably with the freedom of artistic expression. Can a form of programming be developed that has some of this freedom, using sketching as a model of expression?

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Software and its engineering → Software notations and tools**.

KEYWORDS

Programming, sketching, thought experiment

ACM Reference Format:
Clayton Lewis. 2019. Why Can't Programming Be Like Sketching?. In *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19), April 1–4, 2019, Genova, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3328433.3338060

1 INTRODUCTION

At a joint meeting of the Psychology of Programming Interest Group and the Art Workers Guild (London, September 5-7, 2018), Charlie Gere asked, why can't programming be like sketching? The ambiance of the meeting included testimonies from Guild members that cast computing, and programming, as repellent, in the literal sense, an activity that people would like to avoid, even it if it is useful. "Sketching" in the question stands for another kind of activity, lacking these repellent qualities, and having the attractive qualities of enjoyable expression. Can programming be like that?

There are many aspects of sketching, and of programming, so any preliminary inquiry must be incomplete, and selective. Some key aspects of the question have already been raised in the critiques of programming offered by Basman and collaborators, using the broad field of craft for perspective [2–5, 7, 9]. My aim is to push a little further into particular issues, and possibilities, raised specifically by using sketching as a model for programming. The discussion will be wholly speculative; the ideas will be illustrated with hand drawings, not by output from an implemented system.

Sketching as programming, or programming as sketching? Work on procedural art brings programming tools to artists, as for example in the Processing language [13]. Thus this work spreads the character of programming into sketching, rather than spreading the character of sketching into programming. I take it that Gere is calling for the latter, not the former. (Some recent work blurs this distinction, however. Jennifer Jacobs and collaborators [15, 16] create tools that use gesture-based interactions, like sketching, to create constraints and procedural forms. In Jacobs' work, these computational structures are used to produce drawings, rather than programs. But it would be interesting to explore how these drawings could be interpreted as programs, permitting Jacobs' tools to be used to program, rather than to draw.)

Sketching as art vs sketching as craft. Collingwood [10] argues that art is distinguished from craft by not being driven by a preexisting plan or goal. While it is easy to quarrel with this definitional boundary, it nevertheless points to a meaningful distinction. Creating something like an engineering drawing, even in its early, perhaps somewhat vague, stages, can involve any amount of deliberation, mastery and use of specific representational conventions, and the like. I take it that Gere is not asking for programming to be more like this kind of activity; indeed, it is already rather like it. So I'll focus here on sketching as art, in which the sketcher is working out an expression as they go, rather than deliberately elaborating a plan.

Collingwood's distinction can be related to the distinction sometimes made between exploratory programming and other forms, as articulated early on by Shiel [24]. The concept arose among AI programmers; as Norvig [20, p 119] says, "AI programing is largely exploratory programming; the aim is often to discover more about the problem area rather than to meet a clearly defined specification. This is in contrast to a more traditional notion of programming, where the problem is completely specified before the first line of code is written." This form of programming isn't driven by a plan, but nevertheless there is a goal, in some form. Likely many works of art similarly start with something to express, even if no details are fixed.

Sketches as static vs sketches as dynamic. We commonly think of programs as dynamic, as describing or specifying a trajectory in time (functional programming people may dissent.) Interestingly, many artists and thinkers stress the dynamic content of sketches. For example, a famous sketchbook of Paul Klee [17] explicitly refers to lines as dynamic, representing the motion of something, rather than static configurations, showing a curved line as "An active line on a walk, moving freely, without goal. A walk for a walk's sake. The mobility agent is a point, shifting its position forward." So by emphasizing this aspect of sketching we may be within reach of some forms of programming.

Centrality of gesture. Drawings can be created by many means, some highly technical and constrained, but sketches are created by gestures of the hands. Cooper [11] suggests a tight link between gestures and creative thought, considering specifically the role of gesture in architectural design. The link is framed by embodied cognition [26], the view that abstract thought is constructed from elements grounded in body movement and kinesthetic sense. I take it that the role of gesture, and perhaps the connection between gesture and thought, is an important part of the appeal Gere sees in sketching.

Dynamics as implicit in form. A tradition traceable to Goethe [27] finds that the dynamics of a form can be seen in the form itself. Proper observation of the parts of a plant, for example, coupled with an understanding of the patterns of morphogenesis in plants, reveals the future development of the plant, along with its history.

Unfortunately, Goethe did not articulate this scheme very fully. Anderson [1] has developed the idea in recent writing on her work as an artist in science, as part of her program on Isomorphology. She identifies common symmetry classes in biological and mineral forms, and these certainly say something about the dynamics of the forms. But she does not articulate a complete space of dynamics. For example, she does not differentiate between different branching patterns, such as alternate and opposite; see Figure 1. Nevertheless, we can take the idea of dynamics as implicit in forms as a starting point for one (of very many) possible exploration of sketching as programming.

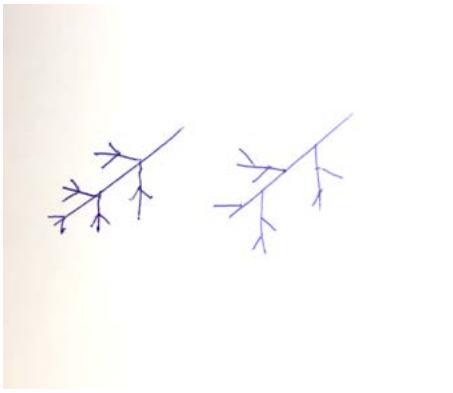


Figure 1: Opposite (left) and alternate (right) branching patterns.

2 A SKETCH OF A SKETCHING SYSTEM

Suppose a system allows the artist to draw **lines** and **buds**, and can distinguish the two. Further, the system can distinguish various forms of buds. All buds are drawn as small closed curves, but can be of different shapes.

A sketch evolves spontaneously, as follows.

> Lines extend themselves in the way in which they are drawn. Wavy lines extend as wavy lines, and rapidly drawn lines extend rapidly. The artist can stop the evolution of a line by tapping near its growing end.
>
> Buds are drawn on the ends of lines (and stop the evolution of the lines). The line a bud terminates what we'll call its stem. If no further lines are drawn from a bud, the bud does not evolve. If one or more lines are drawn from a bud, the bud will branch. These lines can have buds drawn on them, which could be of the same or different kinds as the parent bud. These lines, and their buds if they have them, form the branching pattern for the parent bud. A bud that has a branching pattern, but has not already grown branches, grows lines and other buds as seen in its branching pattern.

Figure 2 shows an example. There are two kinds of buds, with different branching patterns, producing a tree that grows in a particular way: the numbers of buds in each generation form a Fibonacci sequence: 1,2,3,5,8,..., with each term being the sum of the two previous.

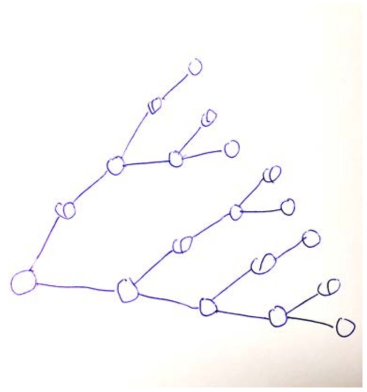


Figure 2: Branching structure generated with two kinds of buds.

2.1 A tiny corner of programming

This system may seem to have nothing to do with programming, but in fact it covers part of the domain of L-systems [1], used to describe the algorithmic structure of plants, and also other fractal structures, such as the Koch snowflake. We can compare L-systems with our sketching system to see if we have made any progress in Gere's direction.

An L-system that corresponds to the example in Figure 4 consists of two symbols (A and B) and two rules:

A ⟶ AB

B ⟶ A

Starting with A, and applying the rules where possible, we get the derivation

A

AB

ABA

ABAAB

ABAABABA

...

[1]https://en.wikipedia.org/wiki/L-system

We can see how these structures relate to the tree in Figure 2, as shown by the labeling in Figure 3.

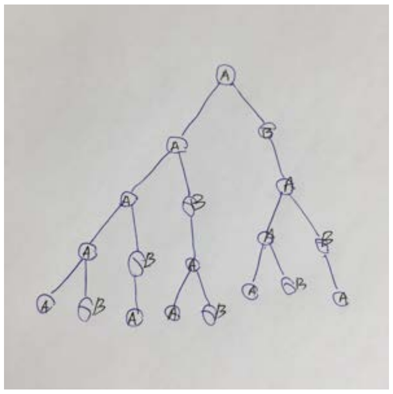


Figure 3: Tree from Figure 4 labeled to show correspondence with L-system.

How does the sketch system compare with the L-system, as a way of creating patterns? Everything is done in the sketch system by drawing. A drawing expresses within itself the dynamics of its growth. The L-system requires writing rules, which are distinct from the structure being generated.

While the sketch implies the rules, in the sense that the rules needed for a corresponding L-system are determined by the sketch, drawing the sketch does not require thinking about the rules, or even being aware of them. There's nothing but the sketch.

How the sketch system compares to other visual programming models. The fact that there is nothing but the sketch differentiates the sketch system from other visual programming systems, including ones based on rules, like AgentSheets [22] or ChemTrains [8]. In a typical visual programming system like Scratch [19], programs are represented visually, as sequences of blocks. These blocks do not represent entities in the domain in which the program operates. Similarly, rules in Agentsheets contain many entities outside the domain being represented.

In ChemTrains, rules are pairs of drawings, representing before and after states of some situation. Yet even though all this material is drawn, and even though all of the entities in the drawings belong to the situation being modeled, the drawings have a role and interpretation that is separate from the situation. The rules stand outside that situation, and are created and edited separately.

2.2 More of programming

How does the sketch system measure up to simple tasks typical of "real" programming, like list reversal? We have to note that even thinking of these things may put us on the wrong side of Collingwood's distinction: we'd be starting with a goal when we sit down to sketch. Trying to beg that issue, can we imagine that someone idly sketching could produce a list reversal, making no assumption about what (if anything) they were trying to do?

In the simplest case, yes. Figure 4 shows a sketch of a reversal of three elements, represented as colored lines. But what if you wanted to reverse a list of different items, or a different number of items? The things in the list, and the size of the list, are built into the sketch.

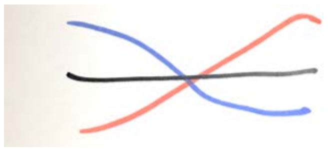


Figure 4: A sketch that reverses a sequence of three colors.

This simple problem, typical of "programming", requires facilities not present in the sketch system:

- Separating roles and values, so that one can put different values into the same roles (list positions, in this example)
- Abstracting over numbers of things

2.3 Adding vines and ribbons

To deal with these issues we allow the artist to draw **vines**, a special kind of line, of various colors. Vines have a special dynamic: rather than being drawn by the sketcher, they grow along nearby lines. Klee considers lines related in this way; showing a smooth curve accompanied by a line that wraps around it. Figure 5 shows how a sequence of vines, say red, blue, black, can be reversed. By removing these vines and placing other ones, a different sequence of three vines can be reversed. That is, the vines specify values, separate from the lines, that specify roles.

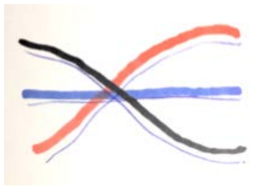
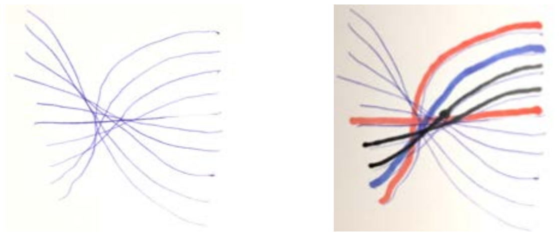


Figure 5: A sketch that reverses a sequence of vines. The vines grow along previously drawn lines, shown in thin blue ink.

Dealing with different numbers of vines can be done in more than one way. If one is content with small numbers of vines, the lines in Figure 6a can be used to reverse any sequence of ten vines or fewer, as shown for example in Figure 6b.

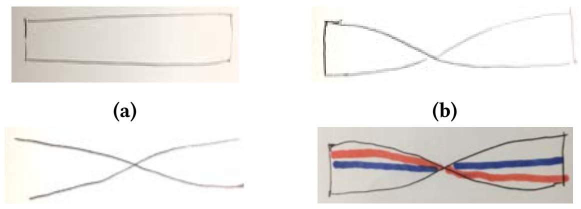
An alternate approach broadens the domain of sketched structures to include **ribbons**, as shown in Figure 7a. A ribbon can be twisted, as shown in Figure 7b. Klee suggests that when lines form closed figures they define not only themselves, having linear character, but also a planar region. In this way two lines that cross, as shown in Figure 7c, are distinguished from the twisted ribbon in Figure 7b. Vines follow ribbons, as they do lines, as shown in Figure 7d.

What should a vine do when it encounters a branch? A natural dynamic is to divide, following both branches, as shown in Figure 8a. Similar behavior can be proposed for ribbons, as shown in Figure 8b.



(a) (b)

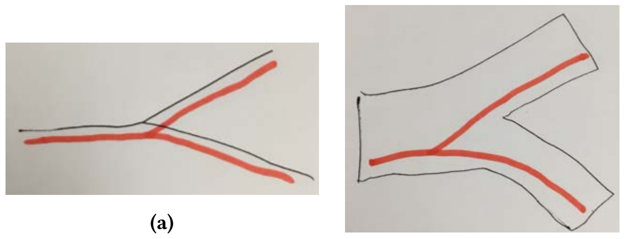
Figure 6: (a) Sketch for reversing up to 10 vines; (b) shown reversing five vines



(a) (b)

(c) (d)

Figure 7: Ribbons and some lines and vines.



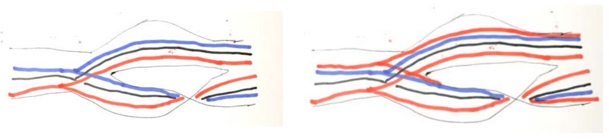
(a)

(b)

Figure 8: A vine branching on a line (a) and a ribbon (b).

2.4 A worked example: Producing palindromes

We can now sketch a more complex example, that uses list reversal as a subpart. Figure 9a shows how a palindromic sequence of vines can be created (that is, one that reads the same forward and backwards). Figure 9b shows that the same system of ribbons can produce palindromes of different lengths.



(a) (b)

Figure 9: Sketch producing a palindrome of length 6 (a) and 8 (b). The underlying ribbons copy the initial pattern of vines (the ribbon splitting does that), then reverses one of the copies (done by the twist), then concatenates the twisted and untwisted copies (where the ribbons join.)

Using this program as an example, we can take stock of where we are with respect to Gere's request, or at least our interpretation of it. We can claim two successes:

- The sketch is created by drawing, using gestures.
- The artist/programmer does not create a specification of the behavior of the sketch separate from the sketch itself.

We can also pause to consider the relationship between our attempt and another approach to rethinking programming: programming by demonstration [12]. Our earlier branching tree program can be seen as a form of programming by demonstration: one demonstrates how buds should evolve by drawing their evolution. But the palindrome sketch does not have that character. We don't have to demonstrate how to construct a palindrome, or even how to reverse a sequence, to create this program.

This contrast arises because our sketch program uses forms that have their evolution prespecified. We don't have to say how vines respond to twisted ribbons; they do what they do.

There are two tradeoffs in play in this distinction. First, the artist/programmer has to understand, or be able to learn about, the prespecified behaviors. In a pure demonstration system, on the other hand, there are no such behaviors that have to be understood. Second, the expressive scope of the sketch system is constrained by the repertoire of prespecified behaviors.

One might think that the free expressiveness we may associate with sketching shows that there aren't prespecified behaviors to learn in that activity. But as Klee [17], Gombrich [14] and many others show, this is far from the case. Learning to sketch requires mastering, at least implicitly, a large number of new and unnatural techniques, and also understanding what visual impressions are produced by particular kinds of marks. Klee's observation about the effect of a closed figure of lines in indicating a planar form, which we have made use of, is an example: if one doesn't understand that, one's sketches are apt to miscarry.

Restricted expressive scope may contribute to some of the virtues that Basman and collaborators associate with craft. It's characteristic of programming that the most minute change in a program can produce arbitrarily large changes in behavior. A schematic argument suggests that this must be true, if one insists on a very large and diverse collection of different expressive forms, accessible with a modest amount of work. Suppose we idealize programs as sequences of symbols, drawn from some alphabet. Pick two programs, both no more than n symbols in length, producing behaviors B0 and B1. There is a sequence of single symbol changes, no more than n steps in length, that connects the two programs. Somewhere in this sequence of programs there must be a difference in behavior of at least one nth of the difference between B0 and B1. If this difference overall difference is very large, and n isn't very large, we have to expect that some small changes in programs will produce large changes in behavior.

With this in mind, we can see virtue in working in a medium in which expressiveness is constrained. But at the same time the constraints are real, and will be unacceptable in many real situations.

Owen Lewis [18] suggests another perspective on these matters. Learning to program a Turing machine is conceptually very easy, in the restricted sense that very few concepts are involved; but deploying the concepts is extremely difficult (cf Perlis [21] on the

Turing tarpit, where "everything is possible, but nothing of interest is easy.") On the other hand, a natural language can take many years to learn, but allows easy, natural expressions of many things, because of its rich vocabulary. This suggests a tradeoff between investing in vocabulary acquisition, and having to work hard at puzzle solving, to express things in a very limited vocabulary. Can the vocabulary of sketching be leveraged to beat this tradeoff, at least for people who know how to sketch? Or would the vocabulary of sketching need to be augmented with indefinitely many new conventions that must be learned?

2.5 Other domains of expression

To consider more situations in which "programming" may be wanted, take calculating a weighted average. Could a sketch do this? An immediate challenge is to introduce numbers into sketches, and, having done so, to provide a useful range of operations on them. Something useful might be done in a dataflow mode, offering nodes that carry out operations on numbers that flow along lines, vines, and ribbons. The twist and copy operations imagined for now are quite limited, in that the operations can't depend on particular values, one could add filters that would support data dependencies. For example, a filter placed on a ribbon could block the passage of any vines other than those of certain colors, or numbers outside a certain range.

More interesting could be representing numbers as geometric magnitudes, with areas representing products. From this point of view the contributions to a weighted average can be seen as rectangles, of different heights and widths, and the weighted average itself as the height of a single rectangle, whose area is the sum of the areas of the contributing rectangles, and whose width is the sum of their widths. One could imagine this operation being performed by allowing fluid levels in a collection of rectangles to equalize. Could the plumbing to do this be added to the sketch system?

Another domain of expression is simulation, as is often used in educational settings. Figure 10 shows a very simple example from the design work on ChemTrains (described in [23]); the program has to show how the state of the water in the beaker changes from solid (the ice cube, as shown) to liquid and then gas as the burner control in manipulated.

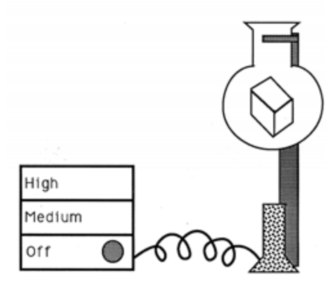


Figure 10: A very simple simulation.

There's an immediate challenge in doing anything of this kind in the sketch system as we've conceived it. Things in a simulation should look like the things they represent. But in the sketch system, the appearance of things determines their behavior. That is, in the sketch system, behavior is determined by appearance, so needed behavior constrains appearance, and so appearance can't be freely chosen to resemble real things

A way forward would be to loosen the connections between appearance and behavior. Carrier nodes could be defined, that move along lines, vines, and ribbons. A carrier could have a form with any appearance attached to it, that would then move with the carrier. There could be provision for hiding things in a sketch, while retaining their influence, so that an invisible carrier could move a visible object of arbitrary appearance along an invisible line.

Another challenge is conditionality. Many programming applications require different things to happen under different conditions. For example, the water in the bunsen burner simulation has to change appearance based on the state of the flame (not shown in the figure, since the burner is off.) The flame, in turn, depends on the setting of the burner control. The sketch system as conceived so far has no way to manage dependencies like those. Can the reader think of a way to convey these relationships within the sketch, rather than in separate rules?

2.6 Sketchiness

Cooper suggests that the value of sketching in creative thought is tied to its controlled vagueness (as also discussed by Tversky [25]). One form of vagueness uses multiple strokes to convey a family of possibilities [11], while another distinguishes lines of different style (heavy vs light, perfectly straight vs slightly wavy) to represent things that are definitely presented or just suggested in a general way. Can our sketch system behave in this way? As defined so far, it does not: a given kind of bud will definitely produce a certain kind of branch, or not. But we can imagine that the system could respond to multiple strokes and to stroke style, in way that would influence the evolution of forms. For example, a stroke drawn heavily, and perfectly straight, could be reproduced exactly, while a light, wavy stroke might be reproduced only approximately. Very many branches drawn from a bud could represent a probabilistic branching pattern (as supported in some variant L-systems) so that similar buds would branch in similar but not identical ways. How much such features would endow the sketch system with useful and pleasing sketchiness would have to be judged from experience with an implementation.

2.7 On to the ecosystem problem

Programs commonly function in an environment that includes many other programs, a large software ecosystem. To work, they have to be able to represent information in ways that other programs can work with, and to accept information as represented by other programs. A weighted average sketch would be quite useless without a means of getting numerical data from other programs, and giving it back to them.

As Basman has pointed out (personal communication; see also discussion of "walled gardens" in [6]), a common failing of innovative programming systems is that they presume that all the software with which they must deal is written within the new system. But this is just not realistic, given the vast amount of software that is in general use, and its complexity.

So we face a choice with our sketch system. We can conceive of it in a software world of its own, used only for those purposes that do not hinge on its connection to other software. Such worlds exist, and can be useful. For example, many children create Scratch programs, or Minecraft programs, and enjoy doing so, despite not having a way to connect their programs to anything else.

Presumably Gere would want us to push farther, if we can. Is there a way to link the expressiveness of sketching to the wider world of software as it is? To what extent would the existence of those links color the experience of the sketcher? Can this coloring be attractive, and not repellent?

ACKNOWLEDGMENTS

I thank the Coleman Institute for Cognitive Disabilities of the University of Colorado for research support. Owen Lewis made valuable suggestions.

REFERENCES

[1] Gemma Anderson. 2017. *Drawing as a Way of Knowing in Art and Science.* Intellect.
[2] Antranig Basman. 2016. Building Software is Not [Yet] a Craft. In *Proceedings of the 27th Annual Meeting of the Psychology of Programming Interest Group (PPIG 2016).*
[3] Antranig Basman. 2017. If What We Made Were Real: Against Imperialism and Cartesianism in Computer Science, and for a discipline that creates real artifacts for real communities, following the faculties of real cognition. In *Proceedings of the 28th Annual Meeting of the Psychology of Programming Interest Group (PPIG 2017).*
[4] Antranig Basman, Luke Church, Clemens Klokmose, and Colin Clark. 2016. Software and how it lives on - embedding live programs in the world around them. In *Proceedings of the 27th Annual Meeting of the Psychology of Programming Interest Group (PPIG 2016).*
[5] Antranig Basman, Colin Clark, and Clayton Lewis. 2015. Harmonious Authorship from Different Representations (Work in Progress). In *Proc. PPIG 2015 Psychology of Programming Annual Conference.*
[6] Antranig Basman, Clayton Lewis, and Colin Clark. 2018. The Open Authorial Principle: Supporting Networks of Authors in Creating Externalisable Designs. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018).* ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/3276954.3276963
[7] Antranig Basman, Philip Tchernavskij, Simon Bates, and Michel Beaudouin-Lafon. 2018. An Anatomy of Interaction: Co-occurrences and Entanglements. In *Programming'18 Companion - Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming.* 188–196.
[8] Brigham Bell and Clayton Lewis. 1993. ChemTrains: a language for creating behaving pictures. In *Proceedings 1993 IEEE Symposium on Visual Languages.* 188–195. https://doi.org/10.1109/VL.1993.269595
[9] Colin Clark and Antranig Basman. 2017. Tracing a Paradigm for Externalization: Avatars and the GPII Nexus. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17).* ACM, New York, NY, USA, 31:1–31:5. https://doi.org/10.1145/3079368.3079410
[10] Robin George Collingwood. 1958. *The Principles of Art.* Oxford University Press.
[11] Douglas Cooper. 2018. Imagination's hand: The role of gesture in design drawing. *Design Studies* 54 (2018), 120–139. https://doi.org/10.1016/j.destud.2017.11.001
[12] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I Do: Programming by Demonstration.* MIT Press.
[13] Ben Fry and Casey Reas. 2004. Processing. (2004). http://processing.org
[14] Art Gombrich. 1960. Illusion: A Study in the Psychology of Pictorial Representation. *New York: Bollingen Foundation and Pantheon Books* (1960), 34–38.
[15] Jennifer Jacobs, Joel R Brandt, Radomír Meĕh, and Mitchel Resnick. 2018. Dynamic Brushes: Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems (CHI EA '18).* ACM, New York, NY, USA, D316:1–D316:4. https://doi.org/10.1145/3170427.3186492
[16] Jennifer Jacobs, Sumit Gogia, Radomír Měch, and Joel R Brandt. 2017. Supporting Expressive Procedural Art Creation Through Direct Manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17).* ACM, New York, NY, USA, 6330–6341. https://doi.org/10.1145/3025453.3025927
[17] Paul Klee and Sibyl Moholy-Nagy. 1953. *Pedagogical sketchbook.* Faber & Faber London.
[18] Owen Lewis. 2019. Personal communication. January 8, 2019.
[19] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 16.
[20] Peter Norvig. 1992. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann.
[21] Alan J Perlis. 1982. Special feature: Epigrams on programming. *ACM Sigplan Notices* 17, 9 (1982), 7–13.
[22] Alexander Repenning. 1993. *Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments.* Ph.D. Dissertation. University of Colorado at Boulder.
[23] John Rieman, Brigham Bell, and Clayton Lewis. 1990. ChemTrains Design Study Supplement; CU-CS-480-90. (1990).
[24] Beau Sheil. 1986. DATAMATION®: POWER TOOLS FOR PROGRAMMERS. , 573–580 pages. https://doi.org/10.1016/b978-0-934613-12-5.50048-3
[25] Barbara Tversky. 2015. On Abstraction and Ambiguity. In *Studying Visual and Spatial Reasoning for Design Creativity.* Springer Netherlands, 215–223. https://doi.org/10.1007/978-94-017-9297-4_13
[26] Francisco J Varela, Evan Thompson, and Eleanor Rosch. 2017. *The Embodied Mind: Cognitive Science and Human Experience.* MIT Press.
[27] Johann Wolfgang von Goethe. 2009. *The Metamorphosis of Plants.* MIT Press.