Sketch-n-Sketch: Output-Directed Programming for SVG

Brian Hempel **Justin Lubin** **Ravi Chugh**
Department of Computer Science, University of Chicago, USA
{brianhempel,justinlubin,rchugh}@uchicago.edu

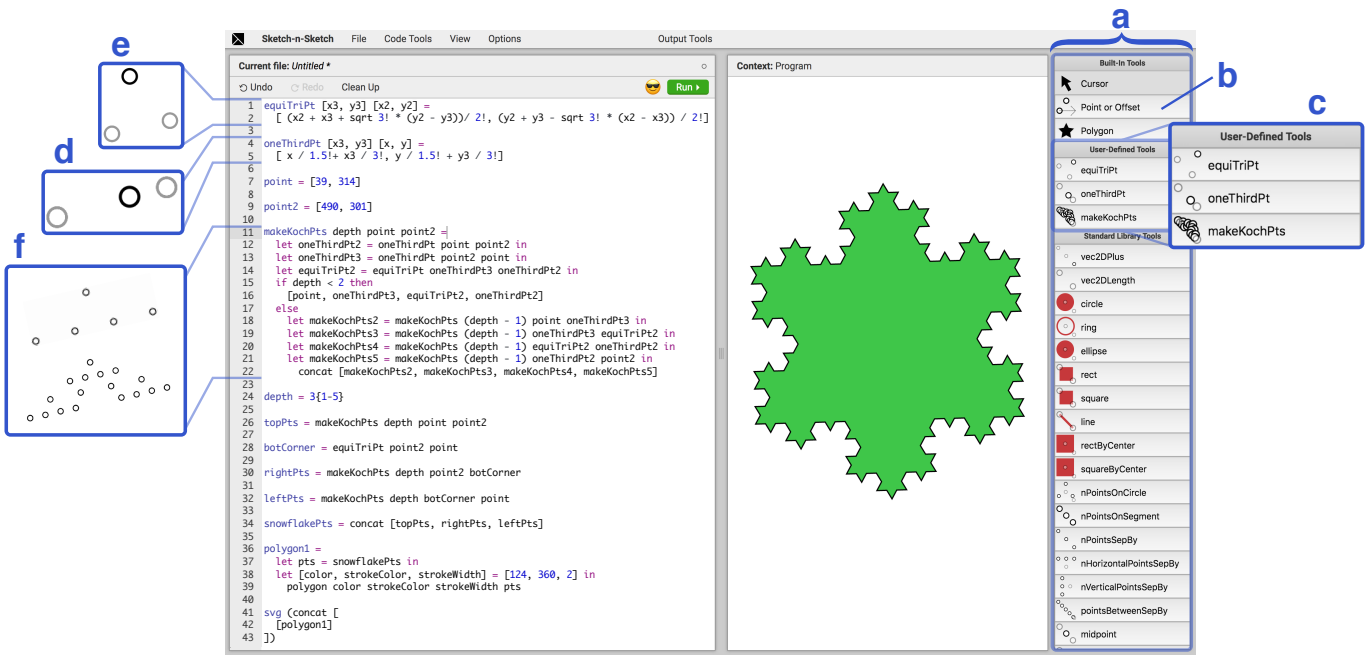


Figure 1. A recursive program built through output-directed program transformations, without any text-based programming.

ABSTRACT

For creative tasks, programmers face a choice: Use a GUI and sacrifice flexibility, or write code and sacrifice ergonomics?

To obtain both flexibility *and* ease of use, a number of systems have explored a workflow that we call *output-directed programming*. In this paradigm, direct manipulation of the program's graphical output corresponds to writing code in a general-purpose programming language, and edits not possible with the mouse can still be enacted through ordinary text edits to the program. Such capabilities provide hope for integrating graphical user interfaces into what are currently text-centric programming environments.

To further advance this vision, we present a variety of new output-directed techniques that extend the expressive power of SKETCH-N-SKETCH, an output-directed programming system for creating programs that generate vector graphics. To enable output-directed interaction at more stages of program construction, we expose intermediate execution products for manipulation and we present a mechanism for contextual drawing. Looking forward to output-directed programming beyond vector graphics, we also offer generic refactorings through the GUI, and our techniques employ a domain-agnostic provenance tracing scheme.

To demonstrate the improved expressiveness, we implement a dozen new parametric designs in SKETCH-N-SKETCH *without* text-based edits. Among these is the first demonstration of building a recursive function in an output-directed programming setting.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

UIST'19, October 20–23, 2019, New Orleans, LA, USA

© 2019 Copyright held by the owner/author(s).
Publication rights licensed to ACM. ISBN 123-4567-24-567/08/06…$15.00
DOI: http://dx.doi.org/10.475/123_4

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces; D.2.6. Software Engineering: Programming Environments; D.3.3. Programming Languages: Language Constructs and Features

Author Keywords

Output-Directed Programming; SVG; Sketch-n-Sketch

INTRODUCTION

Direct manipulation [40] graphical user interface (GUI) applications are ubiquitous. Every day, direct manipulation GUIs are used by millions for office tasks such as presentation and document preparation as well as for specialized creative tasks such as engineering and graphic design. Direct manipulation's intuitive *point-click-modify* experience enables a large number of people to leverage computers to create novel artifacts.

Even so, experts may sometimes forgo a GUI application and create their artifact using a programming language. In contrast to the intuitive experience offered by direct manipulation GUIs, text-based coding does not offer immediate and direct artifact construction, but programmers use text-based languages to gain flexibility not afforded by any GUI application.

Both paradigms—direct manipulation and text-based programming—have proven useful. Naturally, there have been efforts to combine their distinct strengths.

For example, some professional creative applications do offer a scripting API (e.g. Maya [2]), but, beyond perhaps an initial macro recording, script construction is a text editing process with no direct manipulation support. Furthermore, in the programming world, non-text paradigms such as blocks- [4] or wires-based visual programming [43] allow direct manipulation of program elements. Even so, most professional programming remains a strictly plain text activity.

Instead, could direct manipulation *augment* text-based programming, providing user interfaces more like standard creative application GUIs, such as graphics editors? Rather than directly manipulating program elements construed as blocks and wires, could one construct a textual program by directly manipulating a program's *output*?

Output-Directed Programming (ODP)

We refer to this paradigm—in which a programmer constructs a plain text program by mouse-based operations on the program's output—as *output-directed programming (ODP)*. Each operation triggers a transformation on the text-based program, akin to automatic refactoring tools. Successive operations build the program step by step. Because the program is plain text throughout the process, desired changes that are not possible through the provided direct manipulation interactions can be enacted by ordinary text editing. The system treats the plain text as the artifact's primary representation so that text editing does not disable future direct manipulation actions.

This ODP paradigm for altering a program by manipulating its output offers two tantalizing possibilities:

- Creative applications could represent the user's artifact not as an opaque internal data structure but as a visible, editable, textual program. Direct manipulation of the artifact can be freely mixed with computational generation.
- For general-purpose programming, ODP could facilitate *general* program construction via output-directed manipulations. For experts, these ODP interactions might accelerate common program construction tasks, and for novices, ODP interactions might provide an approachable pathway into programming.

A number of systems have made initial steps towards realizing this paradigm. After first drafting a program using ordinary text edits, several ODP systems allow direct manipulation of the output to change constant literals in the program [46, 9, 24, 39, 29]. To also relieve the initial text editing burden, for programs with graphical output a few systems provide ODP mechanisms for program construction [31, 38, 19], akin to drawing tools in graphics editors.

All prior ODP systems presented their workflow as a mix of text edits and direct manipulation. Although the selling point of ODP is that text edits can compensate for any missing direct manipulation features, the standing question is how thoroughly direct manipulation can subsume text-based editing. Therefore, in this work we wonder: **What kinds of programs can be constructed *entirely* through output manipulations?**

New ODP Techniques in SKETCH-N-SKETCH

We extend our prior work on SKETCH-N-SKETCH [9, 19], a programming system for creating vector graphics, with new ODP techniques that enable the system to construct 16 example programs *without* text edits on the code—even though ordinary text editing remains possible at any time during the construction of each example. Specifically, we:

1. Expose *intermediate execution values* for manipulation, instead of just the final output.
2. Offer *expression focusing* to enable contextual drawing.
3. Expose *generic code refactoring tools* through output-directed interactions.
4. Use *generic* run-time tracing to track *value provenance*, to associate output values with source code locations.

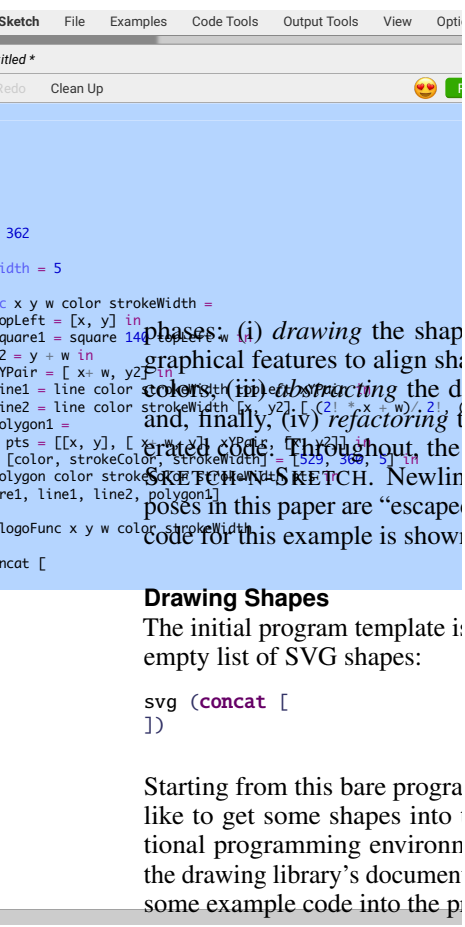
Although the SKETCH-N-SKETCH system is specialized for programs that output vector graphics, the four principles above are relevant for future output-directed programming systems of all kinds.

Paper Outline and Supplementary Materials

To introduce the output-directed programming workflow in SKETCH-N-SKETCH, in the next section we walk through the creation of a simple example program. After this overview, we present the core ODP mechanisms and our four innovations in more detail, followed by an examination of the construction of the 16 example programs. We conclude by discussing current limitations of SKETCH-N-SKETCH, related work, and future directions for ODP. Our implementation, examples, videos, and appendices are available as Supplementary Materials and on the Web (http://ravichugh.github.io/sketch-n-sketch/).

OVERVIEW

SKETCH-N-SKETCH is a bimodal programming environment, as depicted in Figure 1: the left pane is an ordinary source code text editor; the right pane renders the scalable vector graphics (SVG) [47] design generated by the code and also offers a graphical interface for performing transformations on the SVG output (i.e., with mouse-based manipulations). The programmer may perform keyboard text edits on the code at any time during program construction, but this ability will not be used in this paper.



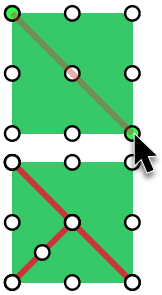
phases: (i) *drawing* the shap
graphical features to align sha
colors; (ii) *abstracting* the d
and, finally, (iv) *refactoring* t
erated code. Throughout, the
SKETCH-N-SKETCH. Newlin
poses in this paper are "escape
code for this example is show

Drawing Shapes

The initial program template i
empty list of SVG shapes:

```
svg (concat [
])
```

Starting from this bare progra
like to get some shapes into
tional programming environn
the drawing library's documen
some example code into the p



```
line1 = line 0 5 topLeft [ x+ w, y2]

line2 = line 0 5 [x, y2] \
          [ (2! * x + w)/ 2!, (2! * y + w) / 2!]
...
```

Figure 4. Code after drawing one square and snap-drawing two lines (right). *Freeze annotations*, written !, tell SKETCH-N-SKETCH not to change those constants when shapes are moved on the canvas [9].

from the top-left to the bottom-right corner and a second line from the bottom-left corner to the center point (these two steps are shown in the right side of Figure 4). SKETCH-N-SKETCH interprets these snaps as constraints which should always hold, i.e., the lines should still coincide with the corners and center even if the square is moved or resized. SKETCH-N-SKETCH encodes these constraints in the program via shared variables: three new variables are introduced for the square's top-left x and y coordinates and its width w, and the spatial properties

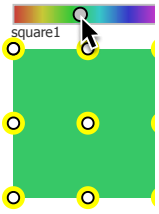


Figure 3. Col

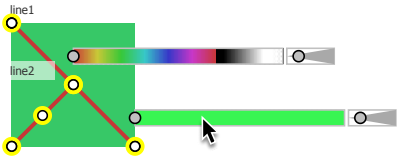


Figure 5. Selecting a feature.

color slider as well.

Whenever an item is selected, SKETCH-N-SKETCH displays a floating menu of output tools. The programmer would like the two selected colors to always be the same, so she moves her mouse to the MAKE EQUAL tool, revealing a list of possible program transformations in a submenu—in this case there is only one, but if the lines had been different colors two results would be offered to let the programmer decide which line's color should take priority (as simulated in Figure 6). Moving her mouse over the single "Equalize by removing line2 color" result in the submenu shows a preview of the new output on

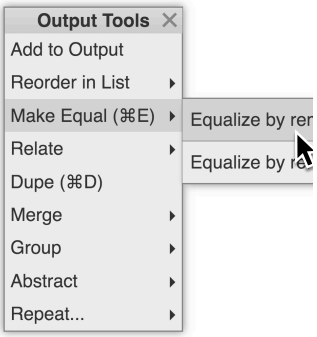
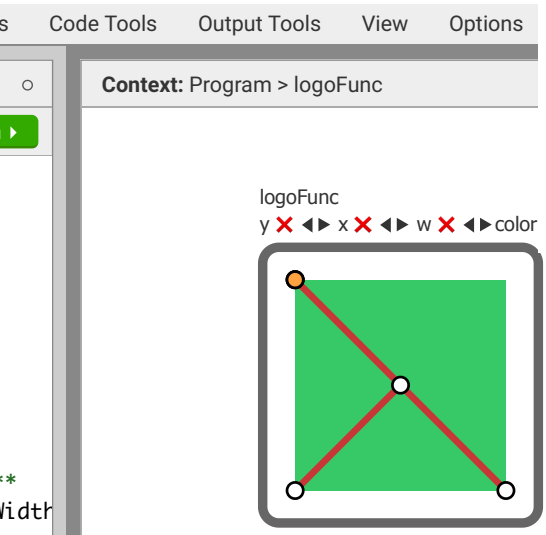


Figure 6. The output tools panel appears w selected. Each tool may offer multiple resul each result previews the change on the code

the canvas and presents a diff of the (not shown)—in this case the appear an appropriately-named `color` variab used for both lines. The programmer i and clicks to apply it. She repeats th the stroke width of the two lines as w



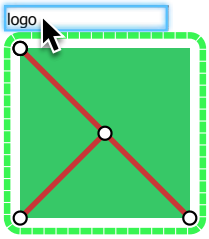


Figure 8. Renaming.

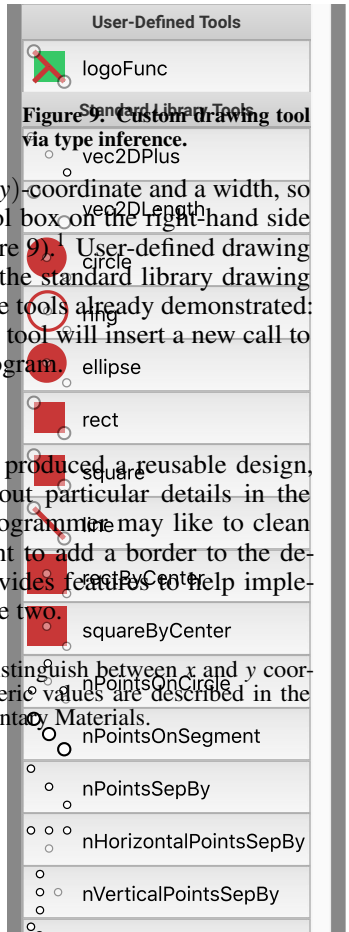


Figure 9. Custom drawing tool via type inference.

niently, however, type inference in SKETCH-N-SKETCH notices that this function accepts an (x, y)-coordinate and a width, so this function appears in the tool box on the right-hand side as a custom drawing tool (Figure 9).[1] User-defined drawing tools behave the same way as the standard library drawing tools such as the square and line tools already demonstrated: drawing on the canvas with the tool will insert a new call to the custom function into the program.

Refactoring

Although the programmer has produced a reusable design, she may change her mind about particular details in the program. For example, the programmer may like to clean up the code and also may want to add a border to the design. SKETCH-N-SKETCH provides features to help implement code changes such as these two.

[1]Type inference mechanisms to distinguish between x and y coordinates, distances, and other numeric values are described in the appendix included in the Supplementary Materials.

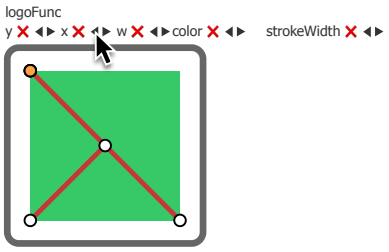


Figure 10. Focused editing.

```
1
2  y = 127
3
4  x = 158
5
6  w = 156
7
8  color = 362
9
10 strokeWidth = 5
11
12 logoFunc x y w color strokeWidth =
13   let topLeft = [x, y] in
14   let square1 = square 140 topLeft w in
15   let y2 = y + w in
16   let xYPair = [ x+ w, y2] in
17   let line1 = line color strokeWidth topLeft xYPair in
18   let line2 = line color strokeWidth [x, y2] \
19                 [ (2! * x + w)/ 2!, (2! * y + w) / 2!] in
20   let polygon1 =
21     let pts = [[x, y], [ x+ w, y], xYPair, [x, y2]] in
22     let [color, strokeColor, strokeWidth] = \
23           [529, 360, 5] in
24       polygon color strokeColor strokeWidth pts in
25   [square1, line1, line2, polygon1]
26
27 logo = logoFunc x y w color strokeWidth
28
29 svg (concat [
30   logo
31 ])
```

Figure 11. The final code for the lambda logo example (Figure 2), produced entirely by output-directed manipulations on the canvas.

programmer was able to create the program entirely through output-directed manipulations on the canvas by using tools for *drawing*, *relating*, *abstracting*, and *refactoring*.

DESIGN AND IMPLEMENTATION

SKETCH-N-SKETCH is a browser application written in the functional language Elm [14] extended with custom providers for mutation and exceptions.

Figure 12 depicts the workflow for output-directed programming in SKETCH-N-SKETCH. The programmer-facing language for users is a standard functional language with Elm-like syntax. A text-based program written in this language is evaluated, and its SVG output value is rendered on the output canvas. The output is overlaid with widgets for selecting and manipulating sub-values (discussed below). The programmer may draw a shape, or drag an item to change a number in the program [9], or make a selection and choose a program transformation from the floating tools menu (Figure 6). The resulting modified program is placed in the code box and is re-evaluated producing new output on the canvas.

Most of SKETCH-N-SKETCH's tools (summarized in Figure 13) operate on selections on the canvas. Different kinds of sub-values of the final output may be selected: whole shapes may be selected, or sub-features of shapes may be selected—namely, positional properties (via points rendered on the corners and edge midpoints of shapes), size properties (via selection regions for width and height), colors (via color sliders), or strokes (via stroke width sliders). When the programmer makes a selection and then invokes a transformation, the selected sub-values are "arguments" to the transformation and are interpreted in a transformation-specific way.

Looking forward to applying ODP to larger programs, manipulation of the final output alone will be insufficient. The output by itself does not represent the process by which the output was produced. We believe some of the intermediate process should be exposed on the canvas for manipulation.

Therefore, in addition to the sub-values of the final output described above, the programmer may also select two additional kinds of terms associated with the program's execution. First, we also expose widgets on the canvas that correspond to *intermediate values* produced during execution. Second, we allow the programmer to select a *sub-expression* in the program to denote the context under focus. These additional kinds of selections—intermediate values and focused sub-expressions—serve as further arguments to program transformations and enable the programmer to modify portions of the computation that do not have a direct representation in the final output.

To highlight the key ideas behind our implementation, we discuss these two novel selection types below. We also discuss the degree to which several of our transforms are ordinary automatic refactorings (i.e. not specific to SVG), as well as the generic provenance tracing that the transformations rely upon to associate canvas selections with program locations.

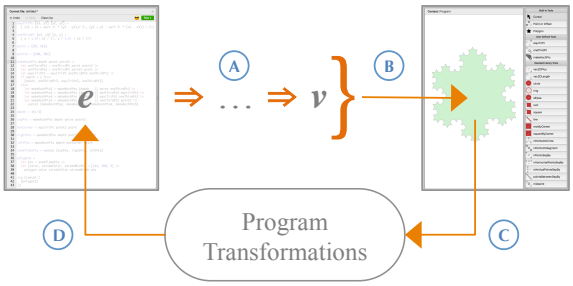


Figure 12. Output-directed programming workflow. (A) Program e evaluates to value v of type SVG. (B) Graphical widgets for manipulating the output value, intermediate values, and the program. (C) Programmer selects from toolbox (cf. Figure 1a) and draws on canvas, or manipulates existing shapes and selects a tool from the floating tools menu (Figure 6). (D) One or more candidate programs are produced; the programmer previews the changes and then chooses one (Figure 6).

Drawing	#Ex
Draw Shape	16
Draw Custom Shape	4
Dupe	2
★ Draw Point	3
★ Draw Offset	9
★ Delete	4
★ Focused Drawing	1
★ Snap Drawing (UI)	15

Abstracting	#Ex
Group	8
Abstract	9
Merge	0
★ Repeat over Function Call	2
★ Repeat over Existing List	3
★ Repeat by Indexed Merge	2
★ Fill Hole	2
★ List Widgets (UI)	9

Relating	#Ex
Make Equal	12
★ Relate	2
★ Distance Features (UI)	6

Refactoring	#Ex
★ Focus Expression	3
★ Rename in Output	15
★ Add/Remove/Reorder Arg.	6
★ Reorder List	4
★ Add to Output	1
★ Select Termination Cond.	1

Figure 13. Program transformations and user interface features (UI) in Sketch-n-Sketch. Those marked with ★ are new to our system; the remainder are improvements upon [19]. The #Ex column indicates the number of examples in Figure 15 in which the feature was used.

Intermediate Value Widgets

To allow manipulation of computation steps before the final output, Sketch-n-Sketch displays widgets for three kinds of intermediate values produced *during* execution: point values, offset values, and list values.

Points (number-number pairs, e.g. `[10, 20]`) produced at intermediate execution steps are exposed for manipulation as *point widgets* on the canvas. Between selected points, we also draw *distance features* for selecting Cartesian distances.

In graphics code, it is common to define offsets from some base x or y values. Therefore, during execution, when a numeric amount is added to or subtracted from an x or y coordinate, an *offset widget* is drawn on the canvas as a horizontal or vertical arrow from the initial point, where the length of the arrow is the numeric amount of the offset. The arrow itself may be selected or dragged to change the offset amount.

To complement the new point widgets and offset widgets, a "Point and Offset" tool in the toolbox (Figure 1b) allows the programmer either to click on the canvas to add a point definition to the program (e.g. `[x, y] as point = [10, 20]`) or to drag on the canvas to add a new offset definition (e.g. `xOffset = x + 5`). Offsets not drawn from an existing point

```
[x, y] as point = [208, 256]

halfW = 102

xOffset = x + halfW

xOffset2 = x - halfW

halfH = 145

yOffset = y - halfH

yOffset2 = y + halfH
```

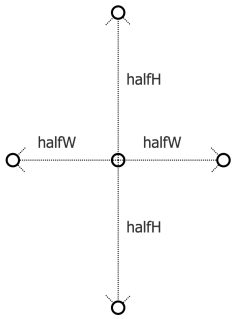


Figure 14. Point and offset widgets are displayed based on program execution to allow manipulation of intermediate computations. Depicted is an early step in the creation of Figure 15xii.

also insert a new point for the base of the offset. To ease the creation of symmetry, offset amounts may snap to each other while drawing. The symmetry is enforced by introducing a shared variable for offset distance. For example, the program in Figure 14 is rendered on the canvas as a point widget from which four offsets emanate. The variables `halfW` and `halfH` are shared offset amounts produced by snap-drawing (and subsequently renamed by the programmer). The ends of the offsets can serve as snap targets for future drawing (e.g. the rhombuses in Figure 15xii). Note that Figure 14 was created by the using the "Point and Offset" tool, but the rendered widgets would be the same if the program were instead written by text edits—only the code matters. Sketch-n-Sketch emits widgets from the *evaluator* when appropriate computation patterns are encountered.

A third type of intermediate widget—a *list widget*—is emitted whenever an execution step evaluates to a list of graphical items. A list widget is drawn as a dashed box encompassing the items (Figure 7). Any list of shapes, list of points, or list of lists in the program can thus be selected. List widgets obviate the need for a graphics-specific construct to denote groups of shapes—a group is just a list of shapes, and the list widget facilitates selection of, and thus operations on, the group.

Expression Focusing

To enable manipulation of subsections of the program, a program expression may be selected to control the syntactic scope of changes made to the code. Most notably, focusing a shape group (i.e. list of shapes) or a function causes drawing operations to insert shapes into that group or function rather than the final output. Accordingly, a recursive design can be created by drawing a function inside itself.

When a function call produces graphical elements, a *call widget* is displayed around the returned elements as a box with a gray solid border. Clicking the gray border *focuses* the function call (Figure 10). The remainder of the program's output is hidden and drawing operations are interpreted inside the focused function. Clicking the call widget again or pressing the "escape" key returns focus back to the entire program.

Literal (non-function) definitions may also be focused. If a canvas selection refers to the right hand side of an assignment, a Focus Expression tool is offered to focus that assignment.

Generic Refactoring

Looking forward to applying ODP to domains beyond vector graphics, a number of SKETCH-N-SKETCH's tools apply standard automatic refactorings.

Most notably, labels are drawn next to most widgets on the canvas to aid comprehension—any label may be clicked to RENAME the associated variable and its uses.

SKETCH-N-SKETCH also includes tools for refactoring functions. The ABSTRACT tool performs a generic "Extract Method" refactoring, building a new function parameterized over the expression's free variables. Also, when a function call is focused by clicking its call widget, SKETCH-N-SKETCH offers actions to add, remove, or reorder arguments. Finally, the MERGE tool performs clone elimination.

Provenance Tracing

While the widgets for intermediate values expose relevant steps of the computation on the canvas, occasionally the transformations require knowledge about parts of the computation that are not directly represented by widgets. For example, the ADD ARGUMENT tool searches for every expression that affected the selected value and separately offers each such expression as a possible new argument to the function.

To help the tools answer these kinds of questions, the SKETCH-N-SKETCH evaluator performs tracing on *every* execution step: the resultant value is tagged with the expression being evaluated as well as pointers to the prior (tagged) values used in the immediate computation; transitively following these prior tagged values reveals the dependencies of the computation. To answer containment queries, we additionally add pointers from list elements to the list(s) containing them. Our tracing discards certain control flow information, most notably pattern match de-structuring operations; future versions of SKETCH-N-SKETCH may adopt the Transparent ML [1] tracing scheme to preserve this information. Neither our tracing nor Transparent ML is specific to SVG—we suspect generic tracing will be useful for ODP in multiple domains.

SKETCH-N-SKETCH becomes sluggish on larger examples. Although tracing theoretically adds memory and time overhead to evaluation, the culprit not the tracing itself but is usually comparison between large traces or widget processing.

CASE STUDY OF ODP EXAMPLES

To explore the expressiveness of ODP in SKETCH-N-SKETCH, we implemented 16 parametric designs summarized in Figure 15. Drawn from various sources [9, 19, 34, 5, 8], the designs exercise different features: 6 designs are parameterized functions that appear as drawing tools at the end of construction, 6 involve repetition by position, 1 involves repetition by other features (radius and color, Figure 15xiii), and 1 uses recursion (Figure 15i). All 16 programs, spanning 427 lines of code total, were built entirely via output-directed manipulations, without any text editing in the code box.

Below, we discuss common patterns in how we used the tools to implement the examples. Next, we recount the key steps in building the recursive Koch curve and explain the process to construct designs involving repetition. Finally, we describe limitations by discussing the tasks from the *Watch What I Do: Programming by Demonstration* [34] benchmark suite that the current system cannot complete without text edits.

Authoring

The lambda logo walkthrough in the overview framed the SKETCH-N-SKETCH workflow as a four stage *draw-relate-abstract-refactor* authoring process. In practice, design construction cannot always be cleanly delineated into precise stages in a fixed order. For example, while constructing the designs of Figure 15, we often specified the relationships *before* drawing the shapes: we laid out the desired parameterization of the design using points and offsets (as in, e.g., Figure 14) and afterwards attached shapes using snap-drawing.

The #Ex column of Figure 13 lists how many of the Figure 15 examples utilized each tool. Indeed, besides shape drawing and variable renaming, the most widely used functionality was SKETCH-N-SKETCH's snap-drawing ability—not surprising given that the goal was to create parametric designs.

It is notable that to encode spatial constraints we more often preferred to snap-draw rather than use the MAKE EQUAL tool. MAKE EQUAL is more flexible, but not only does it require extra clicks compared to snap-drawing, MAKE EQUAL can offer a large number of different but hard-to-distinguish ways to enforce a constraint (Figure 6 is a tame example). To help, SKETCH-N-SKETCH employs a ranking heuristic that seems to work well in practice: MAKE EQUAL prefers changes that rewrite terms near each other and later in the program. The least used tool for specifying relationships was the RELATE tool. RELATE guesses a mathematic relationship between selected items—we only used it for constraints involving thirds.

Offsets plus snaps and MAKE EQUAL was sufficient, but not always convenient, for creating reasonable parameterizations of the designs. Laying out offsets beforehand requires forethought. A future SKETCH-N-SKETCH might benefit from tools to break constraints or invert dependencies after the fact.

As indicated in Figure 13, we did not use the MERGE tool and three tools were used only once, all on the most challenging example, the Koch snowflake. The MERGE tool merges multiple copies of a shape into a function—we prefer ABSTRACT instead because it requires only a single example. The three tools only used for the Koch fractal all played a role in the workflow to specify recursion, which we now recount.

Recursive Koch Snowflake

We highlight the key steps in creating the recursive von Koch snowflake design [45] (Figure 1, Figure 15i). This task requires manipulating intermediates—program has no output throughout most of its development—and focused editing is needed to build the recursive function. Here we emphasize the steps to specify the recursion, although the video in the Supplementary Materials walks through the entire construction.

Each side of the Koch snowflake is based on a recursive motif shown in Figure 1f. The motif requires two helper functions: a function that, given two points, computes a point $\frac{1}{3}$ of the way between them (`oneThirdPt`, Figure 1d), and a function that, given two points, computes a third point that completes an

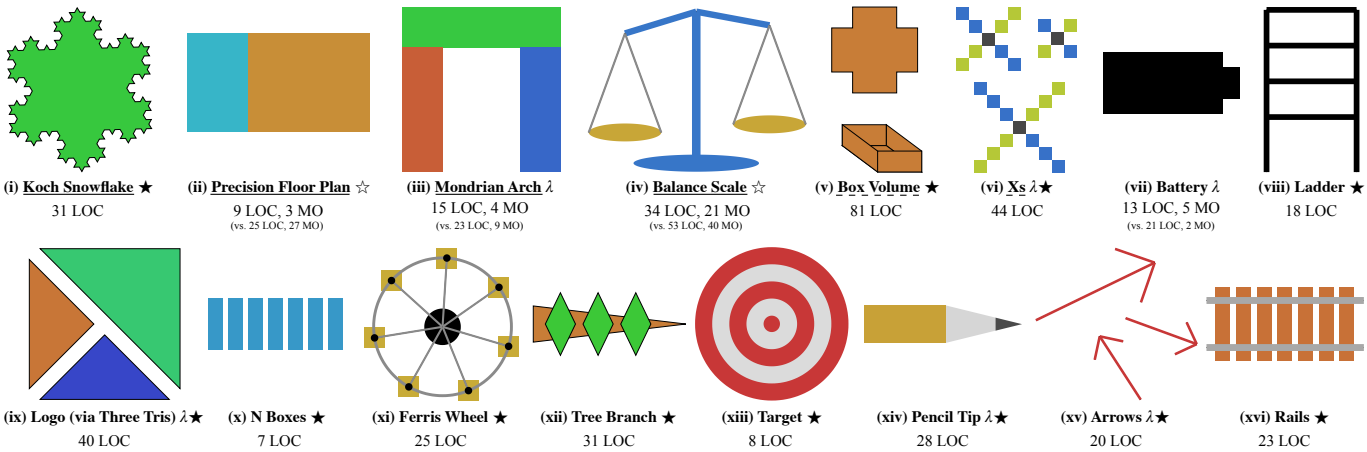


Figure 15. Example programs created solely via output-directed manipulations. LOC = Source lines of code. λ = Final design is a function that appears as a drawing tool. *COMPARISON TO [19]:* If an example can similarly be created in [19], code metrics for [19] appear in parentheses. Math operations (MO) indicate code simplifications due to our improvements. ★ (resp. ☆) = Task cannot (resp. can with undesired parameterization) be created in [19]. *SOURCE OF EXAMPLES:* WWID: PBD [34]: Tasks marked with underline (dashed = only partially completed). Lillicon [5]: *(vii) Battery*. QuickDraw [8]: *(viii) Ladder*. SKETCH-N-SKETCH [19]: *(ix) Logo (via Three Tris)*. SKETCH-N-SKETCH [9]: *(x) N Boxes*, *(xi) Ferris Wheel*.

equilateral triangle (`equiTriPt`, Figure 1e). The two helpers are created without text edits via the RELATE and MAKE EQUAL tools, respectively. Each helper takes in two points and is thus exposed as a drawing tool (Figure 1c).

The Koch motif is created by snap-drawing `oneThirdPt` forward and backwards to yield the $\frac{1}{3}$ and $\frac{2}{3}$ points; `equiTriPt` is used to create a point equidistant from the $\frac{1}{3}$ and $\frac{2}{3}$ points. The bottom of Figure 16 shows this initial motif after abstraction. Note that when a function is focused, as in Figure 16, input points are rendered in orange and output points in blue. The code is not recursive yet:

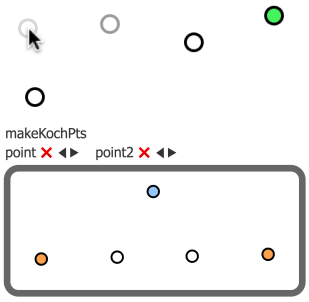


Figure 16. Initial Koch motif.

```
makeKochPts point point2 =
  let oneThirdPt2 = oneThirdPt point point2 in
  let oneThirdPt3 = oneThirdPt point2 point in
  equiTriPt oneThirdPt3 oneThirdPt2
```

Drawing the function inside itself causes SKETCH-N-SKETCH to insert a recursive skeleton and the recursive call:

```
makeKochPts point point2 =
  let oneThirdPt2 = oneThirdPt point point2 in
  let oneThirdPt3 = oneThirdPt point2 point in
  let equiTriPt2 = equiTriPt oneThirdPt3 oneThirdPt2 in
  if ??terminationCondition then
    equiTriPt2
  else
    let makeKochPts2 = makeKochPts point oneThirdPt3 in
      equiTriPt2
```

To avoid infinite recursion, the temporary guard expression `??terminationCondition` evaluates to `false` the first time the function is encountered, and `true` if the function has appeared earlier in the call stack, affecting termination at a fixed depth. The termination condition may be selected later.

After drawing the function between the remaining three pairs of points the design looks like a fractal (Figure 17). However, only one point is output from the function—the white points are only intermediates. Moreover, most of those intermediates are inside calls to the base case. Although focused on the function's recursive case, to modify the output of the base case, the programmer can click the call widget of a call to the base case (the inner light gray border in Figure 17) to focus that call instead (Figure 18).

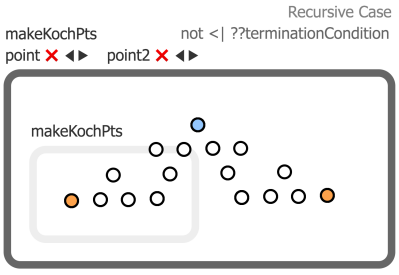


Figure 17. Recursive case.

The ADD TO OUTPUT and REORDER IN LIST tools are used to place the selected points into the output of the base case (resulting in the list on line 16 of Figure 1). Once complete, the recursive case may be refocused, and, using the same two tools, its output is specified to be the concatenation of the lists returned from calls the the base case (line 22 of Figure 1).

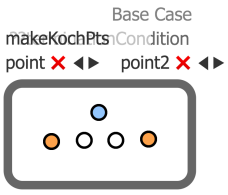


Figure 18. Base case.

What remains is to choose a termination condition. The focused call widget for `makeKochPts` displays the conditional `not <| ??terminationCondition` for the recursive case. Clicking the conditional offers termination conditions. Currently, SKETCH-N-SKETCH implements only one: fixed depth, in which a `depth` argument is decremented on the recursive calls. These changes are visible in the final code (Figure 1f).

With all the points for the Koch snowflake now on the canvas, a polygon is attached to them by selecting the "Polygon" tool and clicking the list widget to select the points. With widgets hidden, the final snowflake is shown in Figure 1.

```
rhombusFunc2 ([x, y] as point) =
  let halfW = 40 in
  let halfH = 83 in
  rhombusFunc point halfW halfH

repeatedRhombusFunc2 =
  map rhombusFunc2 leafAttachmentPts
```

Figure 19. Code fragment from Figure 15xii produced by the REPEAT OVER EXISTING LIST tool. The tool creates a function (`rhombusFunc2`) that produces a shape given a single point, and maps that function over a list of points defined elsewhere in the program (`leafAttachmentPts`).

Repetition

Of the Figure 15 examples, 7 utilize SKETCH-N-SKETCH's tools for creating repetitive designs. A selected shape may be repeated either over an existing list of points in the program or over a new call to any function that returns a list of points (the default toolbox in Figure 1a contains several such functions). Figure 19 illustrates code produced by these repetition tools.

For creating designs that vary over a non-spatial attribute, SKETCH-N-SKETCH alternatively offers a programming by demonstration workflow: after manually laying out the first few shapes of a repetitive design, the REPEAT BY INDEXED MERGED tool syntactically merges the shape expressions into a single function that is called with an argument `i` that takes on the values `0`, `1`, `2`, etc. Using a simple form of sketch-based synthesis [41], a special tool called FILL HOLE resolves syntactic differences between the original shape expressions by synthesizing expressions that refer to the index variable `i`. Figure 15x and xiii were constructed with this workflow.

Limitations: Remaining Tasks

Of the 15 tasks in the *Watch What I Do: Programming by Demonstration* [34] benchmark suite that may be interpreted as parametric drawings, our system is able to fully complete 4. What is needed to complete all the tasks without text edits?

Two of these remaining WWID: PBD tasks can be partially completed in this work (Figure 15v, vi). To fully complete them, "Box Volume" would require an interaction to compute and display the numeric volume of the folded box, and "Xs" would require more precise control over what definitions are abstracted. (Not all uses of a `squareWidth` parameter are pulled into the abstraction, causing the design to render incorrectly when drawing an X with different sized squares.)

The remaining nine tasks are diverse; no single feature would help with more than two or three. A prominent missing feature is arbitrary text boxes, with other elements placed relative to the text size. Beyond this, several examples require various list operations. SKETCH-N-SKETCH would also need to reason about intersections of lines with shape edges, to offer ways to specify overlapping and containment constraints, and to solve different kinds of such constraints simultaneously. Finally, one example would require creating `if-then-else` branches outside of a recursive or hole-filling setting.

RELATED WORK

Many related approaches provide direct manipulation tools to (a) transform programs and/or (b) build parametric designs.

Parametric Computer-Aided Design (CAD)

Feature-based parametric CAD systems record user actions as a series of steps that together act as a program encoding the creation of the design. If an element's property is changed, dependent actions in the sequence are re-run. Among CAD systems, EBP [33] is notable for offering a programming by demonstration workflow to create loops and conditionals.

Drawing With Constraints

Several visual design systems integrate constraint specification (e.g. [48, 22]). Notably, Apparatus [37], Recursive Drawing [36], and Geometer's SketchPad [21] support recursion. Like CAD, but unlike SKETCH-N-SKETCH, these systems do not represent the design in a general-purpose, text-based programming language.

Constraint-Oriented Programming (COP)

Other constraint-oriented systems, following in the footsteps of SketchPad [42], explicitly view building a constrained system as a programming task [6, 20, 15]. While offering varying degrees of visibility into the code, these systems are distinguished by running constraint solvers alongside the program. We instead rely on standard execution semantics.

Programming by Demonstration (PBD)

Several PBD approaches [10] use shape drawing as a domain for exploring non-textual programming techniques. These systems usually rely on a visual rather than representation of the program (e.g. [23, 25]) or show actions step-by-step [28]. We also use shape drawing as our application domain, but we do not hide the program text. Although not as visual as peer PBD systems, Tinker [26] is notable for supporting recursion by demonstration; indeed, any Lisp expression may be created, albeit via manipulations performed on a symbolic representation not far removed from the underlying Lisp. More recently, PBD techniques have been developed for data visualization [44], mobile applications for collaboration [12], web scraping [3, 7], API discovery [49], and hand-drawing recognition [13].

Output-Directed Programming

Several recent systems augment a regular text-based programming experience with abilities to directly manipulate output to enact code changes. The transformations available may be "small" (such as changing constants [9, 24, 29], strings [46, 39, 24, 29], or list literals [29]), but several systems enable "larger" program changes via output manipulation.

Transmorphic [38] re-implements the Morphic UI framework [27] but with stateless views. Transmorphic retains Morphic's ability to manipulate shapes by affecting changes to a view's (text-based) code rather than changing live object state. Adding and removing shapes as well as changing a shape's primitive properties are both supported.

APX [31, 32], like SKETCH-N-SKETCH, is a two-pane (code box and output canvas) environment for creating shape-drawing programs (APX also supports dynamic visual simulations). On the canvas, manipulating shape attributes changes appropriate numbers in the program; a few larger changes (e.g. grouping and inserting shapes) are also supported. However, most of APX's interactions focus on directly manipulating terms in the code box rather than on the canvas.

Prior SKETCH-N-SKETCH

Our prior work on SKETCH-N-SKETCH [19] introduced a *draw-relate-abstract* workflow, including the ability to select sub-values in the output and invoke certain program transformations. Figure 13 indicates which tools also appeared in this prior work. In the present work, besides exposing intermediates for manipulation, offering expression focusing, and adding repetition and generic refactoring tools, we also sought to improve the prior *draw-relate-abstract* tools.

The prior SKETCH-N-SKETCH [19] relied on syntactic restrictions: shapes followed a strict "left-top-right-bot" bounding box parameterization, special functions were needed to compose shapes, and the program needed to be a series of top-level definitions followed by a list literal "main" expression referring to the top-level definitions. If, through text-editing, the programmer veered outside of this syntactic subset of the language, many output-directed tools became unavailable.

In this work, we relaxed these syntactic restrictions and moved the operation of SKETCH-N-SKETCH's tools closer to ordinary programming concepts. The drawing tools now operate based on types rather than a syntactic bounding box construct. No longer are special function calls needed to compose shapes: groups are now ordinary lists . Our generic provenance tracing enables the tools to transform shapes not defined at the top level. We also now offer multiple results for transformations.

We additionally improved code generation. Four of the examples in Figure 15 (ii, iii, iv, vii) can also be constructed without text edits in the prior SKETCH-N-SKETCH [19], albeit with undesirable parameterizations for two of the four (ii, iv). Figure 15 lists the source lines of code (LOC) and math operations (MO) for the programs in both the present and prior versions of SKETCH-N-SKETCH. Reductions in LOC are incidental—our drawing tools now insert single-line function calls instead of wordier multi-line definitions. Reductions in math operations, however, do indicate better code: the prior SKETCH-N-SKETCH sometimes inserted large math expressions too complex for human comprehension. Besides discarding the bounding boxes that required properties such as shape width to be computed (e.g. as `right - left`), we also more aggressively reuse variables and simplify math in the MAKE EQUAL tool, aided by connecting SKETCH-N-SKETCH to an external solver (REDUCE [18]).

During our upgrades, some functionality was disabled but would be useful to restore. Most notably, unneeded by our examples, the current SKETCH-N-SKETCH lacks a path tool. Also, tools to re-parameterize shapes *en masse* could be useful, if, for example, a bounding box parameterization is desired.

CONCLUSION AND FUTURE WORK

We improved expressiveness in SKETCH-N-SKETCH, showing how to construct a variety of non-trivial programs entirely through direct manipulation. Our long-term goal, however, is not to create a practical drawing tool but to use this particular application domain as a laboratory for advancing the expressiveness of output-directed programming for a variety of future programming settings. There are several avenues to explore in this direction.

ODP for Novices

Our work so far has assumed the SKETCH-N-SKETCH user is comfortable working in code to understand program operation. ODP interactions might also help those with little programming experience—such as domain experts or students—to quickly produce rudimentary programs. Design considerations for novices should be investigated.

Widget Visibility

Because program execution may involve a large number of intermediate evaluation steps, even simple programs might clutter the canvas with widgets, making it unusable. Therefore, our current implementation hides most widgets by default and uses heuristics to determine when to show them—generally, upon the mouse hovering over some associated shape. Additionally, widgets from intermediate expressions in standard library code—outside the visible program—are generally not shown. In the future, these visibility choices could be more systematically controlled, e.g., by source code annotations or user interface options.

Customizing Widgets

Mechanisms for customization may help address the open question of how to render graphical representations of program evaluation and of domain objects that are not inherently visual. One approach would be to design an API for "`toSvg`" functions that specify how to render graphical representations of intermediate data structures. Analogous to "`toString`" functions that render text, a `toSvg` API might similarly render composite data structures via recursively calling `toSvg` for each of the structure's elements.

Synthesizing Program Transformations

Each program transformation offered by SKETCH-N-SKETCH has been hand-coded one-by-one. This effort is time-consuming, error-prone, and limits the composability of transformations. Program synthesis techniques have been successfully incorporated into refactoring tasks for class-based languages [35]; perhaps such techniques can be extended to streamline the specification of output-directed transforms that operate simultaneously on both abstract and concrete syntax.

User Interactions for Deciphering Human Intent

There are often multiple valid program changes associated with a specific user action. To resolve ambiguity, our system displays code and output differences and asks the user to choose. Richer interactions are needed to explain the differences (e.g., change impact analysis [16, 11]) and to better resolve the user's intent (e.g., by asking additional questions [30]).

In order for the live and immediate experience afforded by output-directed programming to scale into usable systems for different programming tasks and user scenarios, developing and refining such user interaction techniques remain crucial challenges to tackle in future work.

ACKNOWLEDGEMENTS

Our thanks to Philip Guo and the anonymous reviewers for helpful feedback. This work was supported by U.S. National Science Foundation Grant No. 1651794.

REFERENCES

[1] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2013. A Core Calculus for Provenance. *Journal of Computer Security* (2013).

[2] Autodesk Inc. Maya. `http://www.autodesk.com/products/maya/overview`.

[3] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[4] Andrew Begel. 1996. LogoBlocks: A Graphical Programming Language for Interacting with the World. Advanced Undergraduate Project, MIT Media Lab.

[5] Gilbert Louis Bernstein and Wilmot Li. 2015. Lillicon: Using Transient Widgets to Create Scale Variations of Icons. *Transactions on Graphics (TOG)* (2015).

[6] Alan Borning. 1981. The Programming Language Aspects of ThingLab. *Transactions on Programming Languages and Systems (TOPLAS)* (October 1981).

[7] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Symposium on User Interface Software and Technology (UIST)*.

[8] Salman Cheema, Sumit Gulwani, and Joseph LaViola. 2012. QuickDraw: Improving Drawing Experience for Geometric Diagrams. In *Conference on Human Factors in Computing Systems (CHI)*.

[9] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*.

[10] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.

[11] Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares Vásquez, Denys Poshyvanyk, and Huzefa H. Kagdi. 2014. ImpactMiner: A Tool for Change Impact Analysis. In *International Conference on Software Engineering, Companion Proceedings (ICSE-C)*.

[12] Jonathan Edwards, Jodie Chen, and Alessandro Warth. 2016. Live End-User Programming. In *LIVE Workshop*.

[13] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning To Infer Graphics Programs From Hand-Drawn Images. In *Conference on Neural Information Processing Systems (NIPS)*.

[14] Evan Czaplicki. 2012-2019. Elm. `http://elm-lang.org`.

[15] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. 2014. Babelsberg/JS - A Browser-Based Implementation of An Object Constraint Language. In *European Conference on Object-Oriented Programming (ECOOP)*.

[16] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. 2012. Integrated Impact Analysis for Managing Software Changes. In *International Conference on Software Engineering (ICSE)*.

[17] Michael Gleicher and Andrew Witkin. 1994. Drawing with Constraints. *The Visual Computer: International Journal of Computer Graphics* (1994).

[18] Anthony C. Hearn. 1968. REDUCE: A User-Oriented Interactive System for Algebraic Simplification. In *Interactive Systems for Experimental Applied Mathematics*. Academic Press.

[19] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST)*.

[20] Allan Heydon and Greg Nelson. 1994. The Juno-2 Constraint-Based Drawing Editor. In *Technical Report 131a, Digital Systems Research, Digital Equipment Corporation*.

[21] R. Nicholas Jackiw and William F. Finzer. 1993. The Geometer's Sketchpad: Programming by Geometry. In *Watch What I Do: Programming by Demonstration*. MIT Press.

[22] Jennifer Jacobs, Sumit Gogia, Radomír Mech, 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, Denver, CO, USA, May 06-11, 2017.*

[23] David Kurlander. 1993. *Graphical Editing by Example*. Ph.D. Dissertation. Columbia University.

[24] Kevin Kwok and Guillermo Webster. 2016. Carbide Alpha. `https://alpha.trycarbide.com/`.

[25] Henry Lieberman. 1993a. Mondrian: A Teachable Graphical Editor. In *Watch What I Do: Programming by Demonstration*. MIT Press.

[26] Henry Lieberman. 1993b. Tinker: A Programming By Demonstration System for Beginning Programmers. In *Watch What I Do: Programming by Demonstration*. MIT Press.

[27] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness In the Morphic User Interface Construction Environment. In *Symposium on User Interface Software and Technology (UIST)*.

[28] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. 1989. Metamouse: Specifying Graphical Procedures by Example. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[29] Mikaël Mayer, Viktor Kunčak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA* (2018).

[30] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*.

[31] McDirmid, Sean. 2015. A Live Programming Experience. In *Future Programming Workshop, Strange Loop*. `https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcxdryTyPiuW8` `https://www.youtube.com/watch?v=YLrdhFEAiqo`.

[32] McDirmid, Sean. 2016. The Future of Programming will be Live. In *Curry On*. `https://www.youtube.com/watch?v=bnqkglrSqrg`.

[33] Guy Pierra, Jean-Claude Potier, and Patrick Girard. 1996. The EBP System: Example Based Programming System for Parametric Design. In *Modelling and Graphics in Science and Technology*. Springer Berlin Heidelberg.

[34] Richard Potter and David Maulsby (Eds.). 1993. A Test Suite for Programming by Demonstration. In *Watch What I Do: Programming by Demonstration*. MIT Press.

[35] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with Synthesis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[36] Schachman, Toby. 2012. *Recursive Drawing*. Master's thesis. New York University Interactive Telecommunications Program. `http://recursivedrawing.com/`.

[37] Schachman, Toby. 2015. Apparatus. `http://aprt.us/`.

[38] Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. 2016. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*.

[39] Christopher Schuster and Cormac Flanagan. 2016. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*.

[40] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* (August 1983).

[41] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. UC Berkeley.

[42] Ivan Sutherland. 1963. *Sketchpad, A Man-Machine Graphical Communication System*. Ph.D. Dissertation. MIT.

[43] William Robert Sutherland. 1966. *The On-line Graphical Specification of Computer Procedures*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[44] Victor, Bret. 2013. Drawing Dynamic Visualizations. `http://worrydream.com/#!/DrawingDynamicVisualizationsTalk`.

[45] Helge von Koch. 1904. Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire. *Arkiv för Matematik, Astronomi och Fysik* 1 (1904), 681–704.

[46] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *International Symposium on the Foundations of Software Engineering (FSE)*.

[47] World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.1 (Second Edition). `http://www.w3.org/TR/SVG11/`.

[48] Haijun Xia, Bruno Araújo, Tovi Grossman, and Daniel J. Wigdor. 2016. Object-Oriented Drawing. In *Conference on Human Factors in Computing Systems (CHI)*.

[49] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API Discovery from Demonstrations. In *Conference on Programming Language Design and Implementation (PLDI)*.