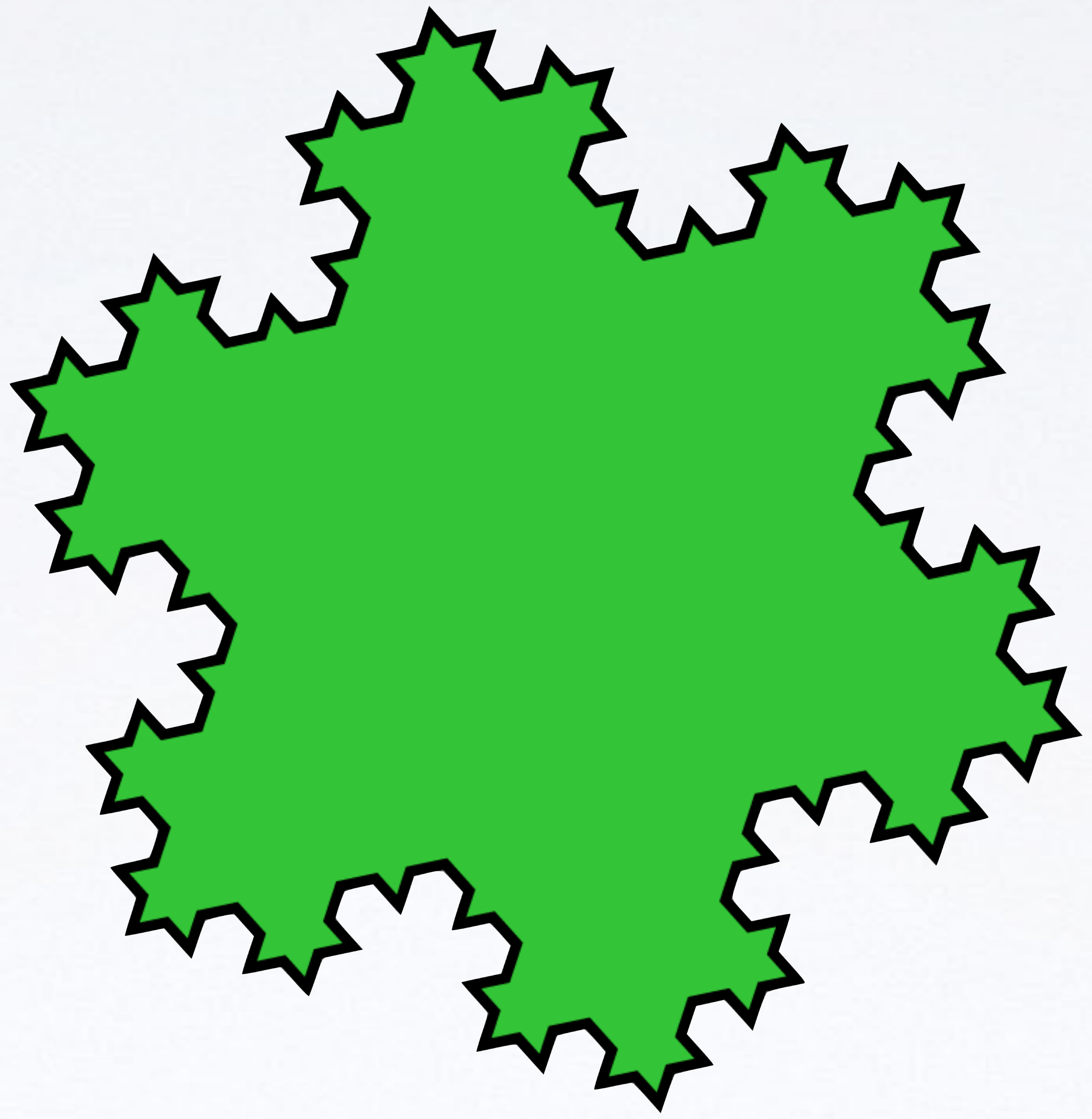
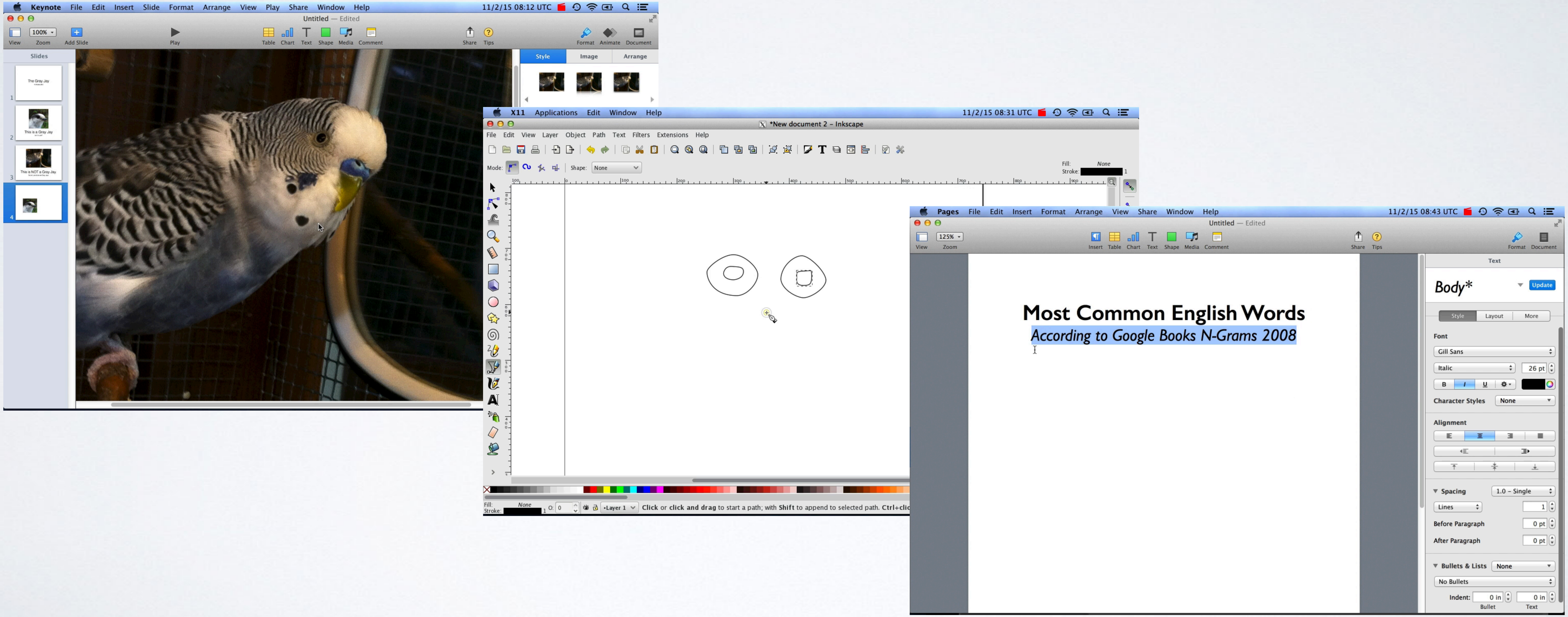
Sketch-n-Sketch: Output-Directed Programming for SVG



Brian Hempel, Justin Lubin, Ravi Chugh

THE UNIVERSITY OF CHICAGO

Direct Manipulation is Everywhere.



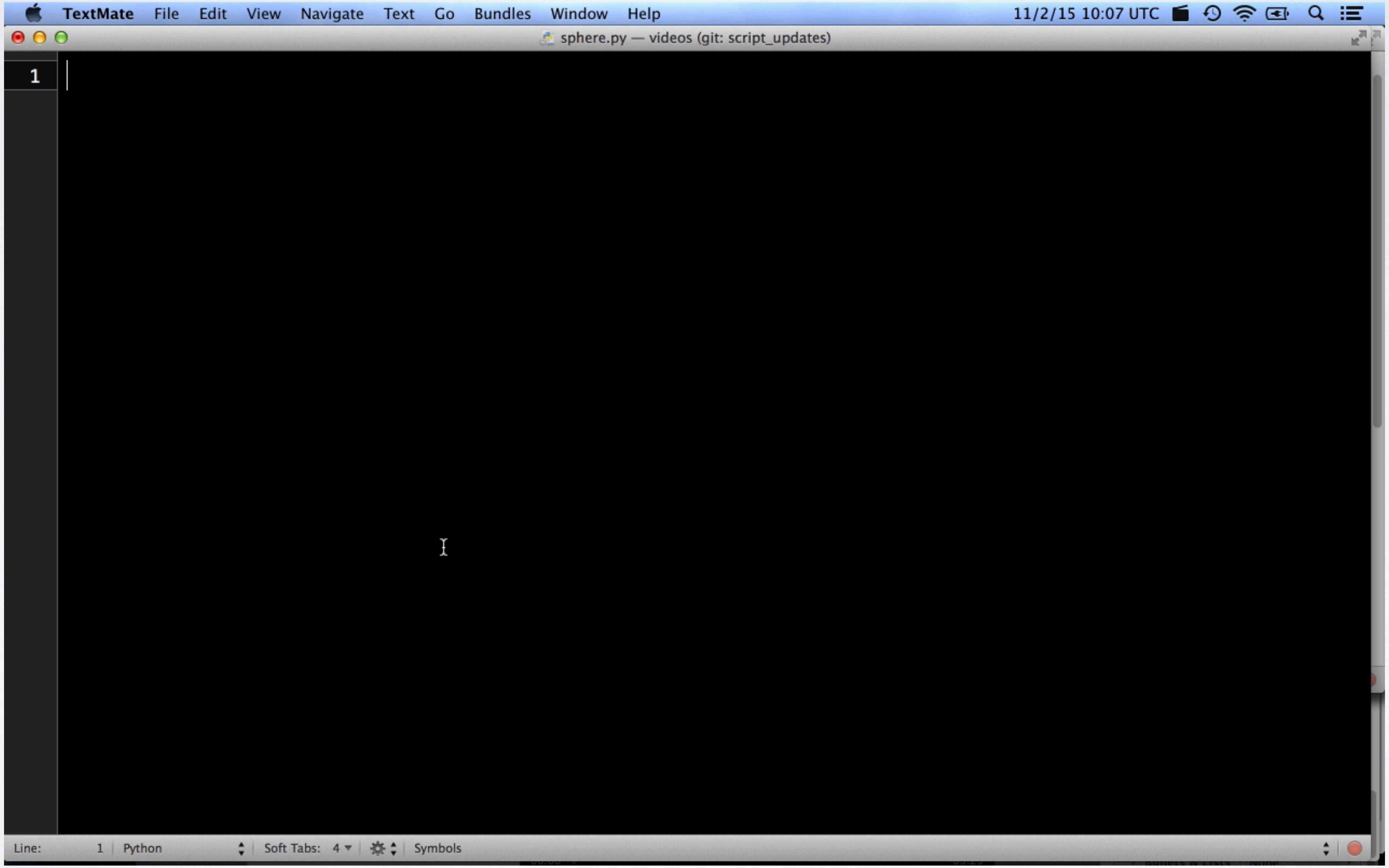
Most Common English Words

According to Google Books N-Grams 2008

Click or click and drag to start a path; with Shift to append to selected path. Ctrl+clic

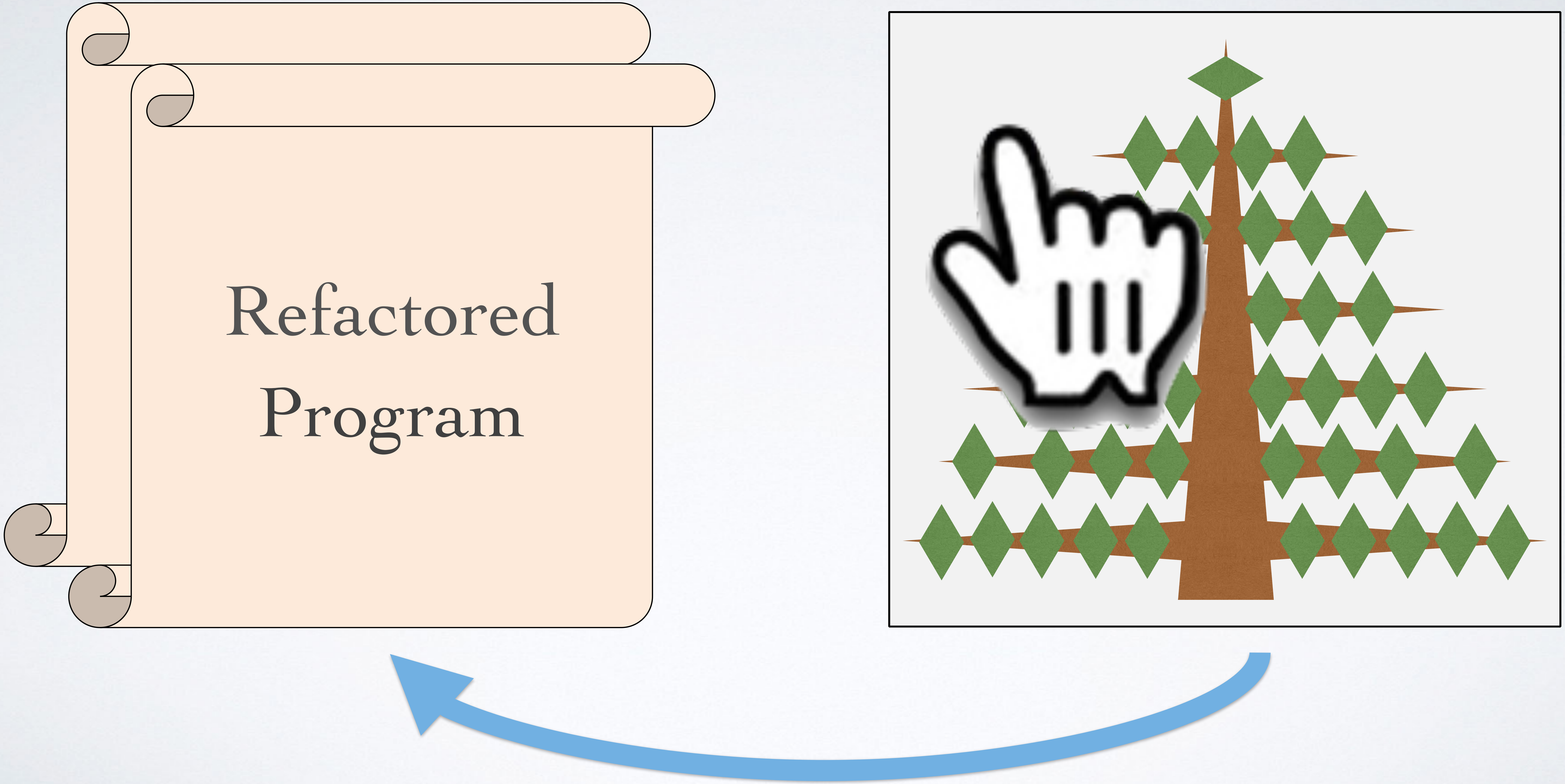
Keynote File Edit Insert Slide Format Arrange View Play Share Window Help
11/2/15 08:12 UTC
Untitled — Edited
View Zoom Add Slide Play Table Chart Text Shape Media Comment Share Tips Format Animate Document
Slides
Style Image Arrange
X11 Applications Edit Window Help
11/2/15 08:31 UTC
*New document 2 - Inkscape
File Edit View Layer Object Path Text Filters Extensions Help
Mode: Shape: None
Fill: None
Stroke:
Layer 1
Pages File Edit Insert Format Arrange View Share Window Help
11/2/15 08:43 UTC
Untitled — Edited
View Zoom Insert Table Chart Text Shape Media Comment Share Tips Format Document
Text
Body* Update
Style Layout More
Font
Gill Sans
Italic 26 pt
Character Styles None
Alignment
Spacing 1.0 - Single
Lines 1
Before Paragraph 0 pt
After Paragraph 0 pt
Bullets & Lists None
No Bullets
Indent: 0 in 0 in
Bullet Text

Programming



TextMate File Edit View Navigate Text Go Bundles Window Help
11/2/15 10:07 UTC
sphere.py — videos (git: script_updates)
Line: 1
Python
Soft Tabs: 4
Symbols

Programming + Direct Manipulation?



Refactored Program

Ordinary, Text-Based Programming

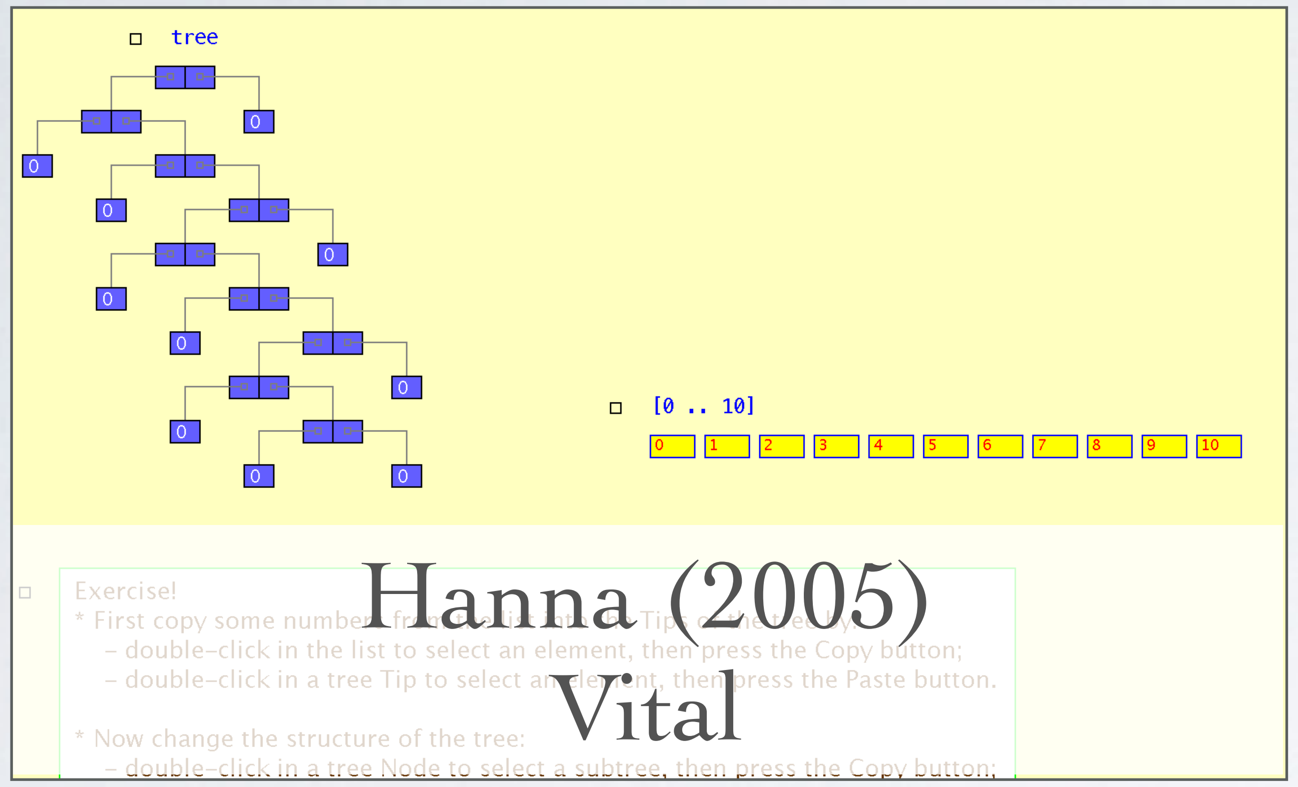
+

Direct Manipulation on Output

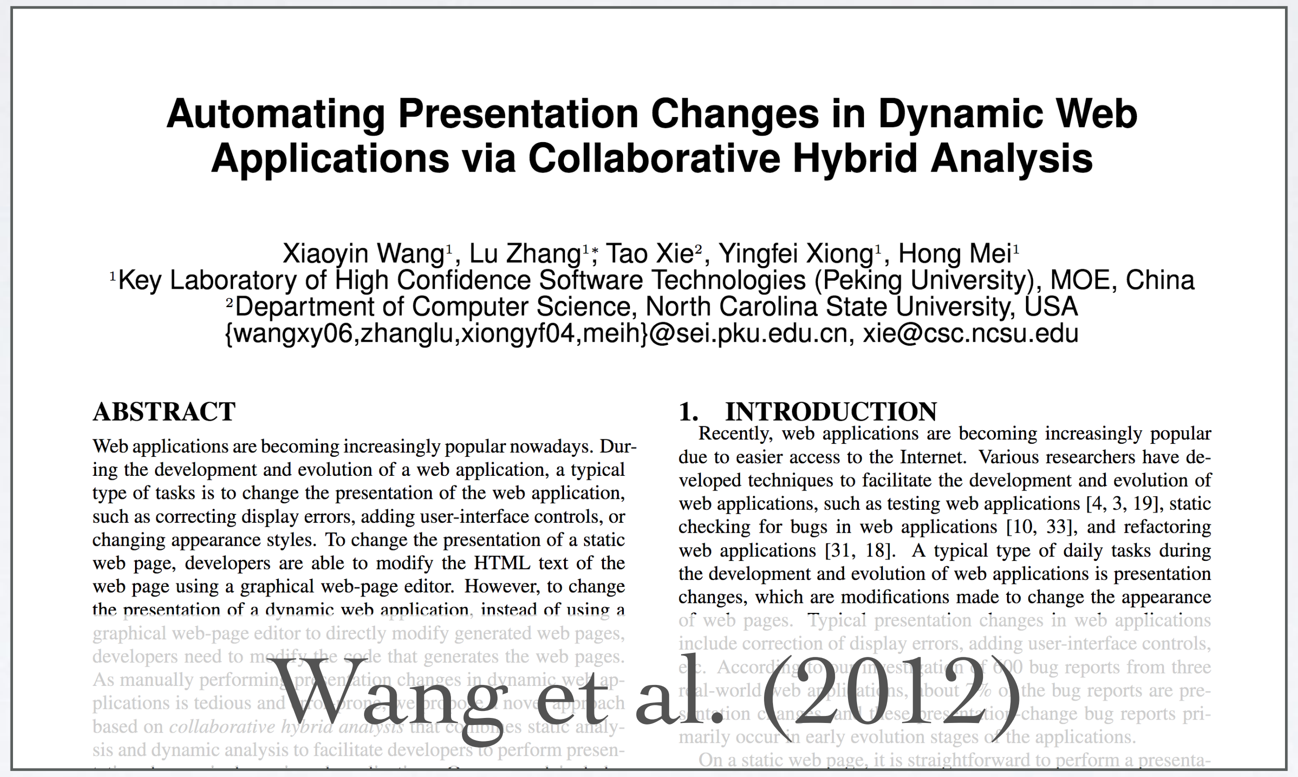
=

Output-Directed Programming

Prior Output-Directed Programming

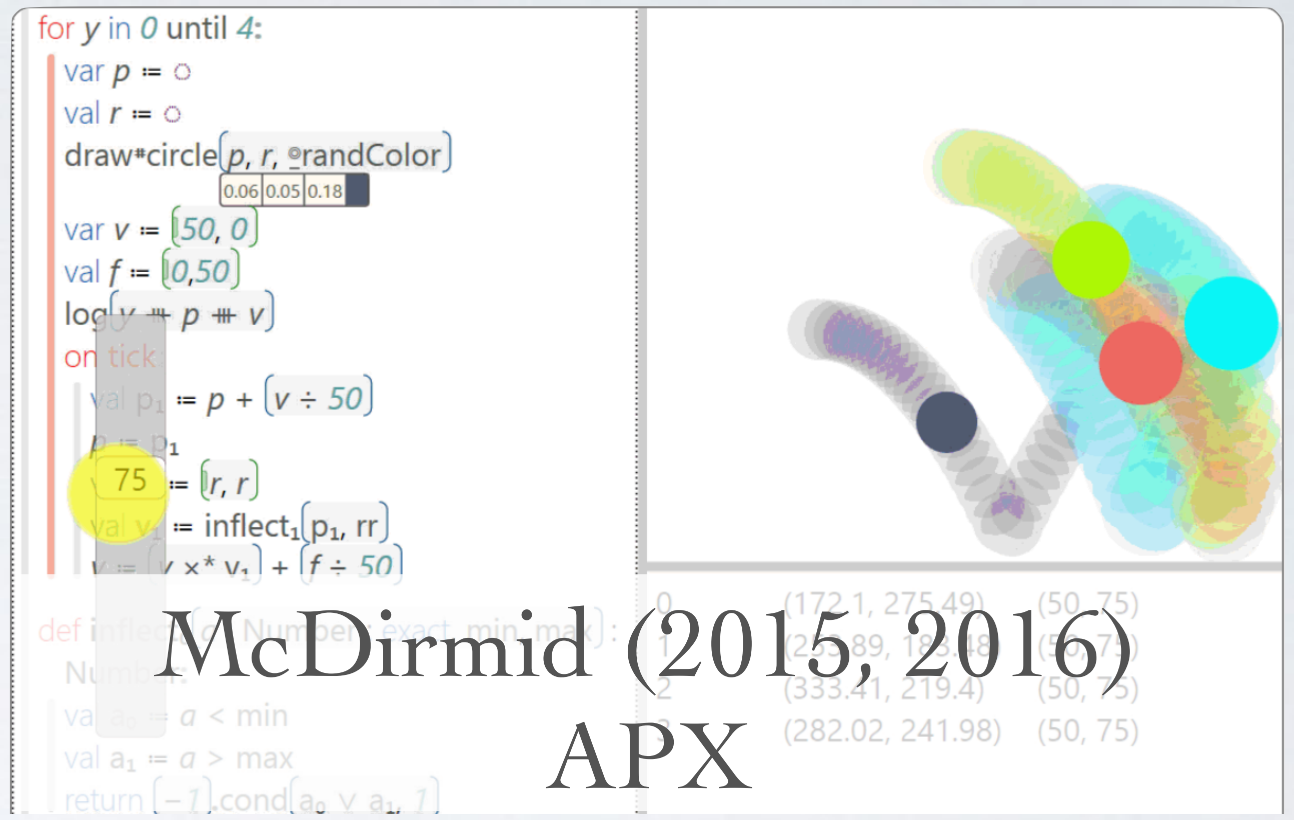


Hanna (2005)
Vital

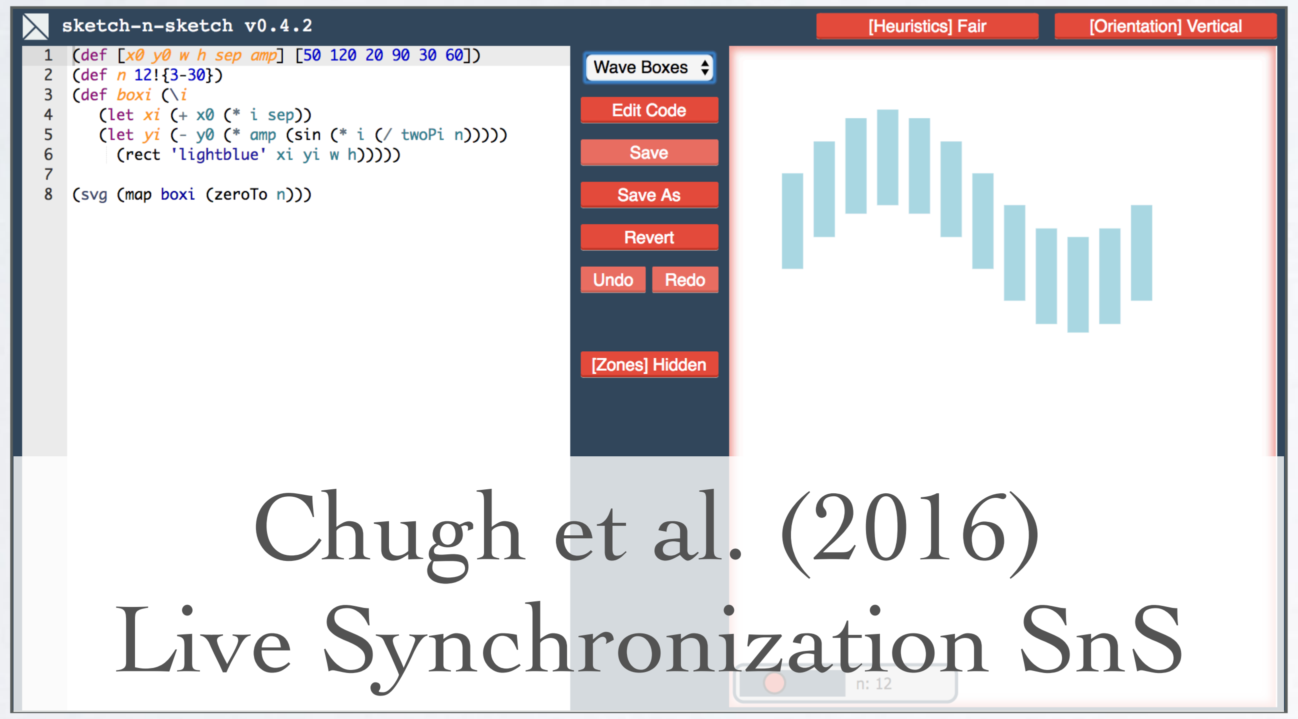


Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

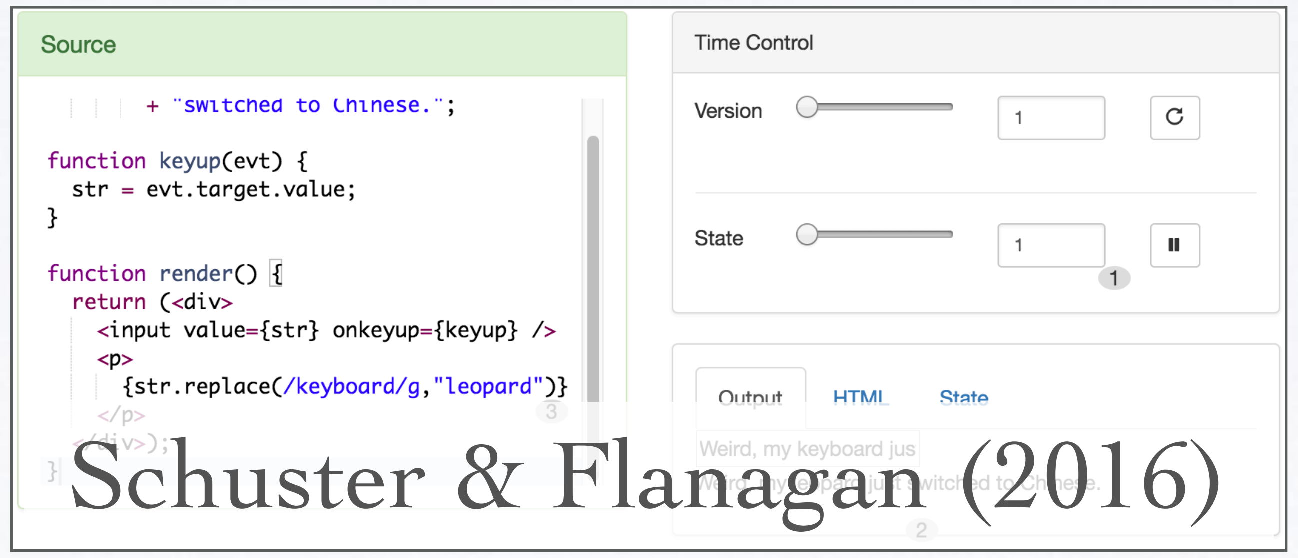
Wang et al. (2012)



McDirmid (2015, 2016)
APX

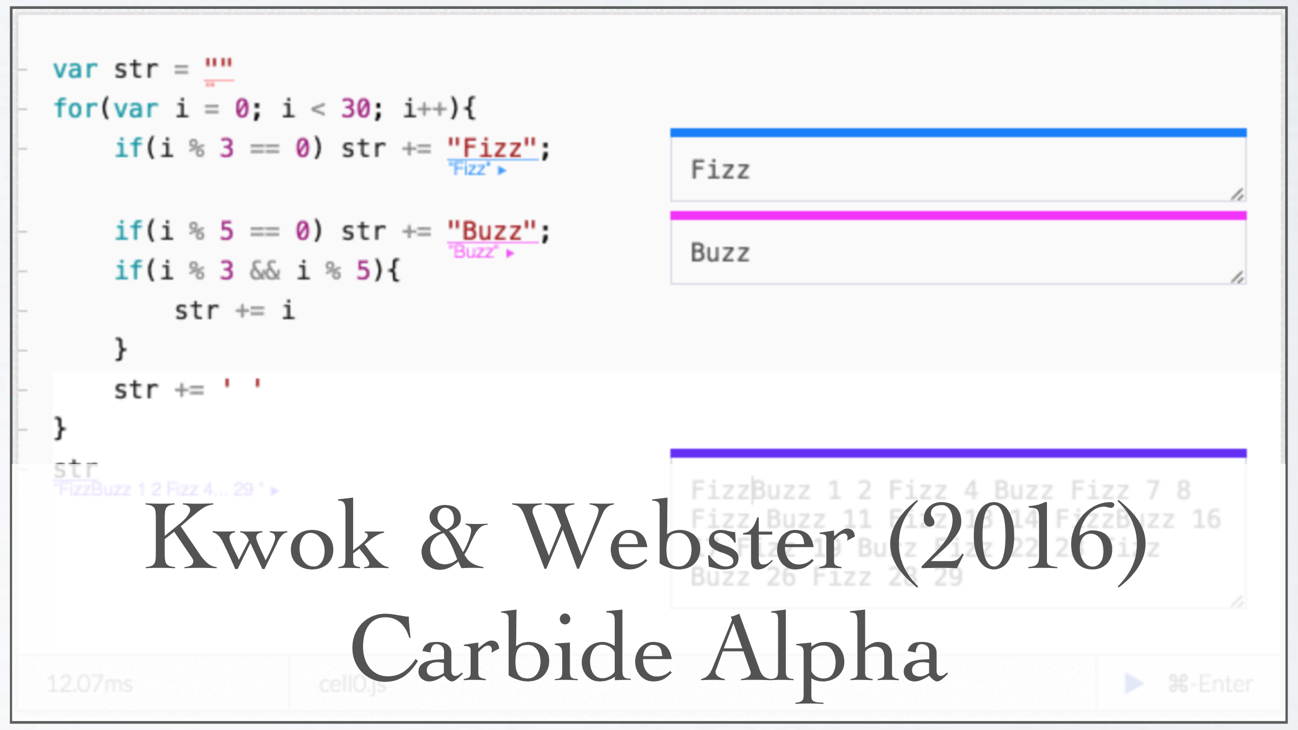


Chugh et al. (2016)
Live Synchronization SnS



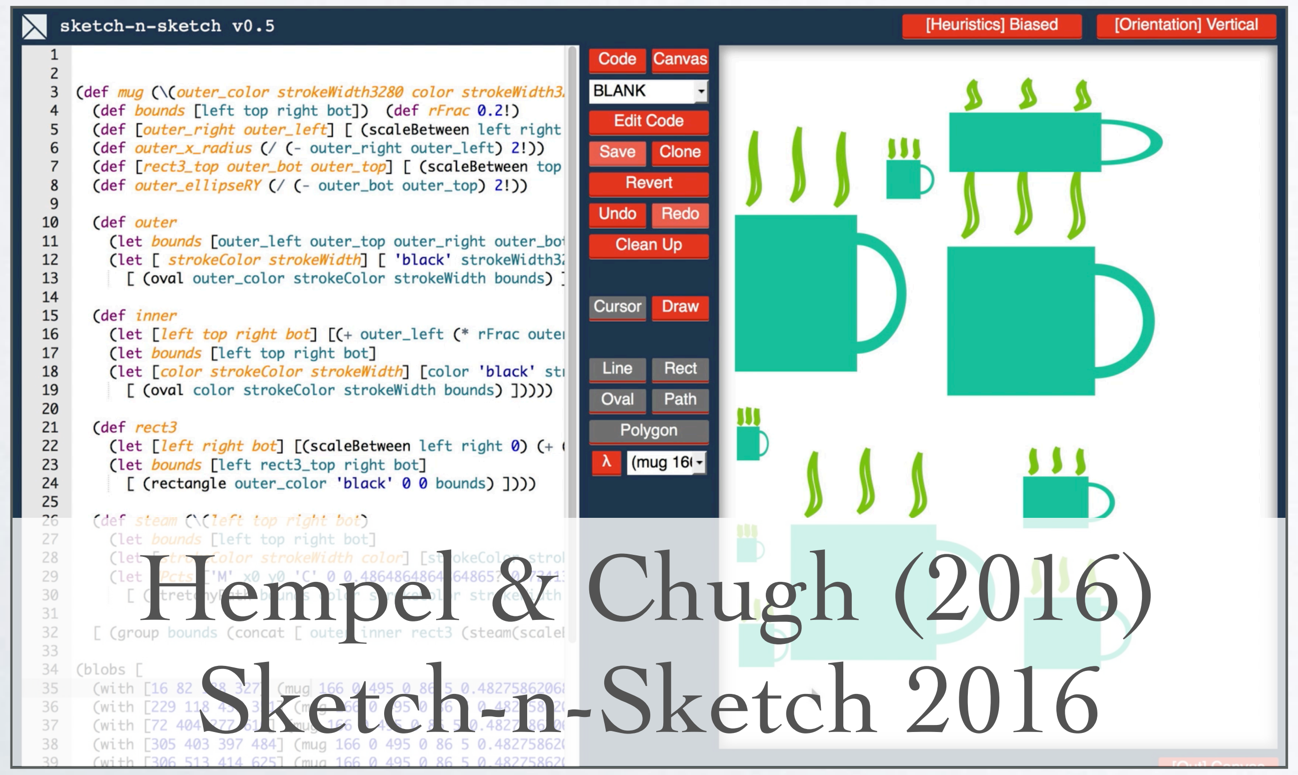
```
function keyup(evt) {
  str = evt.target.value;
}

function render() {
  return (<div>
    <input value={str} onkeyup={keyup} />
    <p>
      {str.replace(/keyboard/g,"leopard")}
```

Schuster & Flanagan (2016)

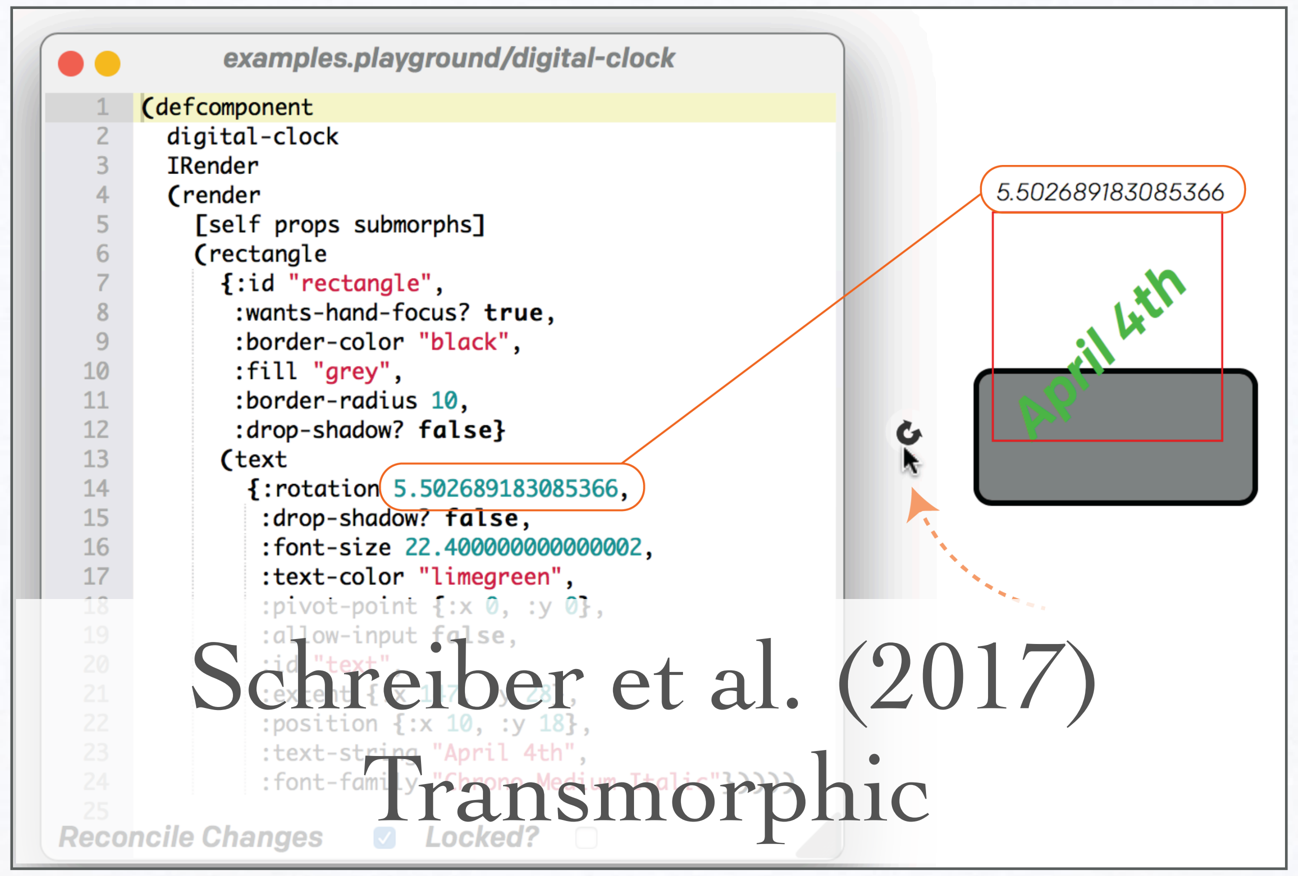


```
var str = "";
for(var i = 0; i < 30; i++){
    if(i % 3 == 0) str += "Fizz";

    if(i % 5 == 0) str += "Buzz";
    if(i % 3 && i % 5){
        str += i
    }
    str += ' '
}
```

Kwok & Webster (2016)
Carbide Alpha

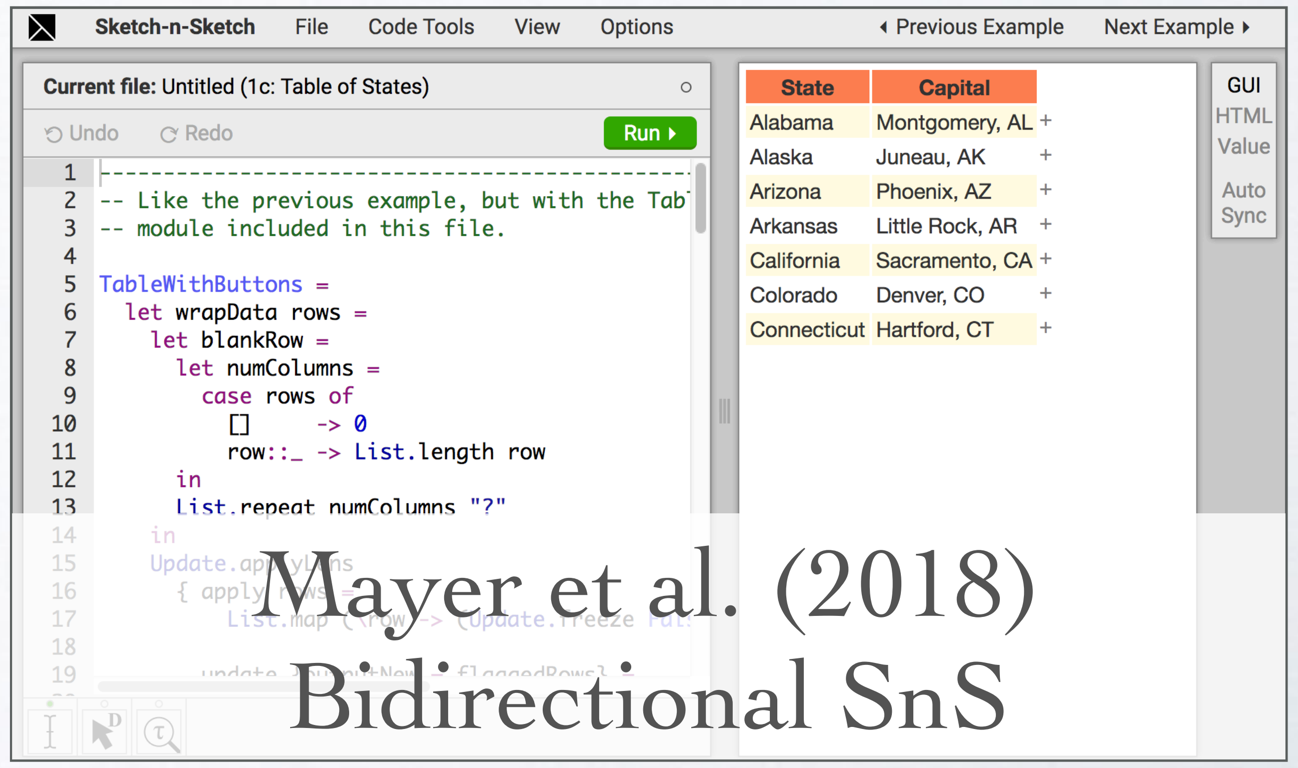


Hempel & Chugh (2016)
Sketch-n-Sketch 2016



```
(defcomponent
  digital-clock
  IRender
  (render
    [self props submorphs]
    (rectangle
      {:id "rectangle",
       :wants-hand-focus? true,
       :border-color "black",
       :fill "grey",
       :border-radius 10,
       :drop-shadow? false}
      (text
        {:rotation 5.502689183085366,
         :drop-shadow? false,
         :font-size 22.400000000000002,
         :text-color "limegreen",
```

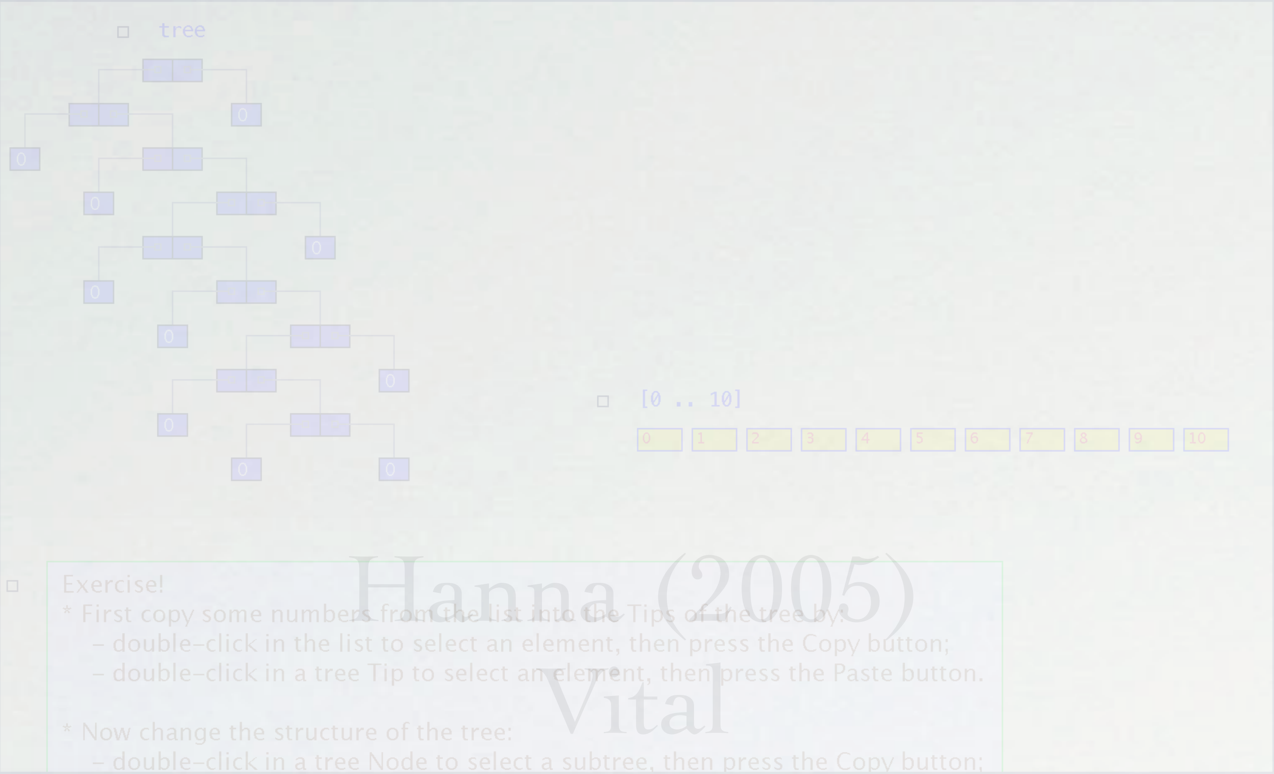
Schreiber et al. (2017)
Transmorphic

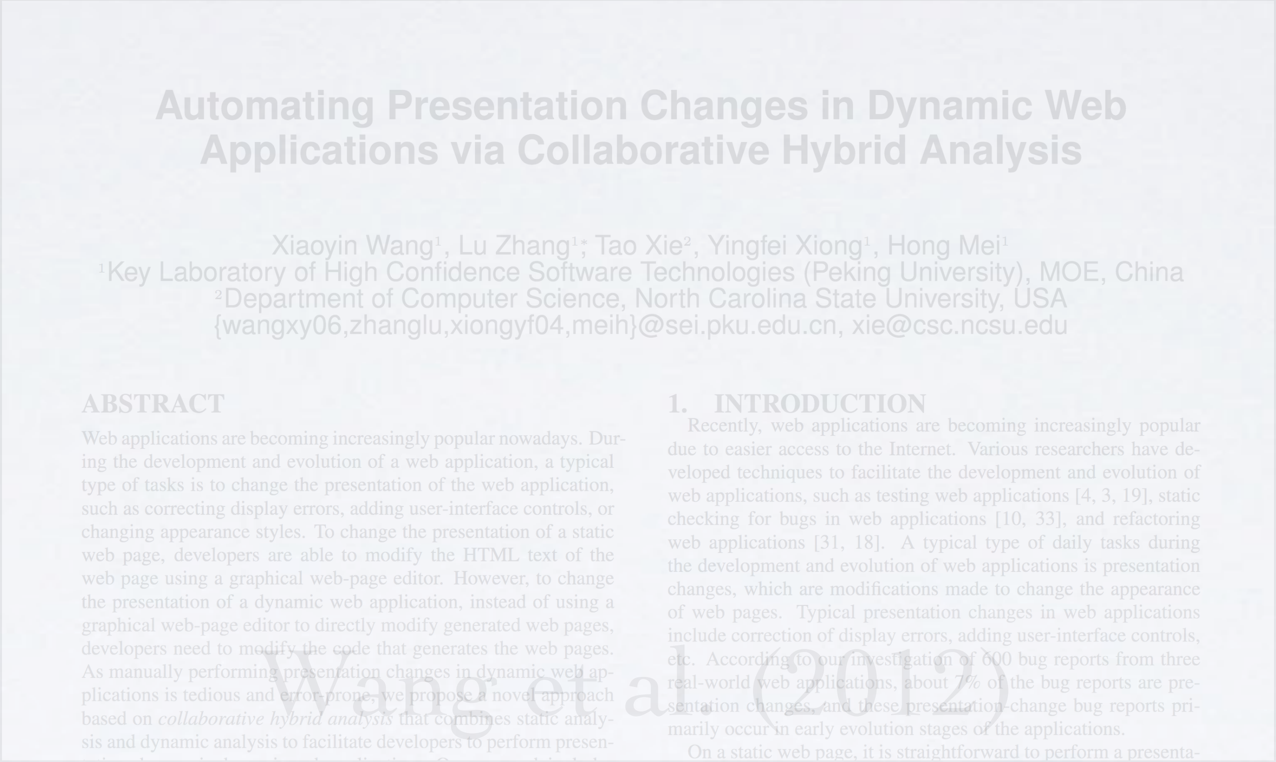


State	Capital
Alabama	Montgomery, AL
Alaska	Juneau, AK
Arizona	Phoenix, AZ
Arkansas	Little Rock, AR
California	Sacramento, CA
Colorado	Denver, CO
Connecticut	Hartford, CT

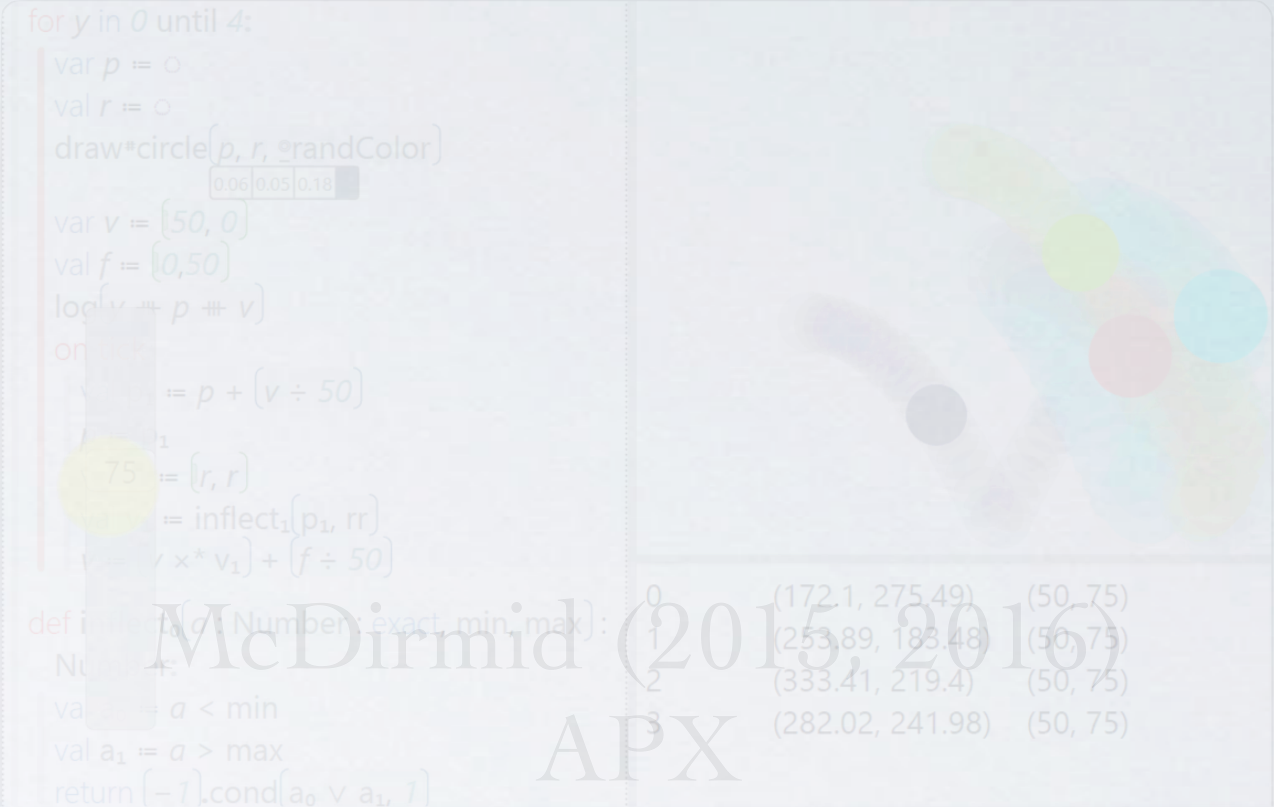
Mayer et al. (2018)
Bidirectional SnS

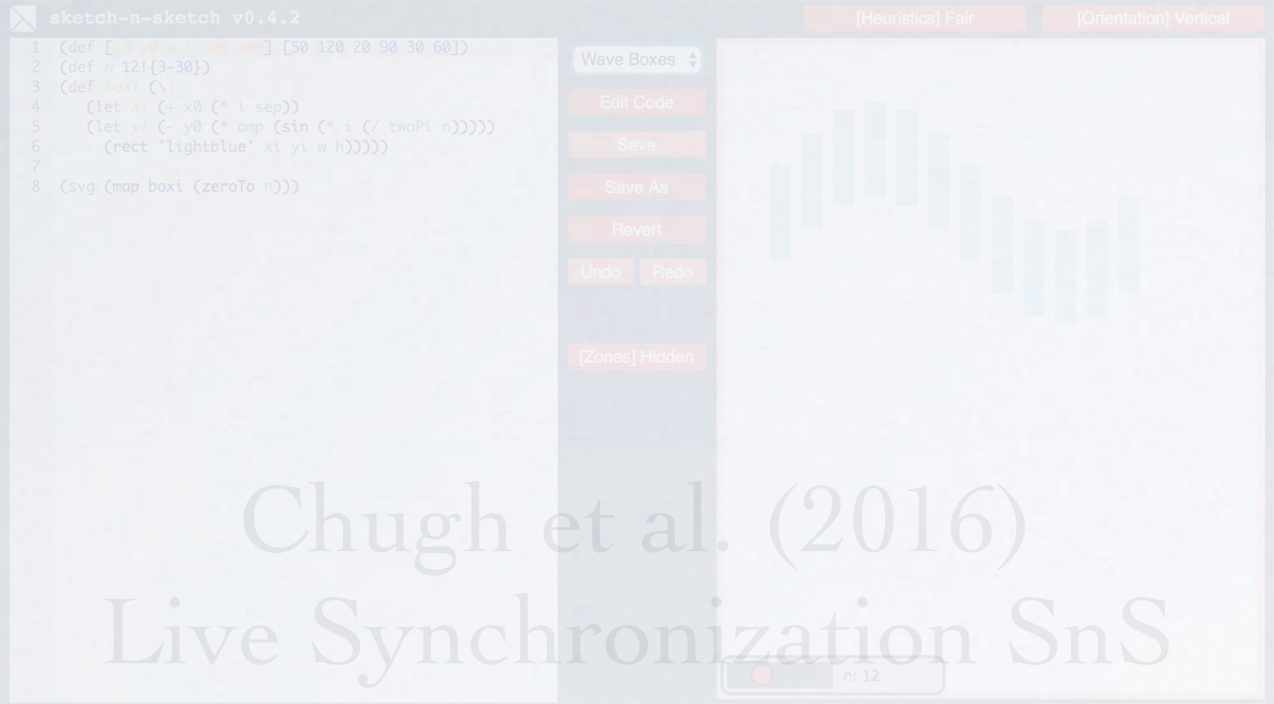
Prior Output-Directed Programming


Hanna (2005)
Vital

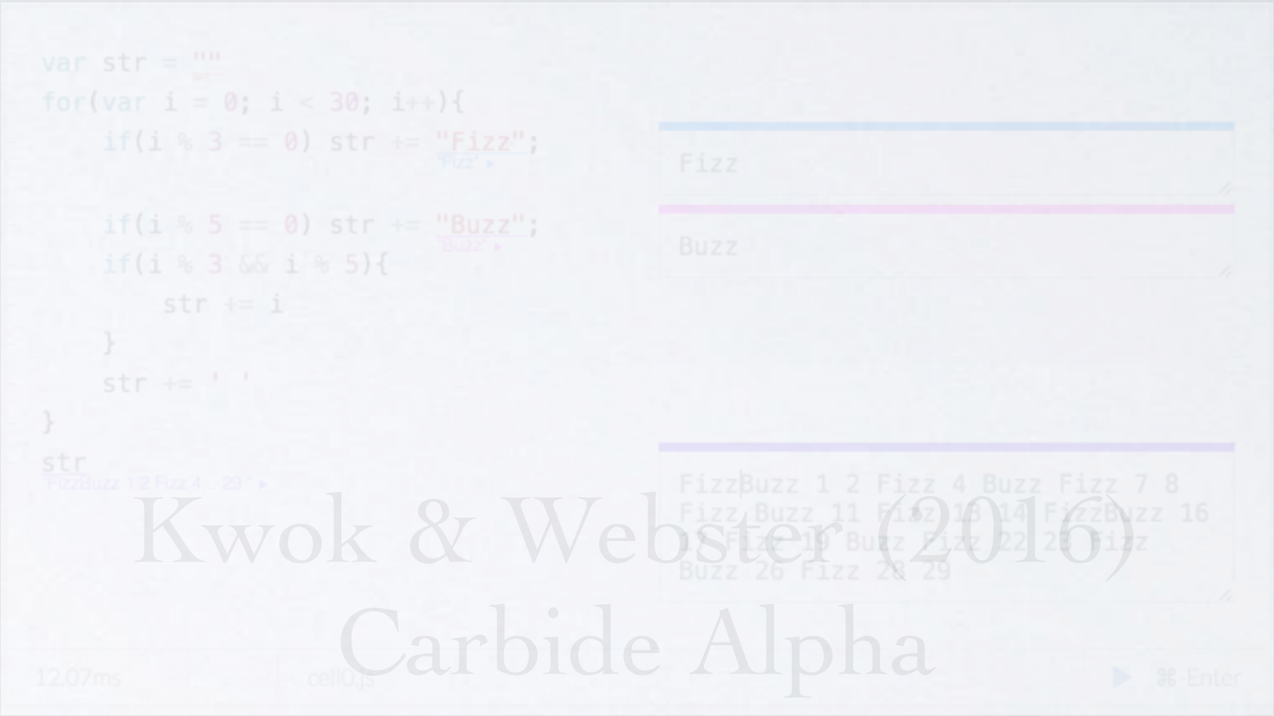

Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

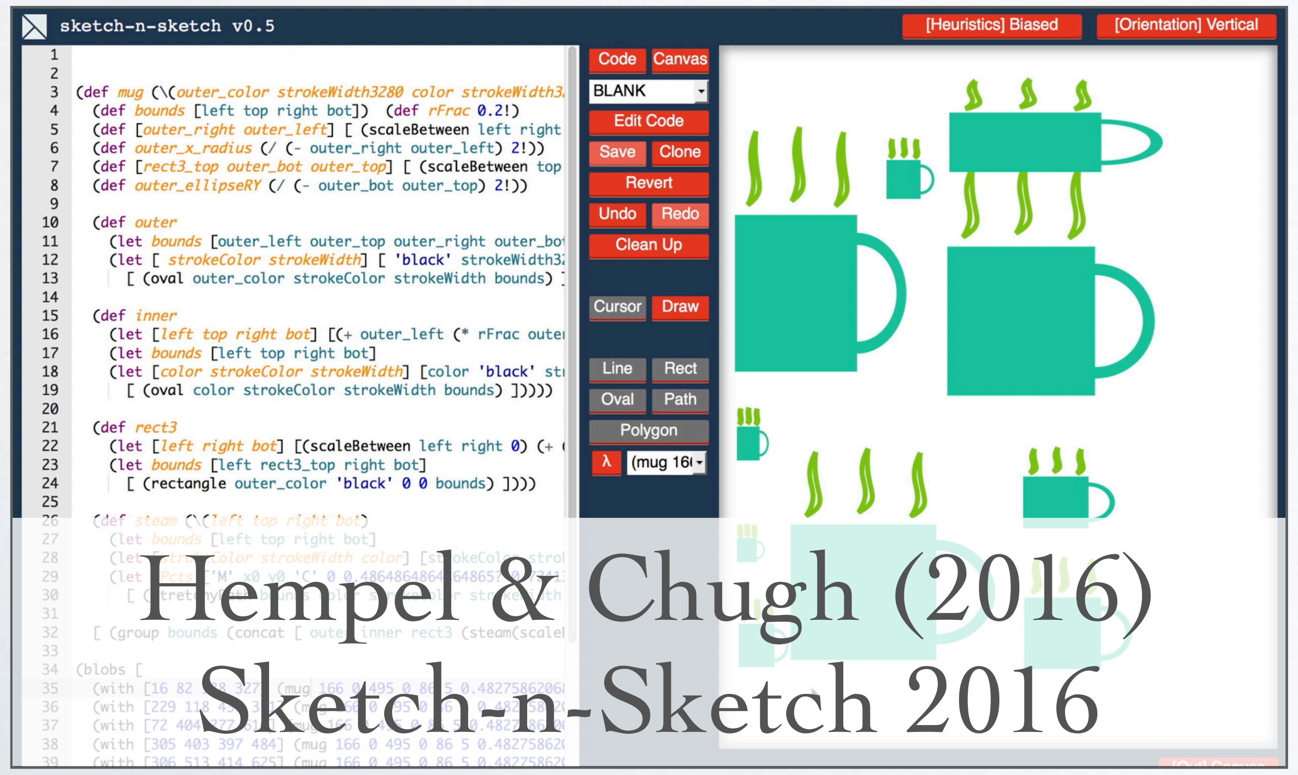
Wang et al. (2012)

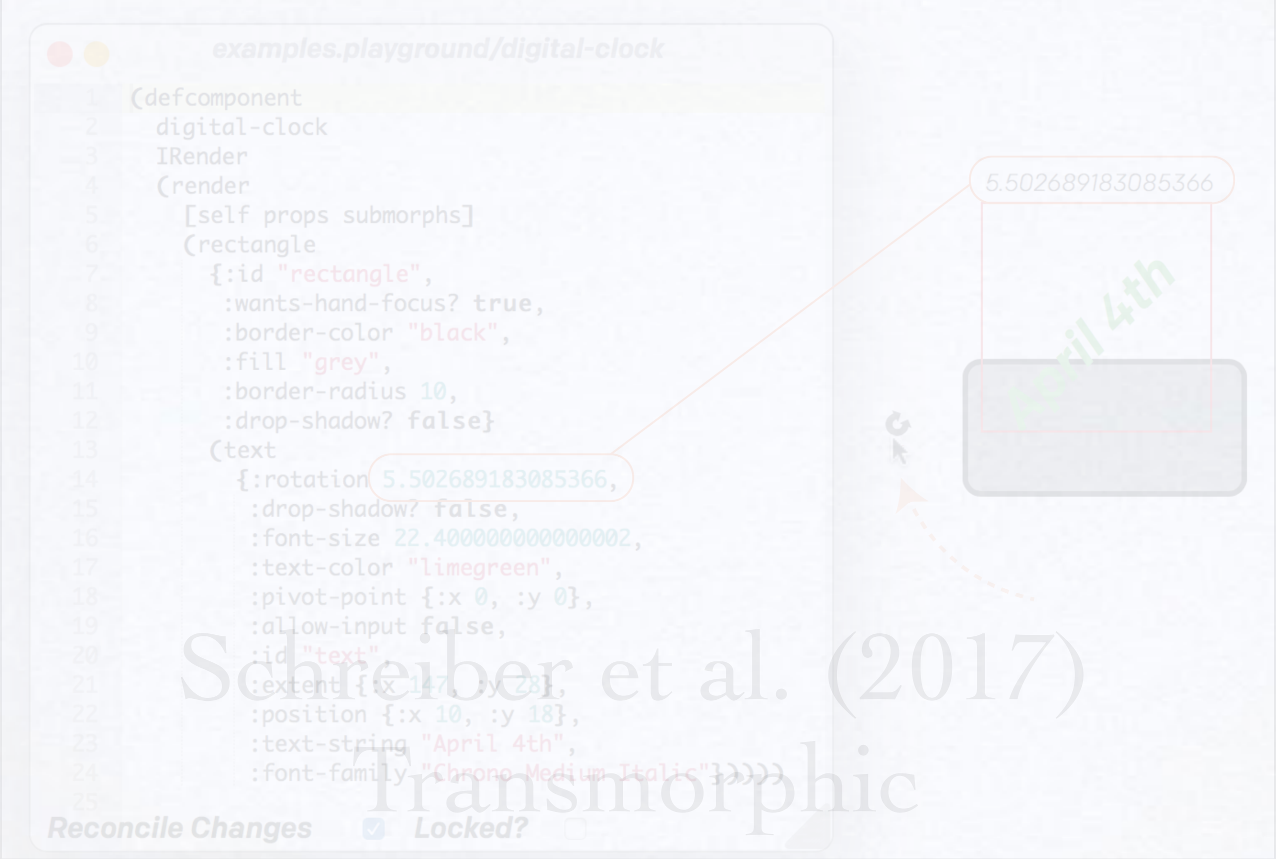

McDirmid (2013, 2016)
APX

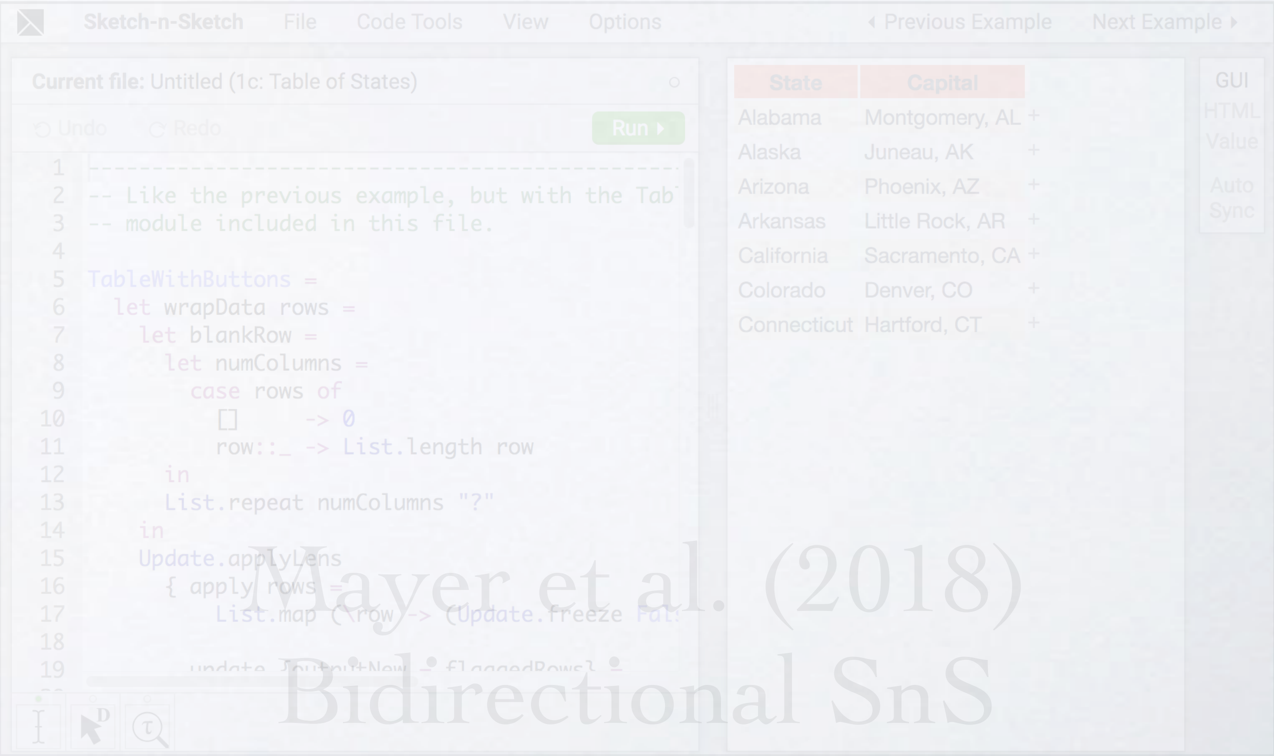

Chugh et al. (2016)
Live Synchronization SnS

Schuster & Flanagan (2016)

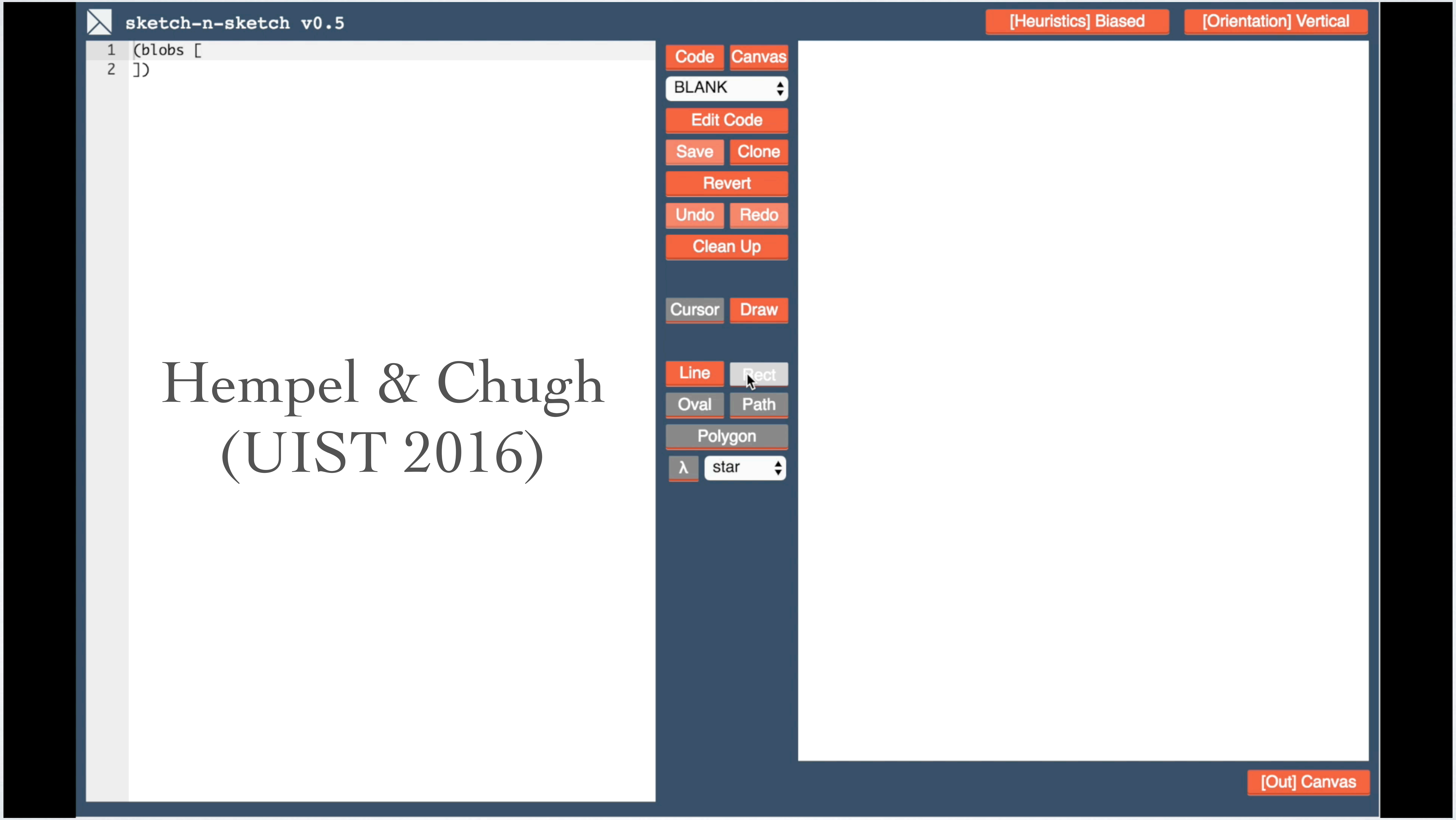

Kwok & Webster (2016)
Carbide Alpha


Hempel & Chugh (2016)
Sketch-n-Sketch 2016


Schreiber et al. (2017)
Transmorphic


Mayer et al. (2018)
Bidirectional SnS

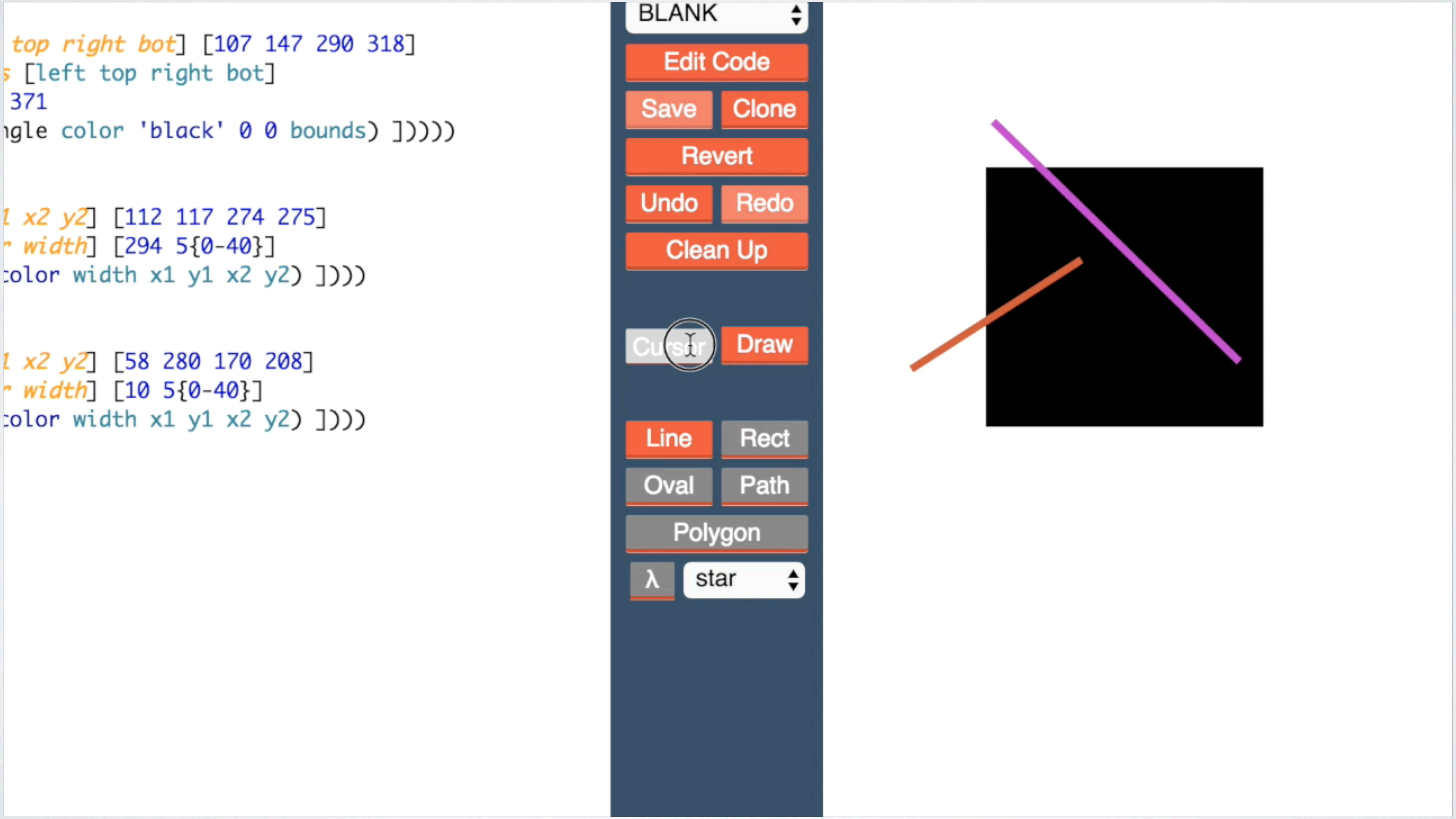
Building on Sketch-n-Sketch 2016



```
(blobs [
])
```

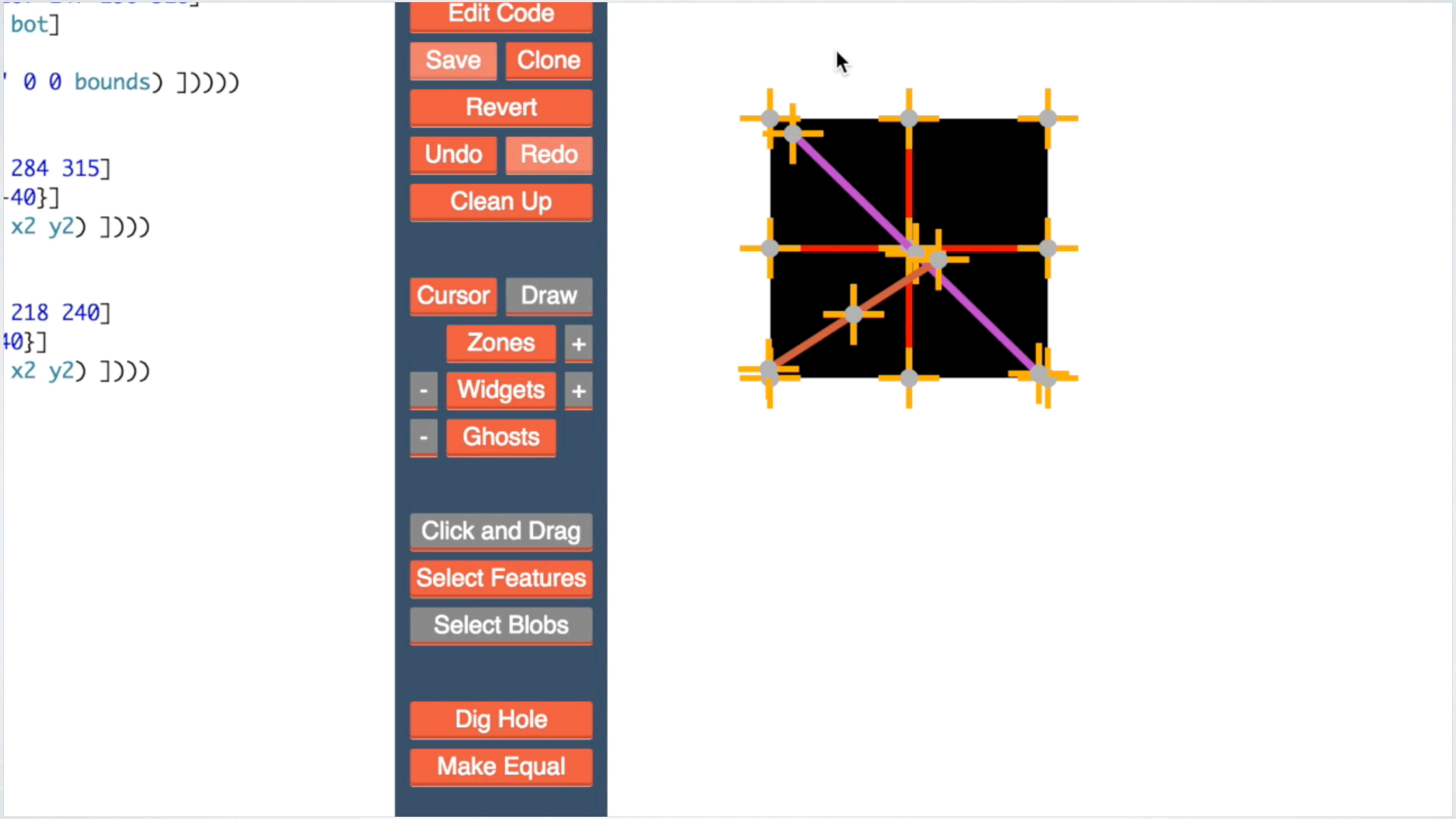
Hempel & Chugh
(UIST 2016)

Building on Sketch-n-Sketch 2016



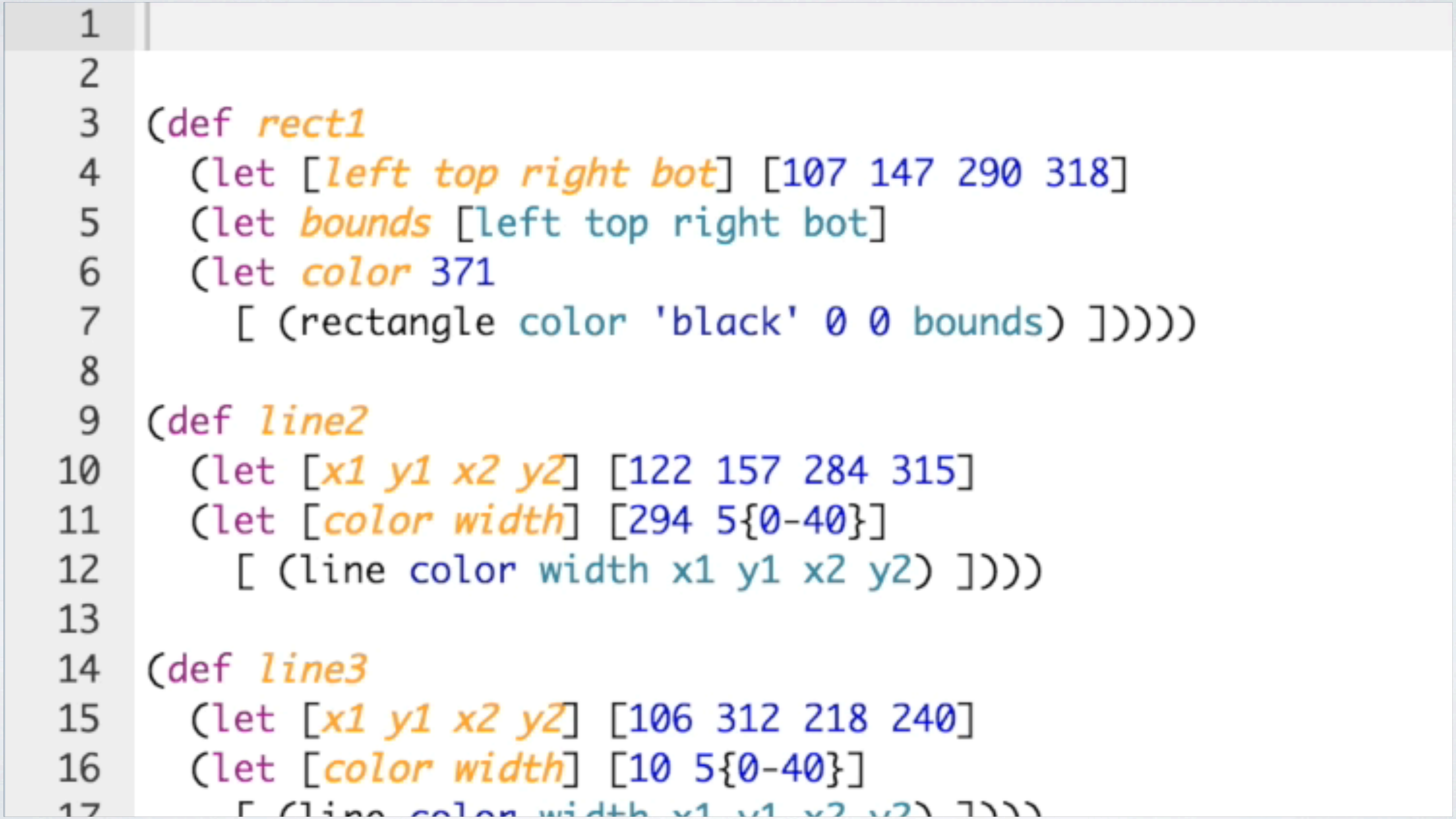
```
 top right bot] [107 147 290 318]
s [left top right bot]
 371
ngle color 'black' 0 0 bounds) ]))))


1 x2 y2] [112 117 274 275]
r width] [294 5{0-40}]
color width x1 y1 x2 y2) ])))


1 x2 y2] [58 280 170 208]
r width] [10 5{0-40}]
color width x1 y1 x2 y2) ])))
```

Building on Sketch-n-Sketch 2016



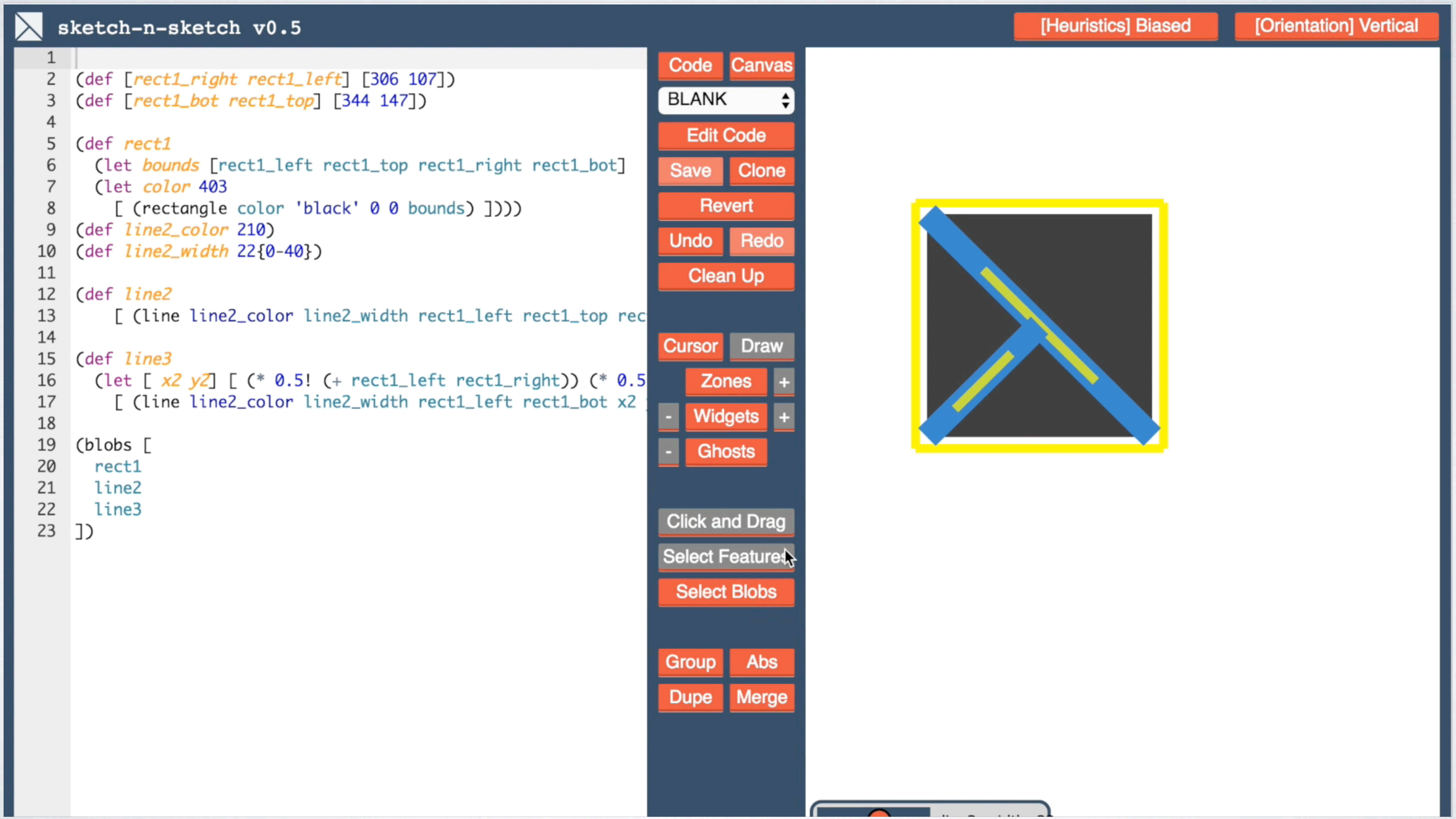
```
bot]

' 0 0 bounds) ]))))

284 315]
-40}]
x2 y2) ])))

218 240]
40}]
x2 y2) ])))
```

Building on Sketch-n-Sketch 2016



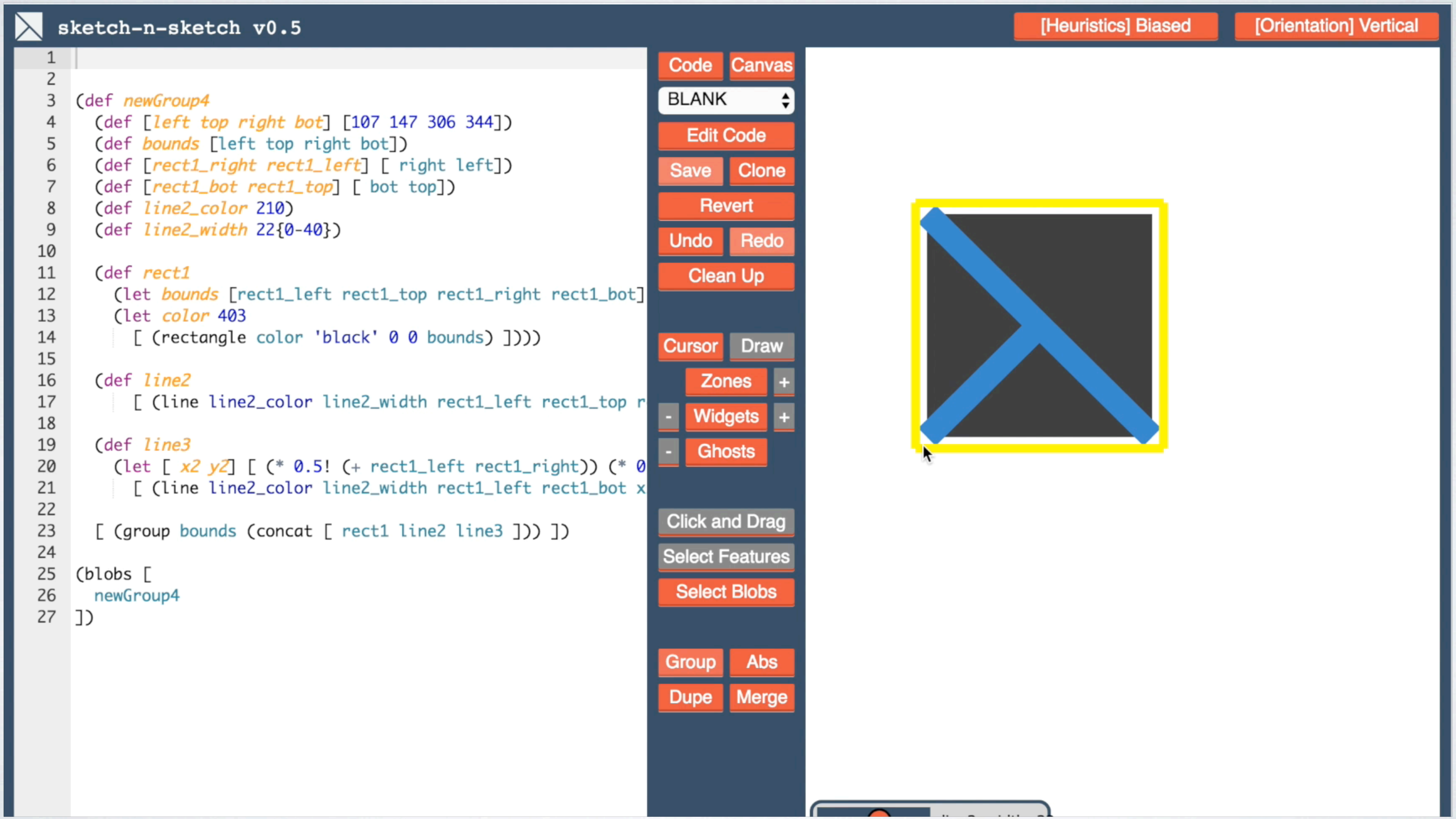
```

(def rect1
  (let [left top right bot] [107 147 290 318]
  (let bounds [left top right bot]
  (let color 371
    [ (rectangle color 'black' 0 0 bounds) ]))))

(def line2
  (let [x1 y1 x2 y2] [122 157 284 315]
  (let [color width] [294 5{0-40}]
    [ (line color width x1 y1 x2 y2) ])))

(def line3
  (let [x1 y1 x2 y2] [106 312 218 240]
  (let [color width] [10 5{0-40}]
    [ (line color width x1 y1 x2 y2) ])))
```

Building on Sketch-n-Sketch 2016



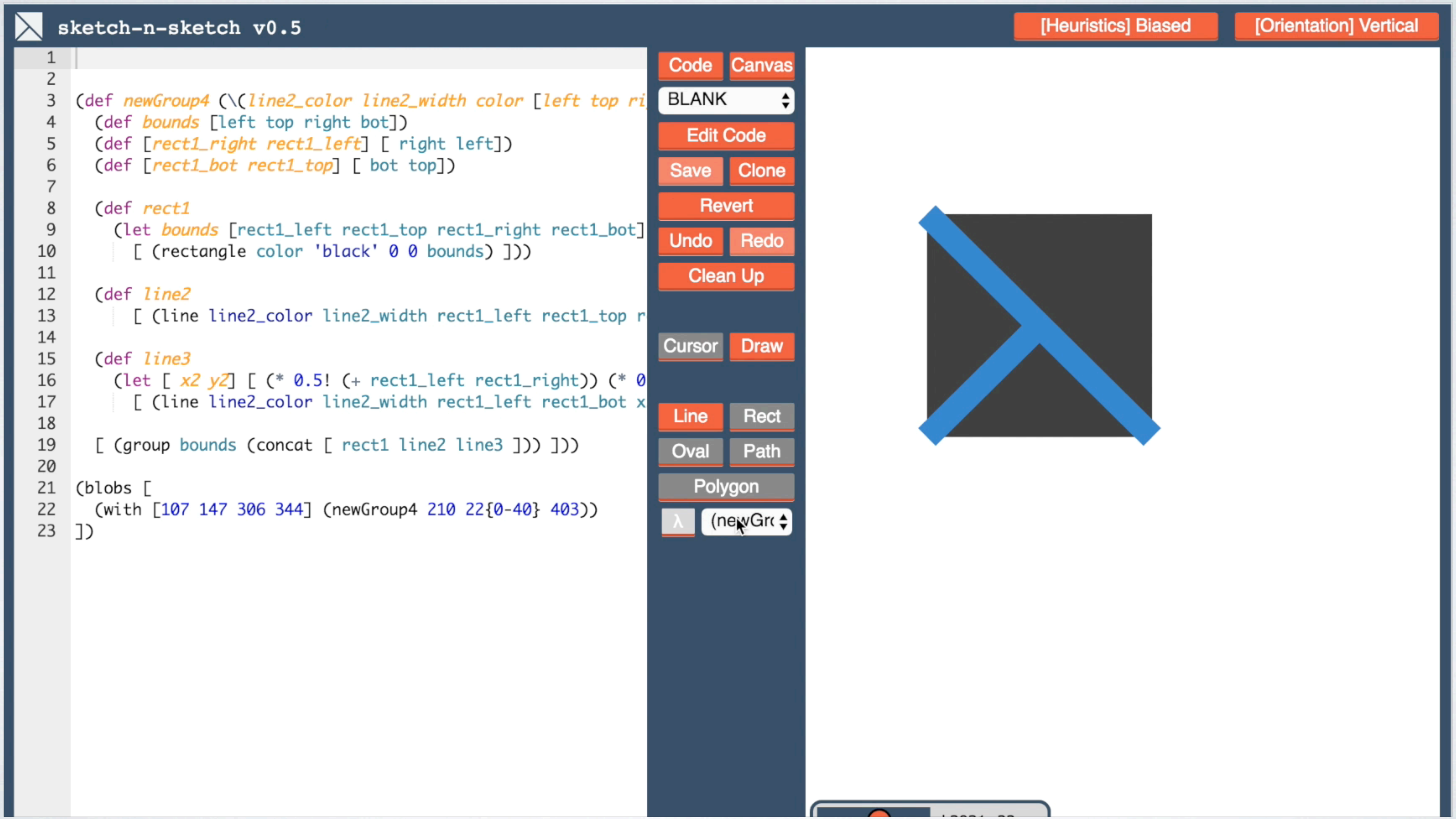
```

(def [rect1_right rect1_left] [306 107])
(def [rect1_bot rect1_top] [344 147])

(def rect1
  (let bounds [rect1_left rect1_top rect1_right rect1_bot]
  (let color 403
    [ (rectangle color 'black' 0 0 bounds) ])))
(def line2_color 210)
(def line2_width 22{0-40})

(def line2
    [ (line line2_color line2_width rect1_left rect1_top rec

(def line3
  (let [ x2 y2] [ (* 0.5! (+ rect1_left rect1_right)) (* 0.5
    [ (line line2_color line2_width rect1_left rect1_bot x2

(blobs [
  rect1
  line2
  line3
])
```

Building on Sketch-n-Sketch 2016



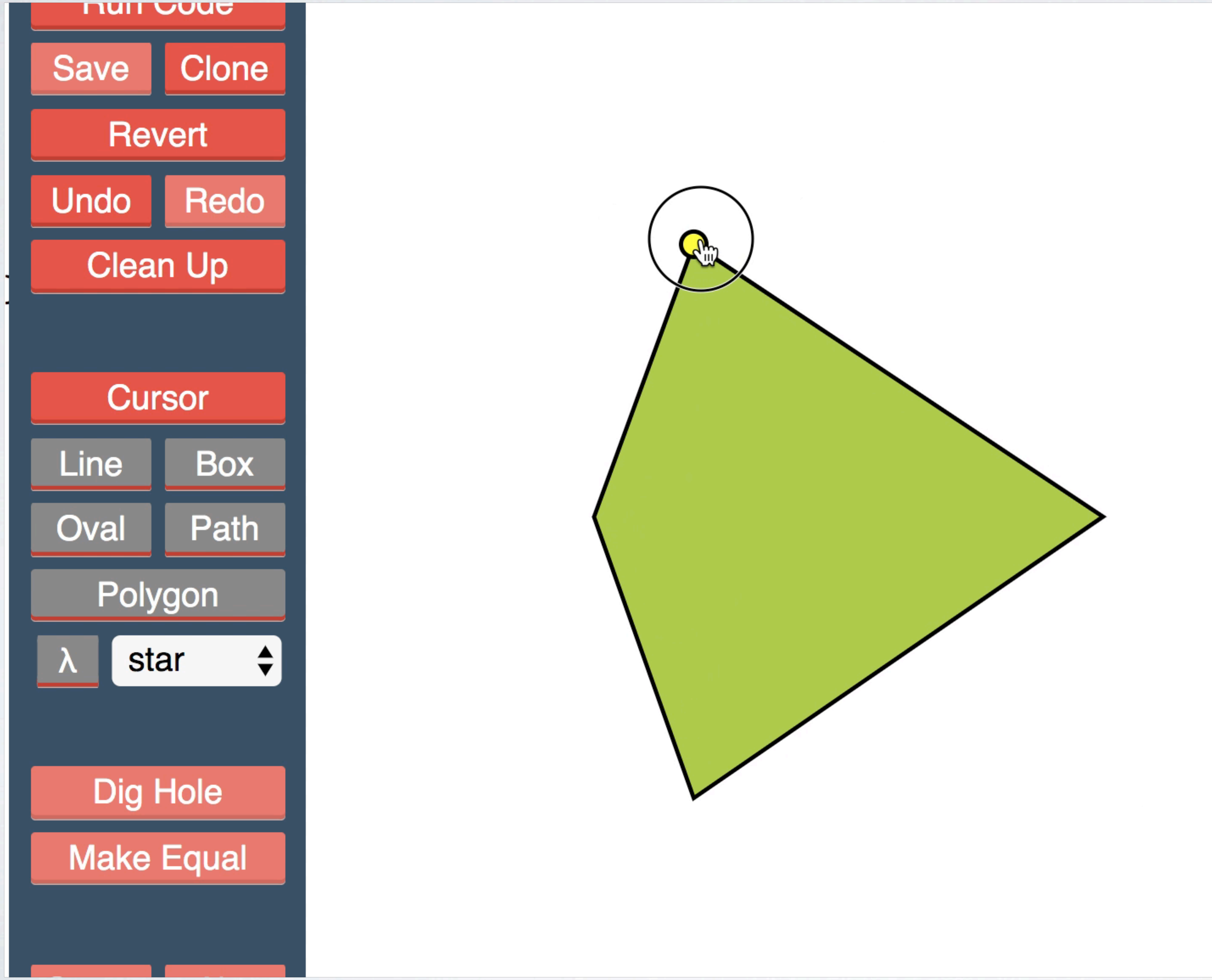
```


(def newGroup4
  (def [left top right bot] [107 147 306 344])
  (def bounds [left top right bot])
  (def [rect1_right rect1_left] [ right left])
  (def [rect1_bot rect1_top] [ bot top])
  (def line2_color 210)
  (def line2_width 22{0-40})

  (def rect1
    (let bounds [rect1_left rect1_top rect1_right rect1_bot]
    (let color 403
      [ (rectangle color 'black' 0 0 bounds) ])))

  (def line2
      [ (line line2_color line2_width rect1_left rect1_top r

  (def line3
    (let [ x2 y2] [ (* 0.5! (+ rect1_left rect1_right)) (* 0
      [ (line line2_color line2_width rect1_left rect1_bot x

  [ (group bounds (concat [ rect1 line2 line3 ])) ])

(blobs [
  newGroup4
])
```

Building on Sketch-n-Sketch 2016

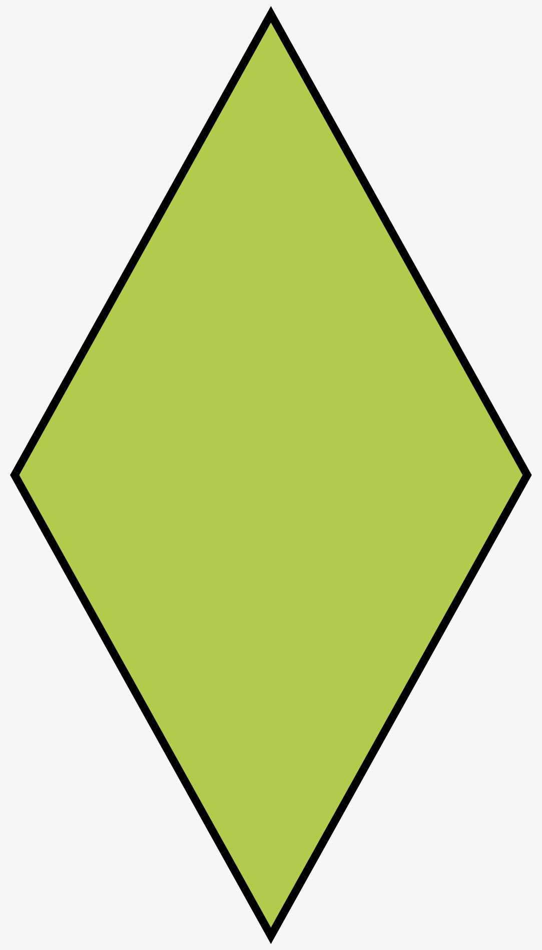


```

(def newGroup4 (\(line2_color line2_width color [left top ri
  (def bounds [left top right bot])
  (def [rect1_right rect1_left] [ right left])
  (def [rect1_bot rect1_top] [ bot top])

  (def rect1
    (let bounds [rect1_left rect1_top rect1_right rect1_bot]
      [ (rectangle color 'black' 0 0 bounds) ]))

  (def line2
      [ (line line2_color line2_width rect1_left rect1_top r

  (def line3
    (let [ x2 y2] [ (* 0.5! (+ rect1_left rect1_right)) (* 0
      [ (line line2_color line2_width rect1_left rect1_bot x

  [ (group bounds (concat [ rect1 line2 line3 ])) ]))

(blobs [
  (with [107 147 306 344] (newGroup4 210 22{0-40} 403))
])
```

Building on Sketch-n-Sketch 2016



Want a rhombus:



But tools overly rigid!

Big Q

What kinds of programs can be constructed *entirely* through output manipulations?

Contribution

UI Insight

DM on More Than Output!

Intermediate Value Widgets

Expression Focusing

PL Insight

Generic Tools, Too!

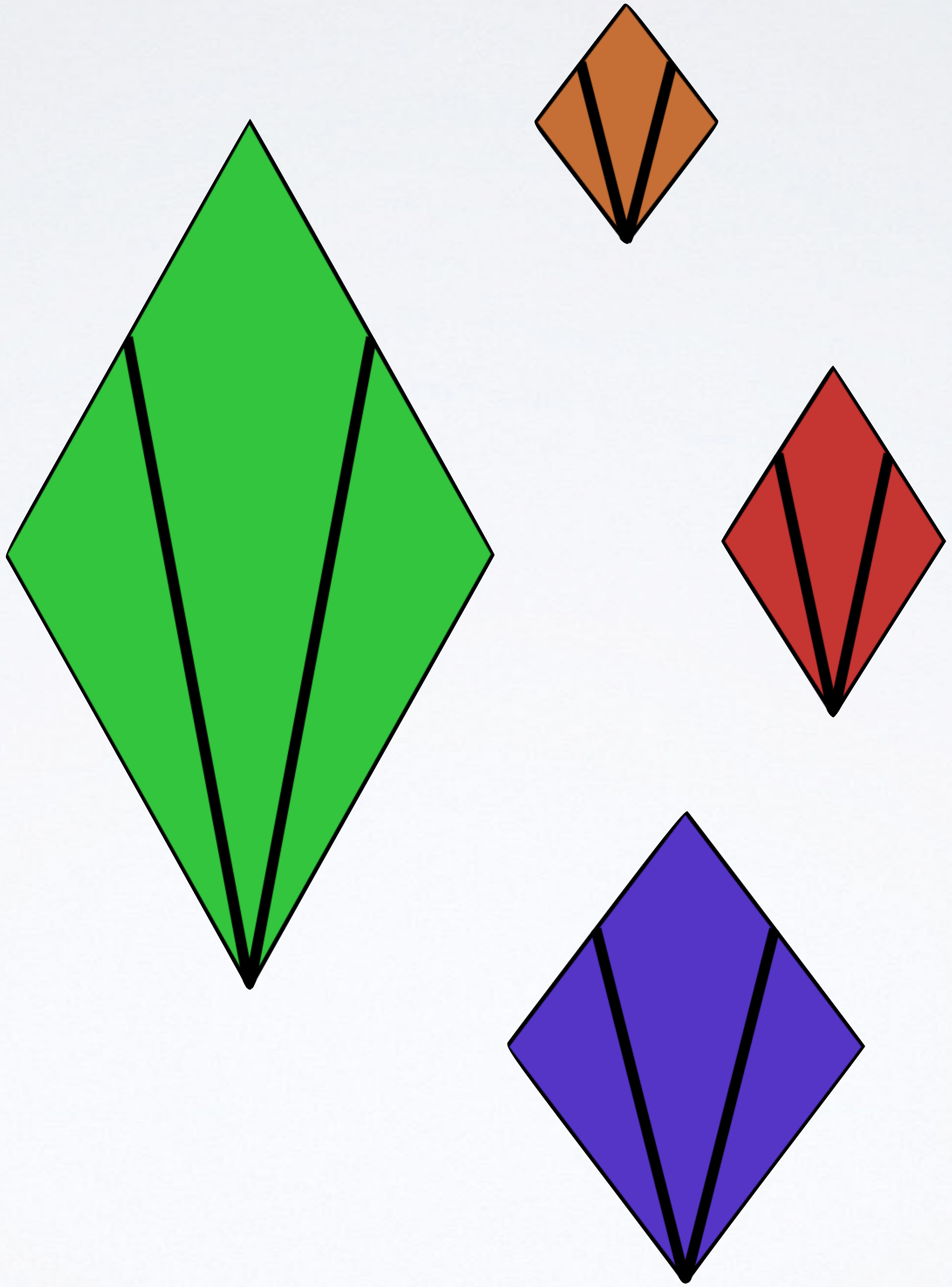
Generic Provenace Tracing

See Paper

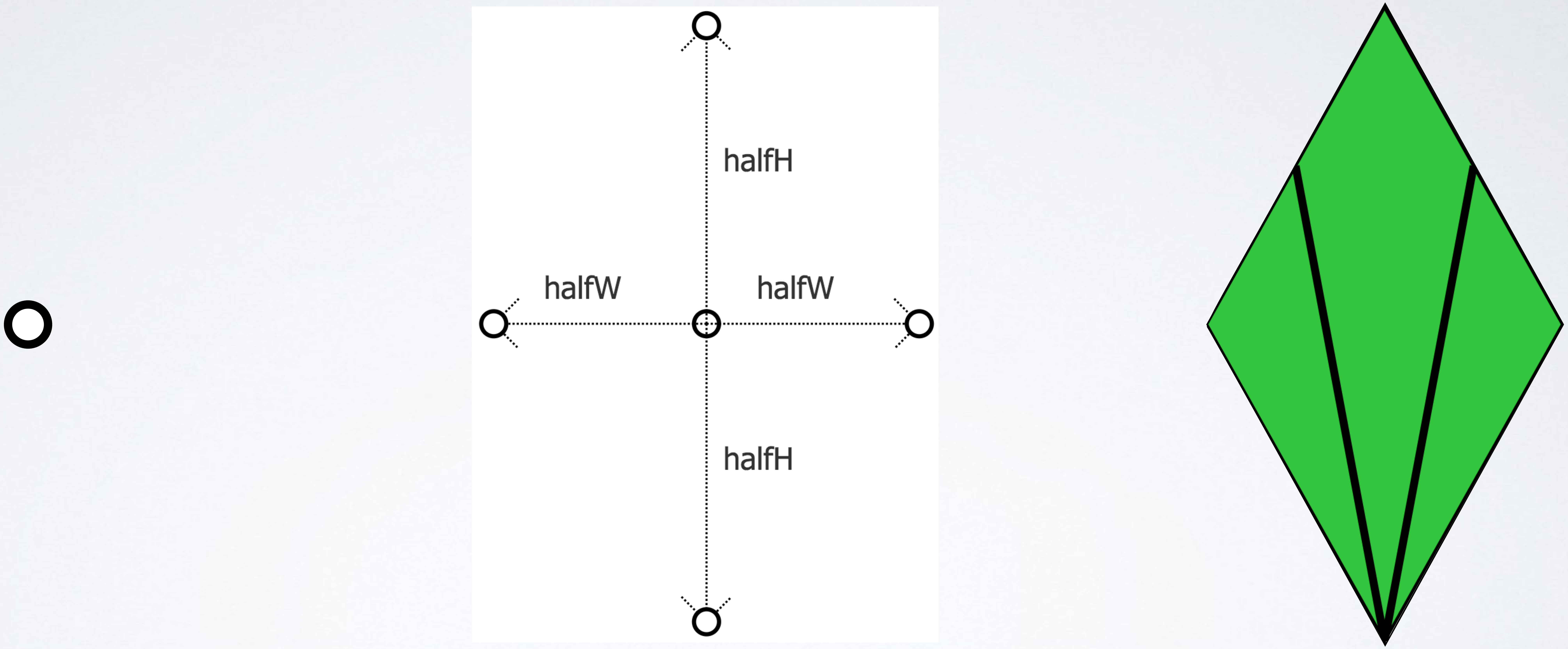
(but not SVG-specific)

Generic Refactorings

Demo



Rhombus with Veins

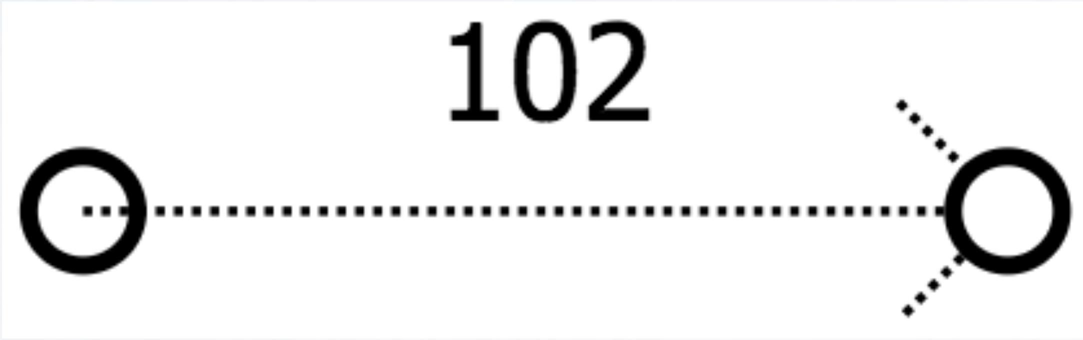


Widgets for Intermediate Values

Points

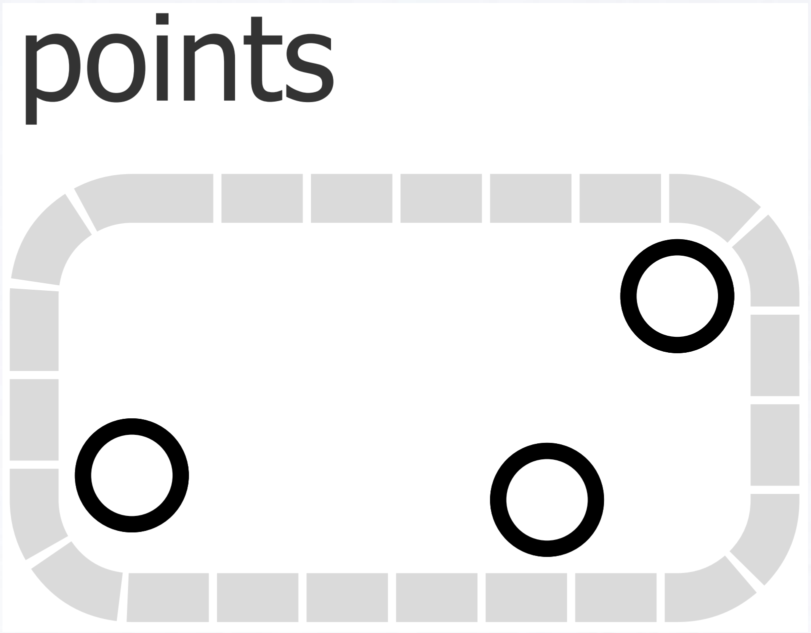
`[79, 89]`

Offsets



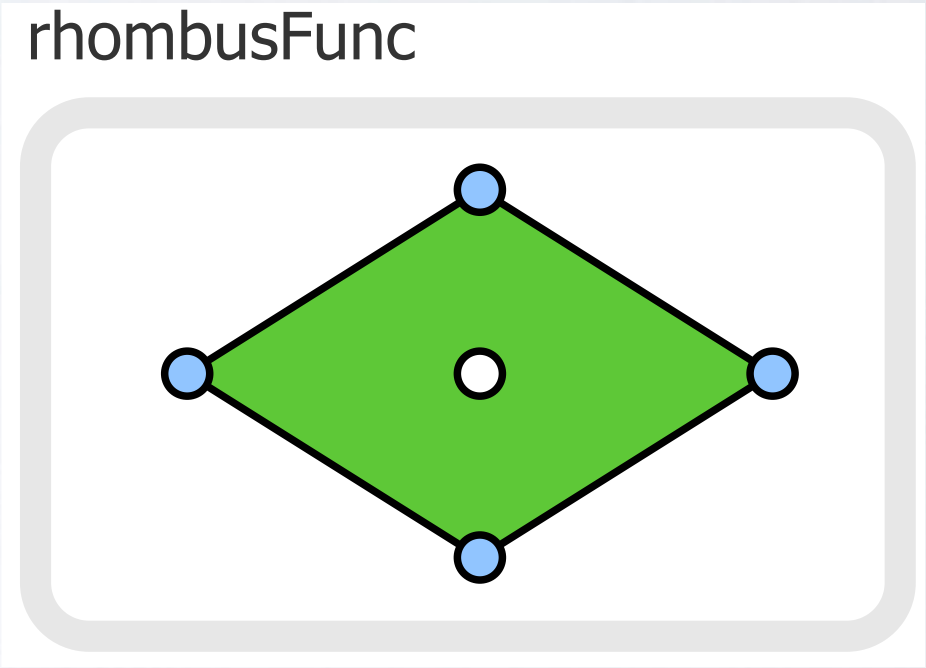
`x + 102`

Lists



`[pt1, pt2, pt3]`

Calls



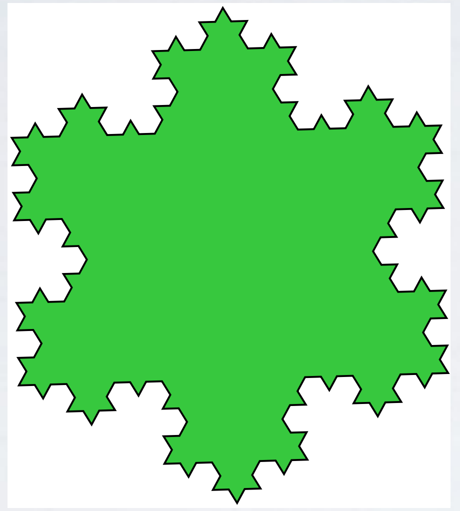
`rhombusFunc [79, 89] 49 78`

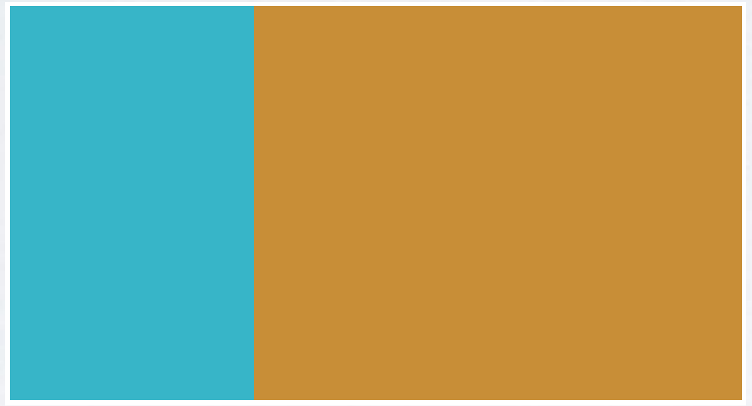
Expression Focusing + Generic Refactorings

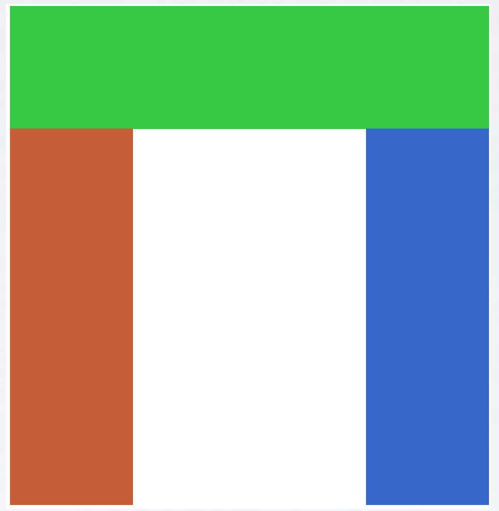
Big Q

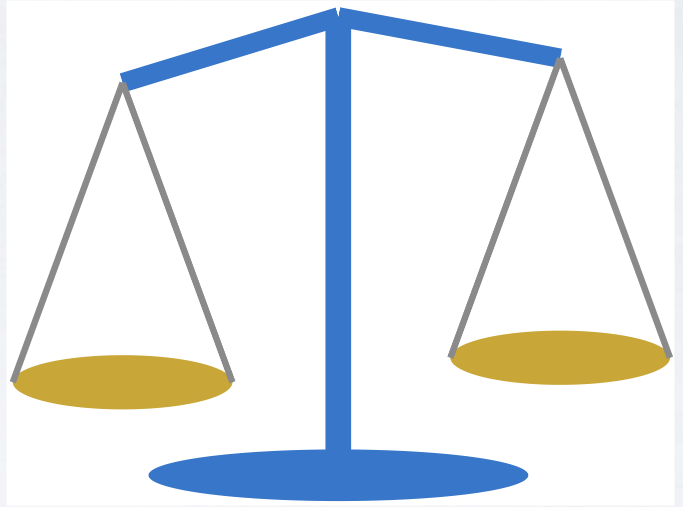
What kinds of programs can be constructed *entirely* through output manipulations?

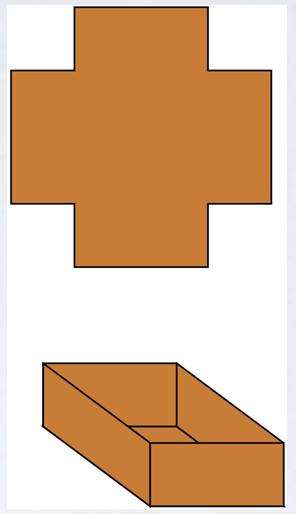
Examples

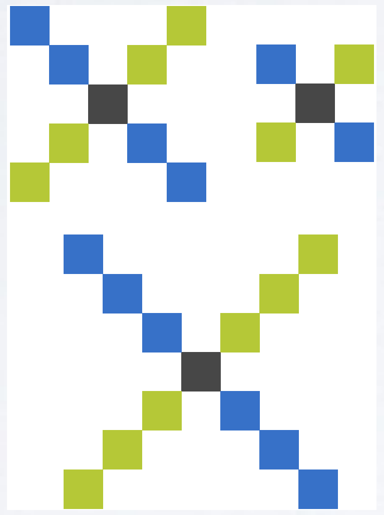

(i) Koch Snowflake

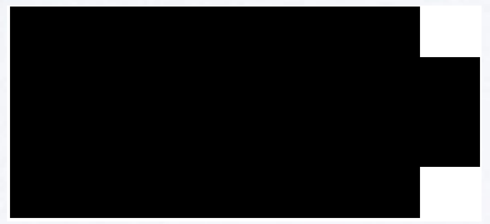

(ii) Precision Floor Plan

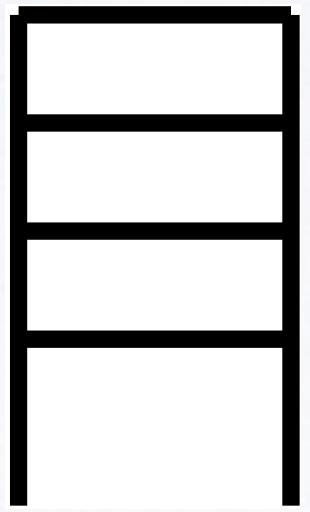

(iii) Mondrian Arch

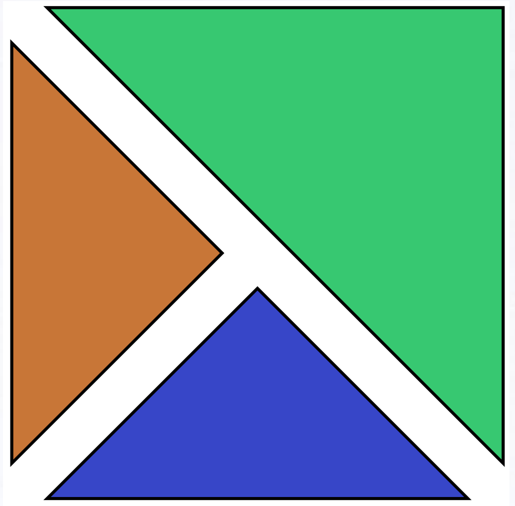

(iv) Balance Scale

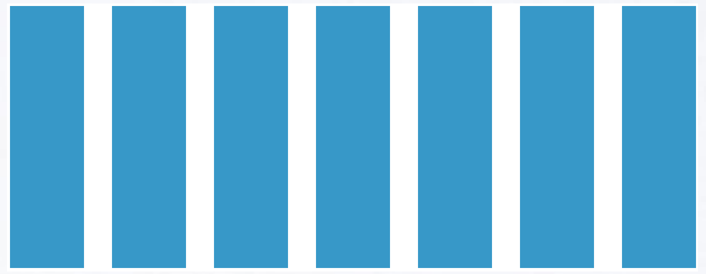

(v) Box Volume

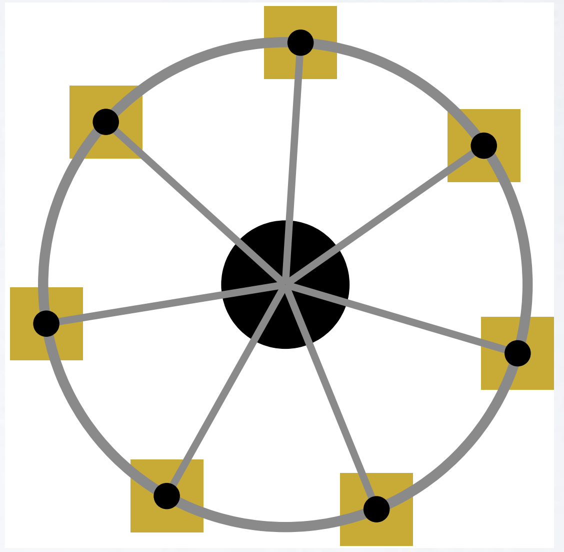

(vi) Xs

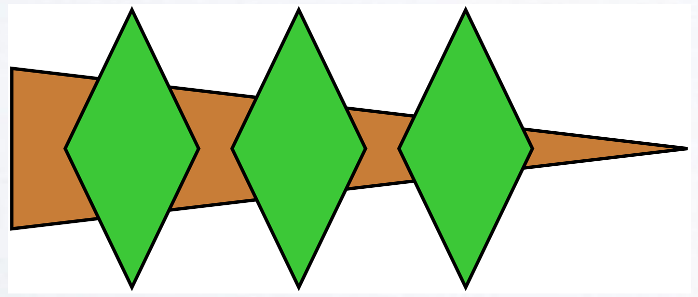

(vii) Battery

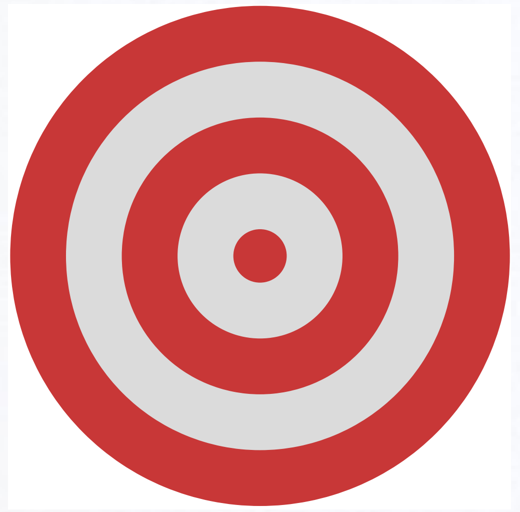

(viii) Ladder

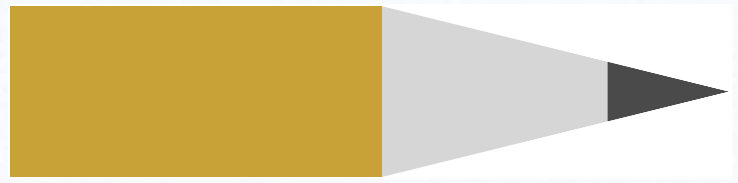

(ix) Logo (via Three Tris)

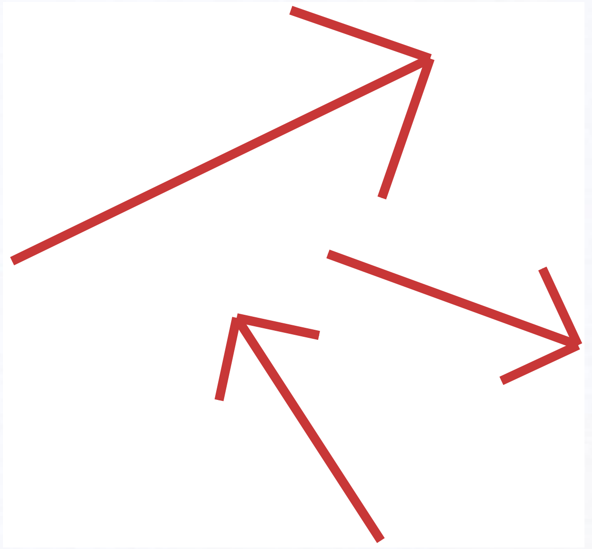

(x) N Boxes

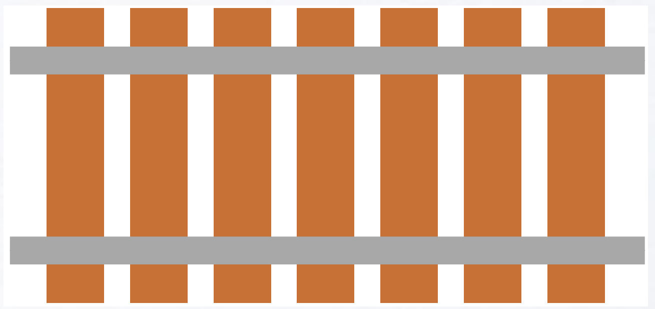

(xi) Ferris Wheel


(xii) Tree Branch

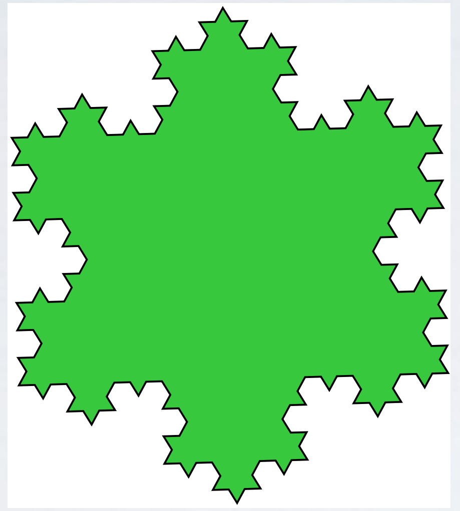

(xiii) Target


(xiv) Pencil Tip


(xv) Arrows

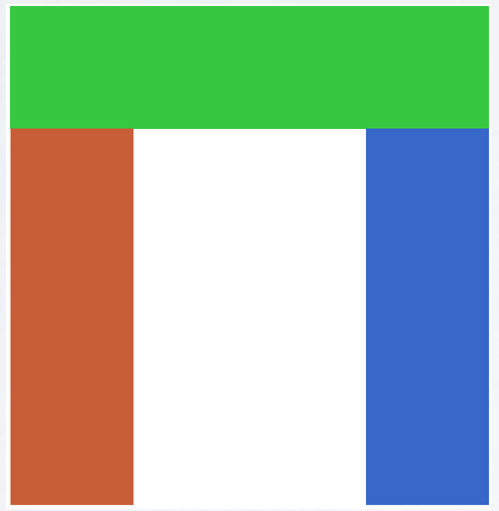

(xvi) Rails

WWID: PBD Benchmarks



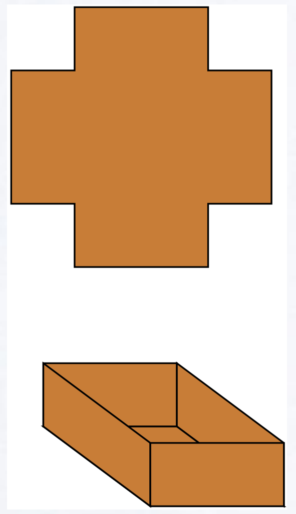
(i) Koch Snowflake

(ii) Precision Floor Plan

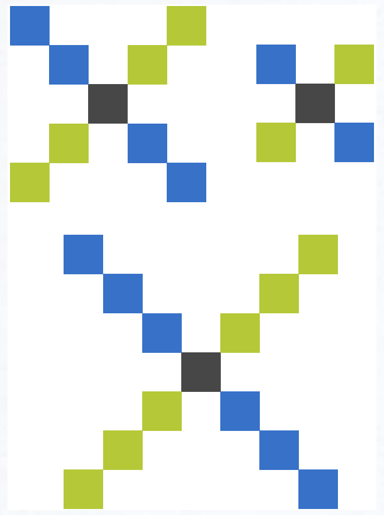


(iii) Mondrian Arch

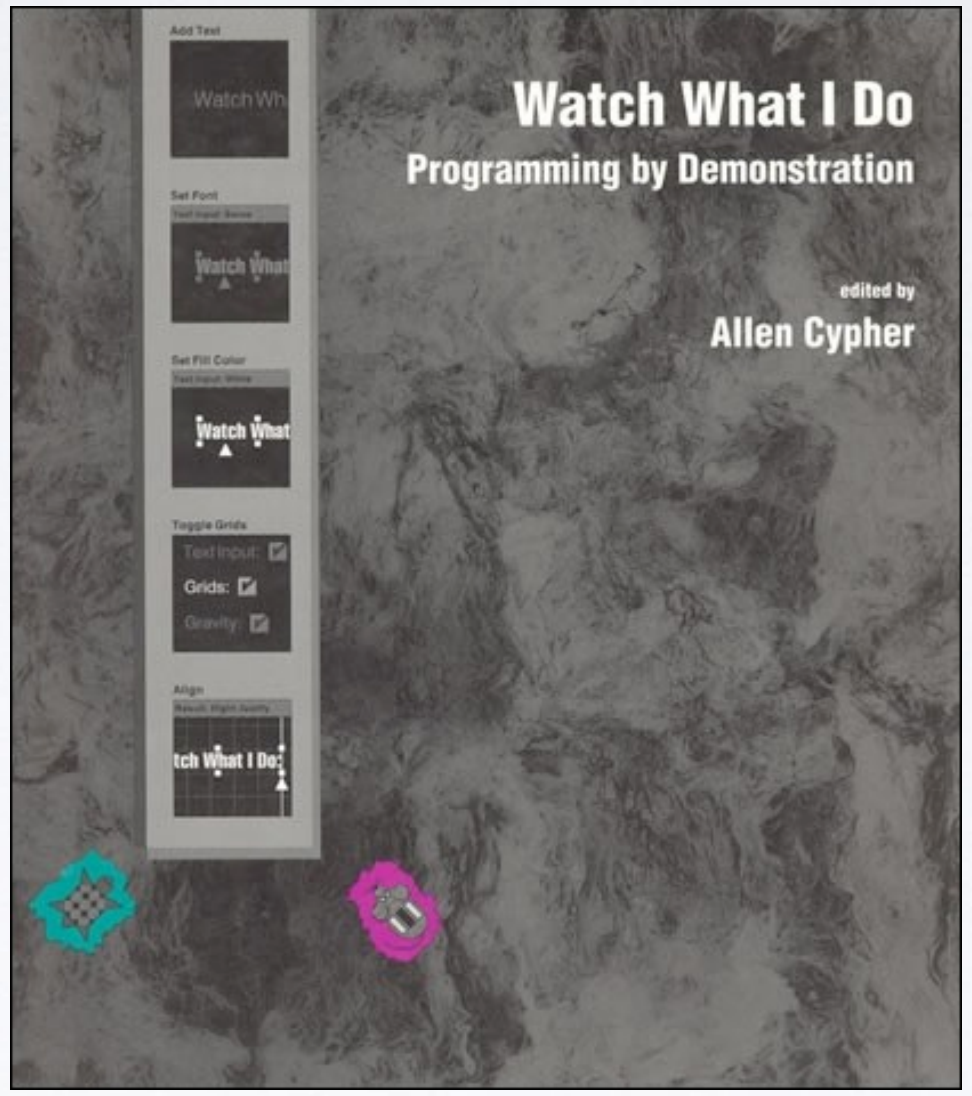
(iv) Balance Scale



(v) Box Volume



(vi) Xs



Watch What I Do:
Programming by Demonstration
Ed. Allen Cypher, 1993

Features needed for 9 remaining tasks:

- **Text boxes**
- list operations
- intersections of lines with edges
- overlapping & containment constraints
- multiple constraint solving
- arbitrary if-then-else branches

Future Work

Widget Visibility

Soooo many!

Contextual visibility only helps a little.

Change Explanation

Multiple results. Necessary, but 🙁

Better change descriptions?

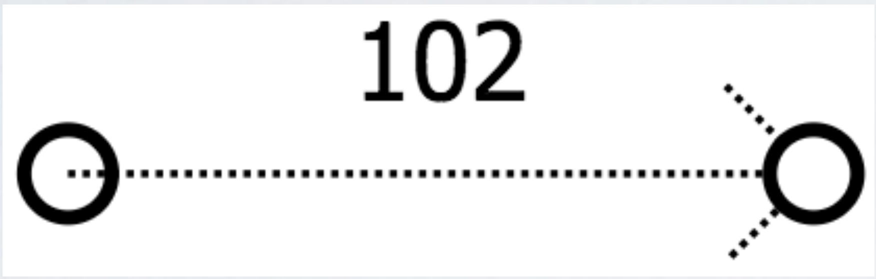
ODP for Novcies

ODP is tantalizing.

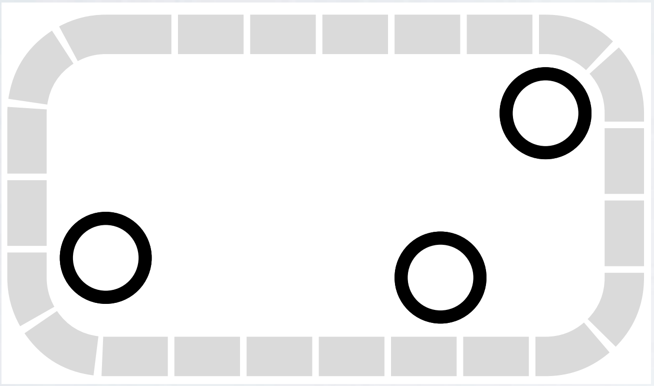
But we haven't shown it's easy.

Points

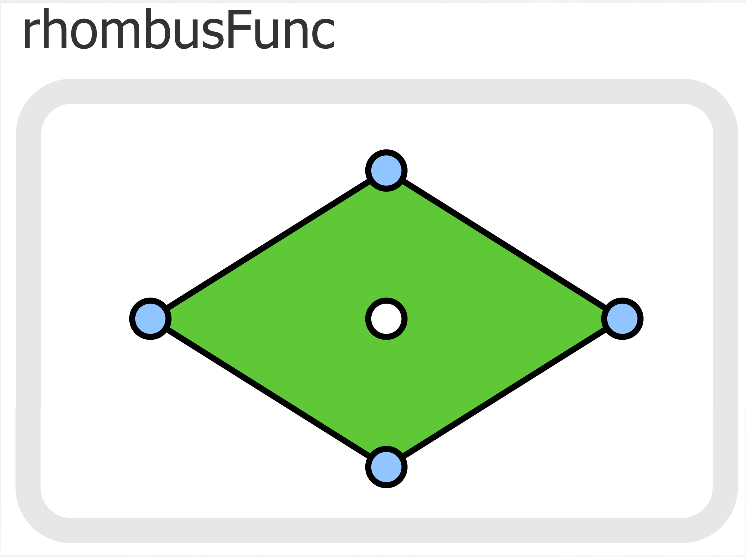
Offsets

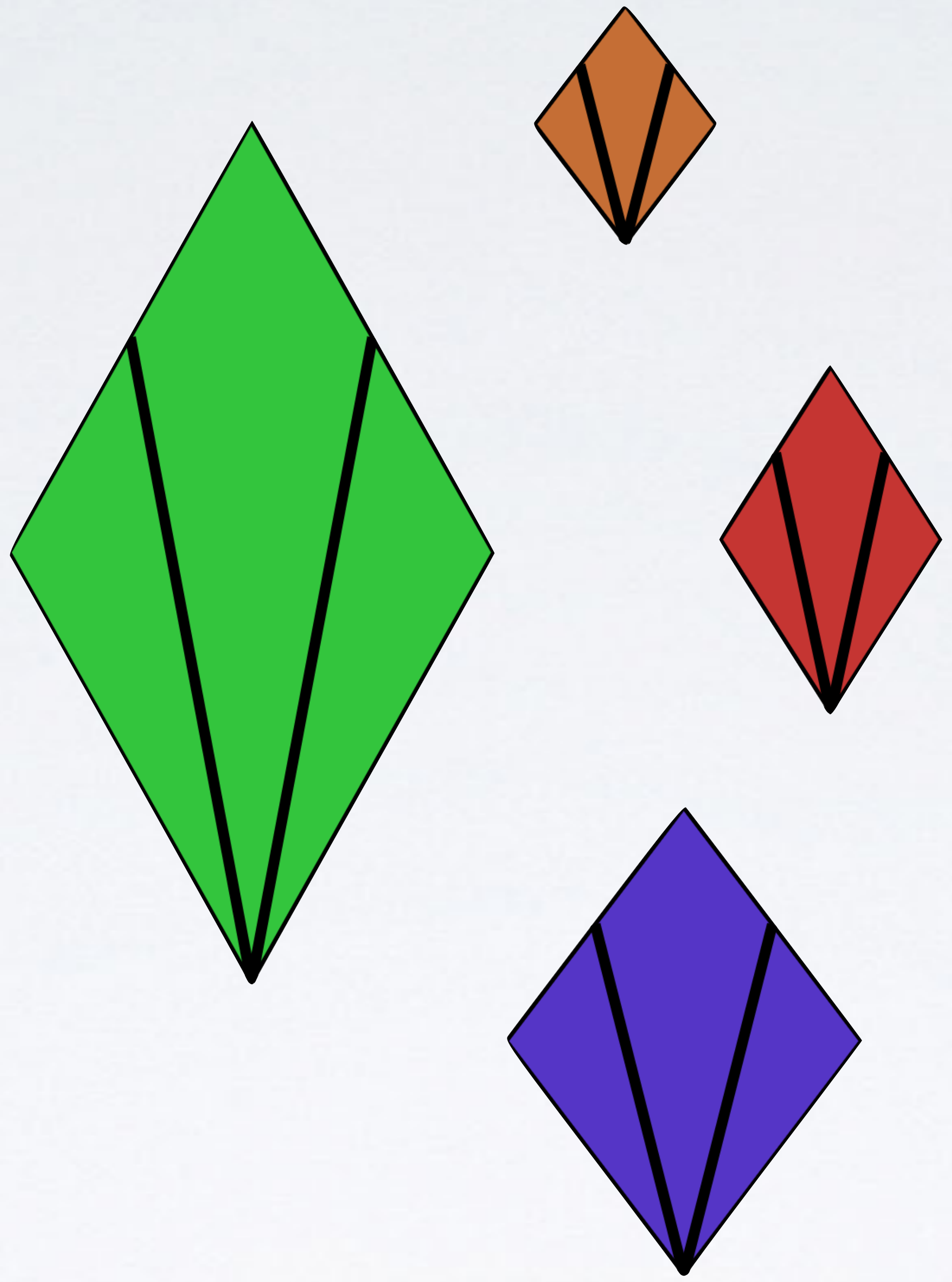


Lists



Calls





DM on More Than Output!

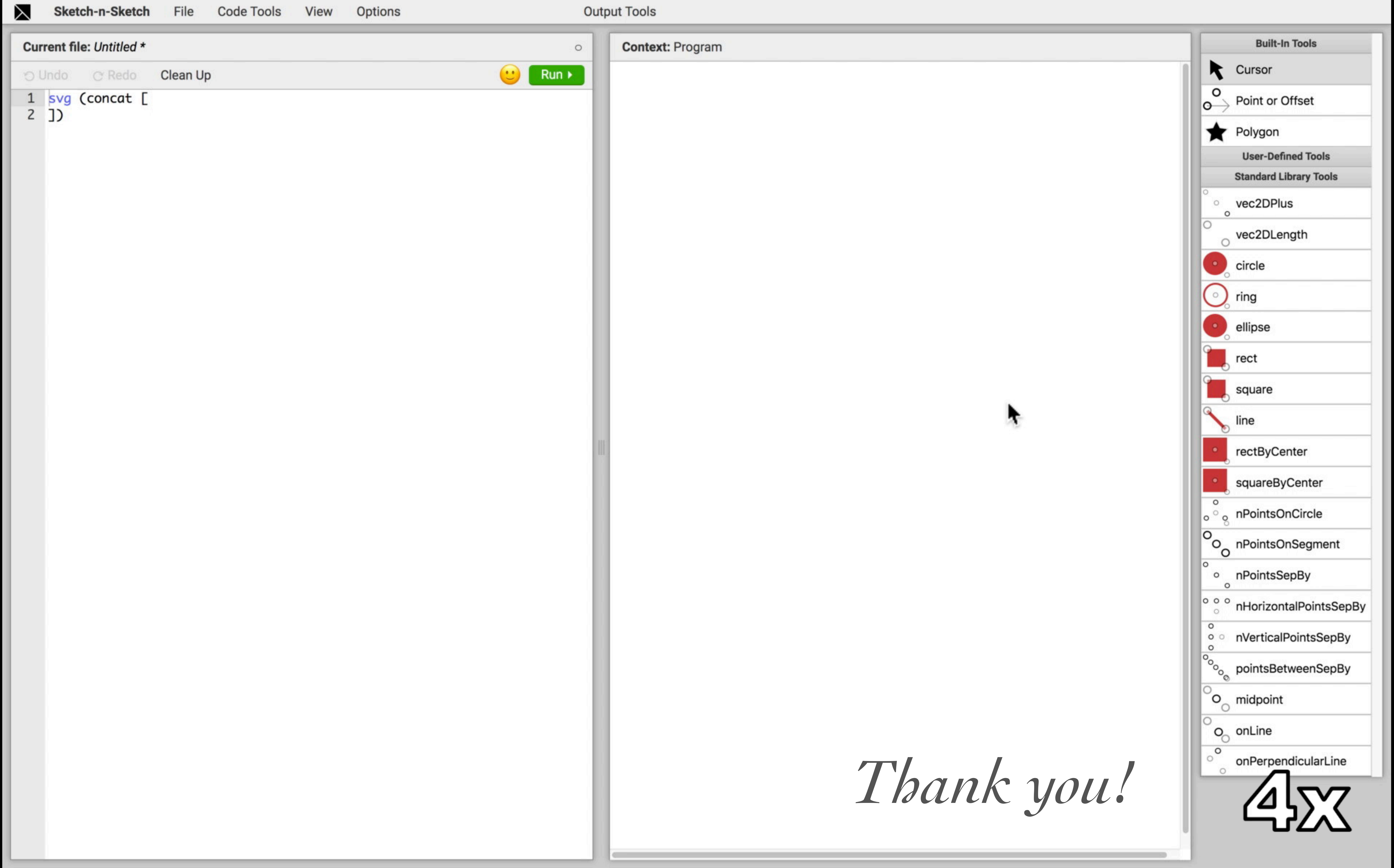
Intermediate Value Widgets
Expression Focusing

Generic Tools, Too!

Generic Refactorings
(via generic tracing)

search online for “sketch n sketch”

Thank you!

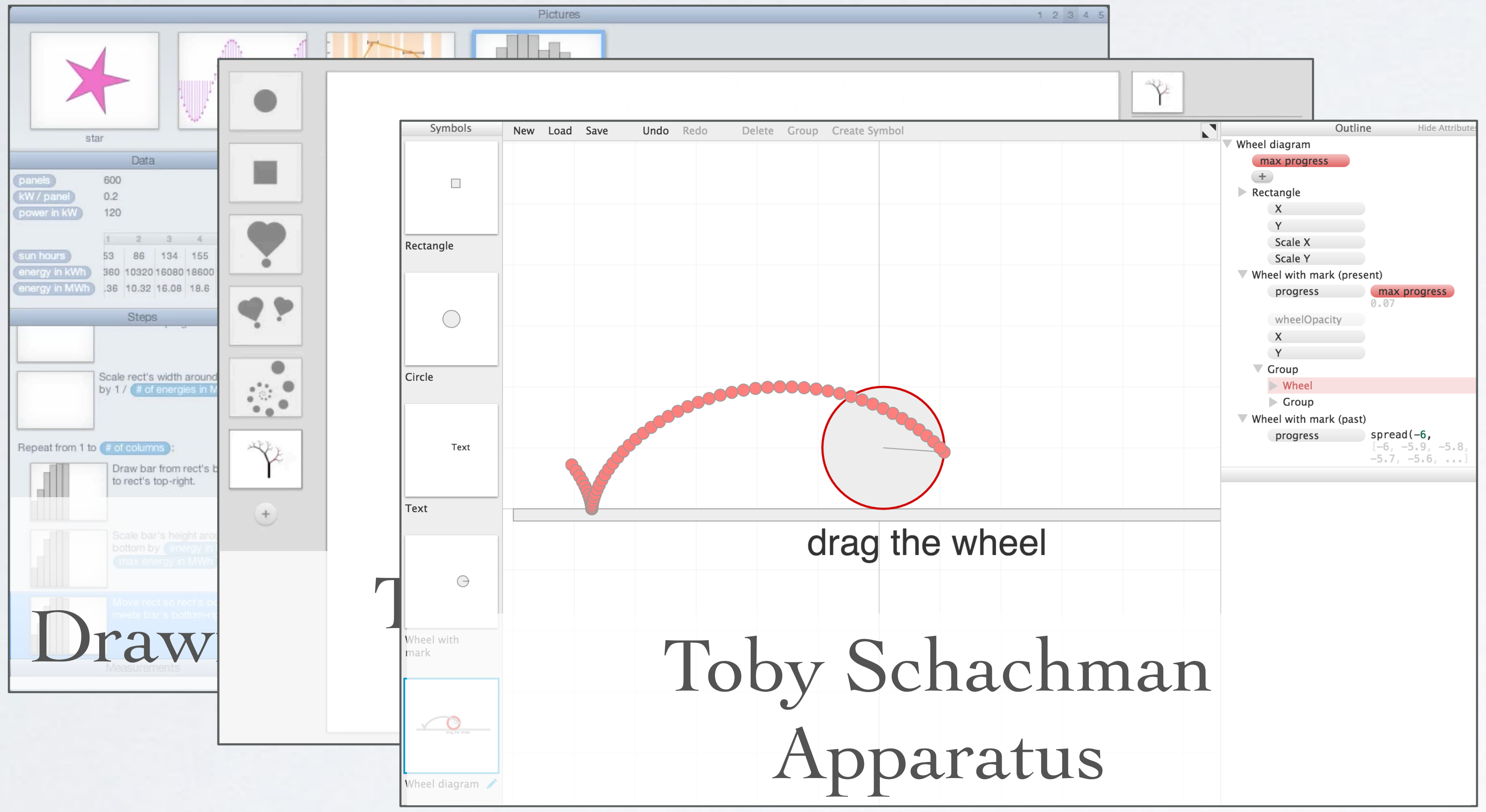


```
svg (concat [
])
```

Thank you!

Related Work: Non-standard Programs

Drawing with Constraints

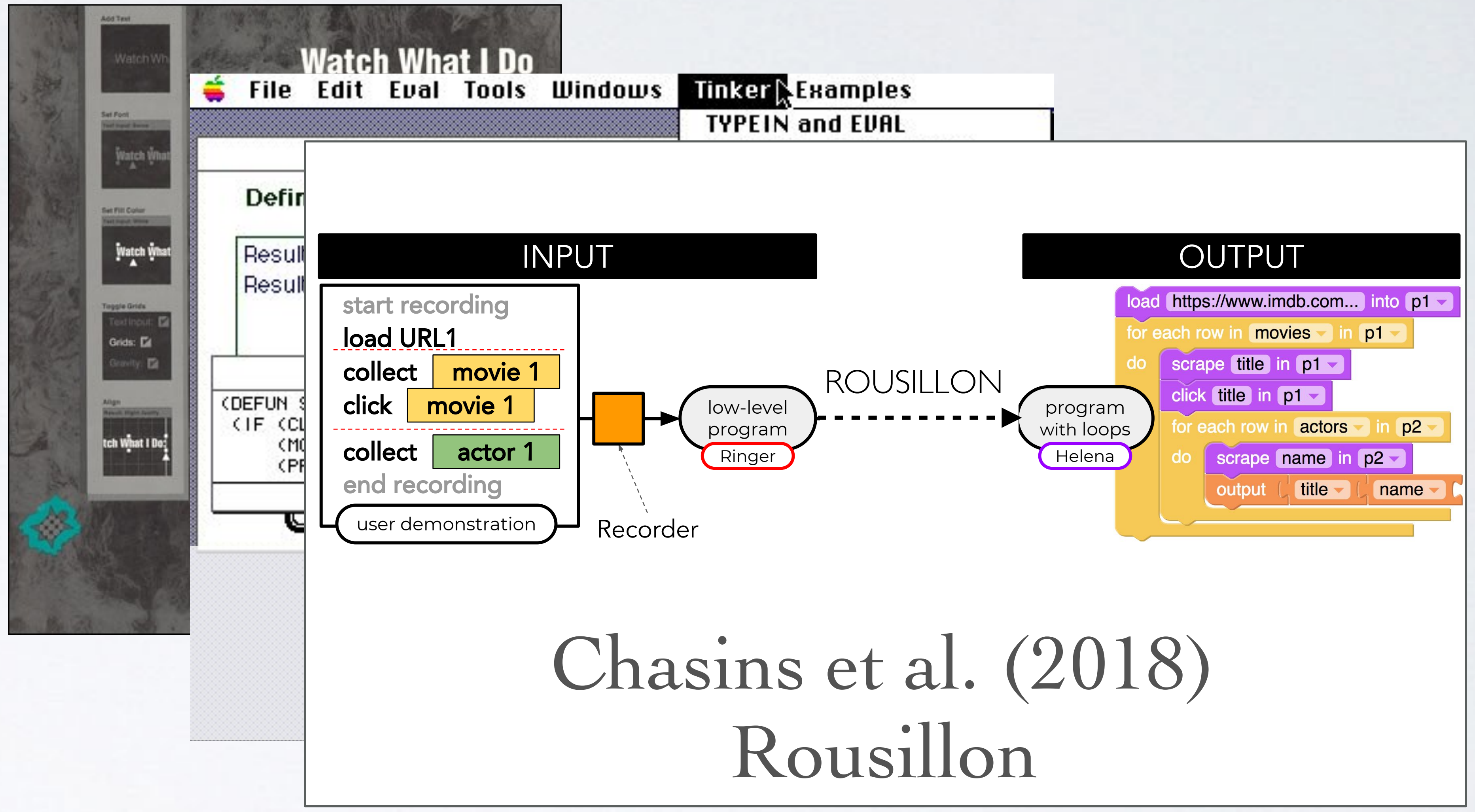


Draw

T

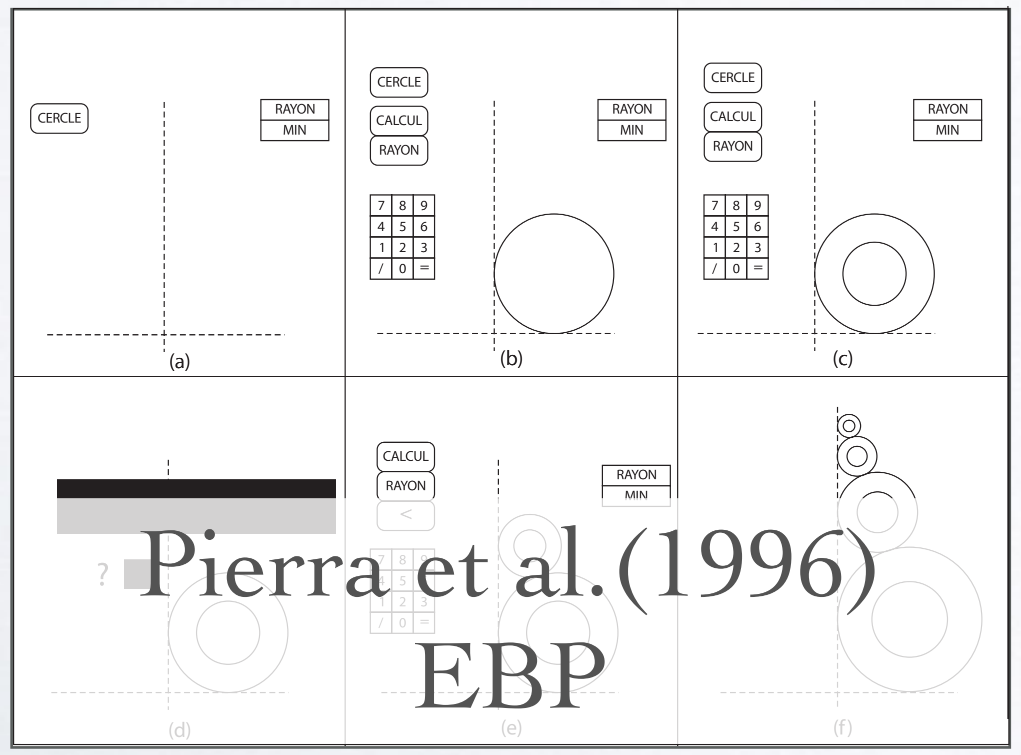
Toby Schachman
Apparatus

Programming by Demo (PBD)



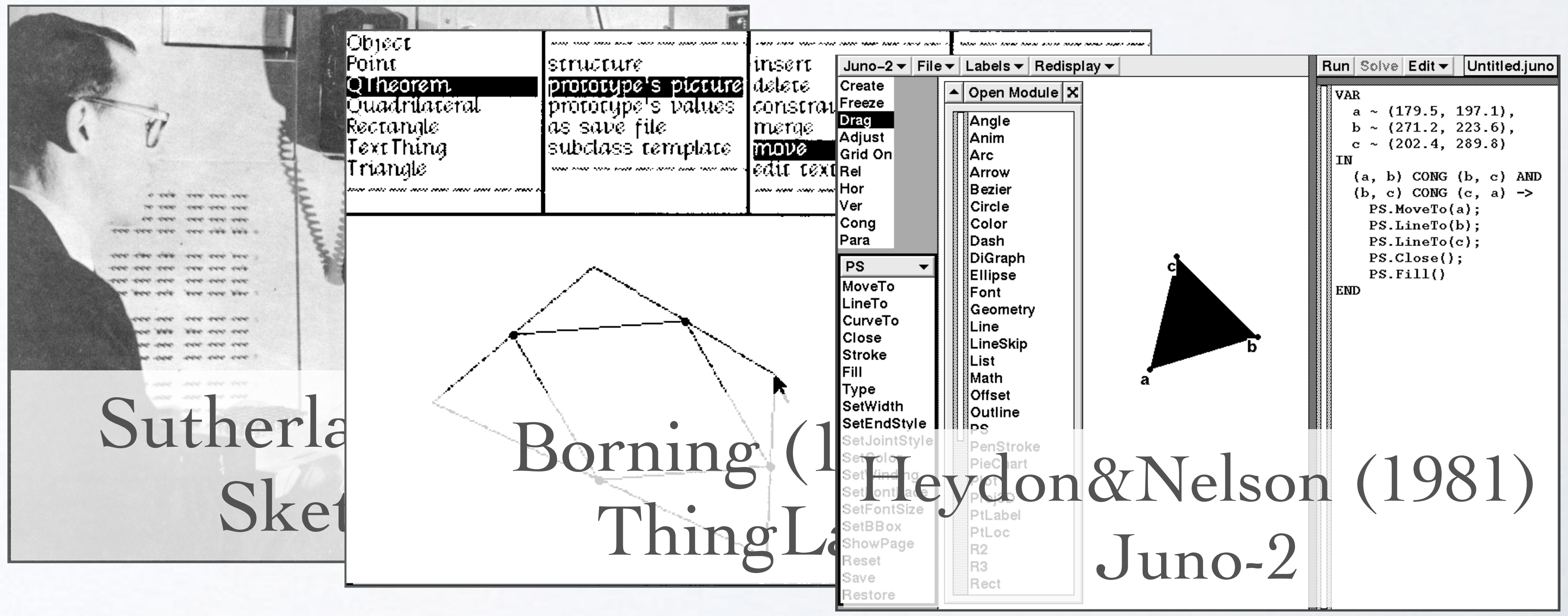
Chasins et al. (2018)
Rousillon

Parametric CAD



Pierra et al.(1996)
EBP

Constraint-Oriented Programming



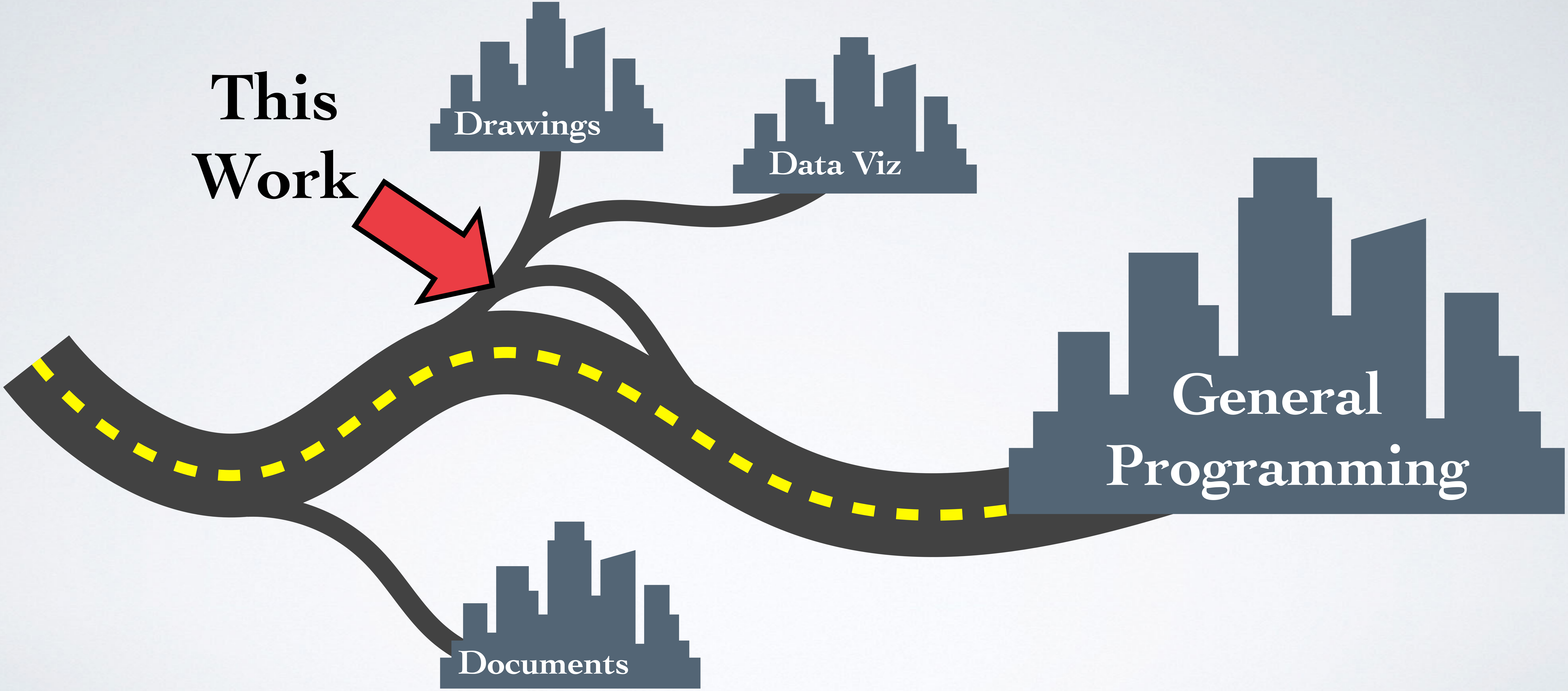
```
VAR
  a ~ (179.5, 197.1),
  b ~ (271.2, 223.6),
  c ~ (202.4, 289.8)
IN
  (a, b) CONG (b, c) AND
  (b, c) CONG (c, a) ->
    PS.MoveTo(a);
    PS.LineTo(b);
    PS.LineTo(c);
    PS.Close();
    PS.Fill()
END
```

Sutherla
Sket

Borning (1
ThingLa

Heydon&Nelson (1981)
Juno-2

Research Roadmap



Draw Shape

1. Inserts function call, assigns it to a variable.

2. Attempts to add `newVar` and `[newVar]` to the list literals in the program.

3. Succeeds when number of shapes in the output increases by the expected amount.

Make Equal

1. Use numeric traces (Chugh et al. PLDI '16) to set up an equation:
   $114_{\texttt{lineX1}} = 245_{\texttt{rectCX}} - 80_{\texttt{rectHalfW}}$

Numeric Traces(Chugh et al. PLDI '16)

```
let a = 3 in
let b = 5 in
   a + b
```

$\Downarrow$

8

Numeric Traces(Chugh et al. PLDI '16)

let a = 3_a in

let b = 5 in

a + b

⇓

8

Numeric Traces(Chugh et al. PLDI '16)

let a = 3_a in

let b = 5_b in

a + b

⇓

8

Numeric Traces(Chugh et al. PLDI '16)

let a = 3_{a} in

let b = 5_{b} in

a + b

⇓

8_{a+b}

MAKE EQUAL

1. Use numeric traces (Chugh et al. PLDI '16) to set up an equation:
   $\texttt{114}_{\texttt{lineX1}} = \texttt{245}_{\texttt{rectCX}} - \texttt{80}_{\texttt{rectHalfW}}$
2. Choose a constant to solve for & remove. Solve. (External solver: REDUCE).
   $\texttt{114}_{\texttt{lineX1}}\ \texttt{\textasciitilde\textasciitilde>}\ \texttt{245}_{\texttt{cxRect}} - \texttt{80}_{\texttt{halfWRect}}$
   $\texttt{80}_{\texttt{halfWRect}}\ \texttt{\textasciitilde\textasciitilde>}\ \texttt{245}_{\texttt{cxRect}} - \texttt{114}_{\texttt{lineX1}}$
   $\texttt{245}_{\texttt{cxRect}}\ \texttt{\textasciitilde\textasciitilde>}\ \texttt{114}_{\texttt{lineX1}} + \texttt{80}_{\texttt{halfWRect}}$
3. If a needed constant is not bound to a variable, insert a new `let` binding at a scope visible to its usages.
4. Ranking heuristic:
   1. Smallest AST (often all the same size).
   2. Shortest distance between constants removed (measured in lines).
   3. Prefer removing constants later in the program (less like to cause a dependency inversion).

ABSTRACT

1. Interpret the selection as a late ("proximal") set of program expressions. (Probably could be looser.)

2. Choose one of those expressions to be the return expression of the function.

3. Iteratively find let bindings that (a) have free variables and (b) are only used in the function body and add those bindings to the function body.

4. Any remaining free variables become arguments.

REPEAT OVER FUNCTION CALL

1. Set up an expression filter: Find `[x, y]` pair values in provenance (execution history) of selected shapes and thereby identify relevant *x* expressions, *y* expressions, and *point* expressions in the program.

2. Interpret the programmer's selections to a single expression that contains either (a) one of the above *point* expressions, or (b) both an *x* and *y* expression from above. Use ABSTRACT to make this single expression a function over a single point.

3. Map that new function over the point list.

Snap Drawing via *Value Holes*

1. Internally: Insert template code with *value holes* in place of the snaps. (A value hole is a temporary expression that contains a value.)

   ```
   [x, y] = [123, 456]
   rect1 = rect … [??123, ??456] …
   ```

2. Examine the provenance of the value in each to fill the hole by either:

   1. Using an existing variable (from the execution environment or from the static scope, possibly moving an existing binding into scope).

   2. Introducing (and using) a new variable for an existing expression.

   3. Deconstructing some variable in the environment with a pattern match to expose a needed value (and using the introduced variable).

      ```
      [_, y] = somePoint
      ```

Draw Custom Func via *Roles*

1. Functions that take two points, or a point and a distance, are drawable.
2. Types may be tagged with a set of *roles*, explaining the type's semantic meaning. (E.g. "This number is a *width*. This number is a *color*.") Called "brands" in APX. Similar to measure types, but not type-checked.
3. Roles are introduced by type aliases.

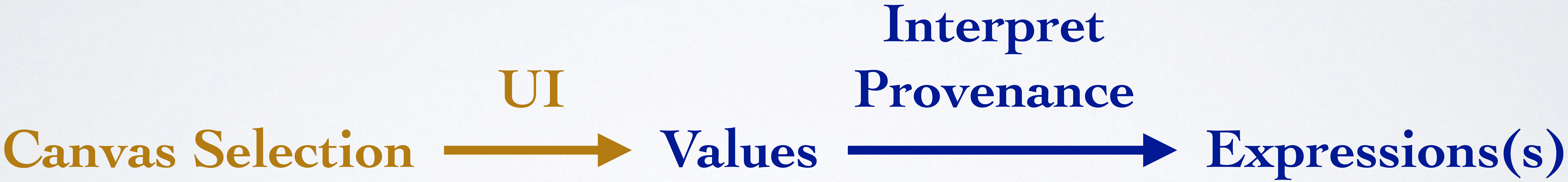
```
type alias Color = Num
rect :: … → Color → …
```

4. Roles propagate during the unification step of type inference.
5. Addition domain-specific rules for propagation, e.g.:

$$\mathtt{a}_{\mathtt{Num:\{X\}}} + \mathtt{b}_{\mathtt{Num:\{\}}} \Rightarrow \mathtt{a}_{\mathtt{Num:\{X\}}} + \mathtt{b}_{\mathtt{Num:\{HorizontalDistance\}}}$$

6. Roles also determine the defaults for arguments.

Provenance



Canvas Selection

UI

Values

Interpret Provenance

Expressions(s)

Four Kinds of Provenance

Numeric Traces (Chugh et al. PLDI ’16)

Offsets (numbers tagged with other coordinate)

“Based On” Provenance

“Parents” Provenance

“Based On” Provenance

What expressions are associated with
a value selected in the output?

For a particular value, what other values at
other execution steps were used to produce it?

$$\Gamma \vdash e \Downarrow v^{e,\{v_1,\ldots,v_n\}}$$

Could you hide the code?

Fundamental limitations?

Other Limitations?

Will the techniques generalize?

Future Work

Could you hide the code?

Maybe for simpler cases.

Can you represent the computation visually? (VPLs 😕)
Code only → simulate computer.
Output only → simulate code.

Consider the hover-to-preview interaction today.

(Later APX demos did hide the code)

Fundamental Limitations?

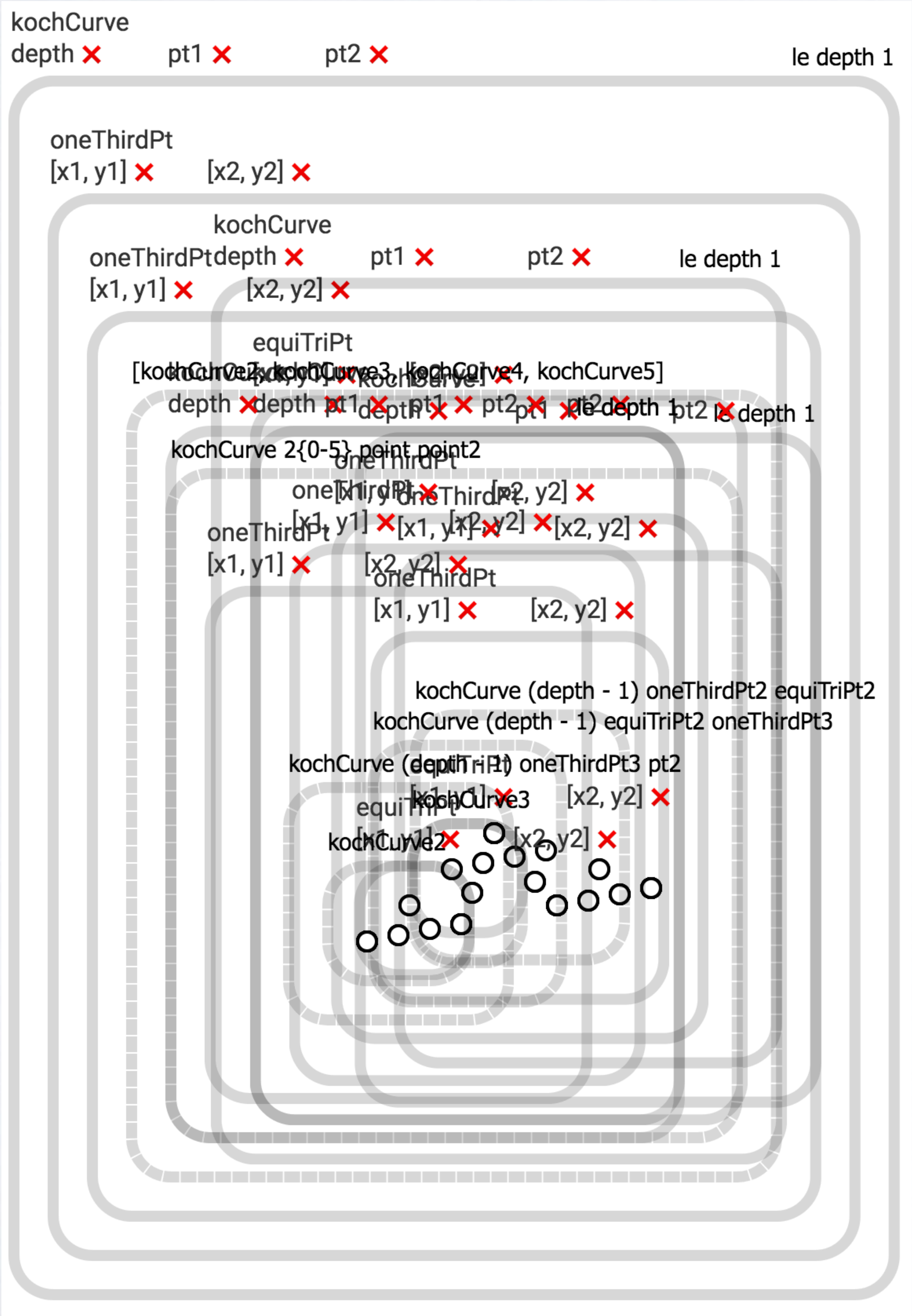
So far: “Select and Act” in small steps.
Good for mouse, because that’s all a mouse can do.
Generally avoided large inference steps: ambiguity.
(exceptions: RELATE, REPEAT BY INDEXED MERGE)

$\text{bandwidth}_{\text{keyboard}} > \text{bandwidth}_{\text{mouse}}$

…voice input?

Fundamental Limitations?

Impossible to display all intermediates.



```
kochCurve (depth - 1) oneThirdPt2 equiTriPt2
kochCurve (depth - 1) equiTriPt2 oneThirdPt3
kochCurve (depth - 1) oneThirdPt3 pt2
```

Solution so far: *contextual visibility.*

But this is fundamental:
#intermediates >>>>> screen space

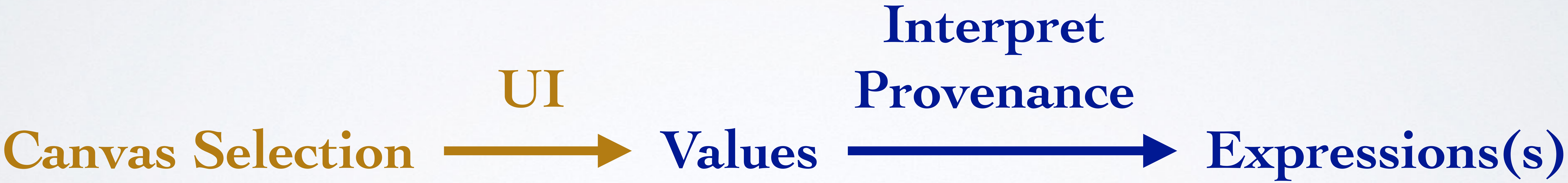
Other Limitations?

Not much work on breaking relationships.
(Edit history?)

More details need to be worked
out so tools compose reliably.
(Syntactic binding locations, e.g. Xs example.)

Will the techniques generalize?

"Select & Act"



UI

Canvas Selection

Values

Interpret

Provenance

Expressions(s)

Future Work

Transform DSL over value selections

Unified provenance

Visualize non-visual code