# Computational Making via Bidirectional Parametric Modeling

Chris Johnson[1] and Ian McCormack[2]

[1]James Madison University; johns8cr@jmu.edu
[2]University of Wisconsin, Eau Claire; mccormack2812@uwec.edu

## Abstract

To develop both aesthetic and algorithmic sense in young learners, we have built Twoville: a bidirectional vector graphics editor meant to be used in makerspaces and schools. Programmer-designers define shapes parametrically in Twoville using code and then adjust the parameters via direct manipulation of the output. Changes in either the code editor or the drawing canvas are immediately reflected in the other editor. When code is updated by a direct manipulation of the output, the syntactic structure of the programmer-designer's original expression is maintained to the extent possible. The resulting design is exported as an SVG file, loaded into a fabrication tool, and physically realized.

## Introduction

Twoville is a programming language and development environment for composing vector graphics files for tools like vinyl and laser cutters and pen plotters. Its users write code to generate, transform, and combine geometric shapes that determine the cutting or drawing path of these fabrication tools. The output of a program is a physical artifact that has a life beyond its digital roots. The tool is open source, runs entirely in web browsers, and is freely available on the web [4].

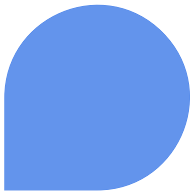
Many digital fabrications begin with a designer taking up a mouse. The designer opens a design tool and uses the mouse to directly manipulate a digital representation of an object. Other kinds of tools allow the designer to indirectly manipulate the object by writing code. We contend that both of these modes of making are vital to a balanced human experience and should not be separate. As we've developed the language, we have come to the unoriginal conclusion that not all design work can be effectively programmed. Code is an indirect interface, and designers often want more direct control driven by aesthetic feel. Therefore, we provide in Twoville bidirectional editing. The programmer-designer can edit the code, which updates the output, or the output, which updates the code. Objects may be shaped both algorithmically and through mouse-based direct manipulation of parameters.

## Language

Twoville is a text-based language with syntax inspired by Python and Ruby. It has features that one would expect of a programming language, including primitive data types, variables, operators, flow control, and procedural abstraction. However, it is a domain-specific language specifically for computational making. We have drawn upon our experience working with young learners in classrooms and makerspaces in public schools in the United States to develop it. In the following sections, we enumerate several of the design decisions that we have made based on these experiences and illustrate the language through examples.

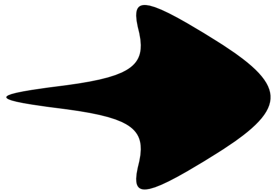
### *Be Domain-Specific*

Instead of using an existing general-purpose programming language, we have intentionally developed a small domain-specific language. Our language's syntax is oriented around the primary activity of describing



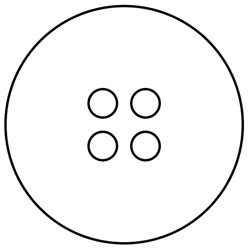
```
with path()
  color = :cornflower
  go().position = [0, 0]
  line().position = [1, 0]
  with arc()
    degrees = 270
    center = [1, 1]
```

**(a)** *A shape composed in piecewise fashion using using line segments and arcs.*



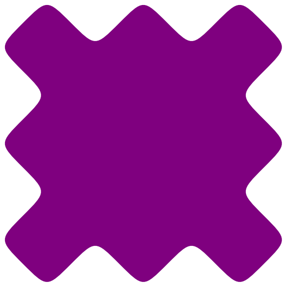
```
with ungon()
  rounding = 0.2
  vertex().position = [-1, 5]
  vertex().position = [7, 4]
  vertex().position = [6, 0]
  vertex().position = [14, 5]
  vertex().position = [6, 10]
  vertex().position = [7, 6]
  color = :black
```

**(b)** *A silhouette of a ray or skate described as a ungon—a polygon with its corners rounded.*



```
to ring(center, radius)
  with circle()
    center = center
    radius = radius
    stroke.color = :black
    stroke.size = 1
    opacity = 0
big = 50
small = 6
gap = small * 1.5
ring(:zero2, big)
ring([-1, -1] * gap, small)
ring([ 1,  1] * gap, small)
ring([-1,  1] * gap, small)
ring([ 1, -1] * gap, small)
```

**(c)** *A button constructed using abstraction and parameterization.*



```
with ungon()
  color = :purple
  rounding = 0
  with turtle()
    heading = 135
    position = [25, 5]
  repeat 4
    repeat 3
      move().distance = 20
      turn().degrees = 270
    around
      move().distance = 20
      turn().degrees = 90
```

**(d)** *A design programmed by a 12-year-old beta tester using* `ungon`*, turtle geometry, and loops.*

**Figure 1:** *Example Twoville Programs*

geometric shapes. If we had used an existing general purpose language, we would have been forced to package our code into heavily parameterized functions whose calls would be difficult to read. Additionally, our source code editor and our drawing canvas are closely linked. Changes made in either the editor or canvas automatically update the other. This coupling requires intimate control of the program structure.

## *Name Everything*

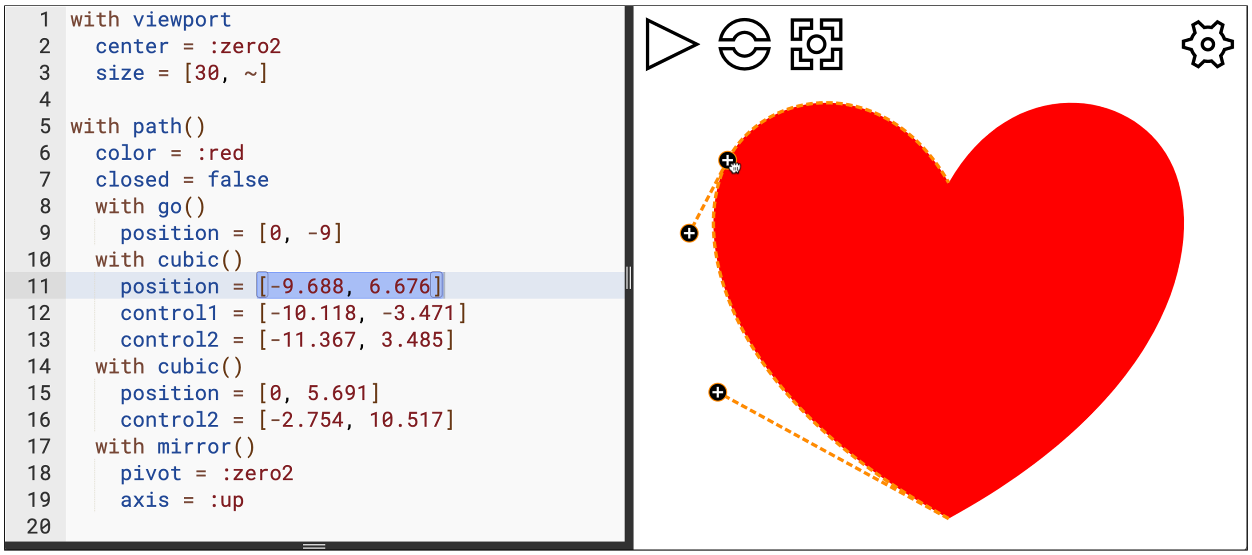
In many conventional programming languages, parameters are positional, which means they are ordered as prescribed by the function definition. The significance of positional parameters may be unclear when they are literal numbers or poorly-named variables, especially if the code was written by others or long enough ago to be forgotten. For this reason, we eschew positional parameters.

Named parameters are more descriptive than positional parameters. Some of Twoville's shapes have many properties and would need functions with many parameters to construct them. An abundance of named parameters leads to long lines of source code and horizontal overflow within the editor, which make code harder to read and navigate. For this reason, we mostly avoid parameters altogether. We favor assigning properties after a shape has been constructed. To allow concise, unqualified assignment of multiple properties, we provide a `with` scoping construct. Its usage is demonstrated in Figure 1.

## *Target SVG*

SVG is an open standard for vector graphics that has been managed by the W3C since its release in 2001. The format is an extension of XML. Many fabrication tools accept SVG files as input, and any modern web browser will display them. SVG files can be unwieldy to write given its heavy syntax and paucity of abstractions. Additionally, there are many implicit constraints on elements' properties that are not easily enforced without visual feedback. Designers rarely author SVG files by hand. Many use the direct manipulation vector drawing tools of Adobe Illustrator, Gravit Pro, or Inkscape.

Like these other tools, Twoville is an editor for expressing vector designs while hiding the complexity of SVG, but it features a hybrid editor that supports both code and direct manipulation. We target SVG because



```
with viewport
  center = :zero2
  size = [30, ~]

with path()
  color = :red
  closed = false
  with go()
    position = [0, -9]
  with cubic()
    position = [-9.688, 6.676]
    control1 = [-10.118, -3.471]
    control2 = [-11.367, 3.485]
  with cubic()
    position = [0, 5.691]
    control2 = [-2.754, 10.517]
  with mirror()
    pivot = :zero2
    axis = :up
```

**Figure 2:** *The Twoville programming environment. The cursor is currently placed in the first cubic Bézier node in the path, so only the handles for its starting position and two control points are displayed.*

of its wide support. The browser-based interpreter translates a program into an SVG hierarchy embedded in the page as it executes. The browser renders the element. When a design is ready to be fabricated, the SVG element is serialized to SVG's standard XML format and imported in the fabrication tool's control software.

Twoville's power is bound up with the SVG standard that it targets. As such, it is limited to generating rectangles, ellipses, lines, polylines, polygons, and paths composed of straight lines, quadratic and cubic Bézier curves, and arcs. Other shapes are built upon these primitives, including the ungon shown in Figure 1b. An ungon is a polygon that has been converted to a series of Bézier curves using Chaikin's algorithm [1], which effectively rounds off the polygon's corners.

Non-line shapes have both fill and stroke properties. Laser-cutters generally cut along strokes and engrave filled areas. For this reason the button in Figure 1c is composed of only strokes. To engrave a well in the button, the user must add a filled inner circle. Masking operations may be used to combine shapes in various logical ways.

## Bidirectional Editing

Schneiderman [3] defined direct manipulation as an interactive mode of editing in which a user's actions are rapidly executed, immediately observable, and easily reversed. Direct manipulation was initially positioned as a friendlier alternative to programmatic manipulation. In Twoville, we allow for both direct and indirect programmatic manipulation.

A shape's spatial properties like size and position are initially set in code, but they can be edited either in code or in the drawing canvas. To edit a property through the canvas, the programmer-designer manipulates a property's *handle* with the mouse, and the mouse movement is used to determine the property's new value. The new value is updated in both the SVG model and the source code in real time on each mouse event. The interface for the Twoville editor is shown in Figure 2.

To interpret the mouse movement in a manner consistent with the semantics of a property, we provide several different kinds of handles. For example, a rectangle's width is controlled by a horizontal motion handle, and its height by a vertical motion handle. A vertex position is controlled by a handle that moves in all directions, like the three shown in Figure 2.

Suppose a visual property currently has value $p$. When the user drags on a property's handle, we compute the new value $p'$ based on the mouse position and in accordance with the type of handle. We must then update the code so that the property is assigned $p'$. Instead of replacing the entire right-hand side expression of the assignment, we attempt a more nuanced editing based on the structure of the expression based on the following heuristics:

1. If the expression is of the form `property = variable`, then we recurse and update the expression used to assign `variable`.
2. If the expression is of the form `property = a ⊕ b`, where $\oplus$ is a binary operation with inverse $\ominus$, then we update only the right operand. Before the manipulation, we have $p = a \oplus b$. After the manipulation, we want $p' = a \oplus b'$. We replace the expression for just the second operator with the value $b' = p' \ominus a$.
3. If $\oplus$ is a non-invertible function like cosine or if the inverse operation would lead to an illegal operation like division-by-zero, we preserve the original expression in its entirety but add an offset $\Delta$. We want $p' = a \oplus b + \Delta$, so we add $\Delta = p' - (a \oplus b)$ without otherwise modifying the original expression.

The Sketch-and-Sketch project of Chugh et al. [2] heavily inspires our work on Twoville. Both projects are environments for indirectly coding and directly manipulating vector graphics designs. Sketch-and-Sketch uses a more complex algorithm for determining what subexpressions in the code to update on a direct manipulation of the output. Twoville relies on simpler heuristics that were intentionally chosen to cater to an audience of computational makers and young learners. In Sketch-n-Sketch, shapes can be made entirely through widgets on the canvas. We do not share this aim in Twoville. When using direct manipulation in GUI builders and drawing applications, designers quickly lose the ability to navigate and understand the code generated by their mouse actions. Since we view Twoville as a means of learning about computation, we do not want programmer-designers to disengage entirely from the code editor. Direct manipulation is used to tweak a design's parameters, not to generate arbitrary program structures.

## Conclusion

We have introduced Twoville, a new platform for computational making. The bidirectional editor allows programmer-designers to shape structures both algorithmically and aesthetically. The digital design is exported as an SVG file and loaded in a fabrication tool, which cuts or engraves it in physical material.

Our primary goal in building Twoville is to use it as a vehicle for empowering young learners in makerspaces and schools to apply the mathematics and computation that they are learning creative physical construction. We are currently using it with middle and high schoolers in our community. The next steps for us in this process are to form partnerships with teachers and media specialists and to develop lessons and exercises that others can use.

## References

[1] G. Chaikin. "An algorithm for high speed curve generation." *Computer Graphics and Image Processing*, vol. 3, no. 4, 1974, pp. 346–49.

[2] B. Hempel, J. Lubin, and R. Chugh. "Sketch-n-Sketch: Output-Directed Programming for SVG." *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, 2019, pp. 281–292.

[3] B. Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages." *Sparks of Innovation in Human-Computer Interaction*, vol. 17, 1993.

[4] Twoville. https://twoville.org.