

# Direct Manipulation and SVG: Creating and Adjusting Graphics Programmatically and Visually

Ian C. McCormack  
University of Wisconsin-Eau Claire  
Eau Claire, WI  
USA  
---  
mccormic2812@uwec.edu

Chris Johnson  
University of Wisconsin-Eau Claire  
Eau Claire, WI  
USA  
twodee.org  
johnch@uwec.edu

## ABSTRACT

Vector graphics design files are widely used in fabrication due to their capacity for lossless scaling and because they align naturally with the physical machinery of laser cutters, CNC machines, and similar tools. Designers use visual editors such as Adobe Illustrator® and Inkscape to create vector graphics design files, but this software can be unwieldy for designs with repetitive or algorithmic structures. To address these obstacles, we are developing an editor for producing vector graphics images both programmatically and aesthetically, maintaining the mouse-based, direct manipulation features of popular image editors. We present our experiences in implementing a hybrid editor and explore how it might be used in parametric design.

## Tools, Skills and Materials

• Tools→Vinyl and Laser Cutters • Skills→Graphic Design • Skills→Programming

## Keywords

Direct Manipulation; Vector Graphics; Programming Languages; Parametric Design

## 1. DEMO DESCRIPTION

### 1.1 Description of the Product

We will demonstrate a programming language for composing 2D vector graphics design files for tools like vinyl and laser cutters. The language was born out of our desire to unite algorithmic construction with aesthetic design. 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.

Rather than use an existing general-purpose programming language, we have intentionally developed a small domain-specific language (DSL) in order to offer more targeted support for programmatic design. Our language's syntax is designed to make descriptions of geometric shapes readable and to support animations. If we had used an existing general purpose language, we would likely be 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 linked. Changes made in either the editor or canvas will automatically update the other. This linkage requires intimate control of the program structure.

#### 1.1.1 Geometric Primitives

Like many vector graphics design tools, our language allows users to compose algorithmic designs out of simpler geometric primitives. We briefly describe the available primitives. Examples of their use are shown in Table 1.

**Rectangle** The `rectangle` command plots a rectangle in the canvas. Core properties include the rectangle's position and size. A rectangle may be positioned using either its center or bottom-left corner coordinates. Its corners may be rounded.

**Circle** The `circle` command plots a circle in the canvas. Core properties include the circle's center and radius.





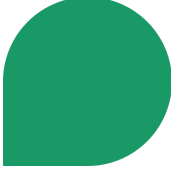
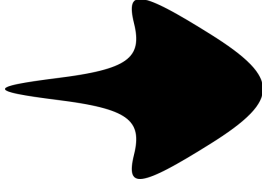
**Line and Polyline** The commands `line` and `polyline` plot line segments in the canvas. Core properties include the positions of the segment vertices. A line connects two vertices, whereas a polyline connects a sequence of many vertices. Both may be decorated with start, end, and mid-markers. Lines and polylines are not directly useful for fabrication tools that expect outlines of solid shapes, but users may convert the lines into a more fabrication-friendly format using other software.

**Polygon and Ungon** The commands `polygon` and `ungon` plot shapes enclosed by a series of vertices. Core properties include the positions of these vertices. An ungon is a polygon with its sharp corners rounded off.

**Path** The command `path` plots a shape defined by the versatile but formidable SVG path element. A path is a series of steps between positions. Each step of the path moves to a new position in one of the following ways: a non-plotting jump, a line, a quadratic Bézier curve, a cubic Bézier curve, or a circular arc. Paths may be open or closed.

Any of the shape commands that expect a series of vertices—namely `polyline`, `polygon`, `ungon`, and `path`—can either be described with absolute Cartesian coordinates or with relative turtle geometry commands as seen in Logo and Logo-inspired languages like Scratch.

**Table 1: Example programs for several of the shape primitives.**

 <pre>with rectangle()   center = [0, 0]   size = [4, 3]   rounding = 0.5   color = :crimson</pre>	 <pre>with circle()   center = [0, 0]   radius = 3   color = :cornflower</pre>	 <pre>with polyline()   for i to 6 by 2     vertex().position = [1, i]     vertex().position = [-1, i + 1]   size = 0.2   color = :orange</pre>
 <pre>with polygon()   with turtle()     position = [0, 0]     heading = 36   repeat 5     move().distance = 5     turn().degrees = 144     move().distance = 5     turn().degrees = -72   color = [0.9, 0.7, 0.1]</pre>	 <pre>with path()   color = [0.1, 0.6, 0.4]   with jump()     position = [0, 0]   with line()     position = [1, 0]   with arc()     degrees = 270     center = [1, 1]</pre>	 <pre>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</pre>

### 1.1.2 Shape Operations

Each geometric primitive has a unique set of core properties, but each also supports a set of universal operations that modify its appearance or combines it with other primitives. We briefly describe each of these operations.

**Transformations** Any of the shape commands may be modified by a sequence of transformations. Supported transformation commands include `rotate`, `scale`, `shear`, and `translate`. The first three of these support arbitrary pivot points. Transformations are applied in order of appearance. Paths additionally support a mirroring transformation that will automatically generate the symmetric complement to the path's steps.

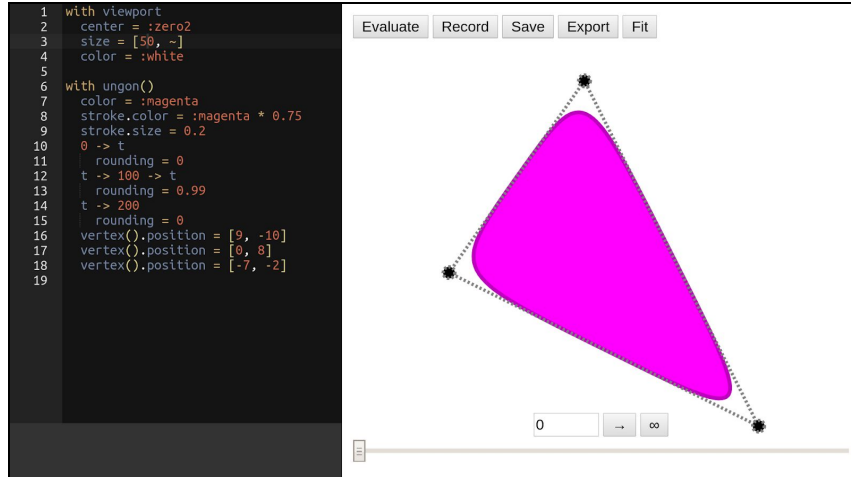
**Groups** Shape primitives may be organized into hierarchical groups. The primary advantage of grouping in our language is that transformations can be applied just once to the parent group instead of individually to each child. The groups are retained in the exported SVG document, which may ease further manipulation in a vector graphics editor.

**Masking** Not all designs can be expressed naturally using just primitives. To compose more complex designs, users may add shapes to masking or clipping layers, which may then be combined with plotted shapes to produce unions, intersections, and differences.

### 1.1.3 Editor

Programs in our languages are written in a split-pane development environment. The left-hand pane contains a source code editor and a small debugging console that displays print output and error messages. The right-hand pane, or drawing canvas, displays the visual output of a program within a resizable SVG element. Users can pan and zoom using the mouse.

**Figure 1: The Development Environment**



An array of buttons at the top of the viewing pane controls the basic functions of the interpreter and the export process. Clicking “Evaluate” runs the program. Clicking “Export” downloads the current canvas as an SVG file. The “Fit” button resets the view to the viewport defined in the program.

Our language serves two purposes. The primary purpose described here is the algorithmic generation of vector graphics design files, but we also support the algorithmic generation of 2D animations. Animations are not the focus of this demonstration, but the user interface (and syntax) is influenced by this secondary purpose. The “Record” and “Save” buttons can be used to render and export animated GIFs, and the timeline at the bottom of the canvas is for scrubbing through frames.

### 1.1.4 Programming Language Elements

In addition to the types of shapes displayed in Figure 1, our language supports common programming language elements such as local and global variables, loops, and conditional statements. We have also incorporated object-oriented design patterns for defining shapes and groups. This is currently implemented through function objects, which combine features of classes and functions along with a flexible syntax for creation and modification.

Function objects are created using the `def` keyword. A function object can be classified as a type of shape with the `as` keyword and can inherit properties from other function objects through the `from` keyword. The properties of a function object can be defined and assigned values by placing a `.` before an identifier. If a `return` statement is present in the body of a function object, it will return a given value.

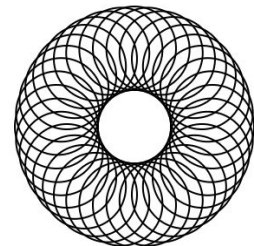
**Figure 2: Creating a Torus with Function Objects**

```
def circleStyles
  .stroke = :black
  .strokeWidth = 1
  .fill = :transparent

def circlePoint(x, y, r) as circle from circleStyles
  .pos = [x, y]
  .r = r

def circleRing(midX, midY, ringRadius, circleRadius, numCircles)
  let degreeStep = 2*:pi/numCircles

  repeat stepCount to numCircles
    let xCoord = ringRadius * cos(stepCount*degreeStep) + midX
    let yCoord = ringRadius * sin(stepCount*degreeStep) + midY
    draw circlePoint(xCoord, yCoord, circleRadius)
```



### 1.1.5 Direct Manipulation

Designing strictly through code quickly leads to cognitive overload. When the only mechanism for setting properties is computation, the designer may feel disinclined to experiment. For this reason, we allow most spatial properties to be manipulated both indirectly through code and directly on the canvas.

Interactive handles appear when a shape is selected, as shown in Figure 1. Each circular handle can be dragged to directly manipulate various properties of a shape, such as its position, rotation, width, height, and control points. When the interpreter runs the program, metadata is recorded that links each handle to the region in the source code where its property is defined. As a handle is dragged, both the linked source code, the internal representation of the program, and the SVG element are automatically updated to reflect the new value of the property. The program is not reinterpreted, which means the response time is effectively instantaneous. Generative designs, such as Figure 2, can also be manipulated. Manipulations to any circle in Figure 2 will result in changes to all circles and will target values passed to the *circleRing* function.

A shape with many directly manipulated properties will necessarily have many handles. To avoid overwhelming the designer, we have chosen not to display all the handles at once. Instead, when the text cursor is placed in the source code editor, only the handles associated with the enclosing property are shown.

Chugh et al. [1] describe a robust algorithm for direct manipulation in a hybrid SVG editor based on *trace-based program synthesis*. They reconcile the designer's changes to the output against the source code that originally generated it by attempting to satisfy a system of constraints relating the two. Our system is simpler in that there is a one-to-one correspondence between the handles and properties. There is no ambiguity about the user's intent.

### 1.1.6 Export & Fabrication

Once a design is complete, the designer exports it to a standalone SVG file with the click of a button. The SVG file conforms to the SVG 1.1 standard, which is widely supported. The SVG file may be further modified in a vector graphics editor as needed. The file is then imported into the fabrication tool's management software and realized as a physical object. We have tested our workflow with two fabrication tools: an AxiDraw v3 pen plotter from Evil Mad Scientist Laboratories and a Cameo 3 from Silhouette America, Inc. Artifacts from both will be shared during our demo.

Currently, our language's editor does not directly interface with fabrication tools as their protocols are not generally open or standardized. Instead we choose to indirectly target SVG, an interchange format supported by many fabrication tools.

## 1.2 Target Audience

Our language is designed to appeal to a broad category of users within the areas of graphic design, computer science, and education. The language is designed to have a low bar for entry with a high capacity for expression. This allows our editor to be deployed in introductory computer science courses as a way to represent fundamental programming concepts while remaining useful as a supplementary tool for graphic design and making.

## 2. CONCLUSION

### 2.1 Results and Benefits

We are just beginning to use our programming language in courses at our university and maker events in our community, but we ultimately want to build a tool that is useful to the broader community of educators who wish to inject computation into making. Our coding environment is freely available on the web, and we invite input from others during the formative stages of this project. The source code is also freely available and distributed under an open license.

### 2.2 Broader Value

Computation is often taught in a disembodied manner with strictly virtual output. With the proliferation of fabrication tools in learning spaces, educators have the equipment that can fuse design and computation. But few tools are available to support this integration. Our language is intended to fill this void, at least for the production of vector graphics design files. Both the language's output and the coding process itself is rooted in a visceral experience.

## 3. REQUIREMENTS

We need a small table on which to set a laptop and access to a standard 120v power source.

## 4. BIOS

Dr. Chris Johnson is an associate professor of computer science at the University of Wisconsin, Eau Claire. Ian McCormack is a Junior and a double major in computer science and English rhetoric at the University of Wisconsin, Eau Claire.

## 5. REFERENCES

- [1] Chugh, Ravi et al. "Programmatic and Direct Manipulation, Together at Last." Proceedings of the *37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016* (2016). <https://arxiv.org/abs/1507.02988>