Shaping with Code and Mouse in Twoville

Chris Johnson[1] and Will Morris[2]

Department of Computer Science, James Madison University, Harrisonburg, Virginia, USA
[1]johns8cr@jmu.edu, [2]morri2wj@dukes.jmu.edu

Abstract

Twoville is a free and web-based bidirectional programming environment for making designs that can be fabricated. Its users describe a design initially in code, but they modify the design and code simultaneously by clicking and dragging directly on the output. The resulting vector graphic may be sent to a vinyl cutter, laser cutter, pen plotter, embroidery machine, or other fabrication tool to produce a physical artifact. We developed Twoville to teach mathematics and computer science in our community's schools—and to make stickers, T-shirts, games, and greeting cards for more personal reasons. In this workshop, we will guide participants through a series of Twoville-based computational making exercises that they may use in their own classrooms and makerspaces.

Introduction

Two methods of interaction are commonly found in design tools: direct manipulation and indirect manipulation. In a direct manipulation interface, designers click and drag on a canvas to manually place and modify shapes. In an indirect manipulation interface, designers write code to algorithmically describe the shapes. Each method has advantages. Direct manipulation is visually intuitive, whereas indirect manipulation distills a shape down to its parametric essence. A design tool that supports both methods of interaction is a *bidirectional editor*. Twoville [6] is a free and web-based bidirectional editor written by us to support computational making in our local schools. Figure 1 shows a composite shape ready to be directly manipulated in Twoville.

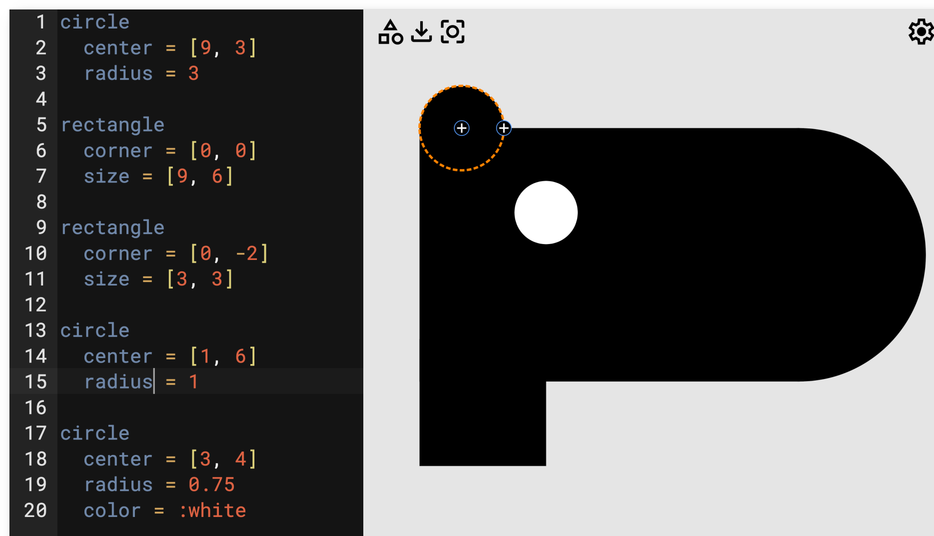


Figure 1: *A camel-horse-capybara composed of rectangles and circles in Twoville. The code editor reveals the parameters of each shape. The canvas displays the shape and presents dramay be handles for modifying the parameters.*

We think Twoville is useful, and we want to share it with others who make and teach. In 2021, we shared an early prototype with the Bridges community [3], and it was warmly received. Now we are offering a workshop in which participants will actively learn Twoville through a series of activities pulled from our own tested curriculum. In this paper we describe the Twoville platform and the planned workshop activities.

Twoville

We built Twoville so that we and others can make computational things. Code is at the heart of computation, but designing a physical object with only code is clumsy. Humans have a very powerful visual system, and we want to put it to good use in Twoville. Likewise, a design tool that requires a complex sequence of clicks and drags to perform every operation is tedious. Humans have a very powerful linguistic system, and we also want to put it to good use in Twoville.

The name Twoville is a nod to Seymour Papert's notion of Mathland [5]. Just as learning French is most effective in France, learning mathematics is most effective in Mathland. Twoville is a space for learning the mathematics and computer science of two-dimensional geometry. Its users speak in numbers, logic, and relationships. They get feedback on every utterance and adjust their thinking accordingly. They produce and consume shapes like the heptagon shown in Figure 2.

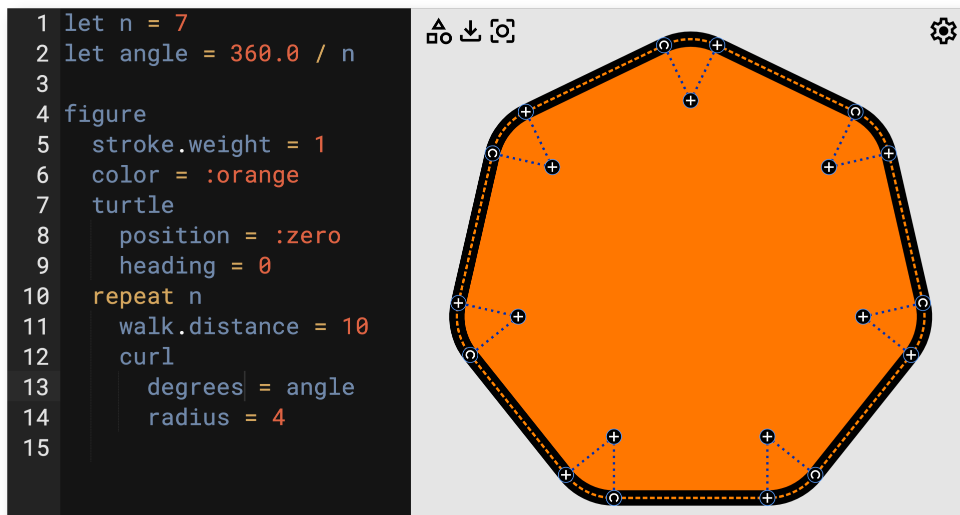


Figure 2: *A heptagon fashioned with turtle geometry and a repeat loop. Instead of the traditional turn command, which immediately redirects the turtle, we use the curl command, which traces out a circular arc. Draggable handles control the parameters for the turtle, walk, and curl nodes.*

Twoville supports a wide range of mathematical expression. Newer users construct designs using circles, rectangles, and polygons. Intermediate users use turtle geometry, data abstraction, and iteration. Advanced users use procedural abstraction, conditional logic, and trigonometric operations. We have successfully employed Twoville in events for audiences with diverse prior experiences in mathematics and English. In particular, we have used it in a weekly after-school program for middle schoolers and in summer camps for high schoolers participating in a refugee resettlement program.

Many educational technologies lock the user's creation to the technology used to create it. Programming an interactive story locks the story to the computer. Programming a line-following robot locks the algorithm to the robot. With Twoville, we aim to keep the user's creation unplugged. The immediate output of a Twoville program is a scalable vector graphics (SVG) file. We feed the SVG file into a fabrication tool like a

vinyl cutter, laser cutter, pen plotter, or embroidery machine to turn a design into a physical object that has a life independent of the computer used to make it. In this way, Twoville users can carry around or wear what they make and share it with family and friends.

Shapes

Currently Twoville supports these six shape commands:

- `circle`, which has draggable parameters for its center and radius.
- `rectangle`, which has draggable parameters for its corner or center, size, and rounding.
- `polygon`, which is a sequence of vertices. The vertices may be exact positions or they may be rounded off as chamfers or fillets. The positions and rounding are draggable parameters.
- `mosaic`, which is an indexed collection of draggable vertices grouped into adjacent tiles or tesserae. The gap between tiles may be adjusted in code.
- `figure`, which is a sequence of nodes that explicitly trace out a shape. Supported nodes include straight lines, arcs, quadratic and cubic Bézier curves, turtle geometry, mirrors across an axis, and tabs for gluing together geometric nets. The parameters of most nodes are draggable.
- `group`, which is used to organize shapes into transformation hierarchies.

These shapes may be composed with union, intersection, and difference operators. They may be transformed using scale, rotate, and translate modifiers, whose parameters are also draggable. Each shape may be filled or stroked. A stroked shape may be solidified into a solid shape so that a cutting or plotting tool traces around the stroke's outer perimeter rather than along its skeleton.

The supported shapes are built on top of the primitives defined in the Scalable Vector Graphics standard. Twoville targets this nearly universal protocol rather than communicating directly with fabrication machines. Most browsers and design tools support SVG. Some tools, like Silhouette Studio, currently require a license to import SVGs.

Direct and Indirect Manipulation

Some designs are driven by aesthetic feel and some by algorithmic contrivance. But this is a false dichotomy. Twoville supports both modes of thinking with its bidirectional editor. The code interface facilitates algorithmic expression. The drawing canvas facilitates aesthetic tweaking of a shape's parameters. Edits made in the code interface are applied to the canvas as soon as the user runs the program, and edits made in the canvas are applied immediately to the code.

Shneiderman [7] in the 1980s identified four characteristics of technological systems that humans of all skill levels enjoyed using:

- The content being manipulated has a persistent visual representation.
- Interaction is triggered by a physical action rather than a linguistic form.
- Feedback must be delivered quickly at each step of an interaction, and each step must be undoable.
- The system welcomes novice users with clear affordances but scaffolds them to advanced usage.

These characteristics lead to a style of interaction that Shneiderman called *direct manipulation*—in contrast to the indirect manipulation of a code interface. There's no reason that a single system can't respond to both direct and indirect manipulation. In fact, vector graphics editors commonly provide a direct visual interface and indirect form inputs for precisely setting a shape's parameters.

A few editors, including Cuttle [1], Sketch-n-Sketch [2], and Twoville, offer fully-featured code interfaces for indirect manipulation. There are advantages to supporting arbitrary code. Variables emerge as a natural

vehicle for extracting parameters and sharing them between shapes. The code behind a design keeps the design's logic transparent and learnable. Given the liquidity of text, code can be shared easily in emails and on the web. Loops repeat shapes across space. Common operations may be factored out to reusable functions.

Code interfaces also present challenges for bidirectional interaction. When a handle is dragged on the drawing canvas, the system must update the source code to reflect the new value of the parameter. That means the system must maintain a mapping from the handle to the source code. In our initial prototype of Twoville, we dispassionately overwrote the parameter value in the source code with a literal value derived from the mouse position. Any complex formulae that the user had entered in the code would be wiped. This felt like a violation, so we switched to preserving expression structures as much as possible. Now when a handle is dragged, we consider its old value p and compute its new value p' from the mouse position. Then we inspect the expression structure and update it according to its form. Our current algorithm recognizes and responds to these four forms:

- $p = number$. There is no precious structure to preserve, so the literal *number* is replaced with p'.
- $p = variable$. The property gets its value from *variable*, on which other shapes may depend. So that all dependent shapes will be updated, we recurse to update the assignment statement that gave *variable* its value.
- $p = a \oplus b$, where $\oplus$ is some invertible binary operation that we wish to preserve. We favor replacing numeric operands, which feel more disposable. If b is a literal number, we solve the new assignment $p' = a \oplus b'$ for b'. If a is a literal number, we solve $p' = a' \oplus b$ for a'. If an operand is surrounded by parentheses, we consider it locked and do not replace it. If an operand is a variable, we recurse on its assignment.
- $p = f(\ldots)$. Operation f is non-invertible or it didn't have any manipulable operands. We preserve the operation in its entirety and tack on an offset Δ to meet p'. We find Δ by solving $p' = f(\ldots) + \Delta$.

What value is computed for p' depends on the property. If the property is an angle, p' is derived from the mouse's angular position around a pivot point. If the property is a distance, p' is derived by projecting the mouse's position along an axis. If the property is a position, p' is derived from the mouse's Cartesian coordinates.

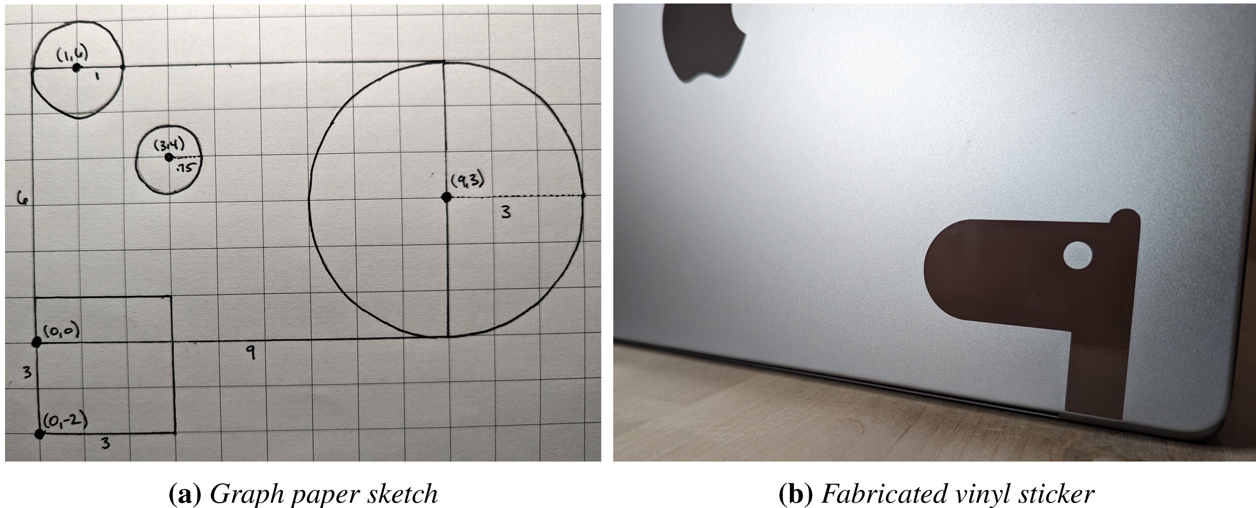
There are some challenges with bidirectional editing that we have yet to overcome. When a handle is just one of many generated by a loop, dragging on it sometimes affects all parameters in a uniform manner. For example, suppose we have generated a line of circles. Dragging one of them to a new position may cause all of them to collapse together. Additionally, dragging on a handle for a multi-valued property can lead to unpredictable behavior if the multiple values depend on a shared variable, as in the assignment `center = [x, x]`.

Workshop Description

Our workshop consists of two 45-minute design activities that we have successfully used with students in our local schools. We have a third activity planned in case there's extra time. All three activities get participants drawing, coding, and producing a physical artifact using a vinyl cutter that we provide. Twoville is freely available on the web, so we will either need a computer lab or participants will need to use their personal laptops. We describe the three activities in detail.

Frankenshape

Our warmup activity is one that we have used frequently in short outreach events and with younger students. The students learn to assemble composite shapes—frankenshapes—out of simple rectangles and circles, as



(a) *Graph paper sketch* **(b)** *Fabricated vinyl sticker*

Figure 3: *The first and last steps of designing a frankenshape. All workshop activities begin with an unplugged activity like drawing or manipulating physical objects. Frankenshapes start as drawings on graph paper. Only after the positions and sizes have been identified do students head to Twoville. The output is a vinyl sticker.*

shown in Figure 3b. This activity is meant to build on and reinforce the students' prior knowledge of the Cartesian coordinate system and geometric properties. We follow these steps:

1. The instructor projects a Cartesian grid onto a drawing surface and discusses how there are "x-streets" and "y-streets". A student chooses an origin, and from there the instructor numbers the streets so that addresses can be communicated precisely.
2. The instructor draws an interesting composite shape out circles and rectangles atop the grid. Our routine shapes include a cloud, a popsicle, and an animal ahead of indeterminate species.
3. Together the students and instructor identify the positions and sizes of each circle and rectangle, reviewing the terminology that the students have usually learned previously.
4. The instructor demonstrates how the design is communicated to Twoville.
5. Each student draws their own design on a piece of graph paper and identifies the positions and sizes.
6. The student gets their drawing reviewed by the instructor and fixes any issues.
7. Once the drawing is in good order, the instructor grants the student access to a computer.
8. The student codes their design in Twoville and sends it to the instructor.
9. The instructor sends the SVG to a vinyl cutter and gives the fabricated sticker to the student.

One rule of all our activities is to that we don't have students do their initial thinking at the computer. Instead they start by drawing or arranging physical manipulatives. Computers give almost instantaneous feedback, which diverts the students' attention into a game of satisfying the machine. With drawings and manipulatives, the students use their own senses to give themselves feedback at a measured pace. By the time they get to the computer, the deeply cognitive work has already been done. Programming is then just a translation of that work into a particular notation. In our experience, students don't find coding so difficult when it is separated from planning and measuring.

At the conclusion of this activity, we present some questions and challenges to the students. What do you think a programmer does? What's a design that we can't make with just circles and rectangles? What

sort of command would Twoville need to support this design? In the next few days, find a frankenshape in your life and reverse engineer it using Twoville.

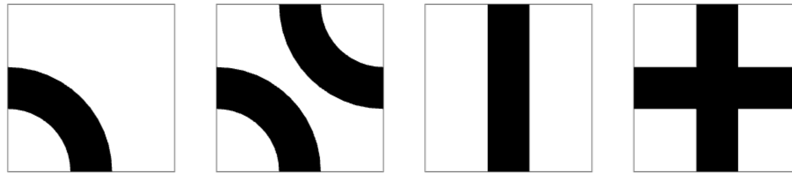


Figure 4: *The four possible tiles for making greeting cards. The student assembles these tiles into a knot, translates the path into a Twoville program using turtle geometry, and then cuts or embroiders the design.*

Knot Mosaic

Our second activity builds on the popular exercise of assembling knot mosaics from a fixed palette of tiles [4]. We restrict ourselves to the four tiles shown in Figure 4, which offer all possible configurations of runs and quarter turns. We ignore crossing order in order to produce a solid artifact. The students learn about distances and angles as they translate the knot that they have assembled into turtle geometry instructions using Twoville's `figure` command. We follow these steps:

1. The instructor gives each student a set of tiles.
2. The instructor gives the students minimal instructions: assemble a single circuitous path, such that a pen can trace it out without being lifted.
3. Students assemble their paths.
4. The instructor projects onto a drawing surface a simple knot with walks, turns, and crossings.
5. The students and instructor cooperatively measure the distances and angles of the path.
6. The instructor translates the example knot into the turtle geometry nodes available in Twoville's `figure` command. Figure 5 shows a radially symmetric knot generated with a repeat loop.
7. The students measure their own knots and show the instructor.
8. Once the knot is in good order, the instructor grants the student access to a computer.
9. The student codes their design in Twoville and sends it to the instructor.
10. The instructor applies the design to the front of a greeting card—by cutting or embroidering—and gives the card to the student.

The four tiles constrain the style of knots that can be formed, which we've found to be appropriate for young learners who sometimes get overwhelmed when measuring intricate knots. In a square mosaic, any walk distance is exactly the number of tiles involved in the walk and any angle is the number of tiles involved multiplied by 90 degrees. Furthermore, we do not expect the students to form or identify looping sequences in their knots as we see in Figure 5.

At the conclusion of this activity, we present some questions and challenges to the students. What's the smallest set of tiles we'd need to make a knot? Imagine outlawing one of the tile of the smallest set but allowing any number of the others. What knots could we make? Sum up all the angles used in your path. What can we say about that sum for any circuit? Suppose we had given you hexagonal tiles instead of square tiles. What might some of the tiles have looked like? What angles would be possible in the `curl` command?

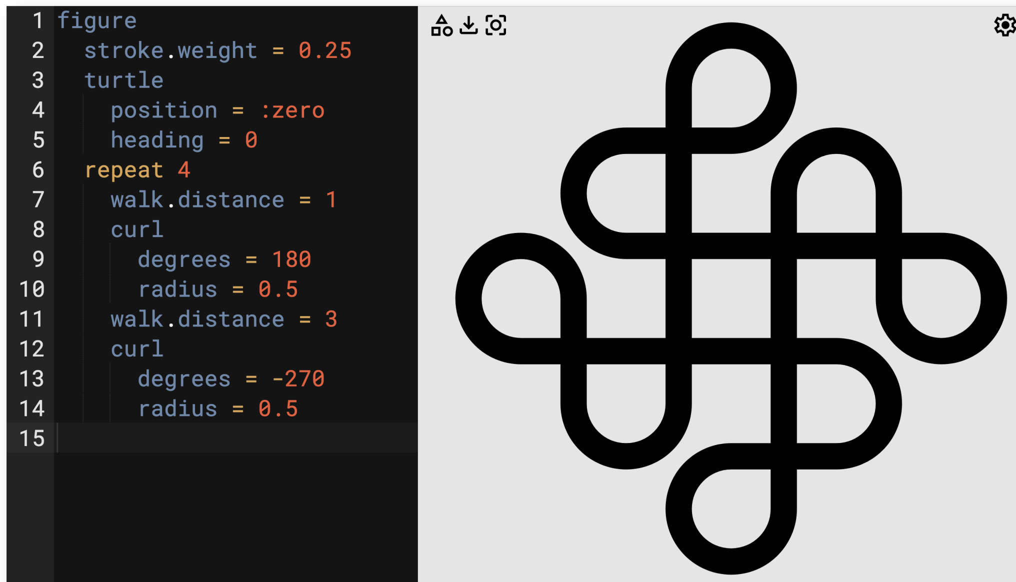


Figure 5: *A knot—which happens to be the unknot—programmed as a four-fold repetition of a walk-curl-walk-curl sequence.*

Foundaround

In the unlikely event that we have additional time, our third activity is modeling a found object with Bézier curves—a foundaround. Students learn about continuity and inflection points as they fit curves to the perimeter of a flat object that has curved edges. We follow these steps:

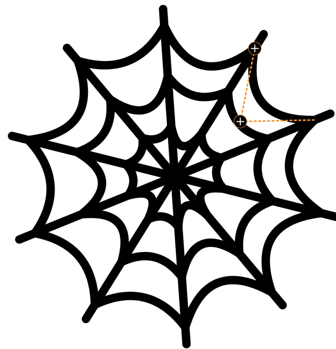


Figure 6: *A spider web made of rings of quadratic Bézier curves, with one selected for direct manipulation.*

1. The instructor briefly discusses the material and aesthetic importance of curvature in industrial design and shows some examples of logos and icons made from Bézier curves such as the one shown in Figure 6.
2. The instructor issues a challenge for students to locate a flat shape with at least some curved edges that they will model. Given the time constraints of the workshop, we will provide a collection of pre-found objects.
3. The students trace or sketch the found object onto paper.

4. The instructor demonstrates in Twoville how the control points of a Bézier curve act like a black hole, pulling a straight line segment into a curve. The instructor also demonstrates how curves switch from bending inward to bending outward and offers some guidance on placing the curves' vertices and control points in accordance with these inflection points.
5. The students identify the approximate locations of the vertices and control points of their shape. Exact coordinates are unnecessary.
6. The students take a photo of their found object and upload it as a *raster* in Twoville. A raster is a rectangle of pixels.
7. The students add vertices and control points to a `figure` by clicking around the raster and dragging on handles to fit the curve to the perimeter.
8. The instructor cuts the design into a sticker.

Because the students are not generating a new shape as they do in the previous two activities, this activity has less algorithmic synthesis. In fact, the curve-fitting could just as easily be done in a normal vector graphics editor that has no code interface. However, we believe that an explicit code representation better reveals the mechanics of Bézier curves.

At the conclusion of this activity, we present some questions and challenges to the students. You've seen Bézier curves with one or two control points. Could there be a curve with three control points? How does adding more control points affect the curve? Why are they called quadratic and cubic curves? In the next day, try making a circle in Twoville using Bézier curves.

Conclusions

We have introduced Twoville, a bidirectional editor for making two-dimensional designs. Twoville supports both direct manipulation of the design through code and indirect manipulation through mouse interaction. The two interfaces are synchronized so that changes in one appear in the other. The output of a Twoville program is a scalable vector graphics file that is ready to be sent to the control software of a fabrication device. We built Twoville in order to teach math and computer science and to support computational making activities in after-school programs and summer camps for students in our community. Students leave our activities with unplugged computational artifacts like vinyl stickers, T-shirts, embroidered greeting cards, and acrylic and plywood sculptures. We have described three activities that we will use in our workshop.

References

[1] *Cuttle*. https://cuttle.xyz.

[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* pp. 281–292. https://doi.org/10.1145/3332165.3347925.

[3] C. Johnson and I. McCormack. "Computational Making via Bidirectional Parametric Modeling." *Bridges Conference Proceedings*, Aug. 2–3, 2021, pp. 359–362. https://archive.bridgesmathart.org/2021/bridges2021-359.html.

[4] S. Lomonaco and L. Kauffman. "Quantum knots and mosaics." *Quantum Information Processing*, vol. 7, pp. 85–115. https://doi.org/10.1007/s11128-008-0076-7.

[5] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. 1980.

[6] *Twoville*. https://twoville.org.

[7] B. Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages." *Computer*, vol. 16, no. 8, pp. 57-69, Aug. 1983.