Blocks In, Blocks Out: A Language for 3D Models

Chris Johnson
Department of Computer Science
University of Wisconsin, Eau Claire
Eau Claire, Wisconsin
Email: johnch@uwec.edu

Peter Bui
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana
Email: pbui@nd.edu

***Abstract*—Madeup is a programming language for making things *up*—literally. Programmers write sequences of commands to move and turn through space, tracing out shapes with algorithms and mathematical operations. The language is designed to teach computation from a tangible, first-person perspective and help students integrate computation back into the physical world. We describe the language in general and reflect specifically on our recent implementation of its block interface.**

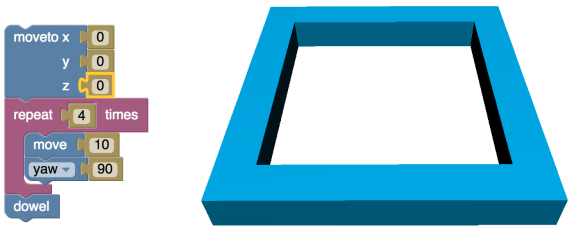
I. Introduction

Madeup is a programming language and development environment for teaching computer science and mathematics. Users write programs that walk along the skeletons or cross sections of geometric shapes and then issue commands to expand these paths into printable 3D models. Madeup was initially a text-based language, but now both blocks and text may be used to compose programs. The language facilitates an imperative and functional style of programming using standard expressions, conditionals, loops, functions, arrays, and turtle geometry commands. An example program is shown in Figure 1.

In this paper, we describe the Madeup language and our experiences in adding to it a block-based editor. In Section II we offer a detailed description of Madeup and some example programs. In Section III, we reflect on our implementation of a blocks interface. In Section IV, we summarize our work. Discussion of related work is integrated throughout the text.

II. Description

Madeup facilitates the generation of 3D models through a Logo-like imperative language and a browser-based development environment, the details of which we describe below. Introducing a new tool or language invokes a variety of reactions in the educational and technology community, and we also attempt to offer some justification for Madeup's entry into an already-crowded marketplace of educational tools.



(a) Madeup Source (b) Generated Model

Fig. 1. A frame generated by walking a square path and surrounding it with a square dowel structure.

A. Origin

Madeup was born out of a desire to generate 3D models in an algorithmic manner. Many 3D modeling programs provide excellent support for mouse-based sculpting or box modeling, but interaction with these programs tends to be driven by aesthetic feel and less by mathematical precision. Furthermore, altering an edit made several levels back in the undo stack is typically a destructive operation: in popping off the undo stack, the user loses the intervening edits. Reverting an edit made in a previous open-save-close cycle is usually impossible. In these programs, the user pays a high cost for mistakes made earlier but discovered later in the modeling process.

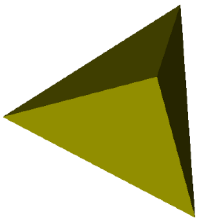
The code-based generation of Madeup solves both of these problems. First, Madeup narrows the scope of models that it can generate to "imperative shapes"—those that can be described by an imperative algorithm using mathematical and logical operators and flow control. Second, the entire process used to produce a model is expressed as a program instead of an ephemeral sequence of keypresses and mouse movement. New commands can be inserted anywhere in the program without eliminating subsequent commands. Parameters of existing commands can be tweaked without undoing or modifying others.

Other tools for programmatic modeling do exist. OpenSCAD and FormWriter [1], for example, provide textual scripting languages with which users may assemble complex models by combining simpler ones. These tools fit nicely in a professional workflow, but its mostly declarative language and its rich library of premade shapes reflects their primary purpose: to enable programmers to make shapes quickly. In Madeup, we are as interested in emphasizing *how* shapes are built as the shapes themselves. Several Logo-like languages exist for laser cutters [2], [3]. Given the constraints of this technology, programmers using these languages are restricted to 2D movement. The Beetle Blocks [4] project is quite similar to Madeup in spirit, but it supports fewer mesh generators and uses different mechanics for generating.

B. Mechanic

Most 3D graphics systems and 3D printers view a solid as a closed mesh of triangles. Internally, such meshes are represented as a list of vertex positions in Cartesian space and

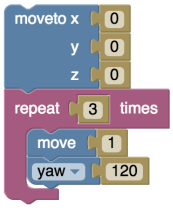
978-1-4673-8367-7/15/$31.00 ©2015 IEEE

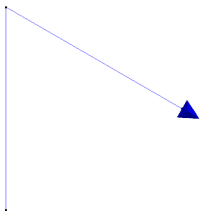


```
v 1 1 1
v -1 1 -1
v 1 -1 -1
v -1 -1 1
f 1 3 2
f 4 1 2
f 3 1 4
f 2 3 4
```

(a) Tetrahedron Model (b) Internal Representation

Fig. 2. A rendered tetrahedron and its internal representation. The internal representation is the OBJ model format, which lists the vertices as 3D positions and faces as 1-based indices into the vertex list.





```
0 0 0
0 1 0
0.866 0.5 0
0 0 0
```

(a) Triangle-walking code (b) Path after 2 moves (c) Path vertices

Fig. 3. A program that walks a triangular path is shown in (a). A preview of the path generated immediately after the second `move` command is shown in (b). The list in (c) is the final enumeration of vertices.

a list of triangular faces, with each face consisting of three integer indices into the positions list. Models of very simple shapes can be generated by hand, as was the tetrahedron shown in Figure 2. Mouse-based modeling programs do give users the power to add individual vertices to a model and connect groups of them into faces. However, most models aren't simple, and working with individual vertices is labor-intensive.

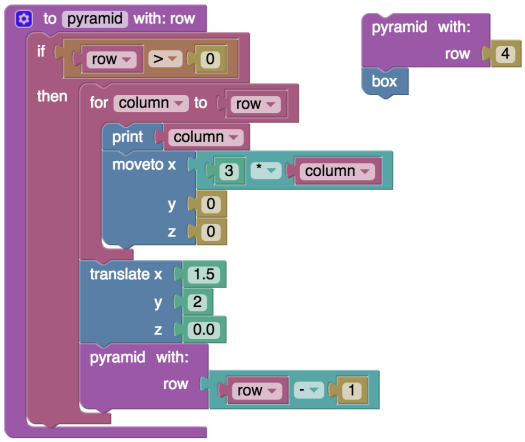
Madeup provides a simple mechanic to alleviate the burden of enumerating vertices. Instead of visiting each and every location on a mesh, the user walks a 1D path through 3D space using move and turn commands. An example is shown in the short triangle-walking program in Figure 3. The walked path is then interpreted in different ways by a handful of solidifiers to produce a solid model. Some solidifiers view the path as a cross section of the model to be expanded, others as a skeleton or list of centroids to be surrounded by solid geometry.

Vertex locations can be computed through standard expressions comprised of mathematical, relational, and logical operations; conditional logic; loops; array manipulation; and functional decomposition. A modest builtin library provides logarithmic, trigonometric, and other mathematical functions.

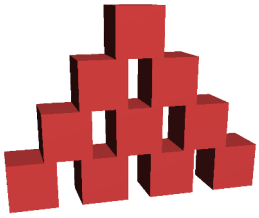
C. Solidifiers

Once the user has finished issuing `move` and `moveto` commands in a Madeup program, the visited vertices are given over to one of several solidifiers to produce a solid model. We describe a few of these supported solidifiers.

After a path has been used to generate a model, the list of vertices is cleared and a new path may be walked and used



(a) Madeup Source



(b) Generated Model

Fig. 4. A pyramid of boxes generated recursively. The first call to `pyramid` visits 4 locations, the next 3, and so on, up to the pyramid's top.

to generate a subsequent model. In this way, a single Madeup program may generate many models.

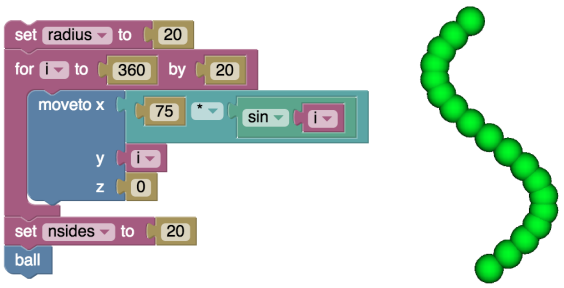
1) Box: The `box` generator surrounds each vertex in the walked path with a rectangular prism. The size of each box is determined by the value of the builtin variable `radius` at the time of the `move` or `moveto`. An example of this generator is shown in Figure 4.

2) Ball: Similar to `box`, the `ball` generator surrounds each vertex in the walked path with a faceted sphere. The radius of each sphere is determined by the value of the builtin variable `radius` at the time of the `move` or `moveto`. The number of facets on each sphere is determined by the value of the builtin variable `nsides` at the point where the `ball` command is issued. An example of this generator is shown in Figure 5.

3) Dowel: The `dowel` generator interprets the walked path as the skeleton of a filled polytube structure. The path is surrounded by geometry whose cross section is a regular polygon. The number of sides of the polygon is determined by the variable `nsides`. The radius of the polygon may change from vertex to vertex by assigning differing values to `radius`. An example of this generator is shown in Figure 1.

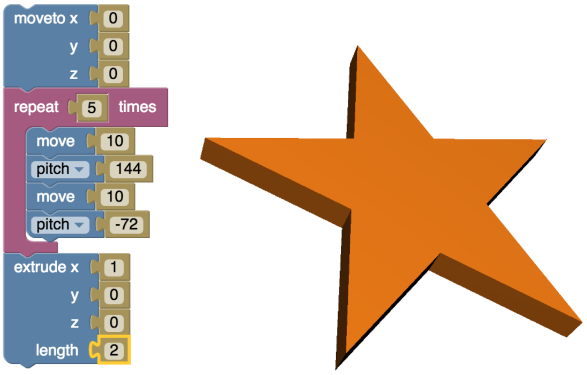
4) Extrude: The `extrude` generator interprets the walked path as the cross section of a solid to be extended along a given axis for a given length. An example of this generator is shown in Figure 6.

5) Revolve: The `revolve` generator interprets the walked path as the cross section of a solid to be revolved or spun



(a) Madeup Source (b) Generated Model

Fig. 5. A program that uniformly samples a sine wave every 20 degrees. The `ball` generator surrounds each vertex in the path with a sphere of a given radius.



(a) Madeup Source (b) Generated Model

Fig. 6. A program that walks a star path in the $x = 0$ plane. The `extrude` generator expands the path in a given direction for a given length.

around a given axis a given number of degrees. An example of this generator is shown in Figure 7.

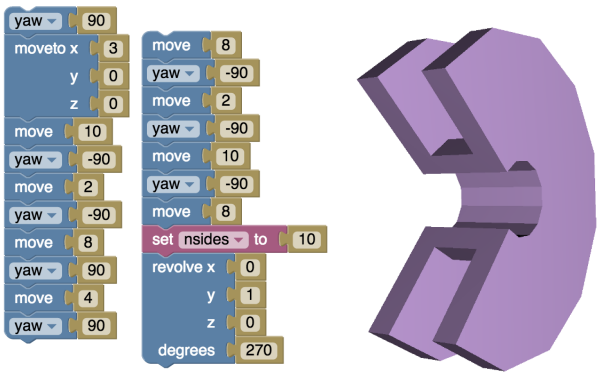
6) Surface: The `surface` generator interprets the walked path as the serialization of an implicit 2D grid structure. The visited vertices position the nodes of the grid, and these vertices join with adjacent nodes to form a surface. An example of this generator is shown in Figure 8, with the implied connectivity illustrated in the wireframe rendering in Figure 8c. This generator is useful for composing 2D parametric surfaces like cones, cylinders, spheres, planes, Möbius strips, Klein bottles, and so on. Unlike the other solidifiers, `surface` may produce models that are not actually solid.

D. Printing

Once the programmer's paths have been solidified, the models may be downloaded in an OBJ mesh format, which most 3D printing and computer graphics packages recognize. At present, many printers are managed by proprietary software, and we make no effort to connect to a printer directly.

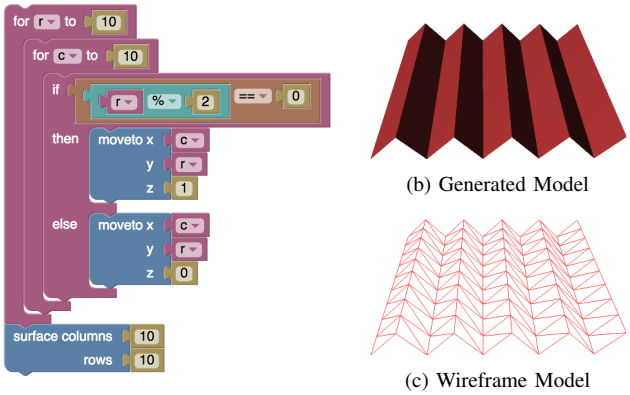
E. Education

As we began developing Madeup, we saw that it provided a context for tangibly exploring mathematics, algorithms,



(a) Madeup Source (b) Generated Model

Fig. 7. A program that walks a C-shaped path 3 units to the right of the y-axis. The `revolve` generator spins the path around the y-axis 270 degrees, stopping 10 times.



(a) Madeup Source

(b) Generated Model

(c) Wireframe Model

Fig. 8. A program that walks a C-shaped path 3 units to the right of the y-axis. The `revolve` generator spins the path around the y-axis 270 degrees, stopping 10 times.

technology, and other STEM-related disciplines. We informally validated its usability and motivational power through several workshops and camps for young learners.

Madeup joins a crowd of existing teaching tools, including Scratch, App Inventor, Pencil Code, and many others. What sets Madeup apart from existing projects is its physical product. The model that a programmer creates does not remain virtual. It can be printed, felt, carried in a pocket, and handed to a parent or friend—all of which may make computation real and relevant in the eyes of the programmer.

We believe the spatial domain is of particular importance to scientific learning. In fact, two specific spatially-oriented cognitive processes have frequently been found to associate with successful participation in STEM: mental rotation and cross section inference.

Shepard and Metzler [5] define *mental rotation* as the ability to imagine how an object would appear if rotated about an axis. An individual's performance on mental rotation tasks has been found to be a strong predictor of success in STEM domains [6]. Strong mental rotation skills have been found amongst dentists [7] and pilots [8]. Bodner and

Guay [9] observed chemistry students and found that spatial ability accounted for 15% of the variance on the students' exam scores. Interestingly, students with lower spatial abilities performed just as well as those with higher spatial abilities on problems requiring only memorization. However, on problems that required more advanced program solving, students with higher spatial abilities significantly outperformed their peers—even on problem solving exercises without a spatial component.

Cohen and Hegarty [10] define *cross section inference* as the ability to infer a 2D cross section given a 3D representation of an object. This skill has been found to positively correlate with success in anatomy [11], radiology [12], geology [13], [14], geometry [15], [16], and engineering [17], [18], [19].

Both rotations and cross sections play significant roles in the generation of solid objects in Madeup. The paths that a Madeup user programmatically generates are interpreted as cross sections that are extruded or rotated to form a 3D model. As its users reverse engineer existing objects that they encounter, they infer cross sections and mentally rotate them in the design process. We hypothesize that students' abilities in these two activities will increase significantly by using Madeup.

Stieff [20] demonstrated that when students have awareness of both spatial and higher-level, non-spatial problem solving strategies, the non-spatial strategies may not be activated until they have been contextualized in a particular setting. This suggests that spatial strategies like mental rotation are versatile and present a lower barrier to application. Thus, spatial abilities provide an important bridge that can lead to advanced understanding of many domains.

If spatial ability is a strong indicator of success in STEM, perhaps we can increase participation in STEM by improving learners' spatial abilities? Tuckey et al. [21] found a two-hour workshop on techniques for translating 2D representations to 3D representations yielded significantly higher exam scores. Sorby [22] administered a spatial training program for university engineering students, which boosted their scores on a spatial aptitude exam by an average of 27% and improved their first-year grades.

III. Blocks

A block interface was recently added to Madeup using the Blockly framework, and we reflect on our motivation and experiences in adding support for block composition of programs.

A. Embodied Learning

Madeup builds significantly on the work of Seymour Papert and the Logo project. In one of Papert's early implementations of Logo, students were supplied a physical robot that could be programmed to move and turn, optionally tracing its path on paper. This physical manifestation of Logo faithfully promoted what Papert called *body syntonic learning*, in which a learner comes to identify closely with an external manipulative. In such learning, an otherwise third-person observation by the learner effectively transforms into a more engaging first-person experience.

The Logo environment can also be simulated virtually. Instead of a physical robot, an onscreen turtle traces out the programmed path. Certainly, a simulated environment has administrative and economic advantages: there's less hardware to manufacture, assemble, and maintain. But are virtual environments a substitute for physical ones?

Eisenberg and Buechley [23] argue that for the power of computers to be fully realized and for us to fully express ourselves, computers' output must be more than digital and printers must produce more than ink on paper. Out of these beliefs came the Lilypad Arduino, a computing platform that can be integrated into one's clothing. Further, Buechley et al. [24] state that we often falsely limit technology to two applications: automation and entertainment. If we only view technology as a vehicle to simplify our lives and increase pleasure, we neglect a large portion of the human experience. They recommend that we also consider "technology as expanding and democratizing the range of human expression and creativity."

Djajadiningrat et al. [25] identify three distinct periods of our interaction with technology products:

- The electro-mechanical period prior to World War II, in which devices were controlled directly by levers and cranks, giving rich and direct feedback to their operators.
- The analog electrical period prior to the 1980s, in which dials and sliders gave operators only indirect control over their products' behaviors accompanied by indirect feedback.
- The digital electrical period of the present, in which our primary means of interacting with a device is pushing on its buttons or screen.

As human society has moved away from direct physical control of our devices, we have fewer opportunities to develop bodily skill. Instead, we place the burden of interaction on cognition, with which we are less likely to receive direct feedback and more likely to make errors.

Programmers write Madeup code in a development environment that draws upon both cognition and physical experience. The user interface, shown in Figure 9, provides continuous immediate feedback about the programmer's location in space and the direction she is currently facing. A preview of each path shows what locations have already been visited. By shortening the time between expression of a command and its execution, we expect Madeup users to engage more readily in syntonic learning.

Block-based interfaces recognize the importance of the physical world in the learning process. Composing a program with blocks is a metaphor for assembling a puzzle. Blocks are colored by kind. Execution flows from top to bottom through a sequence. If learning is made more accessible through an interface with physical elements, then will not learning and engagement be further enhanced by physical output? Madeup espouses the principal that as is the input, so should be the output. We hypothesize that the virtual focus of many block languages is holding back the long-term and widespread adoption of computer science education. By appealing to physical metaphors in the interface, we have made great progress in

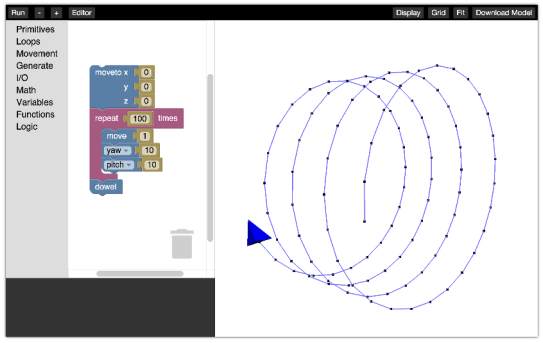


Fig. 9. The Madeup development environment. The left panel toggles between a block-based and text-based code editor. A preview of the paths generated so far appear in the canvas on the right, with the cursor's current orientation and location shown by an arrowhead glyph. Previews are shown in real-time as edits are made. The more computationally-intensive models are generated only when users hit the Run button in the top menu.

lowering the barrier to computational thinking. By further applying block interfaces to applications that produce enduring tangible output, we expect administrators and teachers to take our discipline more seriously.

Thornburg [26] identifies engineering and technology as "the glue that holds the other STEM subjects together." He holds that in the US participation in STEM is low because most K-8 schools do not have mechanical shops where young learners may tinker and build. We are sensitive to schools' limited opportunities to significantly alter their facilities, and we see Madeup and 3D printers as an affordable starting point to bring a shop-like setting into schools. These printers are small and contained, and do not require significant changes in infrastructure.

B. Statements vs. Expressions

Madeup supports a functional style of programming, with every construct of Madeup serving as an expression yielding some value. As we implemented a blocks interface for the language, we had to overcome what may be an inescapable feature of most block languages: they are geared toward imperative thinking.

In an imperative language, conditionals, loops, and procedure calls appear as statements. In imperative block languages, these constructs are represented with blocks having sequence connectors so that other statements may snap in before or after them. In a functional language, these may appear as either statements or expressions—they are *amphibious*. In many functional languages return statements are also amphibious, often appearing as a bare expression. As statements, such blocks need sequence connectors. As expressions, they need value connectors.

We considered four possible solutions to supporting amphibious constructs. Most involve altering how the blocks appear in the palette:

1) Augment blocks to simultaneously have both sequence and value connectors. This approach seems aesthetically displeasing.
2) Represent each amphibious construct in two ways: once as a statement block and once as an expression block, as demonstrated in Figure 10. If there are many amphibious blocks, this approach introduces clutter in the palette. At present, this is the option we have implemented for conditionals and function calls in Madeup.
3) Show the blocks as expressions with value connectors, but provide an expression-to-statement converter block. This is also implemented in Madeup, but we've only found it necessary to use for return expressions.
4) Show the blocks in one form or the other, but allow their connectors to be explicitly modified once placed in the editor. This approach manifestly deceives the programmer about a block's affordances.
5) Show the blocks in an undifferentiated format, but dynamically modify their connectors according to the context in which they are placed.

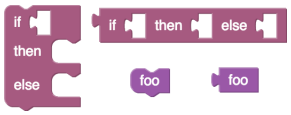


Fig. 10. Conditionals and function calls may appear as either expressions or statements in Madeup. Currently, we provide both forms in the blocks palette.

C. Cursor

A feature that text-based development environments exploit but block-based environments manage without is the notion of a cursor. In a text editor, a cursor provides location and context. The contents of the clipboard can be pasted at the cursor. A smart autocompletion system suggests only names in scope at the point of the cursor.

What could a block interface gain by adding a cursor? Currently, blocks are moved almost exclusively by dragging and dropping. This interaction is not accessible to all users. With the notion of a cursor, we can respond to double-clicking on a block in the palette by connecting it at the point of the cursor. The variables shown in the palette can be restricted to only those that are in scope at the point of the cursor. The cursor can be treated as a momentary breakpoint, and a "run until cursor" feature could be aided to aid in debugging and demonstration.

At least one blocks framework—Blockly—does support selection. (Scratch and Pencil Code do not.) Exactly one block may be selected at a time. This draws visual attention to the block, and it may be copied or deleted using keyboard shortcuts. The functionality of selection could be enhanced to provide context and aid in accessibility.

Selection by definition requires an existing block to select. A cursor on the other hand selects the "nothing" between blocks. Spreadsheet software is cursorless: to insert new cells, one must first select an adjacent cell. However, the insertion point is ambiguous, and the user must further specify whether the new cells should appear before or after the selection. A cursor is not ambiguous.

D. Sequential vs. Random Access Editing

Compared to a text-based language, statements in a block-based language are explicitly linked to one another. This tighter

coupling introduces higher editing costs compared to a textual editor. For example, to swap the order of two blocks, one must first detach any trailing blocks, drag the second before the first, and then reattach the trailing blocks. This mandatory consideration of the neighborhood of block-based code is analogous to editing a linked list: one must maintain the links. By representing code with text, we essentially gain random access and can swap two lines without disturbing any others.

Many editing operations are needed so infrequently that their higher cost can be ignored. However, as we implemented Madeup's blocks interface, one editing operation that we found missing from many block languages is that of temporarily disabling a statement by commenting it out. The importance of this mechanic is not superficial. When debugging an algorithm by commenting out portions, code must be fluid. We've seen users work around the lack of disable-by-comment in two ways:

- By physically breaking the block out of the flow. The cost of disabling a block is high enough to inhibit experimentation.
- By artificially embedding the block in an if-then statement. This enables fast toggling, but applying this to many blocks obscures the intent of the program.

We suggest that more block-based platforms enable temporary disabling of a block through a contextual menu—to keep the visible interface spare—and by maintaining for each block a status flag that can be checked by the code generator. This behavior is already available in Blockly, OpenBlocks, and App Inventor.

IV. Conclusion

We have introduced Madeup, a block- and text-based programming environment for generating 3D models. Using turtle geometry and other spatial commands, learners assume a first-person role in Papert's "Mathland" as they trace out cross sections and skeletons of shapes. Writing Madeup programs invokes cognitive processes that have been found to promote successful participation in STEM learning. The physical output of a Madeup program can help root computer science and mathematical concepts in the physical world.

Recently we added a block interface to the language, and we have offered our reflection on the user experience found in many block languages.

References

[1] M. D. Gross, "Formwriter: A little programming language for generating three-dimensional form algorithmically," in *CAAD Futures*, 2001, pp. 577–588.

[2] F. Turbak, S. Sandu, O. Kotsopoulos, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, Sept 2012, pp. 137–144.

[3] G. Johnson, "Flatcad and flatlang: Kits by code," in *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, Sept 2008, pp. 117–120.

[4] "Beetle Blocks," http://beetleblocks.com, [Online; accessed 5-July-2015].

[5] R. Shepard and J. Metzler, "Mental rotation of three-dimensional objects," *Science*, no. 171, pp. 701–703, 1971.

[6] J. Wai, D. Lubinksi, and C. P. Benbow, "Spatial ability for STEM domains: Aligning over 50 years of cumulative psychological knowledge solidifies its importance," *Journal of Educational Psychology*, vol. 101, no. 4, pp. 817–835, 2009.

[7] D. Moreau, J. Clerc, A. Mansy-Dannay, and A. Guerrien, "Enhancing spatial ability through sport practice: Evidence for an effect of motor training on mental rotation performance." *Journal of Individual Differences*, vol. 33, no. 2, p. 83, 2012.

[8] I. E. Dror, S. M. Kosslyn, and W. L. Waag, "Visual-spatial abilities of pilots." *Journal of Applied Psychology*, vol. 78, no. 5, p. 763, 1993.

[9] G. M. Bodner and R. B. Guay, "The Purdue visualization of rotations test," *The Chemical Educator*, vol. 2, no. 4, pp. 1–17, 1997.

[10] C. A. Cohen and M. Hegarty, "Inferring cross sections of 3D objects: A new spatial thinking test," *Learning and Individual Differences*, vol. 22, no. 6, pp. 868–874, 2012.

[11] J. Russell-Gebbett, "Skills and strategies–pupils' approaches to three-dimensional problems in biology," *Journal of Biological Education*, vol. 19, no. 4, pp. 293–298, 1985.

[12] M. Hegarty, "Components of spatial intelligence," *Psychology of Learning and Motivation*, vol. 52, pp. 265–297, 2010.

[13] Y. Kali and N. Orion, "Spatial abilities of high-school students in the perception of geologic structures," *Journal of Research in Science Teaching*, vol. 33, no. 4, pp. 369–391, 1996.

[14] N. Orion, D. Ben-Chaim, and Y. Kali, "Relationship between earth-science education and spatial visualization," *Journal of Geoscience Education*, vol. 45, pp. 129–132, 1997.

[15] E. H. Brinkmann, "Programed instruction as a technique for improving spatial visualization." *Journal of Applied Psychology*, vol. 50, no. 2, p. 179, 1966.

[16] M. Pittalis and C. Christou, "Types of reasoning in 3D geometry thinking and their relation with spatial ability," *Educational Studies in Mathematics*, vol. 75, no. 2, pp. 191–212, 2010.

[17] R. T. Duesbury *et al.*, "Effect of type of practice in a computer-aided design environment in visualizing three-dimensional objects from two-dimensional orthographic projections." *Journal of Applied Psychology*, vol. 81, no. 3, p. 249, 1996.

[18] H. B. Gerson, S. A. Sorby, A. Wysocki, and B. J. Baartmans, "The development and assessment of multimedia software for improving 3-D spatial visualization skills," *Computer Applications in Engineering Education*, vol. 9, no. 2, pp. 105–113, 2001.

[19] S. P. Lajoie, "Individual differences in spatial ability: Developing technologies to increase strategy awareness and skills," *Educational Psychologist*, vol. 38, no. 2, pp. 115–125, 2003.

[20] M. Stieff, "Mental rotation and diagrammatic reasoning in science," *Learning and instruction*, vol. 17, no. 2, pp. 219–234, 2007.

[21] H. Tuckey, M. Selvaratnam, and J. Bradley, "Identification and rectification of student difficulties concerning three-dimensional structures, rotation, and reflection," *Journal of Chemical Education*, vol. 68, no. 6, p. 460, 1991.

[22] S. A. Sorby, "Educational research in developing 3-d spatial skills for engineering students," *International Journal of Science Education*, vol. 31, no. 3, pp. 459–480, 2009.

[23] M. Eisenberg and L. Buechley, "Pervasive fabrication: Making construction ubiquitous in education," *Journal of Software*, vol. 3, no. 4, 2008. [Online]. Available: http://ojs.academypublisher.com/index.php/jsw/article/view/03046268

[24] L. Buechley, M. Eisenberg, J. Catchen, and A. Crockett, "The LilyPad Arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2008, pp. 423–432.

[25] T. Djajadiningrat, B. Matthews, and M. Stienstra, "Easy doesn't do it: Skill and expression in tangible aesthetics," *Personal Ubiquitous Comput.*, vol. 11, no. 8, pp. 657–676, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00779-006-0137-9

[26] D. Thornburg, "Hands and minds: Why engineering is the glue holding STEM together," *Thornburg Center for Space Exploration. Retrieved from http://www.tcse-k12.org/pages/hands.pdf*, 2009.