Blocks Languages for Creating Tangible Artifacts

Franklyn Turbak, Smaranda Sandu, Olivia Kotsopoulos, Emily Erdman,
Erin Davis, Karishma Chadha, and Johanna Okerlund
Computer Science Department, Wellesley College
Wellesley, Massachusetts, USA
Email:{franklyn.turbak, smaranda.sandu, olivia.kotsopoulos,emily.erdman,
erin.davis, karishma.chadha, johanna.okerlund}@wellesley.edu

Tinkerblocks Technical Report 2012-1
Draft version as of July 4, 2012

Abstract

Logo turtles and Henderson's picture language have long been used to teach computational thinking by inspiring learners to construct programs that create complex geometric designs. We have developed visual blocks-based versions of these languages, *TurtleBlocks* and *PictureBlocks*, that allow users to transform their designs into tangible artifacts produced by laser cutters and vinyl cutters. Our languages embody two novel features. First, they use constructive area geometry to convert the geometric designs generated by our programs into formats suitable for laser and vinyl cutters. Second, they leverage static typing and polymorphism to provide a new way to reference the names of procedure parameters and local variables in a blocks language.

Note: This is an extended technical report version of our VL/HCC 2012 conference paper [1].

I. Introduction

Rapid prototyping machines such as vinyl cutters, laser cutters, and 3D printers allow designers to quickly turn complex designs into tangible artifacts made out of card stock, wood, and plastic. As their costs decrease, these machines are becoming more commonplace, especially in educational settings, ushering in an era in which ordinary people can fabricate artifacts from their own designs ([2], [3]).

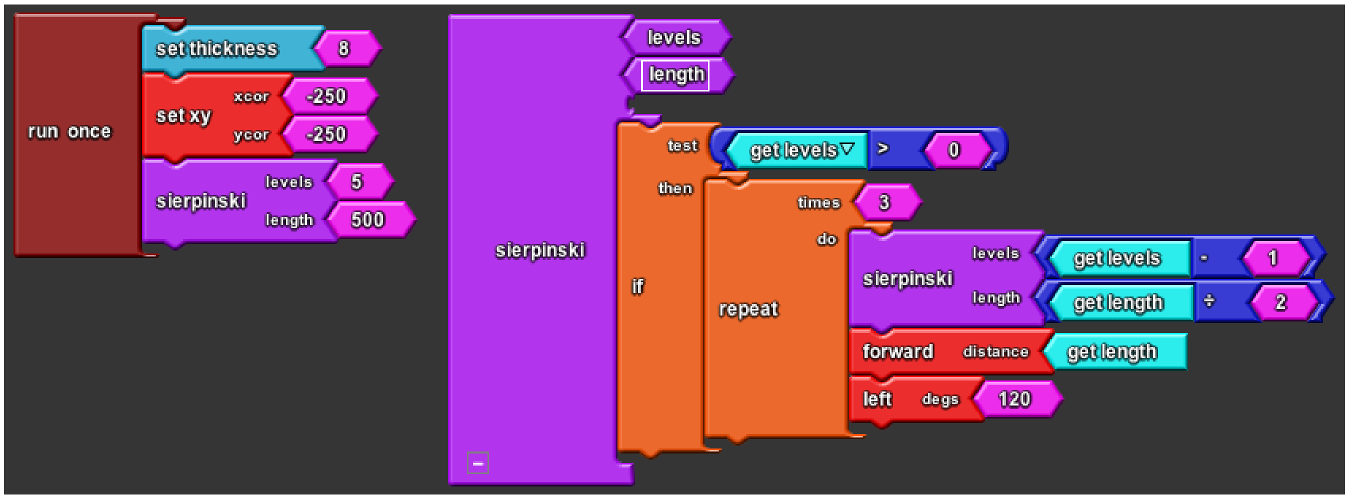
At Wellesley College, we are seeking to make it easier for our community members to create tangible artwork. Although the 3D printer offers exciting possibilities, the simplicity of the 2D designs for the vinyl and laser cutters have made these machines our primary focus. In the simplest case, designs are just a set of lines and curves to be cut. The vinyl cutter stylus can cut these designs on thin materials such as card stock and adhesive-backed vinyl sheets. The laser cutter can cut these designs in sheets of wood and various kinds of plastic of thicknesses up to a quarter inch. Designs for the laser cutter can also specify areas to be engraved and the depth of engraving.

Any 2D drawing application can be used to create designs for the vinyl and laser cutters, but many applications (e.g., Adobe Illustrator and Corel Draw) have a steep learning curve that makes it difficult to create artifacts that are satisfying to novices. Other details further complicate turning a 2D design into an artifact. For example, only red lines with hairline thickness specify where the laser should cut; any other color or thickness is ignored. And our vinyl cutter only accepts files in DXF format, which is not an available export option in many 2D drawing applications.

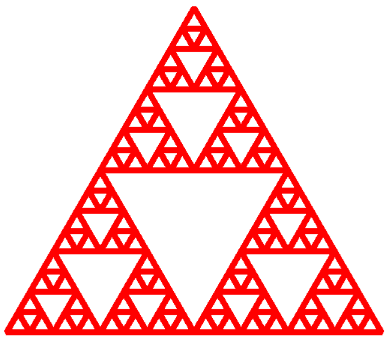
Our goal was to create simple 2D design environments in which users can quickly create interesting artifacts for vinyl and laser cutters. There are numerous domain-specific environments that allow novices to make 2D and 3D designs for fabrication, many of which involve sketching or gesturing (e.g., [4]–[6]). But we also wanted the environments to introduce nonprogrammers to *computational thinking* [7] and give them hands-on experience with techniques like procedural abstraction, modularity, and divide/conquer/glue problem solving. We were inspired by the Eisenbergs' *Craft Technology Group* at the University of Colorado, Boulder, which focuses on computer science activities that involve creating personally meaningful physical artifacts, including those fabricated on rapid prototyping machines ([8], [9]). Another exemplar of this spirit is Johnson's FlatLang, a Logo-like language for specifying laser-cut construction kit parts [10].

We adapted two existing environments for creating geometric designs. The first is turtle geometry, in which a virtual creature with a pen in its belly draws shapes by following a series of simple commands written by the programmer. Decades before the phrase "computational thinking" was coined, Papert and the Logo community used turtles to teach children to think computationally by having them instruct turtles to draw geometric shapes ([11], [12]). The second is Henderson's picture language ([13], [14]), which facilitates the construction of complex geometric designs from simple primitive pictures by transforming (rotating, flipping) pictures and composing them (putting one picture above, beside, or over one another). This picture language was popularized in [15], and for over a decade we have used both it and turtles in Wellesley's introductory Java programming class as visual contexts for teaching methods and recursion [16].

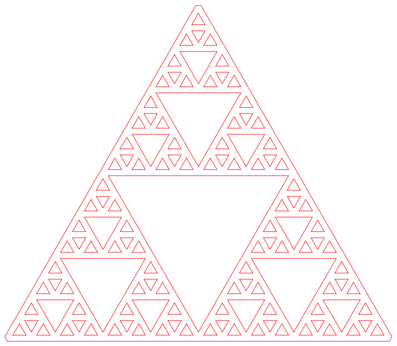
We developed blocks programming languages, *TurtleBlocks* and *PictureBlocks*, for these two computational approaches to 2D geometric design. Blocks languages were pioneered in Glinert's BLOX [17] and are currently epitomized by Scratch ([18]–[20]), App Inventor ([21], [22]), and StarLogo TNG ([23],



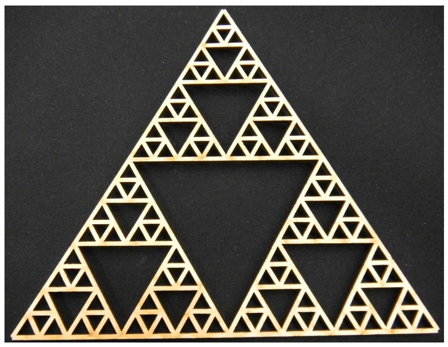
a. A *TurtleBlocks* program for drawing Sierpinski's gasket.



b. Turtle drawing from program.



c. Boundary of the turtle drawing.



d. Laser-cut artifact

Fig. 1. A *TurtleBlocks* program and its output.

[24]). In these languages, programs are created by connecting visual program fragments shaped like jigsaw puzzle pieces. The shapes of these blocks help novices avoid frustrating syntax errors commonly encountered in textual programming languages by visually suggesting how expressions and statements are combined to form programs. Blocks are arranged into drawers according to function, so that programmers can search for an appropriate block rather than having to remember the name of the construct, ameliorating another problem with text-based programming.

PictureBlocks is the first block–based implementation of Henderson's picture language, making the latter accessible to a much broader audience. On the other hand, there have been many previous turtle implementations in blocks languages. Indeed, *LogoBlocks* [25], an early blocks programming language, was first targeted at turtle programming[26]. Sprites in Scratch can be programmed to create turtle drawings. StarLogo TNG has turtles, but they are used mainly for multi-agent simulations rather than for generating geometric figures. TurtleArt [27] is a blocks-based turtle environment whose main focus is creating geometric designs.

TurtleBlocks and *PictureBlocks* offer the novel ability to produce tangible artifacts on laser and vinyl cutters, a feature unsupported by existing turtle and picture environments. For example, Fig. 1a shows a complete *TurtleBlocks* program that draws a level 5 Sierpinski gasket. Running this program generates the drawing in Fig. 1b. Selecting the printing option for a laser-cut design yields the cutting lines in Fig. 1c, and cutting a wooden sheet yields the artifact in Fig. 1d. We present our technique for generating tangible output in Sec. II.

Another feature that sets our languages apart from other blocks languages is the way that they leverage static typing and polymorphism to reference the names of procedure parameters and block-structured local variables. This fixes naming problems in blocks languages like App Inventor and StarLogo TNG. We describe this in Sec. III.

We conclude in Sec. IV with a discussion of user feedback on our languages from several workshops.

Due to space limitations, here we summarize only the key aspects of our work. For a more detailed discussion with additional examples, see our companion technical report [28].

II. Creating Tangible Output

A. Cutting

A novel feature of *TurtleBlocks* and *PictureBlocks* is that they generate tangible artifacts on laser and vinyl cutters. The core problem encountered in providing this feature is that the input to these machines are specifications of lines and curves to be cut, but these can be challenging to specify in many drawing programs or in traditional turtle drawing. For example, the cutting lines for the star shapes in Fig. 2a and Fig. 2b can be difficult to specify in many drawing applications. The simplest turtle program for creating a star — repeat 5 times going forward a fixed distance and turning left by 144 degrees — yields the lines in Fig. 2c. This drawing is not suitable for cutting because it specifies 6 disjoint pieces (5 triangles and a pentagon). This can easily be modified to specify the outline in Fig. 2a by repeating 5 times the following code: go forward the side length of one stellation, turn left 144 degrees, go forward the same length, and turn right by 72 degrees. But a turtle program to specify the lines in Fig. 2b would be much more complicated.

Our key observation regarding the problem of specifying cutting lines was that these are often the natural boundaries of 2D areas that are significantly easier to specify. For example, the cutting lines in

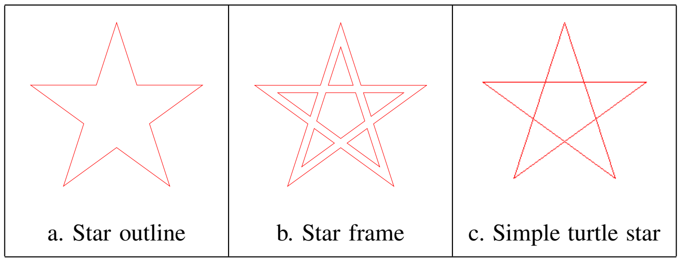


Fig. 2. Cutting lines for two star shapes.

Fig. 2a and Fig. 2b are the boundaries of the areas in Fig. 3. The star area in Fig. 3a is just a filled version

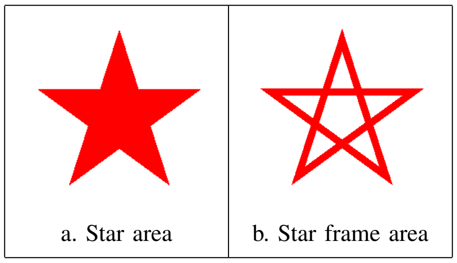


Fig. 3. Cutting lines for two star shapes.

of the simple turtle star in Fig. 2c, and the star frame area in Fig. 3b is just a version of the simple turtle star drawn with a thick pen. Both of these pictures are easy to create in *TurtleBlocks* because there are commands for filling the path generated by an arbitrary sequence of turtle commands and for setting the pen thickness of the turtle.

This leaves one problem: how do we transform areas like those in Fig. 3 to boundaries like those in Fig. 2? Our solution is to use *computational area geometry (CAG)*, which defines a notion of area and a set of operations on areas:

- An *area* is a set of points on a Cartesian plane bounded by a closed path consisting of straight or curved lines.
- Two areas can be combined by union, intersection, difference, and exclusive-or to yield another area. For example, Fig. 4 shows two areas and the areas that result from these four operations on the two areas. Note that a single area can have multiple disjoint components bounded by a single path with disconnected components, such as the intersection and difference areas in Fig. 4d and Fig. 4e.
- From any area, it is possible to extract a path for that area (the red lines in Fig. 4).

CAG operations are provided by some programming libraries and 2D drawing applications. We use those provided by Java's `java.awt.geom.Area` interface and implemented in the `java.awt.Shape`

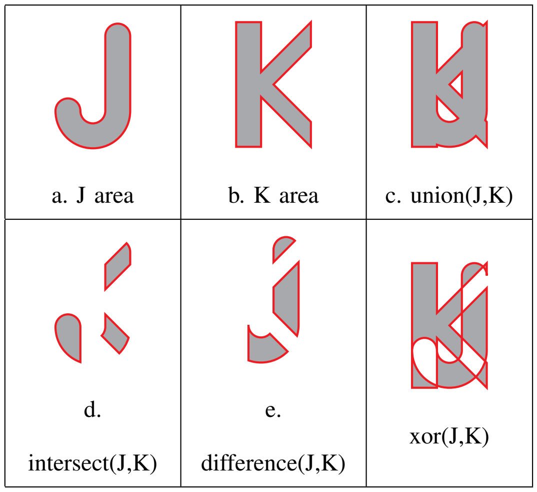


Fig. 4. Areas J and K and combining operations on them.

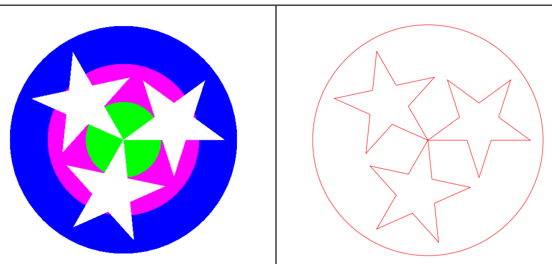
class. This library supports areas created from rectangles, ellipses, arcs, and the interiors of paths composed of lines and splines.

In *TurtleBlocks* and *PictureBlocks*, our approach to generating cutting patterns is to use these CAG operations to calculate boundaries as follows:

1) The languages provide simple ways to create primitive areas. In *TurtleBlocks*:
   - Each line drawn by the turtle is a rotated rectangular area determined by the length of the line and the turtle's thickness and heading.
   - There are commands for generating filled rectangles, ellipses, and arcs centered at the turtle. All of these are rotated according to the turtle's heading.
   - There is a command for filling the path generated by a given sequence of turtle commands.

   In *PictureBlocks*:
   - There are blocks to create primitive pictures for lines and filled and unfilled rectangles, ellipses, arcs, and polygons.
   - Users can specify collections of these filled areas in a file that can be loaded into *PictureBlocks* as a single picture.
   - Using a simple sketching application provided by *PictureBlocks*, users can draw colored areas (rectangles, ellipses, arcs, and polygons) that can be saved to a file and loaded as a single picture.
2) By default, multiple areas in drawings are combined using the CAG union operator. In *TurtleBlocks* all lines and other areas drawn by a turtle are simply unioned together. In *PictureBlocks*, areas in

pictures that touch or intersect are unioned together. The result of this step is a single area that might consist of multiple disconnected components. In the union operations of this step, colors of areas are ignored, except that white areas are subtracted from the other areas by the CAG difference operator. For example, the picture Fig. 5a yields the boundary in Fig. 5b because the white stars are effectively "cut out" of the rings (whose colors are ignored).



a. Nested rings with stars "cut out".

b. Boundary of the rings with stars.

Fig. 5. White areas are subtracted from non-white areas.

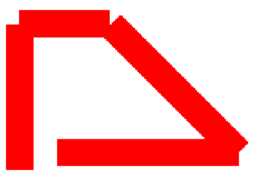
3) In the final step, the CAG operation for enumerating the path enclosing an area is applied to the result of the previous step. The lines and splines in this path can be displayed to the user and written to file formats such as PostScript and DXF that can be used to specify cutting on a laser or vinyl cutter.

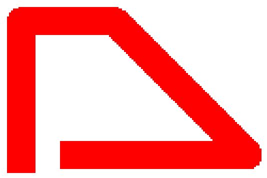
In *TurtleBlocks* and *PictureBlocks*, the user first creates designs on the screen. When they print their designs, they can specify that they want to generate a file for the laser or vinyl cutter. In this case, the boundary for their drawing is calculated using steps #2 and #3 as described above, and the lines and splines of this boundary are displayed on the screen and also saved to a file in an appropriate format.

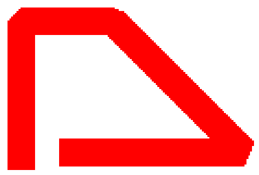
The above discussion glosses over the issue of dealing with corners in *TurtleBlocks*. When a turtle turns between drawing two thick lines (represented as filled rectangles), it leaves notches that are an eyesore (Fig. 6a). The turtle can draw various kinds of corners to fill the notches between two lines; these can be controlled by the programmer (Fig. 6b–d).

B. Engraving

A laser cutter can also engrave materials. In engraving mode, the laser cutter "prints" a picture by converting it to a bitmap, and then processes each row of the bitmap by moving the laser in a line and adjusting its power according to the color of the corresponding pixel. The laser cutter operator can specify the relationship between color and laser power, and thus engraving depth.


a. NONE corner type.


b. ROUNDED corner type.


c. BLUNT corner type.

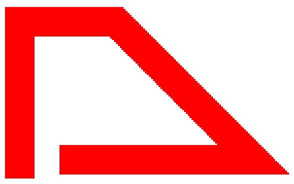
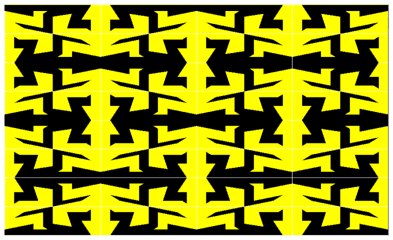
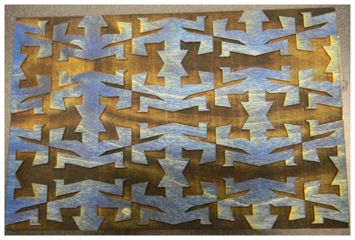

d. SHARP corner type.

Fig. 6. Output from a *PictureBlocks* program.


a. Sample picture


b. Picture etched into wood.

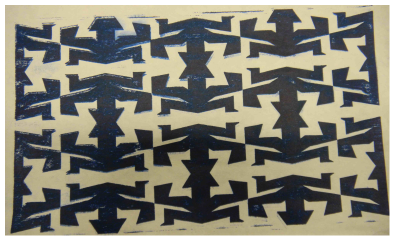

c. Print made from the etching.

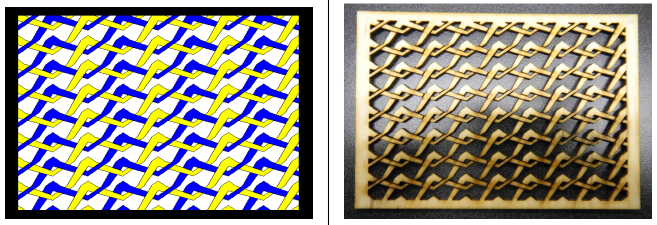
Fig. 7. Output from a *PictureBlocks* program.

TurtleBlocks and *PictureBlocks* support an engraving mode in which the colored lines and areas of the patterns are passed to the laser cutter unaltered. Users can make designs with multiple colors, and then control the depth for each color by adjusting the settings on the laser cutter. Fig. 7b shows the result of engraving into wood the *PictureBlocks* picture in Fig. 7a. In the engraving, the black areas of the picture were engraved, but the yellow areas were not. The engraving was deep enough that the wood piece could be used to create the print in Fig. 7c using traditional wood block printmaking techniques.

C. Combining Engraving and Cutting

For some designs, we wish to combine engraving and cutting. Consider the ring-and-star design in Fig. 5a. When creating a wooden artifact from this design, we might like to cut out the stars and engrave the rings on the remaining wood. For this purpose, we provide an engrave-and-cut mode that overlays the cutting boundary in Fig. 5b on top of the colored picture in Fig. 5a. When given this picture, the laser cutter first engraves the colored areas and then cuts along the red boundary. In this mode, the color red is treated specially (for cutting) and so cannot be used for engraving areas. But any red hairlines in the original picture will also be cut.

As another example, consider the knitting pattern in Fig. 8a, which is expressed in *PictureBlocks* by composing patterns created by M. C. Escher [29]. Suppose we want to cut out the blank areas between the pieces of yarn, but where two pieces of yarn cross, we want to engrave the black border of the yarn to indicate which piece of yarn is on top. *PictureBlocks* provides an engrave-lines-and-cut mode that is similar to the engrave-and-cut mode but (1) ignores the colors of areas and (2) pays attention to the colors of existing lines (Fig. 8b).



a. *PictureBlocks* picture. b. Laser-cut artifact

Fig. 8. Engrave-lines-and-cut example with Escher's knitting pattern.

D. Pragmatics

In the near future, many fabrication devices may become inexpensive enough to be commonly owned by individuals and used at home [3]. Currently, however, laser cutters are rather expensive, usually costing tens of thousands of US dollars, though there are now models less than \$10K. But schools with engineering or architecture programs may already have laser cutters they are willing to share with other departments. An alternative is to send jobs out to laser cutter services, but the turnaround time interferes with iterative experimentation.

In contrast, vinyl cutters are much more affordable. Wellesley has a low-end desktop version that costs just a few hundred US dollars and uses relatively inexpensive supplies (cardstock and vinyl sheets). Inexpensive desktop 3D printers are now available [3], and we plan to explore using 3D printers as an alternative to laser cutters for producing artifacts in *TurtleBlocks* and *PictureBlocks*. It is worth noting that desktop vinyl cutters and 3D printers are the basis of the Fab@School project for introducing elementary school children to engineering design via personal fabrication ([30], [31]).

III. Language Design

Here we describe the design of *TurtleBlocks* and *PictureBlocks*, emphasizing novel and unusual aspects.

A. Blocks Framework

Both *TurtleBlocks* and *PictureBlocks* are implemented using the *OpenBlocks* framework ([32], [33]), which also underlies StarLogo TNG and App Inventor, giving all four languages a similar look and feel. Program constructs are represented as jigsaw-like blocks that are organized into drawers of related blocks. Users drag blocks from these drawers into a workspace and snap them together to form program fragments.

Expressions (program fragments that denote values) are represented as blocks whose left side has a plug whose shape indicates the type of the value. For example, in Fig. 1a, the blocks with angled sides are numerical expressions, while the rounded sides of the "$>$" (greater-than) block indicate that it is a boolean expression. Some expressions have argument sockets whose shape indicates the type of the argument. For example, the "$\div$" (division) block is an operator that takes two number operands and produces a number as a result.

Commands (program fragments that perform actions) are represented as blocks with a notch at the top and a bump at the bottom. These naturally compose vertically to create so-called "stacks" — sequences of commands that are performed in order from the top down. All the commands in Fig. 1a take one or more expressions as arguments. Control commands, such as `if` and `repeat`, also take one or more command substacks for code that may be executed as part of the command.

Procedure, function, and global variable declarations are specified by top-level blocks that cannot be nested in other blocks. In Fig. 1a, the tall `sierpinski` block declares a two-argument procedure that is invoked by the two shorter `sierpinski` blocks.

TurtleBlocks and *PictureBlocks* inherit other important usability features from *OpenBlocks*, including: the ability to copy and paste collections of blocks in the workspace; a so-called "type blocking" features that allows users to type the prefix of a block name and select a block from a list of blocks matching the prefix rather than selecting the block from a drawer; and the ability to save blocks programs to and load blocks programs from the file system.

B. Connector shapes and static vs. dynamic typing

Types in programming languages describe collections of related values, such as numbers, booleans, and lists; each type is associated with a set of operations on the values of that type. In a *dynamically typed* language, such as Python or JavaScript, each value carries type information that is used to determine the validity of operations at run time (e.g., division can be applied only to two numbers). In a *statically typed* language, like Ada or ML, the compiler can determine the type of each variable and expression (often aided by user type declarations) and the validity of operations at compile time. Many languages lie between these two extremes, such as Java, which does much type checking at compile time, but leaves certain type checks for run time. There are other language design choices related to typing, such as whether conversions from one type of value to another are done implicitly by the language or explicitly by the programmer [34].

In statically typed blocks languages, types can be represented by connector shapes, which visually indicate which block connections are valid. Socket shapes specify the allowed types of inputs to a block and the plug shape indicates the type of the value denoted by an expression. For example, in *TurtleBlocks* and *PictureBlocks*, angled connector shapes denote numbers and rounded connector shapes denote booleans. For dynamically typed languages, such as App Inventor, it is natural for all sockets and plugs to have a single shape for all values, indicating that all combinations of blocks are syntactically legal (and type errors are caught only at run time).

However, in practice, many dynamically typed blocks languages use connector shapes in ad hoc and potentially confusing ways [35]. E.g., Scratch is a dynamically typed language with two plug shapes: an angled shape for booleans and a rounded shape for numbers and strings. There is a third input shape (rectangular) that accepts any shape of value. Fig. 9 shows Scratch expressions for adding two numbers (a), comparing two numbers (b), and concatenating two strings (c).

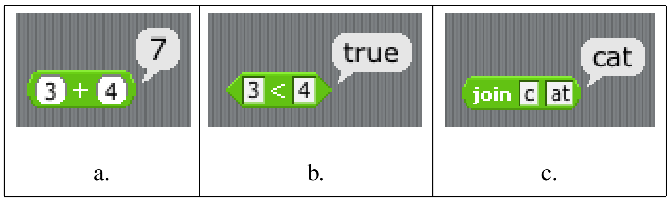


Fig. 9. Simple examples of Scratch shapes/types.

Block shapes in Scratch prevent simple type errors, such as using the result of an addition operator (a number) as the test expression of a conditional (which must have a boolean shape). But because numbers and strings share the same shape and there are operators (like `join`) whose arguments may have any shape, it is possible to compose blocks in nonsensical ways. Scratch handles these situations in a *failsoft* way by automatically converting values to avoid error messages [20]. For example, strings used in a numeric context are converted to 0, and numbers and booleans used in a string context are converted to strings (Fig. 10).

Scratch's design also makes it hard to express certain computational patterns. For instance, it is possible to store a boolean into a variable, but since all variable references have rounded shape, it is not possible to use this variable value directly in the test expression of a conditional. Instead, it is necessary to compare this variable value (which is automatically converted to a string) to the string `true`. (Fig. 11)

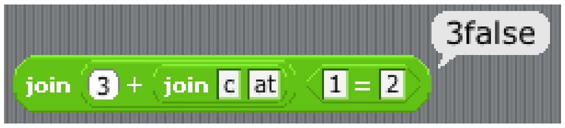


Fig. 10. Scratch converts value types to avoid dynamic type errors.

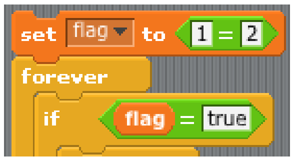


Fig. 11. Portion of Scratch example involving a boolean variable.

TurtleBlocks and *PictureBlocks* avoid these problems by employing a static type discipline in which each distinct kind of value is represented by a different shape. *TurtleBlocks* has distinct shapes for numbers, booleans, strings, colors, arc types, and pen corner types(see the discussion in section II-A). *PictureBlocks* additionally has a shape for picture values.

Surprisingly, connector shapes that consistently express static types are relatively rare in blocks languages. It is found mostly in blocks languages with a very small number of types. For example, TurtleArt [27], PicoBlocks [36], and ModKit [37] have only two types (integer and boolean) and use connector shapes to distinguish these types. But there are type-related limitations and/or inconsistencies in these languages. For example:

- In TurtleArt and PicoBlocks, procedure blocks with parameters can be defined only in the text language, not the blocks language, their parameters can have only integer types, and their output type (if any) must be an integer. It is possible to define procedure blocks with output type boolean, but they cannot take any parameters.
- ModKit doesn't have procedures, and it uses integer sockets with drop-down menus to represent special-purpose values like motor identifier and direction (Fig. 12a). But these sockets can be filled with arbitrary integer expressions, whose meaning is completely unclear (Fig. 12b).

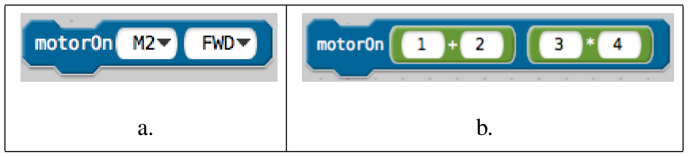


Fig. 12. ModKit `motorOn` block.

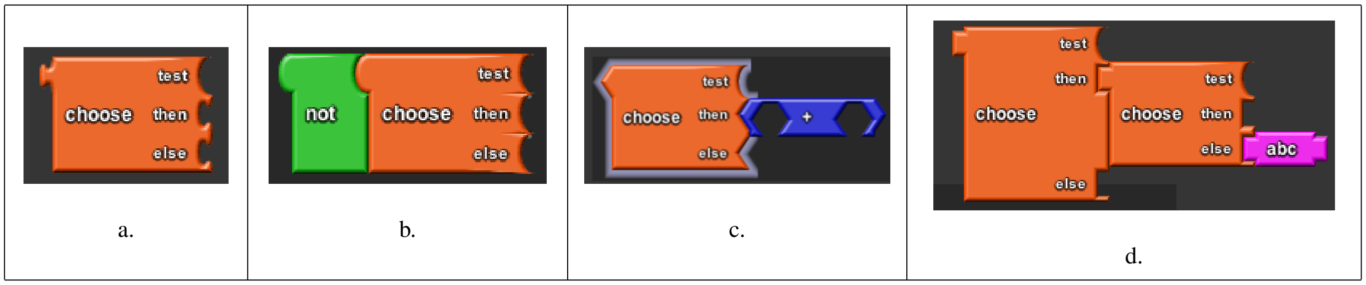


Fig. 13. The *TurtleBlocks*/*PictureBlocks* `choose` block illustrates polymorphism.

StarLogo TNG ([23], [24]) is the only other well-known blocks language that uses shapes to consistently distinguish types in a full-featured language with global variables and procedures. It has separate shapes for six types (integer, boolean, string, integer list, boolean list, and string list) and allows global variables, procedure parameters, and procedure results to have any of these types. A way to express more complex types (arbitrary list, pair, and function types) via connector shapes is described in [35].

C. Polymorphism

Representing each type by a different shape can lead to an explosion of other kinds of blocks. Consider a `chooseNum` block that takes three inputs (a boolean test expression and two numerical expressions) and produces a number output. If the test expression evaluates to true, it returns the value of the first number expression, otherwise it returns the value of the second one. The problem is that a `choose`*X* block is needed for every type *X*, so drawers will become cluttered with copies of essentially the same block that differ only in their types.

Statically typed programming languages like ML solve this problem via *universal polymorphism*, in which universally quantified type variables can be instantiated to any type [34]. In blocks languages, we can represent such a type variable using a connector with a particular shape. E.g., the `choose` block in *TurtleBlocks* and *PictureBlocks* (Fig. 13a) has poly (i.e., polymorphic-shaped) sockets for the `then` and `else` sockets and a poly plug for its result. If the usage context for this block constrains any one of these three connectors, the others change accordingly. If the output of `choose` is constrained to have boolean shape, the `then` and `else` sockets change to the boolean shape (Fig. 13b). Or if the `then` socket is filled with a number plug, the `else` socket and output plug change to the number shape (Fig. 13c). Such changes can propagate through multiple `choose` blocks (Fig. 13d).

Poly shapes can also be used to address *ad hoc polymorphism* (also known as *operator overloading*) in which it is sensible to apply the same operator to multiple types. Standard examples include comparing values of the same type via the operators $<$, $=$, and $>$, or a `toString` function that converts a value to a string. E.g., the equality operator in *TurtleBlocks* and *PictureBlocks* has two poly sockets and constraining one fixes the other to be the same.

There have been several approaches to expressing polymorphic types in non-blocks-based visual programming languages (e.g., [38]–[40]). Using poly shapes to express polymorphism in blocks languages was pioneered in StarLogo TNG, which uses universal polymorphism for list operations and `switch` and `output` statements, and ad hoc polymorphism for comparing values and converting values to strings. *TurtleBlocks* and *PictureBlocks* extend polymorphism to variable references (see Sec. III-D) and can distinguish different polymorphic types on the same block (akin to ML's polymorphic tick types). These are novel features that StarLogo TNG does not support.

D. Naming

The ability to refer to values named by procedure parameters or local variables is a critical feature that most blocks languages handle poorly. Many blocks languages, such as Scratch and ModKit [37], don't even provide procedures (though Scratch 2.0 will support procedures with parameters [41]), and some languages provide them only in limited ways (see the discussion of TurtleArt and PicoBlocks procedures in Sec. III-B). In App Inventor and StarLogo TNG, there are procedure declaration blocks to which special formal parameter blocks can be added; separate blocks for referencing the value of each parameter are added to a special *My Blocks* drawer, which quickly becomes cluttered. Even worse, this design requires that all formal parameter names be distinct, so the same parameter name cannot be used in two different procedures. This violates a fundamental locality principle of name scope and gives novice programmers the mistaken impression that procedure parameters are globally defined. Indeed, nothing prevents an attempt to use a reference to a procedure parameter outside the scope of the procedure body (e.g., Fig. 14, in which the parameter `x` of `forwardProc` is used in `leftProc`). This is an error that blocks languages should prevent, not encourage!

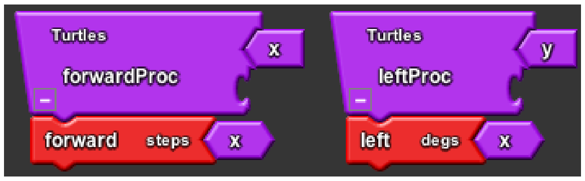
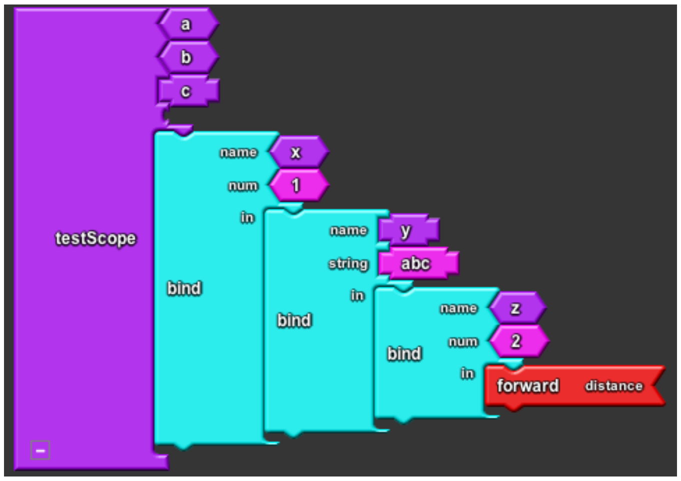
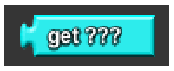


Fig. 14. Out-of-scope procedure parameter reference in StarLogoTNG.

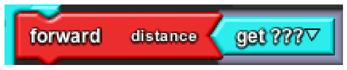
Local variables are even more problematic in blocks languages. Very few blocks languages support



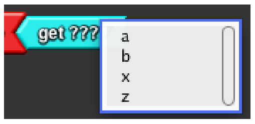
a. A *TurtleBlocks* procedure with local variable bindings in its body.



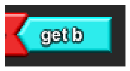
b. A polymorphic variable reference ("getter") block.



c. The getter block plugged into a number socket.



d. Menu of variables of type number in scope at getter.



e. Getter block after variable b has been chosen.

Fig. 15. An illustration of variable scope in *TurtleBlocks*.

the block-structured naming that is common in traditional languages. The only blocks languages we are aware of with local naming features are TaleBlazer and WebLogo, both currently under development [42].

TurtleBlocks and *PictureBlocks* support both procedures with parameters and block-structured local variables. Procedures with formal parameters are declared as in App Inventor and StarLogo TNG, and `bind` command and expression blocks associate a name with a value in a local scope. All procedure parameters and local variables are referenced and set via single polymorphic getter and setter blocks with drop-down menus that list all names of the given type in scope at the position of the getter or setter block.

Fig. 15 illustrates how parameter and local variable names are handled in *TurtleBlocks* (*PictureBlocks* is similar). Fig. 15a shows the declaration of a procedure `testScope` that takes two number parameters named `a` and `b` and a string parameter named `c`. The body of `testScope` has three nested `bind` blocks that bind the number name `x` to 1, the string name `y` to `abc`, and the number name `z` to 2. The body of the innermost `bind` is a `forward` command that is missing its argument. In order to specify a reference to one of the number names as the argument to `forward`, we drag a variable getter block (Fig. 15b) from the appropriate drawer and plug it into the `forward` socket (Fig. 15c). The getter's polymorphic plug becomes the number shape. The "`???`" indicates that the name being referenced has not yet been determined. A drop-down menu on the getter block lists all the names of the getter's output type (in this case, number) that are in scope at the position of the getter block (Fig. 15d). Any one of these can be selected (Fig. 15e).

Any change to a formal parameter name propagates to all getters and setters for that variable. Whenever variable getter and setter blocks are copied or moved, their name reverts to "`???`" if the original variable is not bound in the new context.

This approach to name references is a significant improvement to existing approaches in blocks languages. In contrast with App Inventor and StarLogo TNG, our approach (1) reduces drawer clutter (there is only one variable getter and setter block, rather than one per variable), (2) prevents the out-of-scope parameter references illustrated in Fig. 14, and (3) presents a block-structured view of naming in which the same name can be used in different procedures (or even in nested naming blocks) — an important idea in computational thinking. Scratch 2.0 [41] solves the procedure parameter problem by allowing variable references to be cloned from the parameter declaration, but does not support local variables. In terms of naming, BYOB [43] is similar to Scratch 2.0, but (1) can refer to "holes" in anonymous procedure bodies by position rather than name and (2) can simulate local naming by calling local anonymous procedures. The notions of block-structured local naming constructs and drop-down menus for names that are in scope is being independently investigated in WebLogo [42].

IV. User Feedback

We held four 70-minute *TurtleBlocks* workshops attended by a total of 85 students in Wellesley's CS1 course [16]. They were already familiar with Java versions of the turtle and picture worlds. For *PictureBlocks*, we held two 70-minute workshops attended by a total of 40 CS1 students. We also held a 2-hour laser-cutting workshop featuring *TurtleBlocks* attended by 8 students, 6 of whom had no prior programming experience. All workshops were held with earlier versions of *TurtleBlocks* and *PictureBlocks* that did not have the polymorphism or naming features described in Sec. III-C and Sec. III-D.

After the workshops, we asked students to complete a short online survey about the tools and experience. All 8 of the non-CS1 students completed the survey, but only 29 of the 125 CS1 students completed the survey. We learned the following:

- Almost all (35 of 37) respondents indicated that they were more motivated to create designs when they could get tangible output from the turtle and picture worlds as opposed to just drawings on the screen. They loved the craft-like nature of the activity and the fact that they could show their creations to their friends and families.
- Many students wanted a way to sketch designs (or at least components of designs) rather than having to create all designs from turtle or picture primitives. In response to this feedback, we have implemented a simple sketching application that allows drawing lines, rectangles, ellipses, and polygons. We have integrated this tool into *PictureBlocks* and plan to integrate it into *TurtleBlocks*. We are eager to see if students use the tool *instead of* or *in addition to* the computational facilities provided by the languages. We would like to experiment with more sophisticated sketching tools, like those in [4], [44].
- A sizable minority of the CS1 students indicated they preferred to write their turtle/picture programs in Java rather than in the blocks languages. Many said it was tedious to assemble programs with blocks. If we want more experienced programmers to use blocks environments, we need to provide automatic conversions between text and blocks languages (in both directions). This would also provide a natural learning path from blocks languages to text languages.
- A few students mentioned the difficulty of designing robust, connected structures. Some laser-cut artifacts were fragile due to thin parts. A tool to highlight potentially weak structural parts of a design would be useful.

We plan to conduct further user studies to evaluate the effectiveness of our new features: sketching, polymorphism, and naming. We also plan to develop blocks language environments for designing artifacts for 3D printers, starting with a 3D turtle language inspired by Gross's FormWriter [45].

Acknowledgments

We gratefully acknowledge Chelsea Hoover, who implemented the first version of *TurtleBlocks*; Nichole Burton, who worked on an early version of *TurtleBlocks*; Marie Vasek, who informed us about the problems with types in blocks languages; Michael J. Dawson, who provided extensive feedback on drafts of this paper; and Mike Eisenberg and Orit Shaer, who provided advice and support. We also thank the anonymous reviewers for their suggestions for improving this paper. This work was supported by Wellesley College faculty awards grants and a Brachman Hoffman Small Grant.

REFERENCES

[1] F. Turbak, S. Sandu, O. Kotsopoulous, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in *IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC '12)*, Oct. 2012, to appear.

[2] N. Gershenfeld, *Fab: The Coming Revolution on your Desktop — From Personal Computers to Personal Fabrication*. MIT Press, 2005.

[3] H. Lipson and M. Kurman, "Factory@home: The emerging economy of personal fabrication," 2010, report Commissioned by the Whitehouse Office of Science & Technology Policy.

[4] Y. Oh, G. Johnson, M. D. Gross, and E. Y.-L. Do, "The Designosaur and the Furniture Factory: Simple software for fast fabrication," in *2nd Int. Conf. on Design Computing and Cognition (DCC06)*, 2006.

[5] K. D. Willis, J. Lin, J. Mitani, and T. Igarashi, "Spatial sketch: bridging between movement & fabrication," in *4th Int. Conf. on Tangible, Embedded, and Embodied Interaction (TEI '10)*, 2010, pp. 5–12.

[6] K. D. Willis, C. Xu, K.-J. Wu, G. Levin, and M. D. Gross, "Interactive fabrication: new interfaces for digital fabrication," in *5th Int. Conf. on Tangible, Embedded, and Embodied Interaction (TEI '11)*, 2011, pp. 69–72.

[7] J. Wing, "Computational thinking," *Comm. of the ACM*, vol. 49, no. 3, Mar. 2006.

[8] M. Eisenberg, N. Elumeze, L. Buechley, G. Blauvelt, S. Hendrix, and A. Eisenberg, "The homespun museum: Computers, fabrication, and the design of personalized exhibits," in *Conf. on Creativity & Cognition (C&C'05)*, 2005, pp. 13–21.

[9] M. Eisenberg, A. Eisenberg, L. Buechley, and N. Elumeze, "Computers and physical construction: Blending fabrication into computer science education," in *Int. Conf. on Frontiers in Education: Computer Science & Computer Engineering (FECS '08)*, 2008, pp. 127–133.

[10] G. Johnson, "FlatCAD and FlatLang: Kits by code," in *IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC '08)*, 2008, pp. 117–120.

[11] S. Papert, *Mindstorm: Children, Computers, and Powerful Ideas*. Basic Books, 1980.

[12] H. Abelson and A. diSessa, *Turtle Geometry: the Computer as a Medium for Exploring Mathematics*. MIT Press, 1981.

[13] P. Henderson, "Functional geometry," in *ACM Symposium on Lisp and Functional Programming*, 1982, pp. 179–187.

[14] ——, "Functional geometry," *Higher Order and Symbolic Computation*, vol. 15, no. 4, pp. 349–365, 2002. This is a revised version of [13].

[15] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs (2nd ed.)*. MIT Press, 1996.

[16] CS111 Introduction to Programming and Problem Solving, Wellesley College introductory computer science course. http://cs.wellesley.edu/~cs111, accessed Mar. 11, 2012.

[17] E. P. Glinert, "Towards "second generation" interactive, graphical programming environments," in *2nd IEEE Computer Society Workshop on Visual Languages*, 1986, pp. 61–70.

[18] Scratch project, MIT Lifelong Kindergarten Group, http://scratch.mit.edu/, accessed Mar. 11, 2012.

[19] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Comm. of the ACM*, vol. 52, no. 11, Nov. 2009.

[20] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, Nov. 2010.

[21] App Inventor home page, MIT Center for Mobile Learning, http://appinventor.mit.edu, accessed Mar. 11, 2012.

[22] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor*. O'Reilly Media, Inc., Apr. 2011.

[23] StarLogo TNG project, MIT Scheller Teacher Education Program, http://education.mit.edu/projects/starlogo-tng, accessed Mar. 11, 2012.

[24] E. Klopfer, H. Scheintaub, W. Huang, and D. Wendel, “Starlogo TNG: Making agent-based modeling accessible and appealing to novices,” in *Artificial Life Models in Software (2nd edition)*, M. Komosinski and A. Adamatzky, Eds. Springer, 2009.

[25] A. Begel, “Logoblocks: A graphical programming language for interacting with the world,” Mar. 1996, MIT Advanced Undergraduate Project.

[26] ——, personal communication, Mar. 10, 2012.

[27] TurtleArt home page, http://turtleart.org/, accessed Mar. 13, 2012.

[28] F. Turbak, S. Sandu, O. Kotsopoulous, E. Erdman, E. Davis, K. Chadha, and J. Okerlund, “Blocks languages for creating tangible artifacts,” Wellesley College, Tech. Rep., Jul. 2012, TinkerBlocks TR 2012-1, available at http://www.tinkerblocks.org/pubs.

[29] D. Schattschneider, *M. C. Escher: Visions of Symmetry*. W. H. Freeman and Company, 1990.

[30] G. Bull, C. Maddox, G. Marx, A. McAnear, D. Schmidt, L. Schrum, S. Smaldino, M. Spector, D. Sprague, and A. Thompson, “Educational implications of the digital fabrication revolution,” *Journal of Research on Technology in Education*, vol. 42, no. 4, pp. 331–338, Jun. 2010.

[31] *Imagine. Design. Create. Construct.*, Society for Information Technology & Teacher Education (SITE), Fab@School video, http://maketolearn.org/explore/videos/imagine-design-create-construct/, accessed Jul. 1, 2012.

[32] R. Roque, “Openblocks: An extendable framework for graphical block programming systems,” Master’s thesis, MIT, May 2007.

[33] OpenBlocks home page, MIT Scheller Teacher Education Program, http://education.mit.edu/openblocks, accessed Mar. 13, 2012.

[34] F. Turbak, D. K. Gifford, and M. Sheldon, *Design Concepts in Programming Languges*. MIT Press, 2008.

[35] M. Vasek, “Representing expressive types in blocks programming languages,” undergraduate thesis, Wellesley College, May, 2012. Available at http://www.tinkerblocks.org/pubs.

[36] The Playful Invention Company, PicoCricket Reference Guide, version 1.2a, http://www.picocricket.com/pdfs/Reference_Guide_V1_2a.pdf, accessed Mar. 22, 2012.

[37] ModKit home page, http://www.modk.it, accessed Mar. 22, 2012.

[38] M. A. Najork and E. Golin, “Enhancing Show-and-Tell with a polymorphic type system and higher-order functions,” in *IEEE Workshop on Visual Languages*, 1990, pp. 215–220.

[39] M.-A. Najork, “Programming in three dimensions,” Ph.D. dissertation, University of Illinois, Urbana-Champaign, 1994.

[40] R. W. Djang, M. M. Burnett, and R. D. Chen, “Static type inference for a first-order declarative visual programming language with inheritance,” *Journal of Vis. Languages and Computing*, vol. 11, pp. 191–235, 2000.

[41] J. Maloney, personal communication, Mar. 13, 2012.

[42] D. Wendel and P. Medlock-Walton, personal communication, Mar. 12, 2012.

[43] B. Harvey and J. Mönig, “BYOB 3.1 reference manual,” http://byob.berkeley.edu/BYOBManual.pdf, accessed Jul. 1, 2012.

[44] G. Johnson, “Sketch It, Make It (SIMI),” videos of prototype sketching environment for laser-cutterable artifacts, http://sketchitmakeit.com/, accessed Jul. 1, 2012.

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