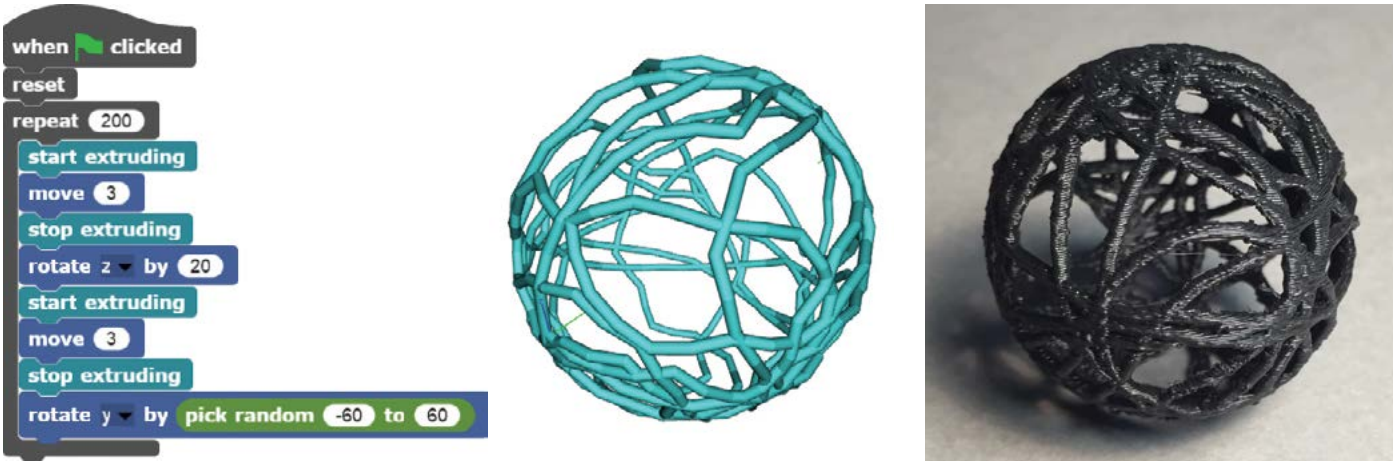
Beetle Blocks

A New Visual Language for Designers and Makers

Duks Koschitz, PhD
Pratt Institute

Bernat Ramagosa
Arduino, Spain

Eric Rosenbaum, PhD
Massachusetts Institute of Technology



1

ABSTRACT

We are introducing a new teaching tool to show designers, architects, and artists procedural ways of constructing objects and space. Computational algorithms have been used in design for quite some time, but not all tools are very accessible to novice programmers, especially undergraduate students. 'Beetle Blocks' (beetleblocks.com) is a software environment that combines an easy-to-use graphical programming language with a generative model for 3D space, drawing on 'turtle geometry,' a geometry paradigm introduced by Abelson and Disessa, that uses a relative as opposed to an absolute coordinate system. With Beetle Blocks, designers are able to learn computational concepts and use them for their designs with more ease, as individual computational steps are made visually explicit. The beetle, the relative coordinate system, follows instructions as it moves about in 3D space.

Anecdotal evidence from studio teaching in undergraduate programs shows that despite the early introduction of digital media and tools, architecture students still struggle with learning formal languages today. Beetle Blocks can significantly simplify the teaching of complex geometric ideas and we explain how this can be achieved via several examples. The blocks-based programming language can also be used to teach fundamental concepts of manufacturing and digital fabrication and we elucidate in this paper which possibilities are conducive for 2D and 3D designs. This project was previously implemented in other languages such as Flash, Processing and Scratch, but is now developed on top of Berkeley's 'Snap!'

1 Example of:
– a stack of blocks (the program)
– the 3D rendered object
– a 3D print

INTRODUCTION

The audience for Beetle Blocks is comprised of architects, designers, and artists, who have no prior knowledge of programming and wish to learn how to use computational concepts as part of their design process. The courses and workshops that have been taught with Beetle Blocks in preparation for this paper have taken place in undergraduate departments of architecture and design.

Programming is not yet fully integrated in art and architecture programs as a foundational skill and students often find that the learning curve is steep. A graphical language that is easy to learn is a first step in making programming more accessible (Kelleher and Pausch 2005). Beetle Blocks is based on Scratch (Maloney et al. 2010) and now implemented on top of Snap!, both graphical control flow languages. This approach to programming requires the user to formulate every explicit step of an algorithm in a visual way, without the frustrating syntactic pitfalls users experience with typical text-based programming languages. Unlike other graphical programming paradigms, it also positions the fundamental concepts of computation early in the learning process, thus making it more likely that learners will build on these concepts and move on to tasks of greater complexity. The goal is to get to the teaching of computational ideas very quickly in a curriculum and to not be bound by commercial software environments that are more suitable for the production of design projects.

The 3D environment of Beetle Blocks is intuitive because it is modeled after 'turtle geometry' (Papert 1993). Turtle geometry uses a relative rather than an absolute coordinate system, allowing learners to relate their knowledge of body movements to the movements of the computer's representation of the turtle. The combination of a graphical control flow language and turtle geometry is not new, but we believe that the specific deployment of our software, together with fundamental lessons in geometry and digital fabrication, can make a difference in teaching programming early in design education.

Thinking about the construction of space procedurally is a powerful alternative to analog thinking and we believe that both approaches should be taught in architecture programs. Design is often judged on its coherence and how well a formal concept is thought through. Using procedures to construct a spatial constellation requires rigor in the design process and that in turn assists designers with a way to create coherent designs. Another goal consists of teaching how to create machine instructions for digital fabrication, which we address visually via the use of a control flow language.

There is no doubt that relational or parametric modeling is powerful for architectural design and we believe that a deeper understanding of how to control geometric relationships in a procedural way will become helpful for designers. By learning how to create procedures, we hope that designers will acquire an understanding of how computation can be used for a design project. After learning how to design with Beetle Blocks, students and users in general can transition to commercial software to realize complex projects, where they can apply the acquired knowledge.

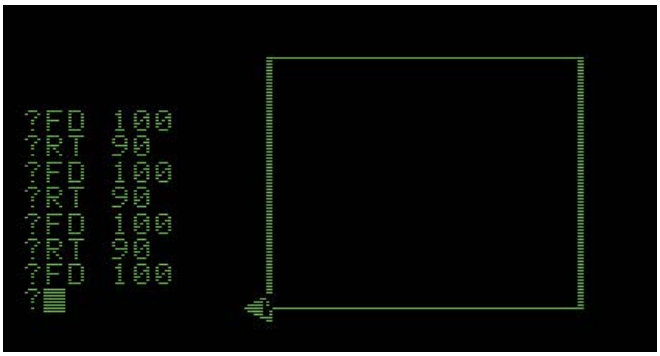
THE GEOMETRY PARADIGM

In order to teach computational steps expediently, we decided to link a visual graphical language to a 3D environment via turtle geometry. The local coordinate system can be controlled step-by-step to move about in 3D.

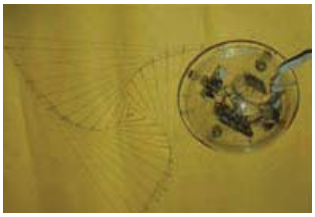
Drawing With a Turtle

In the Logo programming language for children (Papert 1993), the turtle is an object on the screen that is moved around with commands for moves and turns such as 'forward 10' and 'right 90.' Turtle geometry relies on relative movements as opposed to absolute coordinates common on CAD software. Relative movement is intuitive, because we can draw on knowledge of our own bodies' movement in space. This identification between the turtle and one's own body is a form of 'body syntonicity,' which Papert determined to be crucial for learning. It is easy to forget that the idea of an absolute coordinate system (or any invariant) is something that must be constructed or acquired, which we generally do when we are very young or learn CAD software for the first time. We use our intuitive knowledge of a relative coordinate system to construct the idea of an absolute system. Local instruction sets are also concise and easy to read. Relative coordinate systems are therefore a good starting point for learning geometry. In Logo, turtle movement was restricted to a plane (Figure 2) and Papert built a physical corollary in the form of a tethered floor roamer that could drop a pen and draw as it moved (Figure 3).

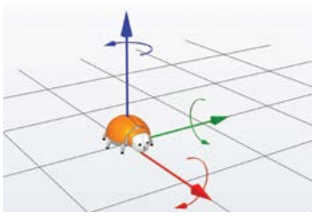
Abelson and diSessa elaborated on turtle geometry, exploring the possibilities of turtle movement in 3D space in mathematics (Abelson and diSessa 1986). They invented many algorithms that relate continuous geometry to turtle geometry. There have been various implementations of Logo that enable turtle drawing in 3D (Petts 1988), such as Elica (www.elica.net) and Logo3D (logo3d.sourceforge.net), but none of these use a graphical language. StarLogo TNG (education.mit.edu/starlogo-tng) (Colella, Klopfer, and Resnick 2001) has both a graphical language and a 3D environment, but it is designed for creating games and simulations, not for exploring geometry. It allows one to move the turtle up and down, but one cannot rotate it out of the plane for example.



2



3



4

2 Logo instructions for a square (rectangular image due to CRT screen).
3 Floor roamer drawing on paper (Seymour Papert).
4 Positive directions and positive Beetle rotations.

We refer to the turtle as 'beetle' and use conventions of positive relative movements and turns known in CAD software. Red, green, and blue indicate the main directions in x, y, and z (Figure 4).

The Necessity for a Discrete Approach

In mathematics and geometry, shapes are generally described in continuous ways and eighteenth-century mathematics have allowed us to understand the world as a construction of smooth functions. When dealing with computational systems, we are typically bound by the capabilities of machines with discrete registers or switches.

The example of a circle helps to illustrate the difference between continuous and discrete representations. A circle is a simple shape of Euclidean geometry consisting of those points in a plane, which are equidistant (r) from a given point, the center (c). There are several representations that invoke the mental image of a circle, for example a circle drawn on paper, or the mathematical definition for its circumference $2r\pi$. In discrete steps we can describe a circle by moving forward 1 unit and turning one degree to the right 360 times.

All three representations have deep relations to the way we teach design: some artistic, some engineering-oriented, and some maybe more scientific. It is important that designers become acquainted with all of them and can control the relevant methods to express what they want to design. We will focus on the discrete approach, as it allows us to break down geometric constructs into individual steps and these steps are conducive to writing code. Moving the local coordinate system through the use of many steps can be thought of as writing algorithms for geometry.

DESIGNING A PROGRAMMING LANGUAGE

In this section, we elucidate our decisions as they pertain to the type of formal language we wanted to create. As the goal is to teach computational ideas quickly, we are faced with questions about syntax, visual feedback, and occluding information that might be unnecessary.

Control Flow Versus Data Flow

Control-flow languages have an execution model in which commands are run in a sequence. There is generally a single program counter that points to the next command to be run. The movement of this counter is the flow of control. In data-flow languages, by contrast, all parts of a program may be executing simultaneously. Data flows from one component to another. Control-flow languages are typically represented as lists of instructions, while data-flow languages are typically represented as nodes connected by lines. Data flow is a popular paradigm for graphical languages because its topological structure is naturally represented as a graph.

Beetle Blocks uses the control-flow execution model, because our emphasis is on making programming understandable. The control-flow model allows us to show the execution of a program explicitly, making it easier for users to see what the program is doing at every single step (Figure 5).

Black Boxes

One challenge in the design of graphical languages for learning is exactly where to put the 'black boxes.' By black box we mean a primitive unit that cannot be opened up in order to see what is inside and learn how it works. All programming languages above the level of assembly code are structured hierarchically, with each command encapsulating a set of other commands, until the language bottoms out at its primitives, which are its black boxes.

For example, a programming language might have a command for drawing a circle. This command might encapsulate lower level commands within the very same language for geometric calculations that the user can access. However, these lower level commands might encapsulate primitives, such as commands for rendering pixels on the screen that are not accessible to the user. The primitives, written in a lower level language, are thus black boxes.

The design question of where to put the black boxes is really the question of which concepts users should have access to, and which should remain hidden. A language designed for learning

5

should consist of a set of commands that combine accessibility and flexibility, and hide the lower-level commands that provide an irrelevant level of detail or are not sufficiently intuitive (Papert 1993).

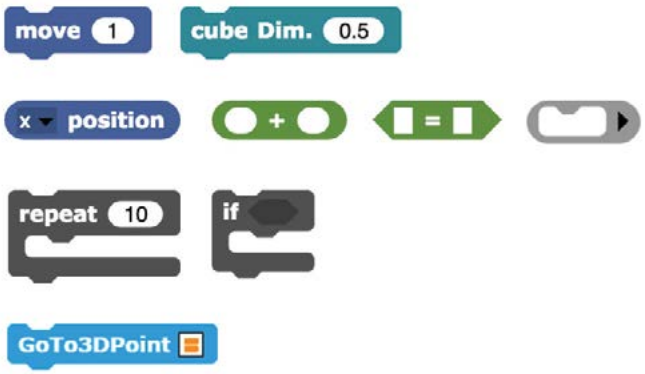
In Beetle Blocks, the commands are for manipulating the Beetle's properties (such as its position and rotation), generating a set of basic shapes, and controlling the flow of the program. We chose this set of commands because it makes simple tasks such as drawing a shape very easy and immediate, while also making available a broad landscape of expressive algorithmic possibilities. All the low-level processes that render shapes to the screen, including their shading, textures, and lighting, are hidden from the user.

Expanding the Scratch paradigm

Beetle Blocks makes use of several of the design principles developed for the Scratch programming language (Resnick et al. 2009). These include immediate feedback, commands as visual building blocks, and error prevention. The choice for Snap! is motivated by its expanded functionality and Javascript basis. The language draws heavily from Mitchell Resnick's Scratch project and has been created in collaboration with Brian Harvey, Jens Moenig, and Bernat Ramagosa. A salient feature of Beetle Blocks is its Lambda functionality (Harvey and Mönig 2010), a specific way to express complex computational ideas, which is particularly useful in a pedagogical context.

Clicking on a block has an immediate effect in Beetle Blocks, and most blocks have a visible result. This encourages a tinkering process through which users can learn what blocks do simply by trying them out. Clicking on a stack of blocks causes it to run immediately, which makes it easy to rapidly test different possibilities without waiting for the compilation step required by many programming languages.

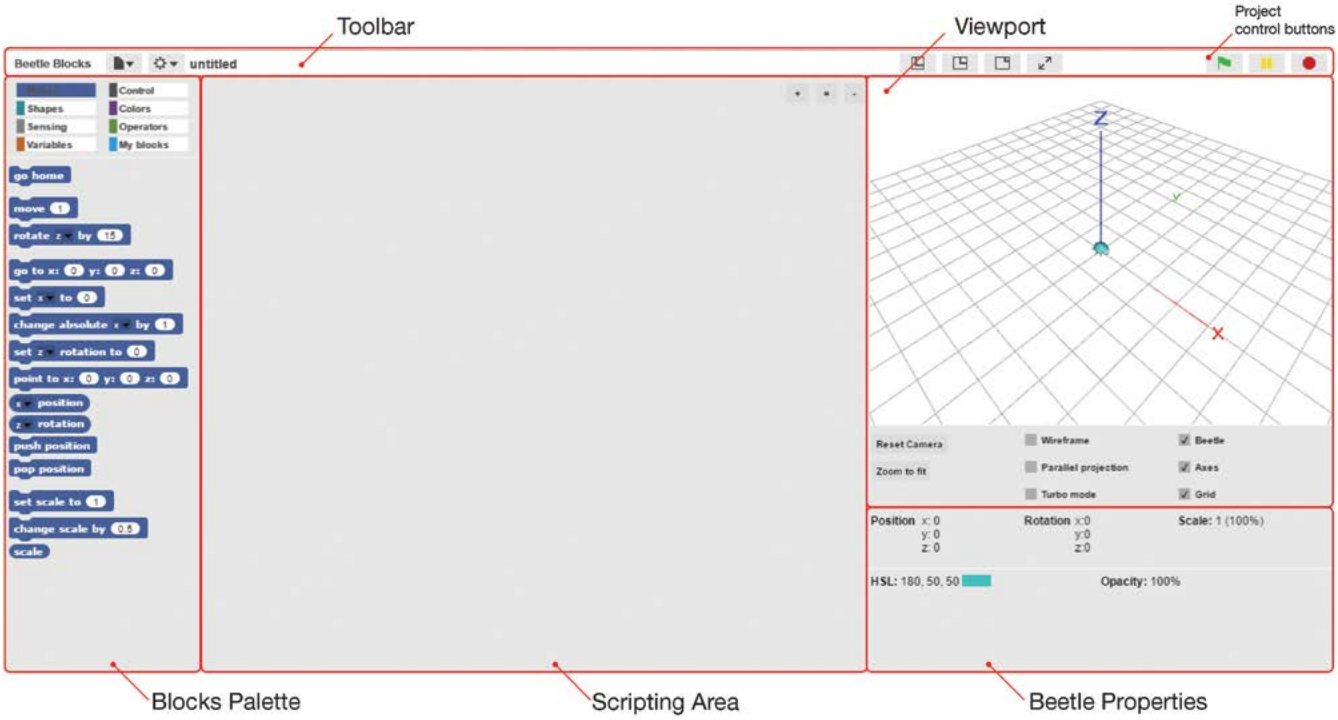
The blocks themselves have connectors at the top and bottom, making it obvious that they snap together in a stack. This visual affordance makes it clear that the basic interaction in Beetle Blocks is stacking together blocks into a program (Figure 6). Beetle Blocks has several properties that prevent frustrating errors. The blocks have certain shapes that only allow them to connect in ways that make syntactic sense (i.e., the stack of blocks will always do something, even if it is not what was intended). There is nothing like a 'compiler error' at all in Beetle Blocks, and no error messages are necessary. Beetle Blocks also uses a 'fail soft' principle. This means that all parameters in blocks have sensible default values, and will do something sensible even if the parameters are left empty. These properties eliminate many of the frustrations that can be the main reason for novices to give up learning to program.



6

5 The spiral is an effect of using a scaling function on moves and turns.

6 Selection of blocks (movement, shape; position, check for equal value, addition, transform procedure to be used as input; repeat block; user defined block).



7 The new UI.

The User Interface

The new version of Beetle Blocks that is based on Snap! uses a user interface (UI) that will seem familiar to Scratch and Snap! users. The Toolbar at the top has two menus for file management and settings. On the right, we kept the control buttons that allow users to run or stop a program (Figure 7).

The user can drag and drop blocks from the Blocks Palette on the left into the scripting area and can see the 3D model in a Viewport on the top right. The bottom right area allows users to modify properties of the Beetle and the Viewport, and also displays information about the current state of the Beetle.

8 Selection of blocks (movement, shape; position, chec

Regarding the color scheme for the blocks, we decided to keep the color coding for movement, variables, and operators. The control blocks are grey and custom procedures are in a lighter blue (Figure 8). Snap! allows us to use many additional features that are available beyond the functionality of Scratch, such as advanced list manipulation and lambda functionality.

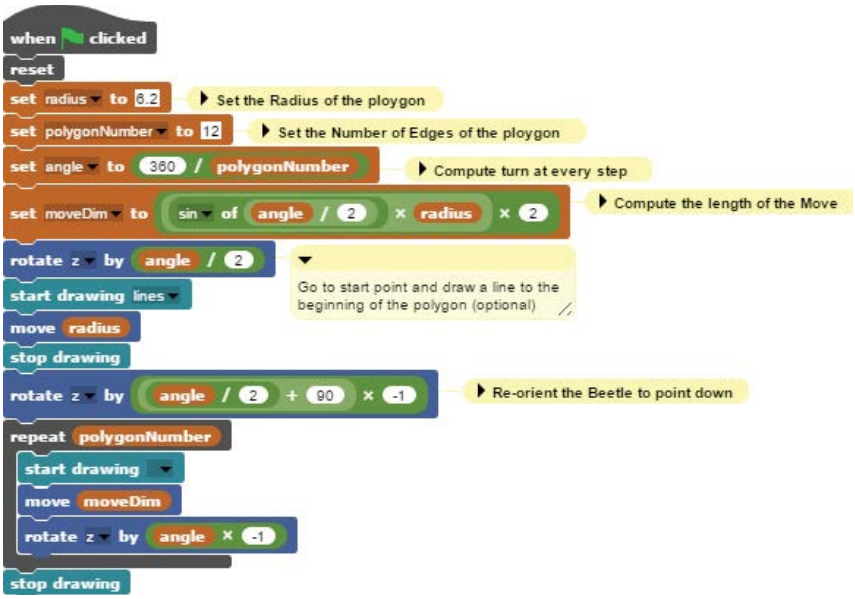
TEACHING COMPUTATIONAL IDEAS

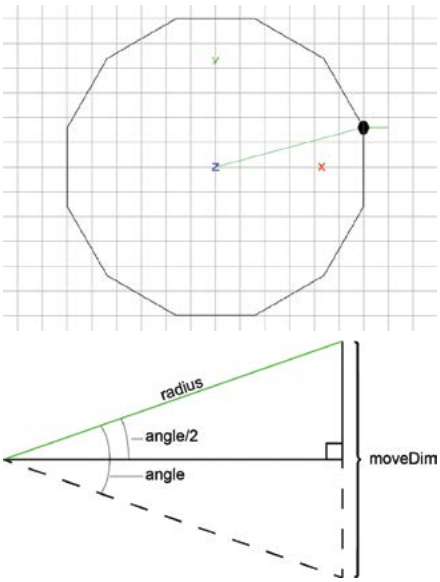
We aspire to connect mathematical and computer science ideas to design (Eisenberg 2002) and use fundamental ideas of constructionism (Rusk, Resnick, and Cook 2009). We demonstrate how our language can be used to convey computational ideas to novice programmers by including several examples we have used in workshops and other courses.

Combining Trigonometry and Turtle Moves

When teaching fundamental ideas in computation and geometry, it is productive to show students how turtle geometry can be used to construct mathematically defined shapes via discrete steps. The example of a polygon relies on trigonometric relations of the distance between its center and a vertex [radius], the distance of a move forward [moveDim], and the angle between the center and two adjacent vertices (Figure 9). A user can specify the size of the radius and how many edges the polygon should have. The comments inside the yellow blocks tell users which values can be altered. This discrete circular polygon around a center uses turtle moves and trigonometry to compute the relevant rotations and steps.

Other trigonometric functions can be used in conjunction with beetle moves to describe mathematical objects in 3D. The following example uses the extrude command and is based on incrementing two angles, [theta] and [phi]. The positioning of





9 Program that draws a polygon (user input: radius, number of edges).

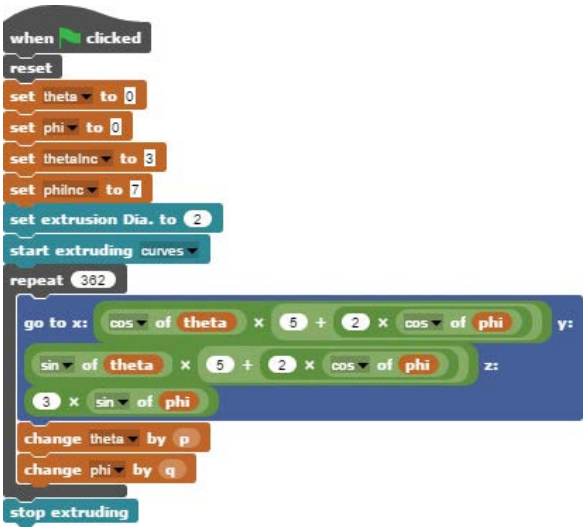
the beetle occurs via the [go to (x) (y) (z)] command, which uses absolute coordinates rather than local moves. Sine and cosine function are used to compute the positions along a torus and the program completes two cycles around a torus (Figure 10). The [go to (x) (y) (z)] allows for global positioning and instructors can teach the relevance of absolute coordinates in conjunction with trigonometric functions.

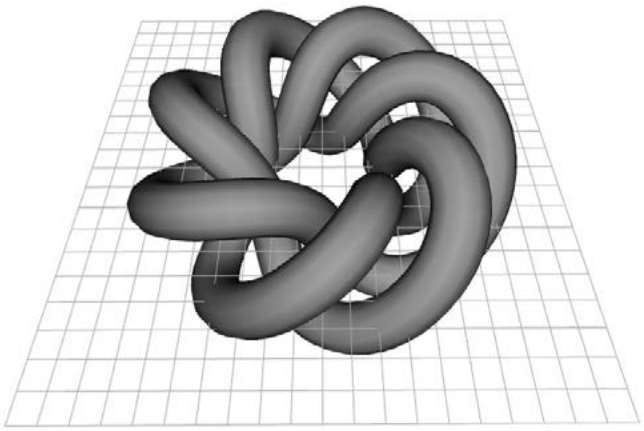
A Boolean Operation that Relates to Geometry

Boolean operations mean different things in geometry and computer science. This example demonstrates how an 'if-statement' can be used to define an area that should not be filled with cubes (Figure 11). A user can specify the size of the square and the circular boundary that is to remain empty. A boolean check is used here as a way to control a boundary within which the geometric boolean effect can be visualized.

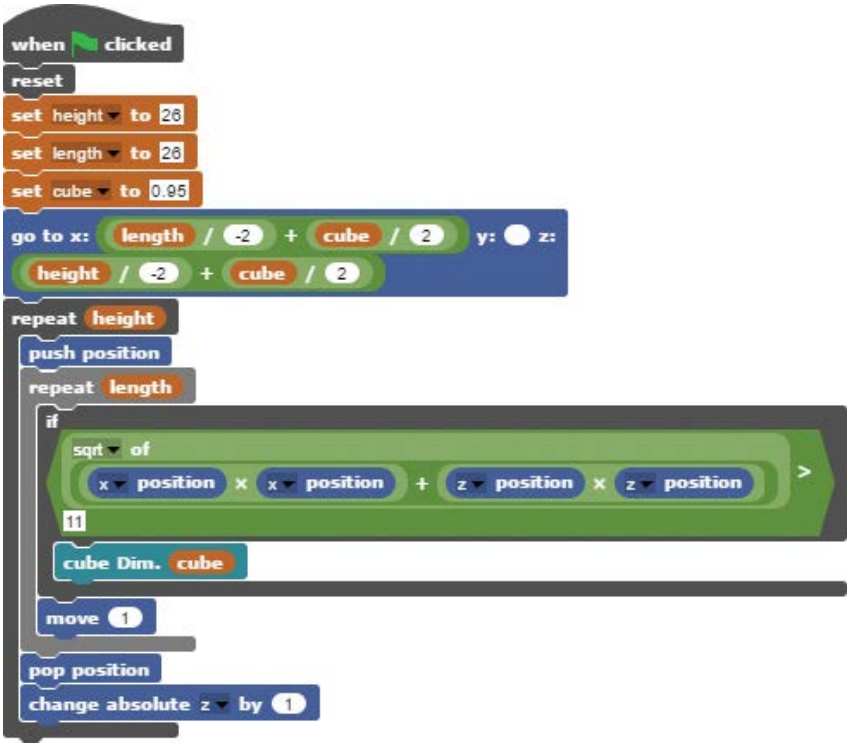
Nested 'For-Loops'

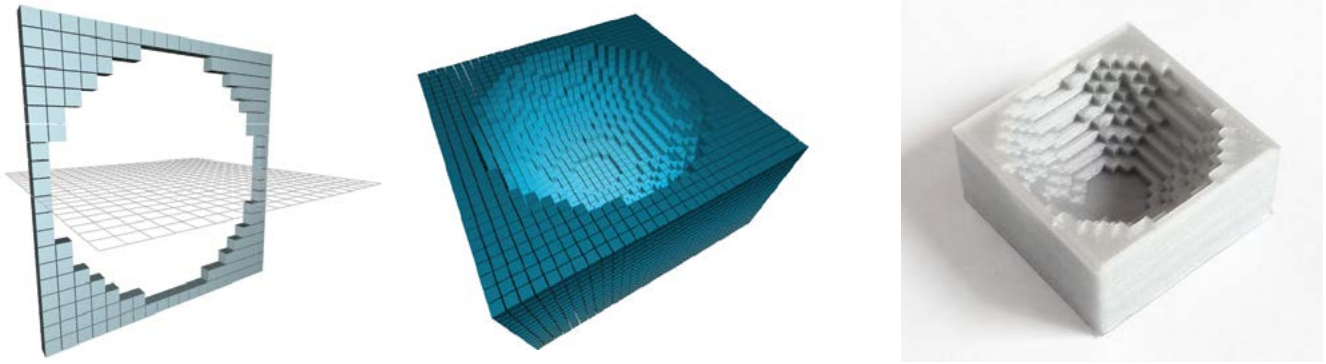
The following example uses a nested for-loop to move and turn the Beetle such that it walks along an undulating path within a certain range in z. The extruded path creates a loop that is made of smaller loops and in a way visualizes the code structure of the program. All moves are local turtle moves and no computation of absolute values is needed to determine the path (Figure 12).



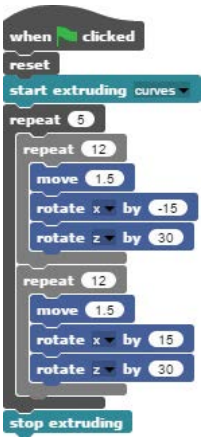


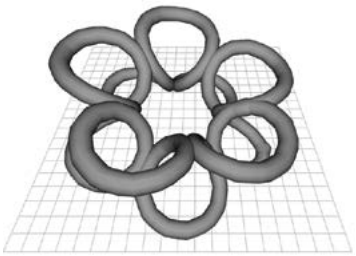
10 Program that draws 2 paths on a torus.


when clicked
reset
set height to 26
set length to 26
set cube to 0.95
go to x: length / -2 + cube / 2 y: z:
height / -2 + cube / 2
repeat height
push position
repeat length
if
sqrt of
x position x position + z position x z position
>
11
cube Dim. cube
move 1
pop position
change absolute z by 1

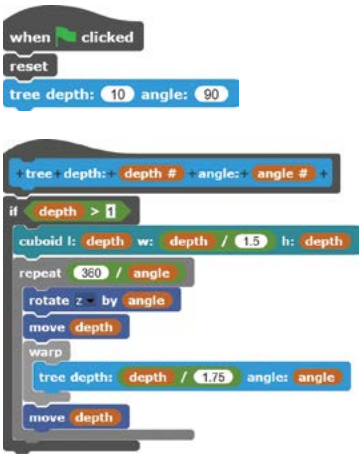


11


when clicked
reset
start extruding curves
repeat 5
repeat 12
move 1.5
rotate x by -15
rotate z by 30
repeat 12
move 1.5
rotate x by 15
rotate z by 30
stop extruding



12


when clicked
reset
tree depth: 10 angle: 90
tree depth: depth # angle: angle #
if depth > 1
cuboid l: depth w: depth / 1.5 h: depth
repeat 360 / angle
rotate z by angle
move depth
warp
tree depth: depth / 1.75 angle: angle
move depth

13

Recursion and Fractals

The relation between recursion and fractals is well known and can be used to teach the computational idea of a function-call within a function. The next example uses the numeric values that are used to count down within the recursive algorithm as a variable for the sizes of the used cuboid. The result is a surprisingly short program, here shown with a custom Block that allows users to create their own methods (Figure 13).

Using Lambda (λ) in Design Pedagogy

With the new implementation in Snap!, we can make use of Lambda functionality, which is a computational concept that was introduced in Lisp. The idea is based on message passing and permits programmers to use function calls as input for other functions (Figure 14). This is a powerful idea that can teach how one can write a program that writes a program.

The next example draws a tree-like structure and uses four items in a list [treeParts] that all execute something individually. The custom Block [branch] uses a recursive structure by selecting random items in [treeParts], which consist of 'ringified' calls of [branch]. The random selection within the list has a bias toward the branch function as it is used twice. The two spheres have different sizes.

The counterpart [run()] allows users to execute functions that were ringified, used twice in the example. All moves are controlled via turtle geometry. The variable [branchAngle] only exists within the [branch] procedure. This is a way to teach scoping and availability of variables (Figure 15).

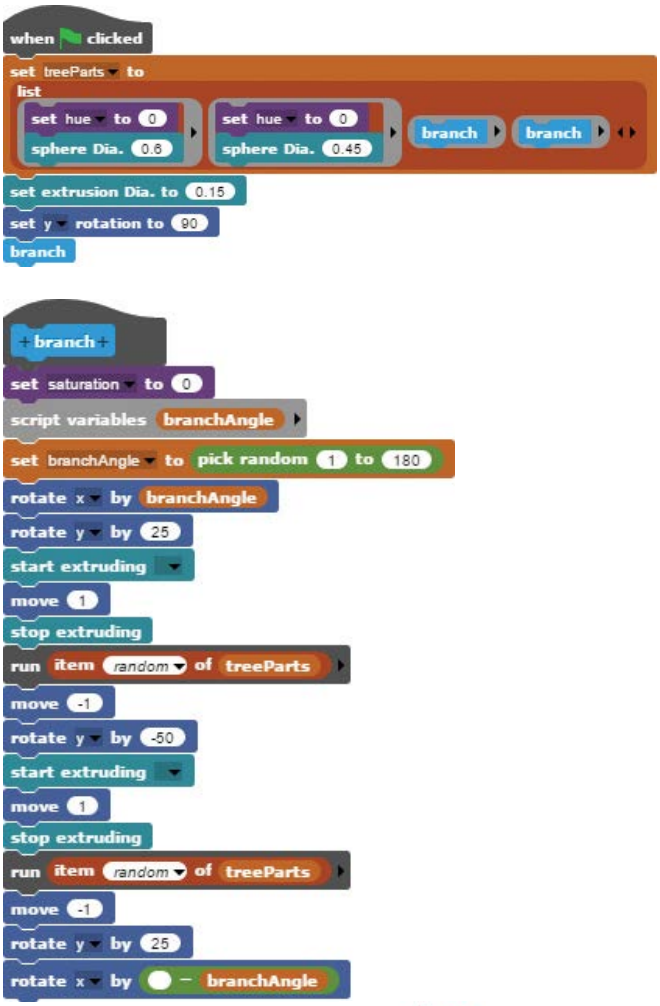
CONNECTING PROGRAMMING TO MAKING

We use the studio model in our courses, which means that students make physical artifacts of their own virtual 3D designs to tinker with (Resnick and Rosenbaum 2013). These prototypes provide the basis for Papert's construction of thought and making, and manipulating materials becomes an intrinsic part of the shared learning environment (Eisenberg et al. 2003). A further ambition lies in teaching foundational manufacturing processes, such as forming, machining, additive manufacturing, and joining. A powerful way of relating manufacturing processes and digital fabrication can be achieved by using Beetle Blocks programs to write machine instructions. We have included a few student examples in this section that show how we link programming to making.

1D to 2D transformation and shaping

Forming is a manufacturing technique that transforms a material without taking any material away or adding to it. We focus on 1D to 2D transformations and use metal wire that can be bent into a 2D shape with a digital wire bender (www.pensalabs.com/

14



15

11 Program that fills a square with cubes, but avoids a circular zone at its center; 3D version of same algorithm with a spherical boundary and a 3D print.

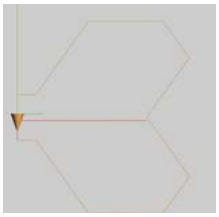
12 Example of extruded path made by a nested for-loop.

13 Recursive algorithm with a custom Block.

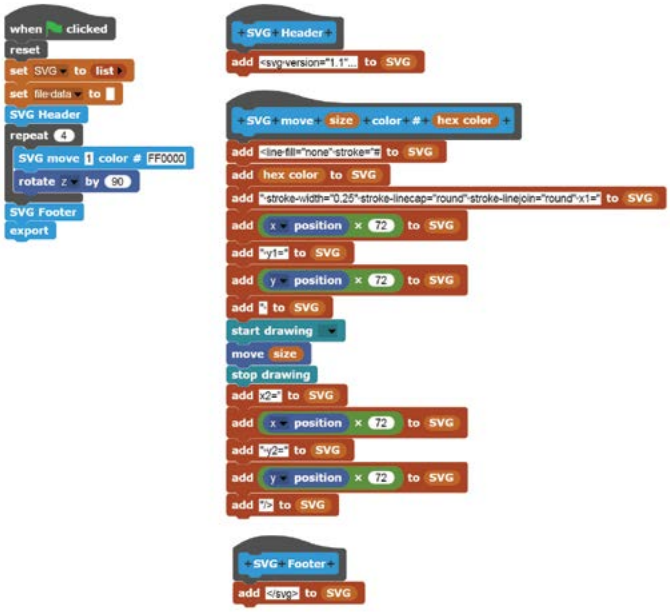
14 Ringify, a block that allows a procedure to be turned into an input.

15 An example of the use of Lambda functionality in Beetle Blocks.

16



17



18

diwire-overview/). The machine takes simple instructions that can easily be coded using Beetle Blocks moves to form a closed polygon (Figure 17). The resulting shapes were arranged to form the framework for a lamp shade.

2D instructions and SVG code generating

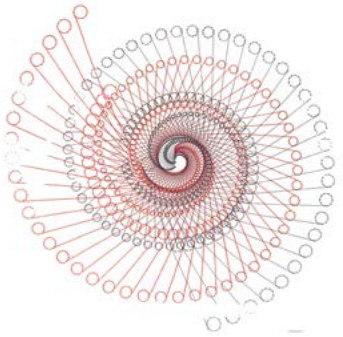
Regarding 3D manufacturing processes, we want to teach how machines use instruction sets that move a gantry. In order to demystify machine instructions, one can use file types that are human-readable such as the SVG file format. The goal is to demonstrate how one can create a file-writer that writes machine instructions. The example below shows how to record beetle moves and to create a file header and footer. The file is subsequently sent to a vinyl cutter with a 2D gantry and customized heads for various pen types (Figure 18). The student decided to use two different pens to create the drawing below.

3D printing

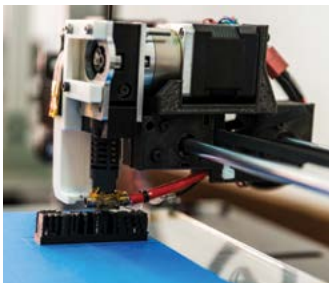
The last example focuses on additive manufacturing, in this case 3D printing. The pedagogical goal consists of showing an example that essentially works similar to g-code. The design below, by one of our students, consists of cubes that follow two curves that are altered with a sine function (Figure 20).

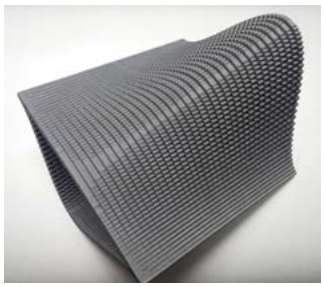
CONTRIBUTION

Beetle Blocks revisits the combination of two paradigms that are highly accessible for learners: turtle geometry translated into 3D and a graphical programming language adapted from Scratch, now based on Snap! The result is a novel teaching tool that is more accessible than other systems for algorithmic design.



19





20

16 Wire bender.

17 An example of a design with bent wire (Schuyler Klein, Alyssa Bearoff).

18 Program for a file writer.

19 Vinyl cutter with a custom pen holder, drawing made with two pens (Jackie Hsia).

20 A 3D print of an example in Beetle Blocks (Joseph Kim).

Drawing and sketching have been used to teach design for a long time and should not be replaced by other methods, but algorithmic design systems represent a powerful addition to that realm and allow us to address STEM learning in the context of architecture and design schools.

We believe that Beetle Blocks is a teaching tool that makes the power of this form of procedural design available to a broad audience of artists, designers, and architects. Beetle Blocks introduces a procedural way of thinking when constructing designs, which enriches the design vocabulary for architecture and design students.

It is essential to relate programming to making meaningful objects that learners use to construct mental models. We demonstrate in three case studies how we relate Beetle Blocks algorithms to digital fabrication and foundational manufacturing methods. Future steps consist of creating a sharing platform and expanding the geometry kernel.

REFERENCES

Abelson, Harold, and Andrea A. diSessa. 1986. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. Cambridge, MA: The MIT Press

Colella, Vanessa, Eric Klopfer, and Mitchel Resnick. 2001. *Adventures in Modeling: Exploring Complex, Dynamic Systems with StarLogo*. New York: Teachers College Press.

Eisenberg, Michael. 2002. "Output Devices, Computation, and the Future of Mathematical Crafts." *International Journal of Computers for Mathematical Learning* 7 (1): 1–44.

Eisenberg, M., A. Eisenberg, S. Hendrix, G. Blauvelt, D. Butter, J. Garcia, R. Lewis, and T. Nielsen. 2003. "As We May Print: New Directions in Output Devices and Computational Crafts For Children." In *Proceedings of the 2003 Conference on Interaction Design and Children*. Preston, England: IDC. 31–39.

Harvey, Brian, and Jens Mönig. 2010. "Bringing 'No Ceiling' to Scratch: Can One Language Serve Kids and Computer Scientists?" In *Proceedings of Constructionism 2010: The 12th EuroLogo Conference*. Paris: EuroLogo. www.cs.berkeley.edu/~bh/BYOB.pdf

Kelleher, Caitlin, and Randy Pausch. 2005. "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers." *ACM Computing Surveys* 37 (2): 83–137.

Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. "The Scratch Programming Language and Environment." *ACM Transactions on Computing Education* 10 (4): Article no. 16.

Papert, Seymour A. 1993. *Mindstorms: Children, Computers, And Powerful Ideas*. New York: Basic Books.

Petts, Malcolm. 1988. "Life After Turtle Geometry With a 3D Logo Microworld." *Mathematics in School* 17 (5): 2–7.

Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. "Scratch: Programming for All." *Communications of the ACM* 52 (11): 60–67.

Resnick, Mitchel, and Eric Rosenbaum. 2013. "Designing for Tinkerability." In *Design, Make, Play: Growing the Next Generation of STEM Innovators*, edited by Margaret Honey and David Kanter. New York: Routledge. 163–181.

Rusk, Natalie, Mitchel Resnick, and Stina Cook. 2009. "Origins and Guiding Principles of the Computer Clubhouse." In *The Computer Clubhouse: Constructionism and Creativity in Youth Communities*, edited by Yasmin B. Kafai, Kylie A. Peppler, and Robbin N. Chapman. New York: Teachers College Press. 17–25.

IMAGE CREDITS

Figure 3: Papert, 1993
Figure 17: Klein and Bearoff, 2015
Figure 19: Hsia, 2015
Figure 20: Kim, 2015

Duks Koschitz is Associate Professor and Director of the Design Lab at Pratt Institute. His wrote dissertation at M.I.T. on Curved-crease Paperfolding and had research positions at M.I.T. and the ETH. He worked at NMDA, Office da, Morphosis, Asymptote, Coop Himmelblau and Ian Ritchie Architects. He graduated from the T.U. Wien in 1998.

Bernat Ramagosa is a software engineer at Arduino SRL. He developed an online programming school and a social knowledge management system at the Citilab (Barcelona). He is the author and lead developer of Snap4Arduino and the lead developer of Beetle Blocks. He holds a Bachelor's and Master's degree from the Open University of Catalonia.

Eric Rosenbaum wrote his doctorate "Explorations in Musical Tinkering" at MIT Media Lab's Lifelong Kindergarten group. He is co-inventor of the MaKey MaKey invention kit. His software projects include Singing Fingers (finger painting with sound), Glowdoodle (painting with light) and MelodyMorph (creating musical instruments and compositions). He holds a Bachelor's (psychology) and a Master's degree from Harvard University.