EXPLORING ALGORITHMIC GEOMETRY WITH “BEEGLE BLOCKS:” A GRAPHICAL PROGRAMMING LANGUAGE FOR GENERATING 3D FORMS

Duks KOSCHITZ and Eric ROSENBAUM
Massachusetts Institute of Technology
Both authors contributed equally to this work

ABSTRACT: We are introducing a new teaching tool with the power to transform the way designers, architects and artists conceive of the construction of space. Our audience is comprised of users with no prior knowledge of programming who wish to learn how to use computational concepts as part of their design process. Computational algorithms open up possibilities for generating 3D structures that are not easily accessible with existing tools. “Beeble Blocks” is a software tool that combines an easy-to-use graphical programming language with an intuitive 3D environment.

Figure 1: A design created with Beeble Blocks

Keywords: Visual Programming, Algorithmic Design, Creative Learning, Procedural Geometry.

1. INTRODUCTION - MOTIVATION AND AUDIENCE
The audience for "Beeble 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. Programming is not yet fully integrated in art and architecture programs and therefore the learning curve is steep. A
graphical language that is easy to learn is a first step in making programming more accessible [4]. "Beetle Blocks" is based on Scratch, which is a graphical control flow language [6]. This approach to programming requires the user to formulate every explicit step of an algorithm in a visual way, without the frustrating syntactic pitfalls typical to text-based programming. It also positions the fundamental concepts of computation early in the learning process (unlike other graphical programming paradigms), making it more likely that learners will build on these concepts, moving on to tasks of greater complexity.

The 3d environment of Beetle Blocks is intuitive because it is modeled after "turtle geometry" [5]. 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.

There is no doubt that relational or parametric modeling is powerful for architectural design, but any geometric relationship can also be created procedurally. By learning how to create procedures, we hope that designers will acquire a deeper understanding of how computation can be used. After learning how to design with Beetle Blocks, they can transition to commercial software for complex projects.

Thinking about the construction of space procedurally is a powerful complement to compositional thinking, which is the prevailing method to teach design in Architecture. Design is often judged based on its coherence and how well thought out a concept is. Using procedures to construct a spatial constellation can encourage a designer to think about how to control an arrangement of objects in space. This way of thinking requires rigor in the design process and allows the designer to create coherent designs.

2. FROM ‘TURTLE’ GEOMETRY TO ‘BEETLE’ GEOMETRY

2.1 Drawing with a turtle

In the Logo programming language for children [5], 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.” The term “turtle geometry” refers to a geometry described by relative movements as opposed to absolute coordinates. 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. We use our intuitive knowledge of a relative coordinate system to construct the idea of an absolute system. Relative coordinate systems are therefore a good starting point for learning geometry.

In Logo, turtle movement was restricted to a plane. Abelson and diSessa elaborated on turtle geometry, exploring the possibilities of turtle movement in 3d space [1]. There have been various implementations of Logo that enable turtle drawing in 3d, such as Elica [www.elica.net] and Logo3D [logo3d.sourceforge.net]. None of these use a graphical language. StarLogo TNG [education.mit.edu/starlogo-tng] has both a graphical language and a 3D world 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.

Beetle Blocks combines the ease of learning of a graphical language with 3d turtle geometry. We refer to 3d turtle geometry as “Beetle geometry,” and to our turtle as a “Beetle.”

2.2 A discrete approach

We know that the world is continuous at many levels of description. However, the way we see the world is fundamentally discrete. Our perceptual systems for instance are in many ways discrete: our eyes are organs that detect photons and convert them to electro-chemical impulses at a certain resolution. It is
important to bear in mind the world's continuity when dealing with computational systems, which are discrete.

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:

- An image of a circle drawn on paper.
- \( c = 2\pi r \)
- For 360 times, move forward 1 unit and turn one degree to the right

All three representations have deep relationships to the way we teach design: some artistic, some engineering oriented and some maybe more scientific. What is important is that a designer knows them all and can control the relevant methods to express what he or she wants to design. We will focus on the discrete approach.

3. DESIGNING A PROGRAMMING LANGUAGE

3.1 Control flow vs. 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; 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 a control flow execution model, because our emphasis is on making programming understandable. The control flow representation allows Beetle Blocks to show the execution of a program explicitly, making it easier for users to see what the program is doing.

3.2 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 encapsulates lower level commands within the very same language for geometric calculations that the user can access; these lower level commands themselves encapsulate primitives, such as commands for rendering pixels on the screen, that are not accessible to the user, being written in a lower level language, and 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 should consist of a set of commands that combines accessibility and flexibility, and hide the lower level commands that provide an irrelevant level of detail or are not sufficiently intuitive [5].

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.
3.3 The Scratch paradigm

Beetle Blocks makes use of several of the design principles developed for the Scratch programming language [6]. These include immediate feedback, commands as visual building blocks, and error prevention.

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 out 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.

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.

4. BEETLE BLOCKS

4.1 The Beetle and its shapes

"Beetle geometry" relies on a "Beetle-centric" pointer that can be moved away from an absolute coordinate system and re-oriented using its own relative coordinate system. The designer can choose to place geometric primitives at the current location of the "Beetle" as shown in Figure 3a. It can also change its state and be used to form an extrusion along the path of its movement or create scaled shapes in various colors. The designer is in full control of a composition of transformations or the positioning of primitives while moving about in 3D space.

![Image](image-url)

Figure 2: a) the Beetle at the origin; b) the Beetle moved by 80 pixels; c) the Beetle rotated 45 degrees about its own x-axis; d) the Beetle rotated 60 degrees about its own y-axis

The absolute coordinate system is shown with its origin where the x-, y- and z-axis cross each other. The x-axis is green, the y-axis is red and the z-axis is blue and points into the paper similar to the convention in 3D animation. The Beetle always points in the z direction of its own relative coordinate system. The {move} block moves the Beetle forward in the direction it is pointing. Where ever the Beetle is located in space it will move and rotate relative to its current position. The {rotate x} block rotates it around its own green axis. The {rotate y} block rotates it around its own red axis as shown in
Figure 2.
Shapes can be drawn at the current position of the Beetle. The shapes drawn in Figure 3a are first a sphere at 40, then a cube at 100 and after that a cuboid at 170 from the absolute origin. The available geometric primitives are spheres, cubes, cuboids and a 3d-trace that consists of a cylindrical extrusion with spherical caps as shown in Figure 3b. Its start and end diameter can be adjusted. The trace can be drawn continuously and will always have smooth joints at its beginning and end as they are spherical.

4.2 Positioning the Beetle
The Beetle can be moved and rotated with the \{move\} and \{rotate\} blocks as shown in Figure 4b. The \{change x/y/z by\} blocks move, but don’t reorient it. The \{change scale by\} block will scale the shape placed at the current position of the Beetle.
Figure 4: The Categories (a) Shapes, (b) Positioning, (c) Color

4.3 The Beetle’s color
Colors can be chosen by setting hue, saturation and lightness as this color system is easy to numerically control once the designer understands the concept behind it. The \{change H/S/L by\} blocks can be used to iteratively change colors as shown in Figure 4c.

4.4 Control
The remaining categories relate to the way a program is controlled rather than geometric manipulation. In the control category, the \{clear\} block empties the scene. The \{repeat\} block creates repetitive loops and the \{if\} block creates conditional statements. These two blocks have a "c" shape indicating that they can stretch to contain a set of blocks, including other control blocks. Complex program structures such as nested repeats can be constructed in this way. The \{push\} and \{pop\} blocks store and retrieve the Beetle’s position and rotation on a stack (see figure 5a). For example, a program can push the Beetle's position onto the stack, move the Beetle elsewhere, then pop the position off to return the Beetle to its earlier position (see the examples in figures 8 and 9).

4.5 Operators
The Boolean and numerical operators enable logical and arithmetic manipulation. Complex expressions can be created by nesting these blocks inside each other, as shown in figure 9. Note that operators with a Boolean output are always hexagonal, so they can only fit into hexagonal slots, such as the one in the \{if\} block as shown in Figure 5a.

4.6 Variables
Variables can be created with any name, set to a value and altered iteratively as shown in Figure 5c.
4.7 Output

In order to make the design process with Beetle Blocks more engaging and to enable connections between the digital and physical worlds, Beetle Blocks allows users to export their designs in a format that can be fabricated on a color 3d printer. Previous work like Javagami (for creating paper craft versions of 3d designs) and Spectre (for printing out layered transparencies showing a 3d design) [2, 3] point out the importance of creating physical versions of a digital design in the learning process.

By printing out a physical instantiation of their Beetle Blocks design, users can evaluate its success in a more intuitive way. The physical object also has properties, such as its strength, connectedness and relationship with gravity, that cannot be explored in the virtual model. This iterative way of going back and forth between virtual and physical designs helps to create a more reflective and rigorous design process.

5. EXAMPLES

Below are four examples that show the Beetle Blocks program on the left and the drawn geometry on the right. Figure 6 is showing a representation of a spiral staircase that changes color and tread width according to the state of the current computation and uses a nested loop. The variable {a} represents the width of the treads, and decreases each time through the outer loop. The inner loop draws each tread.

Figure 7 shows a 3d spiral shape that becomes larger the further away it is from the origin. The Beetle's state gradually scales the trace as it moves along a 3d spiral. This is accomplished by declaring variables that are used as parameters in the {trace} block and are incremented at every step.

The example Figure 8 shows a discrete approximation of a sphere that is constructed of traces and cubes. As the circle is drawn with a full step forward in the very beginning, the following two circular motions would not have the same center point and the shape would not approximate a sphere. This can be controlled...
Figure 6: A spiral staircase with shortening treads using a nested loop.

Figure 7: A 3d spiral using 3 variables and a 3d-trace that is scaled at every step.
Figure 8: A sphere constructed of 3d-traces and cubes using [push] and [pop].

Figure 9: Several chains of cubes along changing curves using the [sin] function.
by using {push} and {pop} to remember the Beetle's position and retrieve it at the beginning of the next circle. Hierarchical and structural concepts can be taught by combining traces and primitives. In this case, the traces function as a skeleton.

Figure 9 shows the use of a continuous function to place cubes in discrete steps. This program uses two variables. The variable \(v\) is incremented and used in the \(\sin\) block to generate the curve. The variable \(x\) scales the sin values such that each of the twelve described curves is slightly modified. This example shows how Beetle geometry can be mixed with Cartesian geometry: the Beetle moves along its heading as it draws each cube, but moves up and down using the \(\text{change y by}\) block.

6. CONCLUSIONS
Beetle Blocks combines two paradigms that are highly accessible for learners: turtle geometry translated into 3d and a graphical programming language adapted from Scratch. The result is a novel tool that is more accessible than other systems for algorithmic design. 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. 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 a spatial constellation. Beetle Blocks empowers designers to create coherent designs.

REFERENCES


ACKNOWLEDGMENTS
This research was supported by Professors Mitchel Resnick and Bill Mitchell. We’d like to thank them, along with the Lifelong Kindergarten group at MIT Media Lab, and others who contributed to the project, including Anisha Deshmane, Neil Chao and Evelyn Eastmond.

ABOUT THE AUTHORS
1. Duks Koschitz is working on his PhD in the Design & Computation Group in the Architecture School at MIT and is interested in design methods in a pedagogical context
2. Eric Rosenbaum is working on his PhD in the Lifelong Kindergarten Group in the Media Lab at MIT and focuses on creative learning concepts in design and music.