From the Turtle to the Beetle

The Beetle Blocks programming environment

Bernat Romagosa
Arduino.org
Via Romano, 12
10010 Scarmagno (TO)
Italy
bernat@arduino.org

Eric Rosenbaum
Google Creative Lab
76 Ninth Avenue, 4F
New York, NY 10011
USA
eric.rosenbaum@gmail.com

Duks Koschitz
Pratt Institute
200 Willoughby Avenue
Brooklyn, NY 11205
USA
duks@pratt.edu

ABSTRACT

Beetle Blocks is a visual, blocks-based programming language/environment for 3D design and fabrication, implemented on top of Berkeley Snap*!* and the ThreeJS 3D graphics library. Beetle Blocks programs move a graphical *beetle* around a 3D world, where it can place 3D shapes, extrude its path as a tube and generate geometry in other ways. The resulting 3D geometry can be exported as a 3D-printable file. Beetle Blocks also aims to offer a cloud system and social platform meant to provide the community with ways to interact and learn from each other. Beetle Blocks was previously implemented as a Scratch extension, and migrated into Snap*!* in 2014. We explain how the project has evolved since this migration, and in particular how the advanced programming features it inherited from Snap*!* shaped the kind of designs that are now possible with the new system.

CCS Concepts

•Social and professional topics → **Computing education; •Software and its engineering** → **Visual languages; •Computing methodologies** → *Shape modeling;* **•Applied computing** → *Interactive learning environments;* **•Human-centered computing** → Social networks;

Keywords

Visual programming; blocks-based programming; constructionism; 3D printing; educational programming;

1. INTRODUCTION

Since the early years of computing, programming has been thought to help develop analytical thinking, decision making, acquisition of mathematical concepts and problem solving abilities, among other skills. However, programming languages are built by people who are already very proficient at computer programming, and for this reason they tend to present several learning barriers that make them unsuitable for beginners and people with no technological background, thus making it very hard to study the hypotheses that link programming to the acquisition of these abilities.

There have been numerous attempts at building languages that aim to overcome these barriers, but our main focus is in the approach started by Seymour Papert with LOGO[18]. With this language, Papert opened the path to several other languages that built on his constructionist principles, such as Smalltalk[12], Etoys[13] or Scratch[9]. The latter has become the de-facto standard of educational programming languages and, with more than 12 million users worldwide, Scratch has made dynamic, parallel, live programming available to all by means of a visual metaphor inspired by building blocks. Scratch has also paved the way for several new blocks-based languages, such as Snap*!*[5], a complete reimplementation of Scratch aimed at bringing powerful computational ideas into the world of visual drag-and-drop programming.

Beetle Blocks is part of this constructionist line of languages and, by being built on top of Snap*!*, it inherits its advanced programming features[6], such as closures, first class lists or continuations. However, Snap*!* still lags behind Scratch in its lack of a social platform and community management tools. The future new cloud system and project sharing social site for Beetle Blocks aim to overcome these shortcomings and provide the user community with a set of tools with which they can learn from each other's projects and both share their creations and discover the creations of others. Eventually, the system can easily be adapted to work with Snap*!* in benefit of its much larger user base.

In Beetle Blocks, the LOGO turtle becomes a beetle that moves around a three-dimensional world and, instead of being programmed in a textual fashion, it embraces the dynamic, live, parallel blocks-based programming paradigm made popular by Scratch. This mixture of programming and computer-generated graphics aims to bring closer the worlds of computation, art, 3D design and the maker movement in both directions: by allowing to approach the more artistic side of the environment from a computational point of view, and by allowing to approach the world of programming from an artistic point of view.

2. RELATED AND PRIOR WORK

2.1 Brief History of Beetle Blocks

The early origin of Beetle Blocks traces back to a couple of

side projects by Eric Rosenbaum dating between 2005 and 2008, in which he explored L-Systems and 3D turtle geometry in different ways. In one of these projects, Scratch for Second Life[20], one could build scripts in Scratch and translate them automatically into text-based snippets that could be later ran by a three-dimensional object in the Second Life virtual world. These objects had the capability of leaving line segments behind them in a similar fashion to what we currently call *extrusions* in the Beetle Blocks system.

The actual project in its current vein was started around 2010 at MIT by Duks Koschitz and Eric Rosenbaum, and was inspired in Rosenbaum's early projects, the Designblocks[3] project, by Evelyn Eastmond, and StarLogo TNG[16], by Mitchel Resnik et al. Beetle Blocks began as a possible doctorate path for Duks Koschitz, who was interested in making programming accessible to designers, with early prototypes being built in Flash, Paperspace and Processing. The latest prototype was built as a Scratch 2.0 extension and presented in a co-authored paper at the ICGG conference in Montreal in 2012[17].

Towards the end of 2013, Jens Mönig and Bernat Romagosa joined the project and began porting it to Snap*!*, with Romagosa becoming its main developer soon after the initial port had been concluded.

Shura Chechel, an undergraduate architecture student and research assistant for Duks Koschitz, has been helping in the graphic design of the user interface and social platform since July 2015.

2.2 Similar Projects

There are several other systems that accomplish similar purposes in that they allow users to create three-dimensional models programmatically. Environments such as OpenSCAD[15] or OpenJSCAD[10] are also aimed at generating 3D geometries in a programmatic fashion, but differ from Beetle Blocks in three fundamental points: they are text-based, they do not embrace turtle geometry and they are non-interactive, in the sense that the whole program must be ran at once and cannot be modified live at runtime.

In contrast, there have been many 3D turtle geometry environments and even 3D LOGO implementations in the past[19][23]. These are generally also text-based and, although they are interactive in the sense that individual instructions can be ran arbitrarily at any time, they do not allow for real-time modification of running scripts.

BlocksCAD[11] is a blocks-based environment that, in appearance, is very similar to Beetle Blocks. However, it does not provide a turtle graphics metaphor and is also non-interactive. In fact, BlocksCAD and other Blockly-based environments are not actual languages[4], but graphical representations of an underlying textual language. In that sense, one could argue that these are just very visually advanced syntax highlighters. They do offer many of the advantages of blocks-based languages, such as removing the need to memorize instructions or making syntax errors impossible, but they lack the constructionist capabilities of exploration offered by a live, real-time system. In comparison, languages like Scratch or Snap*!* -and, thus, Beetle Blocks- do not translate blocks into text, but instead directly read them and interpret them live inside their environment[9].

Antimony[14] is a system that presents itself as a Lisp-evolved CAD tool, embracing a different visual metaphor based in flowcharts. This environment is both visual and real-time, and it does offer 3D modeling features that are far more powerful than the ones present in Beetle Blocks, such as revolution solids or proper constructive solid geometry (CSG) operations. The main conceptual differences between Antimony and Beetle Blocks lay in their different approaches to modeling. In Antimony, the user sequences inputs and outputs instead of building computer programs, and is also allowed to modify shapes via direct manipulation, as opposed to the Beetle Blocks idea of static geometry only modifiable by algorithmic means. Lastly, Antimony does not provide first-person-like turtle geometry operations.

Another Snap*!* modification that tackles 3D computer graphics is CSnap[8], by the Renssaeler Polytechnic Institute. CSnap features 3D sprites and allows for generation of 3D shapes, albeit also not in a turtle-graphics manner.

3. THE SNAP*!* FOOTPRINT

Switching to Snap*!* has provided Beetle Blocks with a wide range of advanced programming features, shaping both the way users can construct their geometries and the kind of geometries the system can produce.

The following subsections are dedicated to showing particular applications of these features in Beetle Blocks.

3.1 Custom Blocks

Functions are the natural way in which programmers abstract complexity. In Snap*!*, functions come in the shape of custom blocks that users can create at their own will to extend the standard library of the language.

In workshops aimed at children aged 15 and younger, we noticed most students faced difficulties in grasping the concept of moving the beetle along its three local axis, but they did easily understand what it meant to move an object relative to the six faces of its imaginary bounding box. Similarly, it was hard for them to understand what rotating an object around its local three axes meant, whereas they had no problem understanding it when we rephrased our explanations in terms of rolling left and right, and turning up, down, left and right.

The possibility of creating custom blocks allows us to adapt the system to circumstances like these by building a special library that moves the beetle around in a way that feels more natural to children.

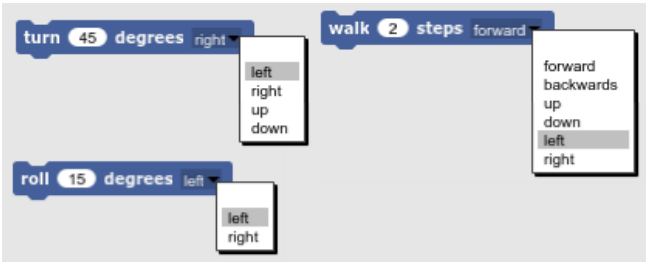

Figure 1: Simplified first person movement library

However, custom procedures are not only useful for teachers to adapt the system to the needs of their students. Being able to abstract complex operations in a three-dimensional environment is especially important, as building 3D geometries usually takes a substantial amount of individual instructions that would otherwise clutter our work space, mak-

ing it very hard to understand, debug and extend our programs.

Take, for instance, the script that instructs the beetle to build a ring of 24 spheres shown in figure 2. Once we have understood how relative movement works and how many times we need to rotate a particular amount of degrees to complete a full circle, it is in our best interest to abstract this process so that we can reuse it later on.

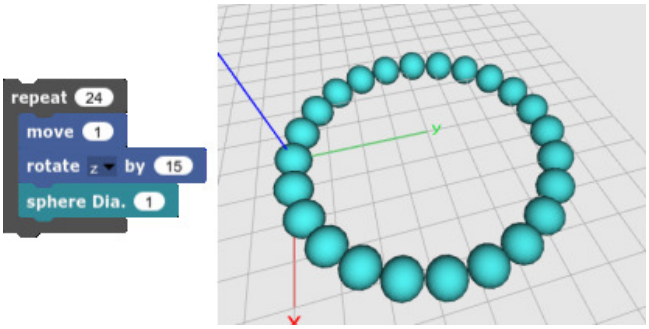

Figure 2: A script that builds a ring, and the result of running it

Abstracting this script into a custom block not only unclutters our workspace, but also serves as a perfect opportunity to think of possible parameters that will make it useful for different cases. In the case of figure 3, we added a parameter to our new block that lets users customize the number of spheres of a ring.

Figure 3: Definition of a custom block for building rings, and an instance of the new block

3.2 First Class Procedures

The ring generator example makes for an interesting exercise that many languages, especially educational ones, would be forced to solve only partially and in a not very elegant way:

> How could we generalize the definition of our custom block so that it can be used to generate any shape we want?

In traditional imperative languages, one can only solve this problem by enumerating a predefined list of possible shapes, like figure 4 shows.

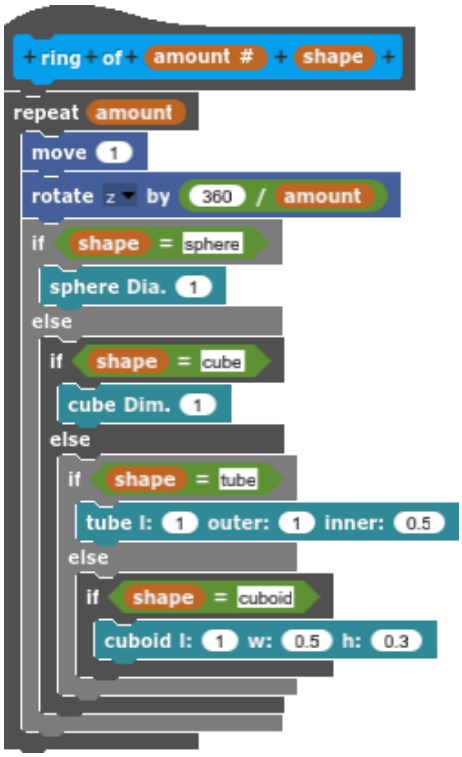

Figure 4: An imperative language approach to generalizing behavior

Although this certainly gives our new block the power to generate different shapes, we are going to need to modify this definition every time we want to add a new case, along with remembering the name we have given to it. Languages with first class procedures solve this by simply allowing functions as regular parameters. We can thus redo our generalized ring generator as displayed in figure 5.

Figure 5: A custom block taking a procedure as an argument, plus an example of the block being used to generate a ring of 30 cubes

Snap*!*'s visual metaphor helps us easily understand that we are parametrizing entire scripts, not just individual shapes,

allowing us to create rings of shapes as complex as we wish.

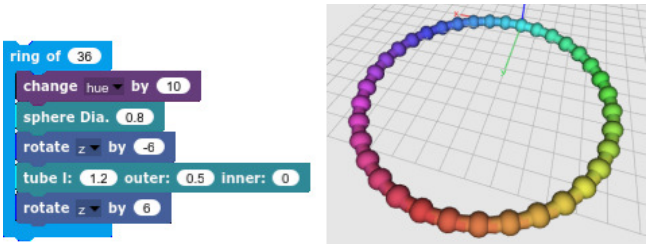

Figure 6: A ring of colorful spheres connected by tubes

3.3 Recursion

Although first class procedures are not at all indispensable for recursion, they certainly help recursion show up naturally. A very likely next step for curious programmers is to embed this ring generator inside another one and see what happens. After adding a new parameter to our block to control spacing between ring steps, we can now generate rings of rings in a very natural way.

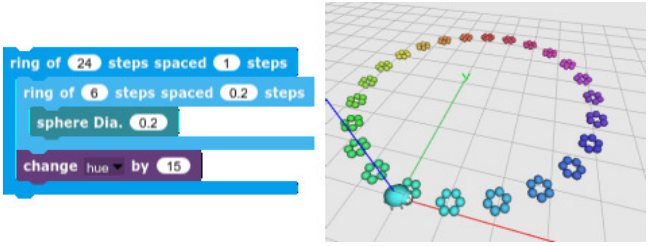

Figure 7: A recursive ring of depth 1, built by embedding a ring generator inside another one

The example in figure 7 shows a first class function being passed as a parameter of itself and how this results in a recursive 3D geometry. The circular pattern repeats itself in a way that resembles what we see when looking at a fractal shape. Fractals are, however, infinitely repeating structures, whereas our ring of rings only repeats itself once.

When drawing fractals we always need to make a compromise between the definition of the term and what is technically feasible. Drawing an infinite structure would take infinite time, and since our screens have a finite number of pixels, it makes no sense to keep on calculating beyond what we can represent with them. That is why, when defining a fractal shape generator, we always implement a base case for which our procedure will stop drawing deeper.

Having real procedures allows us to call a block from inside itself in a recursive manner, so that we can abstract the process shown in figure 7 by parametrizing how many times the procedure is going to be embedded in itself.

Figure 8 shows a generalized ring of rings that lets us choose how many levels of recursion we want the beetle to draw, keeping the amount of steps constant through depth levels, and reducing the space between each step by a factor of half the number of steps at each depth level.

One advantage of working with a dynamic and graphical environment is that it makes it easy to experiment with code and see the results of our tinkering in real time. Often,

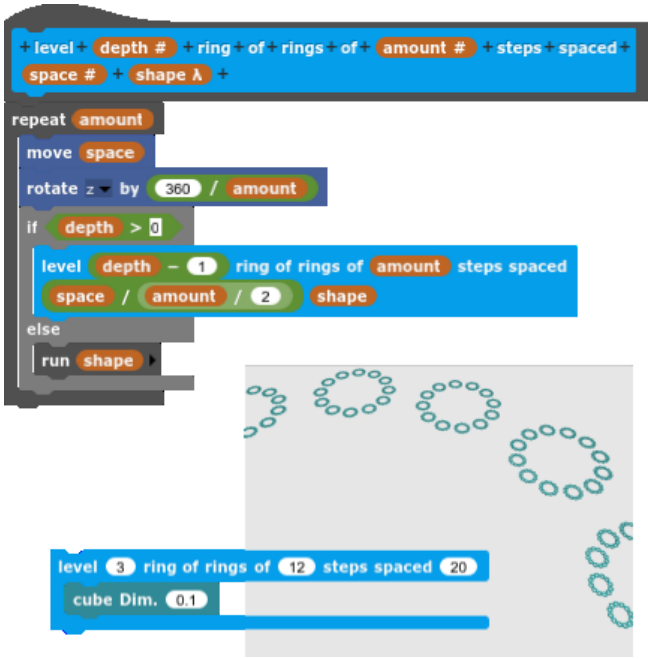

Figure 8: An implementation of a recursive ring of arbitrary depth, along with an example of generating a depth-3 ring of cubes

this can give us hints of properties of code that would have otherwise been very hard to notice, like how we can generate a Koch snowflake fractal by defining it as a recursive 6-stepped ring.

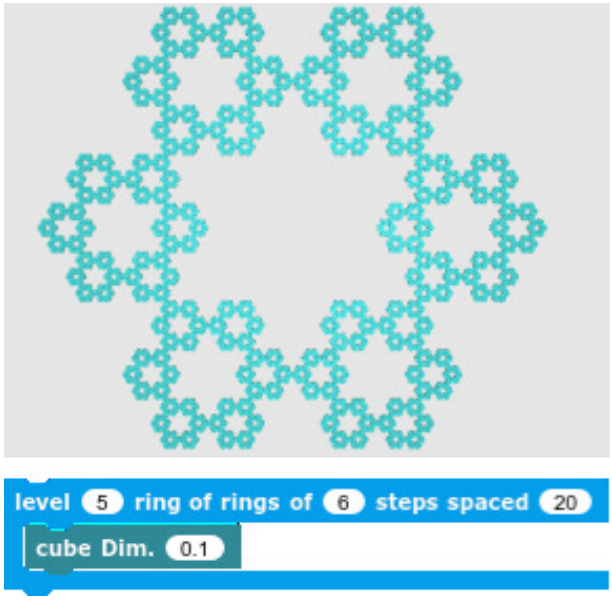

Figure 9: A Koch snowflake emerging from a 6-stepped recursive ring of depth 5

3.4 First Class Lists

When drawing geometries in Beetle Blocks or other turtle geometry systems, we need to think in first person. That is:

we do not draw geometries according to formulas that define positions of an infinite set of points in space, but rather in terms of relative movement and rotation of a drawing head. Drawing shapes such as polygons of definite side length and irrelevant center becomes a trivial task in turtle geometry, but this change of perspective can make it harder to draw shapes such as ellipses or polygons with a definite center.

A useful application for lists of lists in a programming environment for 3D geometries is to represent points in space. In Snap*!* and, thus, in Beetle Blocks, this can be done by initializing a variable to an empty list to later fill it with new lists containing the x, y and z components of different positions. These positions can later be retrieved by accessing the items in our *positions* list, and used to set the position of the Beetle. We can then easily abstract these operations into custom blocks that hide their internal complexity.

Take, for instance, the seemingly easy task of drawing a circle of a definite radius centered at a particular point in space. We know, from having built the ring code, that drawing circumferences in turtle geometry is just a matter of turning a particular amount of degrees and moving forward until we complete a full circle, but centering a circle built in that way and defining its exact radius entails non trivial trigonometric calculations.

A promising approach is to move the beetle to the center of the circle we want to draw, then have it move as many steps as the radius of the circle, go back to the center, rotate a particular amount of degrees and repeat until the whole circle is closed. Indeed, this would work for drawing rings of spheres or cubes, like we have done before, but not if we wanted to extrude our path in a circular way, as figure 10 shows.

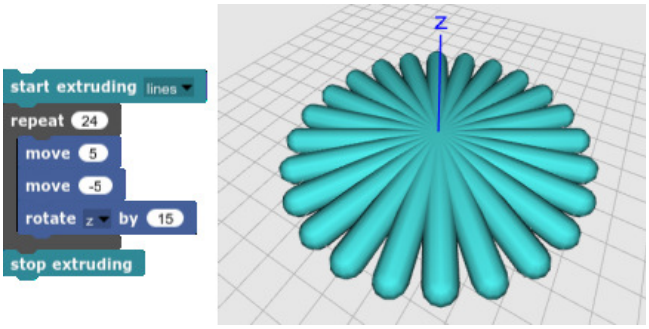

Figure 10: Failed attempt at drawing a radius 5 circle centered at origin

The process of searching for points in a radial way requires us to jump back and forth to the center between each point in the perimeter. Movement does not happen between each point, but between each point and the center. In cases like this, we can always first take the beetle to all positions in the circle and store them in a list, then later tell the beetle to revisit these positions one by one, as figure 11 shows.

As usual, once we have mastered the process, we could abstract it into a new block that lets us also choose how many control points we want our circle to have. As a side note, it is worth noticing that we are using curved extrusions in order to obtain a circle. The same code using linear extrusions would otherwise produce n-sided polygons.

Another advantage of heterogeneous first class lists is that one can implement and convert between any data type[6] by

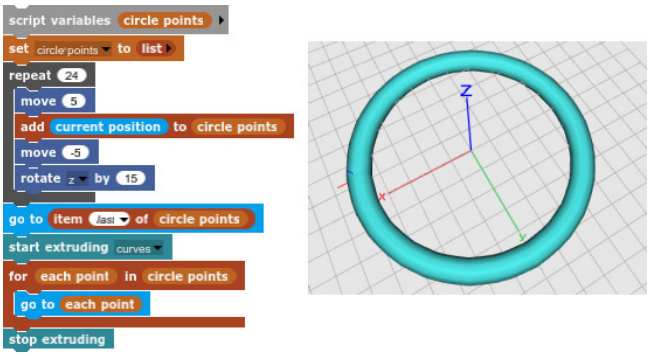

Figure 11: A radius 5 circle centered at origin, drawn by storing coordinates in a list and revisiting them later

means of them. This gives Beetle Blocks users the ability to work with data in elegant ways by building their own powerful abstractions. Additionally, the standard library in Snap*!* allows us to deal with data as lists in several ways, and lets us convert between text and lists by the usual split and join mechanisms. By making use of these blocks we could, for instance, read a point cloud ASCII data file, split it by lines to obtain coordinates, and split those by spaces to obtain each individual component. Telling the beetle to visit each one of these points and drop a solid would then result in the point cloud geometry showing up in our scene.

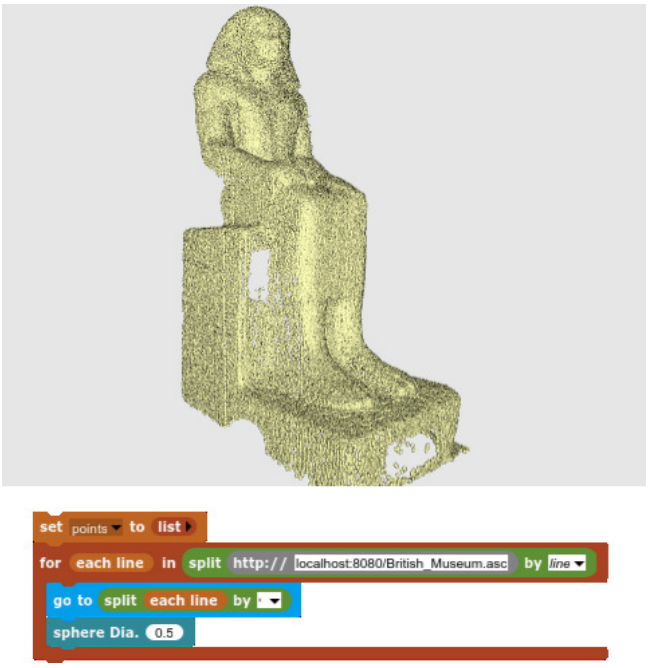

Figure 12: BM Egyptian Antiquities #114 sculpture, read from a locally hosted point cloud file [7] and reconstructed in 51,096 spheres

3.5 The JavaScript Interface

Snap*!* is implemented in JavaScript, but that does not mean it translates blocks into JavaScript. In the same way

a Lua interpreter is written in C but does not translate Lua programs into C, Snap*!* has its own evaluator and runtime environment, and executes instructions in Snap*!* itself.

However, in the same way that Lua lets programmers write inline C straight into Lua source code, Snap*!* has a JavaScript interface that lets us extend the language beyond what is possible with the standard library.

This opens up a wide range of possibilities, from building simple blocks that open a website in a new browser tab (fig. 13), to adding new shape generators (fig. 14) or even importing full JavaScript libraries and abstracting them into higher order blocks.

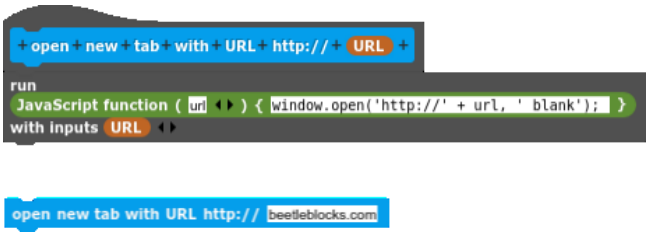

Figure 13: Definition and instance of a block that opens a website in a new browser tab.

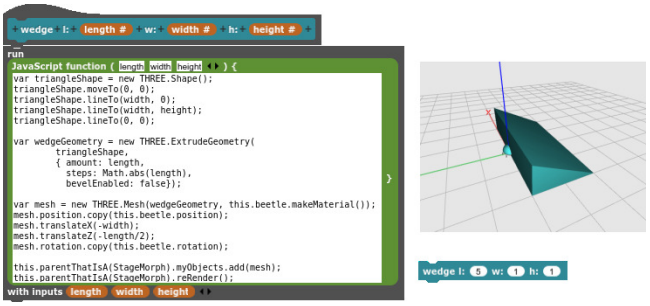

Figure 14: A wedge generator implemented in plain JavaScript

Although Beetle Blocks was designed with 3D fabrication in mind, being built on top of a dynamic, live and concurrent environment allows for different approaches to the system. Even more so given that we can access the underlying JavaScript runtime.

Beetle Blocks makes use of Three.js for all of its 3D-related operations, but builds a very high level interface on top of it so that users do not have to worry about cameras, fields of view, vertices, faces, textures, lighting or clipping planes. Nevertheless, the JavaScript interface opens a window for curious hackers to abuse the system by directly accessing these hidden Three.js objects, modifying their properties and making use of their methods. One could, for instance, make a block that sets the camera position and rotation to be right behind the beetle, then run this block continuously and build a script that lets us move the beetle by means of our mouse and keyboard. Pressing the space key, for example, could toggle between extruding our path or not, thereby allowing us to free draw in space by mimicking the way characters are driven in 3D video games.

Hacking the system in such ways pushes the limits of what is possible in Beetle Blocks, and it lets us produce shapes

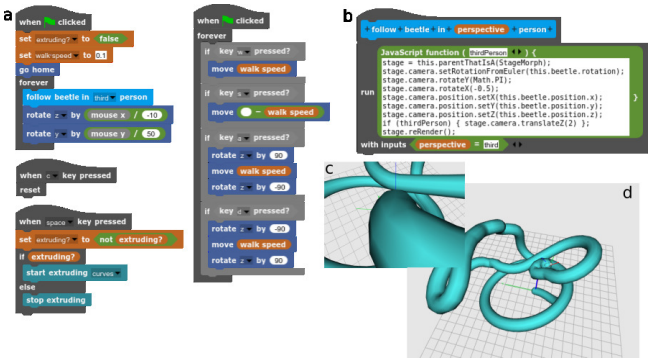

Figure 15: Program that lets us draw free shapes by moving the beetle by means of mouse and keyboard (a), definition of the beetle camera follower (b) and views of an extrusion being drawn (c) and the resulting final shape (d).

that would otherwise be unthinkable in the standard way of generating geometries by composing scripts. One could even go further by storing all these positions and commands into a list that could later be revisited to regenerate our free drawn shapes.

3.6 Input Sliders

One of the key differences between languages in the Scratch family and other block-based systems lies in their emphasis on interactivity. Scratch and Snap*!*, very much in the line of Smalltalk systems, are implemented as full live runtime environments where the language is embedded, whereas other systems like Blockly are built exactly the other way around. This approach allows Snap*!* to implement widgets such as live input sliders inspired by the real time widgets Bret Victor makes use of in his interactive demos[22].

In Beetle Blocks, this feature inherited from Snap*!* has become one of the key tools for live coding and, especially, for finding appropriate parameter values for geometries by trial and error.

Live input sliders execute the script they belong to as we slide their handles around, hence making for a very graphical and interactive way of grasping what kind of results we can obtain from different values, and what these numbers mean in the context of our particular scripts. Moving a slider that controls the amount of iterations in a *repeat* block can give us hints on what the program is doing at each step, and playing with the rotation of a particular axis in real time can help us find out whether it is the x, y or z axis the one that is controlling how wide a shape becomes.

In the ring generator block we created before, it may not be clear at first sight that the amount of steps is what controls the radius of the ring, or how different a ring may look when we make the spheres that compose it bigger or smaller. In non-live languages, playing with these values would mean having to either recompile or rerun the whole program, whereas in live languages we would just have to rerun a small code snippet.

Live input sliders go one step further and make for a more interactive experience, allowing us to see the result of running a script for different parameter values without even having to type numbers into input slots and restart their scripts.

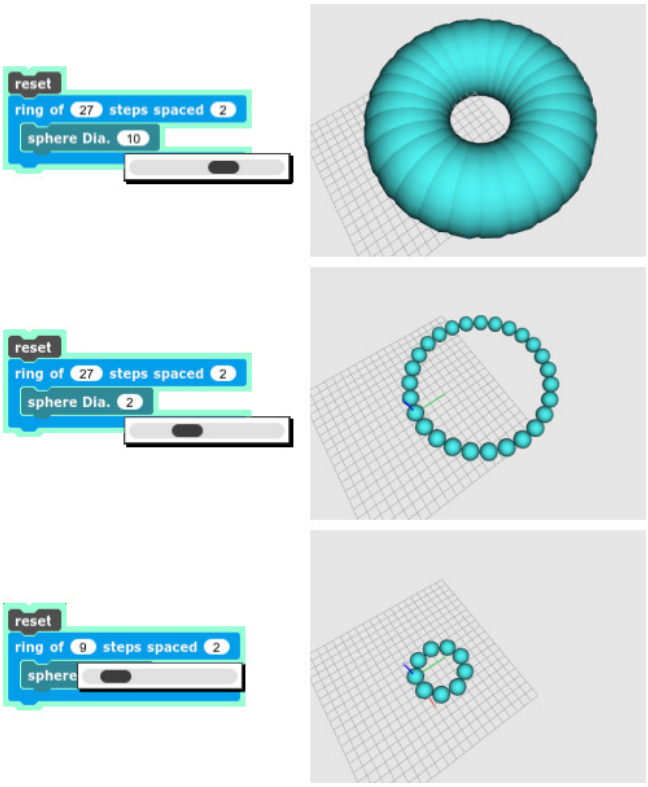

Figure 16: Playing with live input sliders to understand how different diameters for spheres in the ring generator affect the resulting shape (top, middle) and how the amount of steps modifies the radius of the ring (bottom).

4. KEY DIFFERENCES TO SNAP*!*

Although Beetle Blocks is built by reusing a lot of the Snap*!* environment, it presents several substantial differences. We explain the rationale behind these omissions and modifications.

4.1 A Single Beetle

Snap*!* takes from Scratch the idea of building worlds out of multiple programmable objects called *sprites*. These objects can interact with each other according to several events, including global messages that can be intercepted by any object. All these sprites share a common space called *Stage* that is also programmable. This, added to the fact that sprites and stage can take on multiple appearances, makes these environments perfect for building several types of projects, ranging from animations and interactive stories to games and simulations.

The idea behind Beetle Blocks was not to add 3D capabilities to Snap*!*, but to create an environment with one single purpose focused on generating three-dimensional geometries by programming, and for this reason it did not make much sense to inherit the concept of multiple beetles with different appearances that would interact with each other.

Another reason for keeping Beetle Blocks simple by featuring a single programmable object was the fact that adding a third dimension to the Snap*!* world had increased the complexity of moving objects in space and having them generate shapes.

Allowing the beetle to change appearances did not make sense to the purpose of the system either. Beetle Blocks programs always have the very definite objective of generating shapes, and the fact that the shape builder takes the appearance of a beetle is irrelevant to this objective in the same way that the triangular turtle shape is superfluous in the graphical versions of LOGO. The shape of sprites in Scratch and Snap*!* is part of the result of the program, whereas the shape of the beetle object in Beetle Blocks is just a helper that graphically shows us the state and position in space of a programmable shape generator.

Besides supporting multiple programmable objects, both Scratch and Snap*!* also give users the possibility of cloning these objects. Cloned objects keep on doing what their original instances were doing at the time of duplication, and can be given extra behavior upon creation by means of special *When I start as a clone* hat blocks.

Since Beetle Blocks was to be a single object environment, we considered it did not make sense to keep any of the clone-related features either, although being able to clone the beetle would allow Beetle Blocks to tackle recursivity in an additional and entirely different way by delegating behavior to clones, as one can do in Snap*!* (fig. 17).

It is not clear whether this approach to recursion presents any educational or computational advantages to the one presented in section 3.3, which is why we may decide to include it in future releases and study what users make out of it.

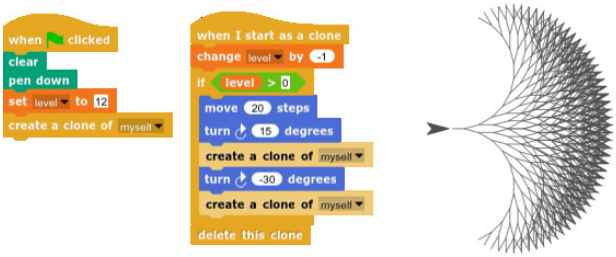

Figure 17: A recursive tree drawn in Snap*!* by delegating behavior to clones.

Although we have removed the ability of objects to switch appearances, we are considering to, in the near future, provide the system with a way to import static images and 3D objects that can serve as references around which geometries can be built. This way one could, for instance, import the 3D scan of a cell phone and design a cover around it without having to take measurements in the real world. One could also import the 2D blueprints of an object and bring its design to completion by using them as reference drawings.

4.2 User Interface

In the aim of building a bridge between programming and 3D design, we have modified the Snap*!* graphical user interface to try to fit users of both worlds. These modifications have been done in a gradual way and often according to suggestions made by students of the Design & Computation course by Duks Koschitz at the Pratt Institute. During the fall of 2014 and spring of 2015, students in this course have also been our main testers, providing us with comments on our design decisions and ideas.

The first quick port of Beetle Blocks to Snap*!* was based on the Scratch extension prototype by Eric Rosenbaum. It added a Three.js canvas into the stage, along with a new category where all new blocks resided (figure 18, top). Scratch extensions are built in JavaScript, which made porting it into Snap*!* an easy task.

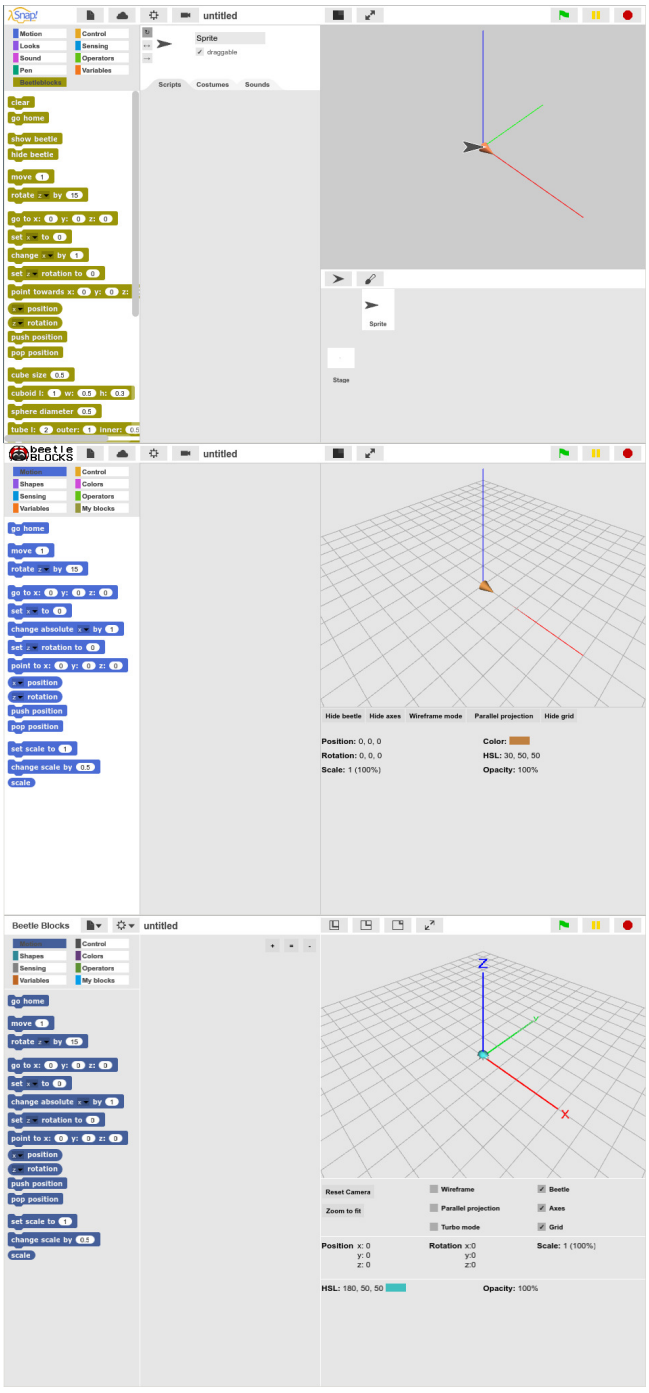

Figure 18: Evolution of the Beetle Blocks interface. Top to bottom: November 2014, April 2015 and June 2016.

In the early versions, we added a new *camera* menu at the top bar that held functions related to the 3D viewport, such as resetting the camera position, changing the background color, or configuring the properties of a helper grid.

The next items to be redesigned would be the block categories and their corresponding block palettes. All blocks and categories that had to do with sprites or media were removed, whereas blocks related to programming constructs and computer interaction were kept in their original categories. All blocks in the old *Beetleblocks* category were spread out across multiple categories according to their functions.

We then began to eliminate all components superfluous to the Beetle Blocks idea of a single object world. We decided to remove everything in the central part of the interface to give more room to the scripting area, where programs are built. The bottom part of the stage, originally dedicated to managing the stage and sprites, was also removed completely, which resulted in a new empty space where we eventually decided to add controls for view-related functions and beetle state monitoring (figure 18, middle).

In the current interface (figure 18, bottom) we have redesigned the space below the stage by turning all buttons into checkboxes and organizing the beetle state monitor into more meaningful sections. With the objective of making all options easy to find and not spread among too many locations, the camera and cloud menus at the top bar have also been removed, and their options have either been reassigned to the other two menus at the top bar or turned into checkboxes below the stage. The original Snap*!* menus at the top bar are very crowded with options and functionalities that Beetle Blocks does not need because of its much narrower domain, which gave us the opportunity to compress the rest of the menu items while still keeping them short and concise.

The block color schema has gone through many iterations along the redesign process, and is in fact still under active discussion, as colors cannot represent concepts such as movement or program flow control. The Snap*!* color schema is inherited from the one Scratch uses, and was designed to be attractive to children and bear enough differences between blocks belonging to different categories to help users find them easily[9]. In Beetle Blocks, our target audience is considerably older than in Scratch, which is why we settled for less saturated colors. We chose to keep their hues close to the originals in categories that Beetle Blocks shares with Scratch and Snap*!*, as many of our users come from these two communities and are already accustomed to these.

It is worth noting that the first beetle had the shape of a cone as a nod to the triangle that represented the LOGO turtle[21], but being a revolution solid, a cone stays the same when rotated around its x axis, and for this reason we decided to turn it into an actual beetle.

Future evolutions of the interface may feature stateful icons instead of labeled checkboxes in the area below the stage. Once the social platform is released, we will have to design new components and redesign some of the existing ones.

5. SOCIAL PLATFORM

Snap*!* has a cloud system that allows users to save and retrieve projects, where one can also mark a project as public and obtain a URL to share it with others. In the Snap*!* community, users send projects to each other by means of email or social networks, but there is no central project repository or meeting hub that gathers all users and projects together.

As a Snap*!* derivative, Beetle Blocks has benefited from the Snap*!* cloud system since the beginning of 2015, when it was made available to forks and modifications. This system has proven very useful to us, but its lack of API calls to retrieve lists of users and their public projects, along with the necessary infrastructure to allow users to *like* and comment on projects, makes it unsuitable for building a social platform around it.

5.1 Node.js Prototype

When faced with the perspective of having to build an entire new system, we decided to begin by implementing an intermediate solution that reused what the Snap*!* cloud was already offering and just added the missing features for the social platform to be feasible.

The Snap*!* cloud offers an HTTP API that exposes some functionality to the outside, but it does not have any endpoints to access projects that users mark as public unless we already know the project name and user name, or unless we have access to credentials of said user. There is no way to know the name of all public projects by a user unless we are that user.

Snap*!* is free software licensed under the Affero GNU Public License v3, but its cloud is a proprietary system. This meant that we could not add functionality to its API, so we had to envision a way to work around its shortcomings. The solution we settled on was to modify the Beetle Blocks interface to the cloud so that each time a user chose to share a project it would also send a request to a different API of ours. This request would contain the user name, project thumbnail, project name and public URL.

With this information, we had everything we needed to build a showcase site for Beetle Blocks, and although this approach would not give us information on previously created and shared projects, we would at least be able to start collecting projects that our users were sharing from then on and gauge the size of our growing community.

This first prototype (figure 19) was quickly sketched in the Express web framework for Node.js. We designed a simple API that exposed a few API endpoints and stored data into a PostgreSQL database. The amount of data we needed to handle was minimal, as the bulk of it was still being dealt with by the Snap*!* cloud system.

Even though the system worked, it was still just a project showcase and very far from a social platform. It was clear that we needed a bigger infrastructure that let users *like* and comment on projects, allowed them to fork projects created by others and was designed from the beginning to be shared by both the social platform and the Beetle Blocks editor, so that users could jump back and forth between the two without the need of additional credentials.

We thereby decided to implement a free cloud system from scratch that would fit our needs and, potentially, also the needs of other projects such as Snap*!* itself.

5.2 The Beetle Cloud

When faced with the task of designing a project sharing social site for our community, we took inspiration from the Scratch website[1][2]. Beetle Blocks belongs to the same family of languages as Scratch, and so the ideas behind its sharing site mostly apply to our case as well. Additionally, the Scratch site is a huge project that has been being used by millions of users around the world for several years, having

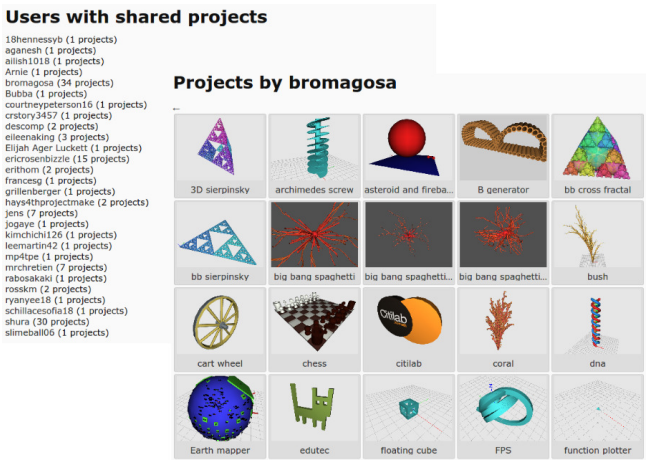

Figure 19: The only two views of the first Beetle-Cloud prototype, showing a list of users with public projects and a list of projects by one of these users. November 2015.

consequently undergone heavy public scrutiny and extensive user experience testing.

The social platform for Beetle Blocks would essentially need to implement some of the basic features of the Scratch community site, namely user and project pages, lists of featured projects, lists of users, lists of public projects by each user, ability to *like*, comment, and fork projects created by others, and shared credentials with the Beetle Blocks environment.

For the new *Beetle Cloud* system and its social platform, we settled on Lapis, a dynamic lightweight server-side web framework for Lua, a solution that made sense for our needs given that our social site does not make heavy use of real-time data, nor does it need instant synchronization with its backend or vice-versa.

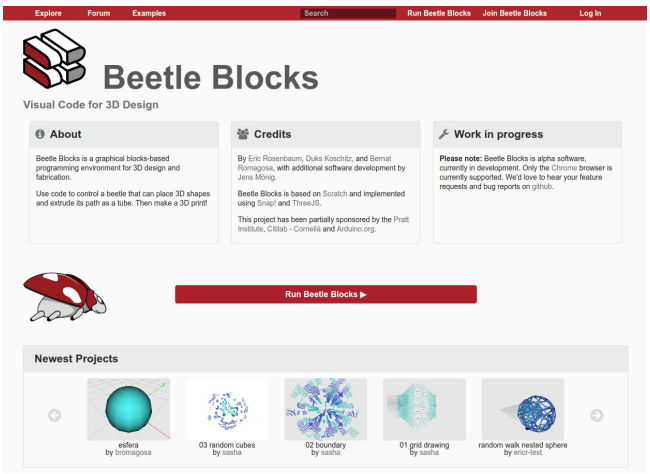

Figure 20: Landing page of the new Beetle Blocks social web platform in the works

For storage, we decided on PostgreSQL because of its proven scalability and the fact that the Lapis framework provides an automatic synchronization mechanism that maps

between PostgreSQL table rows and Lua tables.

The system is currently in active development and testing, and we have so far already implemented the storage backend along with its REST API, and a first version of the social site is being built around it. We have deployed a test version of Beetle Blocks that effectively makes use of the new backend system and can successfully handle the same functionalities that the original Snap*!* cloud system offered, and a first public release of the whole system is expected to be presented at the forthcoming Scratch@MIT conference in August 2016.

6. SUMMARY AND CONCLUSIONS

The Beetle Blocks approach to 3D geometry by means of real-time, dynamic, blocks-based programming is innovative for both the world of design and the world of computation. We have found the powerful ideas of computing in Snap*!* to be a perfect match for Beetle Blocks. Snap*!* provides straightforward abstractions for computational constructs that have supplied Beetle Blocks with ways to produce very intricate shapes in an elegant and natural way. Shapes such as fractals and other recursive or repetitive structures are examples of this.

We have found our simplification of the Snap*!* world into a single-object environment to help tame the complexity of three-dimensional spatial operations. However, we do not rule out providing Beetle Blocks with the object-cloning capabilities found in Scratch and Snap*!* in the future, as they may bring certain benefits to tackling recursivity from a different perspective.

The Scratch project illustrates how important it is for learners to be able to remix and share projects with each other. We expect our new cloud system and social platform to serve as a learning hub for Beetle Blocks users. The social platform should also help us understand how our users are building their programs and whether the abstractions provided by Snap*!* and Beetle Blocks are indeed adequate to the purpose of designing 3D objects.

7. ACKNOWLEDGMENTS

This work was partially supported by the Pratt Institute, Citilab-Cornellà and Arduino.org.

We would like to thank Jordi Delgado Pin for the counseling and academic supervision of this article. Jordi Delgado is both a consultant and teaching collaborator at UOC, and a senior lecturer in the computer science department at the Barcelona Faculty of Informatics, UPC.

8. REFERENCES

[1] K. Brennan and M. Resnick. Imagining, creating, playing, sharing, reflecting: How online community supports young people as designers of interactive media. In *Emerging Technologies for the Classroom: A Learning Sciences Perspective*, pages 253–268. Springer, New York, September 2013.

[2] S. Dasgupta, W. Hale, A. Monroy-Hernández, and B. M. Hill. Remixing as a pathway to computational thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, pages 1438–1449, New York, NY, USA, 2016. ACM.

[3] E. Eastmond. Designblocks.

[4] N. F. et al. Blockly: A visual programming editor, 2013.

[5] B. Harvey and J. Mönig. Snap*!* reference manual.

[6] B. Harvey and J. Mönig. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists? In *Constructionism Conference*, Paris, 2010.

[7] INSIGHT, Cain, Martinez, and Munn. A sculpture in the british museum (bm egyptian antiquities #114), March 2000.

[8] R. P. Institute. Csnap. https://community.csdt.rpi.edu/.

[9] N. R. B. S. John Maloney, Mitchel Resnick and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4), November 2010. Article no. 16.

[10] S. B. Z. D. E. B. G. H. Joost Nieuwenhuijse, René K. Müller. Openjscad. http://openjscad.org/.

[11] J. Y. Katy Hamilton and M. Minuti. Blockscad. http://blockscad.com/.

[12] A. C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, Mar. 1993.

[13] A. C. Kay. Squeak etoys, children & learning. 2005. VPRI Research Note RN-2005-001.

[14] M. Keeter. Antimony. http://www.mattkeeter.com/projects/antimony/.

[15] M. Kintel and C. Clifford Wolf. Openscad, the programmers solid 3d cad modeller, 2011.

[16] E. Klopfer, H. Scheintaub, W. Huang, and D. Wendel. Starlogo tng. In *Artificial Life Models in Software*, pages 151–182. Springer, 2009.

[17] D. Koschitz and E. Rosenbaum. Exploring algorithmic geometry with "beetle blocks:" a graphical programming language for generating 3d forms. In *15th International Conference on Geometry and Graphics Proceedings*, Montreal, August 2012. International Society for Geometry and Graphics. Paper no. 102.

[18] S. Papert. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, Inc., New York, 1980.

[19] M. Petts. Life after turtle geometry with a 3d logo microworld. *Mathematics in School*, 17(5):2–7, 1988.

[20] E. Rosenbaum. Scratch for second life. http://web.mit.edu/~eric_ r/Public/S4SL/.

[21] C. Solomon. Logo, papert and constructionist learning. http://logothings.wikispaces.com/.

[22] B. Victor. Learnable programming, September 2012. http://worrydream.com/LearnableProgramming/.

[23] A. Yeh and R. A. Nason. Vrmath: A 3d microworld for learning 3d geometry. In *World Conference on Educational Multimedia, Hypermedia & Telecommunications*, Lugano, Switzerland, 2004. Association for the Advancement of Computing in Education (AACE).