

The Bricklayer Ecosystem - Art, Math, and Code

Victor Winter

Department of Computer Science
University of Nebraska at Omaha
Omaha, NE 68182, USA
vwinter@unomaha.edu

Betty Love

Department of Mathematics
University of Nebraska at Omaha
Omaha, NE 68182, USA
blove@unomaha.edu

Cindy Corritore

Creighton University
Omaha, NE 68102, USA
cindycc@gmail.com

This paper describes the *Bricklayer* Ecosystem – a freely-available online educational ecosystem created for people of all ages and coding backgrounds. *Bricklayer* is designed in accordance with a “low-threshold infinite ceiling” philosophy and has been successfully used to teach coding to primary school students, middle school students, university freshmen, and in-service secondary math teachers. *Bricklayer* programs are written in the functional programming language SML and, when executed, create 2D and 3D artifacts. These artifacts can be viewed using a variety of third-party tools such as LEGO[®] Digital Designer (LDD), LDraw, Minecraft clients, Brickr, as well as STereoLithography viewers.

1 Motivation

Global interest in teaching coding across the entire K-12 spectrum is rapidly increasing. A number of countries in the EU, including England [3], currently teach computational thinking and coding starting in kindergarten. A noteworthy recent addition (2016) to this list is Finland, whose educational system is consistently ranked among the highest in the Western world and has been touted as taking coding education farther than any country in Europe [18]. On the other side of the globe, Australia now also teaches coding across the K-12 spectrum. Australia is interesting because it has made time for coding in its curriculum by eliminating history and geography [2].

The educational landscape in the United States is less unified. However, progress is being made. Cities such as San Francisco, New York, and Chicago have plans to teach coding across the K-12 spectrum. Recently, Rhode Island has rolled out a comprehensive effort, called CS4RI, to teach coding across the entire K-12 spectrum statewide [12].

The pursuit of coding in the US is also being strongly encouraged at the national level. In his 2016 State of the Union Address, President Obama presented his *Computer Science for All* initiative that calls for \$4 billion in funding for K-12 computer science education [14]. This interest in computer science is reflected in similar initiatives across the United States and around the world [11, 8, 19]. The purpose is not only to provide students with the programming skills needed for high-tech careers, but more generally to provide them with the computational thinking skills they will need for careers in a technology-driven world.

The resulting educational environment has given rise to an increased exploration and development of educational tools and systems whose purpose is to teach coding within the K-12 spectrum. The *Bricklayer* ecosystem is one such system.

The remainder of the paper is as follows: Section 2 gives an overview of some existing educational tools and systems. Section 3 describes how *Bricklayer* has evolved since its debut at TFPIE 2014. Section 4 describes the components that make up the *Bricklayer* ecosystem. Section 5 describes some of the settings in which *Bricklayer* is being used, and Section 6 concludes.

2 Related Work

This section describes 3 major educational systems targeting K12 education. Each system offers a distinct perspective on how to effectively engage K-12 students in coding. We omit discussion of systems such as Alice, Snap!, Greenfoot, Media Computation, Picturing Programs, Yampa, and Soccer-fun all of which target audiences outside the K-12 range (e.g., high school and above).

Scratch [10] is a visual programming language and environment designed to teach programming to young people (8-16) especially those who have no previous programming experience. Building on the ideas of Logo, Scratch seeks to create a constructionist learning environment where young people can pursue projects that are both personalized and meaningful. Learning is achieved through experimentation and peer sharing rather than traditional instruction. To accomplish these goals, the Scratch language has been designed so that it is virtually impossible to create an ill-formed program. They describe the rationale for this design decision in terms of the following LEGO[®] metaphor: “*The [LEGO[®]] brick shapes suggest what is possible, and experimentation and experience teaches what works*”[10]. On the syntactic level language constructs are colorful blocks whose shape govern their composition. On the semantic level, Scratch employs data coercion and pre-defined ranges for data values to virtually eliminate all runtime errors. The result is a programming environment that encourages *tinkering* with programs. This mindset is further enhanced by Scratch’s *liveness* property which enables users to observe the functionality of a block or program fragment by simply clicking on it and to modify programs during runtime.

Bootstrap [13] is a programming language, environment and curriculum designed primarily for middle and high school students. Programs are written in a subset of Racket, a functional programming language, and focus on the creation of video games involving four principle abstractions: (1) a player, (2) a target, (3) a danger, and (4) a background. The video games created are deconstructed into a sequence of frames that center around a player trying to acquire targets while avoiding dangers – all of which can change from one frame to the next. Programming involves the creation and modification of functions that define the rules of the game and govern the behavior of the player, targets, and dangers. A platform called the *World framework* [7] is responsible for creating frame sequences (i.e., a video game) from a given set of functions. In this way, programmers are shielded from having to use looping constructs.

Bootstrap coding is designed to achieve very precise learning objectives, namely, to improve abilities in solving algebra word problems of the kind found in standardized algebra tests given to middle school students in the US. The Bootstrap philosophy is based on the premise that in order for transfer of learning to occur between domains (e.g., coding and algebra) study must be carefully guided. Towards this end, students are taught how to use a process called the (*Bootstrap*) *Design Recipe* [6] to design and develop the functions that underly their video games. This approach has strong ties to key concepts defined in algebra learning standards.

It is worth mentioning that the abstractions of functional programming languages (e.g., variables and functions) are much more closely aligned with their mathematical counterparts than their imperative language counterparts. Thus, along this dimension, functional languages are better positioned for transfer of learning between the coding and algebra domains.

CodeSpells [4][5] is an extensible educational platform designed to teach Java programming in the context of video game play (a JavaScript/Blockly version was recently made available on Steam). CodeSpells targets learning spaces outside of the traditional classroom. The authors believe that it is in such non-institutionalized settings that lifelong passion for coding is most likely to develop.

In contrast to educational systems like Bootstrap, which focuses on *building* video games, the focus of CodeSpells is on *playing* a video game. More specifically, CodeSpells is a non-competitive role play-

ing game in which the student is a wizard in a 3D world populated by gnome-like creatures. The game provides 7 primary spells which include levitation of objects, fire, and flying. Programming consists of modifying and creating spells, which are written in Java. Game play is largely unstructured, though quests can be undertaken to earn badges. The first level of the game introduces students to Java basics like method parameters, conditional statements, and iteration.

Codespells is suitable for both primary and secondary school students. The authors report that the immersive and role playing aspects of CodeSpells has a number of beneficial effects. Foremost among these are (1) countless hours – video games have an ability to engage players for hours on end, and (2) positive attitude toward failure – within the context of game play (a non-institutional setting) when a student creates code containing an error they do not see themselves as having done something *wrong* – rather, such errors are seen as part of the game.

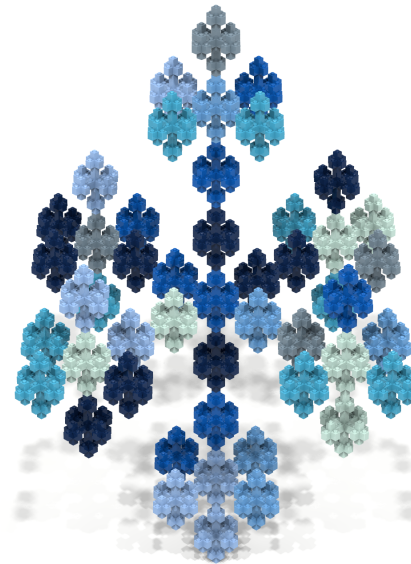
3 The Bricklayer Ecosystem

At its core, *Bricklayer* is a library of graphical primitives, written in SML, which can be used to create pixelated artifacts. A *Bricklayer* program is a program, written in the functional programming language SML, which references elements (e.g., functions) belonging to the *Bricklayer* library.

LEGO[®] Digital Designer (LDD), a freely available tool provided by The LEGO[®] Group, is the default viewer for the pixelated artifacts created by *Bricklayer* programs. Figure 1 shows LEGO[®] artifacts created by *Bricklayer* programs and rendered using the Persistence of Vision Raytracer (POV-Ray)[1] tool.



(a) A pixel art image of Elvis Presley.



(b) The dual of the Menger sponge.

Figure 1: Artifacts created by *Bricklayer* and rendered using POV-Ray.

3.1 Then

The *Bricklayer* library was debuted at TFPIE 2014 [20]. At that time, the library consisted of predicates, iterators, and around 30 LEGO[®] bricks. An abstraction called a *brick function* was also provided which is similar to a predicate except that its return value is a LEGO[®] brick rather than a Boolean value. In 2014, the construction of *Bricklayer* artifacts was limited to compositions involving these primitives.

Though highly expressive from a theoretical standpoint, the sophistication of these abstractions presented a barrier to students in the K-12 spectrum who wanted to learn how to code. For example, it is simply too much to expect elementary school students to have fluency in Boolean algebra in order to create *Bricklayer* artifacts. To address such limitations, a major redesign of *Bricklayer* was undertaken with the goal of making *Bricklayer* more accessible across the K-12 spectrum. It is this redesign that is the focus of this paper.

3.2 Now

At present, the *Bricklayer* library consists of five coding levels (with two more levels in the design stage). Levels 1–3 provide primitives for creating two dimensional artifacts, and levels 4–5 provide primitives for creating three dimensional artifacts. The predicates, iterators, and brick functions that were available in the version of *Bricklayer* reported on in TFPIE 2014 now reside in level 5.

In addition, the new *Bricklayer* library is bundled with a collection of interactive web apps, online videos, documentation, coding examples, and exercises whose collective purpose is to help people of all ages and coding backgrounds learn how to write *Bricklayer* programs. We refer to this collection as the *Bricklayer* ecosystem – an ecosystem which has been designed in accordance with a “low-threshold infinite-ceiling” philosophy. The *Bricklayer* ecosystem can be found at the following URL.

bricklayer.org

4 Additions to *Bricklayer*

At TFPIE 2014 numerous suggestions were made regarding ways that *Bricklayer* might be extended or enhanced. In this section we give an overview of the extensions (some that were proposed, some not) that we have made to the original *Bricklayer* library as well as to the *Bricklayer* ecosystem.

4.1 Vitruvia

Vitruvia¹ is an interactive web app that is part of the *Bricklayer* ecosystem. In Vitruvia, coding skills can be developed through exercises which require input from the user (via mouse clicks or touch screen selections) and which are automatically graded. These interactive exercises are grouped into *concepts*. Figure 2 shows an example of a manually created solution to an exercise and the automated feedback provided by Vitruvia.

Presently, Vitruvia contains 23 concepts (more are in the design stage) which provide a gentle learning curve accessible to people of all ages and backgrounds. In our TFPIE 2014 paper we discussed characteristics of effective example-problem sequences and asserted that the graphical domain of *Bricklayer*, which we described as being *example rich and problem dense*, was well suited to the creation of such sequences [20]. The concept sequence in Vitruvia is an instance of such an example-problem sequence.

¹Legend has it that Marcus Vitruvius had a daughter named Vitruvia.

Vitruvia

Concept 1
Concept 2
Concept 3
Concept 4
Concept 5
Concept 6
 • Demo 1
 • Exercise 1
 • Exercise 2
 • Exercise 3
 • Exercise 4
 • Exercise 5
 • Exercise 6
 • Exercise 7
 • Exercise 8
Concept 7
Concept 8
Concept 9
Concept 10
Concept 11
Concept 12
Concept 13
Concept 14
Concept 15
Concept 16
Concept 17
Concept 18

Concept 6 Exercise 2

```
Execute the following Level 1 program by placing tiles at the appropriate locations.  
  
put2D_2x2_RED(0,0);  
put2D_2x2_RED(2,2);
```

4					
3			RED	RED	
2			RED	RED	
1	RED	RED			
0	RED	RED			
	0	1	2	3	4

Display Co-ordinates:

BLUE GRAY GREEN
RED WHITE
EMPTY YELLOW

(a) Solution submitted by student.

Vitruvia

Concept 1
Concept 2
Concept 3
Concept 4
Concept 5

Concept 6 Exercise 2

(b) Automated feedback provided by Vitruvia.

Figure 2: An example of a Vitruvia exercise.

A brief overview of the Vitruvia concept sequence follows (for a more detailed discussion see [21] or visit bricklayer.org/vitruvia). Concept 1 establishes an understanding of two dimensional coordinate systems. Exercises involve the placement of 1×1 LEGO[®] bricks² (aka “bit bricks”) in cells having specific coordinates. Concept 2 briefly touches on property-based construction of artifacts. For example, the even/odd property of coordinates can be used to place black/white bricks thereby creating a checkerboard pattern. Concepts 3–4 establish a natural language vocabulary for specifying the placement

²Technically the dimension of such bricks is $1 \times 1 \times 1$.

and orientation of LEGO[®] bricks having standard shapes. Concepts 5–6 establish a translation between natural language specifications and *Bricklayer* function calls. More specifically, at this stage artifacts are constructed through a sequence of calls to *Bricklayer* functions that “put” bricks having standard shapes and colors at specified coordinates. Concepts 7–9 focus on *overwriting* (the result of brick collisions) as well as debugging basics. Concepts 10–12 introduce *function declarations* and *offsets*. Concept 13 introduces the *ring* and the *circle* as well as the idea of *curried functions*. Concept 14 improves the “put” function by parameterizing it on shape, brick type, and coordinate. It is now possible for a single “put” function call to create a rectangle of arbitrary dimensions. Concept 15 introduces a function for creating “smooth” lines. Concept 16 discusses clipping and how it can be used to advantage within a program. Concept 17 introduces a function called *setMySpace2D* which enables partitioning of *Bricklayer*’s virtual space. This partitioning enables clipping to be used in more creative ways and also opens up the possibility of creating multi-user artifacts. Also, artifacts containing interlocking rings, like the Olympic symbol shown in Figure 3, can be created using *Bricklayer*’s ring and *myspace* functions.

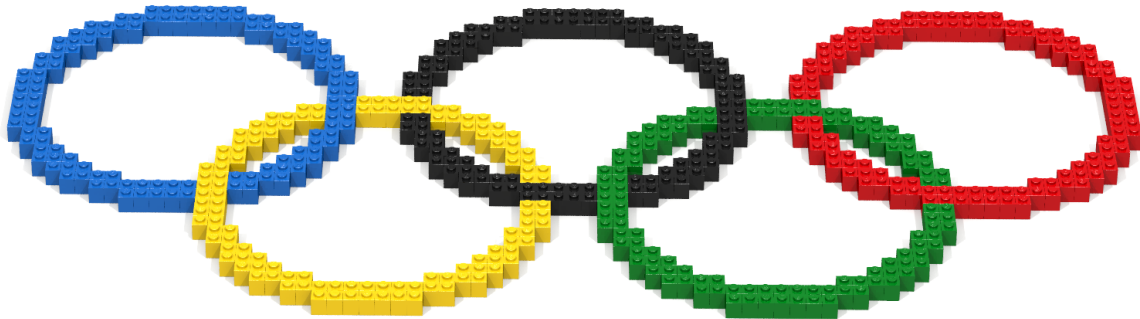


Figure 3: The (interlocking) Olympic rings created using rings, clipping and *setMySpace2D*.

Concept 18 introduces the *let-block* as a mechanism for structuring code. Concept 19 transitions the programmer into three dimensional space and introduces the three dimensional version of the “put” function. Concept 20 focuses on *conditional expressions* and *brick functions*. Concept 21 focuses on nested conditional expressions as well as the logical operator *andalso*. Concept 22 focuses on nested conditional expressions as well as the logical operator *orelse*. Concept 23 introduces the *traverseWithin* iterator.

Vitruvia is extensible and we have plans for additional concepts as well as other enhancements such as the creation of leaderboards and intelligent testing.

4.2 Minecraft

The *Bricklayer* library is integrated with Minecraft through a CanaryMod plugin called RaspberryJuice. This plugin is a PC implementation of the key functionality of the Minecraft: Pi Edition modding API (the Pi Edition is a free version of Minecraft developed by Mojang for the Raspberry Pi computer). The RaspberryJuice plugin provides a function, called *setBlock*, capable of placing a block at a particular position in a Minecraft world. In order for *setBlock* to work, the Minecraft world in which the block is to be placed must be up and running on the CanaryMod server. Another important RaspberryJuice function is called *setPos* which enables a player to be placed at a particular location in a Minecraft world.

It is primarily through the RaspberryJuice functions `setBlock` and `setPos` that *Bricklayer* interacts with a Minecraft world (residing on a CanaryMod server).

An artifact residing in *Bricklayer*'s virtual space can be built in (i.e., exported to) a Minecraft world running on a CanaryMod server by simply changing the output function of a *Bricklayer* program from `show` (which outputs the artifact to LDD) to `showMC`. The `showMC` function will cause *Bricklayer* to create a Python program consisting of one `setPos` function call, to position your avatar (i.e., player), followed by a sequence of `setBlock` function calls, one function call for each cell in *Bricklayer*'s virtual space occupied by a piece. This Python program is written to a file which is then automatically executed using a system-level command. The resulting artifact can then be seen using a Minecraft client – provided it is connected to the CanaryMod server. From a conceptual standpoint, that is all there is to it. The rest is engineering. Figure 4 shows a Minecraft world populated with several *Bricklayer* artifacts.

In addition to `showMC`, the *Bricklayer* library provides several functions facilitating the construction of Minecraft artifacts. For example, functions are provided for (1) establishing correspondences between the coordinate system of the *Bricklayer* virtual space and the coordinate system of Minecraft, (2) specifying Minecraft-specific coordinate offsets, and (3) positioning *Bricklayer* artifacts at specific Minecraft coordinates.



Figure 4: *Bricklayer* artifacts in Minecraft: Worlds collide.

It should be noted that artifacts viewed in LDD are “stateless” in the sense that if an artifact does not appear as desired one simply closes LDD, modifies the *Bricklayer* program, executes the new program, and views the modified artifact (i.e., the previously viewed artifact no longer exists). In contrast, artifacts viewed in Minecraft are “stateful” – they remain in the Minecraft world until they are explicitly erased. To facilitate the artifact development cycle in Minecraft, the *Bricklayer* library provides a function for erasing *Bricklayer* artifacts from a Minecraft world.

Because of “statefulness” and the overhead (e.g., explicit user actions) associated with Minecraft client viewing, a feedback loop involving the design of an LDD artifact is much simpler (and faster) than

the feedback loop involving the design of a Minecraft artifact. For this reason, it is highly beneficial to prototype artifacts in LDD before moving them to a Minecraft world. In this way, the concept of prototyping naturally finds its way into *Bricklayer* coding.

There is a problem though: LDD has difficulty displaying artifacts containing more than 25K pieces³. In contrast, *Bricklayer* artifacts can be constructed in Minecraft worlds containing up to 450K pieces. This difference naturally motivates the parameterization of an artifact on its size. The construction of smaller artifacts can be prototyped using LDD and the size of the artifact can be increased when it is moved to a Minecraft world.

Currently, the *Bricklayer* library supports 97 LEGO[®] bricks and 343 Minecraft blocks. Default mappings are provided between LEGO[®] bricks and Minecraft blocks. This allows artifacts to be fluidly displayed using either LDD or Minecraft with minimal program modification (i.e., simply toggle between the *show* and *showMC* output functions). The *Bricklayer* library also permits users to define custom mappings between LEGO[®] bricks and Minecraft blocks allowing LDD distinctions of Minecraft blocks in an artifact-specific fashion.

4.3 3D Printing

3D printing provides an educational dimension that can be highly engaging. Over the past 2 years the most frequent *Bricklayer* requests/suggestions have been related to 3D printing. Such requests have originated from researchers, teachers, students, and parents alike.

In response to these requests, the *Bricklayer* library has been extended to support the output of artifacts in a STereoLithography (STL) format. The artifacts described by such files can be viewed and 3D printed using viewers such as 3D Builder (Windows OS) and Cura. Figure 5 shows a variety of 3D printed *Bricklayer* artifacts. It is worth mentioning that 3D printed *Bricklayer* artifacts are constructed from cubes (not LEGO[®] bricks).

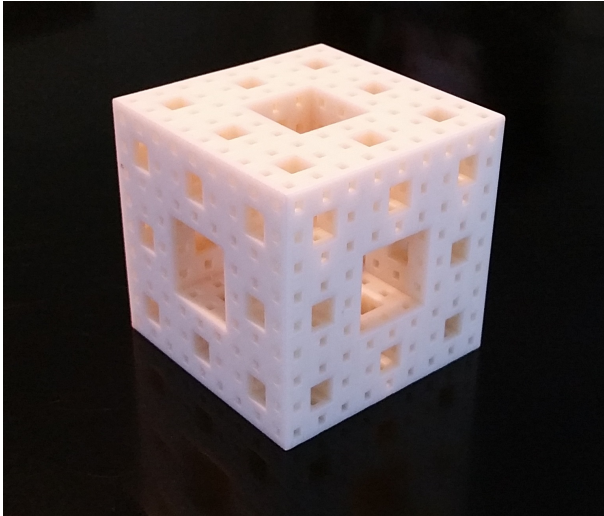
4.4 LDraw

Bricklayer can output artifacts in the LDraw file format thereby enabling artifacts to be viewed using LDraw. The LDraw viewer has capabilities that distinguishes it from the LDD viewer. First, LDraw can display an artifact having 250K bricks – in contrast to LDD which begins to have difficulty when artifacts contain more than 25K pieces. Second, in LDraw it is possible to continuously rotate an artifact around an arbitrary axis (e.g., the x-axis). Though useful, this behavior is not possible in LDD, which only permits unrestricted rotation around the y-axis. The 3D printed Wunderlich cube shown in Figure 5c was originally validated in LDD by rotating the cube around the y-axis. The artifact looked good and was 3D printed. Only later did we realize that the cube we had constructed contained an error – a costly mistake – which would have been revealed had we rotated the cube around the x-axis. A second (and correct) version of the Wunderlich cube was validated using the rotational capabilities of LDraw prior to 3D printing.

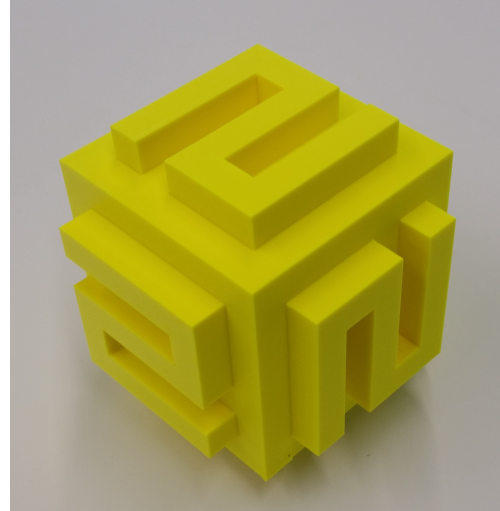
4.5 Brickr

A well-known problem, called *The LEGO[®] Construction Problem*, concerns itself with the construction of 3D objects using a given set of standard LEGO[®] bricks [17]. The starting point for this problem

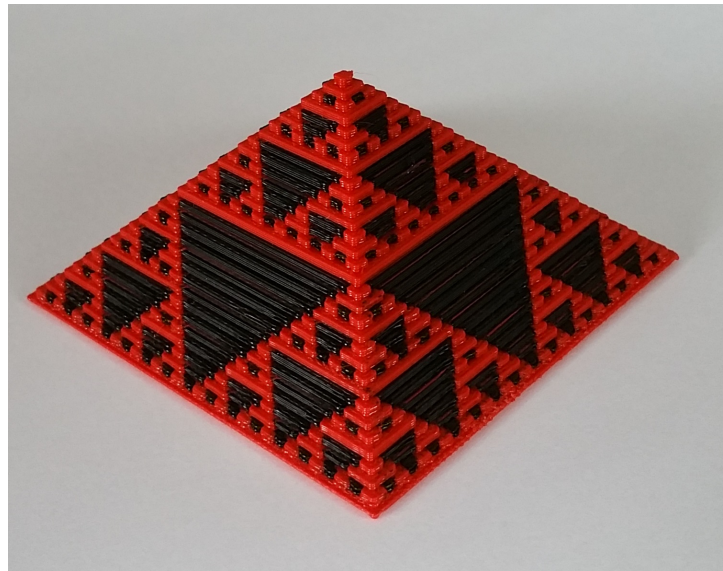
³In special cases, LDD is able to display artifacts having 60K pieces and sometimes even a little bit more. We hope that this limit will be increased in future versions of LDD.



(a) A 3D printed Menger sponge consisting of 8000 bricks.



(b) A Wunderlich cube.

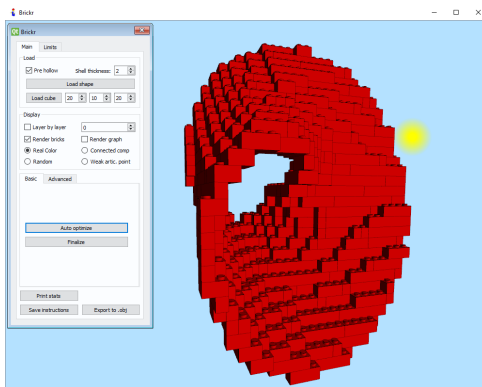


(c) A dual extrusion 3D print of the Sierpinski pyramid.

Figure 5: *Bricklayer* 3D printed artifacts.

is a model of a 3D object consisting entirely of $1 \times 1 \times 1$ LEGO[®] bricks. Such a model is referred to as a “legoized” representation of the 3D object. The goal is to construct a LEGO[®] artifact in physical space corresponding to the “legoized” model. There are two primary objectives the construction must satisfy: (1) the LEGO[®] artifact must have the same shape as the “legoized” model, and (2) the LEGO[®] artifact must hold together under the forces of gravity. Secondary objectives involve issues such as (1) optimization of brick shape usage, and (2) brick orientations to increase the strength of the LEGO[®] artifact.

At this time, all artifacts produced by *Bricklayer* are in “legoized” form. This makes them ideal



(a) A Brickr solution to the LEGO[®] construction problem for the Moebius strip.



(b) A Moebius strip constructed from LEGO[®] bricks.

Figure 6: Using *Bricklayer* and Brickr to create a Moebius strip in physical space.

inputs for tools, such as Brickr[16], that focus on the LEGO[®] construction problem.

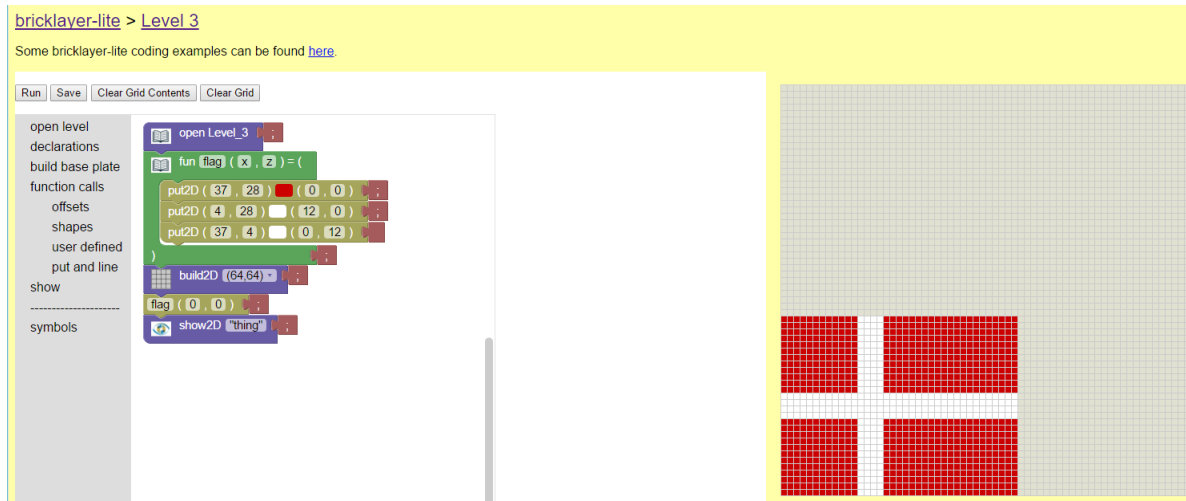
To support integration with Brickr, a special *Bricklayer* output function has been implemented, called *showBinvox*. This function will output an artifact in a binvox format – a file format suitable for input to Brickr. As a result, Brickr is integrated into the *Bricklayer* ecosystem.

In Brickr, the LEGO[®] construction problem can be solved by simply pressing the *Auto optimize* button. Additionally, Brickr has the ability to produce step-by-step instructions showing how to create the resulting LEGO[®] artifact manually. This opens the door to a fascinating problem domain involving the construction of non-trivial mathematical objects including: the Moebius strip, the Klein bottle, the trefoil knot, and minimal surfaces such as the Enneper minimal surface. The feasibility of constructing such creations in LEGO[®] was demonstrated by Andrew Lipson [9][15], a mathematician who earned a PhD in knot theory from Cambridge. Lipson creates C/C++ programs that construct “legoized” models of mathematical objects and then solves the LEGO[®] construction problem manually – which he says is “...where the fun is”. In contrast, we use *Bricklayer* to create a “legoized” model and Brickr to solve the LEGO[®] construction problem. Those who want to share in Lipson’s fun can output Brickr instructions for “legoized” artifacts without first pressing Brickr’s *Auto optimize* button.

Figure 6a shows a Brickr solution to the LEGO[®] construction problem for a “legoization” of a Moebius strip created by a *Bricklayer* program. Brickr instructions for how to manually create this

artifact in LEGO[®] were produced and the authors then (of course) ordered the requisite parts from LEGO[®] and built the LEGO[®] artifact. The result is shown in Figure 6b.

4.6 Bricklayer-Lite



(a) An example of a bricklayer-lite program and the artifact it creates.

```
(* generated by Bricklayer Lite version 0.9 *)

open Level_3;

fun flag (x,z) =
  (
    put2D (37,28) RED (0,0);
    put2D (4,28) WHITE (12,0);
    put2D (37,4) WHITE (0,12)
  );

build2D (64,64);

flag(0,0);

show2D "flag";
```

(b) Auto-generated Bricklayer code produced when running a bricklayer-lite program.

Figure 7: Coding using bricklayer-lite.

Bricklayer-lite is a web app that provides a visual block-based coding environment for creating *Bricklayer* programs. Bricklayer-lite is implemented using the Blockly library developed by Google. Currently, the functions available in Levels 1–3 have been implemented in bricklayer-lite. This was accomplished by defining custom Blockly blocks and then defining their semantics in JavaScript. The semantics includes a state machine which tracks various properties relating to well-formed programs.

When executing a well-formed bricklayer-lite program, the JavaScript semantics include a function

that generates executable *Bricklayer* code corresponding to the bricklayer-lite program. The *Bricklayer* program is displayed in a web form below the bricklayer-lite program. An example is shown in Figure 7. Figure 7a shows a bricklayer-lite program and the artifact it creates. Figure 7b shows the Bricklayer program, auto-generated by bricklayer-lite, corresponding to the program shown in Figure 7a.

Bricklayer-lite runs on any web browser and can be accessed from any device such as a smart phone, tablet (e.g., iPad), or laptop. Bricklayer-lite programs are pictorial in nature and are constructed by fitting together blocks. Depending on the technology used (e.g., laptop, smart phone), blocks can be selected via mouse-clicks or by touch. Some, though very little, typing (e.g., data entry) is required when constructing a bricklayer-lite program. For example, to place a brick at a specific location requires entering, via a keyboard, the coordinates of that location.

Bricklayer-lite programs can be executed by selecting the run button. They can be saved to the cloud and retrieved at a later time as well as from other devices.

4.7 The Lace Animator

The *Bricklayer* ecosystem contains a variety of (pre-coding) un-plugged exercises which focus on the understanding and discovery of algorithms for creating 2D patterns on graph paper. The simplest class of pattern involves the repetition of a set of seed patterns whose size remains constant. We refer to such patterns as *arithmetic patterns*. Exercises involving arithmetic patterns include tessellations of the a 2D plane using pentomino shapes. It should be noted that the spacing of shapes in regular patterns follow arithmetic progressions.

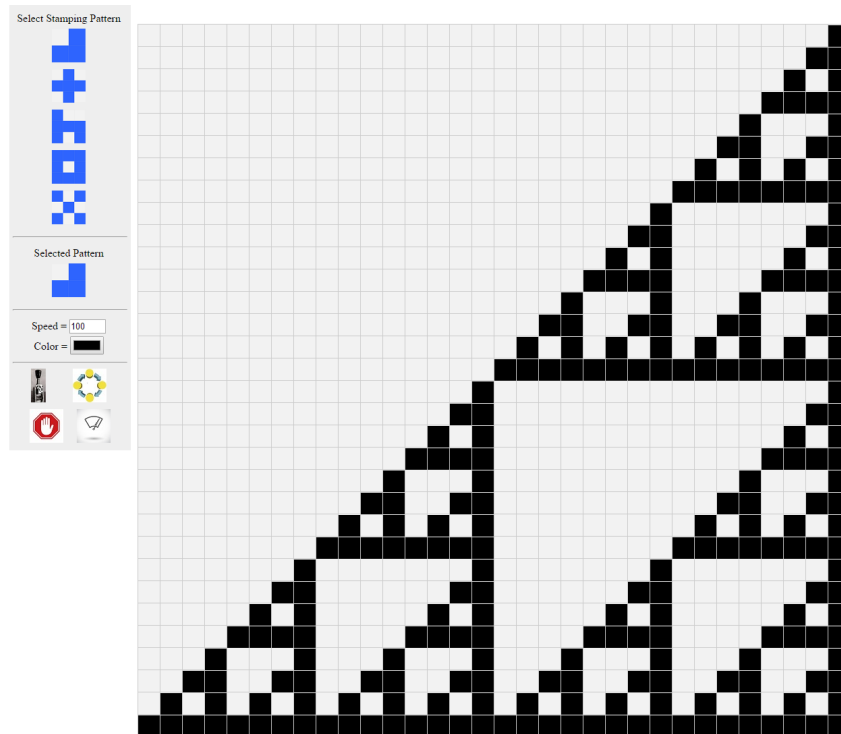


Figure 8: The result of a lace pattern animation.

The next class of pattern we call *geometric patterns*. These are patterns whose spacing follow geo-

metric progressions. As a result, patterns get larger/smaller as they repeat. Fractals fall into this category of pattern. We have coined the term *lace* to refer to a class of geometric patterns created from a seed shape and a stamping pattern. A *lace* is a structure that is connected but not a line.

Empirical evidence suggests that people of all ages have an initial difficulty constructing laces – they have a tendency to employ an arithmetic approach (rather than a geometric approach) to constructing lace patterns. In particular, students want to repeat one pattern over and over without changing its size. To help students understand the nature of the algorithm that underlies lace patterns we have developed an interactive web app called the *Lace Animator* which shows how a variety of laces can be constructed using linear steps (one seed shape at a time) as well as geometric steps (one pattern at a time). Figure 8 shows a lace created using the *Lace Animator*.

5 Outreach

Bricklayer has been used for a variety of K-12 outreach efforts including those listed below.

- Summer Workshops – in the summer of 2014 and again in 2015 *Bricklayer* was used to teach coding to elementary and middle school students. The educational format was a 3 hour/day 5 day workshop. The venue was the Techademy, a summer tech program offered by the College of Information Science & Technology at the University of Nebraska at Omaha.
- Coding Club – An after school *Bricklayer* coding club was held at Rockbrook Elementary School from November, 2014 – May, 2015. The club met once a week for an hour and culminated with a parent’s “Art and Pizza Night” event in which students displayed on the school walls the artifacts they created alongside their code.
- Exhibits
 - NE SciFest – The Nebraska Science Festival is a major annual event in Nebraska hosted by the University of Nebraska Medical Center. Its purpose is to stimulate interest in science. In 2015, *Bricklayer* had an exhibit at the NE SciFest – a 2 day event which was held at the Durham Museum in Omaha. It was estimated that approx 3000 students (mostly from elementary school) were in attendance.
 - Lauritzen Gardens – a botanical center in Omaha periodically hosts a multi-week LEGO® Artists Expo where artists present life-size LEGO® sculptures. In 2015, during this expo, *Bricklayer* was invited to have a 1-day exhibit on the LEGO® robotics day.
 - Omaha Children’s Museum – Beginning in May 2015, the Omaha Children’s Museum had a multi-month LEGO® exhibit. *Bricklayer* was invited to hold an exhibit on the opening day of the Museum’s LEGO® exhibit.
- CoderDojo – In the summer of 2015, *Bricklayer* was used in a topics in mathematical computing course for in-service secondary school math teachers. The course included a 4-week CoderDojo session in which the secondary school math teachers taught *Bricklayer* to a group of students ranging in age from 7 to 17.
- Gifted Education – In the spring of 2016, a 10-week pilot study was undertaken in which six facilitators taught a *Bricklayer*-based curriculum to approximately 120 4th grade students enrolled in the Gifted and Talented Program of the Omaha Public School system.

6 Conclusion

The *Bricklayer* ecosystem has a demonstrated appeal to students in the K-12 spectrum. It continues to evolve in order to more effectively engage all who wish to learn to code or sharpen their coding abilities. *Bricklayer* code has strong mathematical underpinnings including coordinate systems, geometric objects, arithmetic and geometric progressions that govern the structure of tessellations and fractals, and Boolean algebra. *Bricklayer* artifacts are artistic in nature. This engages visual thinking and spatial reasoning – cognitive processes which are closely correlated with STEM abilities. Last but not least, through LEGO[®] and 3D printing, *Bricklayer* spans the physical and virtual worlds.

References

- [1] (2016): *Persistence of Vision Raytracer*. <http://www.povray.org/>. Accessed: 2016-04-20.
- [2] O. Chang (2015): *Australian schools are scrapping history and geography and replacing them with coding classes*. Business Insider Australia.
- [3] S. Dredge (2014): *Coding at school: a parent's guide to England's new computing curriculum*. The Guardian.
- [4] S. Esper, S.R. Foster & W.G. Griswold (2013): *CodeSpells: Embodying the Metaphor of Wizardry for Programming*. In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE '13*, ACM, New York, NY, USA, pp. 249–254, doi:10.1145/2462476.2465593.
- [5] S. Esper, S.R. Foster & W.G. Griswold (2013): *On the Nature of Fires and How to Spark Them when You're Not There*. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13*, ACM, New York, NY, USA, pp. 305–310, doi:10.1145/2445196.2445290.
- [6] M. Felleisen, R.B. Findler, M. Flatt & S. Krishnamurthi (2001): *How to Design Programs: An Introduction to Computing and Programming*. MIT Press, Cambridge, MA, USA.
- [7] M. Felleisen, R.B. Findler, M. Flatt & S. Krishnamurthi (2009): *A Functional I/O System or, Fun for Freshman Kids*. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP '09*, ACM, New York, NY, USA, pp. 47–58, doi:10.1145/1596550.1596561.
- [8] A. Gross (2015): *25 states now require computer science for high school graduation*. Education Dive.
- [9] A. Lipson (2016): *Mathematical LEGO Sculptures*. <http://www.andrewlipson.com/mathlego.htm>. Accessed: 2016-04-20.
- [10] J. Maloney, M. Resnick, N. Rusk, B. Silverman & E. Eastmond (2010): *The Scratch Programming Language and Environment*. *Trans. Comput. Educ.* 10(4), pp. 16:1–16:15, doi:10.1145/1868358.1868363.
- [11] K. Pretz (2014): *Computer Science Classes for Kids Becoming Mandatory*.
- [12] G.M. Raimondo (2014): *Computer Science for Rhode Island (CS4RI) Initiative*. Office of the Governor.
- [13] E. Schanzer, K. Fisler, S. Krishnamurthi & M. Felleisen (2015): *Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap*. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE '15*, ACM, New York, NY, USA, pp. 616–621, doi:10.1145/2676723.2677238.
- [14] M. Smith (2016): *Computer Science For All*.
- [15] Staff (2006): *Minimal Surfaces, Maximal Bricks*. In: *Wired*.
- [16] R. Testuz, Y. Schwartzburg & M. Pauly (2013): *Automatic Generation of Constructable Brick Sculptures*. In M.-A. Otaduy & O. Sorkine, editors: *Eurographics 2013 - Short Papers*, The Eurographics Association, doi:10.2312/conf/EG2013/short/081-084.
- [17] O. Timcenko (1998): *LEGO: How to build with LEGO*. In: *32nd European Study Group with Industry Final Report*, pp. 81–94.

- [18] T. Toikkanen (2015): *Coding in school: Finland takes lead in Europe*. Learning Environments research group.
- [19] M. Wheatley (2016): *Chicago votes to make Computer Science a mandatory subject at all public schools*. Silicon Angle.
- [20] V. Winter (2014): *Bricklayer: An Authentic Introduction to the Functional Programming Language SML*. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, doi:10.4204/EPTCS.170.3.
- [21] V. Winter (2015): *The World Needs More Computer Science! What to do?* In D. Conway, S.A. Hillen, M. Landis, M.T. Schlegelmilch & P. Wolcott, editors: *Digital Media, Tools, and Approaches in Teaching and Their Added Value*, Waxmann Verlag GmbH, Germany, pp. 119–141.