Using Visualization to Reduce the Cognitive Load of Threshold Concepts in Computer Programming

1st Victor Winter
Computer Science Department
University of Nebraska-Omaha
Omaha, USA
vwinter@unomaha.edu

2th Michelle Friend
Teacher Education Department
University of Nebraska-Omaha
Omaha, USA
mefriend@unomaha.edu

3rd Michael Matthews
Mathematics Department
University of Nebraska-Omaha
Omaha, USA
michaelmatthews@unomaha.edu

4nd Betty Love
Mathematics Department
University of Nebraska-Omaha
Omaha, USA
blove@unomaha.edu

5th Sanghamithra Vasireddy
Computer Science Department
University of Nebraska-Omaha
Omaha, USA
svasireddy@unomaha.edu

***Abstract*—This *Full Paper* in the *Research to Practice Category* reports on an empirical empirical study in which novel educational tools and techniques were employed to teach fundamentals of *problem decomposition* – a cognitive task transcending disciplines. Within the discipline of computer science, problem decomposition is recognized as a foundational activity of software development. Factors that contribute to the complexity of this activity include: (1) recognizing patterns within an algorithm, (2) mapping the understanding of an algorithm to the syntax of a given programming language, and (3) complexity intrinsic to the problem domain itself.**

Cognitive load theory states that learning outcomes can be positively affected by reducing the extraneous cognitive load associated with learning objectives as well as by changing the nature of what is learned. In the study reported upon here, a novel instructional method was developed to decrease students' cognitive load. Novel instructional content supported by a custom visualization tool was used in a classroom setting in order to help novice programmers develop an understanding of function-based problem decomposition within the context of a visual domain. Performance on outcome measures (a quiz and assignment) were compared between the new method and the traditional teaching method demonstrated that students were significantly more successful at demonstrating mastery when using the new instructional method.

***Index Terms*—computational thinking, mathematical thinking, visual thinking, spatial reasoning, functional programming**

I. Introduction

Learning to program presents a variety of cognitive challenges. First, there is an abstraction between the commands typed into the computer and the output; novices must be able to understand and predict the outcome of program statements. Second, constructing a complex program is itself also cognitively demanding, as the programmer must keep track of a variety of statements, program state, and predicted outputs.

One way to lower the cognitive load associated with programming is through the use of visualization. Environments such as Scratch [5] use visualization in two ways. First, the block-based syntax is itself a visualization of program statements, with program elements puzzle-shaped to facilitate program-writing. Second, Scratch programs create animations, concretizing program output. Scratch programmers can easily figure out which parts of a program fit together through the use of block shapes, and can test whether their output behaved as predicted by observing the resulting animation when the program runs.

Beyond syntax and predicting program output, there are a number of skills that must be developed in order to construct programs whose execution embody non-trivial computations. *Computational problem decomposition* – the ability to computationally solve a problem by decomposing the problem into smaller composable computations – is an important activity in the construction of programs. Within this activity, the creation and use of functions play an important role.

For novice programmers who have little mathematical background, the *creation* and *use* (i.e., the understanding) of functions represents a *threshold concept* – akin to the usage of tools by early man. In other words, learning to create and use functions is like crossing a threshold, after which novices are transformed through their ability to manipulate powerful program abstractions.

Another threshold concept is the *second order* use of functions within a computation. By this we mean the *creation* and *use* of functions composed of other (previously created) functions. In his book *The Age of Spiritual Machines*, Ray Kurzweil proposes the *Law of Accelerating Returns* [3] which describes the importance of this second order use in terms of tool making: *"Technology goes beyond mere tool making; it is a process of creating ever more powerful technology using the tools from the previous round of innovation. In this way, human technology is distinguished from the tool making of other species. There is a record of each stage of technology, and each new stage of technology builds on the order of the*

This material is based upon work supported by the National Science Foundation under Grant No. 1712080.

978-1-7281-1746-1/19/$31.00 ©2019 IEEE

previous stage." [4]

This article focuses on the advanced feature set of the Bricklayer Grid (a custom web app) and the extent to which these features can facilitate the exploration and understanding of inductively-defined visual artifacts for mathematically-weak novice programmers. The visual artifacts considered belong to a class of fractal which we call *Laces*. The set of Laces is a strict subset of a more complex class of pattern referred to by Stephen Wolfram as *nested patterns* [20]. Laces are interesting because their construction algorithms are (conceptually) concise and the artifacts that result can (depending on scale) oftentimes be surprising, beautiful, unanticipated, and mathematically interesting.

II. Background

The research presented in this paper is informed by and grounded in cognitive load theory [13] and threshold theory [10].

1) Cognitive Load Theory: Cognitive load theory [13] [16] is a learning theory in which the application of an individual's *working memory* is the primary determinant underlying the amount of and rate at which learning can occur. Information to be processed during the act of learning is defined in terms of the *cognitive load* it places on working memory.

Cognitive load theory identifies three sources of (cognitive) load: (1) *intrinsic cognitive load* – the inherent/natural complexity underlying what is to be learned, (2) *extraneous cognitive load* – complexity introduced as a result of instructional design, and (3) *germane cognitive load* – an individual's "working memory resources available to deal with the element interactivity associated with intrinsic cognitive load" [14].

Element interactivity is a load crosscutting concept that provides the yardstick used to measure cognitive load [14]. Element interactivity captures the amount of information that must simultaneously be considered for a given knowledge level. From the perspective of instructional design, a key observation is that element interactivity can be reduced by raising the learner's knowledge level.

Though recognized as important, *learner motivation* lies beyond the scope of cognitive load theory. *"Cognitive Load Theory works on the assumption that the students are fully engaged and fully motivated...Cognitive Load Theory has nothing to say about students staring out the windows and not listening...When I say CLT has nothing to say about, I'm not saying its unimportant. It's just not part of theory"* [15, p. 1]. Research has shown a significant correlation between motivational and cognitive processing [2]. The intuition here is that motivation governs the level of mental investment a learner is willing to make.

2) Cognitive Load Reducing Instructional Techniques: The cognitive load reducing instructional techniques that most closely relate to our work are: (1) segmentation and (2) subgoal labeling. Both of these techniques can be seen as falling in the category of *grouping mechanisms*. That is, mechanisms whose purpose is to decompose a computation and increase the understanding of the resulting decomposition.

Segmentation concerns itself with breaking down the steps of a worked example into separate (conceptually self-contained) meaningful pieces. The segmentation can be provided by the instructor or the students can be tasked with performing the segmentation on their own. Segmentation has been shown to have positive effects in cases involving dynamic visualizations – videos or animations in which unsegmented information is presented in a continuous stream. Spanjers et al. [12] further showed that segmentation of worked examples had positive effects on learning outcomes when compared to corresponding non-segmented worked examples.

Subgoal labeling involves the assignment of labels to worked examples that have been segmented. The purpose of a label is to meaningfully summarize (or capture) the informational content of the segment with which it is associated. This helps learners distinguish structural information from incidental information. Subgoal labeling has been shown to be effective in the STEM disciplines in cases where problem solving has a procedural nature. In [7] [6], subgoal labeling was shown to be an effective instructional strategy for the construction of mobile apps created using a drag-and-drop programming environment.

3) Threshold Theory: Threshold theory [10] is premised on the assumption/observation that in many fields of study, the acquisition of knowledge is not achieved in a uniform manner. Threshold theory posits that, along an educational path, there are certain concepts that, once learned, permit a new and previously inaccessible way of thinking about something. In the theory, such concepts are called *threshold concepts*. A threshold concept "represents a transformed way of understanding, or interpreting, or viewing something, without which the learner cannot progress, and results in a reformulation of the learners' frame of meaning" [10, p. ix]

III. Programming Constructs and Tools

The empirical study on which this article reports revolves around a specific set of features developed for an in-house web app called the *Grid*. For this reason, we provide a detailed look at the Grid and its features as they pertain to the research question of this study.

A. Bricklayer

This work is contextualized within the Bricklayer educational ecosystem – a *low-threshold infinite-ceiling* system designed to teach math and computer science in ways that are engaging as well as technically meaningful. This is accomplished through the integration of a variety of web apps and tools that facilitate a thoughtful and systematic exploration of construction techniques underlying 2- and 3-dimensional block-based art (e.g., pixel art and geometric patterns) as well as the expression of such ideas in code.

B. The Grid

The *Grid* [17], shown in Figure 1, is a Bricklayer web app that allows users to choose a color by clicking on a palette, then clicking on cells in the displayed grid to fill each with

the chosen color. The app was designed to replace graph paper and colored markers in allowing users to create pixel art. The process of creating an image is iterative and often labor intensive.

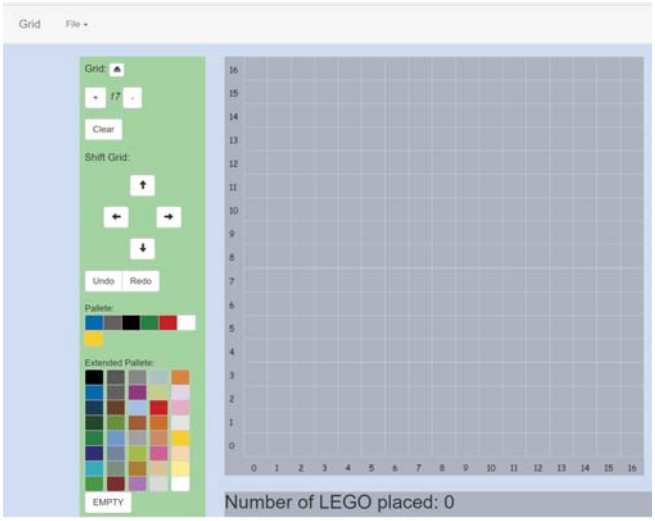


Fig. 1: The Grid.

The Grid's standard capabilities include: (1) saving images of Grid artifacts, (2) saving and loading json files containing models of Grid artifacts, (3) undo and redo operations, and (4) repositioning an artifact within a Grid (e.g., moving an artifact to the left, right, up or down).

C. Exploring Static and Dynamic Elements of Pattern

Grid artifacts are static in nature in the same sense that paintings, drawn on canvas, are static in nature. In more technical terms, an image of a Grid artifact captures the state of the Grid (with respect to cell occupancy) at a given point in time. In such images, dynamic elements of the construction process (e.g., the order in which cells are occupied to create the artifact) are tacit as is the algorithm used to construct the artifact, which is left to inference.

To address this limitation, the Grid was extended to include an advanced set of features and interactions to (1) make explicit various dynamic elements underlying artifact creation, and (2) facilitate the exploration and examination of the exposed dynamic elements.

When exploring dynamic elements of artifact creation, three Grid features are of central importance: *Create Frame* (copy), *Frame Mode* (similar to paste), and *Show Graph*.

It should be noted that the Grid's frame metaphor is similar to, though different from the typical copy-paste metaphor used in text editors. The notion of frame draws on abstractions from film-making where the central focus is on animation. A frame is a finite function (i.e., an array) from integers (i.e., indexes) to grid states. At present, the input domain of the frame function is $\{0, 1, \ldots, 5\}$. The Grid also supports rudimentary animation of artifact construction (which is not discussed in this article).

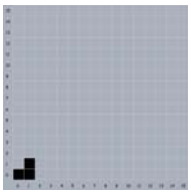
1) Frames: When selected, the *Create Frame* button will take a snapshot of (i.e., capture) the current state of the grid and associate this state with a particular index (i.e., input) of the *frame* function. The first snapshot will associate the current grid state with frame 0, the second snapshot will associate the current grid state with frame 1 and so on. We use the term *frame artifact* to denote a grid state that is stored in the frame function.

Complimenting the Create Frame button is a *Frame Mode* button which can be toggled on and off. When the Frame Mode is *off*, mouse clicks can be used to create an artifact using the standard "one mouse click for each cell to be colored" approach. However, when the Frame Mode is *on*, a mouse click will result in the placement of the entire contents of the most recently created frame artifact at the position selected by the mouse click. More specifically, the lower-left corner of the bounding box of the frame artifact will be positioned at the cell corresponding to the mouse click.

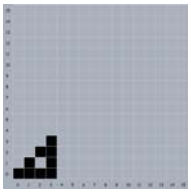
The frame function facilitates the systematic exploration and development of artifact construction techniques based on abstraction and inductive thinking. An example is shown in Fig 2. Beginning with an empty grid, one can use standard construction techniques to create an initial artifact. One can then use the *Create Frame* feature to take a snapshot of the artifact. Taking this initial snapshot will have the effect of updating the frame function so that the input 0 is associated with the artifact (i.e., frame(0) = artifact). With the Frame Mode on, one can then place copies of the captured artifact using single mouse clicks. Thus, the creation of the captured artifact has been abstracted to a single mouse click. When a desired grid state has been reached (e.g., the next artifact in a sequence of imagined artifacts), one can again capture the (entire) artifact using the Create Frame feature. In this way, abstract thinking is exercised through the association of single mouse clicks with artifacts of increasing size and complexity.

2) Artifact Graphs: As seen in Fig 3, using the Grid's *Show Graph* feature, the sequence of mouse clicks that created a frame artifact can also be displayed as a directed graph, overlayed on an otherwise empty grid. In such graphs, vertices capture positional information corresponding to mouse clicks and edges capture ordinal information (the order in which mouse clicks occurred). For example, a directed edge between vertex v_i and v_{i+1} indicates that the mouse-click corresponding to v_i occurred immediately before the mouse click corresponding to v_{i+1}. We use the term *artifact graph* (or *graph* when the context is clear) when referring to the graphical representation of a Grid artifact.

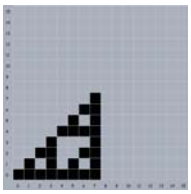
When the *Show Graph* mode is selected, the Grid displays information relating to artifact graphs in both visual and symbolic terms. Graphs are displayed visually in terms of dots (vertices) and lines (edges). Each value (i.e., input) in the domain of the frame function will have a corresponding artifact graph. Artifact graphs corresponding to any subset of the input domain of the frame function can be selected for simultaneous display. In such displays, the artifact graph for each frame input is assigned a unique color to help distinguish



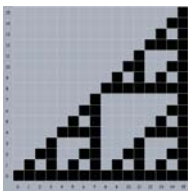
Step 1**:** Using the standard approach, construct the artifact shown using mouse clicks on cells $(0,0)$, $(1,0)$, and $(1,1)$. Use the Create Frame button to capture the artifact.



Step 2**:** With *Frame Mode* set to on, the artifact shown here can be constructed by extending the artifact created in the previous step through two mouse clicks, one mouse click on cell $(2,0)$ and the other on cell $(2,2)$. After the construction is complete, use the *Create Frame* button to capture the artifact.



Step 3**:** Use an approach similar to Step 2 to create the artifact shown here using through two mouse clicks, one mouse click on cell $(4,0)$ and the other on cell $(4,4)$.



Step 4**:** Use an approach similar to Step 2 to create the artifact shown here through two mouse clicks, one mouse click on cell $(8,0)$ and the other on cell $(8,8)$.

Fig. 2: Frame-based artifact construction of the reverseL Lace.

one frame graph from another. Figure 3 shows two artifact graphs.

In the Grid, graphs are also displayed symbolically in terms of sequences of vertex coordinates. Collectively, the graphs provide an important informational layer between the visual representation of a (manually created) Grid artifact and a Bricklayer program that describes a computation capable of creating this artifact.

Because of the features described in previous paragraphs, the Grid can be used as a tool to provide scaffolding for a variety of learning objectives relating to mathematical and computational thinking.

D. Example: The Inductive Construction of a Lace

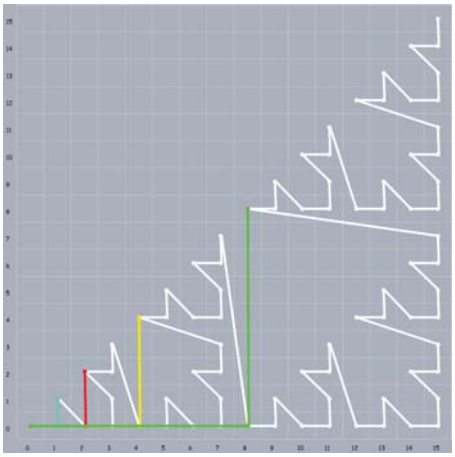
Tessellations and fractals represent two significant classes of patterns (both of which can be constructed using frames). In Bricklayer, we classify an artifact as a fractal if its extension admits *infinite self-similarity*. An interesting type of fractal includes block-based constructions which are porous. In three dimensions, such fractals are referred to as *sponges* with the Menger sponge being perhaps the best known example. In the literature, the two dimensional version of a sponge is sometimes referred to as a *carpet*. We use the term *Lace* to refer to such two dimensional artifacts because we feel the term more appropriately communicates their porous nature.

Figure 2 shows an artifact sequence whose elements are instances of a particular kind of lace which (due to its general shape) we call the *reverseL Lace*. The reverseL lace can be constructed using a variety of algorithms (including elementary cellular automata rule 102 [8]). Figure 3 shows the artifact graphs (with symbolic data omitted) corresponding to two different algorithms, each of which is capable of creating a reverseL lace. In this study, our discussion focuses on creating a Bricklayer program corresponding to the frame-based inductive construction algorithm shown in Figure 2.

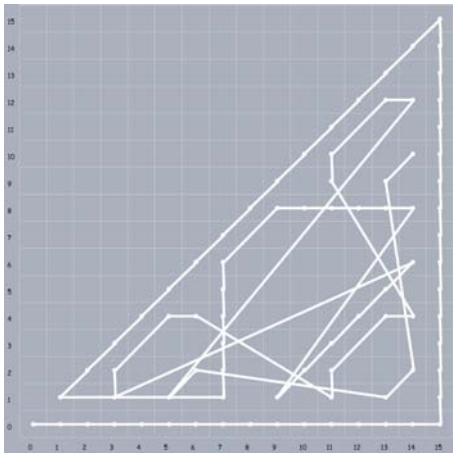
The basics of creating code from the information stored in the Grid's frame function is as follows. For each frame i, starting with $i = 0$, implement a non-recursive Bricklayer function capable of creating the artifact *frame*(i). Bricklayer functions must be parameterized on coordinates enabling the artifact they create to be placed at arbitrary locations. The result of this construction process is a sequence of user-defined function declarations where the implementation of *frame*(0) represents the base case. The body of the base case consists of a sequence of Bricklayer function calls that, when executed, produce the artifact captured in *frame*(0). For the construction algorithm we are considering, the *frame*$(i+1)$ artifact is defined inductively in terms of a composition of *frame*(i) artifacts positioned at various coordinates. From the perspective of coding, this means that the body of the Bricklayer function implementing *frame*$(i+1)$ consists of a sequence of calls to the function implementing *frame*(i). Note that, for each frame the symbolic data corresponding to its artifact graph provides the coordinate sequence to be used as the actual parameters for the function calls occurring in the body of *frame*$(i+1)$.

The generation of recursive code derived from frame information can be approached as a sequence of program transformations. Figure 4 shows the code resulting from the first translation according to the process described in the previous paragraph. Note that, from a technical standpoint, the resulting code is not recursive. Also note that the mindset behind the construction algorithm implemented is compositional (not decompositional). We use the terms *pseudo-recursive* or *inductive* to describe the computational nature of such programs.

A pseudo-recursive program of the kind shown in Figure 4 can be incrementally converted into a recursive program through two additional correctness-preserving transformations. The purpose of the first transformation is to isolate the computational differences between frame bodies. This is accomplished by expressing the bodies of frame functions in terms of *let-blocks* containing locally declared variables. The reason behind the introduction of local variables is to abstract and computationally isolate the numeric differences across frame bodies. Figure 5 shows the code that results from introducing let-blocks with local variables into the bodies of our frame functions.



(a) The composite artifact graph for an inductive (compositional) construction.



(b) The artifact graph for a recursive (decompositional) construction.

Fig. 3: Artifact graphs corresponding to a compositional and decompositional construction algorithm of the reverseL lace.

```
open Level_3;

fun frame0 (x,y) =
    (
        put2D (1,1) BLACK (x  ,y  );
        put2D (1,1) BLACK (x+1,y  );
        put2D (1,1) BLACK (x+1,y+1)
    );

fun frame1 (x,y) =
    (
        frame0 (x  ,y  );
        frame0 (x+2,y  );
        frame0 (x+2,y+2)
    );

fun frame2 (x,y) =
    (
        frame1 (x  ,y  );
        frame1 (x+4,y  );
        frame1 (x+4,y+4)
    );

build2D (32,32);

frame2 (0,0);

show "reverseL lace";
```

Fig. 4: A direct translation of frame information into code.

```
open Level_3;

fun frame0 (x,y) =
    (
      put2D (1,1) BLACK (x  ,y  );
      put2D (1,1) BLACK (x+1,y  );
      put2D (1,1) BLACK (x+1,y+1)
    );

fun frame1 (x,y) =
    let
      val shift = 2;
    in
      frame0 (x      ,y      );
      frame0 (x+shift,y      );
      frame0 (x+shift,y+shift)
    end;

fun frame2 (x,y) =
    let
      val shift = 4;
    in
      frame1 (x      ,y      );
      frame1 (x+shift,y      );
      frame1 (x+shift,y+shift)
    end;

build2D (32,32);

frame2 (0,0);

show "reverseL lace";
```

Fig. 5: Introducing let-blocks.

Performing the second transformation requires the discovery of one (or more) mathematical function(s) whose inputs are integers corresponding to the frame indices i (e.g., 1 or 2), and whose outputs, when added to the x and z coordinates in the frame body, will yield coordinates corresponding to the symbolic data (provided by the Grid) associated with *frame*(i). The purpose of these mathematical functions is to capture (i.e., computationally abstract) the mathematical similarity shared across all frames (with the possible exception of *frame*(0)).

In our running example, this similarity is captured by the following function.

$f(i) = 2^i$, in Bricklayer, 2^i is expressed as *power 2 i*

The function call $f(i)$ can be used to compute the data specific to frame i, for all frames in the range $i > 0$. In

```
open Level_3;

fun frame 0 (x,y) =
    (
       put2D (1,1) BLACK (x  ,y  );
       put2D (1,1) BLACK (x+1,y  );
       put2D (1,1) BLACK (x+1,y+1)
    )

  | frame i (x,y) =
    let
      val shift    = power 2 i;
      val previous = i - 1;
    in
      frame previous (x      ,y      );
      frame previous (x+shift,y      );
      frame previous (x+shift,y+shift)
    end;

build2D (32,32);

frame 2 (0,0);

show "reverseL lace";
```

Fig. 6: A recursive computation.

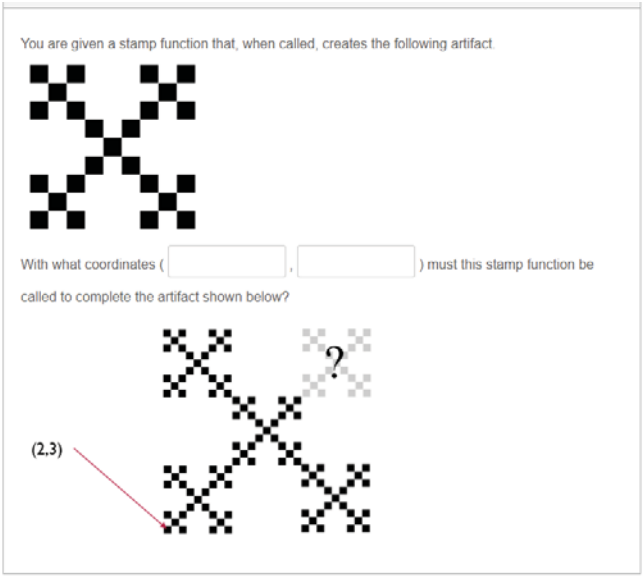


Fig. 7: An example of a quiz question.

turn, this permits the corresponding set of frame function definitions, which are parameterized on coordinates (x, z) to be abstracted (i.e., syntactically collapsed) into a single more general frame function definition that is parameterized on i and (x, z). Figure 6 shows the result of this transformation.

Due to their uniformity, the algorithms underlying the inductive construction of Laces represent pristine (i.e., clean) examples of the second order use of functions. In addition, the complexity of a Lace's fractal nature strongly encourages its construction through functional decomposition. For these reasons, the domain of Laces was chosen for our study.

IV. Method

The participants in the empirical study described in this article are novice programmers enrolled in a general education mathematics course that we have developed. The course curriculum develops the fundamentals of mathematical and computational thinking in a non-algebra based manner suitable for non-STEM majors. This quasi-experiment compared student performance on two quizzes and a create-a-lace assignment where (in Spring 2018) non-frame-based instruction asked students to create a lace in the Grid using a given "stamping pattern", or (in Fall 2018) frame-based instruction asked students to create a lace in the Grid using a given "stamping pattern" supported by frames AND to also create a bricklayer program whose execution creates the lace.

A. Introduction to Mathematical and Computational Thinking

Introduction to Mathematical and Computational Thinking (MCT) is a new course, developed to satisfy the general education mathematics requirement as an alternative to college algebra for undergraduates at the University of Nebraska at Omaha. The course is available to students who do not plan to major in STEM subjects, as STEM majors require college algebra as a prerequisite. Many of the students who enroll have previously failed the college algebra course. The course was offered for the first time in spring 2018 and has been offered continuously since. The content of the MCT course contains elements of quantitative reasoning, symbolic reasoning, computational thinking, critical thinking, and mathematical thinking. The educational objectives of the MCT course are for students to develop an understanding of pattern and algorithm, both informally and formally within the discrete domain of block-based visual art.

The educational objectives of MCT are pursued using an educational ecosystem called Bricklayer [18]. Bricklayer is a collection of interactive web apps, downloadable software, YouTube videos, and reading material that facilitates a thoughtful and systematic exploration of construction techniques underlying 2- and 3-dimensional block-based visual art (see Figure 8). Such exploration provides opportunities to exercise and develop spatial reasoning abilities as well as mathematical and computational thinking skills [1], [19].

In the course, students learn functional programming in the Bricklayer educational ecosystem, writing programs that result in two- and three-dimensional visual artifacts as output. The mathematics in the course is both because of the overlap between functions in mathematics and programming, and due to a focus on patterns, decomposition, and algorithms.

B. Experimental Context

The study presented here was quasi-experimental in nature, comparing student performance between semesters. The same instructor taught the MCT course in Spring and Fall 2018. Although the curriculum was not identical due to natural variations in teaching and slight modifications due to standard curriculum improvement processes, the content was nearly identical.

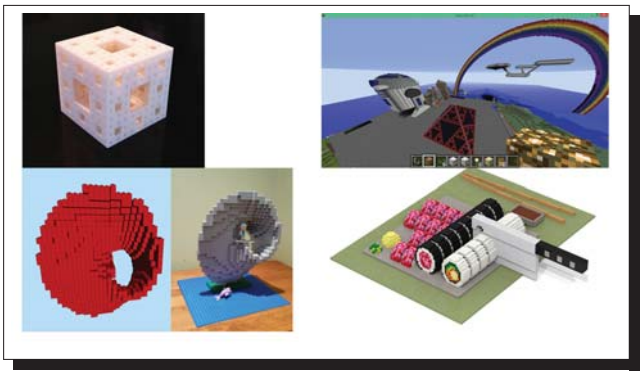


Fig. 8: Artifacts created using Bricklayer.

Within this context, student performance relating to the construction of Laces from the Spring 2018 and Fall 2018 sections of MCT were used in the study. The data analyzed consisted of two online quizzes, and one assignment.

The two online quizzes focused primarily measuring the understanding of the first threshold concept described in Section I – the use of a single abstraction in the creation of a larger artifact. In the quizzes, the larger artifacts being constructed were laces. Quizzes were automatically generated from questions randomly drawn from a question bank and students were permitted to take the quiz multiple times. Figure 7 shows an example of a question taken from the quiz question bank.

The first quiz was given during class at the end of the lecture on Laces. This first quiz consisted of five 1 point questions. The second quiz was given as a homework assignment and consisted of twenty 1 point questions. The questions for both quizzes were randomly drawn from a question bank. Scores are reported as a percentage correct.

The grid assignment involved the construction of a Lace. For the Spring 2018 section, a specification for a Lace was given and students were asked to use the Grid to create a Lace of a particular size through single-cell mouse clicks. The students in the Spring 2018 section were not asked to write a program whose execution would also create the Lace as this was deemed too difficult. This assignment was graded out of 10 points. For the Fall 2018 section, a specification for a Lace was given and students were asked to (1) use frames to construct a Lace of a particular size, and (2) use the information obtained from the Grid to create a program whose execution would also create the Lace. The program students were asked to create was of the type shown Figure 4. This assignment was graded out of 20 points. In both cases the scores are reported as a percentage.

C. Participants

Data was collected from students who completed the quizzes and assignment. Out of twelve students enrolled in Spring 2018, all completed both quizzes, but two did not complete the assignment and were excluded from analysis, resulting in data from ten students. Out of seventeen students enrolled in Fall 2018, one did not complete the second quiz or assignment and two others did not complete the assignment, resulting in data from fourteen students.

D. Research Question

Evidence from instructor-student interactions suggests that, for the target audience, the creation of user-defined abstractions (e.g., functions that create custom artifacts) represent a *threshold concept* [9] [11]. In other words, for these non-STEM students who have weak math skills and are novice programmers, the idea that they can write a function that creates a custom artifact is a significant challenge; when they understand and can use this idea it becomes an important tool they can use to decrease work in creating new and complex artifacts. A second threshold concept is the idea of nesting these abstractions - creating functions that contain previously-written functions. It is this second threshold concept that underlies inductive construction of the Lace assignment. These are powerful ideas, but also extremely challenging to the non-STEM students enrolled in this course.

It is in this context that we ask the following research question: Will the addition of up front (i.e., pre-coding) tool support for visual abstractions related to function-based *creation* and *use* positively impact student learning outcomes related to the understanding of the function-based creation of Laces?

V. Results

To assess the impact of frames the performance of the Spring 2018 and Fall 2018 classes was compared with respect to the two online quizzes and assignment described in Section IV-B. The results are displayed in Table I.

Students in the Fall 2018 section who interacted with the Grid with frames ($m = 94\%$) did better on the assignments than students who interacted with the Grid without frames in Spring 2018 ($m = 79\%$), ($t(22) = -4.11, p < 0.001$). The difference in performance was not seen on the first quiz, where students in the two sections did not perform significantly differently ($md = 1\%, t(22) = -0.30, p = 0.77$), but there were significant differences between the two sections on both the second quiz and the assignment.

Qualitatively, there were substantial differences between the assignments submitted in each of the semesters. In Spring 2018, when students were instructed without frames, submissions on the Laces assignment demonstrated significant confusion. Figure 9 gives a sample of student work taken from the spring (without frames) and fall (with frames) semesters. The top left image was typical of student submissions; the top right image displays the correct solution. Similarly, the bottom left image is a typical student submission, and matches the correct solution shown in the bottom right corner.

Even strong students in the Spring 2018 course were not able to correctly create a Lace that demonstrated the recursive pattern required by the assignment. The top left image in Figure 9 was submitted by a student who did well in the course

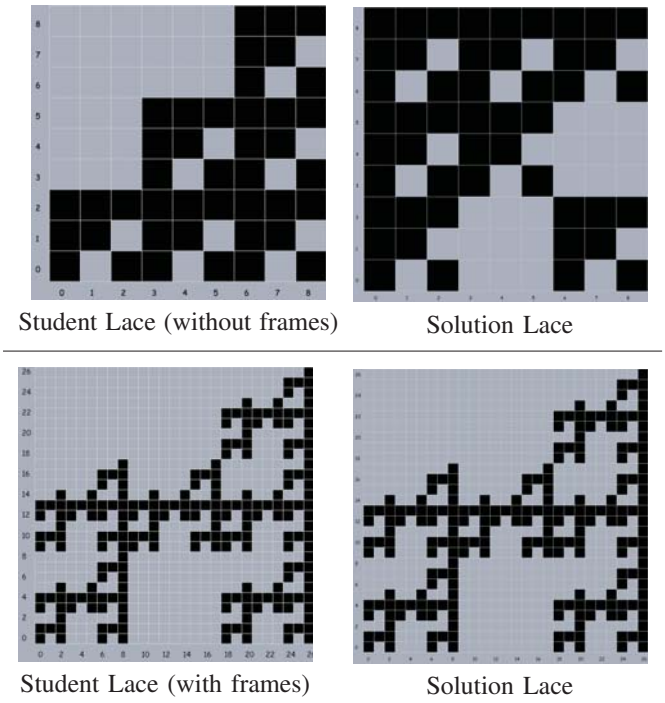


Fig. 9: Examples of Laces created by students and solution Laces.

	Spring 2018 without frames	Fall 2018 with frames	
Quiz 1	m = 96% sd = 7%	m=97% sd = 11%	t(22)=-0.30 p=0.77
Quiz 2	m = 70% sd = 23%	m=91% sd = 13%	t(22)=-2.79 p=0.01
Grid Assignment	m = 72% sd = 22%	m = 93% sd = 15%	t(22)=-2.74 p = 0.01
Total	m=79% sd = 10%	m = 94% sd = 7%	t(22)=-4.11 $p < 0.001$

TABLE I: Comparison of results between semesters.

and demonstrated mastery of most concepts, yet initially struggled with the recursion and pattern decomposition of Laces. The confusion displayed qualitatively in student work is partly masked in the quantitative results by the grading scheme, in which good faith efforts resulted in no less than 50% of the credit, meaning that the true difference in understanding between the two groups is larger than demonstrated by the significant difference in scores on the Grid assignment between semesters. Eleven out of the fourteen students in Fall 2018, who learned with frames, received a perfect score on the assignment, while only three out of the ten students in Spring 2018, who learned without frames, received a perfect score.

VI. Discussion

The Bricklayer educational ecosystem was created to facilitate engagement in mathematical and computational thinking. By relying on humans' natural ability to recognize and process visual patterns, we have been able to help students engage in powerful mathematical and computational structures. Students develop and combine foundational skills needed to understand patterns in this visual domain with the ability to express such patterns in code. In Bricklayer programs, the first and second order use of user-defined functions are threshold concepts whose understanding entails significant cognitive loads.

To alleviate the high cognitive load associated with these threshold concepts, the Bricklayer Grid web app has been extended with a set of frame features. A frame in this context operates like creating a stamp, in which a first-order pattern can be applied to the canvas in the same pattern as a second-order pattern and thus encourages algorithmic understanding involving higher-level abstractions (aka frames). The purpose of this feature set is to reduce the intrinsic cognitive load of the targeted threshold concepts by changing the knowledge level of the learner as well as providing tool support for this new knowledge level.

A preliminary study involving non-STEM majors at our university supports the position that the Grid's frame feature set positively contributes to student mastery relating to the first and second order use of user-defined functions. In this study, the Grid's frame feature was used to reduce intrinsic cognitive load associated with the understanding of Lace construction. The Grid's frame feature encourages algorithmic understanding involving higher-level abstractions (aka frames). The understanding of the frame feature and its relation to abstraction can be developed incrementally: a single frame (i.e., frame(0) = user-defined abstraction) can be used to create a wide variety of tessellations. Through such exercises, fluency in the thought processes underlying these constructions increases thereby increasing the student knowledge level as it relates to the first threshold concept – the use of user-defined functions in the construction of (complex) artifacts. After this has been accomplished, the use of the multiple frame feature can be used to create both tessellations and laces. In this manner, the student knowledge level as it relates to the second threshold concept can be increased.

References

[1] M. Friend, M. Matthews, V. Winter, B. Love, D. Moisset, and I. Goodwin. Bricklayer: Elementary Students Learn Math Through Programming and Art. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 628–633, New York, NY, USA, 2018. ACM.

[2] W.-H. Huang. Evaluating learners' motivational and cognitive processing in an online game-based learning environment. *Computers in Human Behavior*, 27(2):694 – 704, 2011. Web 2.0 in Travel and Tourism: Empowering and Changing the Role of Travelers.

[3] R. Kurzweil. *Age of Spiritual Machines: When Computers Exceed Human Intelligence*. Penguin USA, New York, NY, USA, 1st edition, 1999.

[4] R. Kurzweil. The Law of Accelerating Returns. Essay, March 2001. https://www.kurzweilai.net/the-law-of-accelerating-returns.

[5] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[6] L. E. Margulieux and R. Catrambone. Using subgoal learning and self-explanation to improve programming education. In A. Papafragou, D. Grodner, D. Mirman, and J. Trueswell, editors, *Proceedings of the 38th Annual Conference of the Cognitive Science Society*, pages 2009–2014. Cognitive Science Society, 2016.

[7] L. E. Margulieux, M. Guzdial, and R. Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 71–78, New York, NY, USA, 2012. ACM.
[8] Mathworld. Rule 102. http://mathworld.wolfram.com/Rule102.html.
[9] J. Meyer and R. Land. Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within the Disciplines. In 10^{th} *Improving Student Learning (ISL) Symposium*, pages 412–424. Oxford Centre for Staff Development, 2002.
[10] J. Meyer, R. Land, and C. Baillie. *Threshold Concepts and Transformational Learning*. Educational futures. Sense Publishers, 2010.
[11] J. H. F. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *"Higher Education*, 49(3):373–388, Apr 2005.
[12] I. A. E. Spanjers, T. van Gog, and J. J. G. van Merriënboer. Segmentation of Worked Examples: Effects on Cognitive Load and Learning. *Applied Cognitive Psychology*, 26(3):352–358, 2012.
[13] J. Sweller. Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12:257–285, 1988.
[14] J. Sweller. Element Interactivity and Intrinsic, Extraneous, and Germane Cognitive Load. *Educational Psychology Review*, 22(2):123–138, April 2010.
[15] J. Sweller. John Sweller on Motivation (Founder of Cognitive Load Theory). https://www.cyclesoflearning.com/home/john-sweller-on-motivation-founder-of-cognitive-load-theory, 2018. Accessed: 2018-05-08.
[16] T. van Gog and F. Paas. Cognitive Load Measurement. In N. M. Seel, editor, *Encyclopedia of the Sciences of Learning*, pages 599–601, Boston, MA, 2012. Springer US.
[17] V. Winter. The Grid, 2015. https://bricklayer.org/apps/grid_lite/grid.html.
[18] V. Winter, B. Love, and C. Corritore. The Bricklayer Ecosystem - Art, Math, and Code. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 2016.
[19] V. Winter, B. Love, and C. Corritore. The art of the Wunderlich cube and the development of spatial abilities. *International Journal of Child-Computer Interaction*, 2018.
[20] S. Wolfram. *A New Kind of Science*. Wolfram Media, Inc., 2002.