Learning Science Through Coding: An Investigation Into the Design of a Domain Specific Modeling Experience

Emma Anderson, MIT, eanderso@mit.edu
Daniel Wendel, MIT, djwendel@mit.edu

Abstract: Modeling is a core scientific practice. Today scientists create computer models requiring knowledge and skill in both science and computation. Additionally, computational modeling can provide powerful opportunities for learning. Hence, there is a need within K-12 science education to teach computational modeling. However, there are many barriers to incorporating this practice into K-12 classrooms, including the difficulty of learning how to code. In this paper we present the creation of a domain-specific modeling experience with a "custom block" visual programming language for an agent-based modeling environment for high school biology students. We found that with refined language, which maintained the metaphors and the abstractions vital to the science concepts being taught, students were able to complete a computational model without having been taught how to code. We also found preliminary evidence of students being able to learn science *through* building a computational model composed of customized code blocks.

Keywords: Computational Modeling, Ecosystems, Custom Code Blocks

Introduction

Modeling is a core scientific practice. Scientists create models to understand and explore phenomena, to predict the future, and for a multitude of other reasons (Epstein, 2008). Today scientists primarily create computer models, which requires that a scientist possesses disciplinary knowledge while also being able to think computationally (Foster, 2006). Not only is modeling a key scientific practice, computational modeling can provide powerful opportunities for learning. Building models provides students with a space to engage with scientific phenomena (Papert, 1980; Wilensky & Reisman, 2006; Wiklerson-Jerde, Wagh, & Wilensky, 2015), while revealing the underlying mechanism of how a model functions (Resnick, Berg, & Eisneburg, 2000; Wagh, Cook-Whitt, & Wilensky, 2017). For all these reasons there is a need within K-12 education to teach science students not only content knowledge but also how to build, modify, and use computational models.

The Next Generation Science Standards (NGSS) highlight the importance of computational modeling for science learning (NGSS Lead States, 2013), yet incorporating computational modeling into science classrooms is not a simple task. Many science teachers do not know how to code themselves, let alone teach coding to their students (Ertmer, & Ottenbreit-Leftwich, 2013; Sengupta, Kinnebrew, Basu, Biswas, & Clark, 2013). Learning to code in and of itself is challenging (Guzdial, 1994). Asking teachers to include additional practices or skills in the already long litany of content, practices, and concepts required of them is difficult due to limited time for instruction (Hew & Brush, 2007). Along with all of these barriers, teacher often find it difficult to access high quality curriculum that incorporates science content knowledge, cross cutting concepts, and computational modeling (Barr & Stephenson, 2011).

One solution to help bring computational modeling into science classrooms is through blocks-based coding and simulation platforms such as StarLogo Nova (Hsiao, Lee, & Klopfer, 2019) or blocks-based derivatives of NetLogo (e.g., Sengupta & Farris, 2012). However, while these learning environments provide scaffolds to facilitate coding in classrooms, for example through drag-and-drop visual code blocks that reduce syntax errors, teachers and students still need to be taught the *semantics*, the meaning, of the code (Sengupta et al., 2013). Learning how to code and how to teach coding within these learning environments can take a significant amount of instructional time (Lee, Psaila-Dombrowski, & Angel, 2017). For short, one-time experiences in computational modeling this time can be a high barrier for teachers to cross. An additional solution may be the creation of of Domain-Specific Modeling Experiences (DSME) through customized visual code blocks ("custom blocks") specific to the lesson content and phenomena being taught or explored. Through incorporating language in custom blocks that students are familiar with, custom blocks may provide an entry point into computational modeling without the extra time needed to learn coding semantics, and provide a space for students to engage deeply with the concepts being explored in the model. In this paper we present the iterative design processes for creating a DSME focused on a pond ecosystem. We focus on the design evolution of the language within the custom blocks, and explore if this additional scaffold can provide students with a taste of computational modeling while highlighting core science concepts and knowledge in a one-off lesson.

Literature review

Computational modeling

Building a computer model allows learners to engage with underlying mechanisms of scientific phenomena, allowing for deep understanding (Papert, 1980; Wilensky & Reisman, 2006; Wiklerson-Jerde, Wagh, & Wilensky, 2015). In coding a model, the learner is able to see how a model is created, understand that models are made and can be improved upon, and understand that modeling is a practice and not just a final product (Resnick et al., 2000; Wagh et al., 2017). However, computational modeling can be challenging for students both for syntactic and conceptual reasons. Syntax errors can be reduced or eliminated through the use of visual programming languages, where coding is done through blocks instead of written words, to remove the issues of missing commas or incorrectly spelled words. There are several examples of such languages, for example Scratch (Resnick et al., 2009), Alice (Conway & Pausch, 1997), and StarLogo (Klopfer, Scheintaub, Huang, & Wendel, 2009). Agent-based modeling can help lower the conceptual hurdles of creating computational models (Klopfer, 2003; Klopfer, Yoon, & Rivas, 2004; Resnick, 1994; Sengupta & Wilensky, 2009; Wilensky, 1999). In agent-based modeling the modeler builds the model from the perspective of individual agents (e.g., a gas molecule, a drop of water, a lion in a pride of lions). The modeler codes for how the agent should move and interact with other agents in the simulation. This type of programing allows for patterns to emerge or grow out of the individual interactions. Learning environments that combine visual programming languages with agent-based modeling provide powerful spaces for learners that help them to overcome both syntax and conceptual hurdles (Horn, Brady, Hjorth, Wagh, & Wilensky; Wilensky, Brady, & Horn, 2015). However, even in these environments, computational modeling remains challenging for students (Sengupta & Farris, 2012; Sengupta, et al., 2013; Wilkerson-Jerde et al., 2015).

Since the 1990s, teachers have found that learning science concepts through computational modeling requires taking time to acquire coding skill (Guzdial, 1994). When a science teacher decides to take on the task of teaching how to code in their classroom, it often means that she has to sacrifice time from other concepts and practices (Hew & Brush, 2007). One solution may come through the creation of customized coding blocks for domain-specific models.

Several learning environments have begun to explore these domain-specific code blocks, though all still require considerable investments in time. ViMAP (Sengupta & Farris, 2012) created domain-specific coding primitives for kinematic phenomena. Even with domain-specific coding blocks, Sengupta and Farris mention the need for teacher professional development to learn how to use ViMAP. DeltaTick (Wilkerson-Jerde et al., 2015) allows for the creation of 'core domain' customized coding blocks. A goal of DeltaTick is to allow for teachers to create their own customized coding blocks in response to student modeling needs. This can be an ambitious request for teachers who are novice programmers themselves and may not have the modeling experience necessary to identify core abstractions (Li, Turbak, Mustafaraj, 2017). CTSiM (Sengupta, et al., 2013) is a blocks-based programming language and simulation space that allows students to compare their models to an 'expert'-created model. Along with the modeling environment, lessons were created to guide students in engaging with the programming space. Sengupta et al. (2013) found that students did best in using and learning with CTSiM when there was a one to one student to teacher ratio to help facilitate scaffolding for the learners. This is not tenable for most science classrooms of twenty or more students with only one teacher.

Hasan and Biswas (2017) published a design guide for creating domain-specific modeling languages (DSML). They highlight four important design features of DSMLs:1) *simplicity* of language where one makes sure the language is intuitive; 2) a *conciseness* of the concepts present in the language; 3) *separation of concerns,* meaning that if concepts can be described independently they should be made into separate block sets; and 4) *consistency* of constructs, all of which "should contribute to the purpose of the language"(p. 29). Building on the lessons learned from each of the domain specific modeling language environments and the design guidelines presented by Hasan and Biswas, this paper explores the iterative design processes we undertook to create what we are terming a "domain specific modeling experience" (DSME) by providing customized code blocks to facilitate science learning through model building.

Research context and methods

StarLogo Nova: A visual blocks-based computational modeling platform

StarLogo Nova is a visual blocks-based modeling platform for agent-based modeling (Hsiao et al., 2019). StarLogo Nova evolved from Logo (Papert, 1980), a single agent programing environment. Unlike Logo, StarLogo Nova allows users to program thousands of agents at once. A web-based environment, StarLogo Nova is composed of three areas: the *information window* where a user can see the history of the project and also add notes for later; *Spaceland* where the programmed simulations run, buttons and sliders exist, and graphs or data tables can be displayed; and the *workspace* which is composed of a drawer of code blocks and tabs where different types of agents can be programmed. The interface is designed to allow for modelers to easily move between *Spaceland* and the *workspace* to facilitate quick linkages between built code and the run of the simulation.

Computer-supported complex system curricula

This paper is about a small slice from a larger research project. The larger study is exploring teaching high school biology from a complex systems perspective. The larger study is composed of an online teacher professional development course to learn how to teach five three-day computer-supported complex systems lessons. Each lesson is composed of teacher and student guides and a StarLogo Nova simulation. The lessons have students engage with and run experiments on the StarLogo Nova simulations in order to answer argumentation questions. In the fall of 2017, four expert teachers conducted a critical review of each of the five lessons from the larger study. These reviews critiqued aspects of the lessons that did not work, highlighted parts of the lessons which are successful, explained why certain lessons continue to be used in their classrooms, and pointed out why some lessons are not being used. Only one lesson was being taught by none of the four teachers: *Modeling a Pond*. It was clear this lesson needed to be revamped. One of the goals of this redesign (and the focus of this paper) was to include the practice of computational modeling as a key aspect of the lesson. With this in mind, *Waterville: Modeling a Pond Ecosystem* was created. The lesson focuses on a town, Waterville, where a local farmer has asked to lease the land around the town pond to grow corn and soybeans. The town holds an annual bass fishing tournament in the pond, and would like to know if they can lease land to the farmer and still run their fishing tournament. In order to answer this question, students have to complete a computer model of a pond to be able to explore the relationship between land cultivation and the health of the pond.

Students are given a link to a partially completed computer model of a pond ecosystem. The first task students have is to complete the code for the algae and bass. There are several scaffolds put into place to help students 'code' without having any prior experience. First, the world tab and the minnow tab of the model are fully coded, and the worksheets walk the students through a guided decoding of the minnow instructions. Second, the algae and bass tabs have all the code blocks needed, but presented as a Parsons problem, disconnected and scattered across the page. The third scaffold, which is the focus of this paper, is the use of custom blocks that are intuitive enough to not require learning a new vocabulary of 'coding' semantics.

Methods: Design cycles

A design-based research methodology (The Design-Based Research Collective, 2003) was utilized to hone this lesson through iterative rounds of design, implementation, analysis, and redesign. The design process involved several rounds of design with varying degrees of change in order to create the current version of the custom blocks. In this paper we focus on two major design cycles which we are referring to as cycle A and cycle B. In cycle A, a high school teacher implemented the lesson in his freshman biology course. We captured three pieces of data from this class implementation: screen shots of the code, field observations, and notes on a debrief discussion with teacher about the custom blocks. Cycle B was implemented in two high school biology classes by two different teachers in the same suburban high school. Similar to cycle A, we collected screen shots of the code and field observations. However, due to time constraints, discussion with the teachers occurred throughout the lessons rather than separately afterward.

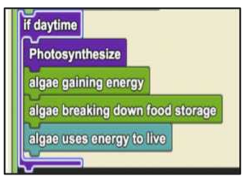
Data from both cycles went through rounds of open coding (Strauss & Corbin, 1997). The classroom observations and teacher debriefs were coded for moments of struggle with the customized blocks and for 'ah ha' moments with the model-building experience. Once the moments of struggle were identified, they were categorized (Maxwell, 2013) by whether or not the struggle related to the language of the custom blocks or a general block-based coding error. See Figure 1 for an example of an observation instance, the coding error, and how these two pieces of data were analyzed. For the struggles related to the language of the custom blocks, the blocks were reevaluated relative to the four guiding principles for domain specific modeling languages from Hasan & Biswas (2017) with a particular emphasis on how well the language aligned with the learning goals of the overall lesson. The language was adjusted to better meet the criteria and to avoid a similar struggle in the future.

Observation Report Sept 2018

Observation: A pair working together put several blocks nested below "if daytime." I [the observer] asked the students if these actions only happen during the day for the algae or all the time? The students read over the blocks and removed all expect for photosynthesis

Inference: Upon being directed to reread their code and think about what 'if daytime' means the students realized that only photosynthesis is time limited and removed the other actions that take place all the time

Code Error



```
if daytime
  Photosynthesize
  algae gaining energy
  algae breaking down food storage
  algae uses energy to live
```

Data Analysis

Code: Struggle
Category: code block error not understanding 'c blocks'

Figure 1. Data analysis for struggle.

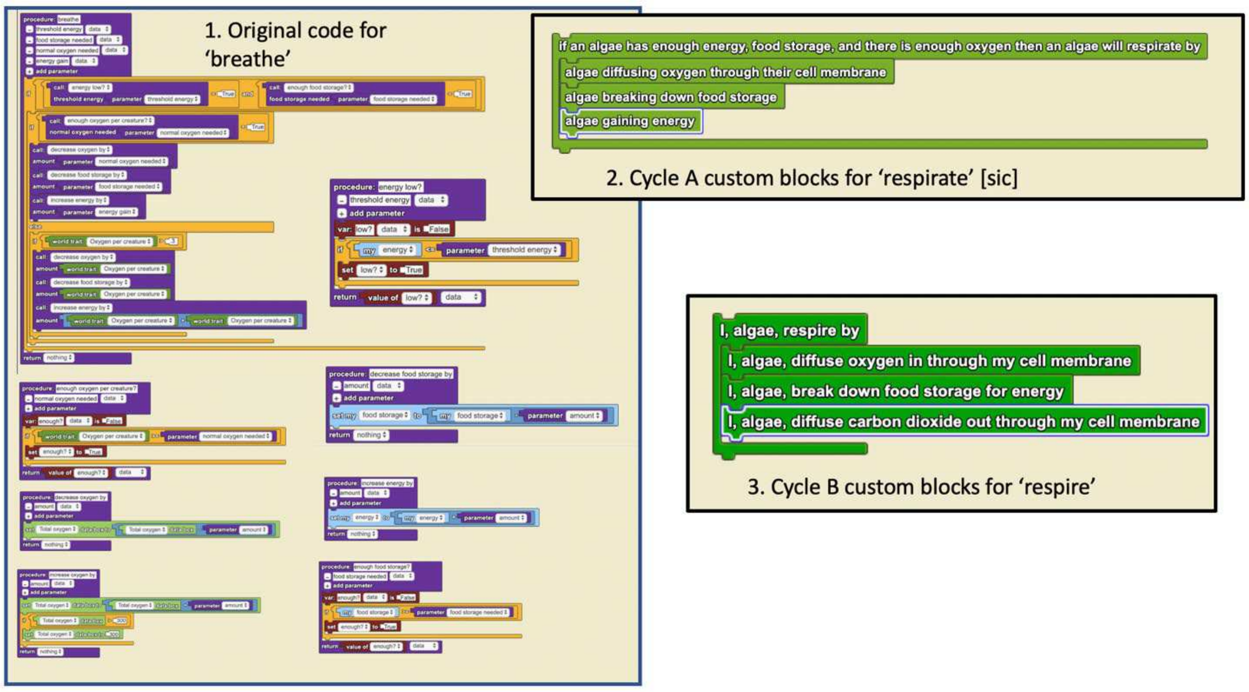
In the section below, we present the evolution of the coding language for the concepts of respiration and photosynthesis in cycle A and B. We also share lessons learned from observing students engaging with these two designs of the custom blocks.

Findings: Evolution of the coding language for *Waterville*

From the original code to custom blocks

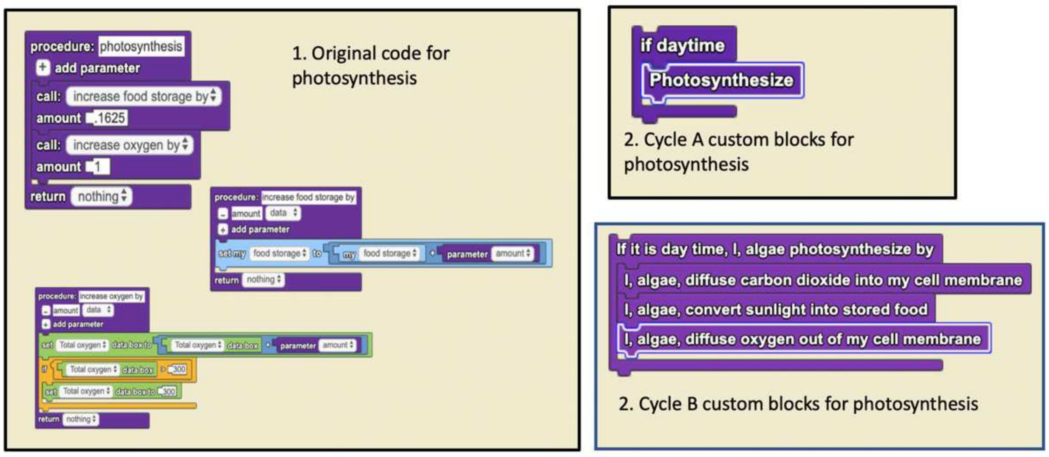
The original code for respiration, which was labeled 'breathe' as seen in Figure 2, involved a complex set of 'if else' statements pulling in several other procedures of 'if then' statements. The commands for 'breathe' were long and complex, a daunting set of code for a novice to read, let alone attempt to program. The original code for photosynthesis, while much shorter and only calling on two other procedures, included 'magic numbers' whose values were empirically chosen to stabilize the base model but were not conceptually relevant to the biological process being represented. For these reasons, custom blocks were created to help scaffold the model building experience for students.

In cycle A's round of creating custom blocks for respiration and photosynthesis, the multiple procedures calling other procedures were compressed into single 'if then' statements. Along with collapsing the procedures into each other, the language was written more like an English sentence and less like code language. Blocks that represent similar actions were color-coded so all the respiration blocks were colored green while the photosynthesis blocks were colored purple. See Figure 2 for images of the original model, cycle A's custom blocks and cycle B's custom blocks.


1. Original code for 'breathe'
2. Cycle A custom blocks for 'respirate' [sic]
3. Cycle B custom blocks for 'respire'

```
if an algae has enough energy, food storage, and there is enough oxygen then an algae will respirate by
  algae diffusing oxygen through their cell membrane
  algae breaking down food storage
  algae gaining energy
```

```
I, algae, respire by
  I, algae, diffuse oxygen in through my cell membrane
  I, algae, break down food storage for energy
  I, algae, diffuse carbon dioxide out through my cell membrane
```


1. Original code for photosynthesis
2. Cycle A custom blocks for photosynthesis
2. Cycle B custom blocks for photosynthesis

```
procedure: photosynthesis
  add parameter
  call: increase food storage by
  amount .1625
  call: increase oxygen by
  amount 1
return nothing

procedure: increase food storage by
  amount data
  add parameter
  set my food storage to my food storage + parameter amount
return nothing

procedure: increase oxygen by
  amount data
  add parameter
  set Total oxygen data box to Total oxygen data box + parameter amount
  if Total oxygen data box > 300
    set Total oxygen data box to 300
return nothing
```

```
if daytime
  Photosynthesize
```

```
If it is day time, I, algae photosynthesize by
  I, algae, diffuse carbon dioxide into my cell membrane
  I, algae, convert sunlight into stored food
  I, algae, diffuse oxygen out of my cell membrane
```

Figure 2. Custom block language evolution for respiration and photosynthesis.

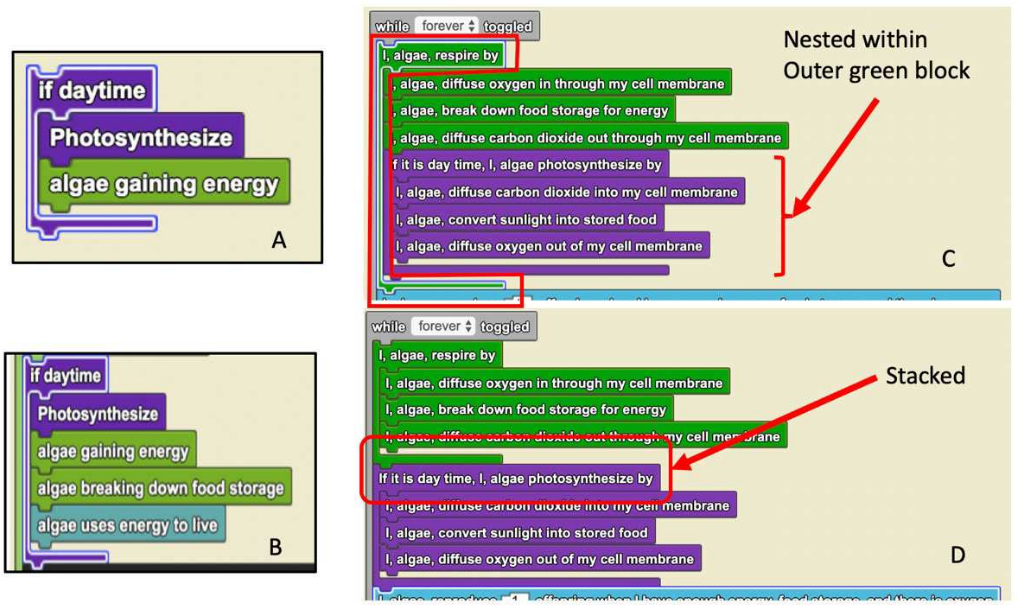
Struggle with Language

There were several issues with the cycle A set of custom blocks. These issues were highlighted by the struggles students had in completing the model. We observed mistakes in coding that emphasized a lack of *simplicity* and *conciseness*. For example, in respiration, students had a hard time figuring out which blocks should be connected into the "If an algae has enough energy, food storage, and there is enough oxygen then an algae will respirate by" block. There are several issues just with this single custom block. First, the number of words on the block is very high, and they are joined into an awkwardly written sentence. Second, the information on the block makes assumptions of student understanding, for example, knowing what is meant by 'food storage'. There were also issues of *consistency.* The metaphor of agent-based programming—where the language is from the agents' perspective—is also not present in this block nor in the other respiration blocks. The blocks were written in third person ('an algae') instead of first person. Some of the language was too general, such as "algae diffuse oxygen through their cell membrane," leading to a lack of *separation of concerns*. Students were unsure if the block meant oxygen going *into* or oxygen *leaving* the cell. This confusion resulted in students including this block either in the respiration set of code or in the photosynthesis section. The code block "algae gaining energy" was also too general, as both the students and the teacher wanted to place this code into the photosynthesis section, yet we the designers had intended it to be part of the respiration commands.

In a debrief conversation with the teacher after class about the language in the customized code blocks, he pointed out how the language failed to consistently align with the concepts being covered, causing confusion. In particular, he opined that the metaphor presented for the Krebs Cycle as 'respiration', and the commands given for photosynthesis, were not specific enough. He suggested that 'gaining energy' should be 'using energy to break chemical bonds to gain ATPs.' Not only was the language an issue, but the levels of abstraction of concepts were not consistent across the blocks. For example, respiration was composed of four commands, while photosynthesis was just two, with one command being simply "photosynthesize." The different levels of abstraction made it less clear which code blocks go with each other. See Figure 3 for examples of language coding errors due to these issues.

Redesign

Given all the struggles that both the students and the teacher experienced in attempting to complete the model, and acknowledging that a main learning goal of the lesson was for students to understand how oxygen moves through the ecosystem, it was clear the language needed to be rethought. In redesigning the language of the custom blocks for respiration and photosynthesis, it was important to highlight the role of oxygen in both cellular processes and to keep the level of abstraction similar for both concepts. With all of this in mind, the customized code language was updated resulting in respiration and photosynthesis each consisting of four custom blocks. The metaphor of agent-based programming was also better presented as seen in a shift from third person to first person language. The language became more specific, and the word oxygen was used for both process commands. The order in which the commands are stacked are inverse to each other; for respiration the blocks stack "I, algae, diffuse oxygen in through my cell membrane"; "I, algae, break down food storage for energy"; and finally "I, algae, diffuse carbon dioxide out through my cell membrane." Photosynthesis goes in the reverse order: "I, algae, diffuse carbon dioxide into my cell membrane"; "I, algae, convert sunlight into stored food"; and finally "I, algae, diffuse oxygen out of my cell membrane."



```
A
if daytime
    Photosynthesize
    algae gaining energy
```

```
B
if daytime
    Photosynthesize
    algae gaining energy
    algae breaking down food storage
    algae uses energy to live
```

```
C
while forever toggled
    I, algae, respire by
        , algae, diffuse oxygen in through my cell membrane
        , algae, break down food storage for energy
        , algae, diffuse carbon dioxide out through my cell membrane
        f it is day time, I, algae photosynthesize by
            I, algae, diffuse carbon dioxide into my cell membrane
            I, algae, convert sunlight into stored food
            I, algae, diffuse oxygen out of my cell membrane
```

```
D
while forever toggled
    I, algae, respire by
        I, algae, diffuse oxygen in through my cell membrane
        I, algae, break down food storage for energy
        I, algae, diffuse carbon dioxide out through my cell membrane
    If it is day time, I, algae photosynthesize by
        I, algae, diffuse carbon dioxide into my cell membrane
        I, algae, convert sunlight into stored food
        I, algae, diffuse oxygen out of my cell membrane
```

Figure 3. ‘Mistakes’ made with the custom blocks. Image A and B show examples of errors in coding due to struggles with the language in the custom block code from cycle A. Image C is a syntax/semantic error with the blocks incorrectly nested. Image D shows the same set of blocks without the nesting error.

Challenges remain

When this updated version of the lesson was taught, one of the two teachers had her students copy her completed code which was projected on a screen in front of the classroom. While it is not totally clear why this teacher chose to not have her students engage in figuring out the code themselves, this teacher did struggle herself with building the code. Before class began, she approached the researcher observing her classroom and asked for help debugging her code. She could not get her model to run for more than 500 ticks (500 iterations of the code). In a quick look at her code, it was clear that she hadn’t made a mistake based on the language or biological concept, but based on a limited understanding of how the programming environment works. She did not realize that we use c-shaped blocks to indicate where one coding idea ends and the next idea begins, and as a result had nested her photosynthesis code within her respiration code. The effect on the model was that rather than having two independent processes, photosynthesis was only able to happen at half the rate of respiration (because it was only daytime for half of the time respiration was occurring). The researcher helped her to uncouple the nested code and to rather stack the code sections on top of each other. Figure 3 image C shows the error and image D reveals the correct stacking of this code. It was this debugged code that the teacher shared with her class to copy.

Preliminary evidence of science concepts reinforced through coding

In the second classroom that implemented the revised lesson the teacher let her students complete the build of the model on their own or with a partner. While some students struggled in programming the algae, the reasons for their challenges in coding were not due to unclear language, but rather issues with a conceptual understanding of respiration and photosynthesis. As a researcher watched a student code the algae, the student suddenly exclaimed, “I didn’t even know that algae went through respiration!” The act of coding the model, for this student, was able to reinforce the scientific concept that algae (plants) complete the processes of respiration *and* of photosynthesis.

Discussion

We found that by including the additional scaffold of customized code blocks tailored to a specific lesson and content area, we enabled students to complete the build of a computational model without any additional coding lessons. This was true for both cycle A and cycle B of the customized code blocks, though the latter version was easier. It is important to point out that the one teacher did have trouble putting her code together, resulting in a model that would not run for long. The mistake that this teacher made was not due to the language of the code blocks, but to a misunderstanding of how to build code in StarLogo Nova. Unlike Sengupta and Farris (2012) who felt that ViMAP with its customized code still necessitated teacher professional development on learning to code, however, we do not see the misunderstanding as necessitating additional prior teacher professional development. A short note in the teacher guide of how nesting in StarLogo Nova works could anticipate this mistake and still require little to no coding expertise on the teacher's part.

A major advantage of having learners build models is to engage deeply with the phenomena being explored (Papert, 1980; Wilkinsky & Resiman, 2006; Wiklerson-Jerde, Wagh, & Wilensky, 2015) and to make transparent how a model works (Resnick et al., 2000). We saw some evidence of students thinking through and changing their understanding of the underlying phenomena being explored though coding the pond ecosystem with the cycle B version of the customized blocks, as a student struggling to build her model had an 'ah ha' moment of understanding that even photosynthesizing organisms respire. In CTSiM Sengupta et al. (2013) found that students did the best when there was a very low student to teacher ratio for building within their platform. In our case we see that students are able to engage in coding without the low ratio of students to teacher. This highly scaffolded computational modeling experience allowed for students to engage meaningfully with the concepts in the scientific model without requiring previous training or high levels of teacher intervention.

In the reworking of the language we followed the guiding principles for writing DSMLs as presented by Hasan and Biswas (2017) which include: *simplicity* of language; *conciseness* of concepts; *separation* of concerns; and *consistency* of purpose in the code. In the processes of using this framework to create our Domain Specific Modeling Environment (DSME), we found the idea of consistency to be multi-dimensional and to include factors such as: alignment with learning goals, consistent metaphors (e.g., agent-based modeling), consistent levels of abstraction, coherent language (e.g., tense, terminology, and grammatical structure), and even consistency of semantic hints such as block colors. Additionally, while covered by the ideas of simplicity and consistency, we found it helpful to think in terms of *parsimony*, a term that precisely encapsulates what we aspired to achieve.

Of course, many challenges remain. While we found that students were able to construct the model without high levels of teacher intervention, they were not given the opportunity to struggle with the code when the teacher felt less than fully confident. Additionally, while there is evidence of some productive struggle on the part of the students, not all struggle was productive. And indeed, even in the writing of this paper we found a multitude of additional areas of inconsistency or lack of parsimony that we believe could be improved with further revision.

We believe that for one-off computational modeling experiences, the additional scaffold of customized code blocks shows promise. By emphasizing alignment with the scientific concepts in the design, we showed it is possible to create a Domain Specific Modeling Environment in which the customized code blocks were able to provide a taste of computational modeling. The modeling aspect of this lesson did not require teachers to be expert coders or modelers nor did the lesson need additional time to teach students how to code.

References

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *Inroads 2(1),* 48-54.

Conway, M. J., & Pausch, R. (1997). Alice: Easy to learn interactive 3D graphics. ACM SIGGRAPH Computer Graphics, 31(3), 58 – 59.

Design-Based Research Collective. (2003). Design-based research: An emerging paradigm for educational inquiry. *Educational Researcher*, *32*(1), 5-8.

Epstein, J. M., (2008). Why model? *Journal of Artificial Societies and Social Simulation 11(4),* 12.

Ertmer, P. A., & Ottenbreit-Leftwich, A. (2013). Removing obsticles to the pedagogical changes required by Jonassen's vision of authentic technology-enabled learning. *Computers & Education, 64,* 175-182.

Foster, I. (2006). A two-way street to science's future. *Nature 440,* 419

Guzdail, M. (1994). Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments, 4(1),* 1-44.

HASAN, A., & BISWAS, G. (2017). Domain specific modeling language design to support synergistic learning of STEM and computational thinking. *Siu-cheung KONG The Education University of Hong Kong, Hong Kong*, *28*.

Hew, K. F., & Brush, T. (2007). Integrating technology into K-12 teaching and learning: Current knowledge gaps and recommendations for future research. *Educational Technology Research and Development, 55(3),* 223-252.

Horn, M.S., Brady, C., Hjorth, A., Wagh, A., Wilensky, U. (2014, June). Frog pond: a codefirst learning environment on evolution and natural selection. In *Proceedings of the 214 conference on Interaction design and children* 357-360. ACM.

Hsiao, L., Lee, I., & Klopfer, E. (2019). Making sense of models: How teachers use agent-based modeling to advance mechanistic reason. *British Journal of Educational Technology, 50(5),* 2203-2216.

Klopfer, E. (2003). Technologies to support the creation of complex systems models—using StarLogo software with students. *Biosystems 71(1-2),* 111-122.

Klopfer, E., Scheintaub, H., Huang, W., & Wendel, D. (2009). StarLogo TNG. In *Artificial Life Models in Software* (pp. 151-182). Springer, London.

Klopfer, E., Yoon, S., & Rivas, L. (2004). Comparative analysis of plam and wearable computers for participatory simulations. *Journal of Computer Assisted Learning, 20(5),* 347-359.

Lee, I. A., Psaila Dombrowski, M., & Angel, E. (2017, March). Preparing stem teachers to offer new mexico computer science for all. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 363-368). ACM.

Li, I., Turbak, F., & Mustafaraj, E. (2017, October). Calls of the wild: Exploring procedural abstraction in app inventor. In *2017 IEEE Blocks and Beyond Workshop (B&B),* 79-86.

Maxwell, J. (2013). *Qualitative research design: An interactive approach.* Los Angeles, CA: Springer.

NGSS Lead States (2013) Next generation science standards: for states, by states. The National Academies Press, Washington, DC

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc..

Resnick, M. (1994). Changing the centralized mind. *Technology Review, 97(5),* 32.

Resnick, M., Berg, R., & Eisenberg, M. (2000). Beyond black boxes: Bringing transparency and aesthetics back to scientific investigation. The Journal of the Learning Sciences, 9(1), 7–30.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... & Kafai, Y. B. (2009). Scratch: Programming for all. *Commun. Acm*, *52*(11), 60-67.

Sengupta, P., & Farris, A. V. (2012, June). Learning kinematics in elementary grades using agent-based computational modeling: a visual programming-based approach. In *Proceedings of the 11th international conference on interaction design and children* (pp. 78-87). ACM.

Sengupta, P., Kinnebrew, J. S., Basu, S., Biswas, G., & Clark, D. (2013). Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework. *Education and Information Technologies*, *18*(2), 351-380.

Sengupta, P., & Wilensky, U. (2009). Learning electricity with NIELS: Thinking with electrons and thinking in levels. *International Journal of Computers for Mathematical Learning*, *14*(1), 21-50.

Strauss, A., & Corbin, J. M. (1997). *Grounded theory in practice*. Sage.

Wagh, A., Cook-Whitt, K., & Wilensky U. (2017). Bridging inquiry-based science and constructionism: Exploring the alignment between students tinkering with code of computational models and goals of inquiry. *Journal of Research in Sceince Teaching 54(5),* 615-641.

Wilensky, U. (1999). NetLogo. Evanston, IL: Center for connected learning and computer-based modeling, Northwestern University.

Wilensky, U., Brady, C. E., & Horn, M. S. (2014). Fostering computational literacy in science classrooms. *Commun. ACM*, *57*(8), 24-28.

Wilensky, U., & Reisman, K. (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories—an embodied modeling approach. *Cognition and Instruction, 24(2),* 171-209.

Wilkerson-Jerde, M., Wagh, A., & Wilensky, U. (2015). Balancing curricular and pedagogical needs in computational construction kits: Lesson from the DeltaTick project. *Science Education, 99,* 465-499.

Acknowledgements

We want to thank Luis Xavier Ramos Tormo for building the underlying model of the pond ecosystem and Hyeonsu Kang for the creation of the custom blocks feature in StarLogo Nova. This study was funded by NSF DRL_1721003 and NSF DRL_1742138.