Code-first learning environments for science education: a design experiment on kinetic molecular theory

Umit Aslan, *umitaslan@u.northwestern.edu*
Learning Sciences, Northwestern University, Evanston, IL, USA

Nicholas LaGrassa, *nick.lagrassa@u.northwestern.edu*
Learning Sciences and Computer Science, Northwestern University, Evanston, IL, USA

Michael Horn, *michael-horn@northwestern.edu*
Learning Sciences and Computer Science, Northwestern University, Evanston, IL, USA

Uri Wilensky, *uri@northwestern.edu*
Learning Sciences and Computer Science, Northwestern University, Evanston, IL, USA

Abstract

Code-first learning entails the use of computer code to learn a concept, and creating computational models is one such effective method for learning about scientific phenomena. Many code-first learning approaches employ the visual block-based programming paradigm in order to be accessible to school children with no prior programming experience, providing them with high-level domain-specific code-blocks that encapsulate the underlying complex programming logic. However, even with the aid of visual clues and the benefit of simpler primitives like “forward” and “repeat,” many phenomena studied in classrooms such as the behavior of gas particles in Kinetic Molecular Theory (KMT) are challenging to describe in code. We hypothesized that code blocks designed from a phenomenological perspective to model the behavior of familiar objects and events would both promote students’ authoring of computational models and their ability to encode and test their beliefs within their models. We created these phenomenological blocks within a code-first gas particle sandbox and integrated it into a KMT lesson plan. Two high school teachers taught this curriculum to 121 students, from which we gathered and analyzed video footage from lesson activities and student focus groups. We found that the phenomenological blocks gave students the ability to start programming right away and to express their intuitive understanding of KMT through computational models. This exploratory study demonstrates the potential for phenomenological programming to broaden the application and accessibility of code-first computational modeling for learning scientific phenomena.

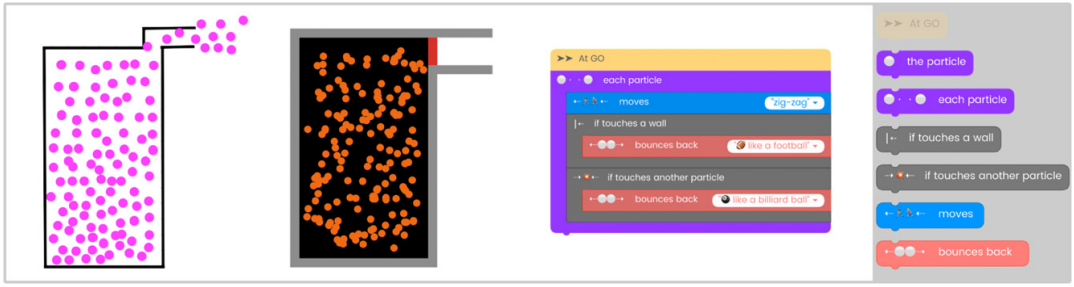


Figure 1. The code-first gas particle sandbox with phenomenological blocks

Keywords

Constructionism, code-first learning environments, agent-based modeling, blocks-based programming, chemistry education, computational thinking

Introduction

Initiatives for integrating computational thinking (CT) into science and mathematics curricula has gained significant momentum in recent years (e.g., Grover & Pea, 2013; Sengupta et al., 2013; Weintrop et al., 2015; Wing, 2006). It is argued that embedding computation across STEM curricula would align science education with contemporary scientific practices, deepen student learning, and promote computational literacy (Wilensky et al., 2014). Theoretical work such as Weintrop et al.'s (2015) CT-STEM taxonomy provide actionable frameworks, but there is also need for significant design innovation in terms of tools and curricular activities. Code-first learning environments aim to promote CT in science education by making the construction of computational models accessible to non-programmer students (Wilkerson-Jerde, 2010; Horn et al., 2014). They achieve this by providing students with domain-specific programming primitives that abstract away the underlying programming logic and formal expressions. Coding their own models affords students the ability to express intuitive ideas about scientific phenomena through a formal computational representation and to "debug" their own thinking along the way (Papert, 1980). Research on code-first learning environments is still in its infancy, but it is accelerating thanks to a recent focus on bringing CT practices into the STEM classroom (e.g., Wilensky, Brady & Horn, 2014; Weintrop et al., 2015), infrastructures such Behaviour Composer (Kahn, 2007), DeltaTick (Wilkerson & Wilensky, 2010), NetTango (Horn & Wilensky, 2012), and early curricular designs such as the Frog Pond (Horn et al., 2014; Guo et al., 2016) and EvoBuild (Wagh & Wilensky, 2017).

In this paper, we present the first iteration of a design-based research experiment (Cobb et al., 2003) in which we followed the examples of Frog Pond and EvoBuild to create a code-first learning environment for the kinetic molecular theory (KMT) as part of a new high school agent-based chemistry unit, adapted from the NetLogo Connected Chemistry unit (Stieff & Wilensky, 2003; Levy & Wilensky, 2007). The original Connected Chemistry (CC'1) unit guided students in model-based inquiry wherein they explored the behavior of different model scenarios. The unit was proved effective in that students made gains on AAAS assessments, and also were able to connect the micro-level physical interactions with the macro-level phenomena (Levy & Wilensky, 2007; 2009). We used Weintrop et al.'s (2015) CT-STEM taxonomy as our unit's design framework. The most challenging aspect of operationalizing the CT-STEM taxonomy was to address the "computational problem solving" category, which included practices such as "troubleshooting and debugging," "programming," and "creating computational abstractions."

We hypothesized that creating a new introductory lesson in which the students were introduced to KMT through a programming activity would be a valuable outcome because it would allow students to express their prior understanding about the behavior of gas particles at the microscopic level by constructing an agent-based model with NetLogo (Wilensky, 1999a). KMT was an important topic for us because it is taught universally, yet research shows that students have difficulty in making sense of the variety of phenomena exhibited. Moreover, students come to the classroom with a number of incorrect conceptions that stay intact despite formal instruction (Lin & Cheng, 2000). Smith et al. (1994) argue that it is essential to bridge students' prior conceptions with formal scientific explanations to facilitate meaningful and robust learning (Smith et al., 1994). Otherwise, students will leave the classroom perhaps able to answer test questions according to the formal scientific explanation, but they will keep relying on their intuitions when making sense of real-world phenomena (diSessa, 1993; 2015). However, it is also not easy to design a code-first learning environment that would enable students to "teach computers how they think" about KMT because designing custom code-blocks for this topic is challenging especially when it comes to particle-particle elastic collisions. Such calculations require command of vector mathematics. Computationally, one needs to know concepts such as variables and collision detection. These skills are typically not expected from learners with minimal-to-no programming experience.

In this paper, we present the preliminary results of a design experiment on developing a code-first gas particle sandbox and supporting learning activities for KMT which culminated in a new blocks-based programming paradigm that we call "phenomenological programming." We designed

higher-level code blocks not as procedural commands, but as phenomenological statements that leverage students' intuitive knowledge of real-world events, objects, and patterns. For example, we designed a "bounce" block that can be modified with phenomenological statements such as "like a football" or "like a billiard ball." A particle that bounced *like a football* lost kinetic energy on impact and changed direction randomly, while one that bounced *like a billiard ball* conserved momentum and bounced back at a reflective angle. In what follows, we describe our design experiment in detail, including the preliminary results of a research implementation in which two high school teachers taught KMT to 121 high school students using the code-first gas particle sandbox. We begin with the theoretical underpinnings of our study. We then present the design of our learning activities and the idea of phenomenological programming in detail. Lastly, we describe our first classroom implementation and present vignettes from our data that show how the students engaged with phenomenological programming.

Theoretical framework

Constructionism

Our research is situated within the greater constructionist learning paradigm which maintains that learners construct and learn strong mental models when they engage in constructing personally meaningful, public entities (Papert, 1980; 1991). While constructionist literature is rich with many branches of study, computers tend to play a significant role because they afford learners the opportunity to construct a wide range of dynamic models that can be easily inspected, manipulated, and debugged. Papert and colleagues' design of the Logo programming language (Papert, 1971; 1980) and development environment is the most influential example of a constructionist, code-first learning environment. The primary way to interact with Logo is by programming. The primitives of the language (i.e., commands, branching statements) are designed to be easy to learn by children as young as grade school level. Studies show that Logo promotes powerful learning, especially in mathematics and geometry (e.g., Harel & Papert, 1990; Hoyles & Noss, 1992). Our approach takes inspiration from four constructionist ideas: syntonic learning (Papert, 1980), embodied modeling (Wilensky & Reisman, 2006), code-first learning environments (Horn et al., 2014; Kahn, 2007; Wilkerson & Wilensky, 2010; Wilkerson-Jerde et al. 2015), and blocks-based programming (Bagel, 1996; Bau & Bau, 2015; Resnick et al., 2009; Weintrop & Wilensky, 2015).

Syntonic learning refers to Papert's design of the original Logo turtle, which was *body-syntonic* and *ego-syntonic* (1980). The turtle was controlled by primitives such as *forward* and *right* that relate to children's sense of their own bodily interactions with the physical world. It was also designed to be coherent with children's sense of themselves as people with intentions, goals, and desires. Topics that are counterintuitive when taught formally, such as the definition of a "circle," can be expressed in Logo in a way that is *natural* for children and grounded in their embodied schema (Fig. 2).

Embodied modeling research shows that when learners 'knowledge of individual objects is aligned with their embodied ways of thinking and their own point of view, "they are enabled to think like a wolf, a sheep, or a firefly" (Wilensky & Reisman, 2006). In embodied modeling, learners put themselves in the place of the agents that make up complex systems. For example, when a learner constructs a model of ideal gas laws, they do not solve aggregate-level algebraic equations that are disassociated from the actual underlying real-world events. Instead, they define how each particle behaves autonomously. They take the perspective of a particle and reason that "I would move forward on a straight path," "If I hit a wall, I would bounce back with a straight angle," and so on. Converting embodied agent-rules to a computer simulation using a constructionist agent-based modeling environment such as NetLogo (Wilensky, 1999a) affords learners to see what happens when the same rules are followed by many agents simultaneously. This enables them to connect how micro-level events lead to the emergence of macro-level patterns, properties, and phenomena (Wilensky, 2001). Moreover, they can modify their models easily and test various alternative scenarios, deepening their understanding of scientific phenomena along the way.

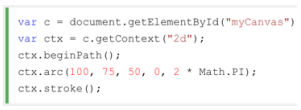
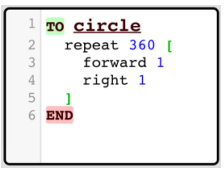
"A circle is a plane figure contained by one line such that all the straight lines falling upon it from one point among those lying within the figure equal one another."	A circle with center (a, b) and radius r is the set of all points (x, y) such that $(x-a)2+(y-b)2=r2$		
(a) The Euclidian definition of a circle	(b) The algebraic definition of a circle	(c) With the JavaScript programming language	(a) With the Logo programming language

Figure 2: Comparing formal definitions a circle with the Logo way

Code-first learning environments are constructionist learning environments that make embodied modeling accessible to students with no prior programming experience (Horn et al., 2014; Kahn, 2007; Wilkerson-Jerde et al. 2015). Research shows that well-designed code-first learning environments promote powerful learning by exposing underlying mechanisms of scientific phenomena better than interacting with pre-existing models or simulations (Guo et al., 2016; Wagh et al., 2016). They achieve this goal by providing students easy-to-learn visual programming environments with pre-composed higher-level primitives that abstract away the underlying complex programming logic. For example, the Modeling4All environment comes with an extensive library of small, independent program fragments called *micro-behaviors* that translate into NetLogo code. In DeltaTick (Wilkerson-Jerde & Wilensky, 2010) and NetTango (Horn & Wilensky, 2012), on the other hand, custom modeling primitive libraries are designed for specific topics or phenomena. For example, the Frog Pond code-first learning environment for natural selection has code-blocks such as "*chirp*", "*hop*", "*hunt*" and "*hatch*" to model the behavior of colorful virtual frogs on a virtual lily pad. This allows students to quickly learn programming and create short programs that result in population-level evolutionary outcomes.

Like many code-first learning environments, we employ the blocks-based programming paradigm. Initially developed by Begel (1996) for the Logo Blocks project in 1996, blocks-based programming is a visual paradigm that represents programming constructs (e.g., commands, branching statements, etc.) as visual blocks that resemble physical Lego blocks. Users assemble algorithms by dragging blocks into a code area and attaching them to each other. Blocks-based programming offers some significant advantages for novice programmers compared to traditional text-based programming languages. For example, it is not possible for students to get side-tracked by syntax errors because the user does not type anything. In addition, there is no need to memorize exact commands because they are always present in the blocks library. Many blocks-based languages such as Scratch and PencilCode even implement visual cues to improve their usability, such as assigning categories of blocks to the same color. In PencilCode, motion blocks (e.g., forward, speed) are blue and control flow blocks (e.g., if-else, key-down) (Bau & Bau, 2016).

Connected Chemistry

Our design of the code-first gas particle sandbox also builds on three decades of studies conducted in CCL beginning with Wilensky's "gas-in-a-box" studies (1999b; 2003) and the subsequent curricular units created for high school chemistry (Stieff & Wilensky, 2003; Levy & Wilensky, 2007;2009; Levy, Novak & Wilensky, 2006; Brady et al., 2014). The study of gaseous matter is particularly suitable for computational modeling because macro-level properties (e.g., temperature, pressure), as well as the scientific laws that describe the relationship between them, are shown to be challenging topics that lead to a multitude of robust misconceptions (Kind, 2004; Lin & Cheng, 2000; Nakhleh, 1992). At school, the study of these relationships often encompasses memorization of equations such as PV = nRT. In contrast, the CCL-developed constructionist units frame these topics in terms of micro-level particle interactions that lead to the emergence of the macro level patterns. Students are guided through computational explorations with agent-based

models instead of equation-based problem-solving activities. These units are still actively used by many high school teachers and previous research implementations showed significant learning gains. For example, in one such unit titled Connected Chemistry 1 (CC1; Levy & Wilensky, 2009), students ran NetLogo models to examine the relationship between gas particle behaviors and key variables like the temperature and pressure. They conducted computational experiments and used statistical methods to derive their own versions of the ideal gas laws. Interviews before and after the intervention showed that the students were able to form multi-level explanations of the chemical system at the end of the unit. In pre-interviews, 84% of the students explained gaseous phenomena only using macroscopic terms. In post-interviews, 85% of them explained the same phenomena in terms of the connections between micro-level interactions and the macro-level patterns (Levy et al., 2004; Levy & Wilensky, 2009).

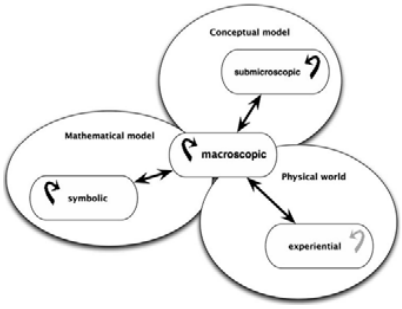


Figure 3: Conceptual framework of the Connected Chemistry 1 (CC1) curriculum (Levy & Wilensky, 2009).

Our code-first gas particle sandbox is inspired by two of these prior units developed at CCL: Levy & Wilensky's Connected Chemistry 1 (CC1) unit and Brady et al.'s (2014) Particulate Nature of Matter (PNoM) unit, part of the ModelSim project (NSF# DRL-1020101). The CC1 unit introduced the idea of a particle sandbox, while the PNoM unit built on CC1 and further developed it to provide students with emergent systems sandboxes (ESSs) (Brady et al., 2014) within which they could construct computational models of dynamic systems that exhibit emergent phenomena without the need for writing any code. For example, in the PNoM unit, students constructed computational models to explore the diffusion of odor in a room when a warm container is opened versus when a cold container is opened. In the sandbox, the students could use a drawing tool to add static walls to represent containers, removable walls to represent valves or doors, and particles that were pre-programmed to move and interact according to kinetic molecular theory (KMT).

Design overview

The design experiment we present here is situated within a greater project to design a new version of the Connected Chemistry Ideal Gas Laws unit, which we call the CC'19 unit (see Aslan et al., 2020a), itself part of the CT-STEM research project (Weintrop et al., 2015; Wilensky et al., 2014). The idea of the code-first gas particle sandbox first emerged as we were designing a brand-new introductory lesson for the CC'19 unit. We had three overarching objectives: a pedagogical objective, a computational objective, and a content-learning objective. Pedagogically, we wanted to bootstrap the rich ideas that students bring into the classroom prior to instruction (diSessa & Minstrell, 1998; Smith et al., 1994). Computationally, we wanted students to be able to express their intuitive understanding of gas particles in terms of simple computer programs. In order to promote chemistry learning, we wanted these activities to build towards the main assumptions of the Kinetic Molecular Theory (KMT) because we wanted students to be able to explain how gas pressure, a macro-level property, emerges from numerous gas particles interaction with each other and the container.

To achieve our pedagogical objective, we designed a beginning activity in which the students examined an air duster can. We chose the air duster because it is a simple real-world object that has a fixed volume and only gas particles inside. The students answered some beginning

questions about what happens when the valve is pressed, and they also illustrated their answers by drawing sketches. The teachers projected each student's sketch on the screen and conducted whole-class discussions. These discussions served as benchmark classes (diSessa & Minstrell, 1998) for teachers to survey the students 'intuitive understanding of gas particles and cultivate an exchange of these ideas among students. The discussions also served as an anchor for the idea of computational modeling because the students 'sketches served as static models of the air duster can.

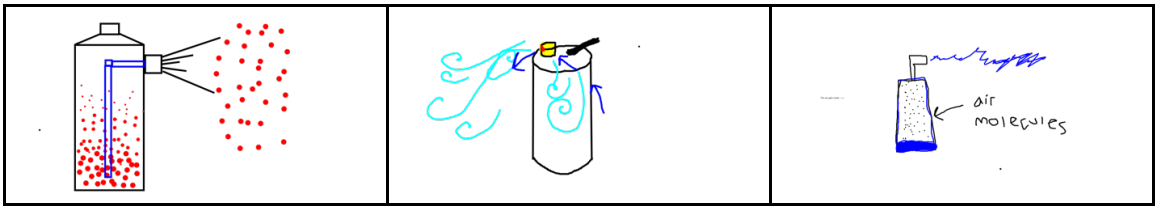
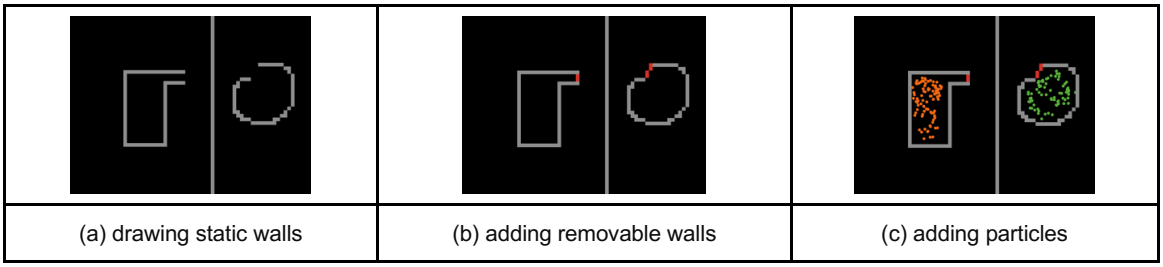


Figure 4: Examples from students' hand-drawn sketches

To achieve our CT objective, we designed a multi-step scaffolded activity. First, the students used a static modeling toolkit that resembled the initial sketching activity. They constructed the initial state of a computer model by adding stationary walls, removable walls, green particles, and orange particles. Second, they used a simplified version of the code-first gas particle sandbox to develop a small-scale model of gas particles (max. 4 particles). Lastly, they re-loaded their static air duster models from the first step into the sandbox to see whether their air-duster model behaved as they anticipated. This process allowed them to design and construct a computational model to test their initial hypotheses.

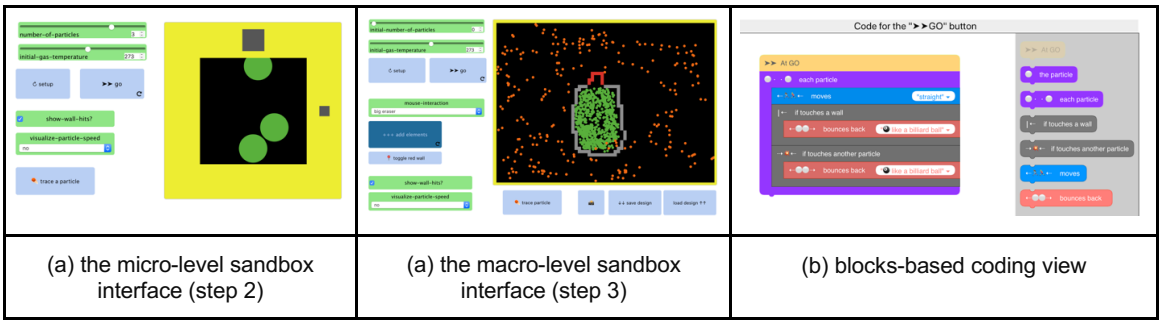


(a) drawing static walls	(b) adding removable walls	(c) adding particles

Figure 5. Sketching a static computational representation of real-world gas containers (step 1).

Designing domain-specific primitives for KMT was a challenging task because we assumed no prior programming experience. A traditional approach would require students to use computational constructs such as variables, vector calculations, and collision detection. Instead, we formulated a new approach that we call *phenomenological programming* (Aslan et al., 2020b). Beside building on the four constructionist ideas that we discussed in the theoretical framework section, phenomenological programming is also partially inspired by diSessa's theory of phenomenological primitives (p-prims in short). By definition, p-prims are "bits of knowledge that contribute to our intuitive 'sense of mechanism;' that is, what kinds of occurrences are natural and to be expected" (diSessa, 1993; diSessa, 2015, p. 34). They are phenomenological because they are encoded non-verbally, probably as images or kinesthetic schemes. The activation of p-prims is instantaneous. They are evident in our daily experience and we see situations in terms of them. They are primitive because we often cannot analyze or justify our p-prims. We hypothesized that code-blocks designed in accordance with students 'p-prims would (1) be easily recognizable for the students, (2) embed implicit assumptions about their function, (3) facilitate easy mental simulation and hence help students express their mental models computationally, and (4) most importantly bridge students 'prior understanding with the formal scientific explorations by facilitating a process of "debugging one's thinking". Furthermore, we hypothesized that code-first learning environments with phenomenological programming could be more approachable for

teachers as well because they would not require a lot of teacher training and they would appeal to teachers with little-to-no prior computing experience.



(a) the micro-level sandbox interface (step 2)	(a) the macro-level sandbox interface (step 3)	(b) blocks-based coding view

Figure 6. The main components of the code-first gas particle sandbox.

We designed phenomenological blocks that provide students with procedural templates such as *moves* and *bounces* that can be modified with phenomenologically transparent statements such as "*spinning*" and "*straight*" for the "*move"* block, and "*like a balloon" and "like a billiard ball"* for the "*bounce"* block. Each statement embeds simple assumptions about gas particles. Some of them embed the assumptions of KMT (e.g., move straight, bounce like a billiard ball), while others correspond to potential misconceptions. For example, KMT assumes that particles move straight until they collide another particle or hit a wall. However, we also designed a "*move erratically*" option because research shows that some students might believe that gas particles change direction randomly and haphazardly without any collision (Kind, 2004). This way, students can quickly start programming the particles minimal introduction to programming and without the challenging task of converting their intuitive understanding of gas-particles to formal computer-code. The blocks would be instantly recognizable for them and they can hypothesize about the outcome of the code they put together because it would be easy to mentally simulate the movement of gas particles.

A summary of each code-block we designed for the code-first gas particle sandbox is presented in Table 1. There are only 7 code-blocks in this iteration of the code-first gas particle sandbox (Figure 7). However, combined with the static freehand modeling tools (Figure 5) and the phenomenological sub-statements, these blocks are enough to develop very complex and detailed gas particle simulations. Moreover, they are sufficient to create conflicts between students' intuitive understanding of micro-level gas particle behavior and macro level patterns. For example, if a student chose to make the particles move in circles (i.e. spinning) and collide elastically, it would still generate close-to-expected behavior at the micro-level, yet when tested with an air-duster design at the macro level, the particles would not leave the valve as intended. Similarly, particles that bounce like basketballs may slow down so gradually that students who do not run their micro-level models long enough may not notice the difference initially, but they would notice it when they run their models at macro level with hundreds of particles. Prior research shows that such conflicts can be great opportunities for students to debug their intuitive understanding of gas particles 'behavior as well as bridge their intuitive knowledge with the formal scientific explanations (Wilensky, 1999b; 2003; Levy & Wilensky, 2009).

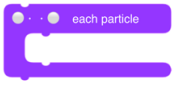
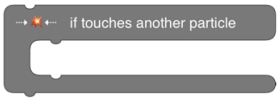
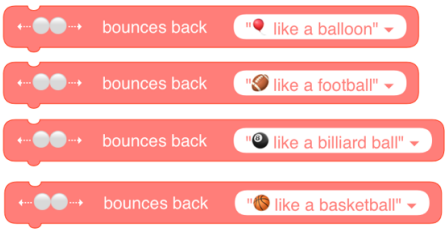
Block	Explanation
	Procedural block. Code that is attached to this block is executed in a continuous loop when the GO button of the model is clicked.
	Procedural block. The code encapsulated by this block is executed only by the selected particle (e.g., particle 1, particle 10, particle 87).
	Procedural block. The code encapsulated by this block is executed by all particles separately and autonomously.
	Procedural block. The code encapsulated by this block is only executed when a particle is touching a container wall.
	Procedural block. The code encapsulated by this block is only executed when a particle is touching another particle.
	Phenomenological block. If a particle is executing this code, it moves 1 unit forward based on the chosen phenomenological statement: Straight: Moves forward 1 unit without changing direction. Spinning: Moves forward 1 unit, changes direction to follow a circular path. Zig-zag: Moves forward 1 unit, changes direction to follow a zig-zag path. Erratic: Moves forward 1 unit, changes direction to follow a path that resembles random walk.
	Phenomenological block. If a particle is executing this code, it changes its momentum and kinetic energy based on the chosen phenomenological statement: Like a balloon: Changes direction as if it is an elastic collision. If collides with another particle, exchanges momentum as if it is an elastic collision. Total kinetic energy is decreased significantly. Recalculates its speed based on its kinetic energy. Like a football: Changes direction randomly. If collides with another particle, exchanges momentum as if it is an elastic collision. Total kinetic energy is decreased slightly. Recalculates its speed based on its kinetic energy. Like a billiard ball: Changes direction as if it is an elastic collision. If collides with another particle, exchanges momentum as if it is an elastic collision. Total kinetic energy is preserved. Recalculates its speed based on its kinetic energy. Like a basketball: Changes direction as if it is an elastic collision. If collides with another particle, exchanges momentum as if it is an elastic collision. Total kinetic energy is decreased slightly. Recalculates its speed based on its kinetic energy.

Figure 7. The function of the code-blocks of the code-first gas particle sandbox and the assumptions embedded in the phenomenological blocks

Research implementation

4.1. Participants & Settings: Our study is a design experiment as outlined by Cobb et al. (2003). We designed our code-first gas particle sandbox over multiple iterations. The initial versions were tested by our research team members. Then we met with the two teachers who were going to implement the lesson and updated our design according to their feedback. The first research implementation of our design experiment took place in Spring 2019. Two teachers and a total of 121 high-school regular chemistry students at a U.S. Midwest public high school participated (Table 2). The implementation lasted a total of 10 class periods over the course of 8 days. The students used Chromebooks to access the lesson over the CT-STEM student portal. The final version of this lesson can be accessed through the CT-STEM webpage (https://ct-stem.northwestern.edu/curriculum/preview/513/0/).

4.2. Data collection and analysis: We collected all the open-ended written responses and sketches students posted on the portal. In addition, we asked students to upload screenshots of their static container models and blocks-based algorithms. In order to gain further insight on the students ' thought processes and the interactions between them, we video recorded four focus groups each containing 2 or 3 students. We also attended each class, took field notes, and sometimes even walked around the classroom and asked some non-focus group students to do quick demos of their work on video. Here, we present a preliminary analysis of this data through vignettes from students 'block-codes and excerpts from the video data.

Preliminary findings

We observed that almost all of the students successfully engaged in phenomenological programming. We begin presenting our findings with some examples from the students 'blocks-based algorithms. Figure 8 provides four snapshots from the students 'blocks-based algorithms after the step 2 (micro-level modeling) and their own explanations on why they chose specific code blocks. We chose to ask the students to upload their code at this step, not at the end, because we wanted to observe their assumptions about individual gas particles before they tested their code with hundreds of particles. This data is valuable to show how students 'programming of micro-level gas particles was informed by their intuitive understanding of macro-level phenomena. We also include the students 'own explanations on the right side of Figure 8 to highlight their reasoning behind choosing the specific coding blocks and phenomenological explanations.

Each example in Figure 8 describes particle movement differently and each student has different reasoning behind their coding decisions. However, we argue that one trend is salient: the students' programming decisions are informed by what kinds of occurrences they found to be natural and expected (sense of mechanism; diSessa, 1993) instead of the formal science terms or explanations. For example, the Student #2 reasons that "*there is not one way each particle moves each time*", while the Student #3 reasons that "*straight*" movement is the "*most realistic*" one. Three of the students believe that particles bounce like billiard balls. One of them specifically reasons that billiard balls do not lose energy when they bounce off of the walls and other particles. Another student simply states that "*they don't slow down*." The last student, on the other hand, neither mentions direction nor speed. In contrast, the Student #2 takes her understanding of a football that they "*don't always have the same direction come back*", while not mentioning the energy exchange at all. Overall, we observe that the students were comfortable in expressing their intuitive understanding of gas particles through the phenomenological blocks. More importantly, there was a great diversity of algorithms created by the students and the underlying student reasoning. Given the apparent difficulty of expressing their intuitive understanding even verbally, it is encouraging that these students could do so computationally.

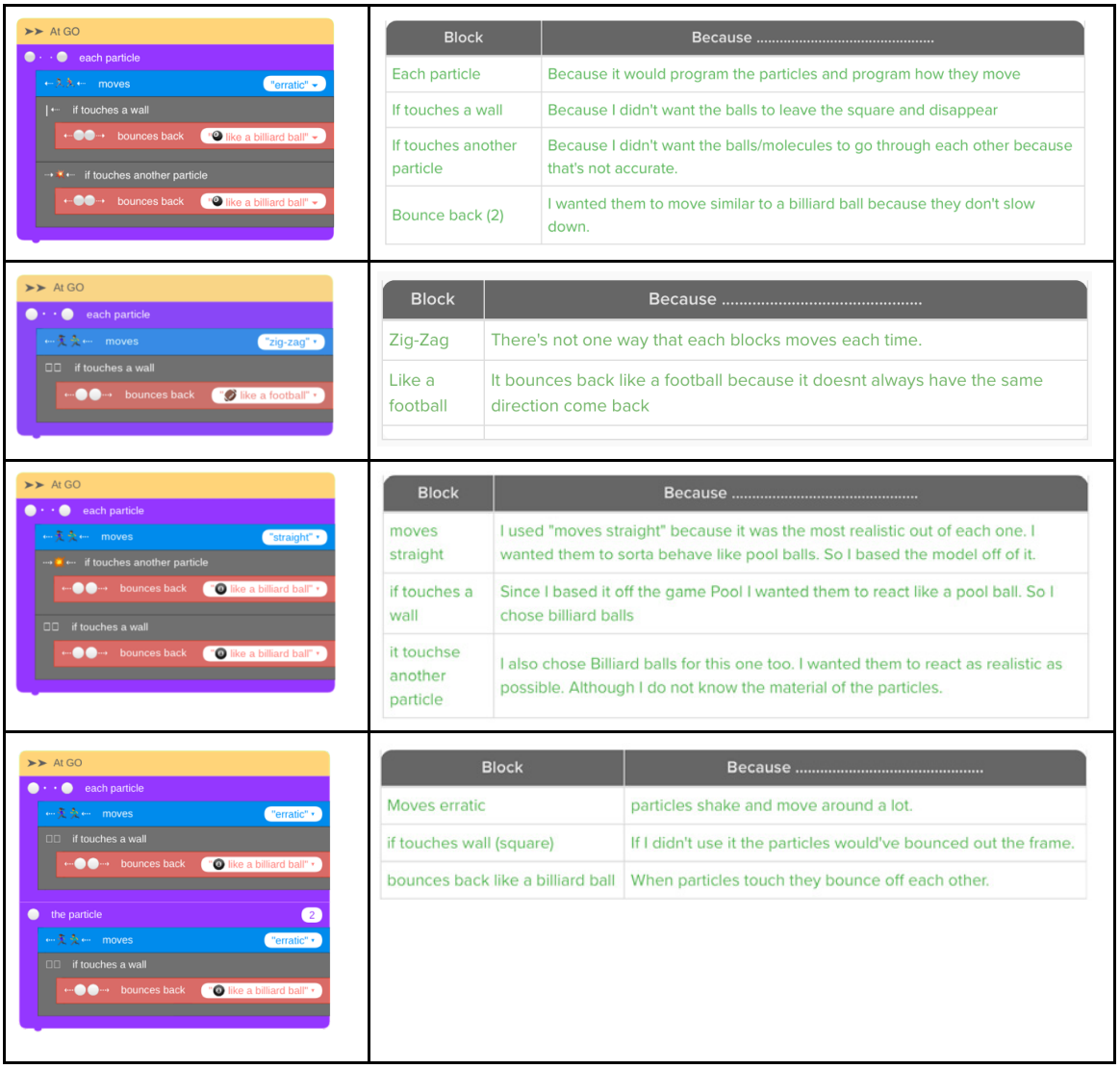
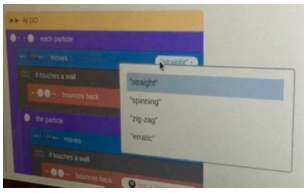
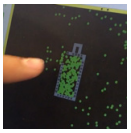


Figure 8. Three examples from the students' blocks-based algorithms and with their own explanations

The video data from quick student demonstrations also showed that the students 'programming process was highly influenced by their sense of mechanism. Moreover, we were able to observe how students debugged their code after to the conflicts between their assumptions about the micro-level gas particle behavior and macro level outcomes. In the Excerpt 1 below, the Student #4 from the Figure 8 explains her computational modeling process and how in the last step she had to fix two major bugs in her project.

Dialogue	Video Snapshot (cropped)
Researcher: So, this was your original static model, right?	
Student: Yeah. And first I added the red wall.	
Student: And then another change I made was change the direction it moved. *It was erratic for both and then I changed it to straight.* Researcher: Why did you change it? Student: Because I wanted to represent, like, the way the particles move once it comes out the spray can. Researcher: They weren't spraying out as you liked?	
Student: Oh no. *It was just like, spreading out, instead of going towards one direction and then spreading out.* And then, (runs the model with the red wall closed) this is how it was. It was all cramped in there. And then when I took off the red wall, it's all going in one direction and it spreads out, which is what I wanted to show.	

Except 1. A student's explanation of her programming process to the researcher

The first one was a simple design bug: she forgot to design a removable wall to represent the valve. This might also be caused by the activity prompts on the lesson itself or the teachers ' omission. She explains how she solved it quickly. The second one, though, is directly related to her assumptions about the specific particle movement pattern and how it exhibits itself at the macro level. In this quick demo, she explains to the researcher that she initially made her particles move "erratically." As her explanation on Figure 8 shows, she thought that "particles shake and move around a lot." However, in the last step of the activity, when she tried to run her simulation, she noticed that the particles did not behave like they do in real life. In other words, they did not spray in one direction, but they spread out. This prompted her to make the particles move straight instead. In addition, to a question on the portal that asked if their air duster model worked as expected, she responded: "*At first it didn't work as I expected it to, because the particles were spreading into the air which is what I didn't really want to represent. So, I changed my move block from erratic to straight to show the pressure of the air and how they spread into the air.*"

In the end, this student did not only fix her model, but she also debugged her own thinking during the process and learned about how simple micro-level behavior may result in surprising macro-level patterns. She did this all while she did not have to learn formal theories, solve equations, or even articulate her ideas coherently. This is not only a desired outcome for computational modeling, but a very critical idea for learning kinetic molecular theory. We even observe that she uses the term "pressure" in her reasoning about the changes she made in the code.

Discussion and future work

Our design-based research study is still in its early stages and the code-first gas particle sandbox we present in this paper is our first attempt at creating phenomenological code-blocks. The findings we present are preliminary, with further analysis needed to inform the next iteration of our design in order to further study and clarify our findings. Nevertheless, we are encouraged both by the ease with which students were able to start programming with the phenomenological blocks

and how well the blocks corresponded to the kinds of occurrences they found natural and to be expected. We are also encouraged by how the student presented in Excerpt 1 was able to *debug* her own thinking as she debugged her model. We argue that such experiences with code-first learning environments help students develop a "feel" of real-world phenomena that is better aligned with correct scientific explanations. In our future work, we will conduct more research implementations and collect more data on how students interact with the code-first gas particle sandbox and the impact of these interactions on their learning. We will continue to improve our design and explore principles for the design of new phenomenological blocks. We are optimistic that if successful, code-first learning environments with phenomenological programming can accelerate the diffusion of computational thinking practices and powerful learning in science and mathematics classrooms.

Acknowledgements

This work was made possible through generous support from the National Science Foundation (grants CNS-1138461, CNS-1441041, DRL-1020101, DRL-1640201 and DRL-1842374) and the Spencer Foundation (Award #201600069). Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.

References

Aslan, U., LaGrassa, N., Horn, M., & Wilensky, U. (2020a). Putting the Taxonomy into Practice: Investigating Students' Learning of Chemistry with Integrated Computational Thinking Activities. *American Education Research Association (AERA) 2020 Annual Meeting*. San Francisco, CA.

Aslan, U., LaGrassa, N., Horn, M., & Wilensky, U. (2020b, *in press*). Phenomenological Programming: a novel approach to designing domain specific programming environments for science learning. *Paper accepted to the ACM Interaction Design and Children (IDC) conference*. London, UK.

Begel, A. (1996). LogoBlocks: A Graphical Programming Language for Interacting with the World (Doctoral dissertation). Retrieved from https://llk.media.mit.edu/papers/archive/begel-aup.pdf

Bau, D., & Bau, D. A. (2014). A preview of Pencil Code: A tool for developing mastery of programming. *Proceedings of the 2nd Workshop on Programming for Mobile & Touch* (pp. 21-24).

Brady, C., Holbert, N., Soylu, F., Novak, M., & Wilensky, U. (2014). Sandboxes for model-based inquiry. *Journal of Science Education and Technology*, *24*(2-3), 265-286.

Cobb, P., Confrey, J., DiSessa, A., Lehrer, R., & Schauble, L. (2003). Design experiments in educational research. *Educational researcher*, *32*(1), 9-13.

diSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and instruction*, *10*(2-3), 105-225.

diSessa, A. (2015). Alternative conceptions and P-prims. *Encyclopedia of science education*.

diSessa, A. & Minstrell, J. (1998). Cultivating conceptual change with benchmark lessons. *Thinking practices in learning and teaching science and mathematics*, 155-187.

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational researcher*, *42*(1), 38-43.

Guo, Y., Wagh, A., Brady, C., Levy, S. T., Horn, M. S., & Wilensky, U. (2016, June). Frogs to Think with: Improving Students' Computational Thinking and Understanding of Evolution in A Code-First Learning Environment. In *Proceedings of the 15th International Conference on Interaction Design and Children* (pp. 246-254). ACM.

Harel, I., & Papert, S. (1990). Software design as a learning environment. *Interactive learning environments*, *1*(1), 1-32.

Horn, M. S., & Wilensky, U. (2012). NetTango: A mash-up of NetLogo and Tern. *AERA 2012*.

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

Hoyles, C., & Noss, R. (1992). A pedagogy for mathematical microworlds. *Educational studies in Mathematics*, *23*(1), 31-57.

Kahn, K. (2007). Comparing multi-agent models composed from micro-behaviours. *M2M*,165-177.

Kind, V. (2004). Beyond appearances: Students 'misconceptions about basic chemical ideas.

Lin, H. S., Cheng, H. J., & Lawrenz, F. (2000). The assessment of students and teachers' understanding of gas laws. *Journal of Chemical Education*, *77*(2), 235.

Levy, S. T., Kim, H., & Wilensky, U. (2004, April). Connected chemistry—A study of secondary students using agent-based models to learn chemistry. In *annual meeting of the American Educational Research Association, San Diego, CA* (pp. 12-16).

Levy, S., & Wilensky, U. (2007). How do I get there... straight, oscillate or inch? High-school students' exploration patterns of Connected Chemistry. In *2007 meeting of the American Educational Research Association*.

Levy, S. T., & Wilensky, U. (2009). Crossing levels and representations: The Connected Chemistry (CC1) curriculum. *Journal of Science Education and Technology*, *18*(3), 224-242.

Nakhleh, M. B. (1992). Why some students don't learn chemistry: Chemical misconceptions. *Journal of chemical education*, *69*(3), 191.

Papert, S. A. (1971). Teaching children thinking. In *Proceedings of IFIPS World Congress and Education.* Amsterdam, Netherlands.

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

Papert, S. (1991). Situating constructionism. *Constructionism*, *36*(2), 1-11.

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

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.

Smith III, J. P., diSessa, A., & Roschelle, J. (1994). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The journal of the learning sciences*, *3*(2), 115-163.

Stieff, M., & Wilensky, U. (2003). Connected chemistry—incorporating interactive simulations into the chemistry classroom. *Journal of Science Education and Technology*, *12*(3), 285-302.

Wagh, A., & Wilensky, U. (2018). EvoBuild: A QuickStart Toolkit for Programming Agent-Based Models of Evolutionary Processes. *Journal of Science Education and Technology*, *27*(2), 131-146.

Wagh, A., Levy, S., Horn, M. S., Guo, Y., Brady, C., & Wilensky, U. (2017, June). Anchor Code: Modularity as Evidence for Conceptual Learning and Computational Practices of Students Using a Code-First Environment. In *Computer Supported Collaborative Learning (CSCL)*.

Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 199-208). ACM.

Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education*, *18*(1), 3.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, *25*(1), 127-147.

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

Wilensky, U. (1999b). GasLab—An extensible modeling toolkit for connecting micro-and macro-properties of gases. In *Modeling and simulation in science and mathematics education* (pp. 151-178). Springer, New York, NY.

Wilensky, U. (2001, August). Modeling nature's emergent patterns with multi-agent languages. In *Proceedings of EuroLogo* (pp. 1-6).

Wilensky, U. (2003). Statistical mechanics for secondary school: The GasLab multi-agent modeling toolkit. *International Journal of Computers for Mathematical Learning*, *8*(1), 1-41.

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.

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

Wilkerson-Jerde, M., Wagh, A., & Wilensky, U. (2015). Balancing Curricular and Pedagogical Needs in Computational Construction Kits: Lessons from the DeltaTick Project. *Science Education*, *99*(3), 465-499.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33-35.