

Phenomenological Programming: A Novel Approach to Designing Domain Specific Programming Environments for Science Learning

Umit Aslan¹

Nicholas LaGrassa¹

Michael Horn^{1,2}

Uri Wilensky^{1,2}

Northwestern University

¹Learning Sciences, ²Computer Science

2120 Campus Drive, Evanston, Illinois 60208 USA

{umitaslan, nick.lagrassa}@u.northwestern.edu

{michael-horn, uri}@northwestern.edu

ABSTRACT

There has been a growing interest in the use of computer-based models of scientific phenomena as part of classroom curricula, especially models that learners create for themselves. However, while studies show that constructing computational models of phenomena can serve as a powerful foundation for learning science, this approach has struggled to gain widespread adoption in classrooms because it not only requires teachers to learn sophisticated technological tools (such as computer programming), but it also requires precious instructional time to introduce these tools to students. Moreover, many core scientific topics such as the kinetic molecular theory, natural selection, and electricity are difficult to model even with novice-friendly environments. To address these limitations, we present a novel design approach called *phenomenological programming* that builds on students' intuitive understanding of real-world objects, patterns, and events to support the construction of agent-based computational models. We present preliminary case studies and discuss their implications for STEM content learning and the learnability and expressive power of phenomenological programming.

Author Keywords

computational thinking; constructionism; novice-friendly programming environments; chemistry education; agent-based modeling

CSS CONCEPTS

• **Applied computing**; *Education*; Interactive learning environments;

INTRODUCTION

Both the 2012 National Research Council's framework for K-12 Science Education [25] and the Next Generation Science Standards [26] emphasize the importance of incorporating scientific modeling practices beginning with the earliest grades and with increasing sophistication at each grade. Research shows that computer-based modeling approaches (e.g., as agent-based modeling [42], system dynamics modeling [4]) provide some of the most powerful means to engage with scientific modeling especially when students are programming their own models [e.g., 36, 37, 43, 44]. Thanks to renewed interest in computational thinking as a fundamental skill set for every child to develop [38, 45], educators and stakeholders are more willing to invest in integrating computer programming activities into curricular activities [9, 38]. However, the adoption rate remains low because many science teachers either lack the programming skills associated with computer-based modeling or lack the classroom time to introduce the necessary skills to students (or both). In addition, even the most novice friendly general-purpose programming environments become too complicated when it comes to modeling high-school topics such as magnetism, evolution, or chemical reactions.

Domain-specific programming environments offer more limited capabilities than general purpose programming languages, but they make up for this drawback with custom-designed primitives (e.g., commands, branching statements, events) that are closely matched to specific curricular topics [10, 12, 17, 36, 37, 44]. A small number of primitives that act as self-contained micro-behaviors enable students to construct sophisticated models of scientific phenomena with very little programming instruction [17, 37]. For instance, the Frog Pond learning environment for natural selection has primitives such as *hop*, *hunt*, *spin*, and *chirp* to model the behavior of colorful virtual frogs living in a pond with lily pads [10, 12]. Each of these primitives translate to corresponding NetLogo [39] code in the runtime. Early results from empirical studies with domain-specific programming environments are encouraging [e.g., 10, 37], but some of the more challenging science topics do not

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IDC '20, June 21–24, 2020, London, United Kingdom.

© 2020 Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7981-6/20/06...\$15.00.

DOI: <https://doi.org/10.1145/3392063.3394428>

readily lend themselves to this approach. For example, modeling macro-level phenomenon such as gas pressure require calculating the momentum exchange for each particle-to-particle collision using two-dimensional vectors, which require the use of programming constructs like state variables and trigonometric functions. It is not possible to design self-contained micro-behaviors for such complex programming logic using the conventional procedural programming paradigm.

We propose *phenomenological programming* as a novel approach to designing domain-specific programming environments for science learning. In phenomenological programming, high-level primitives are designed based on students' intuitive understanding of the real-world objects, patterns, and events. Each primitive is made up of two components: a procedural component that indicates function and a phenomenological statement that indicates behavior. Each phenomenological statement embeds assumptions about the modeled objects that students may be intuitively aware of but (1) may not have coherent knowledge, and more importantly (2) may not have competence in formal notations or domain-specific vocabulary to express their thoughts explicitly and unambiguously. It is easy to recognize the behavior of a phenomenological command and predict the outcome through a sort of quick mental simulation. It is also possible to include small doses of ambiguity that can be beneficial for novice learners. For example, students can easily recognize a "move" command with a phenomenological component called "zig-zagging" and predict its behavior even if figuring out its exact nature would require some tinkering. Achieving the same outcome in general-purpose procedural programming languages would require mastering the use of sophisticated constructs such as loops, state variables, and angles.

In this paper, we present the theoretical underpinnings and a preliminary definition of phenomenological programming. We also present a design intervention in which we created a phenomenological programming environment for a high-school ideal gas laws unit. We call this environment *the phenomenological gas particle sandbox*. Its primitives include commands such as "move" and "bounce" that can be modified with phenomenological statements such as "zig-zagging" or "erratically" and "like a balloon" or "like a football" respectively. We present our preliminary findings from an implementation in which two teachers taught the kinetic molecular theory to a total of 121 high-school general chemistry students with curricular activities designed around the phenomenological gas particle sandbox. We discuss the implications of our findings for future research.

THEORETICAL FRAMEWORK

Constructionism

Phenomenological programming is situated within the greater Constructionist learning paradigm, which maintains that learners construct robust mental models when they actively construct personally meaningful public entities [28,

29, 30]. Computer programming has historically been the focus of many Constructionist studies because it allows the construction of *microworlds*: self-contained virtual worlds with limited number of objects and operations to explore concepts, theories, and phenomena [29]. Empirical findings show that Constructionist science learning environments facilitate powerful conceptual and formal learning through constructing dynamic virtual models of real-world phenomena that can be built bottom-up, scrutinized, manipulated, experimented with, iterated over, repurposed, and shared with others to receive feedback [e.g., 37, 43]. Our studies on phenomenological programming attempt to make this modality more accessible to students and teachers who have little-to-no prior exposure to computer programming.

Constructionist literature is rich with many branches of study. Phenomenological programming is particularly inspired by four Constructionist design ideas: syntonetic learning [29], embodied modeling [43], domain-specific programming environments [12, 37], and non-textual programming paradigms [1, 16, 17, 27].

Syntonetic learning refers to Seymour Papert's design of the original Logo programming language and its object-to-think-with, the turtle [29]. In Logo, a screen turtle is controlled by body-syntonetic programming primitives such as forward, right, and pen-down that relate to children's sense of interacting with the physical world around them. The turtle is also ego syntonetic. It is designed to be coherent with children's sense of themselves as conscious agents 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 more natural for children and grounded in their embodied schema thanks to the turtle's syntoneticity. Syntonetic learning provides a foundational theoretical explanation on how to design programming environments that can accommodate children's intuitive ways of thinking.

Our goal in phenomenological programming is to make the construction of computer-based models more prevalent in science education. Embodied modeling provides us a theoretical framing on how to design constructionist learning environments towards this goal. Initially formulated by Wilensky and Reisman [43] for biology learning, the embodied modeling approach argues that when learners' knowledge of individual organisms that make up greater ecosystems is aligned with their embodied ways of thinking, they can start to take the perspective of those organisms they are modeling. This would lead to better understanding of how interactions at the micro-level may lead to patterns that we observe at the macro-level in phenomena such as evolution, homeostasis, and epidemics [41, 42]. The concepts of complex systems theory play a significant role in embodied modeling environments. For example, when learning about the behavior of gas particles inside a container, students do not only solve aggregate-level algebraic equations that are disassociated with the actual

underlying micro-level events. Instead, they define how each particle behaves autonomously. For example, 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 such embodied agent-rules to dynamic models using a constructionist agent-based modeling environment such as NetLogo [39] allows learners to observe what happens at the aggregate level when the same rules are followed by many agents simultaneously. Moreover, the resulting models afford testing various alternative scenarios, deepening their understanding of scientific phenomena along the way. Studies show that embodied modeling does not only contribute to better conceptual understanding, but also helps students understand the relationship between scientific phenomena and their symbolic representations (e.g., equations, diagrams) [e.g., 19, 20, 43, 46].

However, general purpose agent-based modeling environments such as NetLogo require preliminary programming instruction for students to start constructing their own models. Thus, we adopt the domain specific approach in phenomenological programming. In domain-specific programming environments, students are provided with higher-level, domain-oriented primitives that abstract away the underlying complex programming logic [17, 38]. This limits the capabilities of the programming environment but makes it easier to start programming for novice learners because the primitives are more familiar and more aligned with the studied subject matter. For example, the Frog Pond environment for learning natural selection has code-blocks such as “chirp”, “hop”, “hunt” and “hatch” to construct embodied models of colorful virtual frogs on a virtual lily pad [12]. Students quickly learn how to program in the Frog Pond and construct short programs that result in population-level evolutionary outcomes [10, 12]. Research shows that domain-specific programming environments can offer unique affordances for science learning by exposing underlying mechanisms of scientific phenomena better than interacting with pre-existing models or simulations [37].

Lastly, our initial implementation of phenomenological programming is influenced by prior research on visual programming paradigms such as blocks-based programming [1], programming by demonstration [16], and visual scripting [27] because they offer significant affordances for novice programmers. For example, most visual programming environments do not require learners to memorize specific words or statements because there is an always-present library. Programs are assembled not by typing, but by visually arranging these primitives such as dragging code blocks and attaching them to each other. There are also visual cues such as shapes and colors that communicate features and functionality like category (e.g., motion, control) and function (e.g., reporter, command) [1]. Such features help ease the cognitive load and the anxiety related to formal expressions, leaving more room for learners to focus on their algorithms. In our current implementations

of phenomenological programming, we use the NetTango builder environment [11, 40], which provides a domain-specific, blocks-based layer over the NetLogo agent-based modeling environment [39].

The Role of Intuitive Knowledge in Science Learning

Phenomenological programming is also informed by the theoretical perspectives on the relationship between students’ intuitive knowledge and their science learning. More often than not, the primary challenge in science education research is to resolve the conflict between learners’ intuitive understanding of real-world phenomena and the formal scientific explanations [6, 34]. Topics such as electric current or diffusion are difficult for students to learn because the underlying formal scientific explanations contradict novice learners’ expectations about how real-world objects behave. For example, many students see the diffusion of a drop of ink in water as a deliberate process, hypothesizing that ink molecules are moving towards less concentrated areas, while the actual underlying events are completely random [2]. In addition, even when students learn to answer formal questions using formulas and symbolic representations competently, they often revert back to their prior, faulty intuitive understandings when they are asked conceptual questions [5,21,22].

Phenomenological programming inherits the constructivist principle that students construct more sophisticated knowledge from already-existing knowledge elements [6, 7, 29, 30, 32]. Hence, our goal is to find ways to help students learn science by bridging their prior understanding of real-world phenomena with formal scientific explanations. diSessa and Minstrell argue that students’ initial ideas about a topic should be used as scaffolds and reference markers for learning formal scientific explanations [6]. On the other hand, there is great diversity in theoretical characterizations of students’ intuitive thinking. Although we cannot attribute our theoretical framing to one specific theory, we are informed by the theories on mental models [14, 15, 35] and the piecemeal nature of knowledge [5, 23, 34]. The mental models theory argues that when humans perceive the world, they construct a similar but less sophisticated dynamic mental representation; a “simulation of the world fleshed out with our knowledge” [15, p. 18243]. The implication of the mental models theory for phenomenological programming is that students should be able to mentally simulate their programming actions based on their existing knowledge. On the other hand, the theories on piecemeal nature of knowledge, such as diSessa’s Knowledge-in-pieces theory [5] and Minsky’s Society of Mind theory [23], are concerned with the composition of mental structures. They characterize human thinking as composed of numerous smaller knowledge elements that are organized in various arrangements. The implication for phenomenological programming is that students’ intuitive understanding of scientific phenomena should not be considered as monolithic, disconnected mental entities, but as rich and interconnected conceptual ecologies. Expertise in a scientific

topic does not mean acquiring new knowledge completely disconnected to prior intuition but constructing new knowledge as part of this conceptual ecology; acquiring new knowledge elements, while also reorganizing or repurposing the old pieces of intuitive knowledge [6, 7, 33].

A PRELIMINARY DEFINITION OF PHENOMENOLOGICAL PROGRAMMING

We envision phenomenological programming as both a new approach to integrating programming activities into science lessons and a greater design framework on how to achieve this goal. In principle, phenomenological programming environments leverage target group students' familiarity with the physical world, including objects, patterns, and dynamic events, in creating custom-designed, domain-specific programming activities. However, this is a rather broad definition. Our focus in this paper is narrower; we concentrate on domain-specific, block-based implementations of agent-based modeling activities.

Our phenomenological code blocks are procedural as in traditional domain-specific programming environments. They perform a specific task. Yet they are also modifiable with sub-statements that embed assumptions about the nature of the performed task. For example, let's consider a *move* command that makes an object change its position. In a traditional domain-specific programming environment, the *move* command would always make the object move the same amount regardless of the object's location on the screen or its direction. In a phenomenological programming environment, the *move* command would behave differently depending on its context and its phenomenological modifier (e.g., zigzag, spinning). For example, many students would easily recognize that *zig-zagging* objects should go straight for a short distance, then suddenly change their direction, go straight for a short distance again, and repeat this process back and forth while maintaining a linear trajectory. Moreover, the zig-zag command would behave differently based on each object's speed. If an object is moving fast, the zigzags would be larger, and vice versa.

Our research on phenomenological programming is on its early stages. We have not yet reached a definitive set of design principles or guidelines. However, we compiled a retrospective summary of the steps we followed to design phenomenological code blocks once we hypothesized about the plausibility of phenomenological programming:

1. Analyze the target scientific phenomenon and documented student thinking patterns such as:
 - a. Prevalent misconceptions and learning difficulties
 - b. Challenging aspects of the formal science and mathematics concepts
 - c. Challenges in thinking across micro-to-macro levels
2. Create a map of the relevant phenomenological constructs:
 - a. Explore the examples, metaphors, and analogies used in teaching materials (e.g., textbooks, worksheets, videos).
 - b. Interview teachers

- i. Which kinds of examples, metaphors and analogies do they use in their teaching?
- ii. Which experiments do they conduct?
- iii. What is the familiarity of their students with the phenomenological constructs? Which terms or idioms are they more familiar with?

3. Create a base agent-based model of the target phenomenon. Then divide the model into small sub-procedures that correspond to phenomenological constructs from step 2.
4. Create a small set of the phenomenological blocks and iterate on it based on teachers' and subject-matter experts' feedback.

We argue that phenomenological programming can offer significant advantages beyond easy recognition for novice learners. It can decrease the burden of converting one's thoughts to computer code, thus requiring much less programming instruction. For example, implementing the *zig-zagging* behavior in a procedural programming environment (including the traditional domain-specific environments) would require students to write looping algorithms with variables that preserve the state of the zig-zagging object. Each of these programming constructs are alien to most middle-high school students and even found counterintuitive by most novice college students.

We assume that each phenomenological statement will inevitably embed ambiguities. If we consider students' lack of coherent knowledge to express their intuitive thinking in an explicit, unambiguous manner [5, 6], we argue that a healthy dose of ambiguity in code-blocks can actually be beneficial in accommodating diverse ways of thinking. For example, a *bounce* block designed with phenomenological options such as *like a basketball* and *like a billiard ball* can stimulate rich classroom discussions on how these objects behave and help uncover students' intuitive understandings. Some students may expect the two objects to behave similarly, while others may expect basketballs to bounce back faster. The phenomenological *bounce* block would initially accommodate both students' ways of thinking, but they can later discover the exact nature of each object through tinkering.

Phenomenological programming environments can be customized and localized according for target group students including expected misconceptions they bring to the class, their day-to-day vocabulary, and terms relevant to target subject matter. For example, some students may be familiar with billiard balls, yet for others it may be more suitable to use *pool ball*, *hockey puck*, or another object that behaves similarly. Similarly, if we expect from prior research that students may come to the classroom with misconceptions, we can modify our code-blocks with relevant phenomenological statements. For instance, a *bounce like a football* statement would accommodate misconceptions about objects bouncing back in random directions at each physical collision.

We argue that when designed well, phenomenological programming environments can allow students to start constructing computational models of scientific phenomena with minimal instruction. They can express their intuitive understanding through code and the resulting computational models would be much more sophisticated compared to those that can be realistically achieved with general purpose environments. Each time students run their models they would get a chance to catch the “bugs” in their thinking. Teachers can leverage the incompatibilities between the students’ models and the formal scientific explanations to conduct rich classroom discussions and facilitate more constructivist learning environments.

We believe the best way to explain phenomenological programming is to demonstrate a real-world implementation and provide evidence on how it may impact students’ learning of subject matter. In the next section, we present a phenomenological gas particle sandbox that we designed for teaching the basic assumptions of the kinetic molecular theory. Then, we present preliminary findings from a research implementation with 2 teachers and 121 high-school students.

A DESIGN INTERVENTION: THE PHENOMENOLOGICAL GAS PARTICLE SANDBOX

The idea of phenomenological programming emerged while we were designing hands-on programming activities for the kinetic molecular theory (KMT) within a larger ideal gas laws unit. The KMT defines simple and idealized assumptions about the behavior of gas particles at the macroscopic level. It is universally taught and represents a conceptual understanding of the behaviors of gaseous matter as well as other chemistry topics such as chemical reactions and phase transitions [21,31]. However, prior research documents a myriad of student misconceptions related to the KMT that remain intact after school instruction [18, 21, 22]. We hypothesized that if students could develop their own simple, agent-based models of gas particles, they would be able to connect their intuitive thinking about micro-level gas particle interactions with the gas properties that they observe at the macro-level (e.g., pressure, temperature). After multiple prototypes and feedback sessions with teachers, we realized that it was not plausible to design a domain-specific environment with conventional procedural blocks because modeling particle-to-particle collisions on a 2d plane required solving complex quadratic equations to calculate the momentum exchange. This led us to reconsider the design of our primitives and culminated in a multi-step gas particle sandbox activity with a phenomenological programming component. In this section, we share the design of this phenomenological gas particle sandbox and our findings from the first research implementation of the CC’19 unit with 121 high school general chemistry students.

Design Overview

We developed our phenomenological gas particle sandbox using the NetTango builder authoring environment [11, 40]. NetTango allows the creation of custom, domain-specific

blocks-based programming layers for the construction of NetLogo models. Being a domain-specific programming environment, our sandbox inherits many features from its predecessors such as the Frog Pond [10, 12] and the EvoBuild [37]: a small vocabulary of programming primitives, high-level code-blocks that abstract away formal logic, and a custom design for the target curricular material.

Our phenomenological code blocks are procedural templates, such as *moves* and *bounces*, that can be modified with phenomenologically transparent statements, such as *spinning* or “*straight*” for the former and *like a balloon* or *like a basketball* for the latter. An explanation of each code-block in the phenomenological gas particle sandbox is presented in Table 1. Each phenomenological statement embeds simple assumptions about gas particles. Some of them are compatible with the formal assumptions of the KMT (e.g., moving straight until a collision, bouncing elastically like a billiard ball), while others correspond to potential student misconceptions (e.g., moving randomly, losing momentum) [18, 21, 22, 31]. We hypothesized that students could start modeling gas particles with minimal introduction to programming and without the challenging task of converting their intuitive understanding of the behaviors of bodies in motion into unambiguous, procedural computer-code. The phenomenological blocks would be instantly recognizable. Students could predict and reason about the outcome of their code because it would be easy to mentally simulate gas particles’ behavior.

We embedded our phenomenological gas particle sandbox in a lesson plan with a multi-step, scaffolded activity (Figure 1). We adopted the lesson plan from Levy & Wilensky’s Connected Chemistry 1 unit [19,20,46]. The students were asked to hypothesize about how an air duster canister works. We chose the air duster as the real-world object of study because it was a fixed-volume gas container with nothing but gas molecules inside. Moreover, it was very easy to observe the existence of gas pressure: the air coming out of a full air duster could move small physical objects like dust and small pieces of paper. In the beginning of the lesson, the students hypothesized about how an air duster works by providing textual answers. Then, they used a freehand drawing tool to explain their answers. We designed these two activities to stimulate students’ expression of their intuitive thinking.

The teachers conducted whole-class discussions after the freehand sketching activity. They showed some of the students’ drawings and asked them to share their hypotheses.

In the second step, the students used a static sandbox tool that resembled the freehand sketching activity. They constructed the initial state of a computer model within NetLogo by adding stationary walls, removable walls, green particles, and orange particles (see Figure 2b). We designed the static modeling activity to reduce the complexity of creating a computational model from scratch. It also served as a transition layer from the open-ended sketching activity to computational sketching with limited options.

In the third step, the students used a simplified version of the phenomenological gas particle sandbox to develop a small-scale model of gas particles ($n \leq 4$). We designed the micro-level modeling activity to both reduce the complexity of the programming activity and to encourage the students to attend to the relationship between individual gas particles' behavior and the aggregate-level outcomes.

Finally, the students uploaded their static air duster models from the third step into the gas particle sandbox and tested their blocks-based algorithms with many more particles. We designed this last activity so that students could observe whether their air-duster models behaved as they anticipated.

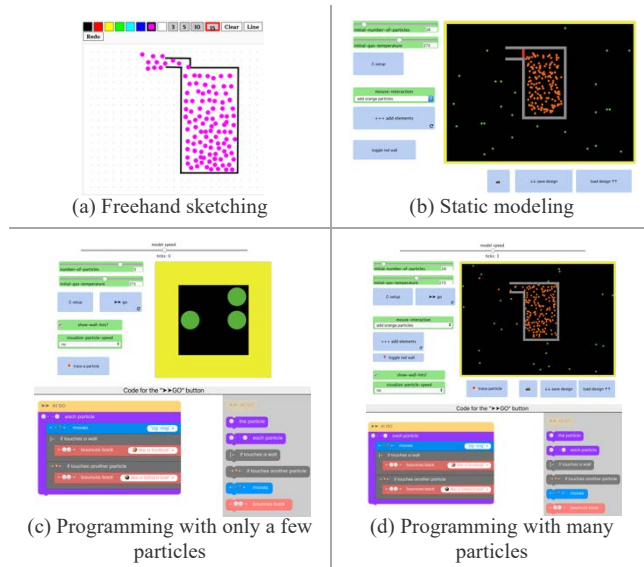


Figure 1. The four stages of the air-duster modeling activity with phenomenological programming.

Overall, the four steps of our lesson plan guided the students through the process of creating a computational model of a real-world phenomenon bottom-up but decreased the complexity of this process by breaking down each step for them and providing necessary scaffolds. At each step, we provided them short video demonstrations on how to use the embedded computational tools such as how to draw particles, how to drag and attach code-blocks to each other, how to run the models, and how to save the models. We also placed reflection questions such as “What were the differences between the freehand sketching activity and the computational sketching activity?” or “Did your air duster model behave as expected? Did you need to change any of your code from the prior activity?”

There were only 7 code-blocks in this iteration of the phenomenological gas particle sandbox (see Table 1). However, combined with the static freehand modeling activities and the phenomenological sub-statements, these blocks were enough to develop sophisticated two-dimensional models of an air duster.

Code block	Explanation
	At GO: <i>Event block.</i> Code that is attached to this block is executed in a continuous loop when the GO button of the model is clicked.
	The particle <n>: <i>Control flow block.</i> The code encapsulated by this block is executed only by the selected particle (e.g., particle 1, particle 10, particle 87) at each simulation step.
	Each particle: <i>Control flow block.</i> The code encapsulated by this block is executed by all particles autonomously and simultaneously at each simulation step.
	If touches a wall: <i>Control flow block.</i> The code encapsulated by this block is only executed when a particle is touching one of the container walls.
	If touches another particle: <i>Control flow block.</i> The code encapsulated by this block is only executed when a particle is touching another particle.
	Moves: <i>Phenomenological block.</i> 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 at each step to follow a path that resembles the Brownian motion.
	Bounces back: <i>Phenomenological block.</i> If a particle is executing this code, it changes its momentum 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 but total kinetic energy is decreased significantly. Like a football: Changes direction randomly. If collides with another particle, exchanges momentum as if it is an elastic collision but total kinetic energy is decreased slightly. 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.

Table 1. The function of the code-blocks of the phenomenological gas particle sandbox and the assumptions embedded in the phenomenological blocks.

Research Implementation

The first research implementation of our lesson plan with the phenomenological gas particle sandbox took place in Spring 2019. Two teachers and a total of 121 high-school regular chemistry students at an ethnically diverse U.S. Midwest public high school participated (Table 2). Neither teacher had prior experience in computer programming and this was their first experience teaching with the phenomenological gas particle sandbox. Most of the students did not have prior experience with programming, either. The intervention lasted a total of 3 class periods over the course of 2 days. The students used Chromebooks to access the lesson over an online portal. Both teachers decided to teach the lesson as a lab. They asked students to form groups and follow the lesson activities autonomously except the whole-class discussions. Each student worked on the activities individually but they were encouraged to share ideas and help each other when necessary. The demographics of the participating students are presented in Table 2 below.

Our research questions in this implementation were:

1. How successful is the phenomenological gas particle sandbox in accommodating the students' intuitive ways of thinking about gas particles?
2. To what extent did the students use the phenomenological code blocks in expressing their intuitive ways of thinking about gas particles?
3. Does engaging with the phenomenological gas particle sandbox help students learn the main assumptions of the kinetic molecular theory?

	n	Male	Female	Non-binary
White	41	27	13	1
African American	34	14	20	-
Latinx	17	9	8	-
Asian	4	3	1	-
Middle Eastern	2	1	1	-
Multiple	23	14	9	-
Total	121	68	52	1

Table 2. Demographics of the study participants (self reported).

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 their blocks-based code. In order to gain further insight on the students' thought processes and the interactions between them, we video recorded four focus groups (2-4 students each). We also sometimes walked around the classroom and asked some non-focus group students to do quick demonstrations of their work. We present a preliminary qualitative analysis of this data through vignettes from our video data and student-generated artifacts from the online portal.

Preliminary Findings

Students Expressed Their Intuitive Thinking through Phenomenological Programming

K & Y are two 10th grade African American female students. From the interactions among them, we learned that K attended an out-of-school blocks-based programming workshop at the fifth grade, but Y did not have any prior programming experience. We selected them as one of our focus group students because they have been in the same lab group since the beginning of the year and they both consented to video data collection.

In the beginning of the lesson, the students were shown a short video about how pressing the valve of an empty air duster had no effect on small pieces of paper, while doing so with a full one moved the pieces. Then, they were asked to explain the difference between a full and an empty air duster can. K answered the question as follows: “*There was no air pressure in the before video, but by pumping air into it, it increased the air pressure allowing the air when released to reach top speed and blow the paper.*” Y answered the question similarly: “*There was no air pressure in the first part but by pumping air into it, it made a high pressure of air come out of it.*”

Neither of the students mentioned gas particles inside the air duster container. On the other hand, both of their freehand sketches were richer than their textual explanations (Figure 2). First of all, each sketch included particle-like representations. K's sketch showed that she envisioned particles coming out of the air duster, but the particles inside were much larger than the particles outside. In addition, there were arrows representing the direction of the particles outside, while there were semi-rings around the particles inside which may represent either vibration or direction. Y's sketch showed large circle-like particles densely packed at the bottom of the air duster but they moved up when the valve was pressed. She drew blue lines to represent the air coming out of the air duster. Overall, both sketches involved ambiguities and multiple contradictory representations of gas particles. This finding is consistent with diSessa's [5, 7] theory that novice understanding of science is a fragmented and incoherent ecology of smaller knowledge elements.

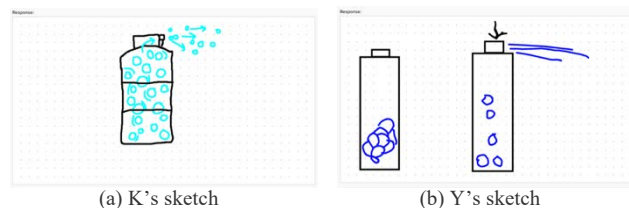
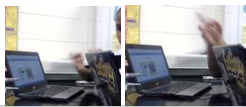






Figure 2. K & Y's freehand sketch models of the air duster can.

The interaction we present in Excerpt 1 below happened during the micro-level phenomenological programming activity (Figure 1c) when Y expressed her frustration with the way her particles behaved.

	Dialogue	Non-verbal indicators
Y	Yeah, but I wanna make them as slow as possible.	<i>Talks to her peers loudly.</i>
K	Why do you wanna make them slower?	<i>Enters the frame, sits next to Y.</i>
Y	Because like, I feel like they are more tight. So, I don't want them to move like psychopaths.	<i>Gestures fast random movement with her hands when saying "like psychopaths" and laughs.</i> 

Excerpt 1. K & Y's initial interaction.

We hypothesize that Y wanted to make her particles move slowly and eventually concentrate at the bottom as she drew in her sketch (Figure 2b). Her sketch also implies that she expected gas particles to only move upwards when the valve is pressed. This might be a misconception due to the similarity between the air duster can and other aerosol cans, which have liquid sitting at the bottom at room temperature. This initial interaction was followed by K's inquiry about Y's code, which provided us with more insight on Y's thinking:

	Dialogue	Non-verbal indicators
K	Are yours in a pattern or is it erratic?	
Y	Erratic. And they're going, but it's bouncing like a balloon.	<i>Scrolls down to her code and points at the blocks. Makes a bouncing gesture with her fists (similar to ).</i> 
K	Oh!	
Y	I feel like they shouldn't be like...	<i>Makes fast random movements with her hands.</i> 
K	...	<i>Laughs.</i>
Y	Why does setup always make it? ... Wait! But I want them to... No! They are not ... Like, yours are fine.	<i>Leans over to her computer. The particles start fast but eventually slow down again.</i>
K	Mine are.	
Y	Mine aren't. Like he should be going over here! Look at it! He's not doing anything!	<i>Points at a stationary particle and then points at the opposite corner of the container, indicating where she expects the particle go.</i> 

Excerpt 2. K & Y's second interaction.

We observe that both K and Y made their particles move "erratically." It is a documented misconception that many students think gas particles change direction at random [18],

while KMT assumes that gas particles move in straight trajectories until they collide with other particles or container walls.

Y told K that her particles "bounce like balloons," which explains why Y's particles initially acted the way she wanted them to, but eventually came to a stop. This contradiction between her understanding of how gas particles should behave and the output of her model frustrated her. Her demonstrations with her hand-gestures and her final remarks showed that Y was continuously trying to mentally simulate the outcome of her model. Unfortunately, this interaction was cut short because their teacher announced the class to wrap up the programming activity.

This is just one case study and therefore not generalizable, but these short excerpts from K & Y's interaction provide preliminary evidence for some of our main design arguments.

K & Y were able to express their intuitive ways of thinking about gas particle behavior using the phenomenological code blocks. There was a direct correspondence between their freehand sketches and their blocks-based code. They made their particles move the way they "felt like" the particles should move (Excerpt 2). They expressed these thoughts by using the embedded phenomenological statements as part of their vocabulary, using bodily movements, and even other analogies such as Y's remark "I don't want them to move like psychopaths" (Excerpt 1).

Their prior knowledge played an important part in their programming choices. Y expected the "bounce like a balloon" block to behave differently than it actually did. She expected her *balloon like* particles to eventually sit at the bottom of the container but keep moving slowly (there is no gravity in the sandbox). When she noticed the contradiction, we were able to observe the role of dynamic mental simulation in her thinking processes once again. She showed K with her fingers which trajectory a particle should have followed.

Finally, we observed some of their misconceptions surfacing during the process. Both K & Y believed that the particles should be moving "erratically." Y believed that the particles should slow down. K's surprised reaction indicated that she found this an unusual idea. She programmed her particles to bounce like billiard balls. Although neither of them mentioned any formal scientific concepts during this interaction, we believe some of their misconceptions might have been informed by their prior exposure to relevant concepts (e.g., random motion, aerosol cans).

This interaction also highlighted some of the limitations of our sandbox design. For example, our blocks library did not include any phenomenological blocks that corresponded to the exact way Y wanted her gas particles to slow down but never stop. We plan to address these limitations in the future iterations of our phenomenological gas particle sandbox design.

Students' Diverse Ways of Thinking Were Reflected in Their Phenomenological Programming Practices

We asked the students to upload a screenshot of their block-based code once they finished the micro-level programming activity (Figure 1c) and explain their programming decisions in writing. Figure 3 below provides a summary of the screenshot data for the three main considerations of the KMT: particle movement, particle-wall collisions, and particle-particle collisions.

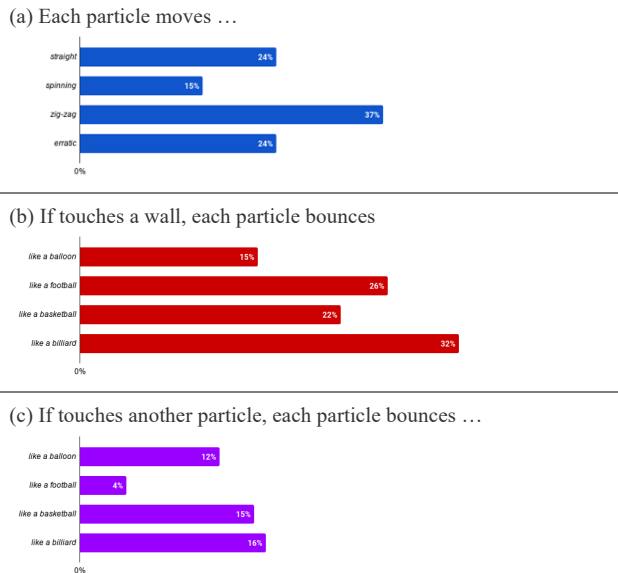


Figure 3. The frequencies of the code blocks used by the students in the first phenomenological programming activity.

We argue that the diversity in the students' algorithms reflects the greater diversity of intuitive student thinking that was documented in prior research [2,5,6,7,13,35]. Unfortunately, we do not have any additional data to further investigate the student reasoning behind some of the more interesting outcomes such as the lack of particle-to-particle collisions in half of the students' blocks or the sharp decrease in the popularity of the "bounce like a football" block in those that included inter-particle collisions in their algorithm.

The diversity in the students' intuitive thinking was also present in their written explanations of their algorithms. We present example student explanations in Figure 4. In all four of them, we observe an underlying pattern: the students did not mention any formal explanations or even their knowledge from prior lessons. They were rather hypothesizing. As in K & Y's interaction, all four students in Figure 4 created their models based on how they *feel*, not on what they learned in prior classes. For example, the student in Figure 4a created an algorithm that is compatible with the basic assumptions of the KMT. However, she simply did so because she based her model "off the pool game." She even mentioned that she did not know the "material of the particles." Similarly, the student in Figure 4b simply wanted her particles to "bounce off in a calm and controlled manner." The students in Figures 3c and 3d expressed

uncertainty in particle movement but they chose the zig-zag movement instead of the erratic option.

	Algorithm	Student explanation
(a)		<i>I used [moves straight] because it was the most realistic. I wanted them to sort of behave like pool balls. Since I based it off the pool game, I wanted them to react like a pool ball although I do not know the material of the particles.</i>
(b)		<i>I used [like a balloon] because I did not want the particles to bounce too hard and too fast. I wanted them to bounce off in a calm controlled manner.</i>
(c)		<i>I used [zig-zag] because the particles move randomly. I used "like a billiard ball" because they kind of bounce around with more speed.</i>
(d)		<i>I used [zig-zag] because there is not one way each particle moves each time. It bounces like a football because it doesn't always have the same directions coming back.</i>

Figure 4. Example student algorithms and explanations

Unfortunately, our data collection at this stage was preliminary. For example, we do not have additional data to understand why the students used the *moves zig-zag* option. Perhaps they wished to implement a rather patterned randomness rather than unpredictable behavior. However, we are cautious about making further claims in this paper about student's engagement with phenomenological programming beyond the apparent diversity reflected in their code blocks and written explanations. We plan to collect more extensive data in the future iterations of our study such as saving students' algorithms after each change and interviewing some of the students after the phenomenological programming activities.

Students Learned the Kinetic Molecular Theory by Engaging with Phenomenological Programming

H is a 10th grade Latinx female student. She is one of the students who agreed to do on-the-spot demonstrations for our research team. In fact, she volunteered to demonstrate her model when she saw the first author recording another demonstration. She had no prior experience with programming. Her initial answer to the empty vs. full air duster can question was "The papers don't move because there was no air in the can. They moved the second time because the experimenter pumped air into the spray can and used it to blow away the paper." Her freehand sketch had large particle-like circles inside the can, while the air outside was presented as coming out on a straight line and then spreading with a cloud-like image (Figure 5).

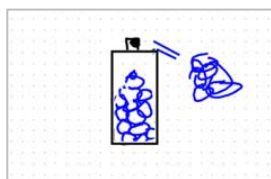


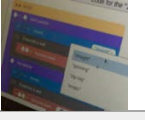
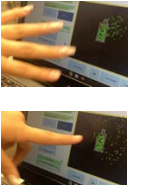


Figure 5. H's freehand sketch model of the air duster.

The interaction in Excerpt 3 happened at the end of the macro-level modeling activity (Figure 1d).

	Dialogue	Non-verbal
R	So, this was your original static model, right?	
H	Yeah. And first I added the red wall.	
H	And then another change I made was change the direction it moved. It was erratic for both and then I changed it to straight.	
R	Why did you change it?	
H	Because I wanted to represent, like, the way the particles move once it comes out the spray can.	
R	They weren't spraying out as you liked?	
H	Oh no. It was just like, spreading out, instead of going towards one direction and then spreading out. And then, (<i>runs the model with the red wall closed</i>) 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.	

Excerpt 3. H walking the researcher (R) through her programming process.

H's experience hints at the full potential of phenomenological programming when everything goes according to design goals. As a student with no prior programming experience, H was able to observe a real-world object, hypothesize about the underlying micro-level particle interactions that lead to the observed macro-level outcome, construct a micro-level computational model of her hypothesis using a blocks-based programming environment, and test her hypothesis at the macro-level. When her initial assumption about gas particles moving erratically did not result in the desired outcome, H was able to debug her code and find the correct solution. Most of what H knew about gas particles prior to this lesson was compatible with the assumptions of KMT. She only had a few simple misconceptions: she expected gas particles to change direction at random. Even if some students may not make as much progress as H, we believe competent teachers can use such opportunities to bootstrap students' prior learning and promote powerful conceptual learning.

Limitations of our preliminary findings

Our preliminary findings are limited in terms of generalizability due to the lack of additional data. We believe

the future implementations can be improved by (1) including a control condition, (2) conducting clinical interviews with students and teachers interviews, and (3) collecting the history of students' programming actions.

CONCLUSION

The two case studies and the block-based student algorithms we presented in this paper are by no means generalizable. More empirical studies are needed to answer the implications of phenomenological programming for science education. In addition, the phenomenological gas particle sandbox we presented is just one such design. We may not be able to determine the full reach of this approach before attempting to create similar environments for other topics.

Nonetheless, our preliminary findings indicate that the phenomenological gas particle sandbox achieved some critical goals of constructionist science education. The students started programming with minimal instruction. The phenomenological code-blocks accommodated their intuitive ways of thinking. They tinkered with the code to test alternative ideas, they exchanged ideas with each other, and the phenomenological statements became a part of their vocabulary. Some of their misconceptions surfaced and we even saw a student overcome a misconception by debugging her code. Most of the resulting NetLogo models were very sophisticated and very similar to the real-world air-duster can. The teachers were comfortable teaching with the phenomenological gas particle sandbox, too.

We argue that paper phenomenological programming is a compelling new direction. For the short term, we are planning to iterate over this design and conduct more design interventions to collect more empirical data on the nature of student learning. For the long term, we hope to design other phenomenological programming environments and explore this approach's implications for computational thinking research.

ACKNOWLEDGMENTS

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 author(s) and do not necessarily reflect the views of the funding organizations.

SELECTION AND PARTICIPATION OF CHILDREN

We recruited two chemistry teachers from a public high school in an ethnically diverse U.S. midwestern city. We provided them with a lesson plan that included embedded phenomenological programming activities. 146 students were taught this lesson. We invited all students to participate in the study. The teachers explained the study before the intervention began and handed out parent consent student assent forms. 121 students agreed to participate with the consent of their parents.

REFERENCES

- [1] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Communications of the ACM*, 60(6), 72-80.
- [2] Michelene Chi. 2005. Commonsense conceptions of emergent processes: Why some misconceptions are robust. *The journal of the learning sciences*, 14(2), 161-199.
- [3] Melanie Cooper, Mike Stieff, and Dane DeSutter. 2017. Sketching the invisible to predict the visible: from drawing to modeling in chemistry. *Topics in cognitive science*, 9(4), 902-920.
- [4] Daniel Damelin, Joseph Krajcik, Cynthia McIntyre, and Tom Bielik. 2017. Students making systems models. *Science Scope*, 40(5), 78-83.
- [5] Andrea diSessa. 1993. Toward an epistemology of physics. *Cognition and instruction*, 10(2-3), 105-225.
- [6] Andrea diSessa and Jim Minstrell. 1998. Cultivating conceptual change with benchmark lessons. In *Thinking practices in mathematics and science learning*, James Greeno and Shelley Goldman (eds.), 155-187.
- [7] Andrea diSessa. 2002. Why “conceptual ecology” is a good idea. *Reconsidering conceptual change: Issues in theory and practice*, 28-60.
- [8] Joshua Epstein. (2008). Why model? *Journal of Artificial Societies and Social Simulation*, 11(4), 12.
- [9] Shuchi Grover and Roy Pea. 2013. Computational thinking in K-12: A review of the state of the field. *Educational Researcher*, 42(1), 38-43.
- [10] Yu Guo, Aditi Wagh, Corey Brady, Sharona Levy, Michael Horn, and Uri Wilensky 2016. Frogs to Think with: Improving Students' Computational Thinking and Understanding of Evolution in A Code-First Learning Environment. *Proceedings of the 15th International Conference on Interaction Design and Children*.
- [11] Michael Horn, and Uri Wilensky. 2012. NetTango: A mash-up of NetLogo and Tern. Paper presented at the *Annual Meeting of the American Educational Research Association*.
- [12] Michael Horn, Corey Brady, Arthur Hjorth, Aditi Wagh, and Uri Wilensky. 2014. Frog pond: a codefirst learning environment on evolution and natural selection. *Proceedings of the 2014 conference on Interaction design and children*
- [13] Cindy Hmelo-Silver, Surabhi Marathe, and Lei Liu. 2007. Fish swim, rocks sit, and lungs breathe: Expert-novice understanding of complex systems. *The Journal of the Learning Sciences*, 16(3), 307-331.
- [14] Philip Johnson-Laird. 2005. Mental models and thought. *The Cambridge handbook of thinking and reasoning*, 85-208.
- [15] Philip Johnson-Laird. 2010. Mental models and human reasoning. *Proceedings of the National Academy of Sciences*, 107(43), 18243-18250.
- [16] Ken Kahn. 1996. ToonTalk™—an animated programming environment for children. *Journal of Visual Languages & Computing*, 7(2), 197-217.
- [17] Ken Kahn. 2007, July. Building computer models from small pieces. In *Proceedings of the 2007 Summer Computer Simulation Conference*.
- [18] Vanessa Kind. 2004. Beyond appearances: Students' misconceptions about basic chemical ideas. Royal Society of Chemistry.
- [19] Sharona Levy and Uri Wilensky. 2009. Crossing levels and representations: The Connected Chemistry (CC1) curriculum. *Journal of Science Education and Technology*, 18(3), 224-242.
- [20] Sharona Levy and Uri Wilensky. 2009. Students' learning with the Connected Chemistry (CC1) curriculum: navigating the complexities of the particulate world. *Journal of Science Education and Technology*, 18(3), 243-254.
- [21] Huann-shyang Lin, Hsiu-ju Cheng, and Frances Lawrenz. 2000. The assessment of students and teachers' understanding of gas laws. *Journal of Chemical Education*, 77(2), p.235.
- [22] Carlos Mas, Juan Perez, and Harold Harris. 1987. Parallels between adolescents' conception of gases and the history of chemistry. *Journal of Chemical Education*, 64(7), 616-618.
- [23] Marvin Minsky. 1988. Society of mind. Simon & Schuster Paperbacks.
- [24] Mary Nakhleh, 1992. Why some students don't learn chemistry: Chemical misconceptions. *Journal of chemical education*, 69(3), 191-196.
- [25] National Research Council. 2012. *A framework for K-12 science education: Practices, crosscutting concepts, and core ideas*. National Academies Press.
- [26] NGSS Lead States. 2013. *Next Generation Science Standards: For States, By States*. Washington, DC: The National Academies Press.
- [27] Oscar Nierstrasz, Laurent Dami, Vicki de Mey, Marc Stadelmann, Dennis Tsichritzis, and Jan Vitek. 1990. Visual Scripting: Towards interactive construction of object-oriented applications. *Object Management*, 315-331.
- [28] Seymour Papert. 1972. Teaching children thinking. *Programmed Learning and Educational Technology*, 9(5), 245-255.

- [29] Seymour Papert. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic books.
- [30] Seymour Papert. 1991. Situating constructionism. In *Constructionism*, Seymour Papert and Idit Harel (eds.). Ablex Publishing.
- [31] S R Pathare, and H C Pradhan. 2010. Students' misconceptions about heat transfer mechanisms and elementary kinetic theory. *Physics Education*, 45(6), 629-634.
- [32] Jean Piaget. 1972. Development and learning. *Readings on the development of children*, 25-33.
- [33] Christina Schwarz, Brian Reiser, Elizabeth Davis, Lisa Kenyon, Andres Achér, David Fortus, Yael Schwartz, Barbara Hug, and Joe Krajcik. 2009. Developing a learning progression for scientific modeling: Making scientific modeling accessible and meaningful for learners. *Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching*, 46(6), 632-654.
- [34] Bruce Sherin. 2006. Common sense clarified: The role of intuitive knowledge in physics problem solving. *Journal of Research in Science Teaching*, 43(6), 535-555.
- [35] Stella Vosniadou, and William Brewer. 1992. Mental models of the earth: A study of conceptual change in childhood. *Cognitive psychology*, 24(4), 535-585.
- [36] Aditi Wagh and Uri Wilensky. 2012. Evolution in blocks: Building models of evolution using blocks. In *Proceedings of the Constructionism Conference*.
- [37] Aditi Wagh and Uri Wilensky. 2018. EvoBuild: A Quickstart Toolkit for Programming Agent-Based Models of Evolutionary Processes. *Journal of Science Education and Technology*, 27(2), 131-146.
- [38] David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. 2016. Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, 25(1), 127-147.
- [39] Uri Wilensky. 1999. NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.
- [40] Uri Wilensky and Michael Horn. 2011. NetTango. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.
- [41] Uri Wilensky and Mitchell Resnick. 1999. Thinking in levels: A dynamic systems approach to making sense of the world. *Journal of Science Education and technology*, 8(1), 3-19.
- [42] Uri Wilensky. 2001. Modeling nature's emergent patterns with multi-agent languages. In *Proceedings of EuroLogo* (1-6).
- [43] Uri Wilensky and Kenneth Reisman. 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.
- [44] Michelle Wilkerson-Jerde and Uri Wilensky. 2010. Restructuring change, interpreting changes: The DeltaTick modeling and analysis toolkit. *Proceedings of the Constructionism Conference*.
- [45] Janette Wing, 2006. Computational thinking. *Communications of the ACM*, 49(3), 33-35.
- [46] Uri Wilensky. 1999. 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.