Proxy-Based Programming: Making Programming More Accessible through Virtual, Physical, and Social Embodiment

ALEXANDER REPENNING, School of Education, FHNW Switzerland, Brugg-Windisch, Switzerland

From a theoretical perspective, this article examines the role of embodiment in K-12 computer science education through three distinct perspectives: virtual embodiment for cognitive understanding, physical embodiment for emotional engagement, and social embodiment for collaborative learning. The research introduces proxy-based programming as a novel programming paradigm that helps novice programmers overcome pragmatic programming challenges by providing a visual proxy that serves as an embodiment of the object being programmed. This proxy features dual temporal representation that simultaneously shows both the present and future situations resulting from programming actions. Unlike programming approaches that primarily address syntactic challenges, proxy-based programming also mitigates semantic and pragmatic aspects of computational thinking. RULER.game, a Collaborative Computational Thinking Tool, implements four core principles of proxy-based programming, enabling safe programming experimentation while proactively preventing errors. Initial studies comparing proxy-based programming with block-based programming show significant reductions in error rates. The article explores how embodiment plays a role in making programming more accessible from both theoretical and practical perspectives. Additionally, the article explores the motivational benefits of physical embodiment, where students create games by first drawing objects on paper before importing them digitally, as well as social embodiment through awareness interfaces that make collaborators' actions and intentions visible in real-time. This comprehensive approach to embodiment presents a novel framework for making programming more accessible and engaging for K-12 students.

CCS Concepts: • **Applied computing → Interactive learning environments**;

Additional Key Words and Phrases: Computer science education, computational thinking, proxy-based programming, block-based programming, live programming, programming by example, embodiment in computing, creativity, scaffolding, preservice teacher education, K-12 primary education

ACM Reference format:
Alexander Repenning. 2026. Proxy-Based Programming: Making Programming More Accessible through Virtual, Physical, and Social Embodiment. *ACM Trans. Comput. Educ.* 26, 3, Article 45 (April 2026), 32 pages.
https://doi.org/10.1145/3786759

1 Introduction

Research in educational K-12 (Kindergarten to grade 12) programming has highlighted a critical need to shift focus from programming approaches such as block-based programming that primarily address syntactic challenges of accessibility toward approaches emphasizing the need to push

This research was supported by the Hasler Foundation under Grant No. 23084.
Author's Contact Information: Alexander Repenning (corresponding author), School of Education, FHNW Switzerland, Brugg-Windisch, Switzerland; e-mail: alexander.repenning@fhnw.ch.

ACM 1946-6226/2026/4-ART45
https://doi.org/10.1145/3786759

beyond syntax [73]. In the past, block-based programming languages (e.g., [3]) have made strong claims regarding their affordances, including that they improve learnability, reduce cognitive load, prevent errors, and enhance understanding of program structure. While these benefits may be true and perhaps even necessary for novice programming, they are by no means sufficient. Early on, Knuth in his seminal analysis of the root problems of programming bugs uncovered nine problem-independent categories [39]. Only one of these categories is connected to syntax, suggesting that the majority of programming challenges lie elsewhere. This raises fundamental questions: What is the nature of these remaining programming challenges, how difficult are they compared to syntactic challenges, and how can a new generation of tools help novice programmers overcome them?

These non-syntactic challenges appear to be fundamentally *pragmatic* in nature. Recent research exploring the types of programming errors with block-based programming has found that students still struggle with simple, Hour of Code-like programming challenges. Ben-Yaacov et al. categorize these as logic errors—problems that arise from misunderstanding what code means in specific situational contexts. For instance, kids programming with Lightbot [26] solve programming puzzles where a robot is controlled through actions such as moving forward and turning left or right. Turning at the wrong moment or in the wrong direction constitutes a logic error. These kinds of errors are not related to syntax and consequently are not mitigated by the claimed affordances of block-based programming mentioned above.

Drawing from semiotics [49], a subfield of linguistics, challenge levels for dealing with natural languages have been ranked in ascending order of difficulty: syntax being relatively low difficulty, semantics at a moderate level, and pragmatics being the highest challenge. Could it be that computer science education, as a field exploring artificial languages, has focused too much on solving the relatively simple problem of syntax while largely ignoring the highest challenge of pragmatics? While syntax and semantics are well understood in computer science, pragmatics—with its roots in linguistics—remains a somewhat underexplored concept in computer science education.

Drawing an analogy from linguistics, where pragmatics is defined as "the understanding of what words mean in specific situations" [Merriam-Webster], we propose a parallel concept in computer science:

"*Pragmatics in programming is the understanding of what code means in specific situations.*"

This definition centers on two fundamental concepts outlined in Figure 1: *code* and *situation*. Code refers to any representation of a program's instructions—whether expressed through visual block-based languages like Scratch, or textual programming languages like Java. Code describes the behavioral logic that governs how objects should act.

The concept of situation is more nuanced and consists of two interconnected components: *world* and *point of view*. The world encompasses what computer science education literature sometimes calls a microworld [1, 2, 56, 58]—a complete description of the state and spatial relationships of all objects within a computational environment.

The point of view represents a user's deliberate selection of a specific object within the world, effectively choosing whose "point of view" to embody when understanding or programming behavior. This selection transcends mere user interface interaction—it constitutes a fundamental act of syntonic embodiment [85], where selecting an object means mentally "becoming" that object. The notion of body syntonicity, originally introduced by Papert [57] as turtle graphics in the Logo programming language, refers to the idea of learning and understanding concepts through relating them to one's own body and physical experience. Connecting abstract ideas to bodily sensations, movements, and spatial awareness makes them more concrete and accessible. Body syntonicity offers multiple benefits in programming education, making concepts more tangible and easier to

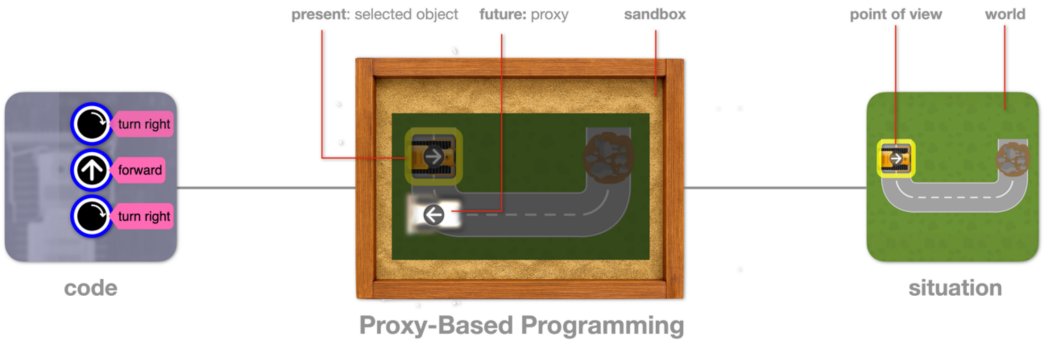

Fig. 1. PBP helps users to understand the meaning of code in specific situations through dual temporal representations simultaneously showing the present situation, the future situation, embodied by a proxy, and the code causing this transformation.

comprehend while fostering spatial reasoning skills, improving concept retention, and boosting learner engagement and motivation.

We have created **proxy-based programming** (**PBP**) as a new programming paradigm helping novice programmers to overcome *pragmatic* programming challenges by providing affordances based on syntonic embodiment. PBP introduces the concept of a visual proxy—a recognizable copy of the object being programmed that serves as an embodied representation simultaneously showing the *present* situation as well as the *future* situation (Figure 1) resulting from proposed code changes. This *dual temporal representation* allows users to mentally inhabit the proxy, i.e., to embody it, and immediately perceive how their programming decisions will affect the object's behavior within its specific situational context. The ideas behind PBP are captured by four named principles—proxy, sandbox, prebugging, and demonstrational palettes—explained throughout the article.

With RULER.game we have created a **Collaborative Computational Thinking Tool (C^2T^2)** implementing PBP that enables users from kindergarten children to advanced students to create games across smartphones, tablets, laptops, and VR devices. The platform supports creative outputs ranging from simple movement-based games that pre-reading children can construct, to multi-agent games featuring AI behaviors like ghost pathfinding in Pac-Man [60]. By C^2T^2 we refer to supporting Computational Thinking processes [70] through virtual, physical, and social embodiment with real-time online collaboration. The tool serves users across developmental stages: kindergarten children can program through visual interfaces with spoken syntonic explanations [85] that provide audio guidance for non-readers, while advanced students can create complex game mechanics involving strategic decision-making and adaptive behaviors.

This article is theoretical in nature, exploring the role of virtual, physical, and social embodiment in programming approaches that address pragmatic challenges. Specifically, this work should be considered a design paper that explains *how* principles of embodiment result in the observed affordances of PBP. Rather than presenting new empirical investigations, this work synthesizes previously documented evidence of efficacy while focusing on the theoretical explanation of how embodied programming approaches function. Using a theoretical lens grounded in educational philosophy and cognitive science, this article examines how understanding the connections between embodiment and pragmatics can make programming more accessible to novice learners. While exploring the theoretical foundations and potential benefits of embodied programming approaches, this article also examines potential limitations and unintended consequences of PBP, including concerns about scaffolding dependency and the challenge of transitioning learners to traditional programming environments.

After outlining related work and introducing the core principles of PBP, this article presents three types of embodiment to explore three different how questions: (1) Virtual embodiment: how do users mentally inhabit digital proxies to mitigate cognitive challenges in understanding abstract programming concepts? (2) Physical embodiment: how do tangible creation processes enhance emotional engagement and motivation in programming activities? (3) Social embodiment: how do collaborative interfaces that visualize partners' actions support distributed learning and peer collaboration in computational thinking?

This article addresses the three research aims not through new empirical investigations, but through theoretical demonstration of how PBP operationalizes embodiment principles to make programming more accessible. The first research aim, how virtual embodiment mitigates cognitive challenges, is addressed in Section 3 by explaining the design principles underlying PBP (proxy, sandbox, prebugging, and demonstrational palettes) and demonstrating through the RULER.game C^2T^2 how these principles work together to reduce programming errors and support pragmatic understanding. The second research aim, concerning physical embodiment and emotional engagement, is addressed in Section 4 by drawing on previously documented evidence from informal learning settings to demonstrate how the paper to digital workflow creates meaningful connections between tangible creation and computational thinking. The third research aim, how social embodiment supports collaborative learning, is addressed in Section 5 by explaining how awareness interfaces in C^2T^2s make thinking visible and support distributed cognition, though this dimension remains the least empirically investigated. Rather than providing traditional empirical evidence with formal hypotheses and controlled experiments, this theoretical contribution demonstrates how embodied programming approaches function by synthesizing existing evidence, analyzing design principles, and examining the mechanisms through which virtual, physical, and social embodiment address different barriers to programming accessibility.

2 Related Work

While the related work outlines here represents only the most directly relevant research to PBP, throughout the remainder of this article we draw upon numerous additional references to systematically craft the argument for how PBP fundamentally connects to embodiment in computational thinking education.

Educational Philosophy still relevant to contemporary computer science education including embodiment can be traced back several centuries. In pedagogy and educational philosophy Pestalozzi developed the influential "head, heart, and hands" educational philosophy [43, 59] in the early 19th century, which emphasized holistic learning that integrates cognitive (head), affective (heart), and psychomotor/tactile (hands) domains of learning. Pestalozzi's framework presents head, heart, and hands as distinct domains, while contemporary embodied cognition theory views brain, body, and environment as an inseparable integrated system. We employ Pestalozzi's tripartite structure as a practical organizing framework for discussing pedagogical approaches, not to suggest these aspects of learning are functionally separate. Building on this progressive educational tradition, Dewey explored the notion of "instrumental motivation" [14] explaining connections between efforts and meaningful outcomes which Papert later called "hard fun" [54] to refer to the large potential to learn from employing programming to solve hard problems. More famously, Papert coined the term constructionism [35, 55] to refer to the idea of learning from creating meaningful physical, or virtual, artifacts.

Syntax, semantics and pragmatics are the three, increasingly complex branches of semiotics, a subfield of linguistics, originally formulated by Morris [49] that remain highly relevant for computer science and programming languages. Computer science education concerned with novice programmers has largely focused on syntactic challenges which it tried to mitigate with visual

programming languages [74] specifically block-based programming languages preventing syntax errors [50], envisioned early as drag and droppable blocks by Blox-Pascal [25] operationalized by AgentSheets [64] and popularized by Scratch [69]. However, addressing syntactic errors is not sufficient. Ben-Yaacov and Hershkovitz [5] found that 39% of student program executions resulted in errors related to issues like counting and orientation misconceptions rather than syntax. These kinds of errors could be considered of a logic or, in Morris' semiotic branches framework, pragmatic in nature. Pragmatic programming support—approaches that support the "understanding of what code means in specific situations"—can be found in foundational work exploring programming by example [10, 42], live programming [31, 46, 47] and combinations such as live programming by example [20].

Embodiment is a powerful approach to move beyond syntactic support. Body syntonicity [85], a form of embodiment introduced early by Papert in his work on Logo [57], refers to learning through relating programming concepts to one's own body and physical experiences. The turtle in Logo was designed to be "body syntonic"—children could imagine being the turtle and physically act out its movements, making abstract programming concepts more concrete and accessible through embodied understanding. This embodied approach to understanding causality builds on foundational work by experimental psychologist Michotte [48], who explored what he called the "perception of causality" through important experiments explaining how people would be able to "see" phenomena such as object collisions based on their embodied experiences in the world. This embodiment has influenced many subsequent educational programming tools. For example, research has shown that when children can connect computational concepts to physical movements and spatial reasoning, it enhances both their engagement and comprehension. However, while block-based programming environments inherited some of Logo's accessibility goals, they haven't fully capitalized on the benefits of body syntonicity. This represents a missed opportunity, as embodied understanding could help address not just syntactic but also semantic and pragmatic programming challenges.

Debugging is the process of eliminating errors that have been found after testing a program. Debugging, a skill different from coding, needs to be explicitly taught [45]. In addition to be a practice, debugging should be supported by debuggers, which are tools that are often provided in interfaces different from the programming interface [45]. The debugging process is particularly challenging in many block-based programming languages as many provide limited debugging support [30] and may even intentionally suppress error messages [13]. Additionally, block-based programming languages have been criticized for potentially leading to code smells and inefficient programming patterns, as demonstrated in studies of Scratch programs [28]. The perceived lack of a debugger in Scratch, for instance, has resulted in a number of third-party debugging tools such as NuzzleBug [13], Blink [76], and TurboWarp [19]. Deiner, the author of NuzzleBug, even goes even further by claiming the lack of debuggers is ubiquitous to block-based programming by suggesting that "the general lack of debugging support in block-based environments is concerning."

Prebugging is a proactive bug prevention approach. In contrast to debugging, prebugging [78] tries to proactively prevent bugs before a complete program has been written. A key insight comes from Eisenstadt who observed that *debugging becomes particularly challenging when there is a significant temporal and spatial gap between a problem's root cause and its visible symptoms* [18]. In the remainder of this article, this phenomenon will be called the *Eisenstadt gap*. The jigsaw-puzzle shapes found in some block-based programming languages could be considered *syntactic prebugging* as they largely prevent the construction of syntactically incorrect programs. Unfortunately, prebugging at the pragmatic level [49] is considerably more complex as the only way to determine the effect of changing code is to execute it and immediately show its consequences [41]. Live programming [46, 47] could be considered a form of *pragmatic prebugging*. Using the

right visualization and narrowing the Eisenstadt gap below the threshold of Michotte's perception of causality could result in powerful embodiment helping users understand what code means in specific situations. In other words, this could help with prebugging. Both live programming and programming by example can powerfully reduce the Eisenstadt gap. Unfortunately, for many game-like use cases live programming may be too proactive, resulting in unwanted side effects. Imagine dealing with new edge cases in programming Pac-Man while multiple ghosts are actively attacking you. Previous work such as small-step live programming [20] has shown that providing users with mechanisms to focus their attention on specific parts of the program can mitigate some of these challenges. Similarly, programming by example, where the user changes the situation and the computer updates the code, struggles when small changes to a situation result in large-scale code modifications.

Having established the theoretical foundations and related work in embodiment and programming education, the following sections examine how PBP employs three distinct forms of embodiment to address different aspects of programming accessibility. Each section begins with a research aim that articulates *how* virtual embodiment (Section 3), physical embodiment (Section 4), and social embodiment (Section 5) contribute to making programming more accessible through different mechanisms of syntonic engagement. These sections demonstrate how PBP operationalizes embodiment theory to address cognitive, affective, and collaborative challenges in novice programming education.

3 Virtual Embodiment: Prebugging with PBP

Research Aim #1: How does virtual embodiment mitigate cognitive programming challenges? How does virtual embodiment work through PBP to significantly lower barriers for novice programmers by reducing error potential and addressing pragmatic programming challenges? Virtual embodiment should narrow the Eisenstadt gap [18] by creating a bidirectional connection between code and situation, enabling students to understand what their programming instructions mean within specific situational contexts—the essence of programming pragmatics.

PBP narrows the temporal gap through proactive feedback that immediately shows the consequences of code changes, eliminating the delay between programming actions and their visible effects. A proxy, contained in a sandbox, is used to connect code and situation by simultaneously embodying *present* and *future* situations (Figure 1). Additionally, it narrows the spatial gap by providing a single user interface that displays both the embodied code view and the situational context together on the same screen (Figure 4).

Consider Wayne, a 7-year-old primary school student participating in a computational thinking challenge [17], programming a digger to follow a road to a dirt pile. For this challenge, the code describes a sequence consisting of forward(), turnLeft(), and turnRight() instructions. The situation encompasses the combination of the *world* and the user's chosen *point of view*. The world, showing the digger, road segments, and the dirt pile, includes the complete state of the game environment with all object properties and spatial relationships. The point of view represents the user-selected object (the digger) and its orientation, with the digger heading to the right.

When Wayne begins programming the digger, the system opens a sandbox and creates a proxy by copying the selected object. This proxy serves as an embodiment and appears as a ghosted version of its original. Wayne can now select instructions to be added to the code and executed by the proxy. The resulting dual temporal representation in the sandbox simultaneously embodies the present situation and future situation of the digger. Wayne has correctly selected the turnRight() and forward() instructions and is about to add the third instruction. He makes a mistake by selecting the turnRight() instruction. Figure 1 shows both situations as well as the code transforming the

present into the future. In the present, the digger is facing right at the beginning of the road. In the future the digger is facing left, on top of the turn.

Wayne instantly recognizes his error—the digger is facing in the wrong direction. He realizes he should have used the turnLeft() instruction instead of turnRight(). Wayne quickly fixes his mistake and continues, finishing the rest of the programming challenge without further errors. After Wayne finishes, the sandbox is closed and all unwanted side effects are removed.

The next section outlines four design principles that operationalize the virtual embodiment approach in PBP environments and differentiate PBP from other approaches.

3.1 Principles of PBP

PBP is defined by four design principles—*proxy, sandbox, prebugging*, and *demonstrational palettes*—that will also serve as analytical frameworks in subsequent sections to contrast PBP with other embodiment approaches.

(1) *Proxy: PBP helps users learn programming by providing a dual temporal representation, called the proxy, simultaneously visualizing the present and future situations as well as the code causing this transformation to make abstract code concepts concrete through syntonic embodiment*. This dual temporal representation is fundamentally different from traditional programming environments such as Logo or Scratch, where you either see the current state or the result of execution, but never both simultaneously in an embodied way. An annotated copy of that object, called the proxy, is like a crystal ball that shows you the immediate future of your programming decisions while keeping you grounded in the present context. In RULER.game, the annotation of the proxy consists of an arrow indicating the current heading of the proxy to help users recognize orientation (Figure 1). To enhance syntonic potential, the proxy is highlighted through its ghost-like appearance while the world is dimmed by rendering it darker, ensuring users can more clearly identify with the proxy and avoid confusion about which object is being programmed. By syntonic potential we mean an object's capacity to enable users to embody it by relating programming concepts to their own bodily experience, thereby facilitating pragmatic understanding of what code means in specific situational contexts. Also important for syntonic potential is to make the selection of the original object visible to users. After all, there could be multiple identically looking objects making it confusing for users to match present and future without the explicit selection.

(2) *Sandbox: PBP allows users to safely experiment with code in a sandbox without having to deal with side effects.* A sandbox is an isolated testing environment that replicates production settings while creating a safe boundary that prevents experimental code or untested changes from affecting live systems. This combination of isolation, safety, and replication enables users to freely experiment and test without consequences beyond the contained environment. In RULER.game, when a user edits code, the system automatically creates a sandbox containing both the proxy and the world. The proxy replicates the object being programmed with all its settings. With read access to the situation, the sandboxed proxy can interpret and annotate both the code and the situation. For example, when editing the digger in Figure 6 (middle), the code is annotated to show that the seeToMyRight ("floor") condition is false while the seeAhead("floor") condition is true. Additionally, the "else if" clause is annotated as ready to fire because all its conditions (there is only one) are true. The safe boundary is established through automatic backup to prevent unwanted side effects. This places the proxy into managed isolation, controlling read and write access to the world. Because of this isolation, all changes made by the proxy during experimentation (such as moving in the world or creating new objects) can be automatically undone when the user finishes editing. The proxy

then fades away, the sandbox closes, and the world is restored, ensuring safe exploration of programming possibilities.

(3) *Prebugging: PBP enables users to immediately perceive causality of adding new code through proactive execution of new code, to prevent the need for debugging, without requiring users to run the entire project.* While debugging is about witnessing and fixing a problem *after* running a program, *prebugging* [78] is about narrowing the Eisenstadt gap by bringing the problem to the user's attention *before* the users is running the program. The ideal time is the very moment when the user changed the program. If the gap can be closed to a certain narrow window of time then, according to Michotte [48], the causality is actually perceived by the user. Proactive approaches, unlike reactive ones, always provide visualization without a user asking for them. Systems implementing prebugging need to carefully balance the potential intrusiveness of proactive interfaces with the potential to significantly widen the temporal Eisenstadt gap of reactive interfaces. The difference between proactive and reactive is essential to PBP. While, of course, users could manually invoke reactive debugging mechanisms to show the consequences of executing code, they typically do not. Part of the problem could be that some programming environments actually make it quite difficult to test isolated fragments of code or that testing results in unwanted side effects. PBP inherits many of the benefits of live programming [47] while avoiding some of its larger problems, such as the need to run an entire project for the sole purpose of getting immediate feedback on code changes. Especially with projects featuring a number of interacting objects, running the entire project may introduce unwanted side effects complicating the debugging process. Ultimately, prebugging is not about adding explicit debugging tools such as steppers and breakpoints to programming environments, but about employing the notion of embodiment to dramatically reduce the need for such tools to make programming more accessible to novices.

(4) *Demonstrational Palettes: PBP offers users serendipitous code through demonstrational code palettes by having the proxy interpret the situation.* The proxy, leveraging aspects of programming by example [10, 21, 40, 42, 51], interprets the current situation to intelligently configure and adapt programming instructions, in what we call *demonstrational palettes*, to make them pragmatically relevant and useful. For instance, rather than simply listing generic conditions, demonstrational palettes proactively interpret the situation and reconfigure the available conditions and actions to maximize their immediate value. For instance, demonstrational palettes will proactively adapt a condition like seeAhead(object) to default to the actual object currently found ahead of the proxy, e.g., seeAhead("wall"). This creates serendipitous moments where the system offers pragmatically adapted code options that programmers might not have initially considered, making the most relevant programming choices naturally emerge from the situation itself. By reducing the cognitive overhead of selecting appropriate code constructs for specific situations, demonstrational palettes enhance programming flow and enable the rapid iterative cycles that characterize *progressive PBP* (Section 3.4).

The next section introduces RULER.game, a concrete implementation that brings these concepts to life through practical examples. By examining how students use proxies to program game objects like ghosts chasing Pac-Man, we can better understand how virtual embodiment helps bridge the gap between code and situation through this dual temporal representation.

3.2 RULER.game

RULER.game [60, 65] is a C^2T^2that implements PBP for novice programmers creating games consistent with Scalable Game Design [67]. Designed to introduce young children to game design

and programming concepts such as sequences, conditional statements, loops, and parallelism, RULER.game makes computational thinking more accessible by supporting prebugging approaches that extend beyond traditional coding to encompass pragmatic understanding of how programming concepts apply within specific problem-solving contexts. Much like how LEGO Duplo relates to regular LEGO, RULER.game provides a highly accessible entry point that allows gradual progression to more sophisticated platforms like AgentCubes [34, 66]. Unlike other beginner programming environments such as ScratchJr, RULER.game enables even pre-reading children to create surprisingly complex games, such as fully functional Pac-Man with intelligent ghost agents capable of tracking and collaborating, by incorporating AI-powered actions [60] while maintaining a low entry threshold. These sophisticated games leverage computational thinking concepts including variables, conditional logic, multi-agent coordination, collision detection, pathfinding algorithms, and event-driven programming, demonstrating that RULER.game addresses far more than basic sequencing despite our use of simple examples for pedagogical clarity throughout this article. Through PBP's ability to bridge code and situation, RULER.game transforms early game design into learning experiences that can grow with developing skills.

RULER.game is implemented as a Progressive Web App [77], providing flexibility across devices. It runs in modern browsers on laptops, tablets, and smartphones, adapting its interface to different screen sizes. The tool also works in VR environments like the Apple Vision Pro, creating immersive programming experiences. RULER.game can even function, albeit with significant limitations, on ultra-small devices such as the Apple Watch, making computational thinking accessible across a wide range of computing platforms.

3.3 PBP in Action

Hypothetical scenarios are used to illustrate the affordances of RULER.game. Imagine Yumna, an 8-year-old primary school student, working on Kodetu programming challenges [5, 17] where a character must be guided through increasingly complex puzzles. The RULER.game interface presents a game world (Figure 2, top) and a selection of projects including tutorials (Figure 2, bottom). Through a tutorial, Yumna learns she needs to program the digger by defining an IF/THEN rule. For the THEN part, she will use basic movement actions: forward(), turnLeft(), and turnRight() (Figure 3, right). These actions are inspired by the turtle commands [57] found in the Logo programming language [29, 36], and are similar to those found in remote control cars or educational robots like the Bee-Bot [72]. When Yumna taps the digger, its current behavior is displayed as an empty IF/THEN rule (Figure 3, left).

When Yumna taps the THEN box to specify the digger's actions, she automatically enters the PBP sandbox. The proxy helps Yumna understand the digger's point of view through body syntonicity [85]—she can imagine being the digger to determine which way to turn. For instance, the forward() action will make the digger move in the direction indicated by the arrow annotation. Yumna experiments by tapping turnRight() followed by forward() to move the digger toward the corner. To help understand each action, she can enable spoken explanations and see explanatory tags that appear during the proxy's movements (Figure 4, left). Most importantly, Yumna can witness the dual temporal representation in Figure 4 (left): the selected digger in the background shows where it is now (present situation), while the ghosted digger in the foreground shows where it will be after executing the code (future situation). The annotated turnRight() and forward() instructions visually demonstrate how these commands would transform the present state into the future state. Because the proxy immediately shows both the code being added and the digger's new position and orientation, Yumna can directly perceive the cause-and-effect relationship [48]—she sees exactly how her programming choices will change the digger's behavior before committing to them.

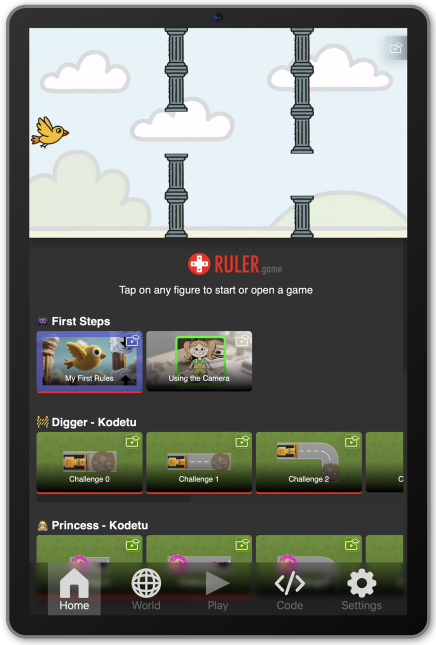

Fig. 2. The RULER.game mobile user interface home screen: game at top, projects and tutorials at the bottom.

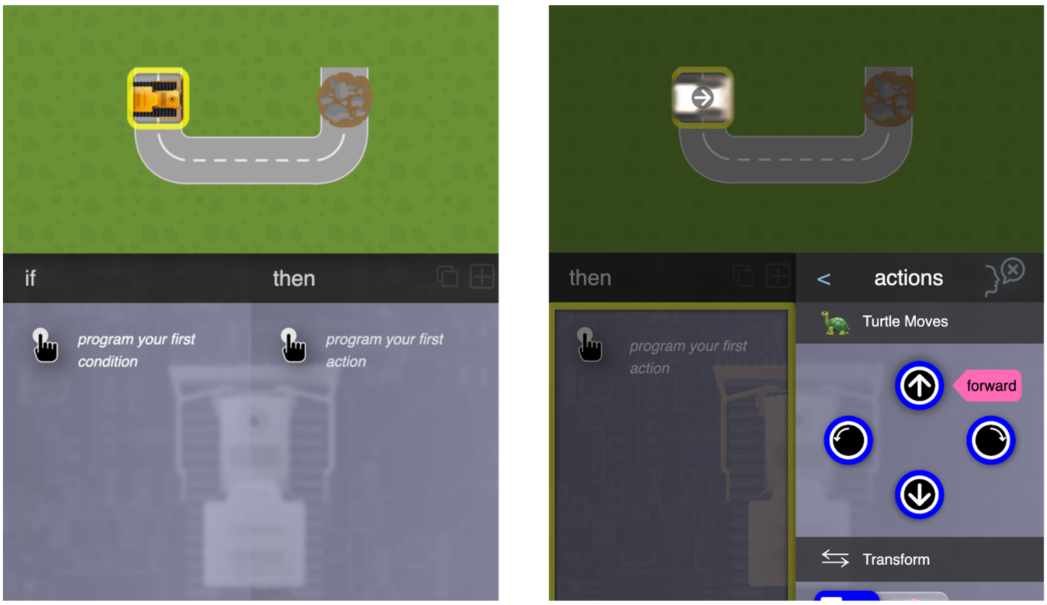

Fig. 3. Digger is not programmed (left). Tap "then" part to enter sandbox, create proxy and show demonstrational palette (right).

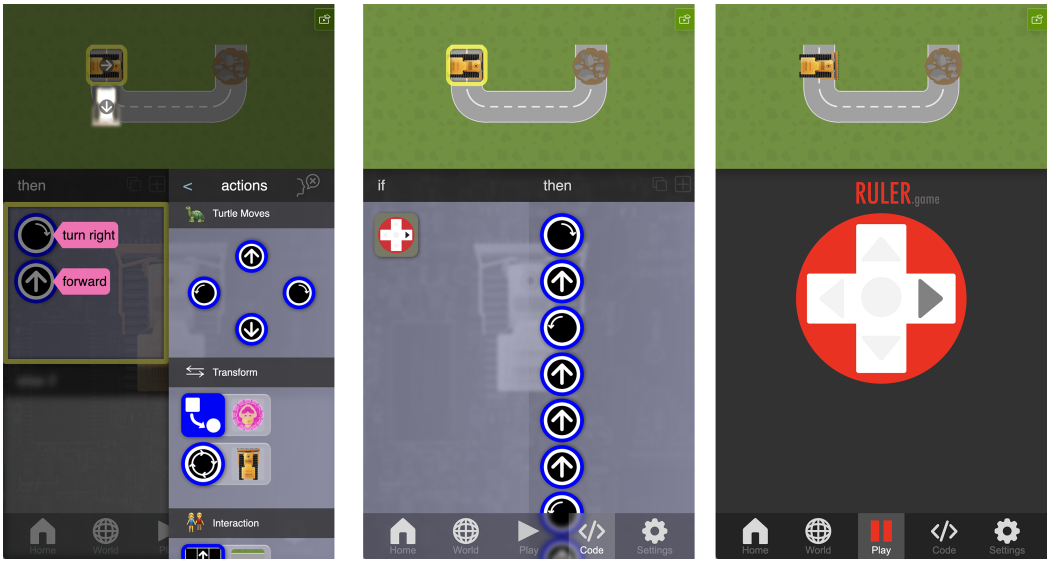

Fig. 4. The proxy executes turnRight and forward actions (left). A D-pad right key is added as a condition to the rule (center). In play mode the digger is started by pressing the D-pad right key (right).

Yumna then faces a common challenge—determining whether the digger should turn left or right. While from a bird's point of view it appears the digger should move right, from the digger's point of view (facing down) it needs to turn left. Figure 1 shows the result of making the wrong decision by turning right. Our usability studies showed many 10-year-olds struggling with this spatial reasoning, often tilting their heads or tablets to better understand the digger's viewpoint. This embodied understanding helps Yumna correctly choose the turnLeft() action and finishes the action sequence. She then sets up the IF part of the rule to trigger the rule with a **directional pad (D-pad)** key (Figure 4, center).

Finally, Yumna can test her complete program. The D-pad shows its right button has been programmed (Figure 4, right), and when Yumna presses that button, the digger successfully navigates to the construction site.

This example illustrates how PBP implements live programming in a focused and safe way avoiding unwanted side effects. While traditional live programming shows the effects of code changes immediately in the actual game world, RULER.game contains these effects within the proxy sandbox. When Yumna experiments with different actions like turnLeft() or forward(), she immediately sees how each instruction affects the proxy digger's behavior—embodying the immediate feedback principle of live programming. However, unlike traditional live programming where code changes could have unintended consequences across the game world, the proxy sandbox ensures that experimentation remains localized and reversible. Only after Yumna is satisfied with the behavior she has developed through live experimentation does the code get applied to the actual digger. This focused approach helps reduce the complexity that often makes live programming challenging in games, where objects have complex interactions and state dependencies.

3.4 Progressive PBP

The extensive scaffolding described in this article, for instance when using spoken explanations, might give readers the impression that PBP's proactive features, while pedagogically valuable for beginners, could create tedious and slow overhead for more advanced programmers. However,

our experience suggests the opposite. The immediate visual feedback and prebugging capabilities actually accelerate the programming process by blending ideas of live programming and programming by example. To demonstrate this efficiency, one teacher created a complete two-player Whac-a-Mole game including scoring functionality in just 56 seconds. This speed demonstrates that the scaffolding mechanisms, rather than hindering advanced users, can streamline the development process by providing immediate clarity about code behavior and preventing the time-consuming errors that typically slow down programming workflows.

The four PBP principles work together to enable this highly fluid programming process that we call *progressive PBP*, which enables learners to iteratively refine their solutions through rapid cycles of observation, coding, and testing:

— *Situation → Code*: Observe the current game state and identify missing or problematic behavior. Use demonstrational palettes to write situationally appropriate code and employ prebugging to visualize how the code will transform the situation.
— *Test and Observe*: Run the code and watch how the game behaves until a new important situation emerges—perhaps an unexpected edge case that reveals the need for additional programming.
— *Iterate*: If all edge cases are handled, stop. Otherwise, return to step 1 with the new situation, building upon previous code to handle increasingly complex behaviors.

Progressive PBP enables novice programmers to tackle complex problems incrementally, starting with simple behaviors and progressively adding sophistication as new situations arise, rather than trying to anticipate all possible scenarios from the beginning. A scenario is used to illustrate progressive PBP.

Kwame, a twelve-year-old 5th grader, has finished the Kodetu challenges and is ready for a more advanced project. He picks the "Maze Solver" project. Unlike with Hour of Code-like puzzles, the goal is not to code one specific sequence of instructions to deal with one specific maze but to become a more advanced computational thinker implementing a universal algorithm applicable to an infinite universe of mazes. The tutorial hints at the so-called "right hand rules" maze-solving algorithm [53].

When Kwame programs his maze-solving digger, the demonstration palettes, employing ideas from programming by example [10, 42], dynamically highlight conditions that are true in the current situation—walls, open spaces, and positional relationships—allowing him to consciously select which factors should influence his rules rather than having the system make potentially flawed inferences automatically. This approach transforms programming by example from a purely inferential system to a collaborative one where the system provides contextually relevant options based on the current situation, but the programmer maintains control over which conditions and actions become part of the rule, resulting in more robust code that works reliably across the intended scenarios without being either too specific or too general.

Kwame employs demonstrational palettes to program the digger to navigate through a maze using the right-hand rule algorithm. Starting with Rule #1, he creates a condition–action pair: if the space to the digger's right is empty, then turn right and move forward one step (Figure 6, left). The "Turtle See" conditions in Figure 5 reflect the current situation—the digger is facing up with walls ahead and to its left, while floors are to its right and behind it. Importantly, all actions and conditions used in this problem are based on "Turtle" geometry—using relative coordinates indicated by circle-shaped instructions. For example, an arrow pointing right means "seeing on my right" from the digger's point of view, not absolute right on the screen. These relative coordinates are essential, as absolute coordinates would not work when the agent to be programmed is rotated.

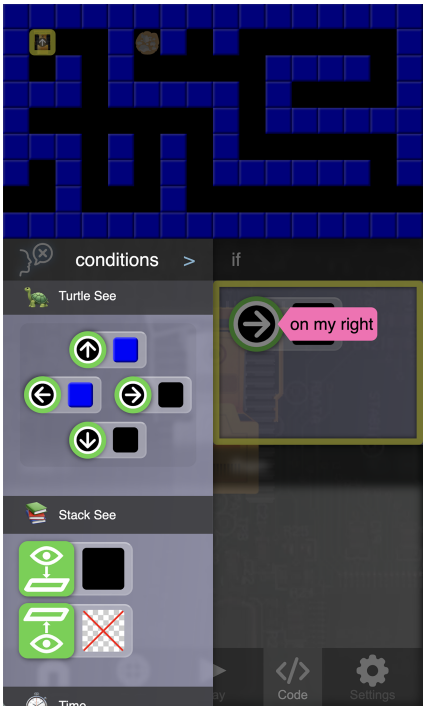

Fig. 5. Demonstrational Conditions Palettes support programming by example by providing relevant true conditions. The "Turtle See" conditions show blue walls ahead and to left of digger; black floor to right and behind digger.

Testing reveals this single rule is insufficient. The digger turns and moves forward one step (Figure 6, center). Notice the onMyRight(floor) condition has turned red indicating that it is no longer true. Now, Kwame adds Rule #2: If there is a floor ahead of me, move forward (Figure 6, middle). With these two rules working together, the digger successfully advances until it reaches a cul-de-sac (Figure 6, right). To handle this final scenario, Kwame initially programs a third rule instructing the digger to rotate left twice (180 degrees). However, Safiya, collaborating with Kwame, suggests a more elegant solution—a single left rotation that accomplishes the same goal with greater efficiency (Figure 6, right). Rule #3 will be applied once to turn left 90 degrees. Then the same rule will be applied again to complete a 180 degree turn.

Throughout this process, programming, playing, and discovering new edge cases go hand in hand through progressive PBP, creating a highly efficient workflow that should be afforded by any computational thinking tool. The demonstrational palettes dynamically highlight relevant conditions based on the digger's current situation, helping Kwame select appropriate rules without having to write abstract code, while each testing cycle naturally reveals new scenarios that guide further development.

3.5 Proxies versus Turtles

In terms of programming environments for kids the Logo programming language developed by Papert [57] has the longest history of advancing the state of embodied computing. Starting originally with physical embodiment by offering programming "turtle" robots, the Logo programming

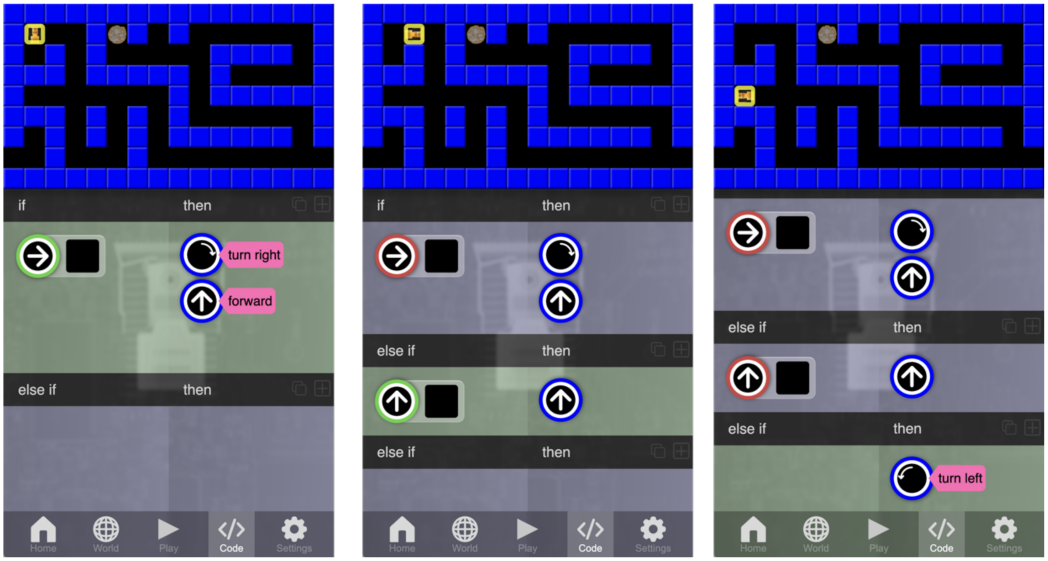

Fig. 6. Progressive PBP. Left: Rule #1 if floor to my right then turn right and move forward; Middle: move forward if there is floor ahead; Right: else turn left. The code is annotated to show true (= green) and false (= red) conditions. Green rules denote current code path.

language migrated from physical to virtual embodiment. Speculating on Seymour Papert's transition from physical robots to virtual turtles reveals a fascinating intersection of technological evolution, educational philosophy, and pragmatic innovation. While the exact motivations can only be partially reconstructed, it appears that Papert recognized a pivotal opportunity to democratize computational learning by transcending the physical limitations of early robotics. The virtual turtle became a metaphorical bridge, transforming complex programming concepts into an accessible, playful medium that could reach far more students than expensive, fragile physical robots. This shift was not merely a technological compromise, but a deliberate reimagining of how learning occurs—creating an environment where students could explore computational thinking with unprecedented freedom, unburdened by the mechanical constraints of physical hardware. By rendering the learning process more immediate, interactive, and scalable, Papert seemingly sought to make programming not just a technical skill, but a form of intellectual exploration that could spark curiosity and creativity in young minds across diverse educational contexts.

Now let's revisit the four PBP design principles by comparing proxies found in RULER.game with turtles found in Logo:

(1) *Proxy*: Logo does not provide a proxy or dual temporal representation. Instead, the built-in turtle serves as a single embodied object that users program directly. The virtual turtle concept emerged from earlier physical Logo robots where students could literally touch and orient the turtle before programming it. Logo's syntonic potential is significant precisely because there is only one object for users to embody, and the turtle's visual design—a turtle with legs and a head—makes orientation immediately apparent. However, Logo shows only the turtle's current state; users cannot preview future positions or movements before executing commands, requiring them to mentally simulate outcomes.
(2) *Sandbox*: Logo lacks a sandbox environment because its simplified world consists primarily of a single turtle creating drawings on a blank canvas with minimal object interactions.

When bugs occur, they typically manifest as incorrect drawings or turtle positions, requiring users to modify their code, clear the canvas, and re-execute the entire program, a cycle that becomes increasingly tedious with complex drawings. The essential sandbox affordances of isolation, safety, and replication are absent from Logo's architecture.

(3) *Prebugging*: Logo does not support proactive exploration of code effects before execution neither of syntactic nor of pragmatic nature. However, Logo does offer reactive exploration through its REPL (Read-Evaluate-Print Loop), which allows users to test individual commands or user-defined functions interactively. This requires users to take initiative to test code fragments, whereas PBP automatically shows consequences of proposed actions.

(4) *Demonstrational Palettes*: Logo does not feature command palettes and typically provides only static command references that lack situational awareness. Unlike demonstrational palettes that adapt to the current context (e.g., showing "seeAhead('wall')" when a wall is actually ahead), Logo's command set remains constant regardless of the turtle's current situation or surrounding objects.

3.6 Proxies versus Sprites

Not all block-based programming environments are alike, but for comparison we use the Scratch programming environment because of its popularity. Many block-based programming languages have been implemented using Blockly [22, 79], a block-programming language construction kit that enables developers to build block-based programming environments for various applications. The widespread adoption of Blockly has resulted in a large ecosystem of Blockly-based visual programming tools, effectively establishing it as a *de facto* standard for visual programming development. While this standardization offers benefits such as consistency across tools and reduced development effort, it may have inadvertently constrained the broader evolution of visual programming concepts. The ease with which Blockly enables the creation of visual programming environments could have led to a form of technological path dependence, where the convenience of using an established framework discouraged exploration of alternative visual programming paradigms. This standardization around Blockly's particular approach to visual programming—jigsaw puzzle-like blocks that primarily address syntactic concerns—may partially explain the limited research exploring programming support beyond syntax. Rather than experimenting with novel visual representations or interaction models that could address semantic and pragmatic challenges, many developers have naturally gravitated toward Blockly's proven but constrained design patterns. Notably, in this era of AI-enhanced coding where tools like GitHub Copilot and ChatGPT increasingly handle syntactic concerns, the focus on syntax-centric programming support becomes less relevant to the fundamental challenges facing novice programmers.

Now let's revisit the four PBP design principles by comparing proxies found in RULER.game with sprites found in Scratch:

(1) *Proxy*: Scratch does not provide a proxy and does not feature a dual temporal representation. While every sprite could be considered a syntonic embodiment, Scratch has limited syntonic potential for several reasons. When only looking at the stage and when there are multiple sprites on stage, users cannot easily tell which object they're currently programming because there are no explicit visual cues such as object selection highlights. User-created sprites often lack clear orientation markers (like RULER.game's annotation arrows), making it difficult for users to understand directional commands from the sprite's point of view. Most importantly, Scratch provides only limited ways to query the surrounding situation—for example, there's no simple way to express a rule like "IF seeAhead(food) THEN move(distance)." This reduces syntonic potential because even when users feel they embody an object, Scratch may not

provide computational affordances to leverage that embodied understanding in a situationally aware way.

(2) *Sandbox*: Scratch does not have a sandbox. While Scratch could be considered a microworld for programming experimentation, it lacks the three main affordances of a true sandbox: isolation of experimental changes from the main project state, safety mechanisms that prevent code execution from causing unwanted side effects, and replication that allows testing without affecting the production environment. Typical experimentation consists of clicking blocks to explore their behavior by executing them. Execution can have unwanted side effects, e.g., sprites may move outside the stage, a sprite may become invisible, or a large number of misinitialized clones may be accidentally created, spamming the stage. These side effects may require tedious, difficult or even impossible manual resetting procedures to return the project to a desired state. Undoing the execution of a block requires manual execution of code that reverses the state change. For instance, when testing a moveTo(420, 780) block that accidentally moves a sprite off the stage, the user would have to undo the action manually by issuing a reversing moveTo(−420, −780) action. When testing compound blocks that combine multiple move and rotation actions, this manual undo procedure quickly becomes difficult to manage. Moreover, some actions are not reversible. Scratch provides an undo function working for code edits but not for code execution. A sandbox allows safe experimentation while avoiding all these pitfalls.

(3) *Prebugging*: Scratch supports limited proactive exploration through two distinct modes: stopped and running. When stopped, Scratch allows testing blocks by clicking them, but this reactive exploration has significant limitations. Testing blocks nested within other structures proves difficult—clicking a nested block executes the entire outermost container, not just the specific clicked instruction. To test individual instructions, users must manually isolate them by dragging them out of the containing block, copying the block, or creating new identical blocks from the palette. This process becomes cumbersome for exploring specific nested behaviors. When running, code changes can manifest proactively if the modified instruction is in the active execution path. For instance, changing the angle parameter of the turnRight(10) block to "−10" in a running forever loop immediately reverses the spinning object's direction, consistent with live programming. However, if instructions are not in the current execution path—perhaps nested in some if statement with unsatisfied conditions—users won't see consequences of their changes. Additionally, running mode can create unwanted side effects, as testing specific code requires running the entire project, potentially advancing game state or triggering unrelated behaviors. Unlike RULER.game's proactive disclosure of instruction behavior when adding code, Scratch does not automatically show what code will do in specific situations. Scratch also lacks condition prebugging—users must manually click isolated conditions to explore truth values, and clicking conditions within clusters executes entire blocks. While Scratch allows reactive exploration with effort, proactive approaches like RULER.game are simpler. Compare manually isolating and testing condition truth values with Figure 6's live visualization, where truth values and active code paths appear as automatic real-time annotations requiring no user initiative. Given these rather cumbersome debugging mechanisms that require significant manual effort and technical know-how to use effectively, it seems somewhat likely that Scratch users—particularly novices—simply avoid using these testing features altogether, relying instead on trial-and-error approaches [12] that lack the seamless, automatic support provided by RULER.game's proactive equivalents.

(4) *Demonstrational Palettes*: Scratch does have instruction palettes offering a wide range of blocks. However, these palettes are mostly static and do not adapt to the current situation—they cannot be considered demonstrational in the pragmatic sense. There is some

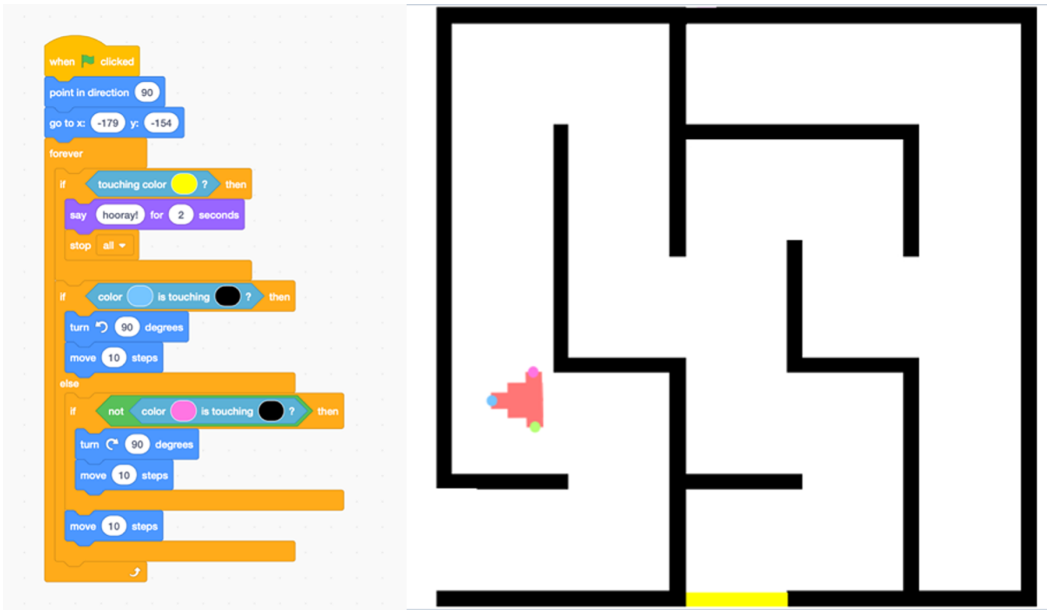

Fig. 7. Right hand rule code and world.

limited reflective awareness of the sprite itself. For instance, when you manually drag a sprite around the stage, the goTo(x, y) block contained in the palette automatically updates to show the sprite's current position. However, this automatic updating does not consider the broader context of other sprites in the scene. Imagine a Scratch game with a cat chasing a mouse. When you manually move the cat so it touches the mouse, a demonstrational palette would automatically suggest touching("mouse") in the condition blocks to help you write relevant code. Scratch does not do this. For conditions especially, offering options that are currently true would be more helpful than random false conditions—true conditions give you immediate, useful building blocks for your code. Compare Scratch's mostly static palettes to RULER.game's proactive approach (Figure 6), where demonstrational palettes automatically suggest the most relevant code options based on what's actually happening in the game world. This provides such useful suggestions that users rarely need to manually adjust instruction parameters. Moreover, the lack of demonstrational palettes also prevents the kind of progressive programming described in Section 3.4.

To ground these principles in a more concrete discussion let's consider the implementation of the right hand rule algorithm in Scratch. This project was created by an actual Scratch user. What is important here is how Scratch affordances, or lack thereof, shape the programming and debugging process in ways different from PBP. The structure, and therefore the McCabe cyclomatic complexity [16], measured by number of decisions points, of the Scratch code in Figure 7 and the RULER.game code in Figure 6 are identical and do also match the general definition of the right hand rule algorithm: three cases that can be captured by three if statements.

The limited embodiment of Scratch sprites can make debugging difficult. In PBP embodiment is about helping users to overcome pragmatic challenges by helping them to understand the connection between code and situation. Users can change the situation to explore which code path will be taken and they can change code to explore how this will change the situation. For instance,

in RULER.game a user could change the situation by changing the position and orientation of the digger in the maze to instantly see which code path would be chosen. Similarly, in Scratch the user could change the situation by changing the position and orientation of the sprite in the maze (Figure 7). However, this would not reveal the code path. Clicking the conditions in the Scratch code manually would provide no insight either as this would trigger the execution of the containing block resulting in resetting the project and then running it from beginning to end.

Beyond debugging challenges, limited embodiment combined with lack of expressive power will result in compromises of gameplay that kids can program—for instance, making it almost impossible to create a compelling version of games such as Pac-Man where ghosts are able to engage in path following to find the shortest path to the Pac-Man, forcing developers to fall back to less compelling gameplay such as ghosts moving randomly or moving through walls. This forces them into what Perlis warned about as the "Turing tar-pit" [60] where "everything is possible but nothing of interest is easy," though RULER.game does provide this expressiveness through special AI-powered actions [60].

3.8 Discussion of Virtual Embodiment

An empirical evaluation of PBP demonstrates significant improvements in error prevention [65] compared to traditional block-based programming approaches with established error baseline data [5]. That study comparing error rates between PBP and block-based programming found that PBP reduced total errors by a factor of 10, from 20.8% in block-based programming to just 2.1% with PBP. This substantial reduction was statistically significant across different error types, with orientation errors dropping from 9.71% to 0%, and counting errors decreasing from 13.4% to 2.1%. These findings suggest that PBP focus on pragmatic prebugging—providing proactive visual feedback about the consequences of programming decisions—effectively prevents logical errors before they occur, making programming more accessible particularly for novice programmers.

Beyond empirical evidence exploring error reduction efficacy, RULER.game has been used in the professional development of hundreds of pre-service and in-service primary school teachers with positive reception. Teachers particularly appreciated the simplicity of RULER.game's interface and found the concept of prebugging compelling for classroom management. They could envision themselves teaching large classes where students working on open-ended programming projects would frequently encounter bugs. In traditional programming environments, this scenario creates challenging classroom dynamics with numerous students simultaneously raising their hands for debugging assistance, potentially overwhelming the teacher. Teachers recognized that prebugging could serve as a first line of defense against this common classroom challenge, reducing the debugging burden by preventing many errors from occurring in the first place.

The evidence shown in [65] demonstrates significant benefits, particularly with respect to error prevention, but the critical question arises: what is the pedagogical cost of PBP? One teacher's comment captures this concern well—when students attempted the Kodetu challenges, RULER.game made them "almost too easy." While he appreciated RULER.game's inclusive nature for less engaged or struggling students, he worried that removing certain challenges might also eliminate the cognitive benefits students gain from what Kapur [37] calls productive failure.

This observation reflects the well-known tradeoff between over-scaffolding and under-scaffolding in educational design. Scaffolding refers to temporary support structures that help learners accomplish tasks they could not complete independently, gradually fading as competence develops [84]. Over-scaffolding may make problem solving more accessible but can remove essential failure points that promote deep learning. Conversely, under-scaffolding can provide an evocative constructionist playground for exploration but presents high potential for frustration resulting in nonproductive failure [37].

3.8.1 Mistakes versus Errors. To address this pedagogical tension resulting from scaffolding, it is crucial to distinguish between different types of difficulties students encounter when programming. PBP still allows mistakes but prevents errors [5]. A mistake refers to a user's incorrect decision-making process, such as choosing the wrong action—for instance, when the digger turns right when it should turn left in Figure 1. An error refers to the incorrect program outcome that manifests during execution, such as the digger appearing in the wrong location—the kind of runtime problem users recognize as a bug.

PBP does not prevent students from making mistakes. However, unlike traditional programming environments, where students write entire programs before discovering resulting errors, PBP acts proactively. The proxy immediately shows consequences of mistaken decisions, allowing students to perceive causality before errors manifest in final program execution. This recognition significantly reduces Eisenstadt's temporal gap, effectively preventing errors from occurring in completed programs while preserving the learning value of mistake recognition.

3.8.2 Limitations and Code Quality Concerns. Despite these error reduction benefits, PBP can result in correct but inefficient code. Students sometimes create redundant solutions when they immediately recognize mistakes through proxy visualization but choose compensation over correction. For instance, when programming a digger to turn left incorrectly, instead of replacing the erroneous instruction, some students add compensatory actions—either continuing to turn left three times (equivalent to turning right once) or adding two right turns to counteract the incorrect left turn. While functionally correct, this approach introduces code inefficiency and potentially reinforces suboptimal programming practices which could be considered code smells [28]. These issues could potentially be addressed by encouraging students to write programs that achieve correctness with minimal instruction length.

4 Physical Embodiment: Draw Your Own Video Game

Research Aim #2: How does physical embodiment reduce affective challenges? How does incorporating tangible, hands-on creation processes into programming activities address the motivational and engagement barriers that often discourage students from pursuing computational thinking?

The growing backlash against technology in education reflects serious concerns about screen time's impact on child development. A recent surge in opposition to computer use in schools can be observed internationally. For instance, Sweden, in spite of compelling negative evidence [44], has restricted the use of computers and tablets in pre-kindergarten education. Other countries, including Switzerland, are considering similar limitations for primary schools (grades 1–6). Political discourse around this issue often centers on children's excessive screen time. These concerns raise important questions about how programming education can maintain its educational value while addressing legitimate worries about digital overexposure and student disengagement.

These concerns span multiple dimensions, including potential harm to eyesight and the diversion of time and energy toward social media. Notably, around 2013, the Flynn effect—a long-documented phenomenon of rising average IQ scores across generations [88]—reversed direction. This "Negative Flynn Effect" [15] has prompted researchers to investigate various potential causes. While environmental factors like microplastics have been suggested, a more widely discussed theory points to excessive social media consumption among children without providing compelling evidence [9].

Attention spans appear to be diminishing in this environment. Recent research [75] indicates that nearly 50% of TikTok users experience stress when watching videos longer than 60 seconds, with many choosing to view content at accelerated 2× speeds. The emergence of this "TikTok generation" represents a concerning societal shift where sustained engagement with challenging tasks is increasingly displaced by brief, algorithm-driven content consumption.

Pestalozzi's "head, heart, and hands" educational philosophy offers a valuable framework for computer science education in today's rapidly evolving technological landscape. With generative AI evolving at breakneck speeds and some already prophesizing "The End of Programming" [86] computer science education needs to carefully assess goals as well as approaches to teach programming and computational thinking [4, 70] in the context of K-12 education. One approach is to fall back to traditional pedagogical models such as the "head, heart, and hands" model of Pestalozzi. The Swiss educational reformer described this educational philosophy in the early 19th century. His philosophy laid the groundwork for modern progressive education and influenced later educators like Maria Montessori [43]. With heart and brain Pestalozzi was referring to cognitive and affective challenges which computer science education has explored for some time [63]. Cognitive challenges, referred to by "brain," have been addressed with block-based programming. "Heart" is about students finding projects that are personally meaningful to them including programming robots, creating stories and designing games. With "hand" Pestalozzi referred to the benefits of touching and therefore sensing physically. In the context of computer science education this has been explored mostly by the maker movement where K-12 students build physical artifacts with their hands. Students solder components for microcontroller-based art pieces. They also design and print physical objects with 3D printers.

Educational robotics reveals both the benefits and limitations of physical computing in computer science education. Some have explored the idea of programmable robots. LEGOSheets [24] was an early prototype of a programming environment for LEGO. The LEGO programmable brick was later developed into the LEGO Mindstorm robotic kit. In this context, we learned much about the pros and cons of programming. On one hand there was a lot of excitement by students programming, for instance, a robot following a black line on the floor. On the other hand, constrained by physical constraints of the real world such as gravity and the need to master sophisticated LEGO building skills, it appeared that the universe of possible creation was quite a bit more limited for most kids than drawing arbitrary worlds on the computer. As outlined above, this may be one of the more conceptual reasons why Papert moved from the physical world of robots to the virtual worlds of the turtle programmed in Logo on the computer screen.

RULER.game bridges physical and virtual worlds through a hybrid approach that honors Pestalozzi's educational philosophy. Supporting the "head, heart, and hands" philosophy but trying to avoid some of the physical constraints experienced in educational robotics RULER.game, running often on tablets devices such as Apple iPads, explores a hybrid model connecting the physical world with the virtual one through camera functions (Figure 8). Children draw characters on paper and then import them into their projects using RULER.game's automatic background removal, which seamlessly extracts their drawings without manual editing.

Physical embodiment of computing concepts aligns perfectly with Pestalozzi's emphasis on tactile learning experiences. In schools and STEAM (Science, Technology, Engineering, Art, and Math) [23] Discovery Fairs, the concept of students drawing their own video game characters has been explored with great success. Students put a lot of energy into drawing elaborate characters (Figure 9, left) using paper and colored pens. They then select backgrounds, which they could also draw themselves, and begin programming their games (Figure 9, right). Many backgrounds were created with generative AI and imported using the camera with the world background function.

Student engagement with physical drawing activities creates powerful connections to digital programming experiences. At one STEAM Discovery Fair, the "draw your own video game" station, aided by 2–3 pre-service teacher students from the FHNW Switzerland, School of Education, was barely able to keep up with the barrage of kids eager to make their own video games. In the few moments, when there was no line waiting to participate, kids were allowed to continue programming to create additional characters and to extend functionality with features like collision

Fig. 8. Shapes can be imported with automatic background removal from paper in RULER.game.

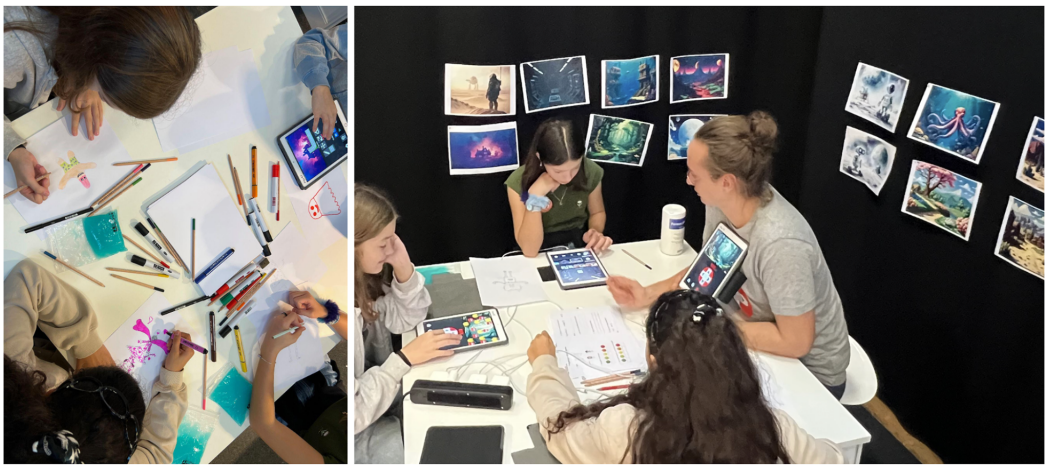

Fig. 9. First Table: Draw your own character (left). Second Table: Programming and play testing (right).

detection or score counting. The minimal expectation was for kids to create a game with one character that they could move around the world (Figure 10).

The physical-digital hybrid approach fosters sustained engagement and social learning opportunities. Many kids returned to the station in the hopes to continue with programming. Some came back waiting for multi-person openings to work on games with their friends. Others returned on different days accompanied by teachers, parents, or even grandparents.

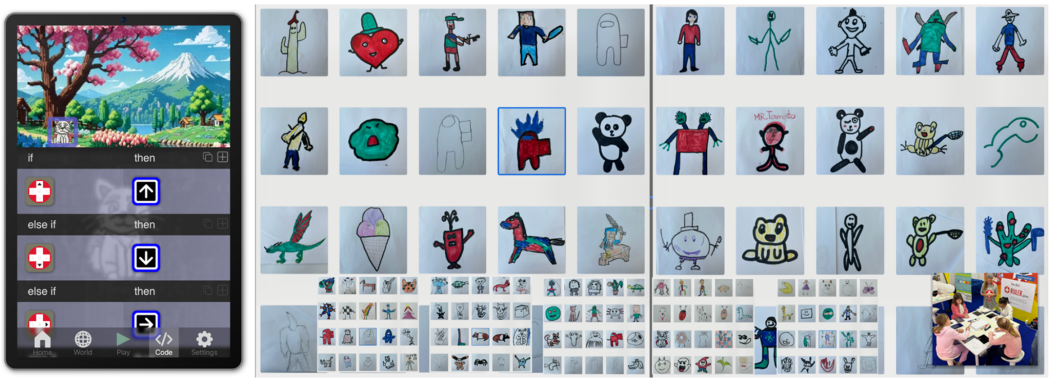

Fig. 10. RULER.game with one character programmed (left). Example characters drawn by 6–14 years old students (right).

4.1 Discussion of Physical Embodiment

A study exploring affective efficacy in informal settings [62] demonstrates that constructionist learning principles can successfully engage today's "TikTok generation" despite their shortened attention spans, with TikTok's own research showing 50% of users find videos over 60 seconds stressful. Crucially addressing Research Aim #2, the paper reveals how RULER.game's physical embodiment approach—where students draw video game characters on paper before digitizing them—directly tackles affective challenges by creating powerful emotional connections to digital programming. This tangible, hands-on creation process transforms abstract programming concepts into personally meaningful artifacts, with students investing significant energy into drawing elaborate characters that become the foundation of their digital games. The physical-to-digital transition bridges Pestalozzi's "head, heart, and hands" educational philosophy, demonstrating that when students create something tangible first, they develop stronger motivation to engage with the programming aspects. The findings revealed high overall engagement levels and striking gender differences: girls derived primary enjoyment from design elements while boys favored programming components, suggesting that physical embodiment provides multiple pathways for sustained computational thinking engagement.

The study does have some limitations in its methodology and scope. As an observational study conducted at discovery fairs rather than in controlled classroom settings, it provides preliminary insights but would benefit from more systematic data collection and longer-term follow-up to assess sustained engagement. The 15-minute activity window, while demonstrating rapid initial engagement, offers limited insight into whether the physical embodiment approach leads to lasting reductions in affective barriers to computational thinking. The research also focuses on a specific context, which may limit generalizability to other educational environments. While the gender-based findings about design preferences are encouraging, they represent early observations that would benefit from validation with larger, more diverse samples to better understand how physical creation processes address motivational barriers across different student populations.

Pestalozzi's "Head, Heart, and Hands" philosophy suggests physical embodiment contributes significantly to affective engagement. Looking back at the Pestalozzi "Head, Heart, and Hands" learning philosophy we can only speculate that physical embodiment, that is the drawing of characters on paper, contributed as "hand" oriented learning activity towards the "heart" aspect. Dewey [14] introduced the notion of *instrumental motivation*. Instrumental motivation refers to the idea that learning is driven by the practical use of knowledge as a tool to solve real-world

problems, emphasizing the connection between education and experience. It is at least conceivable that some students were not intrinsically motivated to learn how to program but that programming was perceived as an "instrument" to bring their designs to life.

The synergistic combination of virtual and physical embodiment creates powerful educational affordances for programming education. Getting students interested in creating their own video games results in learning activities that effectively engage them in programming these games, likely due to the combination of virtual and physical embodiment. Building upon these engagement principles, although the individual components of this approach, including scanning images from paper and utilizing block-based programming for game creation, are not inherently novel, the significant reduction in accessibility barriers represents a unique contribution. It is quite challenging to tease apart the individual contributions of the Pestalozzi triad, yet the ability for users to create sophisticated games like Pac-Man, complete with complex gameplay mechanics such as pathfinding algorithms and collaborative elements, demonstrates that while the underlying technologies may be familiar, their combination and implementation dramatically lower the threshold for advanced game development. All we can say is that the combination appears to be quite promising by not only making programming more accessible through virtual embodiment of PBP, but also more exciting through physical embodiment of drawing personally meaningful characters with pens on paper, ultimately making previously complex programming concepts accessible to a broader range of users.

Beyond direct student impact, the hybrid embodiment approach has garnered positive reception from educational stakeholders and policymakers. Aside from concrete impact on K-12 students, we should also point out the more general positive perception by teachers, parents, and even politicians. Working with thousands of primary school teachers, both preservice and in-service, we are still witnessing strong skepticism toward teaching programming in schools. In some countries, preservice primary school teachers are required to take computer science courses including programming—they are not self-selecting teachers interested in programming but are required to take these courses. In this context, we have found the idea of combined physical and virtual embodiment to be quite attractive. The physical embodiment is not only appealing because it can reduce screen time, but it also provides important practical consequences for school implementation. For instance, single or double programming lessons in K-12 classes (45 or 90 minute slots) are often plagued by limited efficacy because meaningful programming projects are difficult to fit into these timeframes. When combining computer science with Art, however, critical mass of time on task can be achieved. Additionally, the virtual embodiment of RULER.game, which affords significantly fewer errors, could make the difference between a project-oriented programming course running smoothly versus one resulting in chaos with the teacher desperately trying to debug numerous projects simultaneously.

5 Social Embodiment: Learning Through Collective Agency

Research Aim #3: How does social embodiment support collaborative learning in C^2T^2s*?* How does extending programming beyond individual interaction to include collaborative dimensions through awareness interfaces and shared virtual spaces enhance peer learning, coordination, and collective knowledge construction in computational thinking activities for primary school students?

To explore this research aim, we examine how social embodiment manifests in C^2T^2s. Social embodiment in RULER.game supports collaborative programming by extending the embodied experience beyond individual interaction to include collaborative dimensions. When multiple users program together, they become part of a shared virtual space where their actions are made visible through awareness indicators that show who is editing what. This creates a form of social presence within the programming environment, allowing programmers to perceive and respond

to each other's actions in real-time. Just as physical embodiment connects abstract programming concepts to tangible experiences, social embodiment connects individual programming activities to a collaborative social context. Through these awareness interfaces [27], RULER.game transforms programming from a solitary cognitive activity into a socially embodied experience where users can coordinate their efforts, observe others' problem-solving approaches, and engage in collective knowledge construction. This social embodiment enhances the learning environment by making visible the thinking processes of others, creating opportunities for peer learning, and fostering a sense of community within the programming space.

The ultimate goal for RULER.game is to become a C^2T^2 [68] combining virtual embodiment, physical embodiment, and social embodiment. That is, in addition to addressing the Pestalozzi learning philosophy triad ("Head, Heart, and Hands") discussed above, C^2T^2add a social dimension best captured by Vygotsky's theories of social learning. He emphasizes that learning is a socially mediated process. C^2T^2s introduce awareness interfaces to facilitate social embodiment with the goal to make the actions, intentions, and thought processes of multiple users visible in real-time.

Social embodiment has not been researched in depth in RULER.game, though the current implementation allows users to collaborate in real-time on common game design projects. There is little controversy that social approaches to learning and learning to program could be beneficial [83]. Pair programming [6, 8, 52, 71, 80, 82, 87, 89] has been explored for some time as an effective collaboration approach where typically two programmers share one computer to program together. More recent research found that pair programming based on sharing a single piece of hardware is highly problematic with primary school kids [81]. The high probability of conflict [80] has resulted in different approaches to provide simultaneous control to both users. Some have explored the idea of attaching multiple mice to a single computer with limited success [33] while others were more successful using multiple computers that are networked [89]. Few programming languages aimed at kids [7] enable multiple users to control programs. However, without awareness interfaces the students intending to collaborate reported confusion [81] not knowing who was doing what when to what part of a project. Precisely this shortcoming could be mitigated by social embodiment.

Social embodiment in RULER.game works through intuitive collaboration features that enable students to easily join each other's projects in real-time. Let's explore how social embodiment works. Imagine that Miguel has started making a Pac-Man like project. As he is drawing his own Pac-Man and ghost agents Kwame walks by and expresses interest in participating in his project. Miguel starts the Buddy Radar to find potential partners. On their respective iPads they establish a collaboration (Figure 11):

(1) Miguel initiates a search using the Buddy radar. He notices Yumna and Kwame.
(2) Kwame, interested in participation, raises his virtual hand.
(3) Kwame moves into the inner circle of trust. Miguel selects Kwame and hits the OK button to confirm the collaboration.
(4) Kwame gets the confirmation including the current copy of the project.

Social embodiment consists of the representation of collaborators including their screen name and some emoji. Participants can select different levels of engagement and circles of trust:

—*Network presence* (outermost circle)–all potential collaborators in the same network
—*Participation intent* (middle circle)—those actively signaling readiness to collaborate
—*Location trust* (innermost circle)—those willing to share their precise location

Miguel and Kwame can work together in RULER.game using either collaborative or cooperative approaches [11] to game development. They can collaboratively or cooperatively edit the world, introduce new custom sounds, add custom agents, and, perhaps most importantly, program these

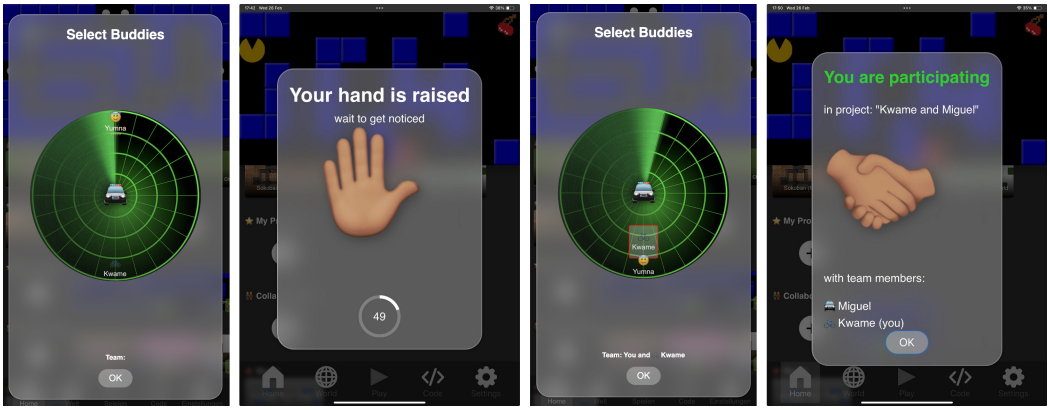

Fig. 11. Miguel and Kwame connect. (1) Miguel starts a search. (2) Kwame raises his hand. (3) Miguel notices Kwame moving into the inner circle and selects her. (4) Kwame and Miguel are now collaborators.

agents. In collaborative work, team members work together synchronously toward shared goals with continuous interaction and joint decision-making throughout the entire process. In contrast, cooperative work involves team members working independently on separate parts of a project with more limited interaction, typically combining their individual contributions at the end to create the final product. A potluck party is a perfect example of cooperation, where each guest brings their own dish or drink with minimal or no prior coordination about what to bring, resulting in a collective meal composed of individual contributions.

The RULER.game awareness interface implements social embodiment through visual indicators that show where each user is working within the shared project. The awareness interface works similarly to those in Google Docs and other shared document editors. Users can spot where they are editing and, more importantly, where others are working. In Figure 12, left, Miguel (blue selection color) is currently working on the Pac-Man agent. He can see that Kwame (green selection color) is working on the ghost. Conversely, in Figure 12, right, Kwame, who is working on the ghost, can see that Miguel is working on Pac-Man. A thick border suggests one's own selection whereas the small borders indicate the selections of collaborators. Disjoint selections typically suggest cooperative edits. While Miguel can see that Kwame is editing the ghost's behavior, he does not actually see the specific changes Kwame is making in real-time.

When transitioning from cooperation to collaboration, Miguel and Kwame work simultaneously on the same game element to create more sophisticated AI behavior. Dissatisfied with the fake AI, i.e., the ghost just moving randomly, Miguel and Kwame transition from cooperation to collaboration (Figure 13). Notice in the world the nested selection annotations of the ghost agent suggesting that two people are editing the same agent. Kwame (1) removes the moveRandom action, while Miguel (2) can see the collaboration awareness information by noticing in the world that both have selected the ghost. In the behavior editor, he can see which part of the code Kwame is editing. Kwame (3) has replaced the randomMove() action with the pursue AI action [60], enabling the ghost to track down Pac-Man using smart pathfinding based on collaborative diffusion [61]. This enhancement allows multiple ghosts to collaborate in pursuing their target. Because AI with collaborating ghost work so well that it would be difficult to win the Pac-Man game Miguel decides to add a speed control action to make ghosts move at slow 0.25× speed. The social embodiment represented by the awareness indicators (thick green frame with nested thin blue frame) shows that both users are editing not only the same agent but even the same exact rule of that agent.

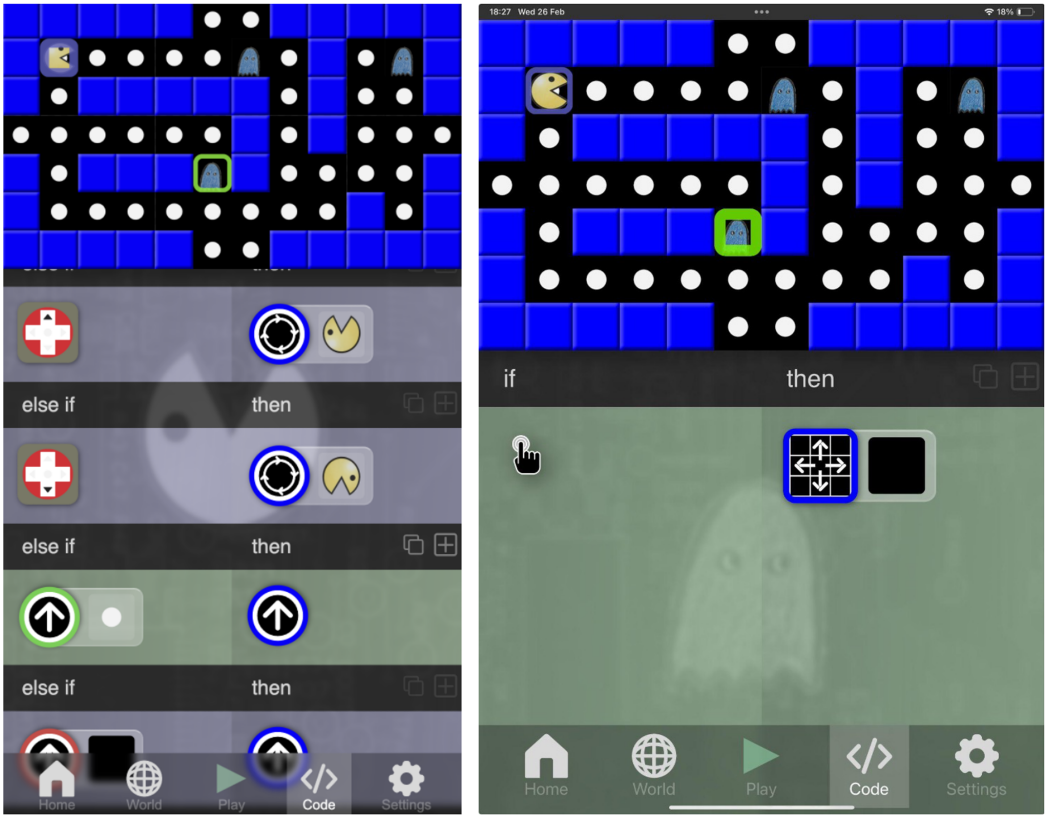

Fig. 12. Cooperation (left) Miguel is programming Pac-Man and (right) Kwame is programming the ghost.

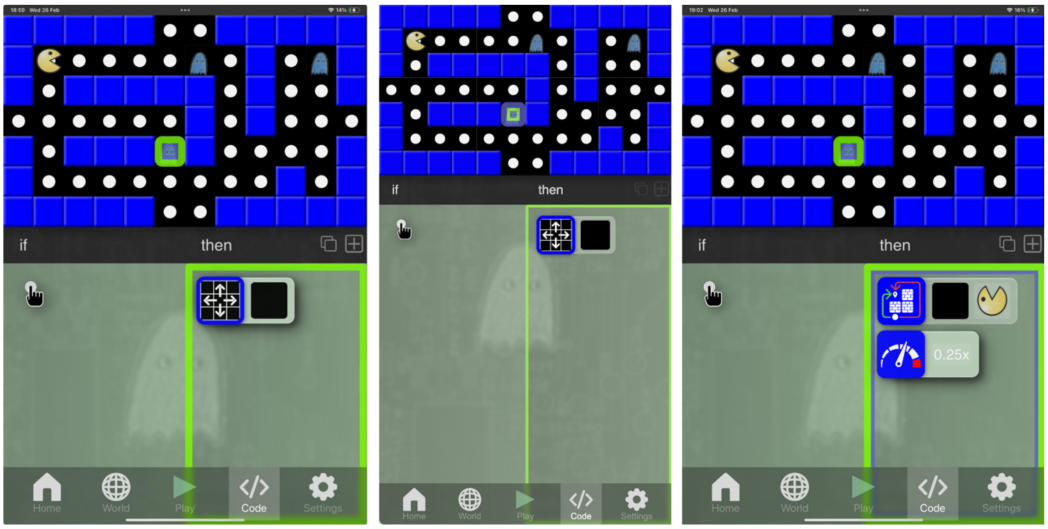

Fig. 13. Collaboration: (1) Kwame edits the ghost behavior, (2) Miguel can see the edit awareness, and (3) Kwame and Miguel simultaneously edit the same rule resulting in potential conflict.

Social embodiment in RULER.game has a hierarchical structure that distinguishes it from traditional awareness interfaces found in document editors. Unlike with awareness interfaces found in most shared document editors such as Google Doc there is a hierarchical nature of selection. Users can select agents and within each agent they can select fragments of code to be edited. Edits can be contradictory. For instance, two users may have different code in mind resulting in tug of war. They may want to edit the same code fragment in different ways. Synchronization in RULER.game implemented as Conflict-free Replicated Data Type [38] with the characteristic of eventual consistency. However, this will only guarantee that the values will eventually be the same but it does not suggest that there is no conflict between users. For instance, Miguel and Kwame may disagree about the "right" sequence in Figure 13(3), should speed be set first or last? In these kinds of situations RULER.game will use the *last write wins* conflict resolution typically used in database systems.

Social embodiment facilitates the detection and resolution of conflicting edits among collaborators. In the struggle outlined above, where Miguel and Kwame are having different opinions they can make use of the awareness information as it reflects intentions of social embodiment. First, because of the nested annotations, they can clearly see that there is a potential conflict and they can also see who the conflict is between. After all, the number of collaborating users is not limited to just two. There could be other users working on the project but not being involved in this particular conflict. If they are aware of the "last write wins" resolution they could play a game of chicken. If they are stubborn they could draw out the conflict resolution for a very long time until the first user loses patience and gives up.

RULER.game supports both synchronous collaboration and asynchronous cooperation through specialized synchronization mechanisms. Collaborators may have started jointly but at one point some collaborators may tap out. When they come back RULER.game needs to synchronize using some merging policy. Because the users of RULER.game are typically novice programmers, such as primary school students, merging mechanisms need to be as simple as possible. For instance, it would not make sense to employ complex merging mechanisms found in distributed version control such as Git aimed at professional programmers. Instead, RULER.game employs the "keep the most sophisticated" version approach. Sophistication is assessed using McCabe cyclomatic complexity measures [16]. This approach could potentially result in some code that was intentionally erased getting accidentally restored because a collaborator joining later may have a more sophisticated version of the code. In these situations, the system applies the heuristic that it is generally simpler for users to delete unwanted code than to restore deleted code that might be needed.

Communication in RULER.game is implemented through peer-to-peer networking for efficient synchronization. Communication necessary for both synchronous and asynchronous collaboration in RULER.game is implemented through peer-to-peer networking. In a classroom setting, where multiple students collaborate with each other, updates are sent directly from one peer to another without needing to route through a central server. This peer-to-peer approach offers significantly better scalability, preventing the central server from being overwhelmed by worldwide traffic and ensuring a smoother collaborative experience.

5.1 Discussion of Social Embodiment

The collaborative features of RULER.game have been implemented as a proof of concept and introduced to preservice teacher education. While these collaboration affordances received positive feedback, they also highlighted the need for new learning designs.

Although the web contains numerous video tutorials demonstrating how to program robots, create games, and author stories with various programming tools, there is very little material to guide teaching computer science collaboratively. This gap exists because collaborative programming

tools for novice programmers are rare. Some guidelines exist for pair programming, but it is unclear how applicable these are for C^2T^2 that do not require traditional roles like driver and navigator due to hardware constraints.

We believe we have entered a largely unexplored research territory that combines collaborative learning and computational thinking. While some programming environments offer basic collaboration affordances, to our knowledge there are no synchronous real-time collaborative programming tools that include critical awareness interfaces specifically designed for primary school students. Current literature has identified significant challenges for younger children programming collaboratively. However, it remains unclear whether these challenges stem from fundamental cognitive limitations (similar to Piaget's stages of cognitive development [32]) or simply from inadequate tool design that fails to support collaboration effectively.

Future research exploring collaborative computational thinking could begin with well-documented programming challenges as baseline data. Researchers should investigate whether children require explicit teacher instruction on collaboration strategies. Data collection should examine collaboration patterns and correlate them with the quality of projects produced, as well as students' enjoyment of social programming processes.

Tools like RULER.game should be instrumented to collect log data on patterns such as synchronous versus asynchronous collaboration and collaboration versus cooperation practices. Complex temporal patterns will likely emerge, with students potentially shifting between cooperation and collaboration throughout a project. The field would advance significantly if we better understood these patterns, particularly those associated with exceptionally successful or unsuccessful projects. Even if we actually would be approaching the prophesied "end of programming" [86], it would be more than unlikely that we approach the end of a need for humans to collaborate. While we may not yet fully understand effective collaboration support in programming for primary-level students, the social aspects of embodiment relevant to collaborative learning may ultimately prove even more important than teaching children how to program.

6 Conclusions

Embodiment in K-12 computer science education offers a powerful framework through three complementary approaches: virtual embodiment via PBP, physical embodiment through hands-on creation, and social embodiment through collaborative programming interfaces. Virtual embodiment through PBP represents the most significant contribution, addressing fundamental programming challenges by providing a visual proxy that maintains dual temporal representations—simultaneously displaying the current state of program objects and their anticipated future states based on planned actions. This syntonic embodiment of programmed objects enables students to visualize consequences before execution, allowing errors to be proactively identified and prevented rather than reactively debugged after they occur. The article provides crucial insights into how this embodiment mechanism achieves error prevention: by making abstract computational processes tangible and temporally explicit, students develop intuitive understanding of program behavior that transcends traditional syntax-focused approaches, fundamentally transforming how novice programmers conceptualize and interact with code. Physical embodiment enhances emotional engagement by allowing students to draw personally meaningful characters on paper before importing them digitally, creating stronger connections to programming tasks. Social embodiment transforms programming into a collaborative experience by making collaborators' actions and intentions visible in real-time through awareness interfaces. This tripartite approach creates a synergistic effect that makes programming more accessible, engaging, and collaborative, aligning with both Pestalozzi's "head, heart, and hands" educational philosophy and Vygotsky's theories of social learning, while addressing contemporary concerns about excessive screen time in education.

Acknowledgements

The opinions expressed in this work are those of the authors and do not necessarily reflect the views of the Hasler Foundation.

References

[1] S. T. Adams and A. A. DiSessa. 1991. Learning by "cheating": Students' inventive ways of using a boxer motion microworld. *Journal of Mathematical Behavior*, 10 (1991), 79–89.
[2] D. J. Barnes and M. Kölling. 2006. *Objects First with Java: A Practical Introduction Using BlueJ* (3rd ed.). Pearson Education/Prentice Hall.
[3] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak. 2017. Learnable programming: Blocks and beyond. *Communications of the ACM* 60 (2017), 72–80.
[4] H. Belmar. 2022. Review on the teaching of programming and computational thinking in the world. *Frontiers in Computer Science* 128 (2022), 997222.
[5] A. Ben-Yaacov and A. Hershkovitz. 2023. Types of errors in block programming: Driven by learner, learning environment. *Journal of Educational Computing Research* 61 (2023), 178–207.
[6] M. Bigman, E. Roy, J. Garcia, M. Suzara, K. Wang, and C. Piech. 2021. PearProgram: A more fruitful approach to pair programming. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 900–906.
[7] B. Broll, Á. Lédeczi, H. Zare, D. N. Do, J. Sallai, P. Völgyesi, M. Maróti, L. Brown, and C. Vanags. 2018. A visual programming environment for introducing distributed computing to secondary education. *Journal of Parallel and Distributed Computing* 118 (2018), 189–200.
[8] S. Campe, J. Denner, E. Green, and D. Torres. 2020. Pair programming in middle school: Variations in interactions and behaviors. *Computer Science Education* 30 (2020), 22–46.
[9] D. S. Carstens, S. K. Doss, and S. C. Kies. 2018. Social media impact on attention span. *Journal of Management & Engineering Integration* 11 (2018), 20.
[10] A. Cypher. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.
[11] N. Davidson and C. H. Major. 2014. Boundary crossings: Cooperative learning, collaborative learning, and problem-based learning. *Journal on Excellence in College Teaching* 25 (2014), 7–55.
[12] A. Deiner and G. Fraser. 2024. NuzzleBug: Debugging block-based programs in scratch. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–13.
[13] A. Deiner and G. Fraser. 2024. NuzzleBug: Debugging block-based programs in scratch. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*.
[14] J. Dewey. 1913. *Interest and Effort in Education*. Houghton Mifflin.
[15] E. Dutton, D. van der Linden, and R. Lynn. 2016. The negative Flynn effect: A systematic literature review. *Intelligence* 59 (2016), 163–169.
[16] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante. 2016. *Cyclomatic Complexity*, Vol. 33. IEEE software, 27–29.
[17] A. Eguiluz, M. Guenaga, P. Garaizar, and C. Olivares-Rodriguez. 2017. Exploring the progression of early programmers in a set of computational thinking challenges via clickstream analysis. *IEEE Transactions on Emerging Topics in Computing* 8 (2017), 256–261.
[18] M. Eisenstadt. 1997. My hairiest bug war stories. *Communications of the ACM* 40 (1997), 30–37.
[19] S. Federici, E. Gola, and E. Sergi. 2023. Is the scratch programming environment ideal for all? Enhancements to the scratch IDE to make it easier to use and more useful for students and teachers. In *Proceedings of the 15th International Conference on Computer Supported Education*, 171–181.
[20] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova. 2020. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 614–626.
[21] W. Finzer and L. Gould. 1984. Programming by rehearsal. *Byte* 9 (1984), 187–210.
[22] N. Fraser. 2015. Ten things we've learned from Blockly. In *Proceeding of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, 49–50.
[23] X. Ge, D. Ifenthaler, and J. M. Spector. 2015. *Emerging Technologies for STEAM Education: Full STEAM Ahead*. Springer.
[24] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning. 1995. LEGOsheets: A rule-based programming, simulation and manipulation environment for the LEGO programmable brick. In *Proceeding of the Visual Languages*, 172–179.
[25] E. P. Glinert and S. L. Tanimoto. 1984. *Pict: An Interactive Graphical Programming Environment*. IEEE Computer, pp. 265–283.
[26] L. A. Gouws, K. Bradshaw, and P. Wentworth. 2013. Computational thinking in educational activities: An evaluation of the educational game light-bot. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, 10–15.

[27] C. Gutwin and S. Greenberg. 2002. A descriptive framework of workspace awareness for real-time groupware. *Computer Supported Cooperative Work* 11 (2002), 411–446.
[28] F. Hermans and E. Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on scratch programs. in *Proceedings of the 2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 1–10.
[29] J. Hromkovič, D. Komm, R. Lacher, and J. Staub. 2020. Teaching with LOGO philosophy. *Encyclopedia of Education and Information Technologies*. Arthur Tatnall (Ed.), 1679–1690. DOI: https://doi.org/10.1007/978-3-030-10576-1_76
[30] J. Hromkovic and J. Staub. 2021. The problem with debugging in current block-based programming environments. *Bulletin of EATCS* 135 (2021). Retrieved from http://bulletin.eatcs.org/index.php/beatcs/article/view/667
[31] R. Huang, K. Ferdowsi, A. Selvaraj, A. G. Soosai Raj, and S. Lerner. 2022. Investigating the impact of using a live programming environment in a CS1 course. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*, Vol. 1, 495–501.
[32] W. Huitt and J. Hummel. 2003. Piaget's theory of cognitive development. *Educational Psychology Interactive* 3 (2003), 1–5.
[33] C. Infante, P. Hidalgo, M. Nussbaum, R. Alarcón, and A. Gottlieb. 2009. Multiple mice based collaborative one-to-one learning. *Computers & Education* 53 (2009), 393–401.
[34] A. Ioannidou, A. Repenning, and D. Webb. 2009. AgentCubes: Incremental 3D end-user development. *Journal of Visual Language and Computing* 20 (2009), 236–251.
[35] Y. Kafai. 2006. Playing and making games for learning: Instructionist and constructionist perspectives for game studies. *Games and Culture* 1 (2006), 36–40.
[36] K. Kahn. 2007. Should LOGO keep going forward 1? *Informatics in Education-An International Journal* 6 (2007), 307–321.
[37] M. Kapur. 2016. Examining productive failure, productive success, unproductive failure, and unproductive success in learning. *Educational Psychologist* 51 (2016), 289–299.
[38] M. Kleppmann, D. P. Mulligan, V. B. Gomes, and A. R. Beresford. 2021. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems* 33 (2021), 1711–1724.
[39] D. E. Knuth. 1989. The errors of TEX. *Software: Practice and Experience* 19 (1989), 607–685.
[40] H. Lieberman. 1987. An example-based environment for beginning programmers. In *Artificial Intelligence and Education*. R. W. Lawler and M. Yazdani (Eds.), Vol. 1. Ablex Publishing, Norwood, NJ, pp. 135–151.
[41] H. Lieberman. 1984. Steps toward better debugging tools for LISP. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 247–255.
[42] H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann Publishers, San Francisco, CA.
[43] S. I. A. Lubis. 2021. The role of school and teaching method through Maria Montessori and Johann Heinrich Pestalozzis view. *International Journal of Economic, Technology and Social Sciences (Injects)* 2 (2021), 87–92.
[44] C. Ludvigsson and V. Malmström. 2024. *Are Computers to Blame? Empirical Analysis of Swedish School Policy*. Master's Thesis.
[45] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18 (2008), 67–92.
[46] S. McDirmid. 2007. Living it up with a live programming language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications* (OOPSLA '07), 623–638.
[47] S. McDirmid. 2013. Usable live programming. In *Proceedings Of The 2013 ACM International Symposium On New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '13)*.
[48] A. Michotte. 1963. *The Perception of Causality*. Methuen & Co. Ltd., London.
[49] C. Morris. 1938. Foundations of the theory of signs. *Foundations of the Theory of Science*. Otto Neurath (Ed.), Vol. 1.
[50] C. Mouza, Y.-C. Pan, H. Yang, and L. Pollock. 2020. A multiyear investigation of student computational thinking concepts, practices, and perspectives in an after-school computing program. *Journal of Educational Computing Research* 58 (2020), 1029–1056.
[51] B. Nardi. 1993. *A Small Matter of Programming*. MIT Press, Cambridge, MA.
[52] J. R. Nawrocki, M. Jasiński, Ł. Olek, and B. Lange. 2005. Pair programming vs. side-by-side programming. In *Proceedings of the European Conference on Software Process Improvement*, 28–38.
[53] R. Niemczyk and S. Zawiślak. 2018. Review of maze solving algorithms for 2D maze and their visualisation. In *Proceedings of the International Conference of Students, PhD Students and Young Scientists*, 239–252.
[54] S. Papert. 1985. Hard Fun. Retrieved from http://www.papert.org/articles/HardFun.html
[55] S. Papert. 1993. The children's machine. *Technology Review-Manchester NH* 96 (1993), 28–28.
[56] S. Papert. 1987. Microworlds: Transforming education. In *Artificial Intelligence and Education, Vol. 1: Learning Environments and Tutoring Systems*. R. W. Lawler and M. Yazdani (Eds.), Ablex Publishing Corp., Norwood, NJ, 79–94.
[57] S. Papert. 1980. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, New York.

[58] R. Pelánek and T. Effenberger. 2022. Design and analysis of microworlds and puzzles for block-based programming. *Computer Science Education* 32 (2022), 66–104.
[59] H. Pestalozzi. 1932. Wie Gertrud ihre Kinder lehrt, ein Versuch, den Müttern Anleitung zu geben, ihre Kinder selbst zu unterrichten in Briefen von Heinrich Pestalozzi. 1801. In *Band 13 Schriften Aus Der Zeit Von 1799–1801*, De Gruyter, Berlin, Boston, pp. 181–359.
[60] A. Repenning. 2024. Escaping the Turing Tar-Pit with AI programming blocks. In *Proceedings of the 19th WiPSCE Conference on Primary and Secondary Computing Education Research*.
[61] A. Repenning. 2006. Excuse me, I need better AI! Employing collaborative diffusion to make game AI child's play. In *Proceedings of the ACM SIGGRAPH Symposium on Videogames*, 169–178.
[62] A. Repenning. 2025. From TikTok to hard fun: Progressive engagement in computational thinking through game design. In *Proceedings of the Constructionism Conference*, 227–238.
[63] A. Repenning. 2016. Transforming "hard and boring" into "accessible and exciting". In *Proceedings of the Workshop on Cultures of Participation in the Digital Age: From "Have to" to "Want to" Participate (NordiCHI '16)*.
[64] A. Repenning and J. Ambach. 1996. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In *Proceedings of the 1996 IEEE Symposium of Visual Languages*, 102–109.
[65] A. Repenning and A. Basawapatna. 2024. RULER: Prebugging with proxy-based programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 6.
[66] A. Repenning, D. C. Webb, C. Brand, F. Gluck, R. Grover, S. Miller, H. Nickerson, and M. Song. 2014. Beyond minecraft: Facilitating computational thinking through modeling and programming in 3D. *IEEE Computer Graphics and Applications* 34 (2014), 68–71.
[67] A. Repenning, D. C. Webb, K. H. Koh, H. Nickerson, S. B. Miller, C. Brand, I. H. M. Horses, A. Basawapatna, F. Gluck, R. Grover, et al. 2015. Scalable game design: A strategy to bring systemic computer science education to schools through game design and simulation creation. *Transactions on Computing Education* 15 (2015), 1–31.
[68] A. Repenning, A. R. Basawapatna, and N. A. Escherle. 2017. Principles of computational thinking tools. In *Emerging Research, Practice, and Policy on Computational Thinking. Educational Communications and Technology: Issues and Innovations*, H. C. Rich P (Ed.). Springer, Cham, 291–305.
[69] M. Resnick, J. Maloney, A. Monroy-Hernndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. 2009. Scratch: Programming for all. *Communications od the ACM* 52 (2009), 60–67.
[70] M. Saqr, K. Ng, S. S. Oyelere, and M. Tedre. 2021. People, ideas, milestones: A scientometric study of computational thinking. *ACM Transactions on Computing Education* 21 (2021), 1–17.
[71] T. Schümmer and S. Lukosch. 2009. Understanding tools and practices for distributed pair programming. *Journal of Universal Computer Science* 15 (2009), 3101–3125.
[72] M. J. Seckel, C. Salinas, V. Font, and G. Sala-Sebastia. 2023. Guidelines to develop computational thinking using the Bee-Bot robot from the literature. *Education and Information Technologies* 28 (2023), 16127–16151.
[73] R. B. Shapiro and M. Ahrens. 2016. Beyond blocks: Syntax and semantics. *Communications of the ACM* 59 (2016), 39–41.
[74] N. Shu. 1988. *Visual Programming*. Van Nostrand Reinhold Company, New York.
[75] C. Stokel-Walker. 2022. *TikTok Wants Longer Videos—Whether You Like It or Not*. WIRED.
[76] N. Strijbol, C. Scholliers, and P. Dawyndt. 2023. Blink: An educational software debugger for scratch. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education*, Vol. 2, 648.
[77] S. Tandel and A. Jamadar. 2018. Impact of progressive web apps on web app development. *International Journal of Innovative Research in Science, Engineering and Technology* 7 (2018), 9439–9444.
[78] M. Telles and Y. Hsieh. 2001. *The Science of Debugging*. Coriolis Group Books, Scottsdale AZ, USA
[79] J. Trower and J. Gray. 2015. Blockly language creation and applications: Visual programming for media computation and Bluetooth robotics control. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 5.
[80] J. Tsan, J. Vandenberg, Z. Zakaria, D. C. Boulden, C. Lynch, E. Wiebe, and K. E. Boyer. 2021. Collaborative dialogue and types of conflict: An analysis of pair programming interactions between upper elementary students. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 1184–1190.
[81] J. Tsan, J. Vandenberg, Z. Zakaria, J. B. Wiggins, A. R. Webber, A. Bradbury, C. Lynch, E. Wiebe, and K. E. Boyer. 2020. A comparison of two pair programming configurations for upper elementary students. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 346–352.
[82] J. Vandenberg, A. Rachmatullah, C. Lynch, K. E. Boyer, and E. Wiebe. 2021. The relationship of CS attitudes, perceptions of collaboration, and pair programming strategies on upper elementary students' CS learning. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education*, Vol. 1, 46–52.
[83] L. S. Vygotsky. 1986. *Thought and Language*, Revised edition. The MIT Press.
[84] J. Waite and S. Grover. 2020. Worked examples & other scaffolding strategies. In *Computer Science in K-12: An A to Z Handbook on Teaching Programming*. Shuchi Grover (Ed.), 240–249.

[85] S. Watt. 1998. Syntonicity and the psychology of programming. In *Proceedings of the 10th Annual Meeting of the Psychology of Programming Interest Group*, 75–86.
[86] M. Welsh. 2022. The end of programming. *Communications of the ACM* 66 (2022), 34–35.
[87] L. Williams and R. Kessler. 2003. *Pair Programming Illuminated*. Addison-Wesley Professional.
[88] R. L. Williams. (2013). Overview of the Flynn effect. *Intelligence* 41 (2013), 753–764.
[89] Z. Zakaria, J. Vandenberg, J. Tsan, D. C. Boulden, C. F. Lynch, K. E. Boyer, and E. N. Wiebe. 2021. Two-computer pair programming: Exploring a feedback intervention to improve collaborative talk in elementary students. *Computer Science Education*, 32 (2021), 1–28.

Received 9 October 2024; revised 6 November 2025; accepted 20 November 2025