


Multidimensional Framing of Environments Beyond Blocks and Texts in K–12 Programming

Ndui Okechukwu Ezeamuzie 
University of Hong Kong

Mercy Noyenim Ezeamuzie
International Christian School, Hong Kong

Computer programming provides a framework for interdisciplinary learning in sciences, arts and languages. However, increasing integration of programming in K–12 shows that the block-based and text-based dichotomy of programming environments does not reflect the spectrum of their affordance. Hence, educators are confronted with a fundamental hurdle of matching programming environments with learners’ cognitive abilities and learning objectives. This study addresses this challenge by analyzing 111 articles evaluating the affordances of programming environments to identify both structural and theoretical models to support educators’ choice of programming environments. The following dimensions of programming environments were identified: connectivity mode, interface natural language, language inheritance, age appropriateness, cost of environment, output interface, input interface, and project types. For each of these dimensions, the synthesis of the literature ranged from examining its nature and effect on learning programming to the implications of choosing an environment and the critical gaps that future studies should address. The findings offer instructors useful parameters to compare and assess programming environments’ suitability and alignment with learning objectives.

KEYWORDS FROM SCHOLARONE

Substantive: Computers and Learning, Engineering Education, Instructional Technologies, Interdisciplinary Teaching and Research, Learning Environments, Technology

Methodology: Qualitative Research, Content Analysis

Keywords

from Abstract: programming, computational thinking, K–12, coding, learning environment, Scratch, Python

I was recently exposed to a demonstration of what pretended to be educational software for an introductory programming course. With its “visualizations” on the screen, it was such an obvious case of curriculum infantilization that its author should be cited for “contempt of the student body.” (Dijkstra, 1989, p. 1403)

The above pejorative epigraph by Edsger Dijkstra, an Association of Computing Machinery Turing awardee, captures an interesting but fierce debate on teaching programming and computing (Denning, 1989). It would have been thought-provoking to revisit Dijkstra's position in the current landscape of computing education: Either Dijkstra's warnings were not heeded, or he underestimated his argument about "radical novelty" (Dijkstra, 1989, p. 1398), which arguably has made programming relatable to younger children. By radical novelty, Dijkstra contended that the enormous computing power of the computers in the era represented a massive leap that cannot be explained or understood by the conventional mindset, where gradual and seemingly imperceptible changes are the prevailing educational practices. Although Dijkstra's (1989) target seemed to be undergraduates, his dissatisfaction with certain pedagogical practices was obvious when he suggested imposing fines for use of anthropomorphic terms such as "bug" as substitute for "error" in programming classes, a rationale that contrasts with our present reality. For instance, present arguments postulate that the anthropomorphic characteristics of the programming environment, such as block-based and drag-and-drop features, create a low learning barrier and a high ceiling for programming achievement (Repenning et al., 2010; Resnick et al., 2009).

Since Dijkstra made this comment, programming has moved from an expert and domain-specific literacy to an interdisciplinary skill deemed useful for cross-domain applications. Although the extent of coverage and quality of programming skills demonstrated at different learning levels may be contentious, the benefits of learning programming across grade levels are abundant, including in early childhood (Bers et al., 2014; Kanaki & Kalogiannakis, 2018; Sullivan & Bers, 2019). These benefits include improvements in mathematical thinking, creativity, metacognition, confidence, language literacy, collaboration, and host of other 21st-century skills (Denner et al., 2019; Popat & Starkey, 2019; Scherer et al., 2019).

Attempts to teach programming in K–12 date back to the 20th century. Notably, Seymour Papert's seminal work on Logo programming, procedural thinking, and constructionism promoted a vision of children teaching computers to think through programming (Papert, 1980). Fast forward to the early 21st century; Wing (2006) called on computer science educators to extend the treasures of computing by teaching everyone "ways to think like a computer scientist" (p. 35), also referred to as computational thinking. Although computational thinking was explicitly and conceptually differentiated from programming in Wing's (2006) seminal call, the association between them remains strong. Most researchers have viewed programming activities as the underlying practice in learning computational thinking (Ezeamuzie & Leung, 2022). Furthermore, the socio-digital transformations that influence the ways we live, study, and work in the 21st century hinge on computer programs and reasonably require children to be conversant with their operation literacy.

The lucidly supported need for programming education in K–12 and the concomitant awareness in schools evoke concerns about how to support programming literacy effectively. In this regard, appropriate programming environments and pedagogies stand out as pillars that influence programming education. Several pedagogies have been highlighted in reviews (see Hsu et al., 2018; Lye & Koh, 2014; Scherer et al., 2020). Although noteworthy progress has been recorded in K–12 programming education in the 21st century, some of the associated practices were

once challenged. For example, Dijkstra (1989) strongly criticized the argument that a lack of appropriate programming platforms was responsible for software development challenges. Dijkstra's criticism is consistent with Soloway's (1986) assertion that the primary cog in the wheel of learning programming is the logical aspects of programming (e.g., decomposition, abstraction) that entail "putting the pieces together" (p. 850). However, several studies (e.g., Broll et al., 2018; Deng et al., 2020; Yildiz Durak, 2020) found that features of programming environments aid programming ability. This trajectory raises critical questions about the features and affordances of the programming environments that influence learning.

Often, programming environments are described as either block-based or text-based environments. In block-based environments, programmers drag and drop visual blocks of code in logical patterns. Text-based environments indicate platforms that require typing of textual codes with adherence to semantics and syntactical formats. To support educators who are increasingly tasked with supporting learners in acquiring programming and to increase our knowledge of the nature of programming environments beyond the dichotomy of block-based and text-based, this study systematically examines the literature on K–12 programming to unearth features of programming environments that influenced programming education in the 21st century. Gaining clarity on how the features have evolved will make the boundaries to programming more permeable and facilitate cross-disciplinary applications of programming literacy. Furthermore, recommendations of specific programming platforms, while contemporarily useful, fade in impact as programming environments evolve. Therefore, identifying the features that define environments rather than simply highlighting some environments as effective, avoids the problem posed by the transient nature of programming environments. This systematic review taps into the riches of prior research to inform such understanding by synthesizing and cross-checking the affordances of various programming environments.

Explicitly, the research question that guided this review is: Within the literature that evaluated the development, design, and application of programming environments in K–12, what dimensions informed the choice of programming environments? In this study, programming environment is loosely conceived as platforms, gadgets, tools, or applications that offer learners the capability to write program codes.

Background

In this section, we synthesize existing research by examining how programming environments have been framed. These include evaluating features of specific environments, comparison of two environments, cross-analysis of multiple environments and extant reviews on the influence of environments.

Insights From Evaluation and Comparison of Programming Environments

Studies that evaluated the effectiveness of specific programming environments are insightful for understanding the underlying factors in choosing a programming environment in K–12. For example, Messaging and Remote Procedure Call are the unique features of NetsBlox, a visual programming environment used in secondary schools (Broll et al., 2018). Similarly, studies that compared programming environments such as Yildiz Durak's (2020) comparison of the effects of Scratch and Alice, are equally resourceful in choosing programming

environments. In Deng et al. (2020), Visual Basic and PencilCode were compared for their influence on learning programming. Findings from these studies suggested that block-based environments eliminate syntax errors. Also, text-based environments support more versatility than the limited functionality of block-based environments. More so, PencilCode, a hybrid environment, bridged gaps between block-based and text-based environments by combining blocks and texts.

However, evaluations of specific environments are prone to developers' bias. Also, isolated comparisons of two programming environments, while useful, do not explain their relationships with the growing list of programming environments. The genre of studies (i.e., empirical studies evaluating and comparing environments) are briefly reviewed here because they constitute suitable data for the present systematic review, which seeks to determine how programming environments relate or differ from one another holistically. Therefore, studies that reported characteristics that facilitated or diminished the usage of specific programming environments were analyzed in this study.

Cross-Analyses of Multiple Programming Environments

In the early 21st century, Gómez-Albarrán (2005) selected 20 widely adopted programming environments and categorized each as a reduced-development environment, example-based environment, visualization/animation environment, or simulation environment. The reduced-development environments refer to environments that recognized the overheads of complicated platforms and were purposefully designed to mitigate the complexities by simplifying the interfaces. The example-based environment created extensive programming examples as scaffolded starting points for learners to solve new programming problems. Visualization/animation environments were deemed to facilitate programming by demonstrating the behavior of the codes visually. The simulation environment executes the program codes in an imaginary world. Gómez-Albarrán (2005) argued that reduced-development environments (e.g., BlueJ) are appropriate for novice programmers, whereas the other three categories of environments are provisioned for in-depth programming learning. However, these claims were not empirically validated, and most of the studied programming environments (except BlueJ and Alice) are rarely used in today's K–12 classrooms.

In a more recent study, Kraveva et al. (2019) focused on block-based programming environments. They analyzed the environments in four dimensions: usability and support, availability of learning resources, capabilities of the environment, and the closeness of features to a conventional programming language. They flagged Scratch and Code.org as the recommended environments. In another cross-analysis of 26 block-based environments by in-service computer science teachers, João et al. (2019) designated programming environments such as ScratchJr, Lego Bricks, and Bee-bot as appropriate for kindergarteners, whereas Scratch, Tinker IDE, m-Block, and Code Monkey were adjudged to be the most flexible.

However, programming environments are transient. It is important to understand how programming environments could be theoretically modeled. This understanding will inform the design or selection of environments that meet learners' learning objectives.

Lessons From Review Studies of Research Articles

The influence of programming environments was one of the concerns in a scoping review (Palumbo, 1990) that sought to unpack the relationship between programming and problem solving. Palumbo (1990) found that Logo, BASIC, and Pascal were the predominant languages for learning to program. Also, BASIC was reported as unstructured and the least effective in problem-solving. These environments, which were dominant in the 1990s, are highlighted here to demonstrate how much programming platforms have changed. Languages such as BASIC and Pascal are rarely mentioned in K–12 and distinctly contrast the ever-growing list of programming environments, including visual platforms.

As most visual programming environments are designed to mitigate the cognitive challenges of learning programming, Kuhail et al. (2021) were interested in understanding the usage trends of visual programming environments when they analyzed 30 articles that reported activities of end-users or novice programmers between 2010 and 2020. They categorized visual programming platforms into four categories according to the implementation technique: block-based, diagram-based, form-based, and icon-based. Through their inquiry, Kuhail et al. (2021) showed that block-based and diagram-based platforms were the predominant programming environment type. However, the underlying characteristics of the environments in facilitating learning programming are unknown as most of the articles did not report on their platforms' usability. The rarely used text-based environments (Palumbo, 1990) and a dearth of understanding concerning the underlying features of virtual programming environments (Kuhail et al., 2021) leaves questions regarding the suitability of environments, the rationale for choosing programming environments, and the platforms' adaptability to K–12.

Additional insights about programming environments were reported in the following tool-specific reviews. Zhang and Nouri (2019) sought to understand what students can learn in Scratch in their systematic review of 55 empirical studies on the use of Scratch in K–9 education. Results showed that Scratch supported the development of predictive thinking and human-computer interaction. Also, learners' development of computational thinking concepts, practices, and perspectives as framed in Brennan and Resnick's (2012) three-dimensional framework was validated in Scratch. However, Zhang and Nouri's (2019) systematic review did not explain how the features of Scratch enabled the acquisition of the above skills. Another platform-dependent study was reported in a meta-analysis that examined the effectiveness of Alice software in comparison with conventional programming languages (Costa & Miranda, 2017). Although Alice was determined to be marginally more effective than other conventional programming environments, this finding was based on six quasi-experimental studies and confounded by this statistical limitation.

Method

The Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) was adopted for this study. Originally designed for systematic reviews and meta-analyses in medical research (Moher et al., 2009), PRISMA offers invaluable standards for systematic reviews in other fields. Figure 1 summarizes the identification and screening procedures for eligible studies.

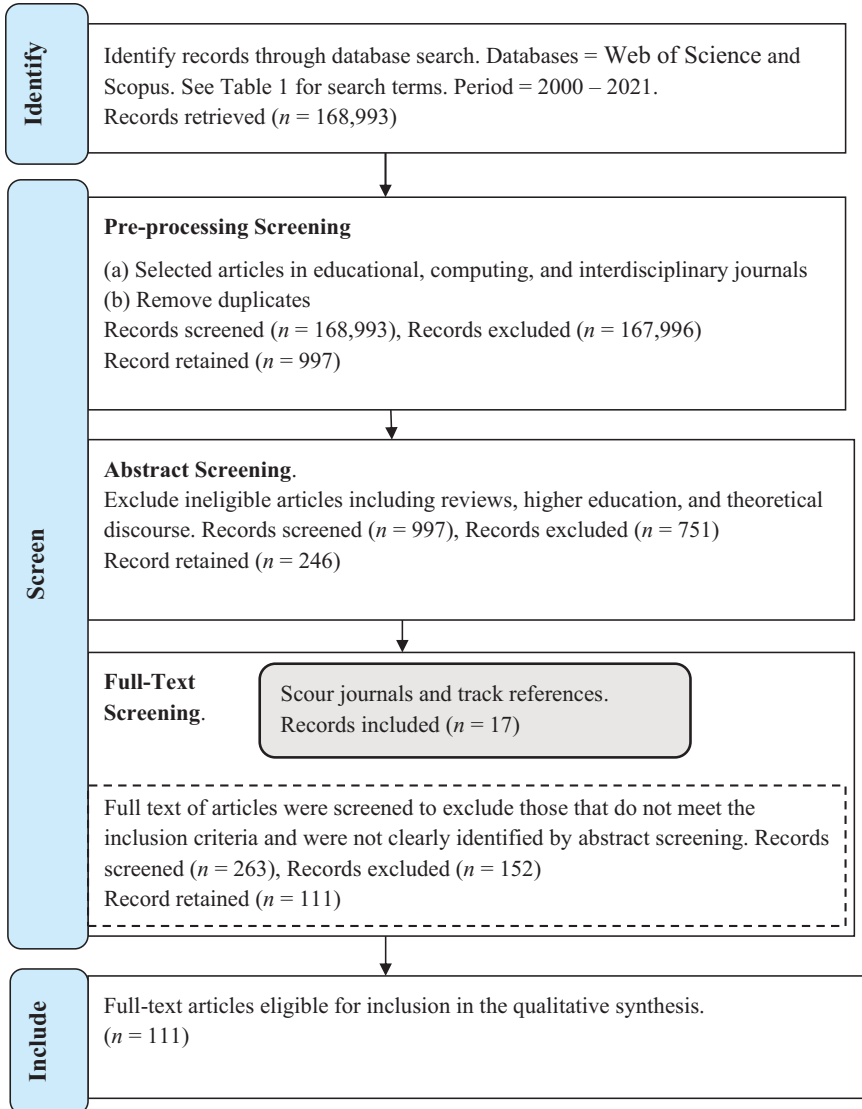


FIGURE 1. *Flow Diagram of Search Strategies and Screening for Eligible Studies for Review.*

Identification of Studies

Web of Science and Scopus were selected as databases because they index abstracts from multidisciplinary research with comprehensive coverage of high-quality journals and proceedings. To select search terms that are broad yet capable

of identifying high-quality studies, we adopted Luxton-Reilly and colleagues' (2018) approach and tested various search terms iteratively. Table 1 shows the selected optimal phrasal keywords for the database search. Although this review is focused on programming in K–12, we included computational thinking as part of the search term because of the evidence that many computational thinking studies in K–12 engaged students in programming activities (Ezeamuzie & Leung, 2022; Lye & Koh, 2014). The search was further restricted to English peer-reviewed articles published between 2000 and 2021. The identification stage yielded 168,993 documents (Web of Science $n = 96,681$; Scopus $n = 72,312$).

Screening and Eligibility of Documents

With the initial pool of documents identified, an empirical study with K–12 participants was selected if the article

1. evaluated the development or implementation of a programming environment, or
2. compared two or more programming environments, or
3. reported the effect of programming environments.

The above criteria formed the eligibility parameters for screening the documents and were implemented in three sequential stages: pre-processing, abstract screening, and full-text screening.

Pre-processing—using the database filters, we limited the pool to educational, computing, and interdisciplinary journals as learning programming was out of the scope of other categories. Duplicates from the independent databases were also excluded. The pre-processing stage resulted in the exclusion of 167,996 articles, leaving 997 articles for abstract screening. The reliability of pre-processing screening was tested by selecting 100 articles randomly from the initial pool of articles ($n = 168,993$). Based on the inclusion criteria, another member of the research team read the titles and abstracts of the articles, and coded to accept or reject the articles independently. The process resulted in 99% agreement with the pre-processing screening.

Abstract screening—100 articles were randomly selected from the remaining pool ($n = 997$). Two members of the research team read the 100 selected articles and classified them as accept or reject. Members were advised to accept an article when in doubt. Interrater reliability analysis showed 97% agreement between coders. For the three articles that were coded differently, the research team discussed the disagreement, reiterated the inclusion criteria, and reached a consensus. The abstracts of the remaining articles ($n = 897$) were screened by the first author. When in doubt, the first author assigned the article to the accepted pool for further review by the research team and full-text screening. The research team met for 2 hours every week to review the outcomes of the abstract screening, which lasted for 4 weeks. The abstract screening resulted in the exclusion of 751 articles, leaving 246 articles for full-text screening.

Full-text screening—Before the full-text screening of the remaining 246 articles, we scoured journals that contributed more than 5% to the pool. Also, we

TABLE 1*Optimal Phrasal Keywords for Database Search*

Database	Phrasal Keyword
Web of Science	TS = ([learn OR acquire OR develop OR teach OR assess OR instruct] AND "programming") OR TS = ([learn OR acquire OR develop OR teach OR assess OR instruct] AND "computational thinking")
Scopus	TITLE-ABS-KEY ([learn OR acquire OR develop OR teach OR assess OR instruct] AND "programming") OR TITLE-ABS-KEY ([learn OR acquire OR develop OR teach OR assess OR instruct] AND "computational thinking")

Note: TS and TITLE-ABS-KEY indicate that the search is limited to the Title, Abstract, and Keyword fields only.

searched for eligible articles from previous reviews and meta-analyses that investigated programming. Scherer and colleagues' (2019) meta-analysis on the cognitive gains of programming was particularly useful, pointing to 15 other valuable reviews. The reference backtracking and journal scouring culminated in the inclusion of 17 articles, leaving 263 articles for full-text screening. Two members of the research team screened the full text of the remaining articles. Differences in coding were discussed and mutually agreed upon. The screening phase concluded with the exclusion of 152 documents and the retention of 111 eligible studies for charting.

Most of the excluded articles were reviews/meta-analyses/theoretical discourse (e.g., Hsu et al., 2018; Hu et al., 2020; Macrides et al., 2022; Scherer et al., 2020), having participants from higher education (e.g., Kim et al., 2013; Wang & Hwang, 2017) and using teachers as study participants (e.g., Hadad et al., 2021; Kong et al., 2020).

Data Analysis

Prior works (e.g., Gómez-Albarrán, 2005; Krалеva et al., 2019; Kuhail et al., 2021) revealed how programming environments may be classified or recommended some specific but transient platforms. For example, Gómez-Albarrán (2005) categorized programming environments into the reduced-development environment, example-based environment, visualization/animation environment, or simulation environment. Scratch and Code.org were recommended in Krалеva and colleagues' (2019) analysis of programming environments' usability, availability, capabilities, and the closeness of their features to conventional language. However, the objective of this review is neither to create a classification nor recommend any programming environment. We aim to deconstruct the spectrum of features that influence the choice of programming environments from the researchers' perspective. Tying back to the research objective, we stepped back to assimilate how the data provided perspective on the research questions through a two-stage process: coding and charting.

First Stage: Coding

According to Saldaña (2016), a “code” is a word or short phrase that sums or captures the salient attribute of a portion of data and a “category” represents a progressive collection of related codes. Guided by Saldaña’s (2016) qualitative coding framework, two members of the research team summarized the 111 eligible articles to highlight the features of interest independently. For each article, the abstract and two summaries were collated as a case for further analysis in NVivo, a qualitative data analysis software (<https://www.qsrinternational.com/>).

For each case in NVivo, relevant codes related to the affordance of the programming environments were extracted. As coding progressed, related codes were mapped into categories iteratively. While the coding exercise was conveyed in an ordered and orderly manner, it entailed several iterations of reading and backtracking as needs arose for adding, deleting, and clarifying code definition. For example, Magerko et al. (2016) is a Nvivo case that described the rationale and evaluation of EarSketch, an environment that combined programming and music production. In the first round of coding, *music production* emerged as one of the codes from the case. Also, Rodríguez-Martínez et al. (2020) were interested in how Scratch supported learning mathematics in grade six. We added *mathematics* as another code. With the coding of other Nvivo cases yielding codes such as the *gameplay* in Autothinking (Hooshyar et al., 2021), further relationship analysis, using NVivo, showed that type of projects supported in an environment (codes: music, mathematics, gameplay) influenced researchers’ choices and design of programming environments. Hence, *project type* was identified as a category. The coding stage culminated in the identification of eight major categories/thematic dimensions of programming environments.

Second Stage: Charting

Charting involved reading the 111 articles (please see online supplementary material) and categorizing their data across eight thematic dimensions. This stage was convened to gain insight into how the selected studies have implemented the dimensions identified from the first stage (coding). Two researchers read and charted the articles independently. The research team met for 2 hours every week to discuss the outcomes of the charting, and mutually resolved any variation. The charting lasted for 6 weeks.

Findings and Discussion

Profile of Charted Studies

Table 2 shows the eight categories/thematic dimensions, sample codes, and sample data that support the codes.

- *Connectivity Mode* – reflects whether a programming environment requires Internet connections or not.
- *Interface Natural Language* – relates to how the linguistic design of environments supports learners in their locale or native local language.
- *Language Inheritance* – denotes whether learners need to master the syntax and semantics of a new programming language in an environment or

TABLE 2
Categories, Codes and Sample Data That Support the Codes

Category	Sample Code	Sample Supporting Data and Quotations
Connectivity	Online	NetsBlox – Builds on these concepts to supply primitives for synchronization and communication across computers, providing a gentle introduction to distributed computing (Broll et al., 2018, p. 191).
	Offline	KIBO – Children to scan the barcodes on the blocks and send the program to their robot instantaneously—no screen time from an iPad, computer, or other digital technology is required (Sullivan & Bers, 2019, p. 1038).
Interface Language	Online and Offline	IRobotQ3D – Virtual robot platform provides online and local modes for students to learn a robotics course (Zhong et al., 2023, p. 5).
	Multilingual	Scratch – Free, available in nearly 50 languages (Maloney et al., 2010, p. 2).
	Spanish	Alcody – Interface and dialog with Alcody is currently only available in Spanish (Morales-Urrutia et al., 2021, p. 6648).
Language Inheritance	Derived	CodeCombat – Supports Python and JavaScript and it can be used online (Kroustalli & Xinogalos, 202, p. 6073).
	Conventional	Java – Does the . . . performed better on the content assessment mean they will continue to perform better in a professional text-based language like Java? (Weintrop & Wilensky, 2017, p. 22).
Age and Grade	Kindergarten, Lower Elementary	ScratchJr – Describes the goals and challenges of creating a developmentally appropriate programming tool for children ages 5–7 (Flannery et al., 2013, p. 1).
	Lower Elementary, Upper Elementary	Torino – Physical programming language for teaching computational learning to children ages 7–11 regardless of level of vision (Morrison et al., 2021, p. 535).
Cost of Environment	Free	Scratch – Free, available in nearly 50 languages (Maloney et al., 2010, p. 2).
	Paid	Phogo – Combines Python, Arduino and 3D printing into a low cost robot that is easy to build and control (Molins-Ruano et al., 2018, p. 428).

(continued)

Table 2. (continued)

Category	Sample Code	Sample Supporting Data and Quotations
Input Interface	Icon-based	Bomberbot – To construct a program in Bomberbot, the necessary programming commands can be dragged from the library on the left and dropped into the main program . . . [The] library contains a selection of pictographical commands, such as forward, backward, turn left, turn right (Fanchamps et al., 2021, p. 6485).
	Hybrid	Flip – Has two main components: (1) a visual language, and (2) a dynamically updating natural language (Howland & Good, 2015, p. 224).
	Tangible	Talkoo – Comprises physical computing plug-and-play modules, a visual programming environment and prototyping material (Katterfeldt et al., 2018, p. 74).
Output Interface	3D	Kodu – Enables children and teenagers to create 3D games by offering . . . cartoonish objects and characters, and a set of manipulation tools to build the games’ landscape (Fokides, 2017, p. 482).
	Audio	Torino – To create code, children connect physical instruction pods and tune their parameter dials to create music, audio stories, or poetry (Morrison et al., 2021, p. 535).
	Deployable	AppInventor – A programming environment that allows everyone, even children, to build fully functional applications (apps) for smartphones and tablets (Seravidou & Douligeris, 2019, p. 2246).
Project Type	Game Design	RoboBuilder – The interface has two distinct components: a programming environment . . . where players define and implement their robot’s strategy; and an animated robot battleground . . . where players watch their robot compete (Weintrop & Wilensky, 2014, p. 309).
	Game Play	AutoThinking – Consists of three levels in which a player should, in the role of a mouse, develop different types of strategies and solutions to complete the levels, . . . at the same time escaping from two cats in the maze (Hooshyar et al., 2021, p. 389).
	Music	EarSketch – A learning environment that combines computer programming with sample-based music production to create a computational remixing environment (Magerko et al., 2016, p. 1)

not. Whether an environment was charted as *conventional* or *derived* depends on how the authors described the underpinning programming language of the environment. The traditional programming languages (e.g., Python, Java, JavaScript) are generally regarded as conventional. When the programming language of an environment is an extension of an existing language, it was charted as derived. For example, CodeCombat (Kroustalli & Xinogalos, 2021) was charted as derived because students wrote codes in Python—a conventional language.

- *Age and Grade Appropriateness* – shows the relationship between learners’ developmental stage and the suitability of the programming environments. The programming environments were classified into five categories based on the recommended users’ age: (a) kindergarten (less than 6 years of age), (b) lower elementary (6–9 years), (c) upper elementary (9–12 years), middle school (12–15 years), and high school (15–18 years).
- *Cost of Environment* – identifies the cost involved in using a programming environment, whether it requires a paid subscription or a free platform.
- *Input Interface* – describes approaches for keying in codes in a programming environment. Besides the block-based and text-based code entry, other formats reported in the literature include icon-based, hybrid, and tangible input interfaces. Icon-based environments are special type of block-based environment where pictorial symbols are used as commands (Kuhail et al., 2021). Hybrid environments provide both text-based and block-based coding input interfaces. Tangible environments represent a special approach to keying in codes with physical blocks; without using the computer mouse, keyboard, and screen.
- *Output Interface* – describes how program codes are rendered. Often, vocabularies such as block-based, drag-and-drop, visual and text-based refer to environments without distinguishing the nature of input (keying in codes) and output (rendering the codes). For example, in analyzing the learning outcomes and attitudes of primary school students in a visual environment, Sáez-López et al. (2016) described Scratch as a visual environment. Although the execution of the Scratch program renders as a visual output of a game and interactive agents, differentiating between the approach for keying in codes (input) and the forms of rendering the codes (output), emerged as an important dimension for describing programming environments. The formats of rendering output in programming environments include visual, 3D, deployable, audio, hardware control, virtual reality, and text.
- *Project Type* – provides insight into how diverse programming environments are provisioned to support different types of learning activities. Although not exhaustive, common types of projects include animation, digital stories, game design, gameplay, music composition, simulation, mathematics, mobile apps, and standard apps.

Table 3 summarizes the profile of the programming environments from the eligible studies ($n = 111$) in eight dimensions. These attributes and how they influence programming are examined in this section.

Missing data is an important attribute of the charted data in Table 3. As the name implies, it represents the number of studies from which the value of each dimension could not be extracted. This ranged from 24% of input interface ($n = 27$) to 93% of interface natural language ($n = 103$). The considerable proportion of dimensions that could not be determined is understandable, as the values in popular environments were often implied in the articles, but charting was strictly based on the data extracted from articles. For example, in an investigation of how the nature of a task influences programming, Erümit (2020) did not specify features of Scratch, such as whether it was provisioned to work online or offline. Although we knew that Scratch has both offline and online modes, the connectivity dimension of the article was coded as missing data as the author did not specify this feature. The same approach was adopted in coding the other dimensions. Other alternative explanations exist for the missing data. For instance, it might reflect dimensions of a programming environment that researchers have focused on or ignored. Also, brevity, journals' word count restrictions, and other communication purposes might have hindered authors from reporting certain details. The input and output interface attracted the most attention from prior work, and the interface natural language, grade level/age appropriateness, and programming language inheritance received the least attention.

Figure 2 is a simple word cloud visualization of the programming environments presented in the charted articles. The font size of a word is directly proportional to the observed frequency of the word. Scratch, Lego, Python, Kibo, and Alice are the most popular programming environments within the eligible studies. Note that the objective of this study was neither to compare environments nor to elevate certain environments as ideal platforms. As highlighted in the following thematic discussion, an ideal platform does not exist without context. Rather, the focus of this review was to synthesize the nature of programming environments as reported in peer-reviewed articles that embodied researchers' perspectives. The highlighted models will assist in making informed decisions about programming environments.

Connectivity

About 77% of the charted articles ($n = 86$) did not specify the connectivity mode of the discussed environment. We pondered whether researchers had perceived connectivity features as implied or the likelihood that a substantial number of the eligible studies might have been conducted in regions with good penetration of high-speed and affordable Internet connections, which invariably may have diminished concern for connectivity. Since the geographical distribution of the studies was not captured in the chart, no conclusive inference could be drawn. Moreover, mapping the regions by levels of internet access may be an oversimplification of connectivity issues. For instance, Katz et al. (2017) found that the quality of internet connection remains a serious issue in the United States.

TABLE 3
Summary of the Profiles of the Programming Environments

Study Characteristic	Value (n, %) ^a
Connectivity	Online (16, 14%), offline (5, 5%), both (4, 4%), missing data (86, 77%)
Interface Natural Language	Single non-English (3, 3%), multi-language: range 2–70 languages (5, 5%), missing data (103, 93%)
Programming Language Inheritance ^b	Derived (15, 14%), conventional (12, 11%), missing data (93, 84%)
Grade Level/Age ^b	Kindergarten: < 6 years (7, 6%), lower elementary: 6–9 years (15, 14%), upper elementary: 9–12 years (9, 8%), middle school: 12–15 years (9, 8%), high school: 15–18 years (6, 5%), missing data (95, 86%)
Subscription Mode ^b	Free (13, 12%), paid (33, 30%), missing data (75, 68%)
Output Interface ^b	Visual (37, 33%), 3D (11, 10%), deployable (5, 5%), virtual reality (1, 1%), audio (1, 1%), text (3, 3%), hardware control (21, 19%), missing data (42, 38%)
Input Interface ^b	Block-based (45, 41%), text-based (22, 20%), icon-based (5, 5%), hybrid (7, 6%), tangible (13, 12%), missing data (27, 24%)
Programming Project Type ^b	Animation (5, 5%), digital story (7, 6%), game design (15, 14%), game play (10, 9%), music composition (3, 3%), simulation (17, 15%), mathematics (1, 1%), mobile apps (1, 1%), standard apps (1, 1%), missing data (61, 55%)

Note. ^aPercentage (%) was rounded to a whole number and the sum of mutually exclusive items of a study characteristic may not round to 100. ^bValues for the study characteristics are not mutually exclusive.



FIGURE 2. *Word Cloud Representation of Programming Environments From Eligible Articles.*

Although the United States may stand in the league of technology advanced regions, in a national telephone survey of parents of school-aged children that reported less than the national median household income, 52% of participants described their internet connection as too slow for practical use (Katz et al., 2017). Apart from the studies that did not provide any information about connectivity mode, most of the studies that identified connectivity types did not highlight them as a feature of concern—whether as a promoter or inhibitor of learning programming. Four studies explicitly identified the programming environments adopted as both online and offline: Scratch (Erol and Çırak, 2022; Maloney et al., 2010), IRobotQ3D (Zhong et al., 2023), and mBlock (Matere et al., 2023).

Although the paucity of studies that highlighted connectivity seems to diminish the concern, using offline environments impedes programming in some cases. In regions without Internet service or in which such a service is costly, students are often confronted with additional challenges, such as instability in downloading or updating the programming environment, demanding configuration of the environment, and limited computing power/resources of some personal computers in solving computationally intensive tasks. Ezeamuzie (2023) highlighted some of the challenges of internet connection in an exploratory intervention in a

technology-deprived school. The choice of Python IDLE, an offline programming environment for middle school learners in Ezeamuzie (2023), was primarily a result of the connectivity constraint. However, these challenges could be mitigated in some online programming environments such as Google Colaboratoy—a browser-hosted Jupyter notebook service that requires no configuration, provides access to free cloud computing resources, and makes sharing of projects easy (Google, n.d.). How programming environments can enable social interaction in remote places with limited Internet connections remains unclear. In the lens of Spector’s (2005) educratic oath, which encapsulates educators’ obligation to design learning in ways that neither impair learning nor discriminate against learners, future research needs to investigate the impacts as well as formulate solutions for learning programming amongst learners with limited internet connectivity.

Interface Natural Language

A large proportion of the charted articles ($n = 103$; 93%) did not specify the natural language nor the number of languages supported in their programming environment. Considering that only English publications were reviewed, a plausible inference is that majority of the studied environments support, at minimum, learners who can read and write in English. Furthermore, conventional programming languages (e.g., Java, Python, JavaScript) are English dominated.

The importance of learners’ first or native language for K–12 learners’ programming ability cannot be over-emphasized. For example, Lau and Yuen (2011) found that Chinese students who were taught programming in their native language outperformed their peers who received instruction in English. The natural language was influential in the design of Alcody, an emotional-learning support system for programming in Spanish (Morales-Urrutia et al., 2021) and Let’s Code in Arabic (Almanie et al., 2019).

Although English assumes a central role in programming, educators may find that for students, especially in kindergarten and elementary school, situating a programming environment in a localized and relatable linguistic context may facilitate their learning. A preferable alternative is designing environments with multilingual support. For example, CodeCombat and Scratch support over 50 languages (Kroustalli & Xinogalos, 2021) and 70 languages (Erol & Çırak, 2022), respectively. Environments that support multiple languages (e.g., Scratch), especially free platforms, promote both wide and cross-cultural applications of the environment.

Using learners’ native language as the natural language in a programming environment seems promising as learners can focus on the intricacies of programming without linguistic barriers. While the influence of the interface natural language seems to be overlooked in charted studies, future studies can add to the knowledge of K–12 programming by investigating the effect of programming in various localized languages.

Programming Language Inheritance

Environments in the conventional category ($n = 12$; 11%) include Python (e.g., Efecan et al., 2021; Sentance et al., 2019), Java (Weintrop & Wilensky, 2017, 2019), ActionScript (Navarrete, 2013), C (Sun & Hsu, 2019), and Visual

Basic (Deng et al., 2020). Amongst the conventional category, Python was most dominant, appearing in seven studies. The conventional languages (e.g., Python, Java, or C) offer cross-platform reusability. For example, students programmed with Python in CodeCombat (Kroustalli & Xinogalos, 2021) and Java in Greenfoot (Kölling, 2016). Although conventional languages offer the flexibility of reuse in other environments, studies in the charted pool focused on the language per se, without emphasizing the environment. In the lens of cognitive load theory, the way information is presented on an interface constitutes the extraneous load (Sweller et al., 1998). Therefore, educators need to pay attention to the effect of the interfaces that support conventional languages, which may influence learning too.

Most of the programming environments ($n = 93$; 84%) did not identify any underpinning conventional programming language. In the absence of this information, one possible interpretation is that such environments require students to learn their syntax and semantics. On the contrary, and understandably, categorizing some environments as implementing distinct programming languages may be contentious. For example, block-based environments such as Scratch, App Inventor, and Webduino (Wu & Chen, 2022) are derived from the Google Blockly JavaScript Library (Blockly, n.d.). In any case, even when the environments derived their underpinning programming language from Blockly, they are independently and distinctly provisioned in their respective environments. Implicitly, they have different semantics from Blockly and may not be regarded as the same language.

For developers of programming environments, it is important to consider the aspect of programming language reusability. MaLT2, a 3D game design platform for creating dynamic objects, inherited Logo (Grizioti & Kynigos, 2021). Java is the underlying language in Greenfoot and BlueJ (Kölling, 2016). CodeCombat (Kroustalli & Xinogalos, 2021) inherited Python and JavaScript too. OpenSim with S4SL, a block-based environment, inherited Scratch (Pellas & Vosinakis, 2018). In the derived programming environments ($n = 15$, 14%), the syntax and semantics of the programming language of the environment are the same as those of the parent language. Hence, learners can migrate to other environments, especially when some environments support limited functionality.

Age and Grade Appropriateness

The awareness that learners' needs differ according to age necessitated the development of ScratchJr, a minified implementation of Scratch, to permit younger children to learn programming in developmentally appropriate ways (Flannery et al., 2013; Strawhacker et al., 2018). Other developmentally appropriate environments have embedded core principles in designing learning in early childhood education, including minimal screen exposure, tangible components, and extendibility to other crafts (Bers et al., 2013). Examples include KIWI (Sullivan & Bers, 2016), Bee-bot (Angeli & Valanides, 2019), and Kibo (Relkin et al., 2021; Sullivan & Bers, 2019), which are provisioned as tangible and screen-free platforms for kids.

Sullivan and Bers (2019) recommended Kibo for children between 4 and 7 years. Using Kibo, Relkin et al. (2021) found that Grades 1 and 2 students

(between 5 and 9 years old) developed sense of algorithm, modularity, and pattern representation. Although the age of participants in Relkin et al. (2021) did not differ significantly from Sullivan and Bers's (2019) recommendation for KIBO, learning may be impaired when much older learners use the environment. Developmentally appropriate environments are not restricted to finding suitable environments for kindergarteners and elementary school children only. It also demands ensuring that the choice of environments does not limit learning too. Limited vocabulary in some environments may hinder effective learning for older children. For instance, PhysGramming was designed with the theme of "object is everywhere" to teach object-oriented programming to children aged between 4 and 8 years (Kanaki & Kalogiannakis, 2018). Learning object-oriented programming is non-trivial even for expert programmers, and the notion that it could be taught in early childhood and lower elementary is interesting. However, close observation of the activities involved in teaching younger cohorts shows that PhysGramming teaches object-oriented programming through games of solving puzzles, matching objects, and grouping objects. The activities differ from object-oriented programming in conventional languages like Java, where learners deal with concepts such as inheritance, polymorphism, and encapsulation. Examining whether PhysGramming activities are representative of object-oriented programming is outside the scope of this study. However, the central point is that although PhysGramming may be a suitable environment for the designers' recommended age (i.e., between 4 and 8 years), repurposing the environment for older children may limit learning.

Most of the studies ($n = 95$; 86%) did not provide information about the age-appropriateness of the programming environments they examined. Except for Strawhacker and Bers's (2015) comparison of the influence of interface style (tangible, block-based, and hybrid) in learning programming among kindergarteners, the few studies that highlighted the age-appropriateness of their programming environments focused on the features of their specific environment only. Strawhacker and Bers (2015) found inconclusive evidence of any association between the nature of the interface and students' understanding of programming concepts. Future studies should consider comparing environments for their suitability to learners of different ages and levels in terms of both developmental appropriateness and limiting learning opportunities.

Cost of Environment

Only 12% of the articles ($n = 13$) identified their environments as free. Examples include LightBot (Yallihep & Kutlu, 2020), Scratch (Erol & Çırak, 2022; Iskrenovic-Momcilovic, 2019; Sáez-López et al., 2016), ScratchJr (Strawhacker et al., 2018), mBlock (Matere et al., 2023), and Blockly (Unal & Topu, 2021). Articles that were classified as paid environments ($n = 33$; 30%) represented studies that adopted hardware and robotic gadgets. Many of the reviewed articles ($n = 76$; 68%) did not disclose whether the programming environments were free or not. Probably, articles that adopted paid environments did not disclose the associated cost, which may be a less interesting feature. On the other hand, given that a free environment is a feature most studies would

highlight, the absence of such information in some articles may be attributed to the notion that their free usage is implied (e.g., Scratch).

The cost of programming environments may appear trivial. However, it becomes a significant dimension when inclusivity is the valued educational goal (Spector, 2005). In the early 21st century, when computers were considered a shared family gadget, the Raspberry Pi Foundation (n.d.) designed low-cost computers to increase young children's exposure to computing through personal ownership. Kölling (2016) narrated their efforts to complement Raspberry Pi's mission, describing the failure and subsequent success in porting Java-based Greenfoot and BlueJ into Raspberry Pi's standard distribution. From the reviewed articles, there is a substantial number of studies that adopted tangible gadgets or robots ($n = 33$; 30%). These hardware and robots come at a cost. Therefore, it is necessary to understand how the cost of programming environments may influence inclusivity in learning.

Although what constitutes affordable programming environments is subjective, among the studies that provided data on the cost of tangible gadgets, Arduino-based environments ranked the most affordable. In a study that compared the influence of solo and pair learning on students' robotic troubleshooting ability, Zhong and Li (2020) noted that Arduino was the least expensive from their analysis of robotic platforms. A similar claim was made to justify choosing Arduino for a primary-school, design-based learning experiment (Materer et al., 2023). Another Arduino-based environment is Phogo (Molins-Ruano et al., 2018), a reimplementaion of the successful Logo programming from a virtual turtle to the physical Tortoise robot that combines Python, an Arduino-like robot, and 3D printing at a low cost (US\$80). Besides, Arduino, Serrano Pérez and Juárez López (2019) reported other affordable tangible gadgets in their analysis of educational tools, including a (US\$25) ultra-low-cost robot. However, Arduino has the unique advantage of a free and open-source programming environment. Environments that build on the open-source ecosystem promote the development of affordable hardware, sensors, and robots that are interoperable (Zhong & Li, 2020).

Input Interface

Tangible environments ($n = 13$, 12%) support younger children and students with special needs to program without typing on the computer keyboard. According to Taylor (2018), the color-annotated Dash robot was chosen for early primary school programming because concrete manipulatives help young learners and students with special needs to learn more effectively. Using Kibo, young children (Relkin et al., 2021; Sullivan & Bers, 2019) and people with Down syndrome (González-González et al., 2019) were able to snap tangible blocks together to form programming instruction. When kids' exposure to screens is a concern, tangible environments mitigate the concern (Bers et al., 2014; Relkin et al., 2021; Sullivan & Bers, 2019).

Other examples of tangible environments include Torino for visually impaired children (Morrison et al., 2021) and Talkoo Kit (Katterfeldt et al., 2018). With Talkoo Kit, the programming process is inverted by de-emphasizing the use of computers and promoting collaborative design of physical circuitry that is

synchronously mapped to real-time virtual interaction. One major limitation of tangible blocks is that learners are restricted by the finite set of physical blocks. For this limitation, a workable solution in the literature was demonstrated in the design of Creative Hybrid Environment for Robotics Programming (CHERP). CHERP compensates for the limited number of physical blocks by creating a hybrid of block-based and tangible environments (Bers et al., 2014).

Icon-based environments ($n = 5$, 5%) are appropriate when the target users have limited reading/writing ability (Pasternak et al., 2017). Autothinking (Hooshyar et al., 2021), Microsoft Kodu (Fokides, 2017), EmpiricaControl (Lavonen et al., 2003) and Bomberbot (Fanchamps et al., 2021) were charted in the icon-based category. For example, in Bomberbot, learners programmatically control virtual robots through interlocking pictorial commands (e.g., *forward*, *jump*, *repeat*, *if/then*). Although the use of pictures as programming commands may be fascinating for younger children (e.g., kindergarteners), the meaning of pictorial commands may be difficult to interpret (Pasternak et al., 2017). Implicitly, icon-based environments have limited functionality and increasing the number of pictorial commands may not scale. Except for EmpiricaControl, the limited functionality constrained the icon-based environments to gameplay activities (Fanchamps et al., 2021; Fokides, 2017; Hooshyar et al., 2021).

Block-based environments ($n = 45$; 41%) were twice the number of studies that adopted text-based programming environments ($n = 22$; 20%). This substantial difference is consistent with the perception of block-based environments as easy programming environments. For instance, Okita (2014) compared the influence of using Lego NXT-G (block-based) or RobotC (text-based) as an introductory programming environment. In Lego NXT-G, “icon of a standing robot with an appended 5sec” represents a command for the robot to wait for 5 seconds. An equivalent command in RobotC is “wait1Msec(5000).” According to Okita (2014), Lego NXT-G has high transparency and supported students in creating mental connections between the Lego NXT-G blocks and the robot’s behavior easily. Similarly, most of the articles that compared programming environments, as illustrated above, are consistent with the anecdotal belief that text-based environments impose higher cognitive overheads for learners than block-based environments.

Hybrid environments include Learn Block (Bachiller-Burgos et al., 2020), Java Bridge Tool (Tóth & Lovászová, 2021), Flip (Howland & Good, 2015), and Pencil Code (Weintrop & Wilensky, 2017, 2019). Java Bridge Tool mediates transfer from App Inventor (block-based environment) to Java (text-based language) by creating a direct mapping in the same window, which extends the functionality of apps by linking App Inventor to full Java libraries. Flip supports the dual input modes and, interestingly, generates the natural language translation of the program as a third language synchronously. According to Howland and Good (2015), the humanistic factor of the natural language made Flip an effective programming environment. Generally, by combining the two input modes, hybrid environments attempt to mitigate the limited functionality of block-based environments without sacrificing their transparency and ease of use.

Output Interface

Excluding studies that did not identify the form of output interface ($n = 42$; 38%), a significant portion of the articles ($n = 37$; 33%) adopted programming environments that rendered outputs in the form of visuals, through activities such as games and interactive stories. Scratch (Gao & Hew, 2022), Autothinking (Hooshyar et al., 2021), CodeCombat (Kroustalli & Xinogalos, 2021), and Bomberbot (Fanchamps et al., 2021) provide but a few examples of visual rendering. In Bomberbot, when the visual operation of the programmable robot is different from the constructed codes, the environment provides tailored feedback quickly.

3D output interface is a special form of visual rendering. Unlike visual rendering in two-dimensional space, 3D enhances visualization by creating naturalistic engagement. According to Félix et al. (2020), the essence of 3D output lies in creating an immersive experience for both the game design and the gameplay. Microsoft Kodu, a 3D game-creation environment with cartoonish objects and characters, was designed to promote engagement (Fokides, 2017). Other forms of 3D engagement include the gamification and emotion-recognition add-on features in EasyLogic3D (Félix et al., 2020). High school students who used OpenSim with S4SL, a 3D visual platform, significantly improved their problem-solving and algorithmic design in comparison with peers who programmed in Scratch, a two-dimensional visual environment (Pellas & Vosinakis, 2018).

Text output interface renders the results of the programs as strings on the console (sometimes referred to as shell or terminal). Programming environments with outputs in this category often have commands for accepting and displaying the textual output and are predominantly conventional programming languages. For example, Java's textual output was harnessed in comparing the influence of block-based and text-based languages (Weintrop & Wilensky, 2017) and transitioning from block-based to text-based programming (Weintrop & Wilensky, 2019). Nonetheless, most conventional languages, such as Java and Python, also have libraries for creating rich graphical and visual outputs.

Hardware control requires physical/tangible objects including robots, toys, or electronic boards to enact their program. Examples include Arduino (Zhong & Li, 2020), Phogo (Molins-Ruano et al., 2018), and Lego (Okita, 2014). Audio and virtual reality output interfaces, although less popular, are insightful approaches to performing the codes in a computer program. For instance, Torino, a tangible and tactile-enabled block that outputs sound, was a suitable output interface for learners with mixed visual abilities (Morrison et al., 2021). With VR-OCKS, for example, students are immersed in a virtual, 3D environment and program by organizing floating action blocks as code to solve puzzles (Segura et al., 2020).

Deployable programming environments create distributable packages that can be deployed on other platforms. Although this overlaps with other output formats, such as visuals and 3D, the ability to deploy programs across platforms serves as extra motivation for learners. For example, Seralidou and Douligeris (2019) found that students accepted App Inventor, a block-based input programming environment for designing android apps, which can be deployed to reach wider audience on mobile phones and tablets seamlessly. Most conventional languages offer similar

support. Since deployable environments support learners to design programs that engender social interactions and are commercializable, future research may investigate whether such motivation could influence both learning and disposition toward programming.

Programming Project Type

Game design environments ($n = 15$, 14%) provide platforms for learners to both create and play games. This category is aptly captured in Weintrop and Wilensky's (2014) description of RoboBuilder and similar environments as "program-to-play" platforms. Other environments that support game design activities include Scratch (Maloney et al., 2010), Microsoft Kodu (Fokides, 2017), AgentSheets and AgentCubes (Leonard et al., 2016), Alice (Hartl et al., 2015), and NetsBlox (Broll et al., 2018). These environments are designed to support parallel execution of codes to create synchronous effects of the game elements. In NetsBlox, parallelism is implemented across distributed computers.

Gameplay environments ($n = 10$, 9%) support students to code solutions for predetermined problems in the form of playing games. Studies showed that students' programming achievement (Yallihep & Kutlu, 2020) and learning attitude (Hooshyar et al., 2021) improved in gameplay environments. However, gameplay environments such as CodeCombat (Kroustalli & Xinogalos, 2021), LightBot (Yallihep & Kutlu, 2020), and Autothinking (Hooshyar et al., 2021) do not offer the flexibility of game creation. Plausible impacts of this limitation may include boredom when successive game levels are repetitive, less motivation when access to higher levels is hard, and limited exposure to advanced programming concepts. Therefore, more research is needed to understand the impact of gameplay environment as an introductory programming environment, as well as the long-term impact on programming.

Animation and digital story creation represent other genres of projects that may be well fitted in certain environments, such as Scratch (Maloney et al., 2010) and Alice (Denner et al., 2014). The meaning of animation varies both within and between environments. In Scratch, it ranges from simple codes that render animated effects of a sequence of images and extends to complex codes that tell stories, simulate science projects, and create tutorials (Maloney et al., 2010). Animation and simulations are sometimes used interchangeably to describe the nature of activities that are supported in programming environments. In this review, simulation refers to activities that use tangible objects such as Dash robotic path tracing by students with intellectual disabilities (Taylor, 2018), and controlling LEDs, sensors, and buttons with Arduino (Zhong & Li, 2020).

Music composition and Mathematical representation are other types of projects that programming environments may specifically support. For instance, Taylor and Baek (2019) selected the Lego Mindstorms EV3 for its affordance in creating musical notes and tones. EarSketch was designed for computational music remixing with Python (Magerko et al., 2016). In Mathematical representation of word problems, Rodríguez-Martínez et al. (2020) explored the effect of learning to represent the least common multiple (LCM) and greatest common divisor (GCD) as part of word problems in Scratch. More studies are needed to demonstrate how features of programming environments augment learning of disciplinary topics across sciences and arts.

Since environments may support different types of programming activities (e.g., game design, animation, modeling scientific concepts), the affordance of an environment should be clear and aligned with the programming tasks. For example, Erümit (2020) investigated students' programming experience in Scratch when engaged in three different activities: game design, arithmetic, and animation creation. Findings suggest that game design activities expose students to richer experiences than arithmetic and animation. This shows that the intrinsic nature of programming tasks may engender different experiences in an environment.

Sometimes, the nature of tasks that are supported in some environments may not be well-defined. For instance, Seralidou and Douligeris (2019) described App Inventor as a platform for building apps for smartphones and tablets. However, what constitutes an app may be unclear and hides certain affordances of the platform, including support for designing games, animation, interactive user interfaces, and databases. Also, mapping environments to certain activities may not reveal the complexities of projects. Maloney et al. (2010) described Scratch as a platform for creating interactive and media-rich projects, encompassing activities such as games, simulations, science projects, animated stories, and music/video projects. However, the creation of music in Scratch involves playing a recorded sound. This approach differs from that of environments such as EarSketch, in which music composition is implemented through computational remixing (Magerko et al., 2016).

Limitations and Future Research

Although the significant role played by programming environments in learning has been established, the paucity of understanding about their conceptual framing is clear in the literature. To make programming education more permeable to the growing number of educators who are tasked with teaching programming in K–12, this study systematically synthesized the features of programming environments from existing studies, reported practices related to eight dimensions that could influence the choice of learning platforms, and highlighted clear gaps and inconsistencies that should be addressed by future authors.

To describe the contributions of this study to knowledge and practices, it is important to acknowledge its limitations. First, with the systematic approach of this review, which involved a keyword search and referential backtracking, it is not possible to claim total coverage. In addition, some eligible documents might have been omitted in the screening process, especially in the abstract screening by a coder. Focusing on K–12 studies inevitably excluded studies in higher education. Although the decision to impose this limitation was guided by the goal of understanding the unique context of the K–12 experience, insightful studies about programming environments in higher education may have been omitted. Moreover, analyzing only peer-reviewed articles—although it was deemed necessary to mitigate the varied quality of grey literature—excluded some insightful pieces.

Despite these limitations, this study unearths a valuable framework that conceptualizes programming environments in K–12 and how the dimensions of such environments may influence learning. The dimensions explored in this study relate to the environments' connectivity mode, the natural language of the interface, language inheritance, age/grade level appropriateness, influence of cost and subscription, output interface, input interface, and compatibility with various project types.

Given the possible bias that researchers' theoretical perspectives may induce, it is important to highlight key issues of validity. Certain specific programming environments were described as illustrations of the dimensions uncovered to make these features more relatable to readers. The authors of this article have no affiliation with any of the developers and have no intention of promoting any specific platform. Future work could verify the alignment between dimensions that emerged from the data and the actual programming environments, and examine how the choice of environments based on the dimensions could affect students' programming learning experience. Developers of programming environments need to elaborate on how the environment can support learning in these dimensions. More so, empirical investigations are required to verify how the design choices of programming environments support the dimensions, such as age-appropriateness.

The aim of this investigation was neither to compare nor evaluate the relative merits of specific environments. Although this study does provide dimensions to be used by educators in comparing the affordances of programming environments, its main goal was to raise awareness of the need for educators to carefully consider how a given programming environment aligns with their students' learning. This may lead to a sort of "chicken and egg" problem of choosing where to start between teaching objectives and programming platforms. Basing a decision on the learning objectives is a valid path from a learning and instructional design perspective. However, educators must be prepared to acknowledge instances where realizing the learning objectives within a suitable environment may be overly complex for the target users, causing more problems than it solves.

Acknowledgments

We thank Jessica S.C. Leung for her invaluable feedback on the research design and draft.

ORCID iD

Ndudi Okechukwu Ezeamuzie  <https://orcid.org/0000-0001-8946-5709>

References

- Almanie, T., Alqahtani, S., Almuhanha, A., Almokali, S., Guediri, S., & Alsofayan, R. (2019). Let's Code: A kid-friendly interactive application designed to teach arabic-speaking children text-based programming. *International Journal of Advanced Computer Science & Applications*, *10*(7), 413–418. <https://doi.org/10.14569/IJACSA.2019.0100757>
- Angeli, C., & Valanides, N. (2019). Developing young children's computational thinking with educational robotics: An interaction effect between gender and scaffolding strategy. *Computers in Human Behavior*, *105*, Article 105954. <https://doi.org/10.1016/j.chb.2019.03.018>
- Bachiller-Burgos, P., Barbecho, I., Calderita, L. V., Bustos, P., & Manso, L. J. (2020). LearnBlock: A robot-agnostic educational programming tool. *IEEE Access*, *8*, 30012–30026. <https://doi.org/10.1109/ACCESS.2020.2972410>
- Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education*, *72*(C), 145–157. <https://doi.org/10.1016/j.compedu.2013.10.020>

- Bers, M., Seddighin, S., & Sullivan, A. (2013). Ready for robotics: Bringing together the T and E of STEM in early childhood teacher education. *Journal of Technology and Teacher Education*, 21(3), 355–377. <https://www.learntechlib.org/primary/p/41987/>
- Blockly. (n.d.). *A JavaScript library for building visual programming editors*. <https://developers.google.com/blockly>
- Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association* (Vol. 1, pp. 1–25). <http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf>
- Broll, B., Lédeczi, Á., Zare, H., Do, D. N., Sallai, J., Völgyesi, P., Maróti, M., Brown, L., & Vanags, C. (2018). A visual programming environment for introducing distributed computing to secondary education. *Journal of Parallel and Distributed Computing*, 118, 189–200. <https://doi.org/10.1016/j.jpdc.2018.02.021>
- Costa, J. M., & Miranda, G. L. (2017). Relation between Alice software and programming learning: A systematic review of the literature and meta-analysis. *British Journal of Educational Technology*, 48(6), 1464–1474. <https://doi.org/10.1111/bjet.12496>
- Deng, W., Pi, Z., Lei, W., Zhou, Q., & Zhang, W. (2020). Pencil Code improves learners' computational thinking and computer learning attitude. *Computer Applications in Engineering Education*, 28(1), 90–104. <https://doi.org/10.1002/cae.22177>
- Denner, J., Campe, S., & Werner, L. (2019). Does computer game design and programming benefit children? A meta-synthesis of research. *ACM Transactions on Computing Education*, 19(3), 1–35. <https://doi.org/10.1145/3277565>
- Denner, J., Werner, L., Campe, S., & Ortiz, E. (2014). Pair programming: Under what conditions is it advantageous for middle school students? *Journal of Research on Technology in Education*, 46(3), 277–296. <https://doi.org/10.1080/15391523.2014.888272>
- Denning, P. J. (1989). A debate on teaching computing science. *Communications of the ACM*, 32(12), 1397–1414. <https://doi.org/10.1145/76380.76381>
- Dijkstra, E. W. (1989). On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12), 1398–1404. <https://doi.org/10.1145/76380.76381>
- Efecan, C. F., Sendag, S., & Gedik, N. (2021). Pioneers on the case for promoting motivation to teach text-based programming. *Journal of Educational Computing Research*, 59(3), 453–469. <https://doi.org/10.1177/0735633120966048>
- Erol, O., & Çırak, N. S. (2022). The effect of a programming tool scratch on the problem-solving skills of middle school students. *Education and Information Technologies*, 27(3), 4065–4086. <https://doi.org/10.1007/s10639-021-10776-w>
- Erümit, A. K. (2020). Effects of different teaching approaches on programming skills. *Education and Information Technologies*, 25(2), 1013–1037. <https://doi.org/10.1007/s10639-019-10010-8>
- Ezeamuzie, N. O. (2023). Project-first approach to programming in K–12: Tracking the development of novice programmers in technology-deprived environments. *Education and Information Technologies*, 28(1), 407–437. <https://doi.org/https://doi.org/10.1007/s10639-022-11180-8>
- Ezeamuzie, N. O., & Leung, J. S. C. (2022). Computational thinking through an empirical lens: A systematic review of literature. *Journal of Educational Computing Research*, 60(2), 481–511. <https://doi.org/10.1177/07356331211033158>
- Fanchamps, N. L. J.A., Slangen, L., Specht, M., & Hennissen, P. (2021). The impact of SRA-programming on computational thinking in a visual oriented programming

- environment. *Education and Information Technologies*, 26(5), 6479–6498. <https://doi.org/10.1007/s10639-021-10578-0>
- Félix, J. M.R., Zatarain Cabada, R., & Barrón Estrada, M. L. (2020). Teaching computational thinking in Mexico: A case study in a public elementary school. *Education and Information Technologies*, 25(6), 5087–5101. <https://doi.org/10.1007/s10639-020-10213-4>
- Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). Designing ScratchJr: support for early childhood learning through computer programming. In J. P. Hourcade, N. Sawhney, & E. Reardon (Eds.), *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 1–10). Association for Computing Machinery. <https://doi.org/10.1145/2485760.2485785>
- Fokides, E. (2017). Students learning to program by developing games: Results of a year-long project in primary school settings. *Journal of Information Technology Education: Research*, 16, 475–505. <https://doi.org/10.28945/3893>
- Gao, X., & Hew, K. F. (2022). Toward a 5E-based flipped classroom model for teaching computational thinking in elementary school: Effects on student computational thinking and problem-solving performance. *Journal of Educational Computing Research*, 60(2), 512–543. <https://doi.org/10.1177/07356331211037757>
- Gómez-Albarrán, M. (2005). The teaching and learning of programming: A survey of supporting software tools. *Computer Journal*, 48(2), 130–144. <https://doi.org/10.1093/comjnl/bxh080>
- González-González, C. S., Herrera-González, E., Moreno-Ruiz, L., Reyes-Alonso, N., Hernández-Morales, S., Guzmán-Franco, M. D., & Infante-Moro, A. (2019). Computational thinking and Down syndrome: An exploratory study using the Kibo robot. *Informatics*, 6(2), 25. <https://doi.org/10.3390/informatics6020025>
- Google. (n.d.). *Welcome to Colaboratory*. <https://colab.research.google.com>
- Grizioti, M., & Kynigos, C. (2021). Code the mime: A 3D programmable charades game for computational thinking in MaLT2. *British Journal of Educational Technology*, 52(3), 1004–1023. <https://doi.org/10.1111/bjet.13085>
- Hadad, S., Shamir-Inbal, T., Blau, I., & Leykin, E. (2021). Professional development of code and robotics teachers through small private online course (SPOC): Teacher centrality and pedagogical strategies for developing computational thinking of students. *Journal of Educational Computing Research*, 59(4), 763–791. <https://doi.org/10.1177/0735633120973432>
- Hartl, A. C., DeLay, D., Laursen, B., Denner, J., Werner, L., Campe, S., & Ortiz, E. (2015). Dyadic instruction for middle school students: Liking promotes learning. *Learning and Individual Differences*, 44, 33–39. <https://doi.org/10.1016/j.lindif.2015.11.002>
- Hooshyar, D., Pedaste, M., Yang, Y., Malva, L., Hwang, G.-J., Wang, M., Lim, H., & Delev, D. (2021). From gaming to computational thinking: An adaptive educational computer game-based learning approach. *Journal of Educational Computing Research*, 59(3), 383–409. <https://doi.org/10.1177/0735633120965919>
- Howland, K., & Good, J. (2015). Learning to communicate computationally with Flip: A bi-modal programming language for game creation. *Computers & Education*, 80, 224–240. <https://doi.org/10.1016/j.compedu.2014.08.014>
- Hsu, T.-C., Chang, S.-C., & Hung, Y.-T. (2018). How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education*, 126, 296–310. <https://doi.org/10.1016/j.compedu.2018.07.004>

- Hu, Y., Chen, C.-H., & Su, C.-Y. (2020). Exploring the effectiveness and moderators of block-based visual programming on student learning: A meta-analysis. *Journal of Educational Computing Research*, 58(8), 1467–1493. <https://doi.org/10.1177/0735633120945935>
- Iskrenovic-Momcilovic, O. (2019). Pair programming with scratch. *Education and Information Technologies*, 24(5), 2943–2952. <https://doi.org/10.1007/s10639-019-09905-3>
- João, P., Nuno, D., Fábio, S. F., & Ana, P. (2019). A cross-analysis of block-based and visual programming apps with computer science student-teachers. *Education Sciences*, 9(3), 181. <https://doi.org/10.3390/educsci9030181>
- Kanaki, K., & Kalogiannakis, M. (2018). Introducing fundamental object-oriented programming concepts in preschool education within the context of physical science courses. *Education and Information Technologies*, 23(6), 2673–2698. <https://doi.org/10.1007/s10639-018-9736-0>
- Katterfeldt, E.-S., Cukurova, M., Spikol, D., & Cuartielles, D. (2018). Physical computing with plug-and-play toolkits: Key recommendations for collaborative learning implementations. *International Journal of Child–Computer Interaction*, 17, 72–82. <https://doi.org/10.1016/j.ijcci.2018.03.002>
- Katz, V. S., Gonzalez, C., & Clark, K. (2017). Digital inequality and developmental trajectories of low-income, immigrant, and minority children. *Pediatrics*, 140(Suppl. 2), S132–S136. <https://doi.org/10.1542/peds.2016-1758R>
- Kim, B., Kim, T., & Kim, J. (2013). Paper-and-pencil programming strategy toward computational thinking for non-majors: Design your solution. *Journal of Educational Computing Research*, 49(4), 437–459. <https://doi.org/10.2190/EC.49.4.b>
- Kölling, M. (2016). Educational programming on the Raspberry Pi. *Electronics*, 5(3), Article 33. <https://doi.org/10.3390/electronics5030033>
- Kong, S.-C., Lai, M., & Sun, D. (2020). Teacher development in computational thinking: Design and learning outcomes of programming concepts, practices and pedagogy. *Computers & Education*, 151, Article 103872. <https://doi.org/10.1016/j.compedu.2020.103872>
- Kraleva, R., Kraleva, V., & Kostadinova, D. (2019). A methodology for the analysis of block-based programming languages appropriate for children. *Journal of Computing Science and Engineering*, 13(1), 1–10. <https://doi.org/10.5626/JCSE.2019.13.1.1>
- Kroustalli, C., & Xinogalos, S. (2021). Studying the effects of teaching programming to lower secondary school students with a serious game: A case study with Python and CodeCombat. *Education and Information Technologies*, 26(5), 6069–6095. <https://doi.org/10.1007/s10639-021-10596-y>
- Kuhail, M. A., Farooq, S., Hammad, R., & Bahja, M. (2021). Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*, 9, 14181–14202. <https://doi.org/10.1109/ACCESS.2021.3051043>
- Lau, W. W. F., & Yuen, A. H. K. (2011). The impact of the medium of instruction: The case of teaching and learning of computer programming. *Education and Information Technologies*, 16(2), 183–201. <https://doi.org/10.1007/s10639-009-9118-8>
- Lavonen, J. M., Meisalo, V. P., Lattu, M., & Sutinen, E. (2003). Concretising the programming task: A case study in a secondary school. *Computers & Education*, 40(2), 115–135. [https://doi.org/10.1016/S0360-1315\(02\)00101-X](https://doi.org/10.1016/S0360-1315(02)00101-X)
- Leonard, J., Buss, A., Gamboa, R., Mitchell, M., Fashola, O., Hubert, T., & Almughyirah, S. (2016). Using robotics and game design to enhance children’s self-efficacy,

- STEM attitudes, and computational thinking skills. *Journal of Science Education and Technology*, 25(6), 860–876. <https://doi.org/10.1007/s10956-016-9628-2>
- Liu, A. S., Schunn, C. D., Flot, J., & Shoop, R. (2013). The role of physicality in rich programming environments. *Computer Science Education*, 23(4), 315–331. <https://doi.org/10.1080/08993408.2013.847165>
- Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., & Szabo, C. (2018). Introductory programming: A systematic literature review. In G. Rößling & B. Scharlau (Eds.), *23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 55–106). ACM. <https://doi.org/10.1145/3293881.3295779>
- Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K–12? *Computers in Human Behavior*, 41, 51–61. <https://doi.org/10.1016/j.chb.2014.09.012>
- Macrides, E., Miliou, O., & Angeli, C. (2022). Programming in early childhood education: A systematic review. *International Journal of Child–Computer Interaction*, 32, Article 100396. <https://doi.org/10.1016/j.ijcci.2021.100396>
- Magerko, B., Freeman, J., Mcklin, T., Reilly, M., Livingston, E., Mccoid, S., & Crews-Brown, A. (2016). EarSketch: A steam-based approach for underrepresented populations in high school computer science education. *ACM Transactions on Computing Education*, 16(4), Article 14. <https://doi.org/10.1145/2886418>
- Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4), Article 16. <https://doi.org/10.1145/1868358.1868363>
- Matere, I. M., Weng, C., Astatke, M., Hsia, C.-H., & Fan, C.-G. (2023). Effect of design-based learning on elementary students’ computational thinking skills in visual programming maker course. *Interactive Learning Environments*, 31(6), 3633–3646. <https://doi.org/10.1080/10494820.2021.1938612>
- Molins-Ruano, P., Gonzalez-Sacristan, C., & Garcia-Saura, C. (2018). Phogo: A low cost, free and “maker” revisit to Logo. *Computers in Human Behavior*, 80, 428–440. <https://doi.org/10.1016/j.chb.2017.09.029>
- Morales-Urrutia, E. K., Ocaña, J. M., Pérez-Marín, D., & Pizarro, C. (2021). Can mindfulness help primary education students to learn how to program with an emotional learning companion? *IEEE Access*, 9, 6642–6660. <https://doi.org/10.1109/ACCESS.2021.3049187>
- Morrison, C., Villar, N., Hadwen-Bennett, A., Regan, T., Cletheroe, D., Thieme, A., & Sentance, S. (2021). Physical programming for blind and low vision children at scale. *Human–Computer Interaction*, 36(5–6), 535–569. <https://doi.org/10.1080/07370024.2019.1621175>
- Navarrete, C. C. (2013). Creative thinking in digital game design and development: A case study. *Computers & Education*, 69, 320–331. <https://doi.org/10.1016/j.compedu.2013.07.025>
- Okita, S. Y. (2014). The relative merits of transparency: Investigating situations that support the use of robotics in developing student learning adaptability across virtual and physical computing platforms. *British Journal of Educational Technology*, 45(5), 844–862. <https://doi.org/10.1111/bjet.12101>
- Palumbo, D. B. (1990). Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1), 65–89. <https://doi.org/10.3102/00346543060001065>

- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.
- Pasternak, E., Fenichel, R., & Marshall, A. N. (2017). Tips for creating a block language with Blockly. In F. Turbak, J. Gray, C. Kelleher, & M. Sherman (Eds.), *2017 IEEE Blocks and Beyond Workshop* (pp. 21–24). <https://doi.org/10.1109/BLOCKS.2017.8120404>
- Pellas, N., & Vosinakis, S. (2018). The effect of simulation games on learning computer programming: A comparative study on high school students' learning performance by assessing computational problem-solving strategies. *Education and Information Technologies*, 23(6), 2423–2452. <https://doi.org/10.1007/s10639-018-9724-4>
- Popat, S., & Starkey, L. (2019). Learning to code or coding to learn? A systematic review. *Computers & Education*, 128, 365–376. <https://doi.org/10.1016/j.compedu.2018.10.005>
- Raspberry Pi Foundation. (n.d.). *About us*. <https://www.raspberrypi.org/about/>
- Relkin, E., de Ruiter, L. E., & Bers, M. U. (2021). Learning to code and the acquisition of computational thinking by young children. *Computers & Education*, 169, Article 104222. <https://doi.org/https://doi.org/10.1016/j.compedu.2021.104222>
- Repenning, A., Webb, D., & Ioannidou, A. (2010). Scalable game design and the development of a checklist for getting computational thinking into public schools. In G. Lewandowski, S. Wolfman, T. J. Cortina, & E. L. Walker (Eds.), *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 265–269). ACM. <https://doi.org/10.1145/1734263.1734357>
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60–67. <https://doi.org/10.1145/1592761.1592779>
- Rodríguez-Martínez, J. A., González-Calero, J. A., & Sáez-López, J. M. (2020). Computational thinking and mathematics using Scratch: An experiment with sixth-grade students. *Interactive Learning Environments*, 28(3), 316–327. <https://doi.org/10.1080/10494820.2019.1612448>
- Sáez-López, J.-M., Román-González, M., & Vázquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using “Scratch” in five schools. *Computers & Education*, 97, 129–141. <https://doi.org/10.1016/j.compedu.2016.03.003>
- Saldaña, J. (2016). *The coding manual for qualitative researchers* (3rd ed.). SAGE.
- Scherer, R., Siddiq, F., & Viveros, B. S. (2019). The cognitive benefits of learning computer programming: A meta-analysis of transfer effects. *Journal of Educational Psychology*, 111(5), 764–792. <https://doi.org/10.1037/edu0000314>
- Scherer, R., Siddiq, F., & Viveros, B. S. (2020). A meta-analysis of teaching and learning computer programming: Effective instructional approaches and conditions. *Computers in Human Behavior*, 109, Article 106349. <https://doi.org/10.1016/j.chb.2020.106349>
- Segura, R. J., del Pino, F. J., Ogáyar, C. J., & Rueda, A. J. (2020). VR-OCKS: A virtual reality game for learning the basic concepts of programming. *Computer Applications in Engineering Education*, 28(1), 31–41. <https://doi.org/10.1002/cae.22172>
- Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: A sociocultural perspective. *Computer Science Education*, 29(2–3), 136–176. <https://doi.org/10.1080/08993408.2019.1608781>

- Seralidou, E., & Douligeris, C. (2019). Learning with the AppInventor programming software through the use of structured educational scenarios in secondary education in Greece. *Education and Information Technologies*, 24(4), 2243–2281. <https://doi.org/10.1007/s10639-019-09866-7>
- Serrano Pérez, E., & Juárez López, F. (2019). An ultra-low cost line follower robot as educational tool for teaching programming and circuit's foundations. *Computer Applications in Engineering Education*, 27(2), 288–302. <https://doi.org/10.1002/cae.22074>
- Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850–858. <https://doi.org/10.1145/6592.6594>
- Spector, J. M. (2005). Innovations in instructional technology: An introduction to this volume. In J. M. Spector, C. Ohrazda, A. Van Schaack, & D. A. Wiley (Eds.), *Innovations in instructional technology: Essays in honor of M. David Merrill* (pp. xxxi–xxxvi). Erlbaum.
- Strawhacker, A., & Bers, M. U. (2015). “I want my robot to look for food”: Comparing kindergartner’s programming comprehension using tangible, graphic, and hybrid user interfaces. *International Journal of Technology and Design Education*, 25(3), 293–319. <https://doi.org/10.1007/s10798-014-9287-7>
- Strawhacker, A., Lee, M., & Bers, M. U. (2018). Teaching tools, teachers’ rules: Exploring the impact of teaching styles on young children’s programming knowledge in ScratchJr. *International Journal of Technology and Design Education*, 28(2), 347–376. <https://doi.org/10.1007/s10798-017-9400-9>
- Sullivan, A., & Bers, M. U. (2016). Robotics in the early childhood classroom: Learning outcomes from an 8-week robotics curriculum in pre-kindergarten through second grade. *International Journal of Technology and Design Education*, 26(1), 3–20. <https://doi.org/10.1007/s10798-015-9304-5>
- Sullivan, A., & Bers, M. U. (2019). Investigating the use of robotics to increase girls’ interest in engineering during early elementary school. *International Journal of Technology and Design Education*, 29(5), 1033–1051. <https://doi.org/10.1007/s10798-018-9483-y>
- Sun, J. C.-Y., & Hsu, K. Y.-C. (2019). A smart eye-tracking feedback scaffolding approach to improving students' learning self-efficacy and performance in a C programming course. *Computers in Human Behavior*, 95, 66–72. <https://doi.org/10.1016/j.chb.2019.01.036>
- Sweller, J., van Merriënboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, 10(3), 251–296. <https://doi.org/10.1023/A:1022193728205>
- Taylor, M. S. (2018). Computer programming with pre-k through first-grade students with intellectual disabilities. *Journal of Special Education*, 52(2), 78–88. <https://doi.org/10.1177/0022466918761120>
- Taylor, K., & Baek, Y. (2019). Grouping matters in computational robotic activities. *Computers in Human Behavior*, 93, 99–105. <https://doi.org/10.1016/j.chb.2018.12.010>
- Tóth, T., & Lovászová, G. (2021). Mediation of knowledge transfer in the transition from visual to textual programming. *Informatics in Education*, 20(3), 489–511. <https://doi.org/10.15388/infedu.2021.20>
- Unal, A., & Topu, F. B. (2021). Effects of teaching a computer programming language via hybrid interface on anxiety, cognitive load level and achievement of high school students. *Education and Information Technologies*, 26(5), 5291–5309. <https://doi.org/10.1007/s10639-021-10536-w>

- Wang, X.-M., & Hwang, G.-J. (2017). A problem posing-based practicing strategy for facilitating students' computer programming skills in the team-based learning mode. *Educational Technology Research and Development, 65*(6), 1655–1671. <https://doi.org/10.1007/s11423-017-9551-0>
- Weintrop, D., & Wilensky, U. (2014). Situating programming abstractions in a constructionist video game. *Informatics in Education, 13*(2), 307–321.
- Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education, 18*(1), Article 3. <https://doi.org/10.1145/3089799>
- Weintrop, D., & Wilensky, U. (2019). Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education, 142*, Article 103646. <https://doi.org/https://doi.org/10.1016/j.compedu.2019.103646>
- Wing, J. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35. <https://doi.org/10.1145/1118178.1118215>
- Wu, T.-T., & Chen, J.-M. (2022). Combining Webduino programming with situated learning to promote computational thinking, motivation, and satisfaction among high school students. *Journal of Educational Computing Research, 60*(3), 631–660. <https://doi.org/10.1177/07356331211039961>
- Yallihep, M., & Kutlu, B. (2020). Mobile serious games: Effects on students' understanding of programming concepts and attitudes towards information technology. *Education and Information Technologies, 25*(2), 1237–1254. <https://doi.org/10.1007/s10639-019-10008-2>
- Yildiz Durak, H. (2020). The effects of using different tools in programming teaching of secondary school students on engagement, computational thinking and reflective thinking skills for problem solving. *Technology, Knowledge and Learning, 25*(1), 179–195. <https://doi.org/10.1007/s10758-018-9391-y>
- Zhang, L., & Nouri, J. (2019). A systematic review of learning computational thinking through Scratch in K–9. *Computers & Education, 141*, Article 103607. <https://doi.org/10.1016/j.compedu.2019.103607>
- Zhong, B., & Li, T. (2020). Can pair learning improve students' troubleshooting performance in robotics education? *Journal of Educational Computing Research, 58*(1), 220–248. <https://doi.org/10.1177/0735633119829191>
- Zhong, B., Zheng, J., & Zhan, Z. (2023). An exploration of combining virtual and physical robots in robotics education. *Interactive Learning Environments, 31*(1), 370–382. <https://doi.org/10.1080/10494820.2020.1786409>

Authors

NDUDI OKECHUKWU EZEAMUZIE obtained his PhD from the Teacher Education and Learning Leadership Unit, Faculty of Education at the University of Hong Kong; email: amuzie@connect.hku.hk. His research focuses on Computer Science Education, STEM Education, Curriculum and Instructional Design, Computational Thinking, Learning Science, AI in Education, and Educational Technology.

MERCY NOYENIM EZEAMUZIE is a Teacher Librarian and College Board AP Research Teacher at International Christian School, Hong Kong; email: zeamuzie@connect.hku.hk. Her research interests include STEM Education, Library Management, Information Systems, Curriculum and Instructional Design.