

Design and Analysis of Microworlds and Puzzles for Block-Based Programming

Radek Pelánek

Tomáš Effenberger

Masaryk University

ARTICLE HISTORY

Compiled December 4, 2020

ABSTRACT

Background and Context: Block-based programming is a popular approach to teaching introductory programming. Block-based programming often works in the context of microworlds, where students solve specific puzzles. It is used, for example, within the Hour of Code event, which targets millions of students.

Objective: To identify design guidelines and data analysis methods for the iterative development of microworlds and puzzles for block-based programming.

Method: To achieve the objective, we provide a review of the literature, discussion of specific examples of microworlds and puzzles, and an analysis of extensive student data.

Findings: A wide range of programming microworlds share common elements. The analysis of data is useful for iterative improvement of microworlds and puzzles, serving several specific purposes.

Implications: Provided design guidelines and analysis methods can be directly used for the development and improvement of tools for introductory programming.

KEYWORDS

introductory programming; block-based programming; microworld; puzzle; difficulty; log analysis

1. Introduction

In this work, we study the intersection of three topics: block-based programming, microworlds, and well-structured problems (puzzles). This is not a negligible intersection, e.g., it includes many activities within the popular Hour of Code event, which has millions of participants (Wilson, 2015). Any improvement in this area can thus have a significant impact.

The three areas, whose intersection we study, are naturally related, but independent (see Figure 1). Block-based programming (Bau et al., 2017) is a popular approach used primarily for teaching introductory programming. It also has other applications, e.g., in industry (Weintrop et al., 2017). Microworlds are small, but complete version of some domain (Rieber, 1996). For example, a billiard microworld (Bertz, 1997) is a two-dimensional world with Newtonian mechanics. Puzzles are a type of well-structured problems, i.e., problems with clear rules and clear criterion for a solution.

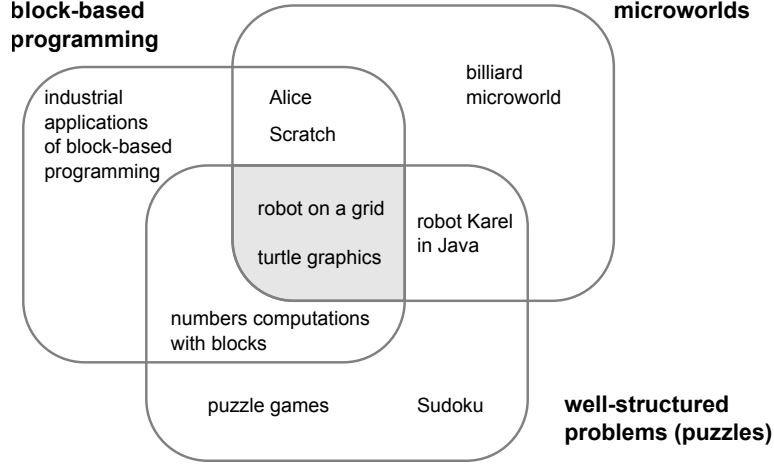


Figure 1. We consider the intersection of three topics: block-based programming, microworlds, and puzzles. Some topics (e.g., “turtle graphics”) can be placed in different positions in the diagram depending on their specific realization.

For something to be called a puzzle, we usually also expect it to have some inherent attractiveness (this aspect is captured in the definition “Puzzle is fun, and it has a right answer.” by Stan Isaacs).

These three areas can be combined in various ways. To clarify their relationships, Figure 1 shows a Venn diagram of different combinations together with specific examples. Environments like Scratch (Resnick et al., 2009) and Alice (Cooper et al., 2000) provide block-based programming in microworlds, but they are not (primarily) puzzles—their focus is on open-ended activities. Block-based programming can be used to solve a common well-structured problem like numerical computation (e.g., common problems in introductory programming like factorial or the greatest common divisor), although these problems are by most people not considered too attractive (“puzzle-like”). A billiard microworld (Bertz, 1997) is used for open-ended exploration in physics, whereas Robot Karel (Pattis, 1981) is a typical example of a programming microworld with a focus on solving specific puzzles. This microworld can be realized in a standard textual programming language like Java (Becker, 2001), but also in block-based programming. Turtle graphics (Caspersen and Christensen, 2000; Papert, 1980) is another typical example of a microworld that can be easily realized both in block-based and text-based programming.

In this paper, we study the intersection of the three described areas. We do not make any claims about the relative merits of approaches within and outside of the intersection. For example, turtle graphics can be used both in an open-ended manner, as well as for solving specific puzzles; microworlds can be programmed in block-based languages as well as using textual ones. Each approach has different contributions to the learning process. We do not aim to address such comparisons; some of these are available in previous work (discussed below). Our starting position is that block-based programming microworlds and puzzles can be a useful tool for teaching introductory programming, and we aim at making this tool as good as possible.

We focus on the design guidelines and data analysis methods that can be used for designing and improving block-based programming microworlds and puzzles. Although there is a rich literature on block-based programming, its focus is mostly on general properties of block programming environments and issues like the comparison

to text-based environments (Bau et al., 2015; Chen et al., 2019; Xu et al., 2019). The methodological literature is concerned mainly with open-ended problems, which are close to the constructivism roots of microworlds (Papert, 1980). However, practical applications today (e.g., activities within the Hour of Code event (Wilson, 2015)) often use puzzles with exact solutions rather than open-ended problems. The design of these applications is sometimes focused too much on “shallow features” like the appearance, animations, and sound effects, while they contain only simple puzzles, a small number of puzzles, or a series of puzzles with a steep learning curve. As Brusilovsky et al. (1997) highlighted already 20 years ago, it is not sufficient to have an attractive microworld; to make it useful, we also need a sufficient number of interesting puzzles to solve in the microworld.

We provide a discussion of design guidelines for block-based programming microworlds and puzzles. We focus only on the design issues related to mechanics and technology; we do not deal with the design issues relevant to aesthetics and story. Although these are clearly highly relevant for creating any engaging activity, the design process with respect to them is similar to other puzzle games (Schell, 2019). Our discussion of design guidelines is based on the review of literature and examples of specific microworlds and puzzles, particularly with the focus on variations of commonly used microworlds “robot on a grid” and “turtle graphics”. Together with our extensive experience in the design of microworlds and puzzles, this allows us to distill common themes and design guidelines.

We also provide a discussion of educational data mining methods for data analysis for iterative improvement of microworlds and puzzles. Specifically, we discuss methods for A) the analysis of difficulty and complexity of puzzles, microworld elements, and programming concepts, B) the analysis of solutions of a single puzzle. For each type of analysis, we explicitly discuss its purpose, and we illustrate the application of the method on data from real applications. We also identify gaps in the current state-of-the-art of educational data mining for block-based programming.

The discussion and data analysis is based on our experience with the iterative design of block-based programming microworlds and with the data collected by their implementations (6 microworlds, over 400 puzzles, solved by tens of thousands of students).

2. Related Work

As depicted in Figure 1, our work is focused at the intersection of block-based programming, microworlds, and puzzles; moreover, we aim to employ educational data mining techniques for iterative improvement of microworlds and puzzles. In this section, we overview related research from the concerned areas. For identifying relevant research, we used a combination of a search based on keywords for individual areas and analysis of papers published in major relevant journals and conference proceedings. In each case, there is a large body of existing research. We selected for our discussion particularly works that are connected to the studied intersection of block-based programming, microworlds, and puzzle design.

2.1. Block-based Programming

For the general principles of block-based programming, Bau et al. (2017) provide a good recent overview, including a discussion of advantages of blocks over text, such

as avoiding syntax errors, providing expert-level view (chunking), and decreasing cognitive load (no need to remember all available commands). Flannery et al. (2013) describe a specific case study in the application of block-based programming. For a wider context, Kelleher and Pausch (2005) provide a wide-ranging discussion of approaches to “lowering barriers” to introductory programming, including discussion of many variations on block-based programming.

The terminology in this area is not entirely standardized. Alternative notions are “visual programming” and “graphical programming”, which are sometimes used as synonyms to block-based programming and sometimes as supersets. For example, Weintrop (2019) considers only a specific realization of blocks with snapping under the title block-based programming. In this work, we consider block-based programming with a broad meaning—as programming done predominantly by dragging and clicking blocks instead of writing code.

A large portion of the research on block-based programming is devoted to the comparison with the standard textual approach and with the transition from blocks to text. For example, Bau et al. (2015) proposed an environment that should facilitate the transition from blocks to code. Price and Barnes (2015); Weintrop and Wilensky (2017) performed controlled experiments to test a hypothesis that it is beneficial to start an introductory programming course with block-based programming even when students will advance to text-based programming later. Chen et al. (2019) tackled a similar question through a large-scale observational study. These studies found some positive effects of using block-based programming but with several limitations and caveats. A recent meta-analysis (Xu et al., 2019) concluded that there is some evidence in favor of block-based programming but not very strong.

Another research direction focuses on user interface aspects of block-based programming, specifically on the Blockly editor, which is probably the most common implementation used to realize block-based programming concepts today. Fraser (2015) discusses what the authors of Blockly learned about designing a block-based programming language, e.g., left-right confusion (resolved by arrows) and loop-conditional confusion (resolved by different colors of the blocks and different categories in the blocks menu). Pasternak et al. (2017) provide tips for creating with Blockly. Weintrop and Wilensky (2015) suggest that the block representation of code should match the programming language to which the students are expected to move in the next phase (for easier interpretation of the effects that the blocks bring).

2.2. *Microworlds*

Rieber (1996) characterizes a microworld as “a small, but complete, version of some domain of interest” and discusses the relation of microworlds to simulations and games. The two most common microworlds used for teaching programming are turtle graphics and robot on a grid world, which is often called Karel the Robot based on the book that popularized this concept (Pattis, 1981). We discuss these two microworlds in more detail in the rest of the paper.

The beginnings of the use of microworlds for programming education are connected to Seymour Papert, who discussed this idea particularly with relation to turtle graphics. He considered microworlds mainly as a tool for explorations and for realizing constructivist ideas in education (Papert, 1980). Later works on microworlds are also often connected to constructivism (Hoyle et al., 2002; Rieber, 1992). In many recent applications of microworlds, however, solving well-structured problems plays an important

role. For example, many activities within the Hour of Code event (Wilson, 2015) use a combination of microworlds and puzzles. Xinogalos (2012); Xinogalos et al. (2006) discuss the application of the robot microworld for teaching object-oriented programming in Java. Papadopoulos and Tegos (2012) provide an overview of several programming microworlds and their goals.

Brusilovsky et al. (1997) provide a link between microworlds and block-based programming by discussing “mini-languages”—simplified languages that are useful particularly for programming within microworlds; they consider specifically several variants of the robot Karel world. They highlight several important features of mini-languages, e.g., simplicity, naturally visible operations, and attractiveness for the intended category of students.

2.3. Puzzles

The design of puzzles for programming microworlds can be inspired by ideas from puzzle games. Linehan et al. (2014) performed an analysis of four commercially successful puzzle games and summarised their common features by the following four principles that have high relevance to programming puzzles: *“1) the main skills learned in each game are introduced separately, 2) through simple puzzles that require only basic performance of that skill, 3) the player has the opportunity to practice and integrate that skill with previously learned skills, and 4) puzzles increase in complexity until the next new skill is introduced.”* Schell (2019) provides a high-level discussion of general game design, including many principles that are relevant for programming puzzles, e.g., that the puzzle should “invite the players to be solved”. Browne (2015) discusses puzzle design principles focusing particularly on logic puzzles.

A key aspect of puzzle design is the understanding of factors influencing puzzle difficulty. This issue has been studied for many puzzles, e.g., Tower of Hanoi (Kotovský et al., 1985) and other transport puzzles (Jarušek and Pelánek, 2011), or Sudoku (Pelánek, 2011). A good predictive model of puzzle difficulty is one of the steps necessary for the automated creation of puzzles, which falls under the more general topic of a “procedural content generation” (Shaker et al., 2016).

Concerning puzzles specifically for programming education, Vahldick et al. (2014) provide an overview of such puzzles and games. Many of these puzzles use block-based programming, e.g., LightBot, Robozone, and many Hour of Code activities (Wilson, 2015). Hicks (2016) explored the possibility of allowing students to create their own puzzles and found that the game editor should impose some constraints to enforce the best practices for puzzle games authoring. For example, it is crucial to require the authors to complete their own puzzles.

2.4. Educational Data Mining

Educational data mining can be used for many different purposes (Romero and Ventura, 2013). A common one is student modeling (Pelánek, 2017) that can be used for the adaptive behavior of learning systems. We are interested particularly in techniques that can be used for iterative improvement of learning systems—Baker (2016) discusses such techniques under the title “stupid tutoring systems, intelligent humans”, Aleven et al. (2016) call this approach “design-loop adaptation”. Other closely related research concerns the analysis of educational games; a common topic in this direction is the analysis of level progression (Harpstead and Aleven, 2015; Hicks et al., 2016; Horn

et al., 2016).

Ihantola et al. (2015) provide an overview of research concerning the application of educational data mining techniques specifically for programming education. We highlight specifically recent research related to introductory block-based programming. Aivaloglou and Hermans (2016) performed an exploratory analysis of data collected from Scratch (Resnick et al., 2009) and found that in an open-ended setting, most students do not use programming abstraction concepts such as functions and exhibit undesirable programming practices such as code duplication. Brown and Altmirri (2014) reported that educators often do not agree on which are the most frequent misconceptions of novice programmers and that the educators’ intuition usually does not match the observations.

Piech et al. (2012) argue that it is necessary to explore not only the final submitted program but also the student’s path towards the solution in order to reveal the student’s misconceptions. They propose a data-driven approach for constructing a finite state machine that describes a given programming problem, where the states correspond to clusters of similar program snapshots. To cluster the program snapshots, they use a similarity measure based on comparing snapshots’ abstract syntax trees and sequences of function calls.

Grover and Basu (2017) developed block-based programming problems to examine misconceptions. They found that even after successful completion of an introductory programming course, middle school students do not understand well basic programming concepts such as loops and variables. Grover et al. (2017) describe an iterative process to develop block-based programming problems with evaluation rubrics to measure computational thinking skills, using detailed log data and even screen recordings. Kelleher and Hnin (2019) analyzed which factors contribute to higher cognitive load in block-based programming puzzles. For example, using methods that are not yet familiar to the students increase the cognitive load, while locking some blocks in the solution (so that the students cannot modify them) decrease the cognitive load. On the other hand, the number of steps and attempts were not good predictors of the cognitive load.

Several studies focus on the automatic generation of hints for introductory programming problems from log data. To deal with the diversity of possible programs, Iii et al. (2014) suggest using world-states (outcomes of the programs, e.g., a path traversed by the robot) instead of code-states. Another strategy to reduce the variability of the collected programs is to canonize them (Rivers and Koedinger, 2017). To compare hint generation algorithms, Price et al. (2019a) define a quality measure using an expert-annotated dataset with both block-based and text-based programming problems.

3. Design of Block Programming Microworlds and Puzzles

In this section, we discuss the design of microworlds and puzzles for block-based programming in general. In the next section, we illustrate the discussed general issues using specific examples.

Our aim is to provide design “guidelines”, not “solutions”. We explicitly highlight decisions that need to be made and list potential options. The specific choice of a suitable option depends on the context of a particular application (e.g., the target audience).

3.1. Goals

Before discussing the design guidelines, it is useful to explicitly formulate the goals we want to achieve by using block-based programming microworlds and puzzles:

- To practice computational thinking (Shute et al., 2017), particularly problem-solving skills and a “debugging approach” to solving problems.
- To introduce (demonstrate) basic concepts in programming, e.g., sequencing, the importance of action order, repetition, conditional evaluation, variables.
- To motivate students to learn more about programming.

Notably, the goal of block-based programming puzzles is not to teach all intricacies of real-life programming, but rather serve as a gentle introduction to programming. As a consequence, we want to make the solution process enjoyable and interesting for solvers, transferring most of the difficulties to designers. Authors of the microworld and puzzles should do the hard work of creating a nice world and puzzles such that they “want to be solved” and have an elegant solution, which may be nontrivial, but should be within reach of a novice programmer. Such a combination can make the solver feel powerful.

The above-stated goals are not, of course, universal. A specific application may have slightly different goals and may differ in the weight given to individual goals. Such differences may (and should) translate into design decisions.

3.2. Designing the Microworld

The design of a microworld is a balancing act—our aim is to find a suitable trade-off between simplicity and complexity of the microworld. On the one hand, we want to aim at simplicity. We do not try to teach everything, but a specific aspect well. Therefore, it is useful to strip the microworld rules and the available commands to the necessary minimum, so that the microworld is easily understandable even to complete novices. At the same time, the microworld needs to have sufficient complexity to allow interesting puzzles and the potential for the illustration of several computer science concepts. Novelty is a key aspect in engagement (Lomas et al., 2017)—the microworld should thus enable sufficiently many elements that can be (incrementally) introduced to keep solvers engaged.

Since motivation is a key goal, the microworld should be attractive and intuitive. An important step to achieving this is the use of “naturally visible operations” (Brusilovsky et al., 1997). This is most easily achieved with microworlds with the following attributes:

- a two-dimensional world with an agent,
- the agent has coordinates x, y , direction, and potentially other attributes,
- the world can contain other objects (e.g., obstacles, treasures, other agents),
- the agent can detect some aspects of the world (e.g., other objects, background color),
- the agent has movement actions and potentially other actions that modify the world (e.g., pick, draw, shoot).

Figure 2 provides an illustration of two commonly used microworlds (which are also discussed in more detail in the next section): a robot on a grid and turtle graphics. Note that through the paper, we use simple, abstract symbols to depict microworlds since our aim is to highlight similarities among different settings. Real applications

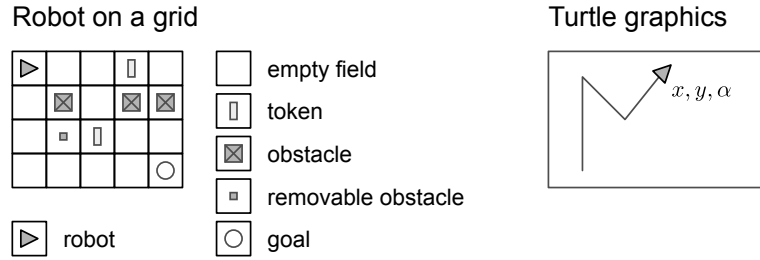


Figure 2. Common microworlds: robot in grid, turtle graphics.

typically use much more concrete and elaborated graphics.

Many variations are, of course, possible. We can use other dimensionality of the world. One dimensional world (e.g., moving along a street) is, however, hard to design in such a way that it provides enough complexity for a sufficient number of interesting puzzles and, at the same time, is intuitive and attractive. On the other hand, we can go for a three-dimensional world (e.g., using obstacles of different height). A three-dimensional world adds the possibility for a more attractive look of the microworlds. For this reason, it is used in many current applications. However, it does not add anything fundamental from the design point of view, and it has the disadvantage of increasing the cognitive load of solvers by making the interface more complex. A two-dimensional world seems like a good compromise between simplicity and complexity.

A natural extension of basic versions of microworlds is the addition of multiple agents—these can be either “friends”, which have to be coordinated, or “enemies”, which have to be defeated. This aspect introduces interesting principles both from the computer science perspective (parallelism, coordination, communication) and from the puzzle design perspective. Often it is also quite natural and intuitive.

3.3. Choosing Blocks and Block Menu

The design of the microworld is closely connected to the choice of available blocks and the design of the block menu. The main choices concerning blocks and block menu are: what blocks to use, how to display them, and how to organize them.

3.3.1. Content of Blocks

The basic types of commands (blocks) are the following:

- (1) *Elementary commands*: basic actions in the microworlds like movement, drawing, pick up, shooting.
- (2) *Basic control flow instructions*: simple repetition (repeat k times), conditional repetition (while), conditional execution (if, if-else).
- (3) *Additional programming concepts*: variables (setting, changing, and using values), functions (definition, call).

Elementary commands are clearly necessary—without them we would not be able to use the microworld. It is possible to use only the elementary commands (e.g., for simple puzzles involving only a sequence of actions in the microworld). However, to provide meaningful practice of programming, we need to include also the basic control flow instructions. Functions without parameters and return values can be still realized rather easily, but for additional programming commands (variables, functions with

parameters), it is challenging to design the block-based programming interface to be intuitive and easy to use—the use of variables in blocks is typically rather clumsy compared to text-based programming. A microworld should offer enough interesting puzzles even without these additional concepts. They can be included, but mainly as an extension for interested students.

3.3.2. Form of Blocks

Blocks can be either *textual* or *pictorial*. Programs built with textual blocks closely resemble textual programs and are more common. Pictorial blocks have lower expressive power, but they are sufficient for many puzzles and have several advantages: compactness, attractiveness, and accessibility for pre-readers. A well-known environment that uses pictorial blocks is ScratchJr (Flannery et al., 2013); in this environment, even nontrivial concepts like message-passing are successfully expressed using blocks. The transition between pictorial and textual blocks can be realized as a smooth transition¹. The design of blocks also requires attention to user interface aspects—these are discussed by Fraser (2015).

Block-based programming interfaces further differ in the way the individual blocks are connected into a complete program. There are three common options:

- *Predefined square-grid* into which blocks are placed. This option provides clear guidance on what to do to create a program but also is the most restrictive.
- *Snapping blocks*, the most common form of blocks, implemented, e.g., in Blockly (Fraser, 2015) and Scratch (Resnick et al., 2009). In contrast to a predefined square-grid, users can create arbitrarily nested programs.
- *Moving tiles*, known from Parson’s puzzles (Parsons and Haden, 2006). Without snapping, it is easier to move the blocks around and swap them; on the other hand, nesting becomes less intuitive.

In addition to the trade-off between simplicity and flexibility, this choice affects how convenient it will be to practice various programming concepts. In particular, as the predefined square-grid does not allow for nested structures, trying to practice loops with them would be rather cumbersome. On the other hand, using a grid with multiple rows, one row per function, naturally forces students to think about a suitable function decomposition.

3.3.3. Block Menu

We also need to decide how to present the available blocks to users. There are two basic options:

- *Flat block menu*, i.e., all available blocks are shown. This approach is easy to use, but it is feasible only with a small set of commands (approximately up to 10).
- *Hierarchical block menu*, i.e., blocks are sorted into categories. This approach allows usage of a rich command set but leads to a slightly slower creation of programs (more clicks necessary for the choice of a block). It can also increase cognitive load as the solver needs to remember the location of blocks within the menu. This risk can be mitigated by the use of a well-designed hierarchy.

¹As done, for example, in OzoBlockly, <https://ozoblockly.com/>.

3.3.4. Scaffolding Blocks

The scaffolding principle (Jumaat and Tasir, 2014) can be used in several ways with respect to blocks. Introductory problem sets can be used with the flat block menu, with only a limited set of commands available (to facilitate easy onboarding). Advanced problem sets can use a full set of commands organized in the hierarchical block menu (to allow sufficient complexity). It is also possible to use the scaffolding of blocks. For example, we can introduce some actions in steps, increasing the flexibility with which they can be used:

- (1) Action with a fixed parameter (“turn right 90 degrees”).
- (2) Action with a choice of a parameter from a limited set (“turn right [30/60/90] degrees”).
- (3) Action with a variable parameter (“turn right X degrees”, keyboard input of X).

3.4. Setting up Limits

An important element of block-based programming puzzles is the use of a limit on the number of blocks. There are two main reasons for the use of limits:

- To enforce the use of more complex blocks (loops, conditions, functions) and thus to force students to learn the programming concepts (instead of making programs as a long sequence of elementary actions).
- To make problems more interesting and challenging and to increase the puzzle-solving aspect.

These goals can also be achieved in other ways. We can require generalizability: the created program should work in several settings (e.g., different initial positions of the agent). This approach may even be pedagogically preferable to the use of limits, as it is more closely connected to typical applications of programming beyond microworlds. However, this approach leads to a more complex user interface and user interaction. The use of limits is typically much simpler, mostly sufficient, and prevalent in current block-based programming puzzles.

The basic idea of a “limit” can be realized in different ways:

- *Hard* limit: The user interface counts the blocks and does not allow users to place more blocks than the limit.
- *Soft* limit: It is possible to finish the puzzle with more blocks than the limit; the solver only gets feedback that the solution was not optimal.
- *Implicit* limit: Even in the absence of an explicit limit, there may be practical implicit limits. When we use a block canvas without scrolling, the size of the canvas automatically creates a limit on the number of blocks.

In the case of explicit limits, the limit may have different scopes. The basic type of limit is on the total count of blocks. Other possibilities are limits on specific types of blocks (e.g., “3 times movement, 2 times shooting, unlimited control blocks”) or limits per functions (e.g., “each function can have at most 5 blocks”). It is also necessary to choose what counts as a “unit” in counting the limit. Is an “if” block a single unit, or do we also count blocks within the condition?

3.5. *Creating and Sequencing Puzzle Sets*

Once we have a microworld and set of puzzles, these puzzles can be presented to students in a wide variety of ways, e.g., a fixed sequence of puzzles, a list of puzzles from which a student (or teacher) can freely choose, or a choice of puzzles by an adaptive algorithm which takes student performance into account. In all these cases, it is useful to have the set of puzzles structured into homogeneous sets (often called “levels”).

Puzzles within one level should be similar along many of the above-discussed design aspects:

- the use of the same blocks in the menu,
- the use of same (or closely similar) microworld elements,
- solutions requiring similar programming concepts,
- similar difficulty,
- similar tightness of limits on the number of blocks (none, loose, tight).

Even though there is the variability of the presentation modes, in all cases, it is suitable to focus on the sequencing, both sequencing of levels and sequencing of puzzles within levels. The sequencing should support scaffolding (Jumaat and Tasir, 2014) and a suitable increase of difficulty to match the increase in skills of students (Nakamura and Csikszentmihalyi, 2014; Schell, 2019). There are also other aspects worth taking into account, e.g., variability and novelty (Lomas et al., 2017).

The creation of good puzzle sets and their sequencing is difficult. For the author of a puzzle, it is very hard to anticipate how novice programmers will approach the puzzle and how difficult it will be. It is, therefore, highly beneficial to approach the development of puzzles in an incremental fashion based on data analysis. This is a key topic which we elaborate on in Section 5.

4. Specific Microworlds and Puzzles

We now discuss several specific microworlds together with examples of puzzles. The goal of this discussion is not completeness, which would not be feasible anyway since too many variants of microworlds currently exist. Our goal is rather to provide representative examples and to show how even slight variations can lead to microworlds with a quite different focus and pedagogical use.

4.1. *Overview of Examples*

Table 1 provides a list of examples of block-based programming puzzles in microworlds. In selecting examples into this table, we tried to cover both “what is popular” and “what is possible”. We include our own examples—for these, we have data for analysis, and we discuss them in more detail in the rest of the paper. We include the most well-known examples, e.g., the most popular Hour of Code puzzles and the highly-rated 7 Billion Humans app. We also included examples described in research papers. Finally, we tried to find specifically illustrative examples with a combination of features that were not covered by previous ones.

The table presents for each example an overview of the key characteristics discussed in the previous section. Its main purpose is to provide an illustration of the wide number of possibilities of how the basic elements of microworlds and blocks can

be combined. This presentation is necessarily simplified. For example, both flat and hierarchical menus are present in UP examples (in a scaffolded manner), the Code.org examples (10 and 11) are lessons that are parts of larger courses that also contain other puzzle games with other characteristics, and Gladiabots use a very specific form of blocks (“decision trees”) that do not fit nicely to the categories used in the table.

The “concepts” columns show which programming concepts are practiced within each puzzle game. This characterization is again simplified. Each concept is usually practiced only in a subset of puzzles, not in all. We consider only the concepts occurring most commonly for each puzzle game. All games include the concept *sequential composition of commands*, which we thus omit from the table even though it is typically one of the important learning objectives of these games. *While loops* are understood quite generally, including repeat until structure and looping using jumps (e.g., in 7 Billion Humans). *Functions* correspond to a wide range of possible realizations, sometimes including creating own functions (e.g., LightBot), in other cases, only calling predefined ones (e.g., Frozen). *Recursion* can also range from very simple (e.g., in LightBot: tail recursion, no parameters, presented to students as loops) to complex (e.g., RoboZ-1e). *Parallelism* denotes basically any use of multiple agents; in some cases, it means writing a single program for multiple robots; in other cases, it means just the need to think about synchronization of parallelly occurring events (e.g., in combat).

4.2. Robot on a Grid

As Table 1 illustrates, a very common microworld for introductory programming is a robot on a grid world. The world is a two-dimensional grid, typically up to 10×10 size. The robot moves through the world with commands like movement forward or turning left. The goal is to program the robot to fulfill a given task, e.g., get to a selected cell on the grid or collect tokens.

This type of microworld has been used intensively for a long time. The first widely-known incarnation of this idea is probably Karel the Robot (Pattis, 1981). Several variants of this approach are discussed in Brusilovsky’s overview of mini-languages (Brusilovsky et al., 1997) and are often used in university courses on introductory programming (CS1), e.g., Becker (2001) discusses version in the Java programming language. Variations of this microworld are also used in artificial intelligence education, e.g., as the Wumpus world (Russell and Norvig, 2016) or Robby the Robot example for illustration of genetic algorithms (Mitchell, 2009). This type of microworld is also used in board games (e.g., the RoboRally game).









































































































































































































The common elements of this kind of microworld are:

- *Obstacles*. The basic version of obstacles (walls) just blocks a specific cell in a grid. Obstacles can be removable (doors that are opened by a key, walls that can be destroyed by shooting or a bomb), or that can be overcome by special actions (e.g., jumped over).
- *Tokens*. We use the notion token to denote objects in the world that can be collected. In specific instances, these are called, for example, treasures, beepers, or cans.
- *Cell properties*. Cells in the grid can have properties, which the robot can sense. A typical property is the color of the cell.

Many additional elements can be used, e.g., conveyor belts, pushers, or teleports.

The basic types of commands that are used for programming the robot are:

Table 1. Examples of block-based programming puzzles.

game	microworld	blocks	limits	concepts		
1 UP: Arrows	robot: 2D	→ 	 	      		
2 UP: Robotanist	robot: 2D	→ 	 f_x	      		
3 UP: Shapes	drawing: shapes	T 		      		
4 UP: Turtle	drawing: turtle	T 		      		
5 UP: Combat	robot: 2D	T 	 Σ	      		
6 RoboMission	robot: 2D	T 	 Σ	      		
7 HoC: Frozen	drawing: turtle	T 	 Σ	      		
8 HoC: Minecraft	robot: 2D	T 	 Σ	      		
9 HoC: Moana	robot: 2D	T 	 Σ	      		
10 Ocean Scene	drawing: lines	→ 	  Σ	      		
11 Artist	drawing: turtle	T 	  Σ	      		
12 LightBot	robot: 2D	→ 	 f_x	      		
13 RoboZZle	robot: 2D	→ 	 f_x	      		
14 Collabots	robot: 2D	→ 		      		
15 Cargo-Bot	robot: 1D	→ 	 $f_x \Sigma$	      		
16 Tynker	robot: 1D, 2D	T 	 Σ	      		
17 Move the turtle	drawing: turtle	T 		      		
18 BOTS	robot: 3D	T 	 Σ 	      		
19 7 billion humans	robot: 2D	T 	 Σ	      		
20 Gladiabots	robot: 2D	→ 		      		
→ pictorial blocks	 placement to grid	 flat menu	 no limit	Σ limit on total length	 for loop	 variables
T textual blocks	 snapping blocks	 hierarchical menu	 soft limit	f_x limit per function	 while loop	 recursion
	 moving tiles		 hard limit	 limit per block type	 conditions	 parallelism
					 functions	
Full names and references:		1–5 Umíme programovat https://www.umimeprogramovat.cz		6 RoboMission https://en.robomise.cz/		
7 Hour of Code: Code with Anna and Elsa https://hourofcode.com/frzn		8 Hour of Code: Minecraft: Voyage Aquatic https://hourofcode.com/mchoc <i>GhasemAghaei et al. (2017)</i>		9 Hour of Code: Moana: Wayfinding with Code https://hourofcode.com/moana		
10 Code.org Course A: Ocean Scene https://studio.code.org/s/coursea-2019		11 Code.org Course E: Functions with Artist https://studio.code.org/s/coursee-2019		12 LightBot https://hourofcode.com/lightbot <i>Gouws et al. (2013)</i>		
13 RoboZZle http://robozzle.com/		14 Collabots http://dps.univ-fcomte.fr/collabots/		15 Cargo-Bot https://twolivesleft.com/CargoBot/ <i>Tessler et al. (2013)</i>		
16 Tynker lost in space https://www.brainpop.com/games/tynkerlostinspace/		17 Move the turtle https://movetheturtle.com/		18 BOTS <i>Hicks et al. (2015), Eagle et al. (2012)</i>		
19 7 Billion Humans https://tomorrowcorporation.com/7billionhumans		20 Gladiabots https://gladiabots.com/				

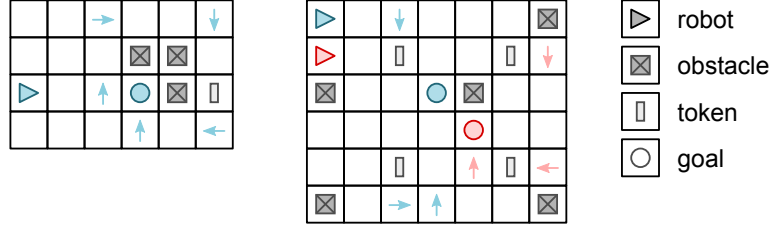


Figure 3. Arrows game puzzles. The arrows in light color show a solution to each puzzle.

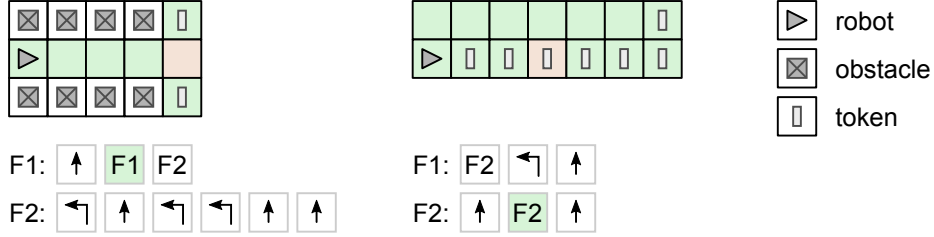


Figure 4. A variation on the robot on a grid theme with a very simple programming interface, yet sufficient potential for complex puzzles.

- elementary movement: step forward, turn left, turn right,
- extended movement: step backward, sidestep, jumping over an obstacle,
- additional actions: pick up a token, put down a token, shoot,
- conditions, sensing: the detection of properties of objects at the current location (cell properties, tokens), the detection of objects around the robot,
- general control flow: repetition, variables, functions.

4.2.1. Programming by Arrows

This microworld can be used to create interesting puzzles with an even simpler “programming” interface than the standard block-based programming. The robot can be programmed by placing arrows (or other elementary pictorial commands) either directly into the grid world or into a predefined command grid. This setting typically uses strict limits on the number of used commands, which forces the solver to find an elegant solution and strengthens the programming aspect of the task.

A specific simple variant is illustrated in Figure 3. The goal is to get the robot to collect all tokens and reach the final cell. The robot is programmed by placing arrows directly into the grid. This microworld is suitable even for very young children, and yet it offers many natural extensions that can lead to challenging puzzles, e.g., with the use of multiple robots (which have to avoid collisions), disappearing arrows, or bombs that explode after a fixed delay and destroy neighboring obstacles.

A more complex variant, which is closer to standard programming, is illustrated in Figure 4. In this case, the robot is still programmed using simple pictorial blocks (particularly arrows), but now the program is created by specifying functions. Functions are simple linear sequences of commands, including calls of other functions and recursion. This leads to a quite simple and intuitive interface, yet the presence of recursion enables the creation of very difficult puzzles. There are many variations on this particular theme, including Robotanist, Lightbot, and RoboZZle examples from Table 1.

4.2.2. Basic Block-based Version

The robot on a grid microworld can be used in a straightforward way together with block-based programming. The microworld makes it possible to develop a wide range of puzzles. Typical levels (puzzle sets) are:

- *Basic movement.* The goal is to navigate the robot to a target cell by a sequence of basic movement commands (forward, turn left, turn right).
- *Extended movement.* The robot can use additional movement commands like sidesteps or jumping over obstacles. The puzzles may use a limit on the number of blocks to enforce planning of the robot’s path.
- *Repetition.* Combination of the basic movement with simple repetition (a fixed number of iteration). The puzzles use a limit on the number of blocks in order to enforce the use of repeat command.
- *Conditions.* The use of conditional execution (if) and conditional loop (while). These puzzles often require finding a suitable “pattern” in the world (e.g., “every right turn on the path to the goal is on a red cell”).
- *Nested repetition.* Use of nested loops to encode more complex patterns.
- *Advanced conditions.* For the practice of the basic use of conditions, it is often sufficient to use simple cell properties like colors. More complex puzzles can be created with additional sensors (position, visibility) and with the use of logic operations.
- *Variables, functions.* More complex puzzles may be created with the use variables (used mainly for “counting” within the microworld) and functions (to practice abstraction of repeated sequences of actions).

A typical example of this microworld is the Karel the Robot (Becker, 2001; Pattis, 1981). In Table 1, examples of this type are Minecraft, Moana, or BOTS. In Figure 5, we provide an illustration of our variation on this theme, called RoboMission². The specific aspect of this variation is an implicit movement forward. The microworld is presented as a spaceship moving through space; in each step, the spaceship flies one step up, potentially combined with other movements (left, right), or action (shooting). This allows more compact solutions and thus more interesting puzzles with programs of limited scope—a disadvantage of the standard version is that a common high-level step “go around obstacle” requires many elementary commands to encode. A consequence of the implicit movement forward is that the robot moves quickly through the grid. To enable more complex puzzles, it thus requires “teleports” that return the robot back (see Figure 6, where teleports are marked by letters).

4.2.3. Combat

Robot combat is a general popular approach to learning programming. Robot competitions exist on a wide scale, from very abstract robot combat with simple rules to soccer matches involving physical robots in the real world. An illustrative example of robot competitions for introductory programming (in Java) is RoboCode (O’Kelly and Gibson, 2006). In Table 1, the competition approach is used in Gladiabots and UP: Combat.

The Gladiabots game uses a rather specific form of program specification (a variation on decision trees). The UP system provides realization of the combat using quite

²Due to the blind review requirement, only an acronym is used thorough this submission. Full name and references to literature will be provided in full version.

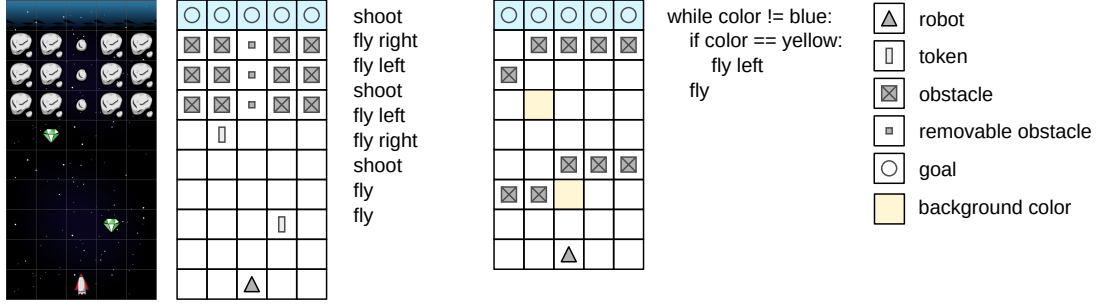


Figure 5. RoboMission is an example of a *robot on a grid* microworld. The goal of the robot (spaceship) is to collect all tokens (diamonds) and reach one of the cells with a circle (the last row). The robot must avoid obstacles (asteroids) and can remove smaller obstacles (meteoroids) by shooting them. To decide on the next action, the robot can measure the current background color and horizontal position. The main innovation of this microworld is the implicit movement forward, which allows for more compact programs.

standard block-based programming; see Figure 7 for illustration. The game uses the basic robot on a grid world with obstacles and treasures (small and big tokens of different values). The available actions include the basic movement and simple fighting (shooting and shielding). Programs can also use repetition and conditions (colors, in-front, visible).

The combat is realized as follows:

- The combat features either 2 or 4 players. Each arena is symmetric.
- Each arena has specific properties: available commands, a limit on the length of a program (number of blocks), a limit on the time to prepare a program.
- Within the time limit, each player creates a program (with no possibility of debugging). After the time limit, the programs are executed, and robots are awarded points according to the number of treasures collected and robots shot.

Although this game uses basically the same microworld as the standard variant, it leads to quite a different practice. In this case, there are no “correct solutions”, as the success of the created program depends on specific solutions submitted by other players. The time limit and the unavailability of debugging lead to the practice of writing short codes on the first attempt (ability to imagine robot behavior and perform the simulation in the head).

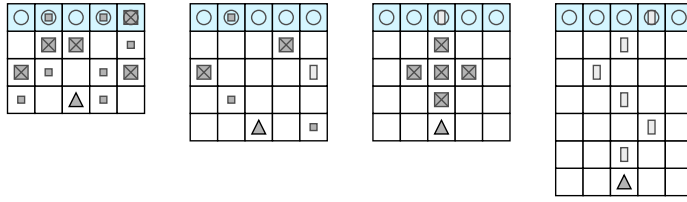
4.3. Drawing Microworlds

The second common microworld for introductory programming is again two-dimensional, but with significantly larger resolution (e.g., 400×400). In this case, the grid represents a canvas, and the agent moves around this canvas and produces a drawing.

This type of microworld can be naturally used for open-ended assignments, where students try to produce a drawing of their own design. In our discussion, we keep the focus on well-structured problems with clear goals—a goal in this setting is a specific drawing that should be created. For this use, we need to be able to automatically check whether a program satisfies the goal, i.e., produces the specified drawing. This step is nontrivial since the drawings are produced with limited numerical precision. If the same drawing is produced by two different approaches, the rounding errors may differ and the produced drawings are thus not exactly the same. A possible solution is to

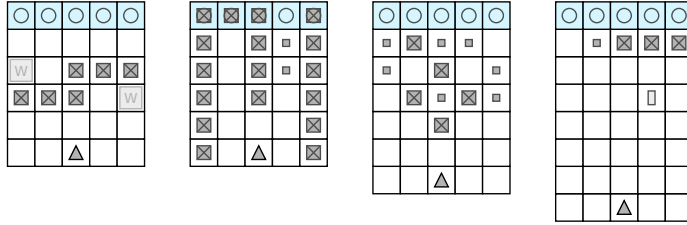
Basic movement

failure rate:
3–5%
median time:
20–30s



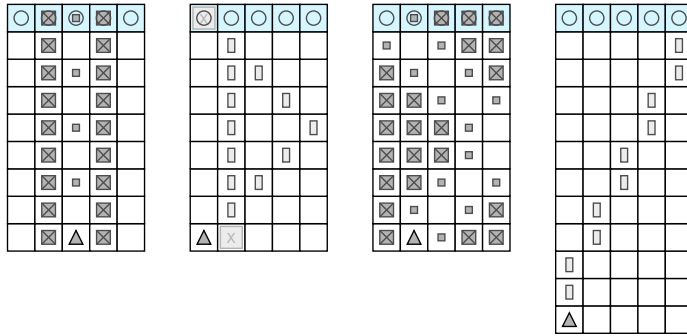
Advanced actions

failure rate:
1–20%
median time:
20–120s



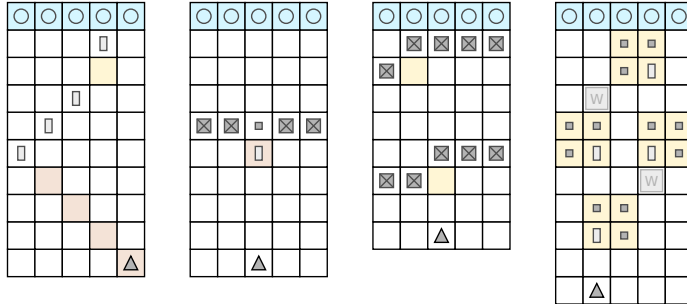
Repetition

failure rate:
4–25%
median time:
50–120s



Conditions

failure rate:
15–45%
median time:
60–120s



Nested loops

failure rate:
20–40%
median time:
100–160s

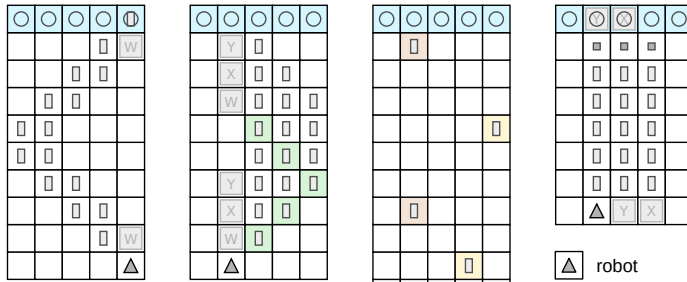


Figure 6. RoboMission puzzles with basic statistics about their difficulty.

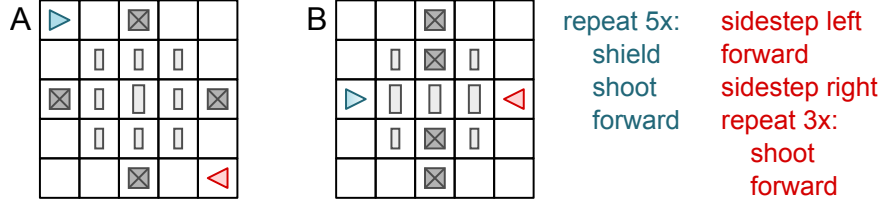


Figure 7. Examples of two combat arenas with codes for the arena B. In this case, the red robot is shot by the blue one in the fifth step of the simulation.

use a heuristic approach—to compare the target drawing and the produced drawing pixel by pixel and to accept any solutions for which the number of different pixels is under a specified limit (which can be tuned based on student data).

From the pedagogical point of view, a specific aspect of this type of microworld is the natural connection of programming and mathematics. It leads to the practice of geometry notions like coordinates, angles, and properties of geometrical objects. From a programming perspective, the most relevant concepts are sequencing (importance of ordering of commands), iteration (including nested loops), and potentially variables, functions, and recursion. On the other hand, this type of microworld is not suitable for the practice of conditions.

4.3.1. Turtle Graphics

Turtle graphics is one of the most well-known and widely used microworlds. The core mechanic of this microworld is that the agent (turtle) moves using commands like “forward d ”, “turn left α ”, and the movement produces lines.

The concept was popularized by Seymour Papert in the *Mindstorms* book (Papert, 1980), where he focused on the constructivist use of the microworld, but others soon proposed specific well-structured “puzzles” (Newell, 1988). The early use of turtle graphics was strongly associated with the Logo programming language, but today turtle graphics is used widely for introductory programming exercises in many other text programming languages (Caspersen and Christensen, 2000; Hromkovič et al., 2016). The concept can also be easily realized in block-based programming languages. In Table 1, UP: Turtle, Frozen, Artist, and Move the turtle provide mostly typical illustrations of such realization.

This microworld again offers a wide range of puzzles, ranging from very simple to complex. Common levels (puzzle sets) are the following (Figure 8 shows examples of specific problems in individual levels):

- *Basic drawing.* Drawing using just a small number of basic movements with fixed parameters (e.g., “forward 50”, “forward 100”, “turn right 90°”, “turn right 45°”).
- *Simple drawing.* Drawing using the basic movement with a free choice of parameters and using iteration (“repeat X”).
- *Patterns with repetition.* Same commands as in the simple drawing set, but more complex patterns that require the use of nested repetition.
- *Correct angles.* Same commands as in the simple drawing set. Puzzles use only simple iteration but require reasoning about angles.
- *Variables.* In addition to the basic commands, we allow the use of variables (set a variable, increase a variable by constant, use a variable). This allows, for example, the easy creation of spirals.

In the case of turtle graphics, the use of limits on the number of blocks is often not necessary because most solutions consist of a large number of lines, and without the use of a reasonably elegant solution, the code is simply too long. Block-based programming typically uses a restricted area for placement of blocks, and even if this area can be extended or scrolling is allowed, it effectively creates an implicit limit.

The basics of turtle graphics can be extended in several directions. One is a richer set of drawing commands, particularly the setting of colors and stroke width. Such extensions can increase attractiveness and are useful particularly for open-ended applications of turtle graphics. They, however, do not significantly influence the nature and design of well-structured puzzles.

Another significant extension is the use of functions and recursion. This is a powerful extension that allows us to draw elegant fractals using quite a short code; it is one of the key advantages of the turtle graphics concept. However, examples utilizing functions and recursion are significantly more difficult than other problems. Moreover, the realization of these concepts in block-based programming is not as intuitive as for basic commands. Puzzles involving functions, recursion, and fractals are possible, but mainly with the heavy use of scaffoldings, e.g., giving students the code which just needs to change few constants.

4.3.2. Shape Drawing

Turtle graphics is based on drawing lines. Another possibility is to allow commands for drawing other shapes, particularly elementary geometric figures (e.g., circles, squares, rectangles, triangles), or even bitmap images. This approach has been used in several demos and research works, e.g., Blockly Shape demo³, BeadLoom game (Boyce and Barnes, 2010), but is much less common than turtle graphics.

In this case, the basic command is “draw a shape”. There are two basic approaches to the realization of drawing. Firstly, the shape can be described by explicitly specifying its coordinates and attributes, e.g., a square is specified by the coordinates of its upper left corner and the length of an edge. This approach is suitable for the practice of cartesian coordinates and geometrical properties of objects but has a limited potential for the practice of programming concepts and for interesting puzzles.

Secondly, we can use an analogical approach to turtle graphics—a “painter” that moves around and draws shapes centered at his position. Drawing a square thus requires only a single parameter (the length of an edge), the coordinates are given implicitly by the position of the painter. This leads to a simpler interface and more interesting puzzles thanks to the interaction of drawing and painter movement.

Figure 9 shows specific examples of images that can be used as puzzles of reasonable difficulty with this approach. For these puzzles, we use the mobile painter and just four elementary types of shapes: a square, a rectangle, a circle, and a “diamond” (square rotated by 45 degrees).

Natural levels for this approach are:

- *Basic drawing.* A straightforward combination of elementary shapes, which leads to the practice of the basic user interface and movement.
- *Simple drawing.* A combination of a few elementary shapes, potentially with the use of overlap and thus the need to think about the correct ordering of actions.
- *Combining shapes.* Creation of new shapes from the provided elementary ones. A useful extension is to allow “xor” drawing, which allows easy creation of elegant

³<http://blockly-shapes.appspot.com/static/apps/shape/index.html>

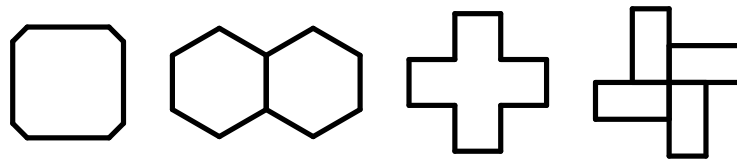
Basics

failure rate:
3–10%
median time:
40–100s



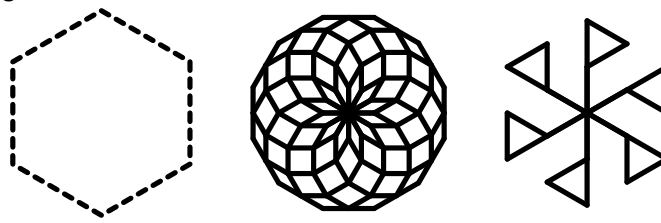
Simple drawing

failure rate:
10–20%
median time:
80–240s



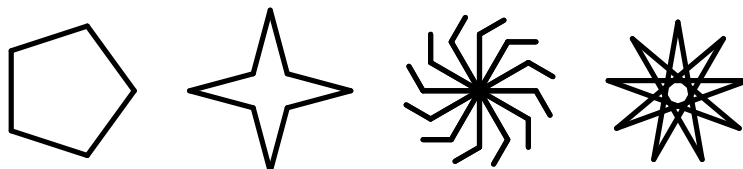
Patterns with repetitions

failure rate:
10–25%
median time:
120–230s



Correct angles

failure rate:
15–25%
median time:
150–230s



Variables

failure rate:
40–60%
median time:
150–280s

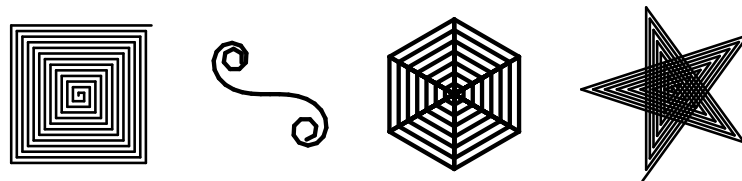


Figure 8. Turtle graphics puzzles with basic statistics about the difficulty of puzzles.

Basic drawing

failure rate:
5–30%
median time:
20–120s



Combining shapes

failure rate:
10–40%
median time:
80–300s



Iteration, variables, functions

failure rate:
20–80%
median time:
100–600s

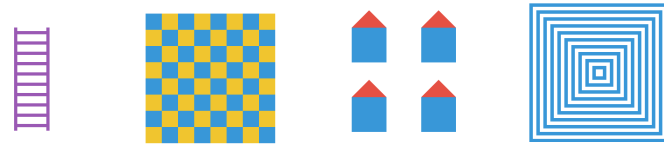


Figure 9. Shape drawing puzzles with basic statistics about the difficulty of puzzles.

images.

- *Iteration and variables.* The use of iteration, potentially with a simple use of variables as “counters” to draw shapes of different sizes.
- *Functions.* Functions can be used quite intuitively in this setting to define new shapes (“house”) and then draw these shapes.

5. Data Analysis

In this section, we provide an overview of data analysis techniques that are beneficial for iterative improvement for block-based programming microworlds and puzzles. We focus on techniques that are simple and generally applicable. For each type of technique, we describe the main principle, explicitly formulate purposes and applications of the technique, and provide specific examples of analysis.

For these specific examples, we use data from our implementations of microworlds discussed in the previous section. One implementation is an open-source project RoboMission (robot on a grid, 85 puzzles, illustrated in Figure 5 and Figure 6), which is available at en.robomise.cz and described by Effenberger and Pelánek (2018). The dataset used for the analysis contains 1.4 million program snapshots from 5800 students and is published online and described by Effenberger (2019).

Other analyzed microworlds are part of a commercial system *Umíme programovat* (umimeprogramovat.cz); these include Turtle Graphics (82 puzzles, examples in Figure 8), Shape Drawing (54 puzzles, examples in Figure 9), Arrows Game (94 puzzles, examples in Figure 3), Robotanist (87 puzzles, examples in Figure 4).

5.1. Data Logging

To analyze data, we first have to collect them. Compared to other learning domains, logging of data from programming exercises is more challenging—it requires us to consider several decisions and trade-offs, which can influence the type of analysis that will be feasible to do with the data. We outline the major decisions that need to be made. Compared to other discussions of data collection and logging, which focus mostly on text-based languages (Brown et al., 2014; Hundhausen et al., 2017; Ihantola et al., 2015), we consider issues specifically relevant to block-based languages and puzzle-solving activities.

5.1.1. Granularity

Programming data can be logged at different levels of granularity. Ihantola et al. (2015) describe six levels of granularity, but some of them make sense only for textual programming in an IDE. In the case of block-based programming in microworlds, the natural choices are:

- (1) only the final solution of each student,
- (2) every execution of the code,
- (3) every edit of the code.

Edit in a block-based programming interface refers to any change to one of the programming blocks, whether it is insertion, deletion, moving to another position, or the update of parameters (e.g., number of iterations, name of a variable); this corresponds to *line-level edits* in textual programming interfaces. Theoretically, even more granular data can be captured, such as every keystroke during the update of parameters and all mouse movements, but such data are relevant more to user-interface aspects than to the problem-solving activity itself.

More granularity means larger memory requirements, but more potential for data analysis. The suitable choice of a trade-off depends on a specific use case, but at least for new systems (with few users), we consider it beneficial to log all edits.

5.1.2. What to Log

The main events to log are code edits and executions. Other potentially useful events include start and end of the attempt (student opened or closed puzzle), code reset, opening a toolbox category, request for a hint, and feedback from the system after an attempt. Table 2 presents an overview of the data to log about these events. In the following, we discuss some important conceptual aspects and decisions concerning what to log.

Attempts. One student can solve the same puzzle multiple times. The series of all events associated with a student–puzzle pair that occurred in temporal proximity is called *attempt* (or, sometimes, *session*—both of these terms are also often used for other related concepts in the literature). Most analyses require only an aggregate summary of the attempts, so it is convenient to have the association of the events to the attempts readily available. Assigning events to the sessions is not completely clear because there is a continuous spectrum of delays between two events. Nevertheless, if the typical solving time for all puzzles in the system does not exceed 30 minutes, then a simple heuristic, such as only starting a new attempt after a delay exceeding 3 hours, is probably sufficient.

The identification of the student and the puzzle. Students typically have a database

Table 2. Information to log about events in a block-based programming learning system.

Attribute	Description / Examples
event ID	Unique identifier of the event.
event type	Attempt start/end, edit, execution, reset, hint, feedback.
event order	Absolute order of the event according to the timestamp.
timestamp	Time when the event occurred (in UTC).
attempt ID	Unique identifier of the student–puzzle interaction.
program	Text representation of the student’s code.
program output	Execution trace as a series of events in the microworld.
correct	Whether the program solves the puzzle.
message	Text of the acknowledgment, error, hint, or feedback.
<i>Student</i>	
student ID	Unique identifier of the student.
student information	E.g. demographic information, preferences, setting.
<i>Puzzle</i>	
puzzle ID	Unique identifier of the current version of the puzzle.
puzzle statement	Microworld, toolbox, starter code, text specification.
puzzle solution	Text representation of a program that solves the puzzle.
puzzle set ID	Unique identifier of the puzzle set containing this puzzle.
puzzle set order	Order of the puzzle set within the learning system.
puzzle order	Order of the puzzle within the puzzle set.
<i>Context</i>	
attempt source	E.g. recommendation button on the homepage.
local timestamp	Local time for the student when the event occurred.
time from start	The number of seconds from the start of the attempt.
tools	Names and versions of used tools, models, and algorithms.
experiment	Specification of the experiment and condition.

ID that can be directly used for this purpose. For puzzles, the choice is more difficult. They can also be identified by a simple ID, but this is sufficient only if the puzzle information is stored in a “append-only” table, i.e., any change to the puzzle—e.g., adding a scaffolding, restating a hint, or moving to another puzzle set—results in a new puzzle ID. If this strict policy is not followed, it may be useful to log complete information about the puzzle, not just ID.

The student’s code. The technical aspect is discussed in Section 5.1.3. A conceptual aspect, specific to block-based programming, is the treatment of disconnected blocks. A naive approach is to log only blocks connected to the “start” block (or some implicit “first block”) since only these blocks are potentially relevant for the execution. However, our experience has shown that this approach leads to a nontrivial loss of useful information—students often disconnect code and perform many edits on the disconnected parts. Therefore, we recommend to log all parts of the code and be explicit about which code will be performed and in which order.

Results. The basic information that needs to be logged is the correctness of the

code (i.e., whether it leads to a correct solution of the puzzle). Additionally, we may also want to log the program output, e.g., as a series of events in the microworld. In principle, this is redundant since these effects can be reconstructed from the code (unless there is some nondeterministic aspect in the microworld). Nevertheless, explicitly logging the effects can considerably simplify and accelerate some analyses, and avoids the potential issues that could arise in the case of bug fixes or other changes in the implementation of the microworld.

Contextual data. It is advisable to log all available contextual data that could influence student behavior, such as how the student got to the puzzle (e.g., through a recommendation on the home page), local time in the student’s timezone, time from the start of the attempt, specification of the tools, models, and algorithms used in the learning system (including version numbers), and possibly current online experiment and experimental condition the student is assigned to.

5.1.3. Format of the Data

Concerning the technical aspect of logging the data, a good starting point is a proposal by Price et al. (2019b), who outline a standard for logging general programming data: one main CSV table of events, a table with metadata (e.g., whether the events are ordered and how the code states are represented), and optional link tables, for example for storing puzzles and code states (only the ID of the code state is stored in the main event table).

Specifically for block-based programming, there are several choices for logging the code itself:

- A general format for representation of hierarchical information, such as XML or JSON. XML is used internally in the Blockly implementation, but it is not well readable and it is unnecessarily long.
- A textual programming language, such as Python. Ideally, it should resemble the text written on the blocks. The advantage over XML or JSON is higher readability and similarity to what the students see; furthermore, interpreting the code can be done just by providing definitions of the actions as functions in the chosen programming language.
- An exercise-specific compact code. For instance, to encode programs in RoboMission, we use a single character per node of the abstract syntax tree, resulting in codes like `R4{fs}` (*repeat 4 times: fly and shoot*). Such codes are space-efficient, relatively readable, and simple to analyze. This approach, however, requires more time to design and does not generalize to more complex programs; already functions and variables become cumbersome to express.

5.2. Difficulty and Complexity

The terms *difficulty* and *complexity* are sometimes used interchangeably, but it is useful to distinguish them. In accordance with previous works (Campbell, 1988; Liu and Li, 2012; Sheard et al., 2013), we use these terms with the following meaning:

- Puzzle *complexity* is an intrinsic puzzle characteristic, which aggregates puzzle aspects that influence how students solve the puzzle. Complexity is concerned with the structure of the puzzle itself and can be computed without any student data.
- Puzzle *difficulty* describes how hard it is to solve the puzzle for students. For the

computation of puzzle difficulty, we consider only data on student performance (without any details about the internal structure of the puzzle).

The basic analysis of difficulty is concerned with individual puzzles. We can also consider the difficulty analysis with respect to “concepts” (programming concepts, microworld elements)—in this case, it makes sense to consider not just difficulty, but also learning curves (how are students improving).

5.2.1. Difficulty and Complexity Measures for Puzzles

The estimation of difficulty and complexity of individual puzzles is one of the core data analysis methods for educational data. It has many applications, including:

- The analysis of homogeneity of levels: all puzzles at the same level should have a similar difficulty.
- The sequencing of puzzles within levels: commonly, the puzzles are ordered from easier to more difficult.
- The identification of problematic puzzles that need to be removed or changed, e.g., an unintentionally too tricky puzzle.

Basic difficulty measures for programming exercises are failure rate, the median time to solve the puzzle, and the median number of actions like edits, executions, or submits. These simple measures are influenced by the context of the puzzle; e.g., the performance on a puzzle from the first level would probably be much better if the puzzle appeared in the last level instead. There are models based on the Item Response Theory (De Ayala, 2008), which provide estimates of the difficulty that are independent of the group of students who solve the puzzle, even if the students are learning (Pelánek, 2016).

Complexity can be measured using either the puzzle statement or its solution program. Some common complexity measures based on the solution program include the number of blocks (possibly considering the length limit imposed on the student’s solution), the number of programming concepts involved, the number of flow-of-control structures (Alvarez and Scott, 2010), cyclomatic complexity (Whalley and Kasto, 2014), and Halstead complexity measures based on the number of operators and operands (Ihantola and Petersen, 2019). Measures based on the puzzle statement can use microworld features, such as the size of the world, length of the path, and the number of game elements.

The complexity can be influenced by other aspects, such as scaffoldings (e.g., hints, starter code, a toolbox of programming blocks, the name of the problem, and the natural language text specification), non-programming concepts required to solve the puzzle (e.g., computing correct angles in the turtle graphics), and the context in which the puzzle appears (e.g., the concepts already practiced in the previous levels, and the common focus of all the puzzles in the current level). These aspects are rather difficult to account for.

However, just computing basic difficulty measures such as failure rate and median solving time can bring useful insights into how difficult the puzzles are for the students and can reveal puzzles that are much more difficult than expected. Figures 6, 8, and 9 provide examples of this simple yet useful analysis.

Another view can be obtained by comparing two measures. Figure 10A shows a correlation analysis between a difficulty measure (median solving time) and a complexity measure (the number of concepts). The puzzles with a mismatched difficulty and complexity (high distance from the diagonal line) deserve attention. There are

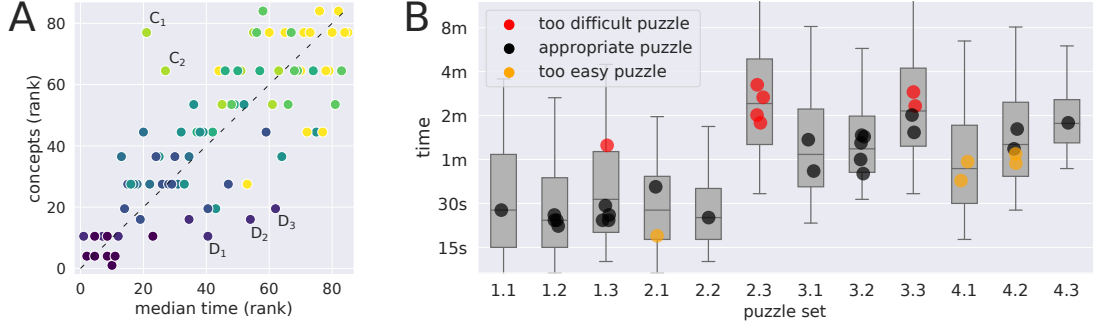


Figure 10. (A) Comparison of difficulty and complexity of puzzles in RoboMission. Colors denote the levels of the puzzles. (B) The difficulty of the puzzles in the context of the surrounding puzzle sets. The boxplots show the distributions of solving times of successful attempts for all puzzles in the given puzzle set. Boxes contain 50% of the attempts, whiskers 90%. The points show median solving times of the puzzles. Puzzles with suspiciously high or low median times are highlighted by red and orange colors. Only the puzzle sets in the first four levels are shown; each level contains three puzzle sets.

two puzzles that are significantly more complex than difficult (labeled as C_1 and C_2 in Figure 10A); these are puzzles that introduce a new concept (*if-else*) and are straightforward to solve with a short program, at least for the students who completed all the previous levels. On the other hand, there are several much more difficult than complex puzzles (labeled as D_1 , D_2 , and D_3); these are “tricky” puzzles, which only require a sequence of commands, but it is difficult to find the correct sequence. These tricky puzzles should be removed from the introductory levels to avoid unnecessary frustrating beginners.

Further insight can be obtained from looking at the difficulty of the puzzles in the context of their placement within the system. As an example, Figure 10B shows the difficulty of the puzzles in RoboMission in the context of the surrounding puzzle sets. This allows us to visually spot potentially problematic puzzles or even whole puzzle sets that are too easy or too difficult. A simple heuristic is used to highlight potential outliers: we compare median times of puzzles in each puzzle set to an expected range $[0.75T, 2T]$ where T is the mean median time in the previous three puzzle sets (with already detected outliers being capped at their upper or lower limits).

5.2.2. Concepts: Difficulty and Learning Curves

The difficulty of the puzzles is influenced by the difficulty of the concepts students need to know to solve the puzzle. These can be broadly divided into programming concepts (e.g., elementary commands, loops, conditions, and integration skills like the use of nested loops), and microworld elements (e.g., obstacles, tokens, and teleports). We can estimate the initial difficulty of each concept when students encounter it for the first time, and how quickly the difficulty decreases with opportunities to practice the concept. The relationship between the number of practice opportunities and the student performance (e.g., success rate or the mean time to solve the puzzle) can be visualized using learning curves (Martin et al., 2011).

This kind of analysis can serve the following purposes:

- Evaluation of the learning environment: checking whether students are actually learning the covered programming concepts.
- Domain modeling: identification of the most important concepts and their relative difficulty, with potential consequences for the user interface, open learner

- modeling, skillometers, and mastery criteria.
- Puzzles authoring: learning curves can reveal concepts that are not sufficiently covered and would benefit from more puzzles for practicing them.

Unfortunately, analyzing the difficulty of concepts is complicated. As there are multiple concepts involved in each puzzle, it is hard to disentangle the contribution of each of them, which is known as the *credit/blame assignment problem* (Nwaigwe et al., 2007; van de Sande and Education, 2016). Furthermore, data collection biases can easily lead to misleading conclusions. For example, Goutte et al. (2018) and Käser et al. (2014) discuss the impact of attrition bias on the learning curves. Another common bias is caused by the ordering of the puzzles: the puzzles are often ordered from the easy to difficult (Čechák and Pelánek, 2019). Therefore, with the increasing number of practice opportunities, the distribution of the puzzles shifts towards more difficult.

Due to these biases, the learning curves might even increase. This is illustrated in Figure 11, which shows three types of learning curves for the data from RoboMission: empirical, idealized, and reweighted. Figure 11A displays *empirical learning curves*, i.e., the relationship between the number of practice opportunities and the mean observed difficulty, measured as the log-transformed time to solve the puzzle. This plot is misleading in two ways: first, it suggests that students are getting worse with more practice opportunities; second, it suggests that some game elements (tokens and teleports) are as difficult as loops. Both issues are caused by not accounting for the difficulty of the other concepts involved in each puzzle.

A common approach to isolate the effects of individual concepts is to fit a predictive student model such as Additive Factors Model (Durand et al., 2017) to the observed data, and then use the parameters of the fitted model to plot the learning curves (Koedinger et al., 2012; Long et al., 2018; Nguyen et al., 2019; Rivers et al., 2016). The result of this approach for the RoboMission data is shown in Figure 11B, which contains *idealized learning curves* for individual concepts according to a fitted Additive Factors Model. Effenberger et al. (2020) describes how to compute these curves in detail. The second issue is alleviated: the plot conveys that the game elements are easier than the programming concepts, as would one expect. The first issue is, however, still present: for some concepts, the time to solve the puzzles does not decrease with the number of opportunities. This illustrates a weak point of the idealized learning curves: if the training data are biased (and in online learning systems, they inevitably are), the model and the resulting curves will be biased as well.

One possible remedy for the puzzle ordering bias is to reweight the attempts to make the contribution of each puzzle the same at each practice opportunity (Figure 11C). This partially removes the puzzle ordering bias but introduces additional noise by putting a high weight on some of the attempts. Furthermore, the reweighted empirical curves again suffer from the issue of correlated concepts—not accounting for the difficulty of the other concepts involved in the puzzle. A robust method for computing plausible unbiased learning curves would be extremely helpful, so we consider this type of analysis to be an important direction for further research.

5.3. Code Analysis

So far, we have discussed analyses that consider data across puzzles. Now we analyze data for a single puzzle. In some cases, we consider only the final (correct) submissions; in other cases, we also consider the unfinished ones. These data are suitable mostly for unsupervised techniques, which try to find informative patterns in data.

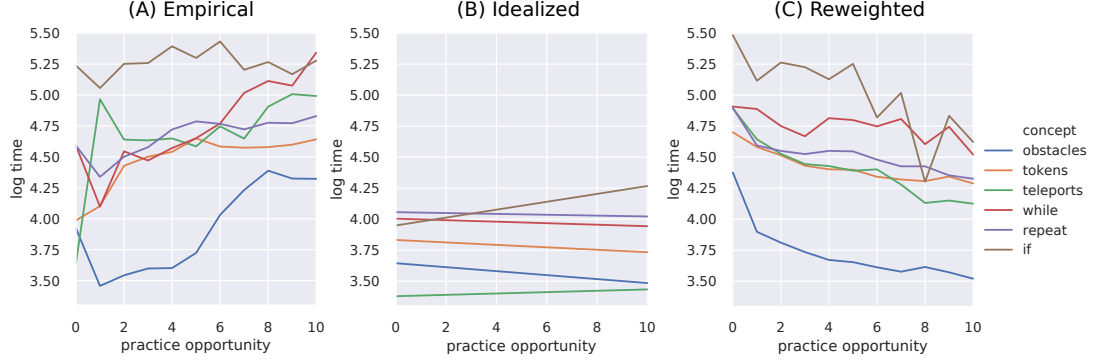


Figure 11. Learning curves for six concepts in RoboMission. (A) Empirical learning curves (mean observed log-transformed time). (B) Idealized learning curves for a fitted extension of the Additive Factors Model. (C) Empirical learning curves computed from attempts reweighted in such a way that the contribution of each puzzle is the same at each practice opportunity. All these learning curves are implausible due to biases in the data; see text for details.

The general purpose of this type of analysis:

- “Debugging” puzzles: for example, we can detect puzzles with unexpected solutions (easier than intended) or attempts (suggesting some misleading aspect in the presentation of a puzzle) and then remove or change these puzzles.
- Improving the sequencing of puzzles: solution patterns may inform the ordering of puzzles within the learning system, e.g., to achieve variability or support transfer.
- Determining priorities for the creation of feedback messages, hints, and explanations for common mistakes and misconceptions.
- Getting impulses for creating new puzzles: common mistakes and misconceptions revealed by the analysis of data can provide inspiration for creating new puzzles to address these problems.

Similar types of analysis have been studied for standard programming languages like Python, e.g., by Chow et al. (2017); Glassman et al. (2014); Piech et al. (2012); Rivers and Koedinger (2017).

5.3.1. Length of Solutions

A basic summary of a solution to a programming problem is the code length. An advantage of block-based programming (compared to text-based programming) is that it is quite clear what “code length” means—a natural measure is the “number of blocks”. Moreover, in block-based programming, the length of a solution is a reasonable proxy for the “quality” of the code—in typical puzzles, a smaller number of blocks implies the use of better coding practices (e.g., capturing patterns using loops).

It is thus useful to analyze the distribution of lengths of solutions. The purpose of this analysis can be:

- To make design decisions about the use of limits on the number of blocks, e.g., whether to use them, how to set their values.
- The identification of puzzles with the occurrence of inefficient solutions, in order to provide clarification, feedback, hints, explanations, or scaffoldings to help students write better code.

A specific example of this type of analysis is given in Figure 12, which shows histograms of lengths of solutions for selected turtle graphics problems. The results in this figure show:

- For simple problems like Letter M, all solutions are short; there is basically no reasonable way to write a long solution.
- For problems that contain “limited repetition” (Cross, Compass, Pentagon), the histograms clearly show that the majority of solvers used compact solutions with loops, but there is a nontrivial group of students with long solutions (i.e., a sequence of commands without loops). A limit on the number of blocks could be useful for these examples.
- For the Diamond example, solutions without (nested) loops are simply too long—they do not occur in the data. Nearly all students that solve this problem use the same, intended solution.

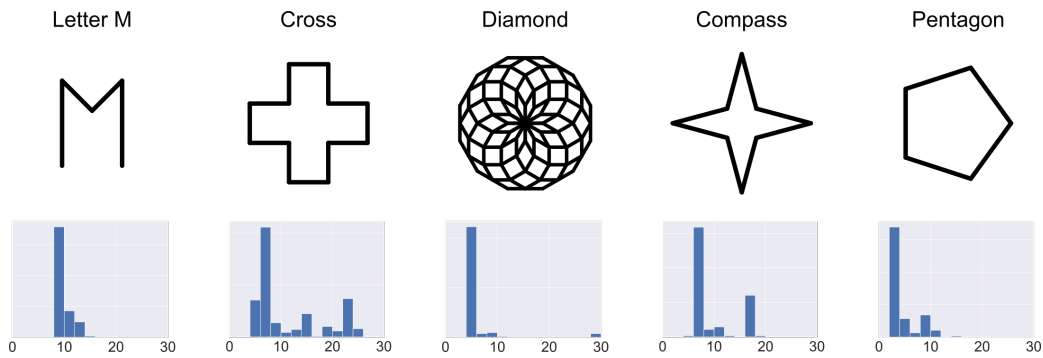


Figure 12. Turtle graphics: lengths of students’ solutions.

5.3.2. Clustering of Codes

To get more detailed insight into solutions, we may look at specific codes submitted by students. In a realistic learning system, the number of logged attempts is very high, so it is not feasible to analyze individual submissions. For analysis, it is useful to perform some clustering of submitted codes, e.g., by merging “equivalent” or “similar” codes, where similarity can be defined by tree edit distance. Once we have the clusters, we can analyze their sizes or manually explore codes from the largest clusters.

The purpose of this type of analysis can be:

- The identification of unintended solutions. It can easily happen that a puzzle has an easier solution than the author intended. Such a puzzle should often be updated or moved to a different level.
- The analysis of common wrong answers to find out specific steps that students struggle with. Identification of these steps can be used to develop hints, feedback, scaffoldings, or new puzzles that specifically address the identified problems.
- Ideas for new puzzles. Analysis of student solutions can show unexpected solutions or interesting wrong attempts that can inspire the creation of new puzzles.

We provide several specific examples of this kind of analysis and describe the obtained insights. Figure 13 shows a basic type of this analysis for the Arrows game. In this case, we consider only correct solutions and consider clusters of completely identical solutions. The example shows two solutions: the first one is the solution in-

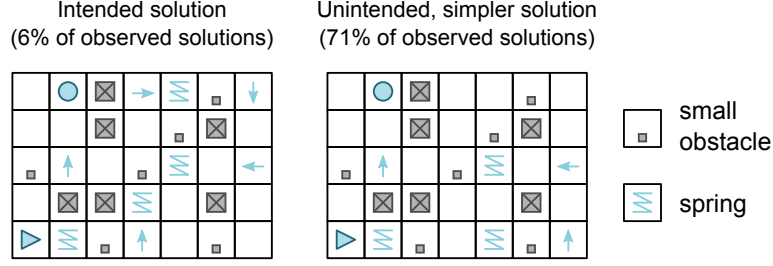


Figure 13. Arrows game – an example of an unintended simpler solution overlooked by the puzzle author.



Figure 14. Turtle graphics: two common wrong solutions to the Compass problem.

tended by the author; the second one shows the most common solution. In this case, the author clearly overlooked a simpler solution—a common case in systems with a large number of puzzles.

For turtle graphics, clusterings can be done with respect to code or the resulting images. Each of these approaches provides different insights. For the *Cross* example, the clustering with respect to images is not very interesting, whereas codes are quite varied. Even the basic histograms of lengths of code in Figure 12 shows that there is a wide variability of solutions. More detailed analysis shows several distinct clusters of solutions (and additionally also different “hybrids” between these basic solutions):

- solutions without loops, consisting of a long sequence of commands,
- solutions with one loop repeated four times (drawing a U shape),
- solutions with two nested loops.

For the *Compass* example, the clustering by resulting image is more interesting, as it shows two common wrong solutions (Figure 14), both indicating problems with the computation of correct angles. Our experience is that in the case of turtle graphics, wrong solutions can often provide inspiration for new problems (e.g., the first mistake, after some tuning, can lead to an interesting example).

In the *Pentagon* example, clustering by image reveals common mistakes in angles: instead of the correct value 72, students tend to use 60, 71, 73, and 75. This suggests that many students solve the problem by iterative debugging rather than deriving the correct value from the properties of polygons. From the analysis, we can conclude that a message with an explanation (how to compute the correct angle without guessing) would be useful for a large portion of students.

5.3.3. Interaction Networks

When we log student actions at the level of individual edits, we can consider the “state” of the code after each edit and analyze the whole state space with student transitions between states. In the educational data mining literature, this approach has been previously discussed also under the name “interaction networks” (Eagle et al., 2012; Johnson et al., 2013).



Figure 15. Examples of interaction networks (RoboMission). Nodes correspond to programs (f = forward, l = left, r = right, Rx = repeat x times); some node descriptions are omitted for readability. The size of nodes and the width of edges corresponds to the frequency of visits by students. Only most often visited nodes and edges are shown. Blue denotes the initial state, green denotes correct solutions, yellow denotes nodes on a direct path towards a solution. For better legibility, blue and green nodes are also highlighted by arrows.

The purpose of this type of analysis can be:

- Insight into the solution process used by students. This can be useful feedback for puzzle authors and an impulse for the design of new puzzles.
- The identification of “dead ends”, i.e., states where many students get stuck. Such states may require hints, explanations, or even revisions of a puzzle.
- Facilitation of design decisions about limits, e.g., when we notice that a common “dead end” is caused by a too strict limit on the number of blocks.

Figure 15 provides examples of interaction networks for RoboMission. In the example on the left, we see that the puzzle has several solutions; the wrong attempts typically branch out of the correct paths, and students spend most of their time quite close to one of the correct paths. In the example on the right, there is basically only one solution (with minor variation). There is quite a distinct part of the state space, where students start their code by the “shooting” (s) action, which does not lead to a solution. Given this insight, we can check whether this was intended by the puzzle author, and consider whether it may be useful to give students a hint (“start by repeat, not by shoot”).

6. Discussion

In this work, we have provided a thorough discussion of microworld and puzzle design guidelines based on literature, specific examples of microworlds and puzzles, and our experiences with the development and data analysis. We conclude with several observations and note the gaps in the current state-of-the-art.

6.1. Design Guidelines

Rather than building one complex microworld that would enable demonstration and learning of a wide range of programming concepts, it seems better to design several

simple microworlds with complementary strengths. For example, turtle graphics provides a natural setting for the practice of iteration and has connections to geometry, but it is not suitable for the practice of conditional commands. A robot on a grid microworlds can be realized in many different ways, placing the focus, for example, on conditions, recursion, or parallelism. When well designed, even relatively simple and clearly focused microworlds can provide sufficient complexity for interesting puzzles—in all microworlds that we discussed, we have developed at least 50 puzzles of varying difficulty.

An important design step is the choice of blocks and limits. Blocks can be textual or pictorial, with different types of user interface and menu organization. Limits can be specified with respect to different criteria and with varying strictness. It is not possible to provide universal guidelines for these choices as they depend on the particular use case (e.g., target audience). It is, however, important to carefully consider these choices and not just use some default (e.g., provided by the easily available Blockly implementation).

A key theme in the design of block-based programming microworlds is the use of scaffolding principles. These can be applied on several levels:

- The use of scaffolding blocks, which are simplified with respect to full commands (e.g., “turn left 90 degrees” as opposed to “turn left X degrees”).
- The use of scaffolding puzzles, which contain the skeleton of the code and students have to fill just a few details.
- Dividing puzzles into puzzle sets that are homogeneous with respect to the used block menu and programming concepts.
- Ordering with respect to difficulty—both of the puzzle sets and of puzzles within the sets.

6.2. Data Analysis Techniques

It is highly beneficial to approach the development of microworlds and puzzles in an incremental fashion based on data analysis. In our overview of techniques, we illustrated a wide variety of techniques and their potential applications. The relevant techniques include the analysis of the difficulty and complexity of puzzles, the analysis of learning curves, the clustering of solutions, and the analysis of interaction networks. The purposes of analysis include the identification of problematic puzzles, getting inspiration for the creation of new puzzles, getting impulses for the organization and sequencing of puzzles and puzzle sets, and getting insight into student behavior in order to create suitable hints and feedback messages.

Our discussion of applicable techniques is not completely exhaustive. We have focused on techniques that are specific to block-based programming rather than to the organization of the system (e.g., system of levels, personalization aspects). For practical application, it is useful to apply additional data analysis techniques that are more specific to the system organization, e.g., engagement analysis, for which the analysis and its interpretation is done with respect to the specific mechanics of progression through levels.

6.3. Directions for Further Research

As our overview shows, there are many data analysis techniques that can be used to analyze data from block-based programming environments. Nevertheless, there is

still potential for significant improvement. As a particularly important direction, we consider the analysis of concepts and their learning curves (discussed in Section 5.2.2). This is a rather complicated topic—it is not clear what are the most important “concepts” in block-based programming, there are nontrivial interactions among concepts, and observed data include significant biases. However, this type of analysis can bring beneficial insights both for the design and evaluation of block-based programming environments and thus presents an important research direction. From the other techniques that we discussed, we highlight interaction networks since we believe that they have a higher potential for insights than currently utilized.

In our discussion of design choices, we mentioned some soft recommendations that are based on our experience and common choices made in other realization of block-based microworlds but are not supported by research studies. It may thus be meaningful to test them explicitly and to explore potential nuances and caveats. Such design recommendations and claims include: “3D world does not add anything fundamental, 2D world is preferable” (Section 3.2), “variables and functions with parameters are not easy to use in block-based programming and should not be a fundamental part of microworlds” (Section 3.3), or “limits are preferable to alternative ways of enforcing the use of complex blocks” (Section 3.4).

References

- Aivaloglou, E. and Hermans, F. (2016). How kids code and how we know: An exploratory study on the scratch repository. In *Proceedings of International Computing Education Research*, pages 53–61. ACM.
- Aleven, V., McLaughlin, E. A., Glenn, R. A., and Koedinger, K. R. (2016). *Handbook of research on learning and instruction*, chapter Instruction based on adaptive learning technologies, pages 522–559. Routledge.
- Alvarez, A. and Scott, T. A. (2010). Using student surveys in determining the difficulty of programming assignments. *Journal of Computing Sciences in Colleges*, 26(2):157–163.
- Baker, R. S. (2016). Stupid tutoring systems, intelligent humans. *International Journal of Artificial Intelligence in Education*, 26(2):600–614.
- Bau, D., Bau, D. A., Dawson, M., and Pickens, C. (2015). Pencil code: block code for a text world. In *Proceedings of Interaction Design and Children*, pages 445–448. ACM.
- Bau, D., Gray, J., Kelleher, C., Sheldon, J., and Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, 60(6):72–80.
- Becker, B. W. (2001). Teaching CS1 with karel the robot in java. *ACM SIGCSE Bulletin*, 33(1):50–54.
- Bertz, M. D. (1997). Billiards in the classroom: Learning physics with microworlds. *NASSP Bulletin*, 81(592):31–38.
- Boyce, A. and Barnes, T. (2010). Beadloom game: using game elements to increase motivation and learning. In *Proceedings of Foundations of Digital Games*, pages 25–31. ACM.
- Brown, N. C. and Altadmri, A. (2014). Investigating novice programming mistakes: educator beliefs vs. student data. In *Proceedings of International Computing Education Research*, pages 43–50. ACM.
- Brown, N. C. C., Kölling, M., McCall, D., and Utting, I. (2014). Blackbox: a large scale repository of novice programmers’ activity. In *Proceedings of ACM technical symposium on Computer science education*, pages 223–228. ACM.
- Browne, C. (2015). The nature of puzzles. *Game & Puzzle Design*, 1(1):23–34.
- Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. (1997). Mini-languages: a way to learn programming principles. *Education and Information Technologies*, 2(1):65–83.

- Campbell, D. J. (1988). Task complexity: A review and analysis. *Academy of management review*, 13(1):40–52.
- Caspersen, M. E. and Christensen, H. B. (2000). Here, there and everywhere – on the recurring use of turtle graphics in CS1. In *ACM International Conference Proceeding Series*, volume 8, pages 34–40.
- Čechák, J. and Pelánek, R. (2019). Item ordering biases in educational data. In *Proceedings of Artificial Intelligence in Education*, pages 48–58. Springer.
- Chen, C., Haduong, P., Brennan, K., Sonnert, G., and Sadler, P. (2019). The effects of first programming language on college students’ computing attitude and achievement: a comparison of graphical and textual languages. *Computer Science Education*, 29(1):23–48.
- Chow, S., Yacef, K., Koprinska, I., and Curran, J. (2017). Automated data-driven hints for computer programming students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization*, pages 5–10. ACM.
- Cooper, S., Dann, W., and Pausch, R. (2000). Alice: a 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116.
- De Ayala, R. (2008). *The theory and practice of item response theory*. The Guilford Press.
- Durand, G., Goutte, C., Belacel, N., Bouslimani, Y., and Leger, S. (2017). Review, computation and application of the additive factor model (AFM). Technical report, Tech. Report 23002483. National Research Council Canada.
- Eagle, M., Johnson, M., and Barnes, T. (2012). Interaction networks: Generating high level hints based on network community clustering. In *Proceedings of Educational Data Mining*. ERIC.
- Effenberger, T. (2019). Blockly programming dataset. In *Proceedings of Educational Data Mining in Computer Science Education Workshop*.
- Effenberger, T. and Pelánek, R. (2018). Towards making block-based programming activities adaptive. In *Proceedings of Learning at Scale*, page 13. ACM.
- Effenberger, T., Pelánek, R., and Čechák, J. (2020). Exploration of the robustness and generalizability of the additive factors model. In *Proceedings of the Tenth International Conference on Learning Analytics & Knowledge*, pages 472–479.
- Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., and Resnick, M. (2013). Designing scratchjr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*, pages 1–10.
- Fraser, N. (2015). Ten things we’ve learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50. IEEE.
- GhasemAghaei, R., Arya, A., and Biddle, R. (2017). Affective walkthroughs and heuristics: Evaluating minecraft hour of code. In *International Conference on Learning and Collaboration Technologies*, pages 22–40. Springer.
- Glassman, E. L., Singh, R., and Miller, R. C. (2014). Feature engineering for clustering student solutions. In *Proceedings of Learning at Scale*, pages 171–172. ACM.
- Goutte, C., Durand, G., and Léger, S. (2018). On the learning curve attrition bias in additive factor modeling. In *Proceedings of Artificial Intelligence in Education*, pages 109–113. Springer.
- Gouws, L. A., Bradshaw, K., and Wentworth, P. (2013). Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of Innovation and technology in computer science education*, pages 10–15. ACM.
- Grover, S. and Basu, S. (2017). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of ACM SIGCSE technical symposium on computer science education*, pages 267–272. ACM.
- Grover, S., Basu, S., Bienkowski, M., Eagle, M., Diana, N., and Stamper, J. (2017). A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments. *ACM Transactions on Computing Education*, 17(3):14.
- Harpstead, E. and Aleven, V. (2015). Using empirical learning curve analysis to inform design

- in an educational game. In *Proceedings of Computer-Human Interaction in Play*, pages 197–207. ACM.
- Hicks, A., Dong, Y., Zhi, R., Cateté, V., and Barnes, T. (2015). Bots: Selecting next-steps from player traces in a puzzle game. In *EDM (Workshops)*.
- Hicks, A. G. (2016). *Design Tools and Data-Driven Methods to Facilitate Player Authoring in a Programming Puzzle Game*. PhD thesis, North Carolina State University.
- Hicks, D., Eagle, M., Rowe, E., Asbell-Clarke, J., Edwards, T., and Barnes, T. (2016). Using game analytics to evaluate puzzle design and level progression in a serious game. In *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge*, pages 440–448. ACM.
- Horn, B., Hoover, A. K., Barnes, J., Folaajimi, Y., Smith, G., and Harteveld, C. (2016). Opening the black box of play: Strategy analysis of an educational game. In *Proceedings of Computer-Human Interaction in Play*, pages 142–153. ACM.
- Hoyles, C., Noss, R., and Adamson, R. (2002). Rethinking the microworld idea. *Journal of educational computing research*, 27(1):29–53.
- Hromkovič, J., Kohn, T., Komm, D., and Serafini, G. (2016). Combining the power of python with the simplicity of logo for a sustainable computer science education. In *Proceedings of International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 155–166. Springer.
- Hundhausen, C. D., Olivares, D. M., and Carter, A. S. (2017). Ide-based learning analytics for computing education: a process model, critical review, and research agenda. *ACM Transactions on Computing Education*, 17(3):11.
- Ihantola, P. and Petersen, A. (2019). Code complexity in introductory programming courses. In *Proceedings of Hawaii International Conference on System Sciences*.
- Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S. H., Isohanni, E., Korhonen, A., Petersen, A., Rivers, K., et al. (2015). Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, pages 41–63. ACM.
- Iii, B. P., Hicks, A., and Barnes, T. (2014). Generating hints for programming problems using intermediate output. In *Proceedings of Educational Data Mining*.
- Jarušek, P. and Pelánek, R. (2011). What determines difficulty of transport puzzles? In *Proceedings of Florida Artificial Intelligence Research Society Conference*, pages 428–433. AAAI Press.
- Johnson, M., Eagle, M., and Barnes, T. (2013). Invis: An interactive visualization tool for exploring interaction networks. In *Proceedings of Educational Data Mining*.
- Jumaat, N. F. and Tasir, Z. (2014). Instructional scaffolding in online learning environment: A meta-analysis. In *Proceedings of Teaching and Learning in Computing and Engineering*, pages 74–77. IEEE.
- Käser, T., Koedinger, K. R., and Gross, M. (2014). Different parameters - same prediction: An analysis of learning curves. In *Proceedings of Educational Data Mining*, pages 52–59.
- Kelleher, C. and Hnin, W. (2019). Predicting cognitive load in future code puzzles. In *Proceedings of Human Factors in Computing Systems*, pages 257:1–257:12. ACM.
- Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137.
- Koedinger, K. R., McLaughlin, E. A., and Stamper, J. C. (2012). Automated student model improvement. In *Proceedings of Educational Data Mining*. ERIC.
- Kotovsky, K., Hayes, J. R., and Simon, H. A. (1985). Why are some problems hard? evidence from tower of hanoi. *Cognitive psychology*, 17(2):248–294.
- Linehan, C., Bellord, G., Kirman, B., Morford, Z. H., and Roche, B. (2014). Learning curves: analysing pace and challenge in four successful puzzle games. In *Proceedings of Computer-human interaction in play*, pages 181–190. ACM.
- Liu, P. and Li, Z. (2012). Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics*, 42(6):553–568.

- Lomas, J. D., Koedinger, K., Patel, N., Shodhan, S., Poonwala, N., and Forlizzi, J. L. (2017). Is difficulty overrated?: The effects of choice, novelty and suspense on intrinsic motivation in educational games. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 1028–1039. ACM.
- Long, Y., Holstein, K., and Alevan, V. (2018). What exactly do students learn when they practice equation solving?: refining knowledge components with the additive factors model. In *Proceedings of Learning Analytics and Knowledge*, pages 399–408. ACM.
- Martin, B., Mitrovic, A., Koedinger, K. R., and Mathan, S. (2011). Evaluating and improving adaptive educational systems with learning curves. *User Modeling and User-Adapted Interaction*, 21(3):249–283.
- Mitchell, M. (2009). *Complexity: A guided tour*. Oxford University Press.
- Nakamura, J. and Csikszentmihalyi, M. (2014). The concept of flow. In *Flow and the foundations of positive psychology*, pages 239–263. Springer.
- Newell, B. (1988). *Turtle Confusion: Logo Puzzles and Riddles: Barry Newell*. Curriculum Development Centre.
- Nguyen, H., Wang, Y., Stamper, J., and McLaren, B. M. (2019). Using knowledge component modeling to increase domain understanding in a digital learning game. In *Proceedings of Educational Data Mining*.
- Nwaigwe, A., Koedinger, K. R., Vanlehn, K., Hausmann, R., and Weinstein, A. (2007). Exploring alternative methods for error attribution in learning curves analysis in intelligent tutoring systems. *Frontiers in Artificial Intelligence and Applications*, 158:246.
- O’Kelly, J. and Gibson, J. P. (2006). Robocode & problem-based learning: a non-prescriptive approach to teaching programming. *ACM SIGCSE Bulletin*, 38(3):217–221.
- Papadopoulos, Y. and Tegos, S. (2012). Using microworlds to introduce programming to novices. In *Proceedings of Panhellenic Conference on Informatics*, pages 180–185. IEEE.
- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.
- Parsons, D. and Haden, P. (2006). Parson’s programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163.
- Pasternak, E., Fenichel, R., and Marshall, A. N. (2017). Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 21–24. IEEE.
- Pattis, R. E. (1981). *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition.
- Pelánek, R. (2011). Difficulty rating of sudoku puzzles by a computational model. In *Proceedings of Florida Artificial Intelligence Research Society Conference*, pages 434–439. AAAI Press.
- Pelánek, R. (2016). Applications of the Elo rating system in adaptive educational systems. *Computers & Education*, 98:169–179.
- Pelánek, R. (2017). Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques. *User Modeling and User-Adapted Interaction*, 27(3):313–350.
- Piech, C., Sahami, M., Koller, D., Cooper, S., and Blikstein, P. (2012). Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 153–160. ACM.
- Price, T. W. and Barnes, T. (2015). Comparing textual and block interfaces in a novice programming environment. In *Proceedings of International Computing Education Research*, pages 91–99. ACM.
- Price, T. W., Dong, Y., Zhi, R., Paaßen, B., Lytle, N., Cateté, V., and Barnes, T. (2019a). A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education*, pages 1–28.
- Price, T. W., Hovemeyer, D., Rivers, K., Becker, B. A., et al. (2019b). Progsnap2: A flexible format for programming process data. In *Proceedings of Learning Analytics & Knowledge Conference*.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Miller, A., Rosenbaum, E., Silver, J. S., Silverman, B., et al. (2009). Scratch: Programming

- for all. *Communications of the ACM*, 52(11):60–67.
- Rieber, L. P. (1992). Computer-based microworlds: A bridge between constructivism and direct instruction. *Educational technology research and development*, 40(1):93–106.
- Rieber, L. P. (1996). Seriously considering play: Designing interactive learning environments based on the blending of microworlds, simulations, and games. *Educational technology research and development*, 44(2):43–58.
- Rivers, K., Harpstead, E., and Koedinger, K. R. (2016). Learning curve analysis for programming: Which concepts do students struggle with? In *Proceedings of International Computing Education Research*, pages 143–151.
- Rivers, K. and Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64.
- Romero, C. and Ventura, S. (2013). Data mining in education. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(1):12–27.
- Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Pearson Education Limited.
- Schell, J. (2019). *The Art of Game Design: A book of lenses*. AK Peters/CRC Press.
- Shaker, N., Togelius, J., and Nelson, M. J. (2016). *Procedural content generation in games*. Springer.
- Sheard, J., Carbone, A., Chinn, D., Clear, T., Corney, M., D’Souza, D., Fenwick, J., Harland, J., Laakso, M.-J., Teague, D., et al. (2013). How difficult are exams?: a framework for assessing the complexity of introductory programming exams. In *Proceedings of Australasian Computing Education Conference*, pages 145–154. Australian Computer Society, Inc.
- Shute, V. J., Sun, C., and Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, 22:142–158.
- Tessler, J., Beth, B., and Lin, C. (2013). Using cargo-bot to provide contextualized learning of recursion. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 161–168.
- Vahldick, A., Mendes, A. J., and Marcelino, M. J. (2014). A review of games designed to improve introductory computer programming competencies. In *Proceedings of IEEE Frontiers in Education*, pages 1–7. IEEE.
- van de Sande, B. and Education, P. (2016). Learning curves for problems with multiple knowledge components. In *Proceedings of Educational Data Mining*, pages 523–526.
- Weintrop, D. (2019). Block-based programming in computer science education. *Communications of the ACM*, 62(8):22–25.
- Weintrop, D., Shepherd, D. C., Francis, P., and Franklin, D. (2017). Blockly goes to work: Block-based programming for industrial robots. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 29–36. IEEE.
- Weintrop, D. and Wilensky, U. (2015). The challenges of studying blocks-based programming environments. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 5–7. IEEE.
- Weintrop, D. and Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education*, 18(1):3.
- Whalley, J. and Kasto, N. (2014). How difficult are novice code writing tasks?: A software metrics approach. In *Proceedings of Australasian Computing Education Conference*, pages 105–112. Australian Computer Society, Inc.
- Wilson, C. (2015). Hour of code—a record year for computer science. *ACM Inroads*, 6(1):22–22.
- Xinogalos, S. (2012). An evaluation of knowledge transfer from microworld programming to conventional programming. *Journal of Educational Computing Research*, 47(3):251–277.
- Xinogalos, S., Satratzemi, M., and Dagdilelis, V. (2006). An introduction to object-oriented programming with a didactic microworld: objectkarel. *Computers & Education*, 47(2):148–171.
- Xu, Z., Ritzhaupt, A. D., Tian, F., and Umapathy, K. (2019). Block-based versus text-based

programming environments on novice student learning outcomes: a meta-analysis study.
Computer Science Education, 29(2-3):177–204.