Pyrates: Design and Evaluation of a Serious Game Aimed at Introducing Python Programming and Easing the Transition from Blocks

Matthieu Branthôme

Pyrates: Design and Evaluation of a Serious Game Aimed at Introducing Python Programming and Easing the Transition from Blocks

MATTHIEU BRANTHÔME*, University of Western Brittany, France

This article reports on a design-based research study centered on the conception and the assessment of the *Pyrates* application. This online serious game aims at introducing Python programming to K12 students while easing the transition from block-based to text-based languages. After we present the various aspects underlying the block-to-text transition as well as the related existing applications, we describe the design of *Pyrates*. Firstly, we built the levels of the game in order to deal with the different fundamental concepts of programming in a constructivist approach. Next, we were inspired by advantageous characteristics of block-based programming editors to create the editing environment of *Pyrates*. In order to assess this conception, we tested the application in eight classrooms with 240 French 14-15 years old students. Students' activity traces have been collected and were augmented by a qualitative online survey. By analysing this data set, we showed that the levels' design generally allows to apprehend the targeted concepts consistently with the constructivist principles. Regarding the editing environment, we established that it supports the block-to-text transition in several aspects: concept transposition (general models and illustrative examples), reduction of errors (beginners aware syntax analyser), command catalog (programming memo for discovery and syntax reference), and program composition (copy button which limits keyboarding). Finally, *Pyrates*, which has already been played over 140,000 times, offers practitioners an environment that facilitates the transition from blocks to text, as well as a serious game to master the fundamental concepts of Python programming, and novel avenues to follow for tool designers.

CCS Concepts: • **Social and professional topics** → **CS1**; **K-12 education**; • **Applied computing** → **Interactive learning environments**.

Additional Key Words and Phrases: block-based programming, CS1, design-based research, learning analytics, Python, secondary education, serious game, Scratch, text-based programming.

ACM Reference Format:
Matthieu Branthôme. 2024. Pyrates: Design and Evaluation of a Serious Game Aimed at Introducing Python Programming and Easing the Transition from Blocks. *ACM Trans. Comput. Educ.* 24, 1, Article 12 (February 2024), 25 pages. https://doi.org/10.1145/3639061

1 INTRODUCTION

Over the years, block programming has become one of the preferential modalities for introducing computer coding to younger children [9]. Research has demonstrated the advantages of this approach over the traditional introduction using text-based languages [5, 35, 52]. However, in

*This article is an expanded version of a paper that was published in the proceedings of the 17th European Conference on Technology Enhanced Learning [13]. It contains an extension of the literature review on the bock-to-text transition, new results about: Scratch and Python students' previous knowledge, the design of the different levels of Pyrates, and the overall perception of the game by students.

Author's address: Matthieu Branthôme, matthieu.branthome@univ-brest.fr, University of Western Brittany, Brest, France.

1946-6226/2024/02-ART12 $15.00
https://doi.org/10.1145/3639061

higher education, textual programming is still predominantly employed for more complex computer science training. This is even more true in the industry, where languages like Python and Java are widely used [36]. Thus, students who started programming with blocks in their early school years might need to eventually shift to text-based programming. How could they be supported in this change? This is one of the open questions in the research field of computer science education [49, 29, 28, 53].

A strategy for assisting students is to design digital tools that facilitate the transition between these two programming modalities. These bridging applications are meant to be used transitionally before switching permanently to full text coding. The *Pyrates* online application [37, 38] was developed with this goal. It is, in the first place, a serious game [3] aiming at introducing the Python textual language to high school students. As such, it provides learning material and a set of assignments allowing to acquire the fundamental concepts of programming in a constructivist approach [34]. Moreover, it is also an editing environment designed by taking inspiration from block-based programming editors intending to benefit from their advantages.

This contribution focuses on the evaluation of both aspects of this design. Hence, based on classroom testing, the research questions we address are:

- RQ1: Do the *Pyrates*'s levels enable students to learn the targeted programming concepts consistently with the constructivist principles?
- RQ2: Do students use the features of the editing environment? How do they employ them? How do they judge them regarding clarity and utility?
- RQ3: What is the overall perception of the game design by the students concerning the handling, the difficulty, the playfulness, and the motivation?

The work presented in this article follows the principles of design-based research [47]. It draws on available literature and existing solutions, aims to meet a well-defined practical goal, and is evaluated in real-life contexts using systematic methods based on qualitative and quantitative data collection and analysis.

The rest of this paper is structured as follows. Section 2 expands on the state-of-the-art related to block-to-text transition and existing applications. Then, Section 3 describes the design of *Pyrates* both in terms of the game's levels and the editing environment. Next, Section 4 presents the methodology adopted to evaluate this design and Section 5 exposes and discusses the ensuing results. Lastly, Section 6 concludes and gives some perspectives to this work.

2 STATE-OF-THE-ART

This section is divided into two parts. First, the results of scientific works analysing the main differences between block-based and text-based environments are summarized. Secondly, existing applications designed to support block-to-text transition or to learn Python programming through serious games are presented.

2.1 Block-to-text transition

Programming with a block-based language versus a text-based language is different in many ways but also shares some similarities. These variations are mainly related to two aspects: the intrinsic constitution of the languages and the environments used to edit programs. The following is a presentation of these different gaps and their consequences for transitioning learners. The elements related to the intrinsic characteristics of the languages are first exposed.

2.1.1 Programming paradigms. The progression from block to text involves a change in programming paradigm. The blocks allow object-oriented programming as the scripts are carried out by objects (characters, robots, etc.) and act by modifying their attributes (position, appearance, sounds,

etc.) [12, 28, 50]. Moreover, block languages implement event-based and parallel programming approaches. Indeed, the executions are triggered by events (click on a zone, key pressed, message received, etc.) and this can lead to the concurrent execution of several pieces of program [12, 50]. On the other hand, text-based languages are generally taught in introductory courses in an imperative procedural way that focuses on a single main function executing a sequence of instructions [50]. It appears that these changes in programming paradigms can pose obstacles to learners because they require cognitive adaptations [26, 54].

2.1.2 Fundamental concepts. Block-based and text-based programming implement almost the same core programming concepts. ACM and IEEE-CS regularly established international curricular guidelines for computer science education. In their latest release [23], they defined the "fundamental programming concepts" as: basic syntax and semantics, variables and primitive data types, expressions and assignments, simple input/output, conditional and iterative control structures, functions and parameter passing, and the concept of recursion. Several studies [33, 51] found that Scratch could successfully be used to introduce learners to: variables, conditional, iterative logic, function parameters and returns, and output/input. However, the concepts of data type and recursion do not seem to be implementable in the block modality, or only partially (see Section 2.1.4 for data type).

2.1.3 Semiotic registers. Even if the implementable concepts are nearly similar in block and text modalities, the "semiotic registers" [19] of their commands are quite different. The block register is made of graphical shapes of different colors containing keywords in English, drop-down lists, text and number entry fields, whereas the instruction register is composed of textual elements only: keywords in English, mathematical and typographical symbols [12]. Moreover, the keywords used in the block register tend to be closer to natural languages in partially imitate their grammar (e.g. incrementing a variable is "x=x+1" in Python and "change x by 1" in Scratch) [28, 49]. The transition from one register to the other can be more or less immediate, and therefore depends on the "semiotic congruence" of the representations [19]. A previous work [12] established that this congruence is strong for variables, conditionals, and functions concepts but is weaker concerning loops. For this reason, prior knowledge of block programming can be helpful for the concepts of variables, conditionals and functions and rather an obstacle for the apprehension of the loop concepts in the text modality. Additionally, research shows that the specifics of textual syntax are particularly problematic for novices [44]. Its dense notation can be a difficulty for beginners as it can overload their working memory. On the other hand, the block register helps learners by showing how to apprehend commands in larger chunks [9].

2.1.4 Syntax and type errors. Block-based systems avoid syntax and type errors which are frequent when using text-based languages. Program editing in a visual block environment consists of joining shapes together. Furthermore, languages like Scratch are just distinguishing booleans from other data types and are weakly typed at runtime. These two characteristics lead to preventing a majority of errors [49, 28]. In contrast, using strongly typed text languages (even dynamically like Python) requires the understanding of data types and the respect of languages grammar and syntax [28]. As a consequence, in this programming modality, errors are numerous [18, 2] and error messages lack of clear formulations [31]. For beginners, learning how to interpret them requires a lot of practice [10].

After having detailed the aspects linked to languages' intrinsic features, let us now compare elements that are associated with their editing environments.

2.1.5 Command catalog. Block programming environments present the user with a browsable "palette" listing all existing blocks organized thematically or conceptually [49]. This makes it possible for neophytes to discover concepts or to remember those they have already learned. On

Table 1. Summary of the differences between block-based and text-based languages

Aspects	Block-based	Text-based
Paradigms	Object-oriented, event-based, parallel	Imperative, procedural
Fund. concepts	Variable, conditional, loops, function	+ data type, recursion
Semiotic registers	Graph. shapes, drop-down lists, fields	Keywords, typograph. symbols
Errors	-	Syntactic, semantic
Command catalog	Navigable palette	-
Composition	Drag and drop	Keyboarding
Execution	Highlighting, step-by-step, var. state	Debugger

the contrary, programmers using text environments must have in mind all code structures and appropriate syntaxes [9].

2.1.6 Program composition. Blocks allow to compose programs by dragging and dropping different pieces to assemble them. The difficulty of typing and looking for symbols on the keyboard, which are a common part of text programming, is substantially reduced by this composition procedure [49]. Actually, the simple mechanical process of inputting the program text might be a cognitive and motor challenge for young learners. Furthermore, the necessity of using the keyboard to rectify the inescapable typing errors increases cognitive distractions [28].

2.1.7 Execution control and visibility. Block settings make it easier to control and follow the execution of programs. They provide highlighting of the block that is being run in an effort to link programs to actual actions. They can offer a step-by-step mode (setting speed, stopping and resuming execution), and display the current state of the variables. These characteristics give novices a better understanding of how programs are executed and what they are doing [9]. Note that some text-based environments include a debugger that can offer such features, but they are difficult for beginners to master.

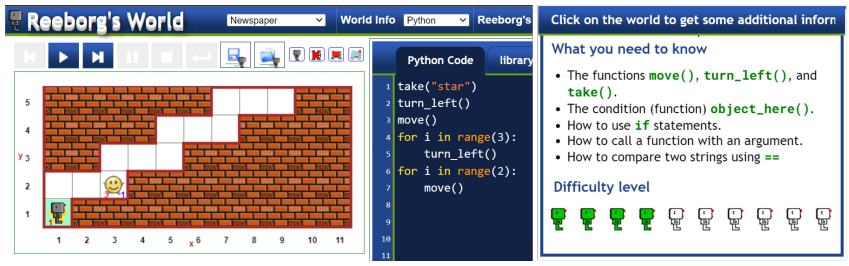
The aforementioned editing environment comparisons are conducted using simple textual code editors. However, some instructional text-based environments, such as *PyScripter* [39], provide helpful features like syntax highlighting, autocompletion, or syntax live checking which can aid in reducing semantic errors and limiting keyboarding.

Table 1 summarizes the aforementioned differences and similarities between block and text languages. Next, we present several existing applications related to our research questions.

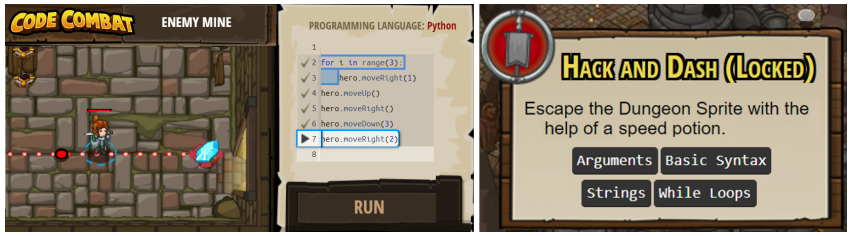
2.2 Existing applications

Researchers and companies have already proposed applications aiming at learning Python via serious games and development environments dedicated to the block-to-text transition. Here is an overview of these solutions.

2.2.1 Python learning games. In this section, we present a set of pre-existing games. They have been selected according to two criteria: i) the application has a pedagogical purpose; ii) characters are controled by Python programs. *Reeborg's World* [41] (see Figure 1a) is a serious game developed by a Canadian physicist and targets programming initiation. *Algoréa* [1] is the platform of a national programming contest organized in France. *Code Monkey* [16], *Code Combat* [15] (see Figure 1b), and *Coding Park* [17] (see Figure 1c) are commercial applications aimed at helping young people and teenagers to learn Python in a recreational way.



(a) Reeborg's world: game and conceptual indications.



(b) Code Combat: game and conceptual indications.

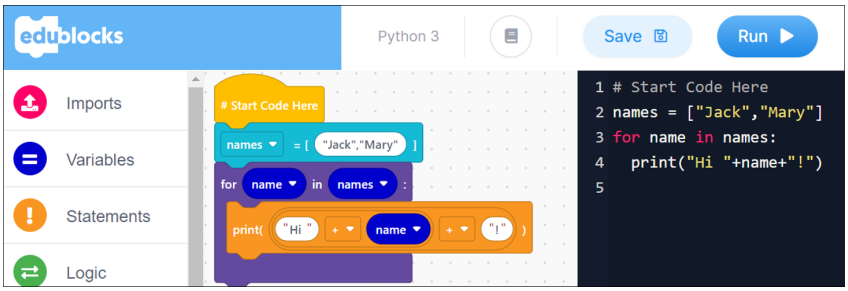
(c) Coding Park: game and conceptual indications.

Fig. 1. Three examples of Python learning serious games.

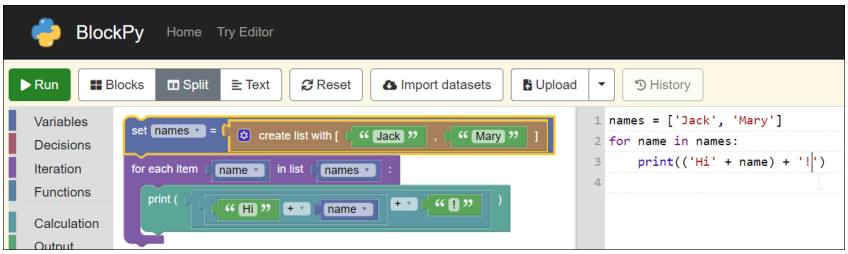
In all these applications, the concepts involved in the different levels are made clearly explicit. For instance, *Reeborg's World* offers a "world info" button which gives several information such as: the objective of the "world", the level of difficulty, and, in the "what you need to know" part, the control functions to be used and the concepts to be implemented (see Figure 1a). In *Code Combat*, before starting a new stage of the game, the concepts to be applied are listed after the given level objective (see Figure 1b). At last, in *Coding Park* the different quests are named after the concept that they each lead to discover (see Figure 1c).

As explained in the introductory section, we decided to propose a different learning approach using the constructivist paradigm. Hence, our design objective is to allow students to understand the different programming concepts by putting them at stake in different levels without making them explicit.

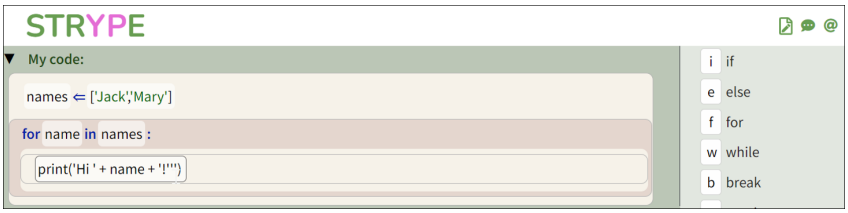
In the next section, an overview of development environments specifically designed to support the block-to-text transition is offered.



(a) EduBlocks one-way transition environment.



(b) BlockPy dual-modality environment.



(c) Strype hybrid environment.

Fig. 2. Examples of the three sorts of transitioning environments.

2.2.2 Block-to-text environments. To assist the transition from blocks to text, a number of digitally based approaches have been investigated. In accordance with the Lin and Weintrop's taxonomy [29], three sorts of environments are presented: one-way transition, dual-modality, and hybrid.

One-way transition environments are composed of two views. Programs can be edited using blocks in one view, and are then automatically translated into a target textual language in the other view. Users may only consult the translated program and, in some case, execute it, but it cannot be changed directly. For instance, the *EduBlocks* environment [20] (see Figure 2a) converts instantly edited block-based programs into Python scripts. *Patch* [43] offers a comparable solution based on Scratch blocks.

Dual-modality environments are similarly made up of two views, but the textual view allows for direct creation and modification of programs. As a result, the block view is automatically refreshed with the translated block script. Examples of existing implementations include *PencilCode* [8], which produces Javascript and more recently Python code [4]. For Java, *BlocEditor* [30] provides an analogous method. *BlockPy* [7] (see Figure 2b) offers another dual-modality environment made specifically for Python.

Table 2. Classification of the presented block-to-text environments

Environments	Sort	Type
EduBlocks [20]	One-way transition	Translation
Patch [43]	One-way transition	Translation
PencilCode [8]	Dual-modality	Translation
BlocEditor [30]	Dual-modality	Translation
BlockPy [7]	Dual-modality	Translation
Stride [28]	Hybrid	Fusion
Strype [48]	Hybrid	Fusion
CodeStruct [24]	Hybrid	Fusion

Finally, blocks and text are being combined into hybrid environments to create a single view. Here, drag-and-drop, point-and-click, or keyboard shortcuts can be used to insert high-level structures (such as loops and conditionals). Traditional text editing augmented with auto-completion or coding hints is employed to introduce the expression-level code. *Stride* offers learners an operational block-and-Java implementation [28]. The recently launched *Strype* [48] provides a Python-compliant "frame-based" environment (see Figure 2c). In addition to the above described features, it allows to manipulate already written text bundles as if they were blocks. *CodeStruct* [24] is another recent hybrid environment which has the particularity to provide explanations and runnable examples about concepts through contextual tooltip.

As shown in Table 2, going beyond this classification, there are actually two types of environments. Those based on translation (one-way transition and dual-modality) whose value is to support the block-to-text transition on intrinsic aspects of the languages (programming paradigm, concepts, and semiotic registers). Indeed, the live translation of code from one semiotic register to the other may allow students to assimilate the syntax of textual languages by observing (one-way) or editing (dual-modality) the generated code. The other elements (paradigms and concepts) being, by construction, constant. The second type of environments are those based on fusion of modalities (hybrid). The benefits are, in this case, on editing environments aspects. Thus, those applications can provide textual editing environments while still benefiting from some of the advantages of blocks (command catalog and program composition). When designing the *Pyrates*' editing environment, both aspects of the transition were addressed: the intrinsic properties and the editing mode.

Having exposed the state-of-the-art of the study, the different parts of the *Pyrates*' design are described in the next section.

3 DESIGN OF THE PYRATES APPLICATION

The *Pyrates* online application consists of a platformer game allowing to control a character using a Python program. This avatar must complete various playful objectives through several successive levels. The design of this software includes two aspects: the construction of the levels following the constructivist model and the design of the editing environment in order to facilitate the block-to-text transition.

3.1 Game levels

The eight levels of this game were designed by implementing the constructivist paradigm which is based on Piaget's psychological hypothesis about adaptive learning [34]. The Piagetian constructivism theory suggests that individuals create their own new understandings, based upon the interaction of what they already know and believe, and the phenomena or ideas with which they come into contact. The purpose of constructivist teaching being to lead toward higher levels of understanding and analytic capabilities [42]. In the context of learning Python in *Pyrates*, this means that the programming concepts at stake in each level of the game are not explicit but are made necessary by the game problem to be solved. This approach should permit to give strong meaning to the concepts being worked on. In the continuity of Piaget, Brousseau [14] qualified these kind of learning situations as "adidactical situations". Consequently, these situations have been conceived by being guided by two of the "adidactical conditions" described by Bessot [11]:

(1) The students must be able to consider an initial response to the posed problem in the form of a low-effectiveness "basic procedure" based on prior knowledge.
(2) The targeted concepts must make it possible, when implemented, to move on to a "winning procedure" that solves the problem at hand.

The targeted programming concepts have been chosen in coherence with the French mathematics and computer science curriculum of grade 10 (i.e. variable, conditional, for and while loops). As explained earlier, each level should lead the students to implement some of these concepts without being explicit.

To achieve this, a recurring objective was first established for each level of the game. This goal is to pick up a key to open a treasure chest. Next, the level maps were designed in order to make necessary the use of the targeted concepts. The players should be able to initially engage in the game by coding a basic procedure using the character control functions (i.e. walk, jump, etc.). Next, they must mobilize the targeted concept if they want to succeed in opening the chest (winning procedure). The *a priori* analysis methodology [6] was followed to conceive the levels. It consists in designing a teaching situation (a game level here), setting out its characteristics (game map and environmental constraints), and then establishing a list of the winning procedures that solve the problem. The next step is to check whether the targeted knowledge is indeed present in the winning procedures, and then to modify the level characteristics iteratively until they are.

Table 3. Concepts and Features of Pyrates Levels

Levels	Main concepts	Game's features	Editor's features
1	For loop (repeat)	Repetitive path	Limited lines
2	For loop (repeat)	Repetitive path	Limited lines
3	Variable	Information retention	-
4	Conditional	Random path	-
5	Conditional	Random path	-
6	For loop (counter)	Structured path	Limited lines
7	For loop (counter)	Structured path	Limited lines
8	While loop	Random repetitive path	-

Figure 12, in the appendix, reproduces the game map and a winning procedure for each level. Table 3 then summarizes the main targeted concept and the features of the game map and of the code editor that should lead to this concept implementation.

Notice that in the context of block-to-text transition, the for loop concept was broken down into two sub-concepts. At first, the for-repeat sub-concept which is the text transposition of the "repeat" block which offer the simple repetition of instructions. Students were asked to use the following syntax trick "for _ in range (n)" to indicate that they do not use the loop variable in the loop body. The second sub-concept is for-counter which adds the management of the automatically incremented loop variable. These loop counters are not directly available in some block languages like Scratch.

The game's levels design were just presented, now let's focus on the features of the *Pyrates*' development environment.

3.2 Editing environment

The design of the editing environment of the application is covered in this section. The presentation is based on Figure 3 which depicts the application's graphical user interface and its different areas.

This part of the application was conceived based on the research findings outlined in Section 2.1. Hence, some characteristics of block programming environments (command catalog, program composition, less errors, enhanced execution) were incorporated because of their advantages evidenced in the literature.

First, a fixed sidebar was added on the left of the screen which incorporated, among other things, a **programming memo** (see Figure 3b). It is a kind of synthesized documentation explaining the basics of Python programming. The command catalog specific to block-based settings served as a model for this element. The memo's contents can be accessed by clicking on the different blue buttons. They are organized following the fundamental programming concepts (basics, variable, conditional, for and while loop). When the mouse pointer is hovered on a button, the title is changed

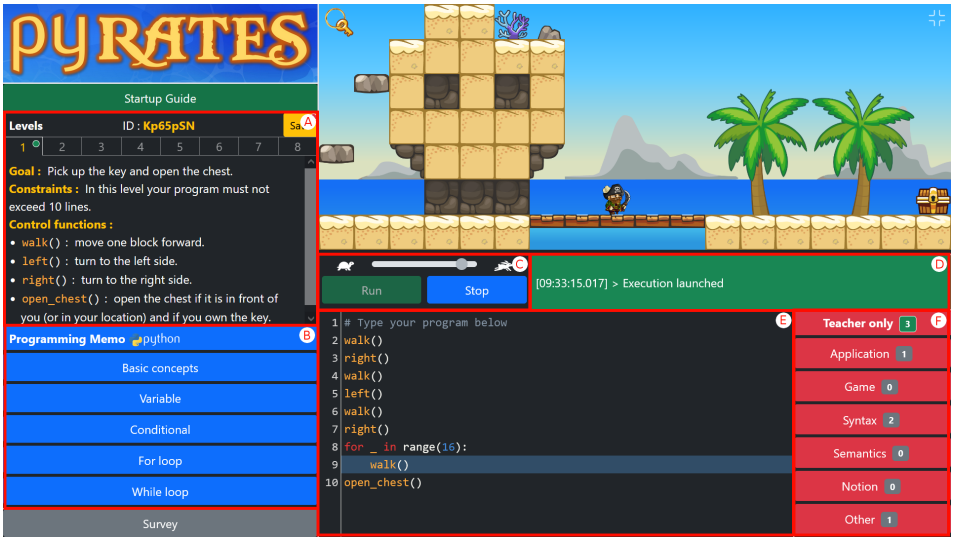


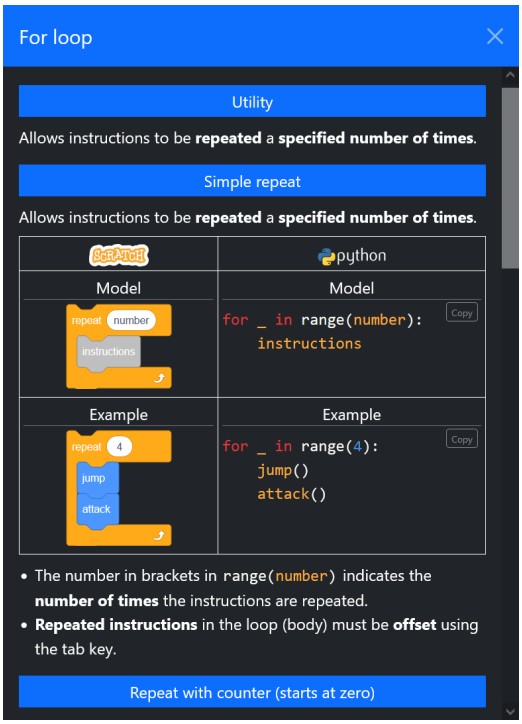
Fig. 3. Different areas of Pyrates' graphical interface: level area (a), programming memo (b), control panel (c), console (d), code editor (e), and teacher area (f)

to reflect the usefulness of the concept in an effort to guide the users through their research. For instance, “variable” turns to “Store information in memory”.

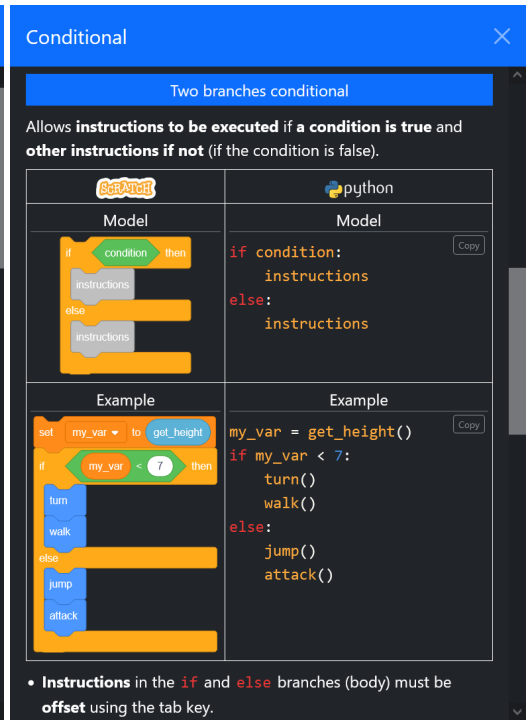
Clicking on a button causes a side panel appearance detailing the concept in sub-concepts (see Figure 4). In this panel, sub-concepts are explained and followed by **a general model and an illustrative example**. Both Python and Scratch are used to express these programs. This choice was made because, in France, the block-based language used to introduce programming to lower secondary students is Scratch. The Python general model and its block translation are aimed to assist learners in the semiotic register shift. The intention is to foster the apprehension of Python syntax in large chunks rather than element by element. To illustrate this point, in the “Simple repeat” part of the “For loop” panel (see Figure 4a), learners should focus on the number enclosed in brackets and treat the remaining code as a single aggregate. Futhermore, Ginat *et al.* [22] report that including worked-out examples in instructional materials leads to reduction of students’ cognitive load and to more efficiency in tackling new problems.

In the memo, each Python code snippet is associated with a **copy button** in order to reduce keyboard inputs. The goal is here to encourage the practice of copying and pasting from the memo to the code editor (see Figure 3e). This method can be seen as a form of continuity for block-based environments’ drag-and-drop functionality.

Even though the design efforts to reduce programming errors presented above, it seems over-optimistic to expect the complete eradication of syntactic and type errors inherent to block editing.



(a) Side panel extract about the “for loop” concept.



(b) Side panel extract about the “Conditional” concept.

Fig. 4. Two extracts of the programming memo.

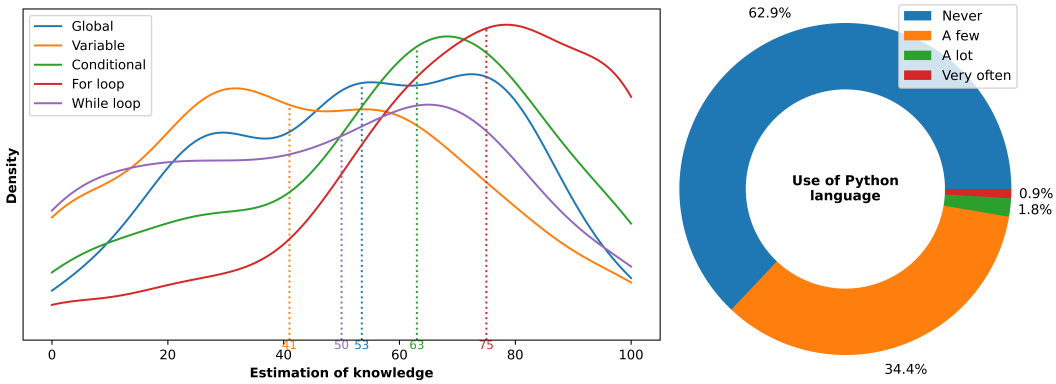
Becker established that understanding error messages is a huge challenge for inexperienced programmers [10]. Therefore, the editing environment has been improved with an existing research-based **syntax analyser** created especially for novices [27]. This module analyses the Python code prior to the Python interpreter. It delivers error messages in the users' native tongues (only French and English at the moment). According to Qian and Lehman [40], students' fluency in English is significantly correlated with their success in learning programming. The statements are also formulated in a less technical, straightforward style that beginners can understand. Additionally, these predefined messages have been slightly modified to make them consistent with the wording of the programming memo. As a result, when a syntax error occurs, an appropriate message is shown in the console section (see Figure 3d) and the faulty code line is highlighted in red in the editing area. A syntax-error-free program does not mean that the code is runnable. Type or other semantics errors may still appear during interpretation.

Lastly, a **control panel** (see Figure 3c) was developed to enhance the administration of executions. Players have the ability to start and stop the code execution, and to change its speed using a slider. This slider alters the velocity of character motions by acting on a multiplying factor. When the game first launches, this factor is set to 1 (tortoise), and it can be increased up to 3 (hare). The monitoring of the execution is guaranteed by the highlighting of the executed line in the code editor area (see Figure 3e). This should help students in making links between the code and the current character action.

The described design has been assessed using the methodology detailed in the following section.

4 METHODOLOGY

This section describes the methodology used to evaluate the different facets of the *Pyrates*' design. The experimentation relies on classroom field testing. The application was used in eight French high school classes (10th-grade: 14-15 years old). According to the French curricula, grade 10 students have supposedly already used Scratch software in middle school (grades 6 to 9) and have not yet been exposed to Python programming. In order to confirm this hypothesis, we asked the 240 involved students to estimate their general knowledge in Scratch (and more precisely concerning some concepts by asking them to estimate their knowledge between two extremities, "Low"and "high", by moving a cursor) and if they had already used the Python programming language before



(a) Scratch prior knowledge (repartition and median).

(b) Python usage before experimentation.

Fig. 5. Prior Scratch knowledge and Python usage declared by students.

Table 4. Details of activity traces.

Activity	Information
Content view	Content identifier, consultation time
Content copy	Content identifier
Program launch	Program code, execution speed, program outcome, error (if occurred)
Action on level	Started, completed, restarted, resumed
Teacher help	Help type
Speed cursor change	New speed value

the experimentation. The results of this survey are shown in Figure 5. Based on these declarative elements, we can overall consider that these students have a general knowledge of Scratch (less for variables), and that they had no prior substantial Python programming experience. In addition, standard French 10th-grade classes are most of the time balanced therefore it can be assumed that gender parity is almost respected in our student sample.

The students played with *Pyrates* during two or three 55 minutes-sessions spaced one or two weeks apart. The application and its functioning were shortly explained during the five first minutes, then the students were free to play on their own for the remaining time. The teachers were asked to step in on students' demand, or when they had been struggling for an extended period of time. In this situation, the teachers were asked to record the topic of the provided assistance (application, game, syntax, semantics, notion, other) by clicking on buttons in a frame dedicated for that purpose on the application (see Figure 3f).

During the whole experimentation, the application tracks, in a wider range, all the players' activities: content viewing and copying, launched programs, programming errors (syntactic and semantic), received helps from the teacher, use of the control panel, and so forth. These traces were automatically generated following the activity of the players and then exported and stored using the xAPI standardized format [25]. Table 4 gives the details of the collected information depending on the students' activity. In the last minutes of the final session, the students were asked to answer an online survey that provides additional data. This survey's objective was to gather respondents' qualitative inputs. The questions focused on prior knowledge (Scratch and Python) and on the overall perception of the application (clarity and usefulness of the contents, handling, difficulty, motivation, and enjoyment). This data collection protocol respects the ethical rules in force insofar as it is compliant with the European GDPR. The data set of this study is then composed of 224 survey replies (due to technical difficulties, some students were unable to respond) and 69,701 activity traces.

Lastly, the data processing was conducted using Python scripts. In particular, we automatically detect the concepts implemented in students' programs using their abstract syntax tree (*AST* package). The *Pandas* library was also used to manipulate and analyse data, as well as *Matplotlib* and *Seaborn* to create the graphs presented in this article.

The following section provides the results obtained from this data collection and analysis.

5 RESULTS AND DISCUSSION

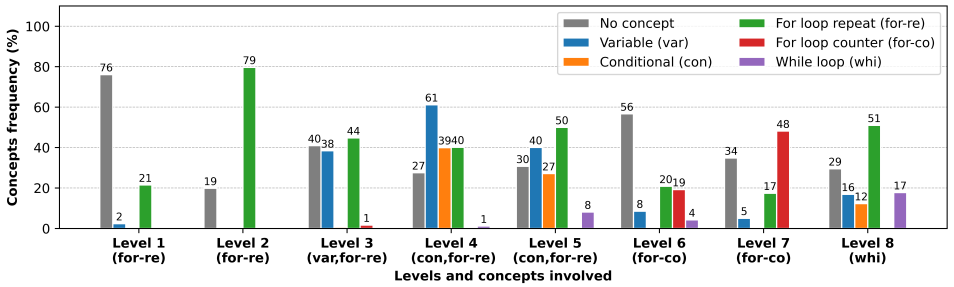
The results exposed and discussed below are focused on two aspects of the design. First, the evaluation of the design of the game's levels (RQ1) and secondly, the assessment of the incorporation

of block-like features in the editing environment (RQ2). Then, the student's overall evaluation of game design (RQ3) is presented.

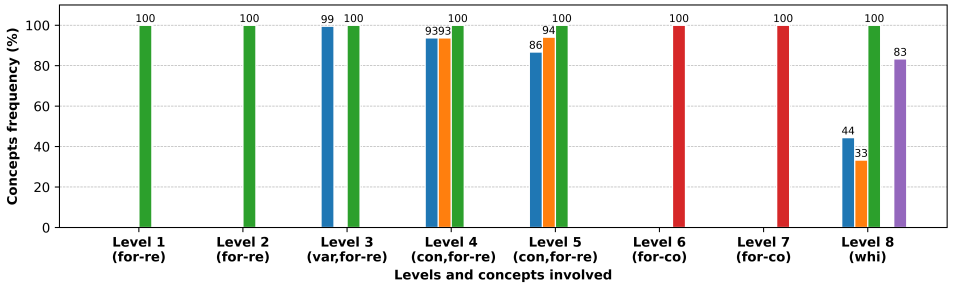
5.1 Levels evaluation

This section answers RQ1 i.e. whether the levels of the game allow students to apprehend the different targeted programming concepts consistently with the constructivist principles. This is understood as providing the opportunity to try out basic procedures before being able to win the levels by implementing the targeted concepts. The following analyses are based on student activity traces about executed programs. Indeed, the programming concepts implemented in these programs were automatically detected using abstract syntax trees. Figure 6 shows the frequency of the implemented concepts on all of the executed programs for each level. This means for example that 21% of the executed non-winning programs in level 1 contain a for-re loop (see Figure 6a).

We first explore whether the level design allowed students to consider an initial response to engage with the problem at hand. Figure 6a represents the frequency of appearance of concepts in the programs leading to non-winning procedures for each level. Note that these non-winning programs have been fully executed, so they are syntactically correct and free of game errors (trying to walk in a wrong place, fall into spikes, etc.). However, they do not allow to finish the level, meaning to pick up the key and open the treasure chest. This figure shows that students implement basic procedures without concepts (i.e. using only control functions) in all levels and especially when a new concept is involved (lev.1, lev.3, lev.4, lev.6, and lev.8). Trial, termed "intermediate procedures", can be observed to implement the targeted concept but which require some semantic adjustments to complete the level. For example, a student may have implemented a for loop at



(a) Concepts implemented in executed non-winning procedures.



(b) Concepts implemented in executed winning procedures.

Fig. 6. Concepts implemented in executed procedures by level.

level 1, but not have achieved the level, because he has yet to adjust the number of iterations of the "walk" control function. To summarize, it can be stated that players are able to engage in the levels through basic procedures composed of control functions only.

Next, to answers the second part of RQ1, does the implementation of the involved concepts make it possible to complete the level and is it the only way to do so? In other words, is the targeted concept in a level a necessary and sufficient condition to establish a winning procedure? Figure 6b represents the frequency of appearance of the programming concepts in the programs leading to winning procedures for each level. When for loops (for-re and for-co) and variables (var) are targeted in a level, the figure indicates that they are almost systematically implemented in the winning procedures. This validates level designs for these concepts.

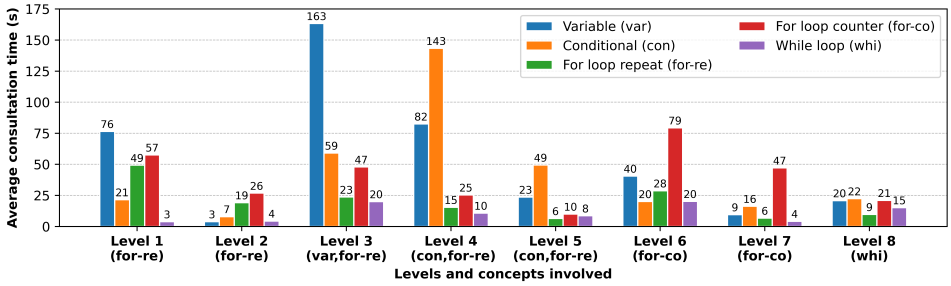
In contrast, as shown in Figure 6b, test-based concepts (con and whi) are sufficient to complete a level but are not necessary (i.e. they are not always present in the winning procedures). A small part of the students (7-8% for the conditional and 17% for the while loop) manage to succeed these levels without implementing these concepts. A randomized game map was intended to make the use of these concepts mandatory (see Table 3). However, some students used a *modus operandi* consisting in a run-stop execution loop until they obtain a random configuration favorable to their non-conditional program. This strategy, which can be coined as "conceptual bypassing", makes it possible to succeed in these levels without implementing the programming concepts at stake. This procedure has very little chance of success because of the large number of different level random maps. These students who remain at any costs in the playful domain are unwilling or unable to enter into conceptual learning by exploring the environment seeking a concept that might allow them to complete the level. In addition, other students manage to replace the while concept (whi) by the conjunction of a for loop (for-re) and a conditional structure (con). To summarize this point, it can be argued that the targeted concepts allow to finish all levels, but that they are not always necessary. From this point of view, the design of levels' maps is very effective in making the variables and for loops concepts mandatory. On the other hands, it is not sufficient to force the use of the concepts based on tests (conditional and while loop), since a small part of the students was able to avoid them.

This section can be summed up by saying that the answer to RQ1 is rather positive. Respecting Bessot's adidactical conditions, the design of the levels overall allows to learn the targeted concepts in accordance with the constructivist approach. The next section is about the evaluation of the editing environment.

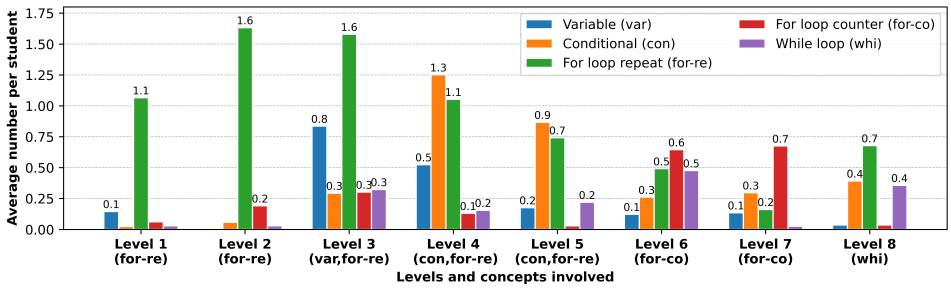
5.2 Editing environment evaluation

The goal of this section is to answer to RQ2, that is the study of the appropriation of the transition features by the students. For this purpose, the following traces are considered: consultations of the memo, copy-paste from the memo to the code editor, errors detected by the syntax analyser and by the interpreter, syntactic and semantic aids given by the teachers during their interventions, manipulations of the speed cursor, and chosen speed during the programs' execution.

To begin, let us look at the usage of the programming memo. Figure 7a shows that this memo is frequently consulted by students. It can be noticed that, like the catalog of block-based environments, it supports the discovery of concepts. Indeed, each time a new concept is involved in a level (lev.1, lev.3, lev.4, lev.6, and lev.8), a great variety can be found in the consulted concepts. This seems to reveal a search process. On the other hand, when the concepts have already been used (lev 2, lev 5, and lev 7), the consultation appears to be more focused on the targeted concepts. It can be hypothesized that, in this situation, the learners need to retrieve the syntax of the concept they want to implement. This process is quite similar to the recall capability of the block catalog.



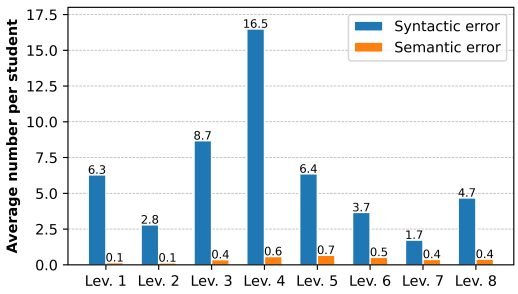
(a) Python memo consultation time by students.



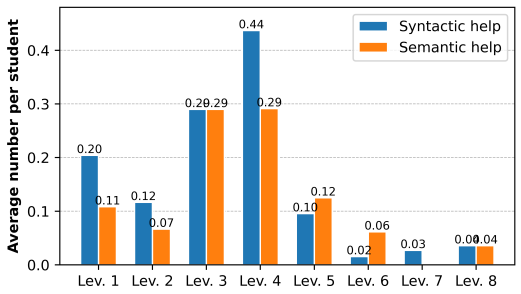
(b) Copy and paste of the Python memo by students.

Fig. 7. Consultations and copy-pastes of the Python memo by level.

Next, according to Figure 7b, students are frequently using the copy-paste tool to implement a concept. Hence, whenever a concept is targeted in a level, there is, on average, a high rate of copy-paste associated with it (from 1.6 to 0.4). Note that, for a same concept, the average number of copy-paste decreases from one level to another. It can be advanced that more and more students succeed in memorizing the syntax after several uses. This copy-paste practice is kind of equivalent to the drag-and-drop of blocks, and can limit keyboard input and, to some extent, help establish code structures.



(a) Errors detected by the syntax analyser and the Python interpreter.



(b) Help received by students from the teacher.

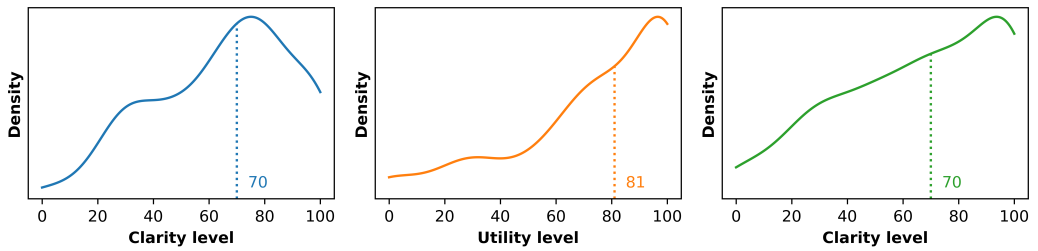
Fig. 8. Errors detected by the application and teacher helps received by students.

Considering errors analysis, Figure 8a shows that syntactic errors (issued from the syntax analyser) are numerous and in a much higher proportion than semantics ones (issued from the interpreter). This is consistent with Altadmri and Brown's results, which suggest that students first have trouble with syntax before they start struggling with type and semantic problems [2]. Looking at the assistance given by the teachers (see Figure 8b), it is interesting to note the low frequency of the syntax interventions regarding the number of errors. Actually, there is one intervention for every 30 to 40 syntactic errors in the first four levels. The students are therefore presumably able to adjust their syntax-erroneous code thanks to feedback from the environment and without asking the teacher. This fact is undoubtedly related to the enhanced messages provided by the employed syntactic analyser. Nevertheless, it is important to keep in mind that, in a general way, the syntax errors are less challenging to solve than semantic errors [2].

The traces generated by the application give quantitative insights regarding the use of the memo and the occurrences of the error messages. To go further, these analyses can be qualitatively augmented with the survey results. The students had to evaluate several aspects of the application by setting cursors between two extremes ("Not clear" - "Very clear", "Not useful" - "Very useful"), which had the effect of generating a score between 0 and 100. The survey included questions related to the Python memo and the error messages. Figure 9 presents the scores distribution (density) and median for the these questions.

In addition to being extensively consulted by students, the memo's explanations are considered as clear by the majority of them (see Figure 9a). Despite of this, a group of students can be distinguished around a score of 30 for whom these contents are more confusing. The comparisons with Scratch are judged as useful or even very useful by the large majority of the students (see Figure 9b). Finally, the error messages, which have been shown to foster students' autonomy, are also deemed to be clear by most of the respondents (see Figure 9c).

Let us now evaluate the use of the program control features. Figure 10a presents the average number of actions on programs per student. Here, a program is considered as correct if it does not contain any syntactic or semantic errors (this does not mean that it allows to win the level). Otherwise, it is identified as erroneous. It can be observed that there is a very large number of programs run on average per student. Many of them are erroneous, suggesting that students are adopting a trial-and-error programming approach to reach syntax correctness. Numerous correct programs are also launched, which shows that students progress through the game levels in incremental intermediary steps to the final opening-chest goal. This trial-and-error and incremental programming method could be explained by the programming practice inherited from the blocks. Indeed, researchers discovered that learners acquired some particular programming behaviors



(a) Clarity of explanations in the memo.

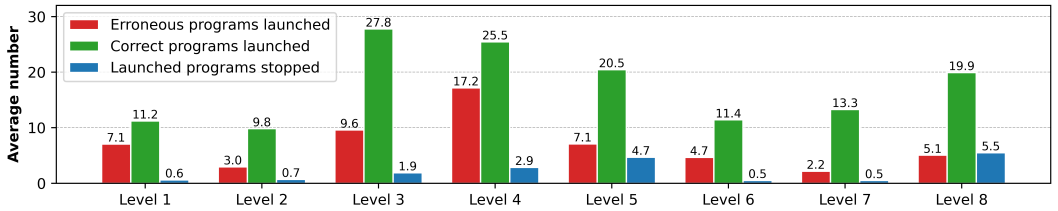
(b) Utility of Scratch-Python comparisons.

(c) Clarity of error messages.

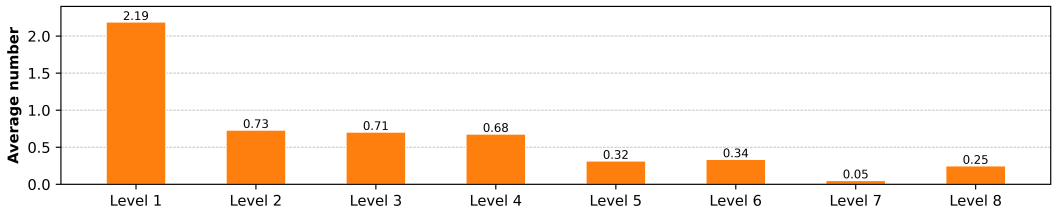
Fig. 9. Results extract from the student survey (score distribution and median).

when they learn to program with blocks. They tend to develop several habits, including a totally bottom-up programming approach and a tendency towards extremely fine-grained programming [32]. The use of the stop button to interrupt the execution of a program is rare. However, it is possible to distinguish two types of behaviors depending on the way the levels maps are generated. For a first set of levels with fixed non-random maps (Lev.1, Lev.2, Lev.6, and Lev.7), students perform on average between fifteen and twenty launches and almost no stops. In levels containing randomly generated maps which change at each run (Lev.3, Lev.4, Lev.5, and Lev.8), students tend to do more launches and to stop some of them. Actually, for these randomly generated levels, some students adopt a transient operating mode consisting of a series of launch-stop actions until they obtain a random map configuration suited to their program. This process has already been described in Section 5.1.

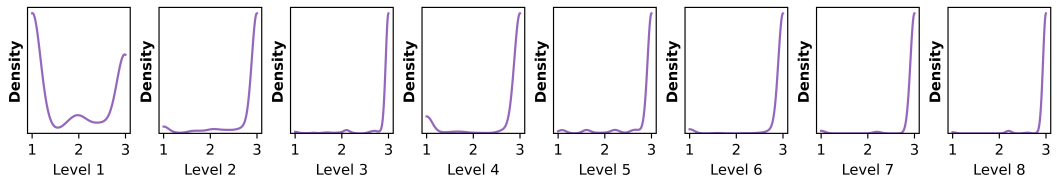
Finally, let us pay attention to the speed change cursor. It is on average rarely used and decreasingly over time (see Figure 10b). Figure 10c shows the distribution (density) of launched programs' execution speeds for each level. From level 2 onwards, the programs are almost all launched at the maximum speed (multiplying factor of 3). The trial-and-error and incremental programming approach earlier described requires this high execution speed. Three students reported in the open-ended field of the survey that "the character does not move fast enough". Nevertheless, a marginal practice can be noted in more advanced levels (level 4 and level 5). It consists of returning



(a) Average number of actions on programs per student.



(b) Average number of execution speed changes per student.



(c) Distribution of programs execution speed by level.

Fig. 10. Data concerning the execution control by level.

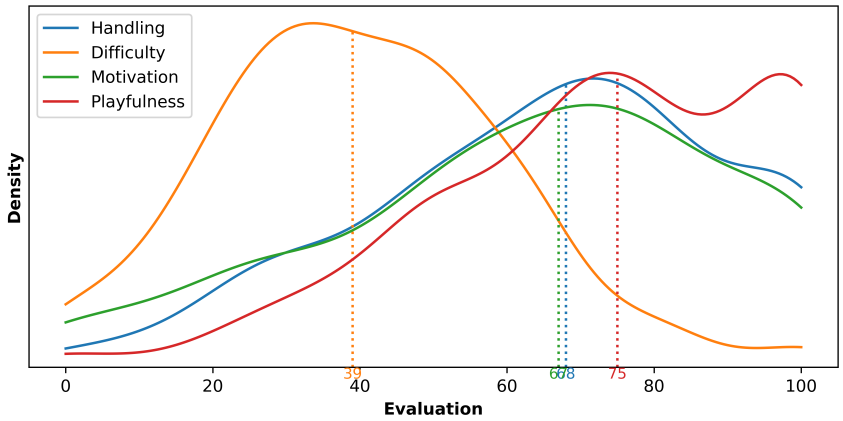


Fig. 11. Player evaluation of different aspects of overall game design (distribution and median).

to slower execution speeds. Observations during the experiments indicate that some students need to follow more easily the executed lines in a step-by-step action mode.

5.3 Overall game design evaluation

This section deals with the RQ3 question and the general design of the game. As shown in Fig. 11, the application is considered as rather easy to use (handling median is 68), which can be explained by the presence of a start-up guide (green button in the graphical interface) as well as by the quick introduction made by the researcher at the beginning of the experiments.

Second, the game is overall considered as difficult by the students (median 39). Indeed, few of them had the opportunity to complete all eight levels of the game and only 60% of the 240 students reached at least level 5.

Surprisingly, students' motivation remained high (median 67) in spite of the difficulty they felt. Few students felt discouraged at the end of the experiment. We replicated here the research results on the benefits of gamification on learner motivation [55].

Finally, the overall experience was rated as entertaining (playfulness median is 75), with a group of students placing the cursor at the maximum (100). This expected outcome is also among the demonstrated benefits of serious games [46].

6 CONCLUSION AND PERSPECTIVES

To conclude this paper, its main results are summarized as follows. The levels of the *Pyrates* serious game were conceived using a constructivist philosophy with the objective of teaching the fundamental concepts of programming in Python. Moreover, *Pyrates* includes an editing environment designed by taking inspiration from block-based programming editors intending to benefit from their advantages.

Both these aspects of the conception were evaluated by analysing students' activity and answers to an online survey. First, it is possible to argue that the levels are overall respecting Bessot's adidactical conditions:

- players have the ability to engage in levels through basic procedures consisting of control functions only;

- the design of the levels' maps is very effective in leading to the implementation of variable and for loop concepts, but is not sufficient to systematically require the use of test-based concepts (conditional and while loop).

As a result, it can be stated that, generally speaking, the levels' design allows the players to implement the targeted concepts in a constructivist way.

About the layout of the editing environment, some design choices have the following positive consequences:

- the programming memo is very frequently consulted by the students, it is the support of the discovery and the recall of the concepts;
- the included comparisons with Scratch are considered as useful by a large majority of students, they should help the apprehension of Python structures in larger chunks;
- copy and paste from the programming memo is widely practiced, this has the effect of limiting keyboarding;
- the feedback provided by the syntax analyser via "clear" error messages makes it possible to correct the programs with very little teacher involvement.

The control panel should allow the students to better understand the execution of the programs. However, it does not produce the expected results:

- the program launch button is frequently used and the speed control slider is very early set to the maximum speed in order to adopt a trial-and-error programming approach which does not foster reflection;
- the button allowing to stop the executions is scarcely used, and when it is, this is mostly to try to succeed in the random-based levels using "conceptual bypassing".

Finally, by analysing survey answers, we found that overall, the handling of the application is considered as good and that the students declare themselves as motivated and entertained even though the game is perceived as difficult.

In order to highlight the strengths of this work, Table 5 compares *Pyrates* with the other games and environments referenced in the state of the art section. One of the main advantages of *Pyrates* is to offer in a single platform both a serious game for teaching Python and a development environment facilitating the block-to-text transition. More precisely, the designed serious game offers a non-explicit pedagogical approach following constructivist principles that should allow students to discover the fundamentals concepts of Python programming by giving them a strong meaning. Moreover, the environment supports the transition from blocks to text in both its intrinsic and editing tool aspects. With regard to intrinsic aspects, the transposition of concepts is supported by general models and illustrative examples, and error handling is facilitated by the integrated syntax analyser. Regarding the editing environment aspects: the addition of the programming

Table 5. Comparison of Pyrates with other works

Other works	**Comparison to Pyrates**
Python serious games [41, 1, 16, 15, 17]	Explicit approach / Constructivist approach
Block-text translation environments [20, 7]	Live translation / Translated models and examples
Block-text hybrid environments [24, 48]	Mix of features / Programming memo, copy button, syntax analyser, and control panel

memo, which acts as a command catalog, facilitates concepts discovery, program composition is assisted by copy-and-paste, and, to a lesser extent, the control panel enhances execution control and visibility.

Beyond these results, our contributions have implications for computing education. Firstly, it provides practitioners and learners with an environment supporting several dimensions of the block-to-text transition, as well as learning materials and assignments that allow them to master the fundamental concepts of Python programming in a constructivist approach. In its first two years online, *Pyrates* has recorded more than 140,000 played games in France and about 2,500 internationally. Secondly, the outcomes of this work can inform designers of learning technology. The characteristics of the designed situations (the level maps in particular) leverage the students' use of the targeted programming concepts (in a constructivist approach), and the *Pyrates* environment combines in a novel way different means of supporting the block-to-text shift, which may suggest avenues to design for this transition.

These results must be considered in light of the limitations of the methodology. Because the students were in a naturalistic context, it was difficult to maintain totally similar experimental conditions between different groups, particularly regarding the teacher's activity and the scheduling of the sessions. In addition, reasoning only on averages allows to identify trends, but masks the disparities of levels and practices between the students observed in classrooms. Lastly, students' actual learning while playing with *Pyrates* was not measured.

Finally, let us mention some perspectives to extend this work. Edwards [21] argues that beginners in computer science are more successful at learning if they move from a trial-and-error approach to a "reflection-in-action" practice. Therefore, it would be advantageous to modify the execution control possibilities in the application in such a way as to force students to do less actions and more reflection. One way could be to limit the number of executions via score penalties. Furthermore, the maps of the levels that are targeting test-based concepts could be modified to make conceptual bypassing strategies more costly in order to limit them. This could be done by increasing the number of possible paths and by hiding the location of random objects for as long as possible to increase the execution time of each trial. Then, students' programs could be analysed to identify the most common errors, which could be taken into account when designing the game's levels. Finally, we could evaluate student learning by proposing a pre-test in Scratch followed by a post-test evaluating the same concepts in Python based on a language-independent standard CS1 knowledge assessment framework [45].

ACKNOWLEDGMENTS

This research was funded by *Region Bretagne* and *Université de Bretagne Occidentale*. We acknowledge students and teachers who participated in the game evaluation sessions. We thank Ghislaine Gueudet, Cédric Fluckiger and Sébastien Lallé for their advice and careful proofreading.

REFERENCES

[1] Algorea 2023. *Algoréa home page.* Retrieved Jun 30, 2023 from https://www.algorea.org

[2] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 522–527. https://doi.org/10.1145/2676723.2677258

[3] Julian Alvarez. 2007. *Du jeu vidéo au serious game : approches culturelle, pragmatique et formelle.* Ph. D. Dissertation. Université de Toulouse. https://hal.archives-ouvertes.fr/tel-01240683

[4] Emma Andrews, David Bau, and Jeremiah Blanchard. 2021. From Droplet to Lilypad: Present and Future of Dual-Modality Environments. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–2. https://doi.org/10.1109/vl/hcc51201.2021.9576355

[5] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2015. From Scratch to "Real" Programming. *ACM Transactions on Computing Education* 14, 4, Article 25 (feb 2015), 15 pages. https://doi.org/10.1145/2677087

[6] Michèle Artigue. 1988. Ingénierie didactique. *Recherches en didactique des mathématiques* 9, 3 (1988), 281–308.

[7] A. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura. 2017. BlockPy: An Open Access Data-Science Environment for Introductory Programmers. *Computer* 50, 05 (may 2017), 18–26. https://doi.org/10.1109/mc.2017.132

[8] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children* (Boston, Massachusetts) *(IDC '15)*. Association for Computing Machinery, New York, NY, USA, 445–448. https://doi.org/10.1145/2771839.2771875

[9] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (may 2017), 72–80. https://doi.org/10.1145/3015455

[10] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[11] Annie Bessot. 2003. Une introduction à la théorie des situation didactiques. *Les cahiers du laboratoire Leibniz* 91 (2003), 1–28.

[12] Matthieu Branthôme. 2021. Apprentissage de la programmation informatique à la transition collège-lycée. *STICEF (Sciences et Technologies de l'Information et de la Communication pour l'Éducation et la Formation)* 28, 3 (2021), 1–35. https://doi.org/10.23709/sticef.28.3.1

[13] Matthieu Branthôme. 2022. Pyrates: A Serious Game Designed to Support the Transition from Block-Based to Text-Based Programming. In *Educating for a New Future: Making Sense of Technology-Enhanced Learning Adoption*, Isabel Hilliger, Pedro J. Muñoz-Merino, Tinne De Laet, Alejandro Ortega-Arranz, and Tracie Farrell (Eds.). Springer International Publishing, Cham, 31–44. https://doi.org/0.1007/978-3-031-16290-9_3

[14] Guy Brousseau. 1998. *Théorie des situations didactiques*. La Pensée sauvage, Grenoble.

[15] Codecombat 2023. *Code Combat home page*. Retrieved Jun 30, 2023 from https://codecombat.com

[16] Codemonkey 2023. *CodeMonkey home page*. Retrieved Jun 30, 2023 from https://www.codemonkey.com

[17] Codingpark 2023. *CodingPark home page*. Retrieved Jun 30, 2023 from https://codingpark.io

[18] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 208–212. https://doi.org/10.1145/1999747.1999807

[19] Raymond Duval. 2006. A Cognitive Analysis of Problems of Comprehension in a Learning of Mathematics. *Educational Studies in Mathematics* 61, 1 (01 Feb 2006), 103–131. https://doi.org/10.1007/s10649-006-0400-z

[20] edublocks 2023. *Edublocks home page*. Retrieved Jun 30, 2023 from https://app.edublocks.org/

[21] Stephen H. Edwards. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 26–30. https://doi.org/10.1145/971300.971312

[22] David Ginat, Eyal Shifroni, and Eti Menashe. 2011. Transfer, Cognitive Load, and Program Design Difficulties. In *Informatics in Schools. Contributing to 21st Century Education*, Ivan Kalaš and Roland T. Mittermeir (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–176. https://doi.org/10.1007/978-3-642-24722-4_15

[23] Joint Task Force on Computing Curricula, ACM and IEEE CS. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2534860

[24] Majeed Kazemitabaar, Viktar Chyhir, David Weintrop, and Tovi Grossman. 2022. CodeStruct: Design and Evaluation of an Intermediary Programming Environment for Novices to Transition from Scratch to Python. In *Interaction Design and Children* (Braga, Portugal) *(IDC '22)*. Association for Computing Machinery, New York, NY, USA, 261–273. https://doi.org/10.1145/3501712.3529733

[25] Jonathan M. Kevan and Paul R. Ryan. 2016. Experience API: Flexible, Decentralized and Activity-Centric Data Collection. *Technology, Knowledge and Learning* 21, 1 (01 Apr 2016), 143–149. https://doi.org/10.1007/s10758-015-9260-x

[26] B. Khazaei and M. Jackson. 2002. Is there any difference in novice comprehension of a small program written in the event-driven and object-oriented styles?. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. 19–26. https://doi.org/10.1109/HCC.2002.1046336

[27] Tobias Kohn. 2017. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. Ph. D. Dissertation. ETH Zurich, Zürich. https://doi.org/10.3929/ethz-a-010871088

[28] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2818314.2818331

[29] Yuhan Lin and David Weintrop. 2021. The landscape of Block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages* 67 (2021), 1–18. https://doi.org/10.1016/j.cola.2021.101075

[30] Yoshiaki Matsuzawa, Takashi Ohata, Manabu Sugiura, and Sanshiro Sakai. 2015. Language Migration in Non-CS Introductory Programming through Mutual Language Translation Environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 185–190. https://doi.org/10.1145/2676723.2677230

[31] Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 1–8. https://doi.org/10.1109/FIE.2014.7044420

[32] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/1999747.1999796

[33] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. 2010. Learning Computer Science Concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research* (Aarhus, Denmark) *(ICER '10)*. Association for Computing Machinery, New York, NY, USA, 69–76. https://doi.org/10.1145/1839594.1839607

[34] Jean Piaget. 1975. *L'équilibration des structures cognitives*. Presse Universitaire de France, Paris.

[35] Thomas W. Price and Tiffany Barnes. 2015. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 91–99. https://doi.org/10.1145/2787622.2787712

[36] Pypl 2023. *PYPL - PopularitY of Programming Language*. Retrieved Jun 30, 2023 from https://pypl.github.io/PYPL.html

[37] PyratesHP 2023. *Pyrates home page*. Retrieved Jun 30, 2023 from https://py-rates.org

[38] PyratesPG 2023. *Pyrates pedagogical guide*. Retrieved Jun 30, 2023 from https://py-rates.org/guide/EN/

[39] Pyscripter 2023. *Pyscripter Github page*. Retrieved Jun 30, 2023 from https://github.com/pyscripter/pyscripter

[40] Yizhou Qian and James D Lehman. 2016. Correlates of success in introductory programming: A study with middle school students. *Journal of Education and Learning* 5, 2 (2016), 73–83. https://doi.org/10.5539/jel.v5n2p73

[41] Reeborg 2023. *Reeborg's World home page*. Retrieved Jun 30, 2023 from https://reeborg.ca

[42] Virginia Richardson. 2005. Constructivist teaching and teacher education: Theory and practice. In *Constructivist teacher education: Building a World of New Understandings*. Falmer Press, 3–14.

[43] William Robinson. 2016. From Scratch to Patch: Easing the Blocks-Text Transition. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (Münster, Germany) *(WiPSCE '16)*. Association for Computing Machinery, New York, NY, USA, 96–99. https://doi.org/10.1145/2978249.2978265

[44] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19 (nov 2013), 40 pages. https://doi.org/10.1145/2534973

[45] Allison Elliott Tew and Mark Guzdial. 2011. The FCS1: A Language Independent Assessment of CS1 Knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 111–116. https://doi.org/10.1145/1953163.1953200

[46] Adilson Vahldick, Antonio José Mendes, and Maria José Marcelino. 2014. A review of games designed to improve introductory computer programming competencies. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 1–7. https://doi.org/10.1109/FIE.2014.7044114

[47] Feng Wang and Michael J. Hannafin. 2005. Design-based research and technology-enhanced learning environments. *Educational Technology Research and Development* 53, 4 (2005), 5–23. https://doi.org/10.1007/BF02504682

[48] Pierre Weill-Tessier, Charalampos Kyfonidis, Neil Brown, and Michael Kölling. 2022. Strype: Bridging from Blocks to Python, with Micro:Bit Support. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 585–586. https://doi.org/10.1145/3502717.3532155

[49] David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (jul 2019), 22–25. https://doi.org/10.1145/3341221

[50] David Weintrop, Alexandria K. Hansen, Danielle B. Harlow, and Diana Franklin. 2018. Starting from Scratch: Outcomes of Early Computer Science Learning Experiences and Implications for What Comes Next. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 142–150. https://doi.org/10.1145/3230977.3230988

[51] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-Based and Text-Based Programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/2787622.2787721

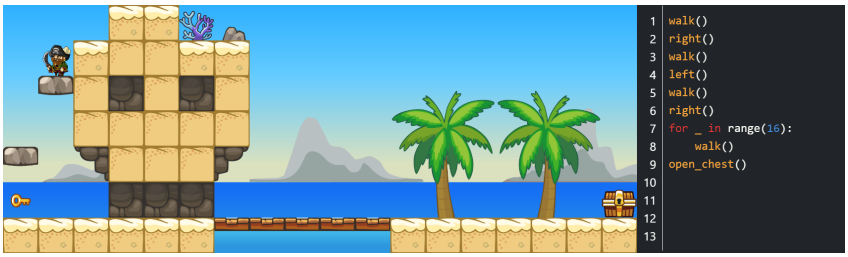
[52] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education* 18, 1, Article 3 (oct 2017), 25 pages. https://doi.org/10.1145/3089799

[53] David Weintrop and Uri Wilensky. 2019. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education* 142 (2019), 1–17. https://doi.org/10.1016/j.compedu.2019.103646

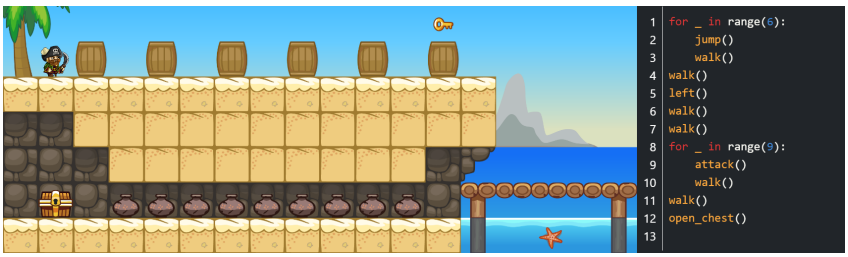
[54] Garry White and Marcos Sivitanides. 2005. Cognitive Differences Between Procedural Programming and Object Oriented Programming. *Information Technology and Management* 6, 4 (01 Oct 2005), 333–350. https://doi.org/10.1007/s10799-005-3899-2

[55] Zehui Zhan, Luyao He, Yao Tong, Xinya Liang, Shihao Guo, and Xixin Lan. 2022. The effectiveness of gamification in programming education: Evidence from a meta-analysis. *Computers and Education: Artificial Intelligence* 3 (2022), 100096. https://doi.org/10.1016/j.caeai.2022.100096

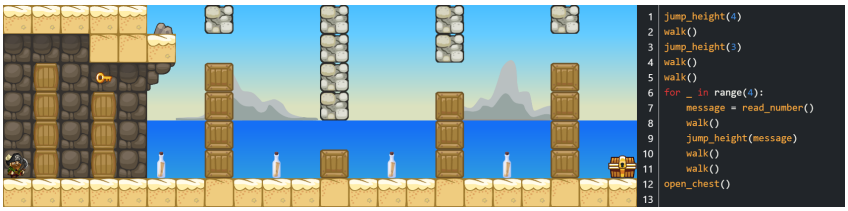
A THE DIFFERENT LEVELS OF PYRATES APPLICATION



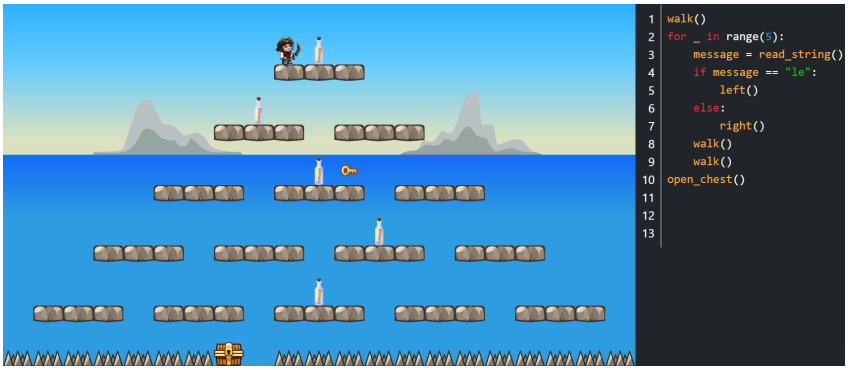
(a) Level 1



(b) Level 2

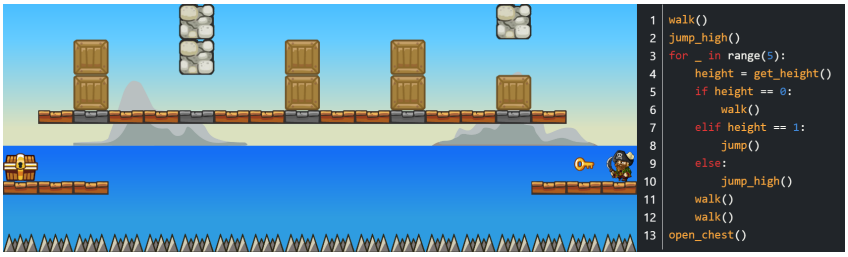


(c) Level 3: the height of each box stacks is random, the messages in the bottles indicate their height.



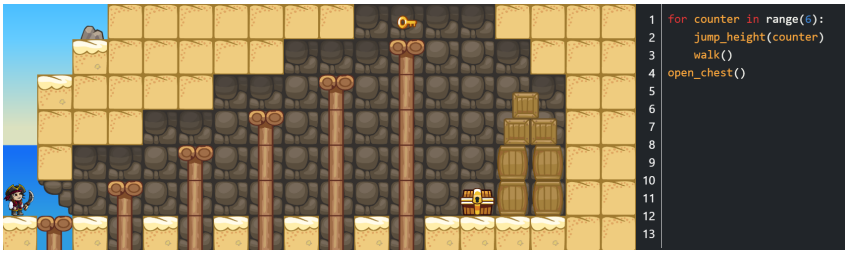
```
walk()
for _ in range(5):
    message = read_string()
    if message == "le":
        left()
    else:
        right()
    walk()
    walk()
open_chest()
```

(d) Level 4: the location of the chest and the key are random, the messages in the bottles indicate the path to follow.



```
walk()
jump_high()
for _ in range(5):
    height = get_height()
    if height == 0:
        walk()
    elif height == 1:
        jump()
    else:
        jump_high()
    walk()
    walk()
open_chest()
```

(e) Level 5: the height of the box stacks is random.



```
for counter in range(6):
    jump_height(counter)
    walk()
open_chest()
```

(f) Level 6

```
for counter in range(2,9):
    shoot(counter)
    turn()
shoot(3)
```

(g) Level 7: different sets of coconuts appear on both sides, the otherpirate must never be reached.

(h) Level 8: the strength of the barrel, the number of pots and the location of the chest are random.

Fig. 12. The different levels of Pyrates application: maps and winning procedures.

Received 28 June 2023; revised 31 October 2023; accepted 30 November 2023