CodeOrama: A two-dimensional visualization tool for Scratch code to assist young learners' understanding of computer programming

Anastasios Ladias, Aristotelis Mikropoulos, Demetrios Ladias, Ioanna Bellou
ladiastas@gmail.com, amikrop@gmail.com, ladimitr@gmail.com, ibellou@uoi.gr

The Educational Approaches to Virtual Reality Technologies Laboratory, University of Ioannina, Greece

Abstract

This paper reports on CodeOrama, a visualization tool that displays the entire source code in a two-dimensional representation created to support the representation of a complex code in block-based programming environments, like Scratch, by using a two-dimension table. CodeOrama can be used by the students for the development of their programs as well as by the teachers in order to support their specific pedagogical choices. This paper demonstrates the main representation capabilities that CodeOrama provides, i.e. a) direct information about the program size, b) identification of the code parts with algorithmic and communicational overload, c) code modularization by procedures, messages, clones, d) ways of interaction between scripts or other objects (based on polling and interrupt techniques), e) program flow representation, f) segment points where code interacts with the user, g) data representation and the kind of programming that was adopted (structured, parallel, event driven). The results of a pilot study concerning 87 programming projects with CodeOrama, in the context of the final phase of the Greek Competition of Educational Robotics, showed the potential of the tool but equally highlight some limitations and students' difficulties such as a manual and time-consuming burden during its creation process.

Keywords: CodeOrama, Scratch, visual programming, educational robotics

Introduction

The objective behind engaging students in project-based learning is to develop their methodological skills, for example, data processing, algorithm design and implementation, solution modelling, creativity and innovation. Furthermore, advanced competencies gained through project-based learning also include critical and analytical thinking, the ability to synthesize and build communication and cooperation skills. To this end, as students learn computer programming, they develop design skills, build on existing knowledge in a stepping-stone process and in turn acquire algorithmic thinking which in turn give them the confidence to program computing devices (Balanskat & Engelhardt, 2015; Topali & Mikropoulos, 2018).

However, and especially in STEM/STEAM based projects, the code often required is long and complex. The need therefore emerges to design programming interfaces that enable novice programmers to develop, manage and control programs. Such environments are known as visual, block-based programming, such as Scratch, Appinventor, StarlogoTNG, TurtleArt, Kodu, etc. It is important to note that the aim here is not for students to acquire specialized knowledge of transient programming interfaces. Instead, thanks to a spiral approach, students initially participate, gain experience and learn through enactive representations, thereby focusing on timeless programming concepts (Grover, Cooper, & Pea, 2014; Grover & Basu, 2017). Nevertheless, and despite student friendliness, difficulties are often encountered in visual programming interfaces and thus alternative ways to illustrate code representations are required to aid student comprehension (Bau, Bau, Dawson & Pickens, 2015).

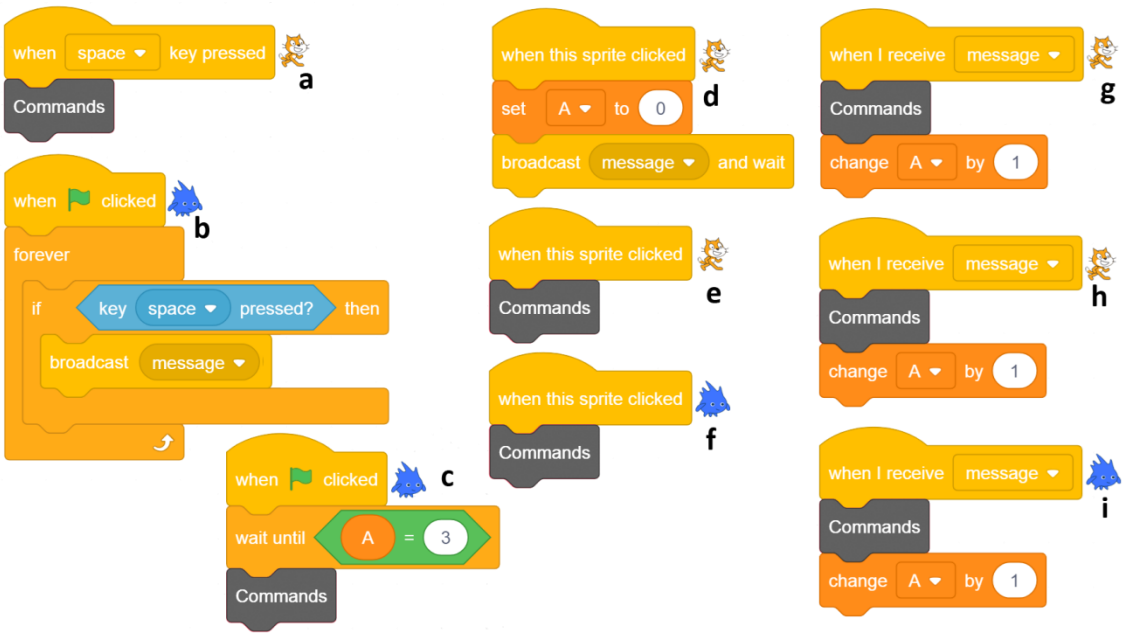

Figure 1. Example scenarios of a Scratch-3 program

A popular object based visual programming environment mentioned above is Scratch. (http://scratch.mit.edu). Created by the Lifelong Kindergarten Group at the Media Laboratory of MIT it is meant to be a rich multimedia system for novice programmers. In a Scratch environment, objects (that appear as roles and sprites in a scene) are provided, the behaviour of which is determined by programs consisting of autonomous code blocks (modules) (Resnick et al., 2009). A program may include several of these autonomous code blocks, also known as "programming scenarios" (hereafter referred to as "scenarios"). Next, the scenarios' code is activated by events. Each event corresponds to a "state" that the system reaches, and an event can be any external activity detected by the computer's peripherals (e.g., a mouse click), or internal messages emitted by other scenarios. As such, because Scratch programs manage events, they are considered event-driven programming examples. As the event occurs, the code describing the behavior that the object must perform is stacked under the original command (Figure 1). Figure 1 shows some example scenarios of a Scratch-3 program, belonging to different objects (a, d, e, g, h and b, c, f, i), asynchronously executed (d, e and f), responding to requests made by the same events, in parallel execution (b, c or d, e or g, h, i), detecting the same event with the interrupt or polling method (a, b), checking the complete execution of certain threads (d and g, h, i) and inter-communication among scenarios through message passing (e.g. b, g, h, i) or memory sharing (d, g, h, i, c).

It is equally possible to execute a number of scenarios in parallel (Kafai, 1995), i.e., scenarios b, c or d, e and g, h, i in Figure 1, otherwise known as object behaviour programming. This concurrent scenario execution (i.e., parallel programming) creates inter-scenario communication requirements, which can be achieved by detecting and fulfilling requests. According to Andrews and Schneider (1983) there are only two request detection and response mechanisms: message passing, as in between scenarios d and g, h, i or b and g, h, i (Figure 1) and shared memory.

In the message passing mechanism, a scenario is detected with the interrupt method, whereas in the shared memory mechanism (using global variables, e.g., variable A between scenarios d, g, h, i, c) detection is achieved though the polling method (e.g., scenarios b and c in Figure 1) (Ladias, Ladias & Karvounidis, 2019). These scenario communication mechanisms are indeed available in Scratch however are not obvious to the programmer, who has to rely on his or her own mental representations (Nikolos & Komis, 2015).

It is well known that communication and synchronization in environments with parallel programming capabilities is complex (Ben-Ari, 2006). Over the last decade, algorithm and program visualization systems have been used in introductory programming in order to visualize and/or simulate what is happening during the execution of an algorithm or a program (Vrachnos & Jimoyiannis, 2014). A range of visualization systems were designed with the aim to demonstrate the fundamental operations of specific algorithms and help students to think in an abstractive way, to construct effective mental models about programming structures and influence their algorithmic thinking (Berry & Kölling, 2016; Halim, 2015; Guo, 2012; Sorva & Sirkiä, 2010; Vrachnos & Jimoyiannis, 2014).

Velazquez-Iturbide, Hernan-Losada & Paredes-Velasco (2017) reported that program visualization promote students' motivation and engagement in programming. Toward this perspective, the need for an integrated representation of complex code in visual, block-based programming environments, like Scratch, led to the development of CodeOrama, a tool for two-dimensional code visualization, which aims to facilitate the teaching and understanding of visual programming.

CodeOrama

CodeOrama uses a two-dimensional table to illustrate all scenarios of all objects within a program (Figure 2). Considering a request is recognized, the rows display the objects involved in a programming problem and the columns present the states in which the objects exist. A CodeOrama is created by the student/programmer and the code output is submitted to the teacher. Every CodeOrama cell (e.g., cell B3 in Figure 2) includes scenario(s) describing the behaviour of the corresponding object (column), in its corresponding state (row).

Therefore, the coordinates of every scenario belonging to a cell are defined by the command that determined the state (row) and the icon of the object accompanying it (column). This process helps the student/programmer to detect a code's intricate detail and directly understand its coordinates, preventing the student from getting disoriented because of the overwhelming amount of code detail. At the same time, it makes it easier for the teacher to directly monitor the programmer's steps.

Links that may exist between scenarios (e.g., arrow from cell C5 to cell D6, in Figure 2), show the communication between code modules and are represented as arrows indicating the potential program flow during execution. Similar to the color encoding format of Scratch, the colors of the arrows indicate the communication type. The arrows between code threads concern either communication synchronization or clone creation. Communication synchronization is achieved via message sending (arrow starting from cell C1, ending in cells B2 and D2, in Figure 2) or via the method of shared memory (arrow starting from cell A4, ending in cell B3, in Figure 2). In the shared memory method, problems like deadlock may occur due to the use of universal variables serving as semaphores. To illustrate the interface between the procedure call and its definition, the definition is placed to the right of its procedure call, thus creating a hierarchy (e.g., in cell A1, the scenario calling the procedure is on the left, while the procedure definition on the right).

At this point, it is important to recall that a program consists of an algorithm and data (Wirth, 1985). To solve a real-life problem, students create a model/program in which reality is abstractly represented, in other words a set of data undergoes processing via the use of specific algorithms. To meet the need for data representation in CodeOrama, an extra row is added at the top of the table which shows the local variables (internal to it) above each object (e.g., ball, robot1, robot2 in Figure 3), whereas the universal variables are placed to the left of the data row (Figure 3). Additionally, there is a "visual grouping" of subsets corresponding to specific logical structures, in the form of "logical records" (such as the x, y couple of a sprite's coordinates). Also, the data row includes as many objects as possible, also known as multimedia data. Consequently, the programmer has the ability to easily note each object's potential for object animation.

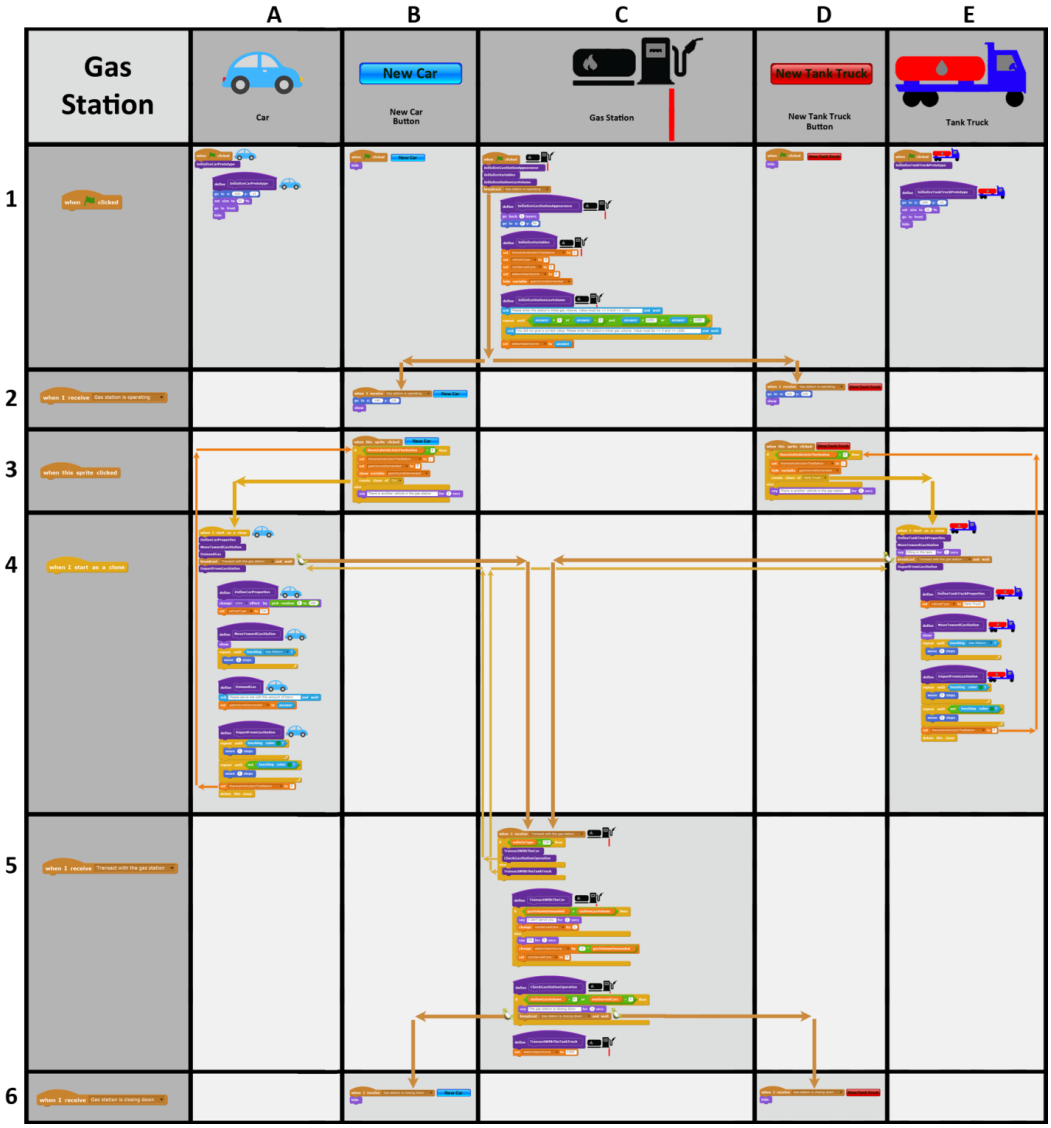

Figure 2. A CodeOrama example

Representation capabilities using CodeOrama

CodeOrama is able to display the code design methodology and design templates of each program. As a result, the program's reader can estimate its direct quantitative properties, like code size (proportional to CodeOrama's surface area) and code "density", which are inversely proportional to empty cell count (like cells C2, C3, C4 and C6 corresponding to object GasStation in Figure 2). It is noted that in standard Scratch, the programmer can only view the code of one object at a time, which results in the restricted view of the program's entire source code and the interaction between the objects.

CodeOrama clarifies the code's modularity, as well as its granularity (Figure 4). Granularity is defined as the continuous segmentation process of a program module and the simplest module is considered an elementary logical entity. Using the code's modularity and the hierarchical structure of programs, because of the abstraction of the hierarchy's higher structures, information hiding and the lower hierarchical organization can be observed.

The spatial layout of objects in the horizontal axis of CodeOrama, if performed properly, can reduce the openings of linked rows, resulting in less visual noise for CodeOrama's reader and shows similar elements (e.g., symmetry) arising from the problem. This symmetry is shown in Figure 2 and results in a proper object layout which in turn facilitates comprehension of the code's functionality.

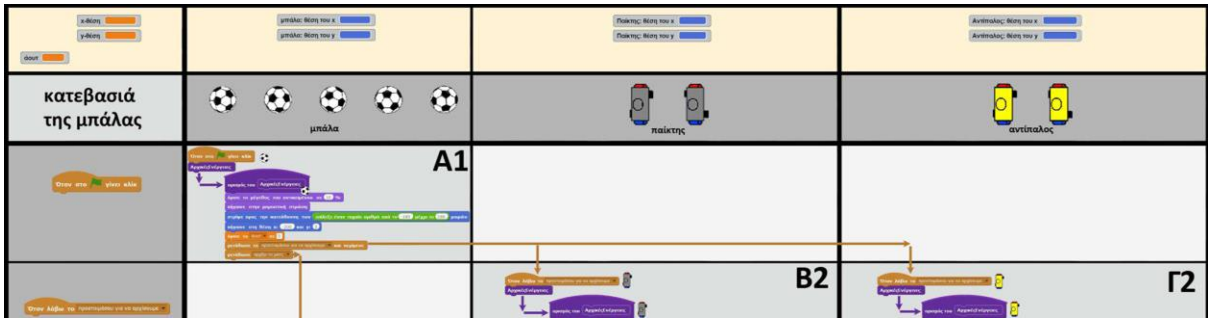

Figure 3. CodeOrama part with the data representation row

Figure 4. Modularization example inside the only scenario created with the use of procedure calls, across four hierarchy levels highlighting the tree structure

The proper spatial layout of states (corresponding to row succession) intends to show the temporal progress of the entire program's operation. The discrete state corresponds to the moment the event occurs and is represented in CodeOrama's vertical dimension. This means that the system's temporal progress is evidenced at the vertical axis. The CodeOrama designer must guide the flow from top to bottom although the nature of certain problems may require the opposite a bottom-top approach. In Figure 2 the flow between states is defined by the arrows which correspond to message transmissions that in turn activate the next code segments to be executed, as well as by the scalable placement of the requested procedures.

The illustrated connections facilitate the comprehension and handling of long threads (see green line on Figure 5) that may be triggered by hierarchically structured procedure calls, message transmission and clone creation. Moreover, the connection representation helps clarify any alternative loops (see blue line on Figure 5) that may be created by recursive procedures, messages received by a transmitting scenario and self-reproducing clones (Karvounidis, Ladias, Ladias & Douligeris, 2019). Any temporal flow discontinuity of segment points (e.g., the flow interruption between the second and third state on Figure 2) are caused by the fact that the system here waits for a user action in order to

proceed from one state to the next. Therefore, this flow discontinuity in CodeOrama shows at which segment point in the program there is interaction between the program and the user.

In CodeOrama, the program's qualitative characteristics become easily distinguished, thanks to Scratch's colour encoding format. Purple segments declare information output channels towards the user through the computer screen, while light blue segments show request detection processes, mainly from the outer environment, meaning possible data input though peripherals or data acquisition through robotic sensors. The color of the initial commands (as in every Scratch scenario) of scenarios declares the origin of events. Segments accumulating orange-colored commands show data processing and if seen at the start of the program, they are possibly initializing the value of variables/constants. Purple (Scratch 2) or Red (Scratch 3) colored code segments indicate procedure usage and therefore modular programming.

Code segments combining yellow (use of selection or/and iteration programming structure) and mint green (logical operator usage) indicate segments of high algorithmic load. Also, segments accumulating arrows – links indicate communicational load, where the link color shows the type of communication (message or shared memory communication). Finally, messages sent by an object to other objects and messages-notifications sent by an object to the user, shown on the screen, are clearly separated thanks to color encoding.

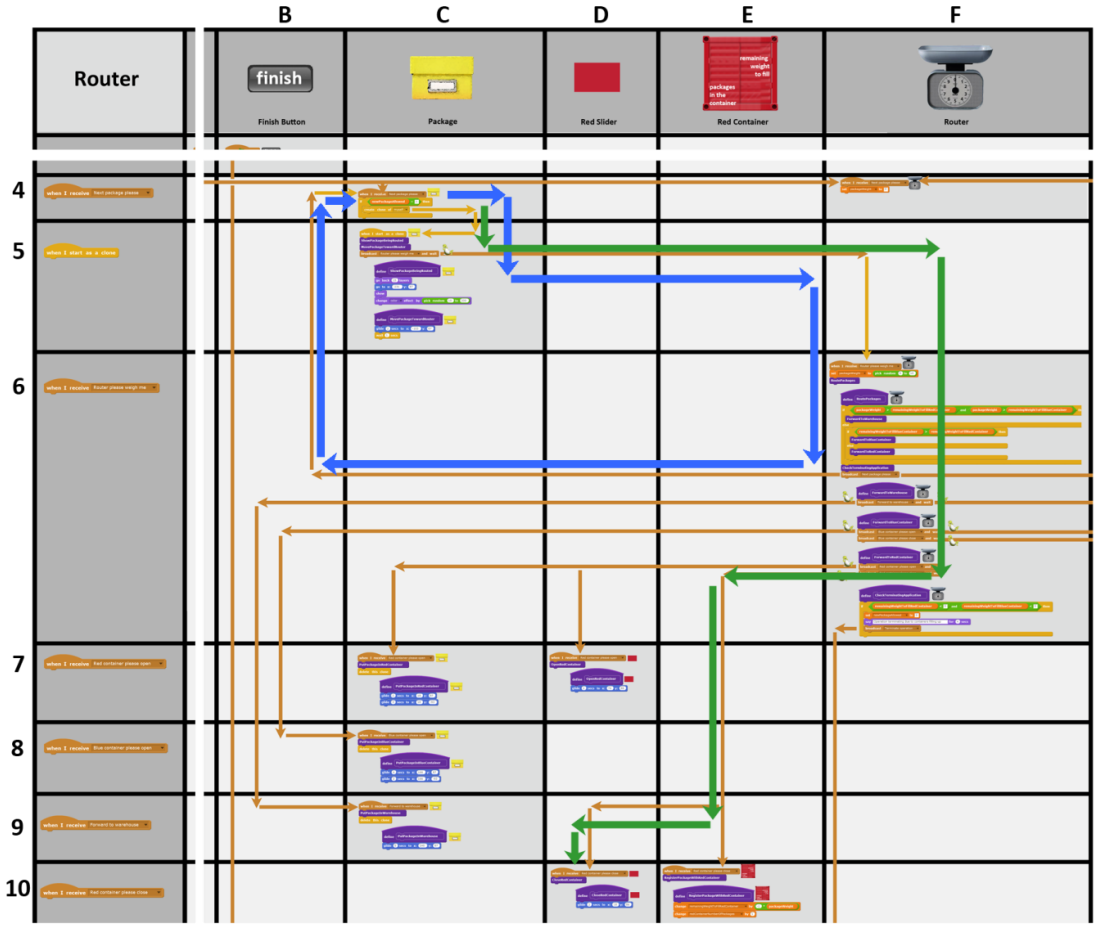

Figure 5. Example of long thread (green line) and alternative loop (blue line) in CodeOrama

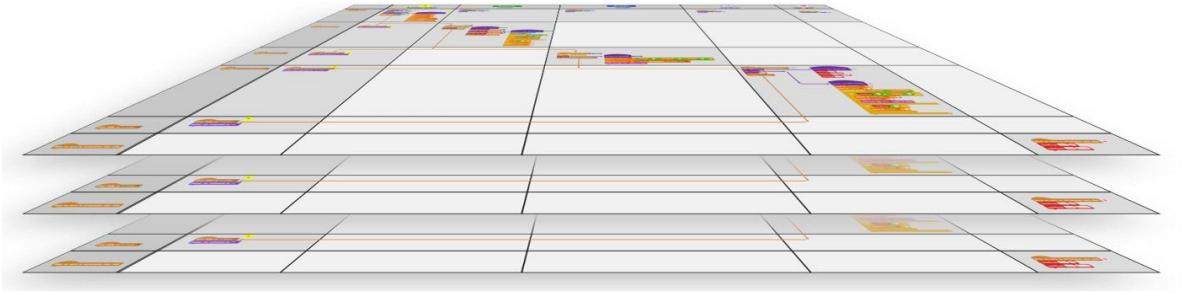

Figure 6. Example of a multilevel CodeOrama, each level including scenarios serving similar purposes

A secondary grouping of scenarios, based on the principal of separation of responsibilities, can decongest visual noise in complex CodeOramas, by producing multilevel CodeOramas (Figure 6). Here, scenarios belonging to the same cell but serving different purposes are distributed on the additional, third dimension axis e.g., scenarios dedicated to the application's audio interface could exist on one level, scenarios related to the visual interface with keys and animation could exist on another level and scenarios solving the algorithmic problem could exist on yet a different level.

The pilot study

Applying CodeOrama to the national contest of educational robotics

The objective of the present pilot study was to evaluate CodeOramas developed by students. The CodeOramas identified for the study were those submitted as a requirement in the World Robot Olympiad-Hellas (WRO-Hellas).

WRO-Hellas organizes two competitions annually: (a) The Panhellenic Competition of Educational Robotics and (b) the National WRO Competition that determines which teams will qualify to participate in its equivalent international competition. CodeOrama applies to the Open Category for Elementary age groups of the Panhellenic Competition, being held since 2015. In the Open Category teams of 3-6 students use the LEGO Education WeDo educational robotics package, in addition to programming environment Scratch. Also, in the Open Category, the main criterion for every entry is that each project must have at least two automations, one of which is required to present a simulation on the computer screen and synchronized with the automation's execution. The code required for this automation and simulation combination to operate is generally the most complex part of the project.

To achieve this objective, the teams use the CodeOrama tool during the evaluation process. Due to the two required automations and the respective two required Scratch programs, many teams choose to design two CodeOramas - one for the automation and another for the simulation – in order to potentially communicate with each other and form a single system. The CodeOrama tool (which aims at better code representation) was optionally introduced, in a pilot form for first time in the 2018 competition, in which 386 teams took part. As an additional incentive to adopt the CodeOrama tool, a prize for "best CodeOrama" is awarded to the participating teams.

Sample and procedure

In the 2019 competition 357 teams participated in the Open Category for Elementary age groups of the Panhellenic Competition. The sample of the present study consisted of 87 CodeOramas developed by the participating teams at the competition's final stage in Athens, on March 16, 2019. 87 projects that qualified from the regional competitions were encouraged to develop CodeOramas. Thirty-four percent of them originated from Attica, 11% from Central Macedonia, 8% from Thessaly, 7% from Northern Aegean, Crete, and Western Greece and the others were distributed in the rest of Greece. Fifty-eight of the projects came from public schools, 20% from private schools, 11% from independent

Table 1. Evaluation criteria for the Panhellenic Competition of Educational Robotics

Evaluation criteria	Score
Project idea	
1. Creativity, research, and development of the idea	15
2. Development and quality of the solution	25
Educational Robotics / Automations	
3. Mechanical construction, aesthetics	25
4. Mechanical performance, automations' functionality	25
Scratch code	
5. CodeOrama – Optical representation of the code	20
Virtual World / Code and Scenes	
6. Logic, complexity of the code and automations	25
7. Automations' animated representation, interface, aesthetics	25
Presentation	
8. Presentation, communication skills, collaboration	30
9. Infographics, Video	10
Total	

teams, and the rest from other organizations. The teams of the 87 projects consisted of 5-6 students, ages 10-12. Table 1 shows the evaluation criteria of the projects. These criteria were established based on project evaluations from 2015. The 87 projects were evaluated by three Computer Science teachers with more than 15 years of experience in Scratch programming and educational robotics. The CodeOramas were also evaluated independently by three teachers. Because of the rigorous structure of the CodeOramas, the consensus of the three evaluators was set at 90% and above.

Projects that hadn't submitted a CodeOrama were rated as "sans". Projects whose code representation did not comply to the CodeOrama development rules and presented incorrect pieces of code were rated as "incorrect". Projects with a representation partly following the CodeOrama rules were rated as "incomplete". These included CodeOramas that randomly presented some parts of the code on the CodeOrama framework. Projects applying CodeOrama's development rules in a limited capacity were rated "rudimentary". Projects providing a CodeOrama with a significant range but without procedures and data were rated as "basic". Projects with a CodeOrama following all required procedures were rated as "complete" and projects with a compliant CodeOrama to its procedures and data representation were rated as "excellent".

Results

Figure 7 shows the evaluation results regarding the 87 CodeOramas qualified in the 2019 regional competitions. It is worth noting that only five CodeOramas were rated as excellent. Of the 87 teams, the majority, namely 25 CodeOramas were incomplete and 16 did not create a CodeOrama. Figure 8 shows an excellent rated CodeOrama for a project entitled "a smart harbour".

Three of this paper's authors interviewed the coaches of the 87 teams which highlighted the benefits of the use of CodeOrama in project-based learning. The coaches confirmed that CodeOrama facilitates:

- complete viewing of the entire code but equally focuses on the details of specific code parts
- reader's comprehension of the code segment that corresponds to a certain behaviour for a certain object, in a certain state
- regulation of the program flow

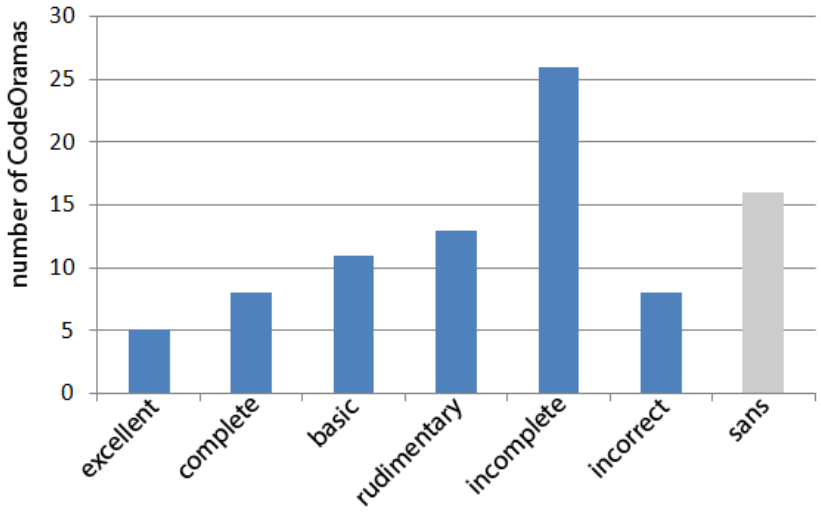

Figure 7. Classification results of CodeOramas developed by 87 projects in regional competitions

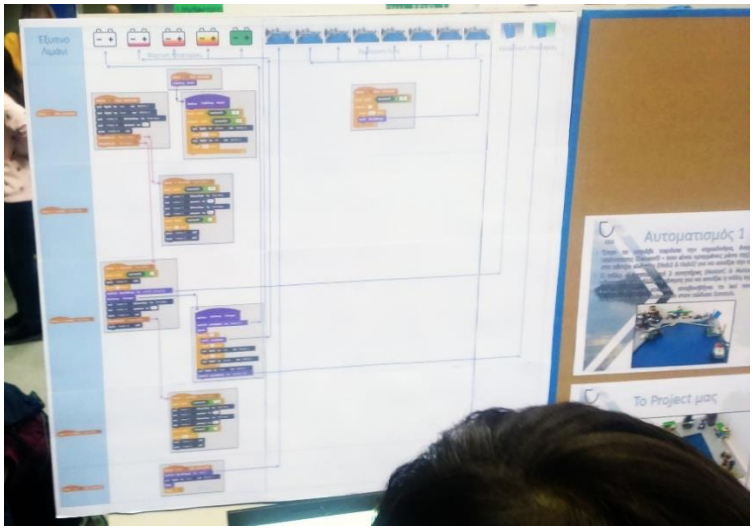

Figure 8. The complete CodeOrama for the "a smart harbour" project

- detecting program segments with algorithmic or communicational load as well as those where program-user interaction takes place
- duty separation between parts of complex projects
- customization at the code modularization and granularity level
- means used for code modularization (procedures, messages, clones)
- application of the code's hierarchical analysis and detail masking
- program philosophy (structured or event-driven programming)
- application of serial or/and parallel programming of an object and between different objects.

Despite the tool's many advantages, it is worth noting that both the teachers and coaches of the participating teams reported that creating a CodeOrama manually is a challenging and time-consuming process and might discourage instructors to adopt it. Several coaches however mentioned that of the students who did in fact engage in the CodeOrama's manual development, it helped them understand programming in greater depth and possibly corrected or/and improved their codes and coding skills. All teachers and coaches agreed that it would be extremely useful to develop software that would automatically produce a CodeOrama out of Scratch source code.

Conclusion

The need for efficient management of complex source code in visual, block-based programming environments, like Scratch, led to the invention of CodeOrama, a two-dimensional visualization tool presenting the entire source code of a program. CodeOrama has a two-fold objective in that it is a tool for both programming development and evaluation. The findings of the present study suggest that CodeOrama can be used in practice as a program visualization tool supporting both, teachers' educational design and students' engagement into programming activities (Chang, Tsai, & Chin, 2017; Sorva, Karavirta, & Malmi, 2013; Vrachnos & Jimoyiannis, 2014). As far as Scratch programming it regards, the visualization through CodeOrama could help students' developing of computational thinking skills, due to their representation capabilities of program decomposition, program patterns, algorithmic structures and data to be processed.

References

Andrews, G.R., Schneider, F.B. (1983). Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15, 3-43.

Balanskat, A., & Engelhardt, K. (2015). *Computing our Future: Computer Programming and Coding – Priorities, School Curricula and Initiatives across Europe*. Brussels: European Schoolnet.

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. (2015). Pencil code: block code for a text world. In M. Umaschi Bers & G. Revelle (eds.), *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 445-448). NY: ACM.

Bell, T., Witten, I., Fellows, M., Adams, R., & McKenzie, J. (2006). *Computer Science Unplugged – An enrichment and extension programme for primary-aged children*. NZ: Computer Science Unplugged. Retrieved on January 14, 2021 from http://csunplugged.org.

Ben-Ari, M. (2006). *Principles of concurrent and distributed programming*. Harlow: Prentice-Hall.

Berry, M., & Kölling, M. (2016). Novis: A notional machine implementation for teaching introductory programming. *Proceedings of the Fourth International Conference on Learning and Teaching in Computing and Engineering (LATICE 2016).* https://doi.org/10.1109/LaTiCE.2016.5.

Chang, C.-K., Tsai, Y.-T., & Chin, Y.-L. (2017). A visualization tool to support analyzing and evaluating Scratch projects. *Proceedings of the 6th IIAI International Congress on Advanced Applied Informatics* (pp. 498-502). Hamamatsu, Japan: IEEE. https://doi.org/10.1109/IIAI-AAI.2017.83.

Grover, S., & Basu, S. (2017). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 267-272). NY: ACM.

Grover, S., Cooper, S., & Pea, R. (2014). Assessing computational learning in K-12. *Proceedings of the 2014 Innovation and Technology in Computer Science Education Conference* (pp. 57-62). NY: ACM.

Guo J. P. (2012). Online Python Tutor: Embeddable Web-based program visualization for CS education. *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education* (pp. 579-584). New York: ACM.

Halim, S. (2015). VisuAlgo – Visualizing Data Structures and Algorithms through Animation. *Olympiads in Informatics*, 9, 243–245. DOI: http://dx.doi.org/10.15388/ioi.2015.20.

Kafai, Y. (1995). *Minds in play: Computer game design as a context for children's learning*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Ladias, A., Ladias, D., & Karvounidis, T. (2019). Categorization of requests detecting in Scratch using the SOLO taxonomy. *Proceedings of the 4th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference* (pp. 1-7). Piraeus, Greece: IEEE.

Karvounidis, T., Argyriou, I., Ladias, A., & Douligeris, C. (2017). A design and evaluation framework for visual programming codes. *Proceedings of the IEEE Global Engineering Education Conference* (pp. 999-1007). Athens: IEEE.

Karvounidis, T., Ladias, A., Ladias, D., & Douligeris, C. (2019). Kinds of loops implemented with using messages in Scratch and the SOLO Taxonomy. *Proceedings of the 4th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference* (pp. 1-5). Piraeus, Greece: IEEE.

Nikolos, D., & Komis, V. (2015). Synchronization in Scratch: A case study with education science students. *Journal of Computers in Mathematics and Science Teaching*. 34(2), 223-241.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60-67.

Sorva, J., Karavirta, V., & Malmi, L. (2013). A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*, 13(4), 15:1 -15:64.

Sorva, J. & Sirkiä, T. (2010). UUhistle: a software tool for visual program simulation. *Proceedings of the 10th International Conference on Computing Education Research* (pp. 49-54). New York: ACM.

Topali, P., & Mikropoulos, T.A. (2018). Digital learning objects for teaching computer programming in primary education. In M. Tsitouridou, J. A. Diniz & T. Mikropoulos (eds.), *Technology and Innovation in Learning, Teaching and Education* (pp. 256-266). Cham: Springer.

Velazquez-Iturbide, A. J., Hernan-Losada, I., & Paredes-Velasco, M. (2017). Evaluating the effect of program visualization on student motivation. *IEEE Transactions on Education*, 60(3), 238-245.

Vrachnos, E., & Jimoyiannis, A. (2014). Design and evaluation of a web-based dynamic algorithm visualization environment for novices. *Procedia Computer Science*, 27, 229-239.

Wirth, N. (1985). *Algorithms and Data Structures*. USA: Prentice Hall.

To cite this article: Ladias, A., Mikropoulos, A., Ladias, D., & Bellou, I. (2021). CodeOrama: A two-dimensional visualization tool for Scratch code to assist young learners' understanding of computer programming. *Themes in eLearning*, 14, 31-41.

URL: http://earthlab.uoi.gr/tel