How can a simulation game support the development of computational problem-solving strategies?

Nikolaos Pellas, *University of the Aegean*
The present paper below is an original authors’ working draft version that led to an article publication. A reference in APA settings to this work should always be done using the following citation:


This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.
How can a simulation game support the development of computational problem-solving strategies?

Nikolaos Pellas, Spyridon Vosinakis
Dept. of Product & Systems Design Engineering
University of the Aegean
Syros, Greece
e-mail: npellas@aegean.gr, spyrov@aegean.gr

Abstract—Game-based learning using interactive environments to impart theoretical and applied knowledge for introductory programming courses is divided in two popular approaches: “game making” and “game playing”. Various studies have been conducted following greatly the former approach in secondary and tertiary education with controversial results. However, there has been relatively little research shown about how game playing can be associated with the development of computational thinking and how fundamental programming concepts can be supported by playing games. This work investigates how a simulation game should be designed to support the development of computational problem-solving strategies through the medium of learning fundamental computer science concepts, by proposing a theoretical game playing framework.

Keywords—computing education; computational problem-solving; game-based learning; Open Sim; Scratch4SL

I. INTRODUCTION

Computer science (CS) is a key field for advancement and innovation in different STEM (Science, Technology, Engineering, and Mathematics) disciplines. One of the most significant scope in modern computing is the way of fostering students’ logical thinking and problem-solving skills in combination with coding, by analyzing a strategy that can be applied as a proposed solution for real-world problems [1-3]. There is a broad agreement that computational thinking (CT), and its core, computational problem-solving, pave a pathway of recognizing the prerequisites for thinking how to solve problems and interest in finding the most efficient way to apply solutions, like a computer scientist [4,5]. Additionally, literature reviews [6,7] have been undertaken to define what skills are necessary to support CT and how interactive environments can become effective tools for students who try to transfer their thinking solutions into workable plans and algorithms with precise instructions in an abstract manner. Worldwide curricula have widely used game-based learning (GBL) as a pedagogical approach for developing CT, following “game making” (exercise in learning programing by designing a game) or “game playing” (exercise in learning programing by playing a game). These are implemented either with visual programming environments, like Alice [8], Scratch [9] or with three-dimensional (3D) virtual worlds (VWs), like Second Life (SL) [10] and Neverwinter Nights 2 [11]. The former approach is referred as the most “mainstream” in learning programming. It is based on algorithmic design and the artistic expression through interactive narrative-based or interactive games that students create [6,9].

However, the existing research has well-documented a wide range of difficulties and deficits for students aged 13-16 years old (high school), who mostly tend to focus on surface knowledge acquisition. Some of the most important reasons are the following [12-14]: (a) lack of skills in problem-solving and different levels of abstract reasoning cause usually failure in expressing problem solutions as programs and (b) lack of logical reasoning and algorithmic thinking skills frame a purely technically-centered process for translating mental representations of problems and solutions into source code. These skills are applied in a natural way of formalizing tacit knowledge during game playing, but students cannot inherently conceptualize their cognitive thinking process in playable modes to concretize logically abstract concepts. Previous studies [9,11] have found out that students by creating and programming simple games cannot articulate CT skills [3] and they fail to internalize computational problem-solving strategies converting a more inferential realistic interpretation in game playing. For example, visual programming environments lower the barrier of writing syntactically correct a computer language, due to the general use of a palette with visualized control-flow and command blocks. Taking into consideration only this feature, students, firstly, do not necessarily take advantage of intuitive, natural modality for user-interaction in activities that required CT skills, and secondly, they are not encouraged for a more general understanding of computational concepts [11,15,16].

Up to date, a game playing approach using a simulation game (SG) is another notable option for introductory CS courses [4,16]. In general, a SG covers a wide range of activities in real-life conditions to apply abstract knowledge that reflected on various learning purposes, such as training, analysis or prediction. In CS courses, a SG can present embodied problem situations fostering students’ problem-solving ability, and thus experience on acquiesce within a scientific discovery process [4]. Particularly interesting is the creation of SGs in 3D VWs, like SL [17] or Open Sim [18]. The representational fidelity of such a “world” can offer a persistent game-like environment with multimedia and a/synchronous communication tools for the creation of engaging and interactive activities to understand better CS concepts. Recent works have focused on VWs’ own scripting language learning [18] or problem-solving
constructions avoiding the syntax complexity with Scratch4SL (S4SL) [10].

Despite the previous efforts [7-9], little is known regarding how game playing in a SG can be related with the development of computational problem-solving strategies for high-school CS courses. This research seeks to propose: (a) an ongoing game playing framework and its benefits in acquiring CT skills for high school introductory CS courses; (b) a case on how a SG supports students to think “computationally” in order to express and apply a logical way of thinking to a solution using programming constructs for a simulated real-world problem; and (c) a rationale on how in-game elements/characteristics created in a 3D VW, like Open Sim should be mapped in the direction of helping students use their problem-solving, logical and abstract skills for the analysis of sub-parts of a computational problem. Students can also use S4SL to implement the proposed solutions using programming constructs that can be applied.

II. BACKGROUND

A. Computational problem-solving and interactive environments

To become well-educated in an increasingly computing-driven world and be prepared for the demands of the 21st-century, students need to have a deeper understanding on the role that the fundamentals of CS concepts can play in real-world problems [3,7]. Wing [19] defines CT as a problem-solving process in relation with conceptualizing, developing abstractions and designing systems, which requires the use of logical thinking with concepts fundamental to computing. Computational problem-solving strategy is a cognitive thinking process for the analysis of steps focusing not only on a solution to a problem using logical and abstract thinking, but also to the implementation of this process with basic programming concepts that are applied with computing tools [1,4]. Consequently, learning to think “computationally” includes the pervasiveness of CS programming constructs and problem-solving strategies. Still today, prior works [2,3,7,14] have pointed out a subtle distinction between computational thinking and programming. In principle approval, CT does not require programming at all, although in practice, representing a solution based on the analysis of strategy for programming to a problem is generally applied as a perfect way to evaluate the thinking process of a proposed solution [4,13].

Understanding how the development and implementation of a computational problem-solving strategy can allow students to solve problems by playing, has recently gained the research interests at a large extent [2-5]. Previous literature reviews of this field [6,7] have been undertaken to define what skills involve CT and how interactive environments can become effective tools for students who must be enable to transfer their thinking solutions into workable plans and algorithms in an abstract manner. The literature in GBL for introductory CS courses focuses on “game making” or “game playing” approaches to facilitate the development of CT skills. The interactive environments that have been extensively utilized for these purposes are separated in two categories: (1) visual programming and (2) 3D VWs. In the first category, students can program one or more “sprites” (i.e. iconic characters) on a “stage” (scene background) using a palette of programming blocks and the result is usually animation, game, or interactive art. The graphical code building blocks has shown considerable promise in programming languages syntax, aimed at giving students a first introduction to computation. The most remarkable features are the applicability and visualization of algorithmic control flow (code tracing) by “dragging-and-dropping” graphical code blocks. These blocks are resembled like jigsaw puzzle pieces through logically and specific instructions (control flow blocks nesting), with a view to avoid syntax errors and finally propose solutions to a problem expressed as design patterns (executive solutions built from combinations of blocks consisted of simple programming constructs) [6,8].

In game playing approaches, students play computer games that promote algorithmic thinking and basic programming skills, mainly, by using Code.org website. Code art and semi-structured story-based or interactive games remain a good starting point to enhance the digital fluency reflected with creative computing experience. Yet, another worthy effort is to create contexts which can allow personal expressions with digital technology for authentic problems, corresponding to the analogies of solving simulated real-life problems [7,8,16]. SGs can present embodied problem situations fostering students' problem-solving ability and thus acquiesce to experience within a scientific discovery process, inducing the expression [7,17,19,20]: (a) on how the pertinent behaviors are considered from logical and abstract thinking (abstraction) in favor of formulating and testing solutions to a problem by specifying computational rules and concepts. This vision poses an understanding of basic computational concepts, in addition with the ability to use and communicate these concepts effectively, and (b) on how obtain solutions to a problem by performing a sequence of steps that are applied as design patterns through acquiring programming skills. It becomes then clear to what extent are these patterns implying the need for simulation to interpret the abstractions (automation). Both are reflected as undisputed keystones for developing computational problem-solving strategies.

On the other hand, 3D VWs offer a sense of authentic problem situations through interactive activities, in which students can provide solutions to simulated real-life problems, track errors optically and understand better the consequences of their actions at the same time [17,18]. Instructional approaches based on simulations can enhance students’ abilities to apply abstract knowledge by situating learning in authentic, virtual contexts similar to the environments in which learners' skills should be used. The 3D visually-rich graphical interface permits users who are spatially distributed (or not) to interact using a-synchronous communication tools with others as embodied representations (called “avatars”). A 3D multi-user VW provides interactive simulations in a plausible illusion and it permits users [17,21]: (i) to construct simulated real-life problem-based contexts, using built in tools to create and modify objects and practice competencies; (ii) to refine rules of
the spatial proximity with high representational fidelity via a multi-user environment that provides prompt feedback; and (iii) to understand conceptual theorems or metaphorical representations (metaphors) of their ideas without spatio-temporal physical constraints through embodied actions, including view control, navigation, and object manipulation by interacting with other peers in a common virtual place. Existing research on 3D VWs straddle several academic disciplines, ranging to all STEM disciplines, including CS as well. Similarly noteworthy are the results from previous studies, which have already found positive attributes in learning, as students were able to: (i) understand better introductory programming constructs by avoiding syntax errors of Linden Scripting language (LSL) via S4SL for constructing 3D artifacts with behavior in SL [10], (ii) allow the communication and interaction with their peers or with the instructor to develop CS basics concepts using LSL [18], and (iii) improve students’ comprehension in basic programming concepts with regard to replace traditional languages, create and integrate behavior by writing simple scripts via LSL [22].

B. Frameworks for analyzing computational understanding

Since not only gaining knowledge in using programming concepts is the key point of CS courses, an understanding about the thinking process that followed as a problem-solving strategy is a crucial point for measuring what students have learned. A conventional way to measure strategy based on CT is to analyze student-made computer games that includes three aspects to game programming: (a) visual palette used for combining fundamental CS constructs in building blocks, (b) design patterns included multiple programming constructs to create instances of patterns, and (c) game characteristics, which are a combination of programming constructs and design patterns, giving the player’s actions to overcome in-game challenges. For this reason, a large body of literature has proposed various frameworks for identifying the types of thinking in which students are engaged while programming. Indicative examples are as follows: (a) with respect to Scratch, Brennan and Resnick [9] have proposed a framework with the three dimensions of computational thinking: computational concepts, computational practices and computational perspectives; and “Foundations for Advancing Computational Thinking (FACT)” aimed to prepare and motivate middle school student engagement with algorithmic problem-solving [5], (b) regarding Agentsheets, the “Design principles for the simulation creation toolkit” for creating and programming computer science simulations [15], (c) in relation to Stagecast Creator, a coding framework for describing strategic analyses of students’ games creations [23], (d) in regards to Alice, the game “Computational Sophistication Model” for measuring students’ computational learning by game making [8], and (e) “Scalable Game Design” to correlate CT skills relevant to game design and simulations [24].

Although visual programming environments are useful for creating enthusiasm and motivation in programming, it is arguable whether students with little or no programming background can develop skills in CT through this approach [6-9]. Two are the most important viewpoints. The first addresses to the emphasis of analysis and process design that stemmed from procedure-oriented programming. A large number of studies [4,7,9,15,16] underlined some drawbacks. In many cases, students try to comprehend source code that implies in a “programming as activity” perspective, rather than a set of combined problem solving, logical and abstract thinking skills, which can be associated with programming constructs in order to be solved computational problems. By “decoding the code” through exhibits relevant output functionality and readability of code commands sequentially or syntactically correct, students are focused explicitly on the declarative aspects of programmers’ knowledge by designing games, in which they seek to solve only problems at a superficial level [9,11]. The second viewpoint is the visualization thinking and the analysis of thinking process that are not emphasized on the development attribute on expressing computational problem-solving strategies, but on describing strong operability in computational practices through code syntax. The main result is the difficulty in decomposing and formulating a problem. The transformation of plans with syntactically correct instructions for execution and assess the consequent results of those instructions inferring to [6,9,11,23]: (a) the presentation of digital artifacts or applications, including stories or digital games that seemed too simple or without purpose and (b) the interpretation of applications, where proper writing code fragments are executed correctly, but the design patterns cannot support any proposed solutions to a problem. Thence, code documentation is neither exactly what student would want to present, nor assist their trials to comprehend source code, causing often significant conceptual gaps [9,23]. Students may not be helped to develop/enhance their problem-solving skills substantially, albeit the implementation of design patterns using a palette with programming constructs seemed to empower their programming skills.

On the other side, the way in which students try to write syntactically correct the scripting language code of the most known 3D VWs, like SL or Open Sim, given its difficulty and similarity with other general-purpose, like C, is still not well-documented. Alternatively, Girvan et al. [10] by using S4SL and following Constructionism as the theoretical underpinning, have placed an emphasis on the process of constructing shareable artefacts in SL. S4SL provides a graphical palette with control flow programming constructs and instruction commands movements, similar with Scratch’s palette features. It is also used as a free-plug module in SL or Open Sim, giving users an easy way to integrate behaviors and interactions to objects without financial cost.

Bearing in mind all the aforementioned and conceiving that computational problem-solving strategies require the cultivation and connection of problem-solving with development of programming skills for presenting design patterns as solutions to a problem, it is crucial to compensating the design requirements and guidelines proposed by relevant studies [5-9], in which their learning scenarios can be implemented, either in visual programing environments or in 3D VWs. Hence, the hypothesis that arises, is whether the combination of the positive effects on learning performance
from design features and characteristics of each category could fill the “gaps” detected in each, and thus facilitate the creation of an innovative environment supporting the development of computational problem-solving strategies. Particularly interesting to meet the required design guidelines needs, which are noticed above, is the combination of S4SL palette with a 3D VW, such as Open Sim. Open Sim is an open source (free-of-charge) server platform that contains a 3D networked, interactive and multi-user virtual environment with a persistent workflow accessible simultaneously by many users in the same time and place.

Due to the surge of game making approaches, an alternative and certainly less explored for the development of computational problem-solving is through digital game playing. Until today, there is little consensus referring the essential characteristics regarding game playing frameworks for SGs, which can support the development of students’ computational problem-solving strategies. Therefore, there is an urgent need to have a better understanding of the impact of SGs in introductory programming courses and the development of CT strategies through game-play.

To address the above issues, Garris et al. [25] game-based learning model explicitly illustrating integral SG features and perceptions for active learning processes. This game-based learning model emphasizes both motivation and process aspects associated with skill-based learning outcomes. According to this model, a game for learning should continuously present a motivating goal at an appropriate level of challenge where learners can have full control of their learning activity. Such a game can promote in-depth learning while interacting with it. Garris et al. [25] categorized game characteristics as “fantasy, rules/goals, sensory stimuli, challenge, mystery and control”. The same authors have also proposed design principles for the conceptualization of design guidelines in SGs. In this line, based on the game cycle of the input – process – output game model, the following principles are extracted: user motivation and persistent engagement (P1), clear and challenging goals (P2), system’s feedback on user’s actions (P3), scaffolding process (P4) and debriefing based on skill-based learning outcomes (P5), both with virtual instructional support. The development of a SG for learning CT using Garris et al. [25] model can be considered as the most appropriate for the following reasons: (a) students can intrinsically identify objections of the main problem, handle its’ sub-parts and propose a solution; (b) students recognize a way of understanding how they to think before start coding based on their judgements and behaviors; and (c) the development of cognitive learning outcomes comes from strategic knowledge that requires learned principles using different skills. The above reasons imply the development of understanding when and why principles should be applied through in-game activities through discovery learning. Referring that a game playing approach can add several prominent learning and instruction principles to computational problem-solving strategies, this work investigates how these principles may be applied to design game-like learning systems and facilitate flow learning experience in GBL.

The focus of this work is to outline a game playing framework for supporting the development of computational problem-solving strategies through a SG. The proposed game playing framework can facilitate students to practice and develop computational problem-solving strategies for high school students, regardless their programming knowledge background. Also, this work intends to elucidate and concretize precise design guidelines and characteristics of a SG created via Open Sim and S4SL with the intention of imposing how to apply programming constructs in simulated real-life problem-based contexts. Open Sim and S4SL can be considered a powerful set of tools serving as bridge the “gap” between problem formulation and solution expression.

III. TOWARDS A GAME PLAYING FRAMEWORK

A. Rationale and design decisions

As SGs are increasingly being applied for introductory CS courses [4,16], how can designers integrate game playing guidelines and principles is becoming more imperative. In sum, it can be beneficial to develop an instructional game-based framework for CT planning development, with the purpose of discussing: (a) what are the game characteristics that should be used for student engagement, (b) in which instructional circumstances students need to have the assistance in pursuance of recognizing, if they really tried to cultivate CT skills, and (c) how to assess if students adopt CT skills into simulated real-life problem-based context, by mapping through explicit abstractions their naturally-expressed thinking solutions built to workable plans and algorithms with precise instructions. Evidence from other works [1,11,24] also suggest that once students understand conceptually how to present a pattern, they are able to transfer and use it in the context they choose as well, highlighting that problem-solving in game-based context is regarded as an activity that can be approached meaningfully with a playful attitude. Understanding of game events and having the ability to describe events in natural language can be a good starting point that would allow them to engage with basic computational concepts. Specifically, where the goal is to help students develop their understanding on computation and/or use computation in simulated realistic settings, a question that arises is whether natural language might be a preferred notation in understanding how students use CS concepts to solve a problem through programming, given its familiarity and ubiquity that exists in real life [11,16]. To this notion, an interactive environment must address not only to syntactic, but also semantic and pragmatic concerns regarding the consequences of programming in solving a problem. Based on the relevant literature [7,12,20,23], three are the presuppositions that need to be supported:

(1) Decomposition and formulation of the main problem (abstraction): Decomposing and formulating the main problem is associated with “abstract conceptualization”. It is the first step that allow students to conceptualize, either verbally, e.g., by trying to formulate a question such as “How can I solve this problem, by using and making in-game elements work for this effort?” or visually, e.g., by describing students’ thoughts in diagrammatic representations or in natural language texts in
place of identifying in-game objects’ behaviors and relationships. At the beginning of a path from problem formulation to solution expression, two are the most prudent characteristics that need to be referred. The first is “visual thinking” for the problem formulation, so that students can organize their thoughts, describe object interactions and improve their ability not only to think, but also to communicate them [23]. The second is the use of “spatial metaphors” that can support a conceptualization approach for improving critical thinking and problem-solving skills in visual thinking, giving to all users the opportunity to organize information visually [20];

2) Description and expression of a solution (automation): The right analogies in expressing the correct computational rules and concepts. It is required an understanding of logical reasoning in order to use and communicate these concepts effectively (e.g. how students can propose a solution based on their natural perceptions and if its rules/concepts can be transferred to programming constructs for execution). Many studies [7,11,24] have addressed the syntactic challenges of end-user programming shifting from syntactic to semantic and pragmatic concerns. This process supports a focus on essence for visualizing behaviors with building blocks consist of motion commands (command blocks) and programming constructs (control flow blocks) that can be integrated to objects by composing programming constructs as design patterns to propose executive solutions for a problem;

3) Execution and evaluation of design patterns (strategy analysis): To known how to program effectively for solving a problem, students need both to explain and understand the syntax and semantics of programming concepts, and combine these features into valid computer programs. The evaluation of students’ solutions and thinking process as a cognitive problem-solving strategy to transfer a solution that defined as a natural perception into workable instructional plans for solving problems can be achieved through programming. To evaluate the development of effective design strategies, a scaffolding teaching approach is suggested to support students in mastering this nesting composition method of programming constructs as design patterns [20,24]. For instance, integrating explicit educational instructions and extra feedback by the instructor on the composition of programming plans with visualized program tracking in game play mechanisms, users can deeply understand how nested control-flow blocks work and what are the subsequent effects of the chosen actions.

Pertaining on the above, a SG should support students to (a) create correct and complete computational rule specifications for problem-solving, using firstly natural expressions; (b) develop an understanding of computational concepts, and the appropriate programming constructs to practice a solution for the given facts of a problem and (c) transform programming problems into workable plans and algorithms using programming constructs. Therefore, this contribution aims to (i) propose a game playing framework for supporting students to learn how to solve computational problems in simulated realistic contexts by interacting with the provided visual programming elements, and (ii) understand the entire process on how students interact with a proposed environment, so as to examine their performance in computational problem solving based on different design patterns that they propose. To address these issues, the present work tries to extend game model cycle of Garris et al. [25] and proposes an on-going game playing framework with specific design guidelines and its benefits in acquiring CT skills to support learning introductory CS courses. The following proposed guidelines are along with a reference to Garris et al. [25] principles from which they arose (indicated as P1 “for user motivation and persistent engagement”, etc. which are referred in the Background):

1st guideline (G1): Motivating students to participate in active learning tasks- While a computer game is intrinsically motivating, there should be developed sub-parts of the main problem with clear and challenging tasks. Users can achieve higher level learning by solving relevant sub-goals, in order to start analyze, create, apply and evaluate each proposed solution. This process will allow them more properly to think logically and critically for the analysis and expression of solutions to a problem (P1);

2nd guideline (G2): Simulating an authentic problem- The simulation of an authentic problem should be available for exploration at the beginning. Data visualization and representation should avail more clearly the operation of learning activities that students can be involved. If students are more interested and involved knowing what they precisely have to do, they devote more time on actively pursuing through challenging activities (P2);

3rd guideline (G3): System’s feedback on user’s actions- A SG should not only simulate a real-world problem that may be encountered in everyday life of students, but it should also provide prompt feedback during the run-time of students’ actions, either optically or acoustically to better conceptualize concrete game playing examples to abstract strategy (P3);

4th guideline (G4): Facilitating the development of computational problem-solving strategies through a scaffolding process- The game should allow students to develop their problem-solving strategies for scaffolding the program construction, i.e. before finalize the solution, program should be broken down into sub-programs which would make the given task manageable by collecting and using the appropriate data to propose a solution for each sub-part of the main problem. Take for example, through an interactive game playing, where students need to know how to insert behaviors in objects with the purpose of making both interact to each other. On the other hand, the instructor should demonstrate how such a sub-program could be constructed. Even if frustrating tasks are observed, the instructor should guide the students by prompting them with questions on their problem-solving process (e.g., which is the main reason of putting that command there?) (P4);

5th guideline (G5): Applying design patterns to propose an answer for the main problem’s question- Students’ embodied experiences/ideas should be simulated through actions that are performed on the sub-parts of the main problem. By helping students to understand how to transfer of behaviors in different objects, it is considered as a crucial process in recognizing how
these actions will (or not) solve a problem. In this perspective, students can propose design patterns to implement their ideas by coding and using programming constructs (e.g., repetition or selection) in control flow instruction commands. This approach can foster computational practices and perspectives, as students need to review and think about the programming process by modifying building blocks of programming constructs, and present program execution consequences for omissions or commissions of the code segments (P5).

Figure 1 illustrates the proposed framework and game guidelines, which can be designed to be an integral part of game characteristics to support the design guidelines (G) that have been previously described. These are: (a) Fantasy: A SG should offer analogies or metaphors for real-world processes that permit users to have experience with phenomena or tasks which sometimes cannot be done in real-world settings. For example, students as embedded systems engineers can integrate behavior in objects responding to events, or issue commands to actuate controls; (b) Rules/Goals: The rules describe the goal structure of the game. The game designer should predetermine also the game mechanics that would help users who lag while play; it could include bonus or subsidies or for their poor performance to have a score board with penalties. If students can understand and specify computational rules, they may use and express relevant basic computational concepts correctly to propose their solutions; (c) Challenge: Games should employ progressive difficulty levels, multiple goals, and appropriate information to ensure certain learning outcomes. Performance feedback and score keeping allow players to track progress toward desired goals. A challenging task is created by things like time pressure and opponent play. Fellowship should be encouraged by sharing information among team members or announcing the winning conditions with awards that are more difficult to achieve each one. In a fading scaffolding process, the instructor’s role is explicitly to “scaffold” and assist students to use their virtual characters and then start to play following sequences of game rules (algorithmic thinking patterns); (d) Mystery: Simulations that incorporate these features become more game-like. For example, in a game-based environment, users are trained in a simulation that incorporates features, such as role-playing and scoring that are not present in the real-world task. Users can be curious about their performance based on their solutions that expressed as design patterns; (e) Control: A sense of freedom inside a gaming task allows users to select/refine strategies, manage the direction of activity, and make decisions that directly affect outcomes, even if actions are not instructionally relevant. This sense, beyond the implementation of each learning task, it gives to each user the ability to explore, recognize the problem space, and propose alternative solutions; and (f) Sensory stimuli in a SG includes sound effects, dynamic graphics, and other media sources. Such an environment should not disrupt at a large extent the stability of normal sensations and perceptions. It can also allow the user to have a more reliable experience, like programming a train and the railway model that is needed for the integration of behaviors in the environment to understand simulated phenomena, such as gravity, in contexts that mimic those measured when driving as in reality.

As it is shown in Figure 1, students need initially to be engaged in game tasks, which will in turn generate their desire for improvement through attractive learning scenarios, such as role-playing (Stage 1). These tasks will help students to develop computational problem-solving strategies regarding the targeted educational content and through several tasks, they will be able to produce a set of learning outcomes (Stage 2). In the end, the learning objectives will be achieved and evaluated after the game experience (Stage 3). It is of great importance to mention that players continuously need to increase their problem visualization and solution-development as they progress through a game. In this demand, segmented four stages (S1-S4) can be aligned with the proposed game design guidelines (G1-G5). The first stage (S1) is aligned with the 1st and 2nd guidelines, the second stage (S2) is collimated with the 4th guideline, the third stage (S3) relates to the 5th guideline, and the fourth stage (S4) is allied to the 3rd guideline for the system’s feedback cycle.

This proposed game playing framework is developed specifically for designing games to identify how CS concepts can be associated with specific learning tasks to support the development of computational problem-solving.

B. An instructional design approach

The next step that will lead to the proper construction of this framework is the identification of the instructional methodology that a game will be based on. According to the proposed framework, a SG prototype may have the following characteristics: (a) the simulation of authentic problem situation and functionalities in fading scaffolding processes for supporting users’ roles (students and instructor), (b) the visual metaphors of Open Sim related to innate CT skills and conceptualize them into algorithmic rules through abstract thinking logic, and (c) the S4SL palette use to eliminate split attention in code syntax, focusing on goals of users’ solutions that applied as results of computational problem-solving.
strategies. By using a game simulation, players can carefully consider how the actions will (or not) solve a problem. Thus, students can have the opportunity to program the transfer behaviors of an object according to their experience in a simulated real-world context. Similarly, novices are allowed to learn the process of solving a computational problem, giving instructions on how a robot vacuum cleaner should clean a room without hitting on tables, humans or animals. While single-user applications in SGs (one single user and the simulation environment) are provided in conventional instructional approaches, 3D VWs, like Open Sim can provide opportunities for designing a multi-user SG to interact with other peers in situations with new identities (i.e. as avatars) in the form of role-playing. By way of illustration, the ability to change the appearance of one’s avatar to fit the role that can play, for example as an embedded system engineer in a problem space, can become a powerful contributor to the veracity of simulation. This ability can affect positively the user’s willingness to participate, for example as an engineer who want to specialize knowledge in exploring the interactions through a process of controlling how various devices and machines can be worked correctly and simultaneously in a modern simulated house.

The rationale for utilizing S4SL and Open Sim can provide multiple opportunities for students to learn programming commands. These are: (a) the emphasis on the design of algorithmic problem-solving activities by avoiding syntax errors from LSL; (b) the visually-rich set of tools to create, edit and syntax multiple artifacts via S4SL to coordinate the learning procedure using a-synchronous communication tools in Open Sim; (c) the direct feedback based on their actions in a common environment by copying and pasting from Scratch’s palette blocks as design patterns to an object’s notecard for integrating behaviors; and (d) the S4SL palette is frequently used in high schools and this feature can help students’ motivation and participation in programming.

Three are the goals that considered as important for helping students articulate and transfer their thinking solutions from natural language to workable plans and algorithms with precise instructions in coding using Open Sim and S4SL. These goals are the following:

(a) Integration of the learning material within the game interface: Providing a natural way of formalizing knowledge to an abstract manner in a simulated real-life problem-based context during game play has a critical place. Decomposing innate thinking into abstract representations using visual metaphors of Open Sim employ an approach that should infer and predetermine the designer specific strategic rules corresponding to multiple movements that are the most appropriate to be done by students. Using this method, students can choose between two or more appropriate options or directly specify another one. In this occasion, although users can previously solve low granularity problems, such as creating a robot, they have also the chance to create a simulation-based environment, integrate to it several objects with behaviors that interact with other objects or avatars in the same environment and find solutions to an authentic problem;

(b) Transfer from tacit thinking to concrete thoughts of computational concepts: Providing activities to a natural way of formalizing students’ ideas during game play is crucial place to assist them explicitly link their game actions to CT. The instructor needs to explain the primary aim of the courses and to suggest students on how they can propose the right rules or directly specify a new rule (e.g. pseudo-code or full sentences in natural language) before programming. Students should try to explain their solutions that require the articulation of CT skills (e.g. abstract logic, expression of rules in naturally game-play settings that should be applied through coding). The technological infrastructure of Open Sim also makes it possible for an authentic problem, not only to be simulated visually, but also simplified by removing factors, which would need to be addressed in real-life. A specific example is when providing ready-made scripts or objects with integrated behaviors for movement, in which users start to work focusing directly on specific sub-parts of a problem; and

(c) Transform students’ thinking knowledge through in-game play settings into formal logic and analysis about a solution in coding: Student’s progress in scaffolding activities requires the process of concreteness in transferring from natural expression to coding via S4SL. Therefore, S4SL palette can be used to minimize the split attention effect by avoiding any misconception in code syntax. More specifically, students can program the objects’ behavior via S4SL, by writing building blocks to copy-paste from the palette into LSL scripts, and receive an immediate visual feedback of how their programs running inside Open Sim. The instructor can coordinate the in-game learning process, give feedback on users’ actions or frustration tolerance for ambiguity movements and evaluate their progress when they present and observe program execution consequences at runtime.

Table 1 associates a process about how students can develop their skills in game play with the previously defined CT skills from the aforementioned analysis, so that supporting computational problem-solving development through in-game settings. This table also validates how cognitive thinking skills (e.g. logical or abstract thinking etc.) can be developed in game playing modes and the foundation of CT skills outlined based on the literature in dwelling on problem solving, understanding problems, and formulating solutions [3]. Using this type of game playing in a SG, students can visualize their efforts according to their dispositions and attitudes that are essential dimensions of CT, as aligned with the proposed game design principles (G1-G5) to understand how they can apply them effectively to every programming construct.

<table>
<thead>
<tr>
<th>Learning tasks associated with CT concepts and CT skill definitions(a)</th>
<th>Proposed game activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describe an algorithmic thinking of logical steps for the proposed solution to the main problem.</td>
<td>Describe a possible solution as an algorithm to compute sub-goals of the main problem using natural language (G1).</td>
</tr>
<tr>
<td>Logical thinking skills: Logical steps required for formulating a solution to the given problem.</td>
<td>Analyze the path that is better to be followed. Try to understand how you can move your object in Open Sim (G2).</td>
</tr>
<tr>
<td>Problem description and decomposition into a collection of intermediate sub-goals.</td>
<td>Try to express your problem using natural language (G3).</td>
</tr>
<tr>
<td>Problem-solving skills: Process of breaking down a large problem into smaller sub-parts.</td>
<td>Try to understand how you can move your object in Open Sim (G2).</td>
</tr>
<tr>
<td>Analysis and data representation as steps in coding algorithm.</td>
<td>Design and implement solution to the sub-problems of the main problem (G4).</td>
</tr>
<tr>
<td>1. Design and implement solution to the sub-problems of the main problem.</td>
<td>Abstract thinking skills: Explain how and which programming constructs can be utilized.</td>
</tr>
<tr>
<td>Abstract thinking skills: Explanation of how and which programming constructs can be utilized.</td>
<td>Debug the algorithm or an object, quickly try to transmit your solution in coding for the object’s movements and observe the results during the run-time. If the naturally-expressed solution that is even implemented correctly based on the proposed instructions and rules? Is it easy to follow a solution algorithm and observe the expected behavior? (G4).</td>
</tr>
</tbody>
</table>
A case study using the proposed framework and design guidelines in a SG can be implemented, following blended instructional format by spending some hours in school’s computer laboratory and some others in supplementary online courses. CS concepts that want to present the instructor, should be made in the conventional computer laboratory and the implementation of any concept that learned in the SG created in Open Sim. Initially, even before the beginning of this study, the instructor need to establish and ensure students’ access in Open Sim and S4SL, either if courses will be done in a computer laboratory or online, with the purpose of resolving any technical issues and allow them to participate seamlessly, like doing a homework. The instructor has also the responsibility: (a) to attend in all courses (face-to-face and/or supplementary online) and assist students’ efforts, (b) to give the appropriate feedback for compilation or execution of nay detected errors and (c) to access on users’ actions, either online via Open Sim or offline in rubrics, which can provide a general understanding on how students start thinking about solving sub-goals of the problem before coding.

**IV. CONCLUSION AND FUTURE WORK**

This work outlines a theoretical game playing framework for the creation of SGs that can support the development of computational problem-solving strategies. Main findings of this work may be of interest to instructional designers who want to take in advance a learning environment for high-school CS courses. The results may be predominantly helpful for instructors or educators who design (in-) formal instruction using a SG to foster students’ computational problem-solving strategies. Based on the proposed design guidelines, a SG can present the core function of CT, in which the key operations of abstracting and modeling can foster students’ thinking about a problem. The results of abstracting and modeling understandings are executed in order to be applied effectively and efficiently the design patterns for simulated real-life problems, as a process that allows the evaluation and debugging of thinking. According to the proposed framework using Open Sim and S4SL, a SG prototype for a real-life problem, with scaffolding leaning tasks reflecting on students’ solutions through coding is also being developed.

**REFERENCES**


