Exploring the development of high school students' computational problem-solving strategies by utilizing three-dimensional (3D) virtual worlds

A dissertation

submitted to the Department of Product and Systems Design Engineering,

School of Engineering of

University of the Aegean

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Nikolaos Pellas

2019

Declaration of authorship

I am the exclusive author of the doctoral thesis titled *"Exploring the development of high school students' computational problem-solving strategies by using three-dimensional (3D) virtual worlds"*. This Ph.D. thesis is original and was written exclusively in order to be submitted at the Department of Product and Systems Design Engineering, School of Engineering of the University of the Aegean in partial fulfillment of the requirements for the degree of Doctor of Philosophy. Every help that I had to prepare it is accurately recognized and stated. I have exactly mentioned all sources, data or ideas based on others' works or ideas, even if their inclusion in the present thesis were indirect or paraphrased. Generally, during the writing process of my Ph.D. thesis, I have strictly abided by the law and what it is defined as intellectual property and I have fully complied with the provisions of the law on personal data protection and the principles of academic ethics.

Nikolaos Pellas

Υπεύθυνη δήλωση

«Είμαι ο αποκλειστικός συγγραφέας της υποβληθείσας Διδακτορικής Διατριβής με τίτλο «Exploring the development of high school students' computational problem-solving strategies by utilizing three-dimensional (3D) virtual worlds». Η συγκεκριμένη Διδακτορική Διατριβή είναι πρωτότυπη και εκπονήθηκε αποκλειστικά για την απόκτηση του Διδακτορικού διπλώματος του Τμήματος Μηχανικών Σχεδίασης Προϊόντων και Συστημάτων. Κάθε βοήθεια, την οποία είχα για την προετοιμασία της, αναγνωρίζεται πλήρως και αναφέρεται επακριβώς στην εργασία. Επίσης, επακριβώς αναφέρω στην εργασία τις πηγές, τις οποίες χρησιμοποίησα, και μνημονεύω επώνυμα τα δεδομένα ή τις ιδέες που αποτελούν προϊόν πνευματικής ιδιοκτησίας άλλων, ακόμη και εάν η συμπερίληψή τους στην παρούσα εργασία υπήρξε έμμεση ή παραφρασμένη. Γενικότερα, βεβαιώνω ότι κατά την εκπόνηση της Διδακτορικής Διατριβής έχω τηρήσει απαρέγκλιτα όσα ο νόμος ορίζει περί διανοητικής ιδιοκτησίας και έχω συμμορφωθεί πλήρως με τα προβλεπόμενα στο νόμο περί προστασίας προσωπικών δεδομένων και τις αρχές Ακαδημαϊκής Δεοντολογίας».

Νικόλαος Πέλλας

Dedicated to the memory of my beloved father

SUPERVISING COMMITTEE

1. Dr. Spyridon Vosinakis, Associate Professor, Department of Product and Systems Design Engineering, University of the Aegean (Main supervisor)

2. Dr. Panayiotis Koutsabasis, Assistant Professor, Department of Product and Systems Design Engineering, University of the Aegean

3. Dr. Konstantinos Tsolakidis, Professor (Emeritus), Department of Primary Education, University of the Aegean

EXAMINATION COMMITTEE

1) Dr. Spyridon Vosinakis, Associate Professor, Department of Product and Systems Design Engineering, University of the Aegean

2) Dr. Panayiotis Koutsabasis, Assistant Professor, Department of Product and Systems Design Engineering, University of the Aegean

3) Dr. Konstantinos Tsolakidis, Professor (Emeritus), Department of Primary Education, University of the Aegean

4) Dr. Thomas Spyrou, Assistant Professor, Department of Product and Systems Design Engineering, University of the Aegean

5) Dr. Stavros Dimitriadis, Associate Professor, Department of Informatics, Aristotle University of Thessaloniki

6) Dr. Georgios Fesakis, Associate Professor, Department of Preschool Education Sciences and Educational Design, University of the Aegean

7) Dr. Georgios Palaigeorgiou, Assistant Professor, Department of Primary Education, University of Western Macedonia

ABSTRACT

Over the last few years, the term of computational thinking (CT) has been increasingly presented in many K-12 curricula around the world and specifically in computer programming courses. Programming tasks can profoundly support the CT instruction and demonstration of computational competencies encompassing a wide range of cognitive thinking skills, such as problem-solving, critical thinking, logical reasoning, and creativity. To this notion, students can learn on how to use such skills to develop their thinking strategies so as to solve logically and methodically problems using CT and its computational core concepts related to abstraction, algorithm, automation, decomposition, debugging and generalization.

Game-based learning (GBL) has the potential to enable new forms of teaching and transform the learning experience through various simulated real-world problems in order to foster CT among school-age students (boys and girls). GBL approaches supported by interactive environments have long been discussed as remarkable and appropriate so that integrate CT instruction inside K-12 programming courses. Two are the most indicative platforms: (a) visual programming environments (VPEs) and (b) three-dimensional (3D) virtual worlds (VWs) combined with visual programming tools. In this perspective, students can apply their computational problem-solving strategies with approaches that include tasks associated either with exercises to design games (game making) or with exercises to play games (game playing). Gaming via VPEs and 3D VWs can greatly fulfill students' learning needs and experiences since various learning tasks correspond to an imitation of an operation of a process or a system consisted of specific simulated problem-solving situations of the real world. Thus, a worth noting GBL approach is the use of simulation games (SGs). A SG is a gaming environment that can permit users to participate actively in having specific task information to learn by doing within interactive and simulated problem-solving contexts.

However, it is still unclear how SGs created in interactive environments can affect boys and girls in order to support CT instruction, thus applying their computational problem-solving strategies in terms of proposing their solution plans. Therefore, the research hypothesis is whether the combination of the most significant design features and characteristics of a 3D VW, like the realistic simulated representational fidelity of OpenSim combined with tools of visual programming such as a palette with code blocks that offers a more user-friendly way for coding tasks or the utilization of a VPE such as Scratch for the creation of SG can assist boys and girls to gain a greater understanding of CT and complete a process from the analysis of problem-solving tasks to the formulation of solutions into code. In such a process, the measurement of students' learning performance requires the assessment not only of the formulation and manipulation of a problem's subparts using core concepts and skills related to CT, but also testing and debugging the correctness of any proposed solution with design patterns through control flow blocks from a visual palette which can be integrated by playing and programming specific visual objects inside a SG.

The aim of this thesis is to propose a computer game called *"Robot Vacuum Cleaner"* (RVC) simulator to support CT instruction and investigate its appropriateness and effects on high school students' learning performance by assessing their computational problem-solving strategies (i.e. computational design, computational practices, and computational performance). The proposed SG was created in OpenSim with S4SL and in Scratch to support the development and demonstration of boys' and girls' skills related to CT with a view of understanding how to use effectively specific programming constructs in several simulated problem-solving learning tasks. Reaching the following three objectives will create a pathway to address the main research question. The first is to propose a theoretical design framework called *"PIVB: Programming for Interactive Visual Behavior"* with specific design guidelines and criteria for the creation of a SG that can be designed in order to support CT instruction and the development of students' computational problem-solving strategies. The second is to observe and identify any usability issues by measuring the learning experience and perceptions of fifteen (n=15) high school students regarding the use of the RVC simulator created in OpenSim. The third is to investigate if the proposed SG that is created in OpenSim or in Scratch can greatly influence students to develop and support their computational problem-solving strategies. For this reason, a quasi-experimental study is conducted to compare and analyze the way that boys and girls design their computational problem-solving solution plans and practices with the purpose to measure their learning performance. A total of fifty (n=50) high school students participated voluntarily and were divided into a control group (n=25) and an experimental (n=25) group that used Scratch and OpenSim combined with visual programming, respectively.

The findings from the preliminary study revealed students' positive acceptance and use of OpenSim with S4SL to foster their computational problem-solving practices and user experience. The findings from the quasi-experimental study indicated substantial differences in students' learning performance. Mean scores on post-questionnaires from the experimental group revealed improvements higher than the control group in two aspects. First, participants from the former group created more complete computational instructions with rules to be specified and delivered the learning goals than those from the latter. Second, participants who used OpenSim with S4SL proposed and applied more correct computational concepts for problem-solving tasks and practices into code than their counterparts who used Scratch. A set of design guidelines and recommendations are also referred. First, features and elements of OpenSim supported students to map out in-game subparts of the main problem greater to explore and understand the consequences arising from their achievements made into the RVC simulator, due to the appropriate feedback given on their actions. Second, the representational fidelity of elements and features in relation to the player's awareness allowed each one to study multiple traces threads so as to consider several alternative choices that could be taken seriously into account for spotting and solving subparts of the main problem using skills related to CT such as problem-solving, algorithmic thinking, critical thinking and creativity.

ΠΕΡΙΛΗΨΗ

Η εισαγωγή στις αρχές της επιστήμης των υπολογιστών και του προγραμματισμού αποτελούν ένα αναπόσπαστο κομμάτι των προγραμμάτων σπουδών της δευτεροβάθμιας εκπαίδευσης σε παγκόσμια κλίμακα. Αντικείμενο των μαθήματων πληροφορικής και συγκεκριμένα του προγραμματισμού είναι η ανάπτυξη δεξιοτήτων γνωστικής σκέψης των μαθητών, όπως η επίλυση προβλημάτων, η λογική σκέψη και η δημιουργικότητα. Οι μαθητές καλούνται να λύσουν λογικά και μεθοδολογικά προβλήματα χρησιμοποιώντας θεμελιώδεις έννοιες του προγραμματισμού, όπως είναι η κατανόηση ενός προβλήματος και η κατάτμηση του σε μικρότερα κομμάτια, η αλγοριθμική σκέψη, ο αυτοματισμός, η απασφαλμάτωση (debugging), και η γενίκευση μιας προτεινόμενης λύσης σε μορφή κώδικα.

Την τελευταία δεκαετία, η υπολογιστική σκέψη (ΥΣ) αποτελεί έναν επιστημονικό όρο ο οποίος έχει κερδίσει την προσοχή ενός μεγάλου μέρους των ερευνητών και καθηγητών στο πεδίο των θετικών επιστημών και ιδιαίτερα του προγραμματισμού. Η ΥΣ είναι μια διαδικασία επίλυσης προβλημάτων που επιτρέπει στον άνθρωπο να σκεφτεί με ένα δομημένο τρόπο σκέψης και να ακολουθεί συγκεκριμένα βήματα βάση μιας στρατηγικής που σχεδιάζει και υλοποιεί σε μορφή κώδικα για την επίλυση προβλημάτων. Η εφαρμογή στρατηγικών επίλυσης προβλημάτων με βάση την ΥΣ σχετίζεται όχι μόνο με τη χρήση δεξιοτήτων γνωστικής σκέψης για να σχεδιαστούν και να αναπτυχθούν ως προγράμματα που θα υλοποιούν τους κανόνες των προτεινόμενων σχεδιαστικών λύσεων (υπολογιστικός σχεδιασμός), αλλά σχετίζεται και με τη ορθή χρήση θεμελιωδών δομών προγραμματισμού, όπως επιλογής, ακολουθίας ή/και επανάληψης (υπολογιστική πρακτική) για την εφαρμογή αυτών των λύσεων. Βασικός στόχευση μιας τέτοιας διαδικασίας είναι να εκτελεστούν και να παρουσιαστούν οι αποδοτικότερες και αποτελεσματικότερες σχεδιαστικές λύσεις σε μορφή κώδικα (υπολογιστική επίδοση).

Η παιχνιδοκεντρική προσέγγιση μάθησης με τη χρήση (ψηφιακών) διαδραστικών περιβαλλόντων έχει εξελιχθεί ραγδαία τα τελευταία χρόνια. Με την εμφάνιση και ευρεία αξιοποίηση των ηλεκτρονικών παιχνιδιών, έχουν καταβληθεί πολλές προσπάθειες για την ανάπτυξη διαδραστικών περιβαλλόντων στα οποία θα μπορούσε να ενσωματωθεί εκπαιδευτικό περιεχόμενο και υλικό με δραστηριότητες όπου οι συμμετέχοντες θα μαθαίνουν παίζοντας. Τα ηλεκτρονικά παιχνίδια μπορούν να ικανοποιήσουν σε ένα αρκετά μεγάλο βαθμό τις μαθησιακές ανάγκες και τις εμπειρίες αγοριών και κοριτσιών μικρότερης ηλικίας. Συγκεκριμένα, τα ηλεκτρονικά παιχνίδια μπορούν να υποστηρίξουν την ανάπτυξη γνωστικών δεξιοτήτων σκέψης των μαθητών μέσα από διαφορετικές δραστηριότητες και την ανάθεση συγκεκριμένων καθηκόντων για την επίλυση προσομοιωμένων προβλημάτων του πραγματικού κόσμου μέσα από διαδραστικές λειτουργίες/διαδικασίες που υλοποιούνται σε ένα ψηφιακό σύστημα.

Μια ιδιαίτερα αξιοσημείωτη παιχνιδοκεντρική προσέγγιση μάθησης είναι με την χρήση παιχνιδιών προσομοίωσης. Ως παιχνίδι προσομοίωσης ορίζεται ένα (ψηφιακό) περιβάλλον που επιτρέπει στους χρήστες να συμμετέχουν ενεργά έχοντας συγκεκριμένες πληροφορίες σχετικά με τις δραστηριότητες τους

μέσα σε ένα ψηφιακό περιβάλλον για να αποκτήσουν γνώσεις βάση συγκεκριμένων ενέργειων που εκτελούν σε διαδραστικές εφαρμογές όπου μπορούν να προσομοιωθούν προβλήματα της πραγματικής ζωής. Η ανάπτυξη παιχνιδιών προσομοίωσης γίνεται όλο και πιο διαδεδομένη σήμερα μέσα από την χρήση υπολογιστικών συστημάτων και κυρίως μέσα από την χρήση διαδραστικών περιβαλλόντων τα οποία περιλαμβάνουν διαφορετικά σχεδιαστικά χαρακτηριστικά και στοιχεία στο γραφικό περιβάλλον διεπαφής. Ιδιαίτερα η σχεδίαση παιχνιδιών προσομοίωσης σε μαθήματα προγραμματισμού αποτελεί μια διαδικασία οπού το περιβάλλον θα πρέπει να ωθεί τους χρήστες στην επίλυση προβλημάτων και τους επιτρέπει να έχουν ένα βαθμό ελευθερίας στην σκέψη τους όσο αφορά το πώς θα μπορούσαν να χρησιμοποιηθούν πιο αποτελεσματικά θεμελιώδεις έννοιες και δομές του προγραμματισμού για την επίλυση προβλημάτων. Με βάση αυτό το σκεπτικό, οι μαθητές θα πρέπει να έχουν την δυνατότητα να εφαρμόσουν στρατηγικές επίλυσης προβλημάτων με βάση την ΥΣ είτε μέσω ασκήσεων για τον σχεδιασμό διαδραστικών παιχνιδιών (game making), είτε μέσω ασκήσεων στις οποίες μπορούν να προγραμματίσουν αντικείμενα ενός παιχνιδιού στο οποίο τις περισσότερες φορές προσομοιώνεται ένα πραγματικό πρόβλημα (game playing). Οι πλατφόρμες που χρησιμοποιούνται σε μεγάλο βαθμό για τις ανάγκες σχεδιασμού παιχνιδιών προσομοίωσης με την χρήση υπολογιστή είναι οι εξής δυο: (α) τα περιβάλλοντα οπτικού προγραμματισμού (ΠΟΠ) και (β) οι τρισδιάστατοι (3D) εικονικοί κόσμοι (ΕΚ). Τα ΠΟΠ είναι διαδραστικά περιβάλλοντα που επιτρέπουν στους χρήστες να κατασκευάζουν παιχνίδια ή ιστορίες με αντικείμενα τα οποία χρησιμοποιήσουν και προγραμματίζουν τις ενέργειες τους χρησιμοποιώντας μια ψηφιακή παλέτα που περιλαμβάνει κομμάτια έγχρωμων μπλοκ κώδικα βασικών δομών προγραμματισμού.

Από την άλλη μεριά, οι 3D ΕΚ είναι προσομοιωμένα περιβάλλοντα στα οποία οι χρήστες μπορούν να δημιουργήσουν «ανθρωπομορφικές» ψηφιακές οντότητες (avatars). Με την αξιοποίηση των 3D ΕΚ, δίνεται η δυνατότητα αλληλεπίδρασης μεταξύ των χρηστών, η εξερεύνηση χώρων και ο προγραμματισμός γεωμετρικών αντικείμενων του «κόσμου», χρησιμοποιώντας σύγχρονες μορφές επικοινωνίας, όπως η ομιλία σε ένα συγκεκριμένο χώρο ή η συνεργατική σχεδίαση αντικειμένων, αλλά και την ασύγχρονη μορφή, όπως το μήνυμα σε άλλους χρήστες (ΙΜ). Λόγω της επέκτασης των ΕΚ σε διαφορετικά εκπαιδευτικά πλαίσια, έχει παρουσιαστεί ένα σημαντικό κομμάτι έρευνας στην επιστήμη των υπολογιστών. Από τις πιο σημαντικές που γίνεται ολοένα και πιο αντιληπτό είναι η αξιοποίηση των 3D ΕΚ σε μαθητές μικρότερης ηλικίας σε διαφορετικά μαθήματα του αναλυτικού προγράμματος και συγκεκριμένα του προγραμματισμού. Η χρήση της παλέτας του Scratch4SL (S4SL) αποτελεί ένα αξιόλογο εργαλείο που προσφέρει στους χρήστες ένα πιο φιλικό και απλούστερο τρόπο προγραμματισμού αντικειμένων εντός των ΕΚ. Η παλέτα αυτή περιλαμβάνει γραφικά μπλοκ κώδικα, για να αποφευχθεί η εκμάθηση ή η απομνημόνευση μιας γλώσσας προγραμματισμού των 3D ΕΚ που θεωρείται πιο πολύπλοκη.

Ωστόσο, μέχρι και σήμερα δεν είναι γνωστό εάν ένα παιχνίδι προσομοίωσης μπορεί να επηρεάσει τους μαθητές (αγόρια και τα κορίτσια) ως προς την ανάπτυξη και εφαρμογή στρατηγικών επίλυσης

προβλημάτων που σχετίζονται με την ΥΣ που περιλαμβάνουν την υλοποίηση της διαδικασίας μιας γνωστικής διεργασίας σκέψης από την κατανόηση ενός προβλήματος έως την έκφραση σχεδιαστικών λύσεων σε μορφή κώδικα. Ως εκ τούτου, η ερευνητική υπόθεση που αναδύεται είναι εάν ο συνδυασμός των σημαντικότερων σχεδιαστικών χαρακτηριστικών και των χαρακτηριστικών που προσφέρονται σε 3D ΕΚ, όπως η ρεαλιστική πιστότητα αναπαράστασής στο OpenSim σε συνδυασμό με το S4SL σε σχέση με ένα ΠΟΠ όπως του Scratch για τη δημιουργία του παιχνιδιού προσομοίωσης, θα μπορούσε να βοηθήσει τα αγόρια και τα κορίτσια να αποκτήσουν μεγαλύτερη κατανόηση της χρήσης δεξιοτήτων ΥΣ και ανάπτυξης στρατηγικών επίλυσης προβλημάτων. Το προτεινόμενο παιχνίδι προσομοίωσης περιλαμβάνει ένα ψηφιακό ρομπότ καθαρισμού, το οποίο θα αποτελέσει το αντικείμενο της πειραματικής αξιολόγησης και ανάλυσης των δεδομένων για την εξαγωγή σημαντικών συμπερασμάτων σχετικά με την μαθησιακή επίδοση. Η αξιολόγηση θα αφορά τις στρατηγικές επίλυσης προβλημάτων των μαθητών με βάση την ΥΣ (υπολογιστικοί σχεδιασμοί, υπολογιστικές πρακτικές και υπολογιστικές επιδόσεις) που δεν έχουν διερευνηθεί επαρκώς μέχρι σήμερα από σχετικές έρευνες της διεθνούς βιβλιογραφίας.

Βάση της παραπάνω προβληματικής που διατυπώθηκε, η επίτευξη των ακόλουθων τριών στόχων κρίνεται ως απαραίτητη για να επιφέρει την επίτευξη του βασικού σκοπού της έρευνας. Ο πρώτος στόχος είναι η σχεδίαση και ανάπτυξη του ίδιου περιβάλλοντος προσομοίωσης το οποίο περιλαμβάνει δραστηριότητες με διαφορετικά επίπεδα σταδιακής δυσκολίας σε κάθε πίστα ως προς την επίλυση προβλημάτων και την επιτυχή επίτευξη συγκεκριμένων μαθησιακών στόχων. Η δημιουργία του παιχνιδιού προσομοίωσης βασίστηκε στην αξιοποίηση ενός προτεινόμενου πλαισίου σχεδίασης με τίτλο «*ΠΔΟΣ: Προγραμματισμός για Διαδραστική Οπτική Συμπεριφορά*», το οποίο περιλαμβάνει συγκεκριμένες κατευθυντήριες προδιαγραφές περιγράφοντας παράλληλα τα βασικά χαρακτηριστικά και στοιχεία που μπορούν να υποστηρίξουν την ανάπτυξη της ΥΣ των μαθητών σε μαθήματα προγραμματισμού. Λαμβάνοντας υπόψη τις προτεινόμενες σχεδιαστικές προδιάγραφες, δημιουργήθηκε ένα παιχνίδι προσομοίωσης σε δυο πλατφόρμες (ΠΟΠ και 3D ΕΚ). Οι παίκτες καλούνται να βοηθήσουν μια γυναίκα μεγάλης ηλικίας με σοβαρά κινητικά προβλήματα προγραμματίζοντας και εισάγοντας αποδοτικούς και αποτελεσματικούς αλγορίθμους σε ένα ψηφιακό ρομπότ, έτσι ώστε να μπορέσει να κινηθεί και να καθαρίσει διαφορετικούς χώρους των δωμάτιων ενός μεγάλου σπιτιού.

Ο δεύτερος στόχος είναι η παρατήρηση και ο εντοπισμός πιθανών προβλημάτων ή/και δυνατότητων ενός πρωτότυπου παιχνιδιού προσομοίωσης το οποίο δημιουργήθηκε στο OpenSim αξιοποιώντας το S4SL για την επίλυση υπολογιστικών προβλημάτων. Για τον λόγο αυτό, κρίθηκε απαραίτητη η διεξαγωγή μιας προκαταρκτικής μελέτης ακολουθώντας μια μεικτή μεθοδολογία έρευνας για τη μέτρηση της εμπειρίας και των αντιλήψεων συνολικά δεκαπέντε (n=15) μαθητών γυμνασίου. Από τα ευρήματα από την ανάλυση του κώδικα της παλέτας του S4SL φανερώθηκε ότι οι μαθητές κατόρθωσαν να παρουσιάσουν σχεδιαστικές λύσεις μέσα από την σύνδεση δομών επιλογής και επανάληψης με μεταβλητές που συνδυάστηκαν με

σαφείς οδηγίες, δείχνοντας έτσι ότι μπορούν να εκτελέσουν αποδοτικούς και αποτελεσματικούς αλγόριθμους.

Ο τρίτος στόχος είναι η διερεύνηση της επίδρασης ενός παιχνιδιού προσομοίωσης, το οποίο δημιουργήθηκε στο OpenSim με το S4SL και στο Scratch, στην μαθησιακή επίδοση αγοριών και κοριτσιών Γυμνασίου. Για τον λόγο αυτό, διεξήχθη μια οιονεί πειραματική μελέτη για την συγκριτική αποτίμηση της επίδοσης εικοσιπέντε μαθητών (n=25) μιας πειραματικής ομάδας, η οποία χρησιμοποίησε το OpenSim με το S4SL και μιας ομάδας ελέγχου με τον ίδιο αριθμό (n=25), η οποία αξιοποίησε το Scratch για την αξιολόγηση της ορθής έκφρασης και εκτέλεσης των λύσεων που προτείνονται σε τρεις άξονες: (α) στην περιγραφή και στον καθορισμό με σαφήνεια των κανόνων, συμπεριφορών ή καταστάσεων που συνδυάζονται ως εντολές/οδηγίες δομών προγραμματισμού με φυσική γλώσσα και σε μορφή αλγορίθμου (ψευδοκώδικα), (β) στην ενδεχόμενη βελτίωση της έκφρασης υπολογιστικών πρακτικών, δηλαδή σχεδιαστικών λύσεων και προτύπων που προτείνονται και (γ) στην διερεύνηση του μαθησιακού κέρδους.

Τα αποτελέσματα της έρευνας έδειξαν ότι υπήρχαν σημαντικές διαφορές στην επίδοση των μαθητών της πειραματικής ομάδας σε αντιδιαστολή με τους μαθητές της ομάδα ελέγχου, διότι φάνηκε ότι κατόρθωσαν σε αρκετά μεγάλο βαθμό: (α) να κατανοήσουν χωροταξικά καλυτέρα τα βασικότερα μέρη του προβλήματος που έπρεπε να αντιμετωπίσουν μέσα στο OpenSim αξιοποιώντας δεξιότητες λογικής συλλογιστικής και κριτικής σκέψης για την καλύτερη δυνατή οργάνωση και εκτέλεση των σχεδιαστικών λύσεων, (β) να εκφράσουν σε μορφή ψευδοκώδικα και έπειτα να εφαρμόσουν πιο αποδοτικότερες και αποτελεσματικές σχεδιαστικές λύσεις κάνοντας λιγότερα λάθη και τέλος (γ) να επιτύχουν μεγαλύτερο μαθησιακό κέρδος ως προς τον προσδιορισμό δεξιοτήτων που σχετίζονται με την ΥΣ, το οποίο κυμάνθηκε σε διπλάσιο ποσοστό (41%) έναντι των συμμετεχόντων που χρησιμοποίησαν το Scratch (20%).

Συνθέτοντας τα ευρήματα των μελετών που διεξήχθησαν, δύο εκπαιδευτικές συνέπειες προκύπτουν. Πρώτον, τα βασικά χαρακτηριστικά και στοιχεία της επιφάνειας διεπαφής χρήστη που υποστηρίζονται από το OpenSim βοήθησαν τους μαθητές της πειραματικής ομάδας να αναγνωρίσουν και να χαρτογραφήσουν πιο εύκολα βασικά σημεία του προβλήματος. Αυτό διαπιστώθηκε και μέσα από την εφαρμογή σχεδιαστικών λύσεων τόσο σε μορφή ψευδοκώδικα όσο και κώδικα που πρότειναν, οι οποίες περιλάμβαναν λιγότερα λάθη σε σχέση με τους μαθητές που χρησιμοποίησαν το Scratch. Οι μαθητές που χρησιμοποίησαν τον ΕΚ κατανόησαν καλύτερα τις συνέπειες των επιλογών τους, λόγω της κατάλληλης ανατροφοδότησης που έλαβαν. Δεύτερον, η ρεαλιστική αναπαράσταση στοιχείων του παιχνιδιού που δημιουργήθηκε στο OpenSim σε σύγκριση με το Scratch βοήθησε σε μεγαλύτερο βαθμό την διερεύνηση και επίλυση προβλημάτων. Αυτό αποδείχθηκε τόσο βάση των εναλλακτικών επιλογών σε σχεδιαστικές λύσεις για την επίλυση των βασικότερων μερών του κύριου προβλήματος που προτάθηκαν από τους μαθητές, όσο και βάση της μέτρησης των δεξιοτήτων που θεωρούνται θεμελιώδεις για την κατανόηση της ΥΣ, όπως η επίλυση προβλημάτων, η αλγοριθμική σκέψη, η κριτική σκέψη και η δημιουργικότητα.

ACKNOWLEDGMENTS

My Ph.D. study was an extraordinary experience in my daily life. Besides the scientific knowledge that it unquestionably has provided to me, it as well as expanded my personal research concerns and strengthen my logical reasoning, critical and creative thinking skills. For these reasons, I would like to express my deepest thankfulness to a small but significant number of people who contributed to the successful completion of my dissertation. It would not have been possible without the constant guidance, support, and encouragement of a number of people, such as my Ph.D. dissertation committee and my family.

First, I would like to thank my supervisor, Dr. Spyridon Vosinakis for giving me the opportunity to work with him and being a continuous source of inspiration with his support, encouragement, insightful comments and constructive criticism. I would also like to thank the other two members of my dissertation committee, Dr. Panayiotis Koutsabasis and Dr. Konstantinos Tsolakidis for their time, advice, guidance, suggestions, and stimulating discussions.

Second, I would like to express my heartfelt gratitude to students and chief administrative officers of all high schools in Syros, where I conducted the empirical study for this dissertation. In particular, I would like to thank the three Computer Science teachers who allowed me to conduct my research studies inside and outside their classrooms, for their time, cooperation, support, and useful suggestions. I am also grateful to all the administrative and technical staff of the Department of Product and Systems Design Engineering for everything they have done all these years in their own way to support and complete my research.

Last but not least, I would be forever grateful to my family for their unconditional love and support throughout my life. I would like to thank my parents, Aikaterini and Stamkos, and my brother Themistocles for their unwavering patience, being my most tireless advisors. My family was, in my entire life, always a source of constant support and patiently seeing me through the daily ups and downs during my Ph.D. journey. Thank you for believing in me and teaching me to believe in myself.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
CG	Control group
CT	Computational thinking
EG	Experimental group
GBL	Game-based learning
GUI	Graphical user interface
IM	Instant message
LE	Learning effectiveness
LP	Learning procedure
L$	Linden dollar
LSL	Linden scripting language
OpenSim	OpenSimulator
OSVWs	Open source virtual worlds
PC	Personal computer
SG	Simulation game
SL	Second Life
S4SL	Scratch4SL
SVWs	Social virtual worlds
UX	User experience
VW	Virtual world

Chapter 1: Introduction

The first chapter provides a brief introduction to the background, the motivation with the problem statement, the methodology, the twofold research aim, and finally the contribution of this thesis. It is focused on the use of game-based learning, and specifically on game-making and game-playing approaches to support the teaching of cognitive tasks involved in computational thinking through computer programming courses. It also states to the empirical evidence from previous studies so as to verify how computer simulation games created via interactive environments, such as visual programming environments and 3D virtual worlds can become effective tools for high school students to impart theoretical and applied knowledge and assist them to overcome problems related to programming.

1.1. Background

Learning computer programming is an indispensable part of Computer science (CS) in K-12 education. Computer programming (or programming) is the process that allows someone to transform a high-level/abstract solution plan of a problem into a syntactically accurate set of instructions expressed in a formal language and evaluate its execution by a computing device (Lahtinen, 2005; Robins et al., 2003). One of the most significant objectives in programming courses is to foster students' rigorous thought process using skills such as algorithmic thinking, logical reasoning and coding so as to understand how to use correctly a set of rules with the precise expression for the formal structure of programming languages in problem-solving situations (Bocconi et al., 2016). By using such skills combined with the appropriate knowledge on how to use algorithms and programming constructs, students can learn how to plan and apply their problem-solving strategies to real-world problems (Tucker et al., 2003).

However, various studies indicate that learning computer programming is not without difficulties. Most students in K-12 education face difficulties about how to design and apply their problem-solving strategies. They need to propose and then apply their solution plans which are associated with two interrelated aspects (Dagdilelis et al., 2004; Webb et al., 2017): a) the decomposition of a problem into smaller subparts to analyze its given facts, and b) the formulation of algorithms to specify a series of steps so that test solutions with syntactically correct programs. The former comprises students' difficulties with abstract concepts in understanding the main problem and in expressing specific steps for decomposing it into simpler and manageable parts to design their solution plans. The latter includes the subdivision of a program into smaller pieces for each subpart of a problem, and the comprehension on hypothetical error situations that make to realize the correctness of their solution plans by testing the consequence of executing specific computer instructions (Qian & Lehman, 2017). Thence, they struggle to understand how to align

correctly the appropriate programming constructs with problem-solving strategies and tend to spend more time on mastering syntax and/or semantics of a programming language to various problems (Grover et al., 2015; Koorsee et al., 2015).

Over the last ten years, the term of computational thinking (CT) has gained much attention in programming courses. It is a problem-solving process that can allow humans to think how to use fundamental programming concepts and constructs in order to solve real-world problems (Bienkowski et al., 2015; Lye & Koh, 2014). CT has received considerable attention because it permits humans to use cognitive thinking skills and concepts which are related to programming (Barr & Stephenson, 2011; Witherspoon et al., 2017). There is common agreement that students mostly inside school context can also learn how to think logically and methodologically using CT in order to formulate their own strategies for developing and applying their solution plans to real-world problems, rather than focusing strictly on a technical activity for improving their computer literacy or coding skills (European Commission, 2016; Grover & Pea, 2013). Generally, with the development of CT, students place more emphasis on developing a set of cognitive thinking skills such as problem-solving, critical, and abstract thinking to decompose a problem into smaller subparts. This process can assist them to propose solutions with a sequence of instructions to each component so as to automate by expressing solution(s) in such a way that a computer can effectively carry out (Kalelioglu et al., 2016). Therefore, CT constitutes an ideal way for students to evaluate the correctness of their thinking solution plans into programs that can be executed by using only precise instructions and programming constructs.

More specifically, CT is regarded as one of the most indicative cognitive problem-solving processes for designing computing systems. Students can learn how to use skills related to CT in the direction of planning their own strategies for solving problems and transforming accurately their solution plans into syntactically correct instructions (Bocconi et al., 2016). The use of CT skills for proposing solutions to a problem is based on a computational problem-solving strategy and is associated with three interrelated processes (Chao, 2016; Liu et al. 2011): a) computational design is the use of logical and abstract thinking skills for the formulation and design of solution plans, b) computational practice is the expression of fundamental programming constructs, like selection, sequence or iteration for the implementation of solution plans, and lastly c) computational performance is the identification of the most efficient and effective solution plans into code that can be proposed to several problem-solving tasks.

Game-based learning (GBL) has been widely exploited in various learning subjects or domains. It is defined as a learning approach in which students can use computer games in order to practice or gain knowledge inside (or not) school contexts (Killi, 2005; Yusoff et al., 2018). The widespread utilization of GBL has today paved the way for a new level of students' engagement giving new opportunities to learn by making or by playing their own games (Dickey et al., 2005). With the emergence of digital games, many

efforts have been undertaken to develop digital environments in order to integrate educational content and materials into games so as to increase students' participation (Maloney, 2008). Gaming can greatly fulfill students' learning needs and experiences by supporting various learning tasks which correspond to an imitation of an operation of a process or a system consisted of specific simulated problem-solving situations of the real world. Thus, a remarkable GBL approach is the use of simulation games (SGs). A SG is a gaming environment that can permit users to participate actively in having specific task information to learn by doing within interactive and simulated problem-solving contexts (Garris et al., 2002).

The CT instruction through programming courses using GBL approaches is of great importance for many educators and scholars in recent years. In such programming tasks, students try to analyze simulated problems or situations and take the most appropriate decisions to propose their solution plans using skills related to logical and algorithmic thinking prior to the writing of a program so as to choose the most appropriate programming constructs to execute those plans (Adler & Kim, 2017; Davies, 2008). One of the most remarkable approaches to support GBL is the use of SGs. A SG created in interactive environments can fulfill the requirements in programming courses since it can present embodied problem-based contexts fostering students' problem-solving abilities to experience within a scientific discovery process in order to interact with digital elements and objects (Werner et al., 2014). This may lead students to learn how to think before starting to program by integrating interactions and rules inside objects/elements to develop and observe game situations in order to generalize those tasks later (Brennan & Resnick, 2012; Liu et al., 2017). Such tasks come in contrast to the most common exercises, in which students tend to formulate and write correctly instructions combined with programming constructs to observe the consequences of executing those constructs or to use certain constructs corresponding simply to specific problem-solving contexts.

The growing popularity of GBL in K-12 programming courses has given students the chance to use interactive environments so as to impart theoretical and applied knowledge for learning how to program using skills related to CT following two problem-solving approaches: (a) "game making" with tasks and/or exercises to design a game (Brennan & Resnick, 2012; Howland & Good, 2015) or (b) "game playing" with tasks and/or exercises by playing a game (Liu et al., 2017; Witherspoon et al., 2017). The most well-known platforms that students can use to create and/or play interactive games are as follows: (a) visual programming environments (VPEs), and (b) three-dimensional (3D) virtual worlds (VWs). A VPE is an interactive environment that allows users to construct programs and visualizations graphically using a palette with colored code blocks. A 3D VW is a computer-based simulated environment in which users can create avatars (digital figures which look like as humans' representations) to interact and explore with various visual objects or elements and participate to a wide range of problem-solving activities using remote synchronous communication, such as VoIP calls and asynchronous communication, such as instant messages and gestures (Topu et al., 2018). Also, the use of Scratch4SL (S4SL), as a visual palette offers a

more simper and user-friendly way for programming to avoid someone the complexity of a 3D VW's programming language to integrate behavior into visual objects/elements (Rosenbaum, 2008).

To date, there is good evidence that the use of interactive environments can significantly influence students in coding tasks but leaves open the discussion whether a computer game can support them to develop a more general understanding and using computational concepts to learn how to program (Denner et al., 2012; Howland & Good, 2015; Werner et al., 2015). For this reason, a game playing programming approach using a SG created in interactive environments is a notable option that needs further study (Hsu et al., 2018; Lye & Koh, 2014; Witherspoon et al., 2017). Therefore, the focus of this thesis is to investigate if a SG interface and elements created in 3D VWs and in VPEs can affect students' learning performance by assessing their computational problem-solving strategies (i.e. computational design, computational practices, and computational performance) for teaching and learning programming.

1.2. Motivation

During the last decade, several literature reviews on the field of CT in K-12 curricula have come to the statement that there is still an open discussion about the effect of computer games to support CT instruction. In their review analysis, Grover and Pea (2013) have unveiled the need to develop and use computer simulated problem-solving tasks using SGs either by developing new interactive environments or by combining already known design features and characteristics of the most well-known interactive environments. Additionally, Kafai and Burke (2015) have recommended the connection of features and characteristics of serious gaming movement over a computer simulation game that can be created in VPEs or in 3D VWs which can provide to software game designers considerable opportunities to design and propose a new one with simulated problem-solving tasks relevant to students' needs and demands. In their literature review report, Lye and Koh (2014) have also mentioned the need to propose directions towards a constructivist (thinking-doing) problem-solving learning approaches using SGs to support the demonstration of skills related to CT and programming. This statement is still intensifying more due to the lack of design frameworks and requirements for the creation of a computer game that can assist the development of students' computational problem-solving strategies (Grover et al., 2015; Hsu et al., 2018; Lye & Koh, 2014). First, it is appropriate to propose a theoretical framework and investigate if and what design features and characteristics either from VPEs or 3D VWs can facilitate the creation of a SG to support the development of students' computational problem-solving strategies.

Second, previous studies (Chao, 2016; Liu et al., 2017; Witherspoon et al. 2017) have argued on what students (boys and girls) can finally learn by playing SGs in programming courses. Due to the lack of conclusive findings, further empirical evidence about the impact of a game playing approach using SGs on students' learning performance is needed. Moreover, there is little evidence on what they finally learn using

skills and strategies related to CT in problem-solving contexts. In particular, there is still today not identified any study to investigate if there are any significant differences on students' pre-and-post learning outcomes based on their computational problem-solving strategies by playing a SG created in VPEs and 3D VWs due to the technological characteristics that make such a game to have different user interface design features and elements (Good & Howland, 2016). The most significant is the intuitive modality consisting of two elements. The first is the realistic simulated representational fidelity that a 3D VW offers by displaying a digital environment in three-dimensions. The second is the sense of presence when user's experience with the feeling of "being there" and the view of changes in objects' motion can lead to a greater perception and subjective sense of being within specific digital contexts (Dalgarno & Lee, 2010). Both elements can assist users to program and predict any possible instructions/movements by programming and integrating behaviors into objects. Such a process allows the observation and execution of their solution plans so as to assess the consequent results of those instructions in problem-solving contexts which are resembled similarly as those to the real world.

In addition to the above, recommendations for engaging boys and specifically girls provide a rationale on the use of interactive environments for playing computer games that strive to bridge the gender "gap" that exists in programming courses (Garneli et al., 2015; Grover & Pea, 2013). The rapid proliferation and utilization of interactive environments for CT instruction through GBL approaches have been significantly influenced not only the motivation and participation, mostly high school students with different gender, but also their learning performance (Carbonaro et al., 2010; Good & Howland, 2016). So far, recent studies have provided evidence regarding the code tracing analysis that boys and girls created using SGs focusing on: (a) game programming competencies through simulations and video-games construction via Scratch (Garneli & Chorianopoulos, 2017), (b) computational practices which are applied by using specific programming constructs so that make simulation-based applications or produce virtual exhibits via Scratch (Mouza et al., 2016), and (c) game creation by defining the interactions among players' characters and/or game objects that exhibited on their programming skills via Stagecast Creator (Denner et al., 2012).

However, less attention has been given to understand firstly whether a SG for game playing can affect boys and girls to develop their computational problem-solving strategies, and secondly, if it can support them to apply their computational practices. This process may lead from problem formulation to solution expression in practice (Chao, 2016; Liu et al., 2011; Lye & Koh, 2014). For such an effort, previous studies (Garneli & Chorianopoulos, 2017; Jakos & Verber, 2016) have stressed the need for conducting comparative studies to assess students' computational strategies by investigating the effects of computer games in simulated problem-solving tasks created in interactive environments so that analyze the way of developing and applying their solution plans into code. Other studies (Girvan et al., 2013; Kafai & Burke, 2015) have pointed out that future works need to investigate the effects of a SG created both in 3D VWs,

and VPEs on students' learning performance conductive to be clarified how appropriate are the attributes of each platform for developing and transforming into workable algorithms their solutions plans in an effort to understand what they finally learn.

According to the above, the main hypothesis is if the combination of the most significant design features and characteristics of a 3D VW such as the representation fidelity of OpenSim with S4SL for the creation of SG can support boys and girls to gain a greater understanding of CT more than if such a game is created in a VPE such as Scratch. In such a process, the assessment of students' learning performance requires not only the formulation and manipulation of a problem into smaller subparts with skills related to CT but also testing and debugging the correctness of solution plans to a problem by integrating with specific design patterns using control flow blocks from a visual palette inside visual objects in order to propose, express and apply their solution plans. Therefore, a twofold research challenge is arising. The first focuses on what SG features and elements can be utilized for the development of a theoretical design framework since there was not identified from the related literature specific design guidelines and recommendations for the development of a SG created in VPEs and in 3D VWs to support CT instruction. The second focuses on whether such a SG can support boys and girls to practice their solution plans so as to investigate the correctness of their decisions that includes a process from problem formulation to solution expression into code by expressing and applying their computational problem-solving strategies.

1.3. Research aim and objectives

The aim of this thesis is twofold. The first is to propose a theoretical design framework for the development and creation of a SG to support CT instruction with simulated real-world problem-solving tasks. The second is to investigate and analyze the effects of a SG on high school students' learning performance in programming courses. The proposed SG is created in OpenSim and Scratch for supporting boys and girls to demonstrate skills related to CT and understand the appropriateness of using specific programming constructs to simulated problem-solving learning tasks. Such a SG will be subject to an experimental evaluation and analysis, in order to provide the empirical evidence regarding students' learning performance by assessing their computational problem-solving strategies (i.e. computational design, computational practices, and computational performance). Thence, reaching the following three objectives will create a pathway to address the above research aim:

a) to develop a problem-solving environment so as to propose a SG that can be created in VPEs and 3D VWs by articulating a theoretical design framework with specific design guidelines and features.

b) to identify any potential problems and benefits regarding the use of a SG prototype created in OpenSim with S4SL to support CT instruction by measuring students' learning experience and first perceptions.
c) to investigate if a SG created in OpenSim with S4SL or in Scratch can affect the learning performance of students (boys and girls) to gain a greater understanding on the use of skills related to CT for developing, applying and transforming their solution plans into code by comparing their computational problem-solving strategies.

1.4. Methodology

Having identified the main hypothesis and the pathway to answer it, a research methodology to achieve the three objectives is planned and analyzed. To achieve the first objective, two steps required to be done: a) to identify the difficulties and problems on what boys and girls can understand in regard to learning how to program based on related works, and b) to explore the differences and similarities from previous studies which have tried to measure the effects of using interactive environments on the learning performance of boys and girls. Accordingly, the current thesis has to investigate the main hypothesis from two perspectives. From an instructional game design perspective, to achieve the first objective, a theoretical game playing framework with specific design guidelines and recommendations is proposed to inform and elaborate a design rationale on how a SG can be designed in order to support CT instruction and the development of students' computational problem-solving strategies in respect of gender equality.

From a methodological research design perspective, due to the lack of studies assessing a game playing framework, this thesis seeks: a) to test a prototype SG following the design guidelines of the proposed theoretical framework by conducting a preliminary and an experimental study, and b) to observe how and what features and characteristics of a SG can significantly support students' learning outcomes in programming courses. Thence, to achieve the second objective, a mixed-methods preliminary study is conducted in order to investigate if the use of a SG created in OpenSim with S4SL can support the development of students' computational problem-solving practices into code. Such a study can give initial evidence to discuss the potential reasons for using the proposed SG in order to identify any potential problems or any design and usability issues.

To achieve the third objective, a total of fifty (n=50) high school students who participated in this study divided into a control group (n=25) and an experimental (n=25) group that used Scratch and OpenSim with the S4SL palette, respectively with a view to supporting and applying their solution plans into code for the same problem-solving tasks. Consequently, such a study can give empirical evidence about the effects of the proposed SG by analyzing boys' and girls' problem-solving strategies focusing on:

a) computational design to express their solution plans for all subparts of the main problem,

b) computational practices to apply those plans into the code, and finally, c) computational performance to measure students' learning performance by identifying the most effective and efficient design patterns.

1.5. The contribution of this thesis

This thesis findings advances the knowledge about the use of interactive environments to support CT instruction through programming courses and provides several contributions. From a theoretical-instructional perspective, this thesis may be of great interest to instructional designers and educators who want to design or select to apply their programming courses through (in-)formal blended instructional formats (in-class and supplementary online) via SGs so that can foster students' computational problem-solving strategies. Therefore, a design framework with specific guidelines and features to support CT instruction is proposed by designing and developing a SG via VPEs and 3D VWs (Pellas & Vosinakis, 2017a).

From an instructional-practical design perspective, results of the main experimental study provide empirical evidence and valuable information on how and if the use of a SG created in two interactive environments can affect students' learning performance by applying their computational problem-solving strategies. The results from the preliminary study (Pellas & Vosinakis, 2017b) and the quasi-experimental study (Pellas & Vosinakis, 2018) can give insights into the appropriateness of a SG and suggestions on how it can support students to learn how to think and practice their computational problem-solving strategies. This thesis contributes to the field of CT in the K-12 curriculum by:

- articulating how a theoretical design framework with specific guidelines and features can be used for the development and creation of a SG in VPEs and 3D VWs to support students through game playing modes. The use of the proposed SG is focused on the support that students can have in order to develop skills related to CT and to apply their computational problem-solving strategies;
- proposing an instructive guided approach for learning how to program inside conventional school computer lab (formal) and through extracurricular activities (informal) outside it (online). An analysis of instructional tasks is outlined focusing on how in-game elements should be mapped to skills related to CT in the direction of helping students to use their problem-solving, logical and abstract skills for the analysis of their solution plans to subparts of a simulated real-world problem;
- testing a 3D prototype SG created in OpenSim with S4SL to investigate if it can increase the learning experience of fifteen (n=15) high school students in a preliminary mixed-methods study that is conducted in blended instructional formats (face-to-face and supplementary online);
- comparing the learning effect of a SG created in OpenSim with S4SL and Scratch through a quasi-experimental study with a larger sample of students (n=50) in order to measure their learning

performance and outcomes by assessing their computational design, computational practices and computational performance; and lastly,

- generating educational implications for theory and practice related to design guidelines combined with specific features building upon the experience gained by subsequent design and evaluation on how a game playing approach via a SG can support students to develop and apply effectively their computational problem-solving strategies.

1.6. Thesis structure

This section aims to provide an outline of this thesis and the relation between the chapters is depicted in *Figure 1-1* below. The current thesis consists of nine chapters. These are the following:

Chapter 1, which is the current chapter, introduces the background and rationale, the motivation, the research aims, objectives, main research questions and the contribution of the present thesis.

Chapter 2 presents the literature review issues related to Computer Science and specifically computer programming. It gives a pathway on how essential can become new directions of thinking and learning programming which are reflecting through the use of CT as a problem-solving process that has integrated into many curriculums across the globe. This chapter also provides some crucial gender challenges and issues about students' participation in programming courses.

Chapter 3 identifies the instructional approaches and educational environments which are widely utilized in programming courses. It gives emphasis to several noteworthy educational technologies and learning approaches such as educational robotics and interactive environments including platforms such as VPEs and three-dimensional (3D) VWs.

Chapter 4 addresses the basic characteristic of GBL and particularly the educational potentials of using games in educational settings focused on the use of VPEs and 3D VWs which can support CT instruction in computer programming courses. It also gives information about the related works which have focused on game making and game playing and whether specific user interface design features of games can foster students' skills related to CT.

Chapter 5 describes a theoretical design framework called *"PIVB: Programming for Interactive Visual Behavior"*. It offers the main design rationale, design decisions and design criteria regarding the use of a game playing framework that has been proposed to guide the design and development of a SG to support CT instruction.

Chapter 6 designates the game design and gameplay overview of a SG called *"Robot vacuum cleaner"* (RVC) simulator that is created via Scratch and OpenSim with the visual palette of S4SL so as to support students develop and apply their computational problem-solving strategies in instructive guided settings (formal and informal).

Chapter 7 demonstrates the experimental design and data from the statistical analyses resulted in two studies. The first aims to examine the effects of using the first prototype RVC simulator created via OpenSim with S4SL on teaching and learning how to program high school students (boys and girls). A preliminary mixed methods study is conducted to provide results about the learning effectiveness, the learning procedure, and user experience. The second aims to describe a quasi-experimental study in order to investigate and present results based on students' design patterns which are proposed and applied for solving the same simulated real-solving environment created as a SG in OpenSim with S4SL and in Scratch, and after that to compare their learning performance.

Chapter 8 gives an overview of the overall discussion and implications for practice and design. This chapter presents a view in regard to the future directions of this work focused on aspects that can improve the current state of the proposed SG and the aspects which can facilitate the acquisition of further implications for design and practice concerning CT instruction in programming courses.

Chapter 9 concludes this dissertation thesis. It summarizes major findings based on the previous chapters, consequences from limitations of both studies, and lastly, it offers conclusions that will lead to the appropriateness of the proposed SG to support CT instruction through high school programming courses.

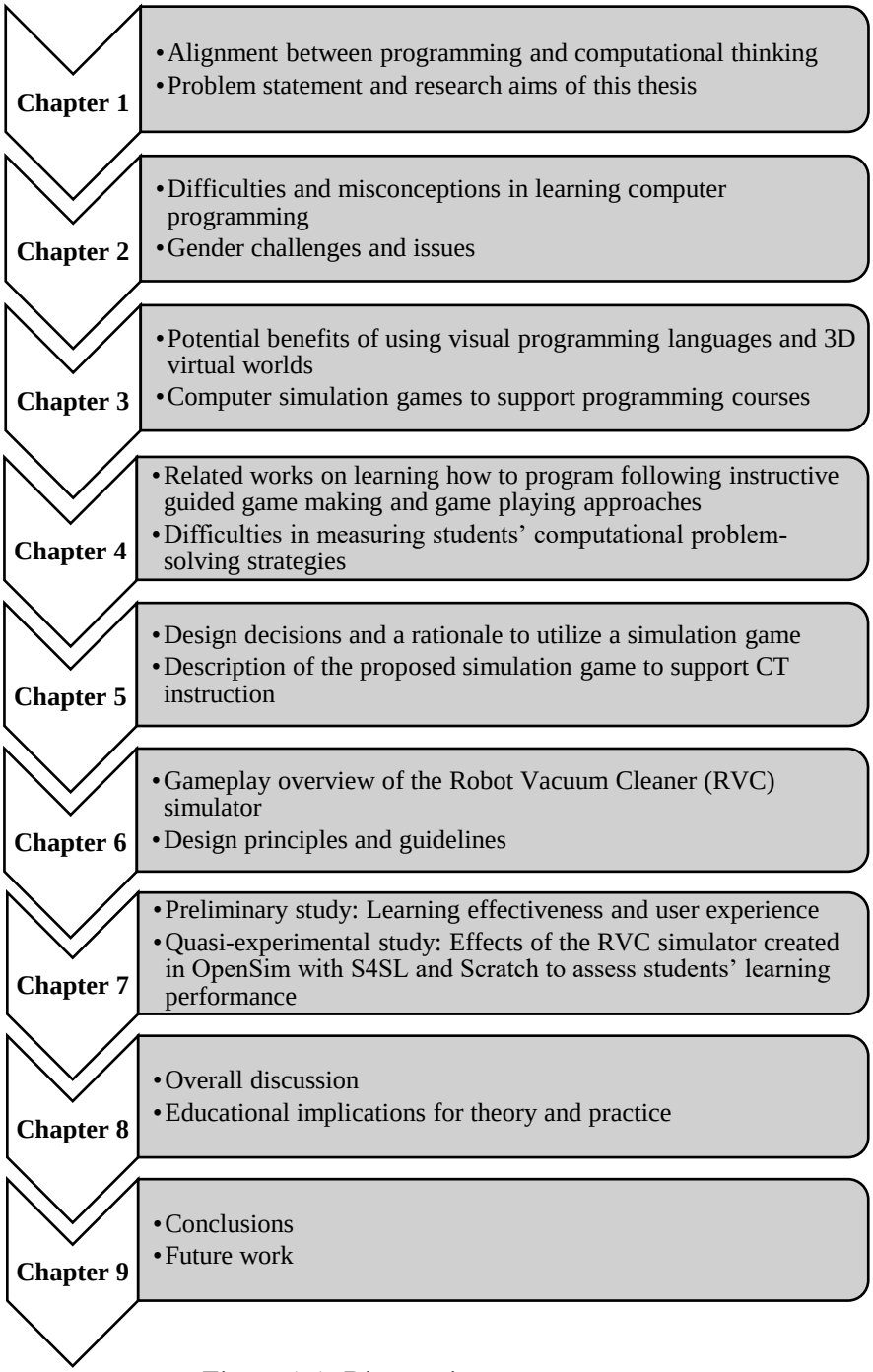


Figure 1-1: Dissertation structure

Chapter 2: Computer Science education

This chapter presents literature review issues related to Computer Science, computer programming and CT. It gives a pathway based on the related works which have identified how essential can become new directions of thinking and learning programming reflected by using CT as a problem-solving process that has been currently integrated as a significant term into many curriculums across the globe. More specifically, the current chapter defines distinctive characteristics upon CT with its cognitive abilities and analyses the major distinctions between problem-solving strategies in programming and computational problem-solving strategies. It also discusses the reasons why there is a dearth of evidence in the literature to support serious games as educational tools for those students who want to learn how programming constructs work properly for solving real-world simulated problems. Lastly, the current chapter investigates the main reasons why students with different gender (boys and girls) find programming difficult to learn, discussing the importance why CT can contribute more in depth to coding.

2.1. Computer Science

Computer Science (CS) is one of the fastest growing formal scientific disciplines. CS scientists learn how to design, develop and use the computing technology. More specifically, the main purpose of CS is to investigate, identify, and finally propose the theoretical foundations, the nature of data structures, algorithms and computations (Rapaport, 2005). Computer scientists use methodologies from both formal and applied sciences with a particular kind of mathematically based techniques in favor of specifying, designing, developing, and verifying software and hardware systems. CS includes the systematic study of the feasibility, structure, expression, and mechanism of processes (or algorithms) such as are processing, storage, communication and access to (big) data. Main domains of study contain artificial intelligence, computer systems and networks, human-computer interaction, vision and graphics, programming languages, software engineering, and theory of computing (Denning, 2000).

The advantages in technology and society due to the rapid proliferation of CS have brought several benefits on humans' daily life. CS is not just important for people who want careers related to technology, but it is also important for those who want to be well-educated in a modern society with advanced demands and needs (Sentance & Csizmadia, 2017). For this reason, CS has been integrated into many curricula across the globe due to the impact of cutting-edge technologies and devices which are existed in everyday life of students with multiple aspects, especially for those at a younger age (Tuomi et al., 2017). A significant aspect is to acquire the appropriate cognitive thinking skills in order to solve (real-world) problems that all people face daily. This aspect requires the study of designing, developing, and analyzing software and

hardware that can be used for solving a variety of problems ranging from business to scientific contexts. In this perspective, students need to learn what CS and its core components can offer. Also, another important aspect is to investigate how CS and its core components can be utilized in real-life and its importance as a learning discipline that can entirely assist humans to solve problems in practice (Hamlen et al., 2018).

Programming as a learning subject is one of the most significant core components of CS that can fulfill the above aspects. An important objective in programming courses is to foster students' rigorous thinking using skills such as algorithmic thinking, logical reasoning and coding so that understand how to use correctly a set of rules with the precise expression for the formal structure of programming languages in problem-solving situations (Bocconi et al., 2016). To succeed in such an effort, students need to have the opportunity to develop a wide range of both cognitive thinking skills and use fundamental concepts of CS. Moreover, it is worthy for students to use a set of the following cognitive thinking skills (Fluck et al., 2016; Qian & Lehman, 2017):

a) problem-solving which are reflected on understanding a problem and its subparts, drawing up a plan with specific instructions in order to develop a strategy to solve it, and
b) higher-order which are referred to the analysis and data synthesis, formulation of conditions and relationships in order to express and communicate a solution plan on how a problem can be solved and evaluate its accuracy.

The above-mentioned skills are essential in a cognitive thinking process because all those skills are associated with students' understanding on how to use correctly algorithms to solve problems following a set of specify steps that can be applied with the appropriate use of programming concepts and constructs (Tucker et al., 2003). Therefore, students in CS and specifically in programming courses must have the appropriate knowledge about programming languages and theory of computing so as to develop and analyze their solution plans to problems. This process makes students able to design solutions in favor of verifying the correctness of their thinking solution plans through coding. A suggestive way to achieve this objective is when programmers (novices and experts) learn how to understand the main problem and its subparts, formulate a solution in a structured form (algorithm or pseudocode), and then transform a proposed algorithm into a source code of a programming language. This approach can allow students to think before starting to code, rather than focusing only on a strictly technical activity to enhance their technological literacy or programming skills (Robins et al., 2003).

2.2. Computer programming

Programming is a significant learning subject in CS and even more to every major technological development in recent times. The importance of student knowledge related to computer competencies and specifically of programming has been to a great extent recognized by many curricula across the globe (Fluck et al., 2016; Webb et al., 2017). Computer programming (or programming/coding) is the process of planning, developing and applying various a set of various instructions that enable a computing device/machine to perform certain tasks with the purpose to solve problems and provide a way of interaction between humans and computers. The instructions and commands combined with specific constructs and concepts that can be written in a programming language are considered as computer programs and can be used by computers or computing devices to operate and execute those certain tasks (European Schoolnet, 2015).

Algorithms and programming concepts are two of the most distinctive terms that someone needs to know in order to learn how to code. The total of instructions and commands which are given to a computing device are firstly expressed as algorithms. The algorithms combined with the operating rules and programming constructs can give deterministic pre-set results that a computing device executes (Tucker et al., 2003). Thus, algorithms are like the recipes that everyone has to follow in his/her everyday life, such as in a doctor's prescription. To this notion a set of specific properties that need to be provided in any algorithm. Accordingly, algorithms to be potentially transformed into workable programs should entail the following (Kirkwood, 2000; Koorsee et al., 2015):

a) an algorithmic plan with specific (step-by-step) instructions in order to be achieved a proposed solution;
b) an effective solution that gives the desired result to finite time;
c) an efficient solution that reaches the best result in a possible way with the less possible use of programming constructs and commands;
d) a series of programming commands and constructs which are applicable and repetitive in order to be executed as many times as possible in similar cases;
e) expression of certain instructions and programming constructs/concepts in order to be transformed as programs of a programming language and to be executed by a computing machine.

To understand better the appropriateness of algorithms and programs, *Figure 2-1* shows a diagrammatic workflow that depicts an algorithm (on the left side) and program (on the right side) to turn on the light using a lamp light bulb that exists into a computing device.

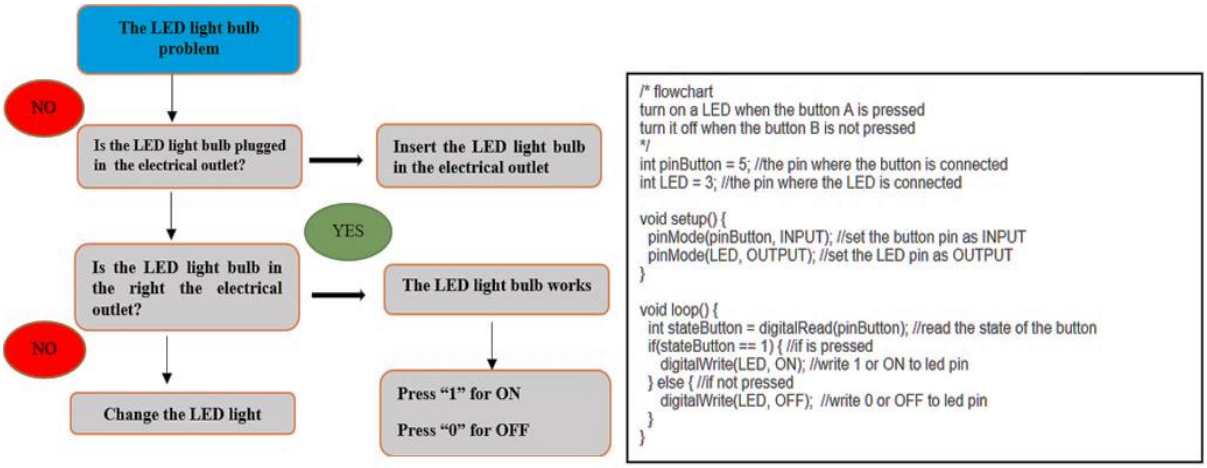


Figure 2-1: A workflow of the control algorithm and the program to turn on the light of a lamp light bulb

To be considered as workable plans and to be executed certain algorithms accurately by a computing machine, a set of fundamental programming constructs and concepts need to be known. The most important are the following (Burke, 2012; Robins et al., 2003):

a) "*Sequence"* is a series of individual steps and comprises a set of commands placed one beneath the other. Programmers can use a sequence of commands in order to solve problems where the order of execution regarding a set of actions is given. It contains a number of actions, but no actions can be omitted in the sequence. The sequence can be combined with other constructs and variables. In this programming construct belongs the following commands:

   i. Input values in variables.
   ii. Output values in a unit.
   iii. Assign a value to a variable.

b) "*Selection"* is to a large extent utilized in problems where it is necessary for programmers to make some decisions related to specific criteria, which may be different for each instance of a problem. The selection process involves checking a condition with two possible values (true or false) and then a decision to execute a command depending on the condition. It is distinguished mostly in the following two formats. The first is the "simple" selection (*"if...else"*). If the condition is true, then it is true, as many commands/instructions are executed within. If the condition is false, the command is executed immediately after the end of the selection. Otherwise, the commands that are underneath are executed differently. The second is the "nesting" selection (*"if...else…if.... else"*). It is the execution of two different codes to investigate if a statement inside a program is executed as true or false. It is also possible to have a choice which includes more than two possibilities. Such a statement refers to the "nesting" *"if...else"* statement. Using "nesting" selection, programmers

can check the correctness of a program with multiple tests by executing different codes for more than two conditions.

c) "*Iteration (or repetition/loop)"* is the execution of the same sequence of commands and constructs multiple times when the condition is true either for more than one time or when the commands of a condition are pre-defined. The logic of iterative procedures is regarded as essential when a sequence of commands is executed as a set of cases that have something in common and it must be performed more than once. An iteration is always controlled by a condition that determines the output from it. Repeat commands are the loop of repetition. This programming construct is expressed in three forms, implemented with the following commands:

   i. *"As long as ... Repeat"* is a process where the repeat check is done at the beginning.
   ii. *"Begin repeat…Until to..."* is a process where the repeat check is done at the end.
   iii. *"For... From...Until..."* is a process where the number of repetitions is known.

In *Figure 2-2* below are presented the three examples of fundamental programming constructs using a visual language via Scratch which are widely utilized for the creation of computer programs.

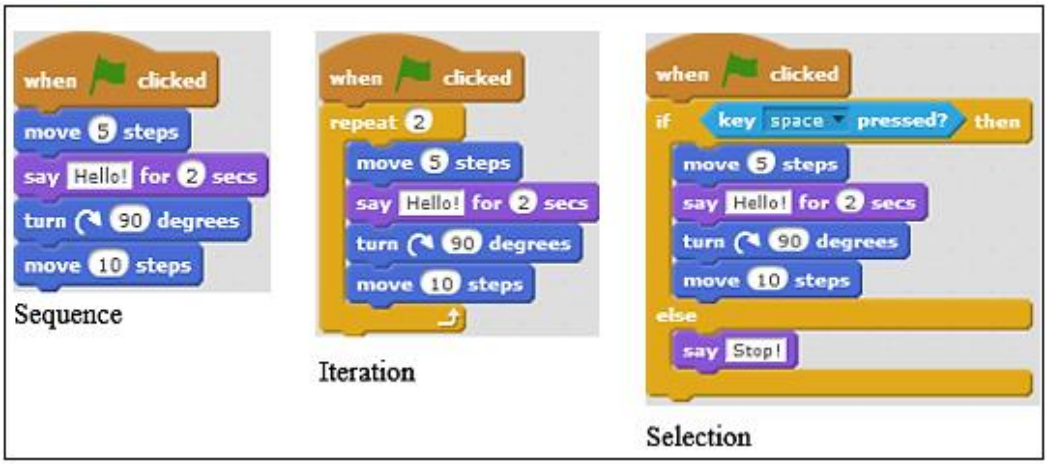


Figure 2-2: Fundamental programming constructs and examples using a visual language

Another significant point of view is that programmers need to demonstrate a number of skills. In particular, code comprehension, code generation, and debugging are three of the most important (Ring et al., 2008; Tucker et al., 2003). First, code comprehension includes specific programming constructs which are used into a program for proposing a solution to a problem. It pertains to the arrangement of constructs and concepts of a programming language into well-formed programs and ensures that someone knows why certain programming constructs are used in programs (Robins et al., 2003).

Second, code generation is the first step and it is more abstract than programming knowledge (de Raadt, 2007). Such a process refers in two aspects (Dalton & Goodrum, 1991; Davies, 1993):

a) to the design and implementation of semantic knowledge is the understanding of basic programming concepts and constructs that are being used in a computer program and
b) to the syntax of a programming language in which are used specific constructs and rules into a program.

Third, the debugging process is related to a specific instance of troubleshooting in a general problem-solving process. A general set of steps that need to be done as a process that entails the identification of a problem, its isolation, and the recognition of what each programming construct that is being utilized may cause in such a problem to confirm its correction and appropriateness (Ring et al., 2008).

2.3. Learning to program

During the last fifty years, learning to program (or programming) is one of the fundamental skills for children to learn in K-12 education. Programming courses in school-age contexts have offered several potentials on how students can think before start coding (Papert, 1980). It is expected that students will know fundamental programming concepts and will try to develop skills related to higher-order and algorithmic to solve problems (Webb et al., 2016). Specifically, students at a younger age (12-16 years old), can start to learn how to program using fundamental programming constructs (Tuomi et al., 2017).

To be successful the first introduction of students in programming without prerequisites, a variety of activities based on specific learning tasks need to be provided. The overall goal is to get students acquainted with programming and solution plans for solving real-world (or computational) problems with specific challenges expected by their learning outcomes and achievements. Some of the most crucial outcomes are the following (Robins et al., 2003; Qian & Lehman, 2017):

a) analyze and explain the behavior of simple (or complex) programs involving the fundamental programming constructs;
b) apply the techniques to break a program into smaller pieces (decomposition) in order to give an answer to each subpart of a problem after that with programming,
c) design, implement, test, and debug a program in which can be integrated fundamental programming constructs;
d) modify and/or expand (smaller or larger) programs using conditional or iterative control programming constructs;
e) choose the most appropriate programming constructs (e.g. conditional, sequence, iteration) for each part of a given problem, and;
f) describe and present how workable is an algorithm and/or code can solve a given problem.

Students in programming courses can also learn to describe algorithms as pseudocodes and written in natural language is indicated as an easier way to formulate subparts of a problem before applying into

code their solutions. Such a process can assist students to propose algorithmic solutions expressed in various formats and use synonymous/analogical terms for the same programming constructs and concepts which they want to use, such as for example, *"repeat"* or *"for"* so as to express repetition (de Raat et al., 2006; Myers et al., 2004). Such a process can assist students to go a step forward to surface features of the syntax complexity and think how those constructs are expressed in a more "natural way" as existed in their daily life (Grover et al., 2015). For example, the description of an algorithm as pseudocode in natural language is referred as a means between algorithms and programs that can deepen knowledge acquisition regarding computing concepts in contrast to superficial syntactical details about a specific programming language (Davies, 2008; Good & Howland, 2016).

The assessment of students' learning performance is one of the most important issues in computer programming courses because until today there are appeared various ways to be measured their outcomes and achievements. One way of assessing of students' learning performance is to check errors or combinations of programming constructs which are used on their final programs (Chao, 2016; Kalelioglu et al., 2014; Liu et al., 2013). Nonetheless, such an assessment gives an incomplete picture about how students can try to understand and utilize properly their algorithmic and cognitive thinking skills (Grover et al., 2015; 2017; Lye & Koh, 2014). Another indicative one is the way of understanding how correct is expressed as a solution plan based on student's rigorous cognitive thinking to describe specific constructs and commands that can be used with a logical sequence of steps. A solution plan can be first formed as pseudocode before starting to code it properly (de Raat et al., 2009; Robins et al., 2003). Such a process encompasses students' decisions ranging from the problem formulation to the solution expression by transforming pseudocode into workable plans and algorithms (Liu et al., 2017; Pane et al., 2001). Other researchers (Howland & Good, 2015; Myers et al., 2004) have argued that CS instructors need to encourage students to use pseudocode, as a step-by-step logical reasoning process so that express a solution before start coding. In such an effort, students as novices can bridge the "gap" between the theories of knowing *"why use"* and *"why need"* to execute into code precise rules, instructions, commands or concepts and/or limitations combined with programming constructs.

2.4. Problem-solving strategies in programming

A problem-solving strategy in programming is related to the design and development of solution plans to real-world problems in practice. It refers to a number of specific instructions which can be combined with fundamental programming constructs, such as sequence, selection or iteration for executing and assessing the consequent results of those instructions (Koenemann & Robertson, 1991). It also relates to the way that someone thinks how to plan and design a solution in order not only to use but also to know

how and why s/-he needs to use and apply any programming construct or concept in computer programming (de Raadt, 2006; Robins et al., 2003).

Recently, there is a common conviction that two specific problem-solving strategies are usually noticed. In particular, two of the most useful composition strategies associated with programming in practice and can be typically utilized into solution plans (Chao, 2016; Soloway, 1986):

- the *"abutment"* that describes a method of gluing two programming plans together in order to create a sequencing process for the transformation of sub-goals into code.
- the *"nesting"* that represents a method of combination between one programming plans into another by permitting a strategy that has a different structure from the previous one.

Both strategies are considered appropriate for solving real-world problems through programming and thus such strategies can also support the formulation of a problem to achieve someone algorithmic sub-goals. The composition of programming plans which can benefit programmers and specifically novices to learn different ways of designing and implementing their solution plans (Chao, 2016; Luxton-Reilly et al., 2018). Related works (Ismail et al., 2010; Ring et al., 2008) have argued that planning a problem-solving strategy facilitates programmers not only to decompose a problem into a set of intermediate subparts but also lead them to use the most appropriate programming constructs for proposing a solution.

The development of problem-solving examples is an indicative way for someone to utilize his/her proposed strategies in programming courses inside well-designed tasks so that solve certain tasks with specific steps using fundamental programming constructs and concepts. Problems at the sub-algorithmic level are the most indicative for several learning tasks in programming courses (Ring et al., 2008). To give answers in any problem-solving example, students need to learn first of all how to decompose a problem and identify its (sub-)parts properly. Decomposition is the process of dividing a problem into component parts in order to become more manageable. It is a process that helps someone to organize and manage large or more complex projects (Koorsse et al., 2015; Webb et al., 1986).

Several notable ways have been proposed to understand and evaluate how students try to develop and apply a problem-solving strategy in programming using algorithms and workable programs. To this notion, students are asked to identify the main problem and to state its subparts in an effort to formulate an algorithm by specifying a series of steps on how to solve each one's sub-goals with a programming language (Robins et al., 2003). Therefore, a twofold substantive way to understand and evaluate how students try to develop a problem-solving strategy encompasses the following two aspects (Kiesmüller, 2009):

a) the proposed solution that is recognized more easily by describing an algorithm with its specific steps in a logical order combined with specific constructs using pseudocode or simply expressions written to a natural language.

b) the execution of proposed algorithmic instructions that is necessary to be applied as an attempt to transform a proposed solution into the source code of a programming language.

2.5. Computational thinking

Computational thinking (CT) is a term that has been much discussed in the past years in CS and specifically in programming. Papert (1980) was the first who proposed the research on CT with programming for young students using LOGO programming projects. In 2006, Wing has actually redefined and reformed the term of CT. She argued that it is a method for *"solving problems, designing systems and understanding human behavior, by drawing on the concepts fundamental to computer science"* (Wing, 2006, p. 33). Also, there has been an ongoing discussion in the research community about the definition of CT, and its relationship with other types of analytical competencies, such as mathematical, algorithmic and engineering thinking. For example, Denning (2009) has defined CT as *"thinking with many levels of abstractions, use of mathematics to develop algorithms, and examining how well a solution scales across different sizes of problems"* (p. 28). Additionally, Fletcher and Lu (2009) have stated that CT is not about thinking like a computer, but it is about *"developing the full set of mental tools necessary to effectively use computing to solve complex human problems"* (p. 260).

A substantial body of literature (Grover & Pea, 2013; Lye & Koh, 2014) and a significant number of policy reports (ACM Education Policy Committee, 2014; Bienkowski et al., 2015) came to the conclusion that CT is a problem-solving process that allows humans to think about how to use fundamental programming concepts and constructs in order to solve real-world problems. CT comprises the following three stages (Kalelioglu et al., 2016; Korkmaz et al., 2017):

a) solution execution and evaluation of a strategy to propose a solution (computational problem-solving strategy).
b) decomposition and formulation of the main problem (abstraction),
c) description and expression of a solution (automation), and

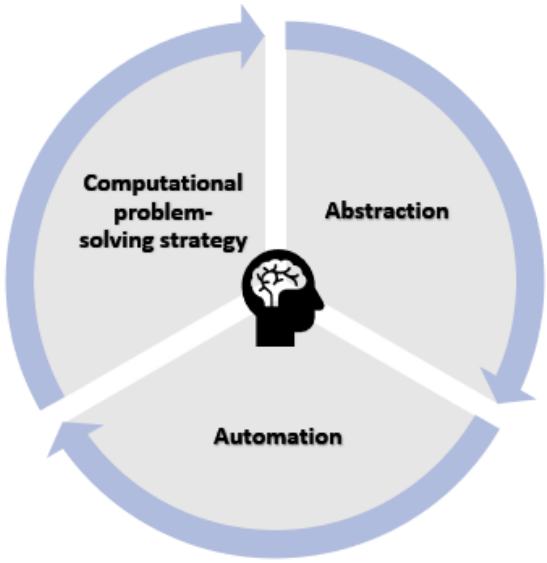


Figure 2-3: A cognitive thinking process using computational thinking

Figure 2-3 above depicts a cognitive thinking process using CT. Within the first stage is the computational problem-solving strategy that refers to two aspects. The first is a cognitive thinking process for the analysis of steps for decomposing and formulating subparts of a problem using critical, logical and abstract thinking skills. The second is the implementation of proposed solution plans that expressed and applied as programs so as to debug and understand the correctness of such a thinking process (Grover & Pea, 2013; Liu et al., 2011).

In the second stage, abstraction refers to the ability that someone has to decide what details of a problem are important to keep, and what details can be ignored when solving it by keeping only the most necessary ones (Selby, 2015).

In the third stage, automation is associated with algorithmic thinking. It is the ability to approach a problem by breaking it into smaller and solvable parts before formulating a specific set of steps to solve them properly. In the context of CS, algorithmic thinking and thus programming is a technical process that involves the use of constructs and concepts such as sequences, conditionals, and iterations (loops). In this perspective, students can understand how CT becomes a thinking problem-solving process before starting to code using fundamental concepts and constructs programming (Lye & Koh, 2014; Grover & Pea, 2013).

A summary of the most significant abilities that human can develop in such a process is presented in *Table 2-1* below.

Table 2-1: Knowledge and abilities gained by using computational thinking

Three stages of a cognitive thinking process related to CT	Abilities
Computational problem-solving strategy	- Decompose a problem to smaller parts - Analyze specific steps for subdividing and formulating all subparts of the main problem - Propose a solution plan using critical, logical and abstract thinking skills - Examine the correctness and appropriateness of thinking solution plans using programming
Abstraction	- Identify the main problem and its subparts - Keep important details and information
Automation	- Formulate a thinking solution to a workable algorithm - Use fundamental programming concepts and constructs

Despite the different definitions, potentials, and benefits of CT in several domains of humans' daily life, its main substance is clear because such a process is focused more on the development and use of a wide range of cognitive thinking skills for problem-solving. CT is considered as a problem-solving method that requires the use of logical thinking with concepts fundamental to computing in favor of conceptualizing, developing abstractions and designing systems (Hsu et al., 2018; Kalelioglu et al., 2016; Korkmaz et al., 2017). It includes a cognitive thinking process related to the formulation of problems and solutions that needs to be presented in a form that can be applied by an information processing agent (Wing, 2011).

To be achieved the above cognitive method from someone who wants to know how to start thinking logically and methodologically in a cognitive-mental process, it is required the development and the use of thinking skills, such as problem-solving and higher-order thinking. The latter can be further analyzed into critical thinking and logical reasoning. These skills are combined with creativity can lead to algorithmic solutions for real-world problems (Korkmaz et al., 2017). For example, students need to develop higher-order thinking skills like critical thinking, logical reasoning, and creativity with CS core concepts, such as decomposition, data analysis or events that may occur as a cause of this problem by clearly articulating the steps leading to a solution. Skills related to CT can assist humans to a great extent (Davies, 2008; Kalelioglu et al., 2014; Wing, 2011):

a) to develop logical reasoning on how to solve problems, regardless the utilization of programming languages, as they try to use such skills to a variety of problems that encountered in different domains of science, such as Formal Sciences or Engineering.
b) to analyze a problem methodologically by decomposing it in specific steps in order to give solutions to its piece using more effectively and efficiently programming constructs and concepts,
c) to propose a solution to more complex or larger problems by applying different solutions and developing design patterns as solutions for similar problems that can be delivered, and

d) to use and evaluate the appropriateness of computational tools so as to apply solutions to problems using concepts and constructs related to CS and programming.

2.6. Computational problem-solving strategy

A computational problem-solving strategy is the most important process of CT. It refers to the execution of programming constructs and reflects on the evaluation of the correctness of a solution into workable plans and algorithms. It encompasses the core concepts of CT related to abstraction, algorithm, automation, decomposition, debugging and generalization which are utilized by someone to understand the main CT concepts for proposing solutions to a problem requiring (Bienkowski et al., 2015; Davies, 2008):

a) the subdivision of a problem into manageable parts (decomposition),
b) the development of instructions to solve problems with specific tasks (abstraction),
c) the recognition of algorithmic solution plans as design patterns which can be applied into code (algorithm design), and
d) the way that a thinking solution can be generalized as a solution plan with certain design patterns (computer programs) to similar problem-solving tasks (pattern recognition).

However, there are appeared major distinctions between a problem-solving strategy in programming and a computational problem-solving strategy. The former is focused on program comprehension and modification of (large) programs which is a complex problem-solving process but not on what types of problems can be solved with those programs. Koenemann and Robertson (1991) have discussed how programming constructs are generated properly to operate the functionality of code that can be (re-)used to build any new or revised hypotheses during the comprehension process. Such a process requires the demonstration of code commands mainly in a "top-down" approach of comprehension in order to be used/revised any missing or failing operation for directly relevant code units that have to be copy-edited to similar/relevant cases. To achieve such a process, specifically high school students as novices usually follow a *"trial and error"* process to start learning how to program (Luxton-Reilly et al., 2018). For example, they usually try to find the correct way using different processes which are not always giving the appropriate answers to any specific problem that they have to face in order to apply their solution plans. Thus, programmers sometimes aimlessly provide different possibilities one-by-one and many times if their workable programs cannot "fit", they abandon any other reasonable effort to identify and understand what and how using programming can solve problems. The consequence is that programmers in such cases provide less attention to decompose a problem and identify its subparts, thus trying to reuse any solution plan faulty in similar problem-solving conditions without known why and if those commands and constructs are the most appropriate to apply their solution plans (Kiesmüller, 2009).

The latter includes the core concepts and concepts related to CT, giving a set of thinking steps which requires a process starting from problem formulation to solution expression to real-world problems. More specifically, it is regarded as a *"bottom-up"* process including the following two perspectives (Hsu et al., 2018; Chao, 2016; Davies, 2008):

a) the use of cognitive thinking skills as problem-solving, critical, and abstract thinking to decompose a problem into smaller subparts before starting to code their solution plans by trying to think with different levels of abstractions which will not contain any unnecessary information, and then trying to combine their practical skills in mathematics combined with algorithmic thinking (computational design).
b) the use of fundamental programming constructs, not thinking like computers but using programming language of computers to know and utilize its major components like selection, sequence or iteration (computational practices) so that can be applied any solution plans in an effort to be performed the most efficient and effective programs (computational performance). This means that programmers should know what constructs can be utilized to solve a problem and not just adopting and changing any particular programming constructs of their (previous/relevant) solution plans using *"trial and error"* methods.

The development of students' computational problem-solving strategies for applying programming knowledge correctly when formulating a solution to a real-world (computational) problem is one of the most crucial topics in contemporary CS courses (Tuomi et al. 2017). More specifically, the process of applying a computational problem-solving strategy is related to the analysis of designing, planning and debugging a proposed solution that is regarded as a perfect way to evaluate the correctness of a thinking process (Bienkowski et al., 2015). Such strategies are focused on a specific domain since novices need to decompose a problem, to analyze given facts, such as input and output to express specific steps/instructions and apply a workable plan as a program for solving it (Webb et al., 1986). A computational problem-solving strategy can also help programmers to organize the data gathered in order to rationalize a proposed solution efficiently for each subpart of a problem by investigating analytically specific issues that are associated with algorithms (Fluck et al., 2016; Webb et al., 2017). In particular, *Figure 2-4* depicts the development process of a computational problem-solving strategy where programmers require to have cognitive thinking and programming skills in order to apply their solution plans following a set of specific steps (CSTA & ISTE, 2011; Grover & Pea, 2013):

- Decomposing and understanding the subparts of a given problem in order to formulate and decide which programming constructs of a workable program can be used as the most appropriate to each part in terms of proposing a solution.
- Producing algorithmic solution plans for proposing a solution to each subpart of the main problem.

- Transforming the main algorithm into code of programming (formal) language that can understand a computing device.
- Testing and debugging a program to evaluate someone the correctness of his/her innate thinking solution into code is required.
- Proposing and generating a solution plan depends on its applicability and operability as a workable program that can be applied in similar problem-solving tasks.

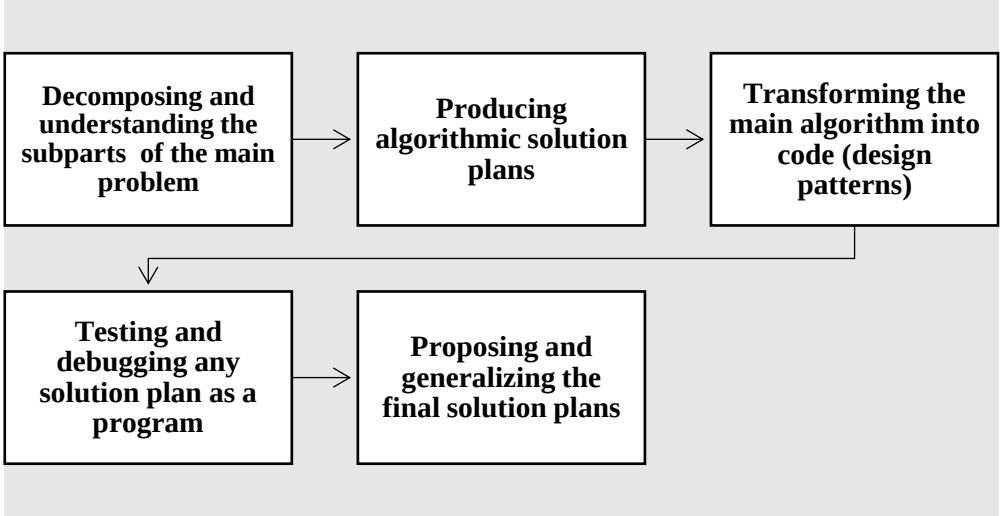


Figure 2-4: A process that provides the development of a computational problem-solving strategy

Within school-age instructional contexts, a computational problem-solving strategy in relation to programming is of particular importance on students' learning performance as it reveals:

a) a rationale that someone has to describe, express and apply his/her solution plans into workable plans and algorithms (Bachu & Bernard, 2014; Davies, 2008),
b) alignment between a thinking process for solving a problem (solution plan) and a coding process that includes *"know how"* the syntax and semantics of a programming language in order to apply such a plan (Brennan & Resnick, 2012; Liu et al., 2011); and
c) a way of using and writing what are the same code parts of a program for larger or more complex problems can be utilized, similarly as those from other subparts (Grover et al., 2015; Repenning et al., 2010). Thus, students can more easily suggest and compare their proposed design patterns which can be utilized in similar problems without interpreting a line-by-line coding process of a "top-down" approach (Denner et al., 2012; Werner et al., 2014).

In addition to the above, the creation and execution of a program based on a computational problem-solving strategy can assist CS instructors to assess the applicability and correctness of such a process and measure properly their learning performance (Bienkowski et al., 2015; Grover et al., 2015; Liu et al., 2017).

Previous research efforts (Repenning et al., 2015; Werner et al., 2015) and literature reviews (Grover & Pea, 2013; Kafai & Burke, 2015, Lye & Koh, 2014) have argued that a computational problem-solving strategy paves a pathway of recognizing the prerequisites in a broad range of analytical and logical ways of human's thinking on how to solve problems finding the most efficient and effective ways in order to apply solutions. Consequently, students should try to formulate their plans and goal constructs based on their computational problem-solving strategies that need to be applied through programming (de Raat, 2007; Robins et al., 2003).

2.7. International policy reports about computational thinking

The widespread deployment of Information and communication technology (ICT) resources has generally contributed to the rapid proliferation of CT. The rapid growth of the educational and scientific community seeks to investigate different ways of promoting CT, and thus extensive and large-scale projects by a significant body of policy reports have given much information about this topic and its impact on educational contexts. First of all, the National Research Council (NRC) has organized two workshops to address the confusion about the definition of CT by bringing together educators and scholars from a broad range of disciplines in 2010 and 2011. In the first workshop, participants discussed the nature of CT and its cognitive parts with several implications in education (NRC, 2010, p. viii). The same report suggested the following:

a) students need to learn thinking strategies such as CT as they study a discipline,
b) teachers and curricula need to provide the appropriate guidelines in order to apply students' computational problem-solving strategies, and lastly
c) the CT integration needs to have an appropriate instructional guidance that can enable students to learn how to use skills and concepts related to computational problem-solving strategies.

In 2011, the second workshop was focused on the exploration of the pedagogical aspects of computational thinking in the context of K-12 education (NRC, 2011). The results revealed that CT is a problem-solving process that is pervasive to all STEM (Science, Technology, Engineering, and Mathematics) disciplines. It is also suggestive for someone to learn and recognize its applications or to start thinking using CT across other scientific domains (NRC, 2011).

An exemplary attempt has been covered by specific pedagogical principles from *Computing at School* so as to investigate *"what is"* CT and which of its cognitive subjects are really necessary for CS (CAS, 2014). CAS is a task force from the United Kingdom. This report investigates the possible reasons that students do not participate in CS and programming courses for proposing several ways to motivate them further. For CAS, CT is a problem-solving process beyond computing itself. It is regarded as the process of recognizing aspects of computation that requires the use of techniques relevant to computing in order to

understand and rationalize about the appropriateness of natural, social, artificial systems and other processes to solve several real-world problems. CAS supports the opinion that students need to learn thinking strategies to solve logically and methodically problems with computational concepts such as abstraction, algorithm, automation, decomposition, debugging and generalization so as to be applied their strategies with fundamental programming concepts (CAS, 2014).

Beyond the publication of several policy reports in regard to the nature of CT, other reports have also tried to propose learning tasks which can assist students to think logically and methodologically using CT. A notable report is the *"CS Unplugged project"* (Bell et al., 2008) for the creation of low-cost scenarios. It is proposed by Canterbury University. The purpose is the development and implementation of low-cost programs that can attract educators' and scholars' interest and specifically those who struggle to have access to ICT (Information and Communication Technologies) resources. Specifically, the *"CS Unplugged project"* aimed at teaching skills and concepts related to CT using programming constructs (such as combined numbers and writing of algorithms) without having a personal computer (PC), and thus students should try to use pseudocode for solving problems (Bell et al., 2008). CS instructors try to introduce students who basically do not have a background in programming to CT through concepts such as debugging, binary numbers, algorithms, and data compression with board games and puzzles. All tasks are applied through a *"programming-free"* way with a view of giving students the opportunity to think about algorithms which can solve problems without worrying about the syntax details of the source code of any particular programming language (Taub et al., 2012). In their article, Bell et al. (2008) have described activities and competitions by playing with physical objects such as cards, pen, and paper showing students how to think through active and kinesthetic learning tasks like being computer scientists. The *"CS Unplugged project"* is recommended by the ACM K-12 curriculum and has been translated into 12 languages.

From 2009 until today, the *"CS Unplugged project"* has widely gained CS educators' and scholars' attention. Well-established initiatives (Rodriguez et al., 2017; Sentence & Csizmadiam, 2017; Taub et al., 2012) have reported several potentials using *"CS Unplugged project"* tasks with activities to become funny and engaging for novices. The *"CS Unplugged project"* is an encouraging and motivating approach for students to learn how to use CS competencies, computing skills and concepts (such as binary numbers and search algorithms) which can assist the development of skills related to CT (Grover et al., 2015). Nonetheless, other studies have the opposite view. For example, Howland and Good (2015) have referred the difficulties to be demonstrated tasks from the *"CS Unplugged project"* since such tasks demand wider and long-term goals to develop skills related to CT. While such activities are suggested to exposure effectively students in motivating tasks on CS topics, there is not much information on how students' learning performance is measured as a process that requires the correctness of their thinking solution plans to workable programs without applying them into code. The results from Taub et al.'s (2012) study using

"Unplugged" tasks have shown that only some growth in students' perceptions about CS was achieved without providing any further evidence. Many studies have suggested some good kinesthetic practices and tasks related to computing tasks without the use of computing devices, albeit there are appeared several limitations. The use of *"CS Unplugged project"* is really promising for younger students to learning how to think "computationally", but much work needs to be made, as it is observed a lack of studies which can present their findings from qualitative and/or quantitative data extracted by younger students' creations, specifically in regard to their computational understanding to more general concepts of their daily life (Sentence & Csizmadiam, 2017).

In Europe and beyond, the number of projects has received rapid growth, along with increasingly widespread interest in understanding the nature of CT. As described earlier, all the above reports have emphasized the importance that is given regarding students' computational competencies. In this perspective, the focus was in solving different problems, the development of cognitive thinking skills, the representation, and organization of the data, the algorithmic approach for solving its parts, and thus the generalization of solutions based on CT. Based on the above reports, the important role of CT and its contribution to 21st-century skills has emerged as a problem-solving process that can assist students to think before start coding.

2.8. Gender issues

Educators and researchers have already considered gender equality in programming courses as an important topic that cannot start to be investigated only at the university level but from lower one, such as those at the primary and secondary (Howland & Good, 2015; Lockwood et al., 2017). Even though, closing the gender "gap" in CT education for a significant number of policy reports remains a crucial issue (Bocconi et al., 2016). Existing research has been conducted that showed the existence of gender differences in CS with many statements about this topic to be made in Europe and the United States of America (Völkel et al., 2018). Gender issues come across as important since a majority of boys seemed to participate more in CS and programming courses or in a professional field than girls do. Consequently, boys in school contexts perform usually better in CS compared to girls (Kong et al., 2018; Moorman & Johnson, 2003).

A substantial body of research studies has tried to explain gender differences by providing certain stereotypes which tend to create a negative impact specifically on girls' learning performance. Culp (1998) have provided a feminist theoretical perspective, including stereotypical gender roles, differences in outdoor recreation opportunities, family expectations, access, and physical and environmental factors. Lack of broad support is another factor that can be crucial that can support girls further through outdoor programs to prevent such constraints. Also, Graham and Latulipe (2003) have analyzed the stereotypes of CS courses which arising in early in high school. In particular, CS is regarded as a boring subject, devoid of interesting

applications and it is more appropriate to "geeks". The authors have identified two main reasons influencing such a stereotype. The first is that many times girls cannot understand how to use correctly computers in real-world applications that will benefit their daily life. The second is that many girls would like to avoid being "geeks with a monitor tan" stereotype that is not productive and thus influencing negatively their efforts to use for example computers to learn how to program. Another study by Anderson et al. (2008) has pointed out that specifically, high school girls perceive advanced computing subjects as boring and irrelevant, and often express a strong aversion to computers. Therefore, an unmistakable issue is that "gender-neutral" tasks can allow someone to categorize "computationally talented" students with specific gender-biased mostly to be focused on males who were at the risk of hiding other female talents in programming courses (Román-González et al., 2017).

Several policy reports and studies have tended to present the main reasons affecting students' participation in programming courses. In specific, the *"She Figures 2015"* report (2016) has presented some interesting points of view about gender issues. The same report mentioned that across Europe just 21% of CS graduates are few with female gender to stay careers in CS and specifically in programming courses. Such a choice is influenced by females' early experience such as those in school, parental influence and a wider lack of female role models in CS. A second policy report comes from the European Commission (2016). It states the importance of using interactive environments for the introduction of important topics in regard to the integration of CT in programming courses which can influence positively students' engagement and motivation. Nonetheless, the European Commission's (2016) report has also referred that boys' and girls' participation may vary due to their different gender, social background, and age. For this reason, the same report has mentioned alternative ways to introduce students generally in CS and more specifically in programming. For instance, gaming tasks can become noteworthy to both engage students in programming activities and teach them fundamental concepts of CS. According to the European Commission's report, perceptions and practices about learning how to use fundamental programming constructs, concepts, and rules to get high school boys and girls close to computing education can be achieved in two ways. The first is to increase the interest and creativity of students about computing by developing and programming interactive games (or stories) following game-making approaches. The second is to introduce students in programming with computational concepts and skills in simulated problem-solving contexts, following game-playing approaches.

Persistent concerns about the underrepresentation of girls in programming courses, particularly in light of the encouraging elimination of the gender "gap" is still today considered as a problematic situation, even if learning conditions are included inside game-based learning contexts. Specifically, in secondary education, game-based learning approaches need to be aligned with students' preferences and habits taking under confirmation the gender equality. Gender equality is usually referred in order to avoid CS instructors

the discrimination arising from the impression that boys and girls may have about what games or applications are suited more to one than to another (Howland & Good, 2015). Previous studies (Good & Howland, 2016; Werner et al., 2015) have many times presented various reasons preventing girls' participation in programming courses. For instance, Steiner et al. (2009) have noticed that games for learning tasks are considered appropriate and appealing for both boys and girls. In Carter's (2006) study, where the data gathered from students' perceptions, it was found that CS and programming courses were boring more for girls in contrast to boys, who often perceive the subject as an exciting area, largely due to their passion in playing computer games. Lack of girls' interest and participation come from an overall negative attitude towards using computers and lack of confidence with software or interactive environments are the most indicative reasons for this situation (Denner et al., 2012).

In recent years, a renewed interest is appeared in regard to the creation of learning tasks which can guarantee gender equality to be avoided possible gender biases. Gender equality in learning activities can increase to a large extent student's participation by avoiding any possible biases against females pursuing in learning computer programming as resulted by their low interest and performance (Kafai & Burke, 2015; Liu et al., 2013; Mouza et al., 2016). Programming environments are generally intended to ensure learning for all students regardless of gender (Kafai & Burke, 2015). Nonetheless, there are relatively few examples of research which compares the use of games by boys and girls in order to investigate the way of how and what they finally learn in computer programming. For example, special focus on the early years has been given on boys' and girls' engagement and participation by creating interactive and game-based environments (Costa & Miranda, 2016; Kafai & Burke, 2015).

Chapter 3: Instructional approaches and educational environments

The present chapter aims to address a critical overview regarding the use of interactive environments in high school programming courses which are greatly mentioned by the relevant literature. It gives information about the most indicative instructional approaches which have been widely followed in programming during the last twenty years. It also presents a discussion about the advantages and disadvantages on the utilization of interactive environments which are to a large extent followed game-based learning approaches and provides several ways on how such approaches seemed to support CT instruction. More emphasis is given to the analysis of related studies which have utilized LOGO, tangible, and interactive environments, including VPEs and 3D VWs.

3.1. Instructional approaches

Many curricula around the globe have recognized the importance of programming courses in K-12 education. Teaching computer programming assists students to acquire analytical and logical thinking that can lead to the development of methodological skills making them able to solve simple, complex or larger problems (Webb et al., 2017). Usually, students participate in tasks that are applied inside a programming environment in order to start thinking methodologically and logically using algorithmic thinking skills (problem analysis, algorithm design, structured thinking, stringency of expression) for proposing solutions to simulated real-world problems (Lahtinen et al., 2005).

Nowadays, two are the well-identified teaching approaches which are broadly proposed in programming courses from many national school curricula (Lindberg et al., 2018):

a) to learn specific programming languages as a practice-based approach and its main objective which contains the structure, vocabulary, and rules in coding tasks; and
b) to learn how to use programming in problem-solving situations. In such a problem-solving approach, the schedule of students' solution plans is treated as a cognitive activity using their cognitive thinking skills.

Teaching how to use programming in problem-solving situations is the most common-in-use approach in school contexts where students need firstly to develop analytical, logical and methodological thinking skills in order to solve real-world problems which are simulated into programming environments. A "conventional" (traditional) teaching approach is the presentation of fundamental programming concepts and constructs of a programming language such as Python, Java etc.). More emphasis is placed on the syntax of a particular language and the presentation of one or more programming languages in which lessons are supported sequentially, such as for example initial presentation of concepts, variables, and

constructs that students need to know on how to use (Oddie et al., 2010). This instructional approach leads students to think in a "tight context" about the use of a programming language considering that solving a problem using a computer is mainly related to the process of coding correctly constructs and instructions of a programming language to apply a solution (Robins et al., 2003).

However, prior studies (Dagdilelis et al., 2004; Ismail et al., 2010) have shown that traditional instructional methods do not usually facilitate students as novices to learn how to think before starting to code. According to Vahldick et al. (2014), "conventional" instructional learning approaches can often lead students to use in a wrong way their programs as they study differently a broad variety of scientific fields, in which other skills are required. For instance, this is to some extent regular, because in most courses, students need to understand the learning material by simply attending to all courses, by memorizing specific learning material or just by reading a text. In contrast to any other course, programming requires not only cognitive thinking skills but also programming knowledge about the semantics and syntax of a programming language that should be applied to real-world problems (Ismail et al., 2010). As a result, students many times tend to spend less time on the development of problem-solving strategies for applying programming knowledge to solve problems and more on mastering syntax and semantics of a programming language (Koorsse et al., 2015). Therefore, programming seems to become a purely technically-centered process for translating mental representations of problems and solutions into code.

There are many views pointed out that conventional approaches are not quite educationally relevant technology-supported since the main focus inside classrooms is mainly based on the instructions that give a CS teacher, and students are the passive receivers of those instructions. To this notion, a substantial body of recent literature (Dagdilelis et al., 2004; Robins et al., 20003; Xinogalos et al., 2015) has converged on some of the most important problems that novices usually face in programming. These are the following:

a) the general purpose languages have a large number of commands and are quite complex,
b) a strong attention is paid on learning a programming language itself (syntax and/or semantics). This prevents students from developing problem-solving skills and using properly concepts and constructs of a programming language to apply their solution plans into code,
c) the knowledge acquisition cannot fulfill the requirements that students require in order to apply this new knowledge for solving problems when they try to use some of the appropriate programming constructs and concepts executed by a computing device,
d) the most traditional approaches are relatively appropriate only for general-purpose programming in which students need to observe and learn how to apply the execution process, albeit in several times lacking to monitor any result of each command or programming construct, and
e) the specific guidelines to solve small problems do not require learning of a large subset command of programming languages and the development of major programs. Thus, students spend their

time learning how to code when they participate in simple or without purpose tasks without properly understand the use of programming for solving real-world problems.

The weaknesses of "conventional" instructional approaches in programming courses have led CS teachers to look for new methods in order to eliminate the above problems and to improve their teaching processes. For example, Ben-Ari (2001) has noticed that programming learning problems can be addressed by converting introductory courses into a playful and enjoyable process. In this direction, to have funny and enjoyable moments all students can learn computer programming, a variety of teaching approaches have been proposed, using various programming tools and technologies. All these approaches focus on the achievement of learning objectives, either in terms of understanding the concepts/constructs or in terms of designing/developing programs, where students engage and participate in tasks corresponding to problems which are significant for them and/or relevant to their needs and demands. The most important are the following:

a) The *"black box"* instructional approach (Haberman & Kolikant, 2001) familiarizes students with new concepts when conducting activities in computer laboratories in order to participate in all courses. Activities include two parts. At the first, students are asked to run simple programs which they do not know the code and function ("black boxes"), start a "conversation" with a computer, and then answer a series of questions related to "computer dialogue". At the second, students learn how to code and answer questions about the commands/constructs that they have used. Nevertheless, such an approach can lead students to the inefficient process of memorizing and executing continuously of using the same programming constructs or referring to small exercises focused on school textbooks' core aspects for learning computer programming. This process allows students to learn how to apply their code only for a specific number of problem-solving contexts, and thus "know how" to use better a small number of programming concepts (Singh & Ribeiro, 2016).

b) The *"discovery"* instructional approach takes place inside a computer laboratory, in which students in several tasks are initially invited to read small programs, to answer questions about its function to apply constructs and concepts to predict, for example, movements of objects, by integrating "behavior" using programming constructs so that compare and control their responses by running such programs. If their predictions do not correspond to the actual results, the CS instructor can ask students to explain/substantiate their answers reasonably (Baldwin, 1996).

c) The *"pair-programming"* is an instructional approach focused on collaborative learning. Two people work together to design and apply their own programs. One member plays the role of a "driver" and controls the pencil/ mouse/keyboard in the development of the program. The second member is the "observer" who constantly controls the work of the "driver" by asking questions,

exploring alternatives, observing shortcomings, and applying for programs. The CS instructor always sets the learning context for the two roles and ensures that they are kept the roles of the "driver" and the "observer" inside the predefined learning contexts in order to ensure a substantial contribution rising from both roles (Webb & Rosson, 2013).

d) The *"learning-by-doing"* is an instructional approach based on Constructivism. Constructivism as a learning theory has changed the "conventional" way for knowledge acquisition that is not transmitted but it can be built from anyone personally (Papert, 1980). In his work, Papert expands Piaget's ideas on constructivism by promoting the view that learning is more effective when students are activated by building and programming objects that are meaningful to them while enhancing their social interdependence-actions (Kafai & Burke, 2015). Lye and Koh (2014) have argued that both learning approaches following Constructivism can assist each student to build knowledge by interacting with his/her environment that is fully compatible to support with his/her ideas. The same authors have also stated that problem-based learning is a constructivist educational approach can allow any for flexible adaptation of guidance without further explicit guidance.

While the extensive use of several learning approaches is widely proposed in programming courses to trigger students' attention, it is arguable if such approaches alone can satisfy their expectations. Beyond the successful utilization of different learning approaches, such an integration in regular school settings alone cannot automatically lead to its successful use in learning or create a good climate in order to increase not only students' motivation but also their learning performance. In addition, as the ability of users to be processed information is expanding quickly, their thought process is also increasing quickly. To overcome any potential constraints that are identified, a considerable number of previous studies (Costa & Miranda, 2016; Koorsee et al., 2015; Lye & Koh, 2014) have tried to integrate simulated problem-solving tasks with the abovementioned learning approaches as more valuable for students' motivation and participation. CS instructors need to find out alternative ways that may not only engage students to participate in programming courses but also assist them to utilize elements and features from a programming environment in order to gather information so that apply their thinking solution plans. There is a common belief that digital or physical environments are regarded as "platforms" in which are performed most in simulated real-world problem-solving situations and can lead students to view computers as "tools" for problem-solving situations. Students usually await recognition of their efforts through (gaming) practice-based tasks, which is given as feedback and encourage them to continue in even more difficult procedures. Accordingly, educational technologies can become useful tools for the active participation of students following *"learning-by-doing"* approaches in align with the development of cognitive thinking and programming skills that students need to gain rather than "traditional" lectures in which they become passive receivers of CS teachers' instructions (Kafai & Burke, 2015).

3.2. LOGO environments

The LOGO language is regarded as a powerful “tool” for the development of algorithmic thinking and the visualization of algorithms, especially for students in compulsory education (Papert, 1996). The most important feature of a LOGO language that differentiates it from all other programming languages is its orientation as a “tool” for analyzing the processes of students’ thinking before starting to learn how to code. The ability to visualize the execution of a program provided by LOGO can help students to understand the operation of programming and to facilitate a debugging process for applying a program (Papert, 1980).

Teaching programming to younger students can be traced back to 1960 with the LOGO programming language to be firstly written in 1968. LOGO language *‘‘designed to provide a conceptual foundation for teaching mathematical and logical ways of thinking in terms of programming ideas and activities’’* (Feurzeig & Papert, 2011, p. 487) and it was first introduced for teaching mathematics. In his book titled *‘‘Mindstorms: Children, computers and powerful ideas’’*, Papert (1980) has suggested the use of exploratory constructivist instructional guided contexts for teaching LOGO. The LOGO language allows someone to develop new *"words"*, using new commands, which are incorporated into the existing vocabulary of a language known as *“procedures”*. The *“procedures”* are developed by using primitive commands and constructs, helping students to create and/or edit a small number of rules which are considered as logical and geometric conceptual microworlds with elementary visual forms that are projected as simple game-like or game-based exercises. Students learn how to use several fundamental concepts of programming by checking the correctness of programming constructs, which are utilized by integrating behaviors inside objects, like those of a “turtle” or a robot as *Figure 3-1* depicts. They need to develop step-by-step programs, execute each part of their code and track the execution result of each command. The "turtle" LOGO is a ground robot that is programmed and guided to make different spatial movements (Maddux & Rhoda, 1984). The turtle is an “object-to-think-with” that provides the entry point for its movement. It seems like being a geometric shape depending on the position and the direction that each user can program it properly in order to be moved (Papert, 1980).

Figure 3-1: The "turtle" LOGO (Papert, 1980)

During the last twenty years, various programming environments have been developed by using LOGO language. *MicroWorlds* is a version of the Logo programming language and presents a visual-rich multimedia environment. It provides a minimalist graphical environment that allows the student to develop a step-by-step process of programming commands and constructs and software visualization techniques for the execution of those commands and constructs. *MicroWorlds* is based on physical or digital metaphors and concrete actors (objects) that are depicted during a program's execution (Papert, 1980). Students have opportunities to explore a cognitive subject with a view of developing a high-level of cognitive skills that can be transferred to diverse situations (Pardamean & Honni, 2001). Students can create and program their projects which are formed as animations, simulations, or geometric designs. An extension is *MicroWorld EX* that can be connected with Internet webpages and can be integrated with Excel spreadsheets.

Another significant point of view is the features and elements that a *MicroWorld* includes. The user interface design features are the simple, stimulating, and adaptable environment, thus allowing students to develop their own microworlds by controlling and programming each element. *MicroWorlds* is truly regarded as constructivist educational technologies which facilitate student to develop skills related to problem-solving and critical thinking and learning trends which are needed through a process that demands exploration, repetition, programming, and assessment of correctness regarding the appropriate use of fundamental programming and concepts. Recently, in their review study, Xinogalos et al. (2015) have noticed that numerous research papers have previously proposed *MicroWorlds* so as to teach students at a younger age how the use of fundamental programming concepts and constructs, such as sequence, selection,

and iterative. The results from the same review revealed the positive acceptance of *MicroWorld* as an instructional approach with improved learning outcomes and achievements.

The above instructional approaches have been generally provided in programming courses so that students can achieve the following learning objectives (Lye & Koh, 2014; Papert, 1980):

a) design problem-solving activities and organize them in smaller and simpler components before start coding,
b) experiment with commands and constructs in order to gain confidence in programming,
c) create programs to apply programming constructs and concepts in the right order,
d) evaluate programs to assess the correctness of its proper function,
e) debug and correct errors in order to (re-)construct their proposed programs, and
f) develop applications with scenarios that can be integrated into simulated contexts.

3.3. Contemporary educational environments

Problem-solving tasks in programming courses require someone to use his/her cognitive thinking process in order to develop a specific strategy and solve properly each of its tasks. This comes in contrast to what happens with cognitive activities that require knowledge or individual skills acquired within repetitive practical training (e.g., reading or listening skills). When students learn how to program with some language such as using LOGO-like environments to accomplish a goal, they need to get an object that is usually the main *"object-to-think-with"* in order to program its behavior and predict its movements within specific spatial contexts. For example, through a maze, what matters, beyond from the end result or the correct use of programming constructs itself, is the experience (Grover & Pea, 2013). Such an experience leads to the development of the required problem-solving strategies, idea design, and correctness by testing and diagnosing errors of code to solve a problem. This may increase students' confidence in their own judgment, improve their self-efficacy and provide efficiently their anticipated outcomes (Koorsee et al., 2015).

The rapid growth of digital products in the global market has made companies move a step forward to fill the demand for educational content into programming courses. Moreover, creating and programming such projects/products offer features for assessing students' progress, thus facilitating CS instructors efficiently organize a learning environment (Tuomi et al., 2017). Many of these products have "ready-made" tools for teachers or students to utilize and develop their own creations for content production. Specifically, novices have to learn how to give rigorous and well-structured solutions to problems in case of applying these solution plans into workable plans and algorithms. Nonetheless, in many cases, they learn wrongly the commands of a programming language with names alongside their appropriate use that is confusing and usually not easy to remember. For instance, a programming language with very strict syntax

can often cause shortcomings on what finally programming is and under which circumstances can be used in problem-solving situations (Webb et al., 2017). To address such problems, programming environments are focused on the design, development, and implementation of programming languages to specific algorithmic problems that are suitable for educational purposes. Thence, the most noticeable characteristics of educational programming environments which can support specific requirements and provide learning contexts are those which can assist students (Kafai & Burke, 2015; Lye & Koh, 2014):

- to explore the programming environment by interacting with it, and then by utilizing tools which can provide tasks inside it with a variety of immediacy features.
- to support algorithmic thinking and programming of specific programming constructs to build programs with a small number of concepts having a simple syntax and semantics.
- to develop visualization elements and features, making it easier for users to track dynamic, hidden, and internal processes that take place when running a program.

A significant number of educational environments and platforms has been developed in order to facilitate students' engagement and participation in programming courses. These environments allow students to understand the interaction of humans with computers by programming elements and objects that exist inside them. Programming environments mainly for those who do not have a strong background in programming (called "novices") are relatively easy to use and allow early experiences to focus on designing and creating solution plans so that solve problems than on mastering syntax of programming languages. Due to the different user interface design features and elements, three are the major categories that must be referred. The first category is tangible environments which include embedded code cubes blocks with electronic devices or power supplies and those which do not need electronic power such as wooden programming blocks.

The second includes educational robotics. The term "robot" is used quite broadly and may include articulated robots, mobile robots or autonomous vehicles of any scale. Usually, students learn how to program a robot, understand its interface and units (sensors, educational or industrial robotic machines) in order to maintain it. In addition, the educational robots come with simulation software, which enables students to practice both with a virtual robot and its simulated environment. The most well-known are *Lego Mindstorm NXT* and *Lego WeDo robotics*.

The third category entails interactive (digital/graphical) environments which have computer-supported media interface responding to users' actions and allowing them to communicate with a computing device so that create various simulated applications/tasks. Various forms of interactions are included such as video, animations, and simulations. Users have various elements and features to create something meaningful in a training interface that encompasses specific mechanisms that are easily manipulated and controlled using a keyboard and a mouse. Users have opportunities to manipulate and

program visual representations inside a digital environment displayed in two-dimensions (2D) or in three-dimensions (3D), in which can be achieved certain learning goals based on the exploration, analysis, and operation of programming tasks for simulated problem-solving situations. Two are the most distinctive platforms that interactive environments can be separated. The first includes visual programming environments such as Scratch, Alice, and AgentSheets. The second contains 3D virtual worlds such as OpenSimulator and Second Life. 3D VWs may not be created for educational purposes per se; however, such platforms have the potential to be regarded as candidate for various disciplines and domains including those of CS and programming.

3.3.1. Tangible programming

A relatively recent approach to facilitate students to learn how to code is tangible programming. Tangible programming environments have user interfaces in which users can interact with digital information through a physical environment. Tangible interfaces can reduce the cognitive load needed for someone to learn how a system works so as to not pay so much attention to learning how to program itself (Marshall, 2007). More specifically, tangible programming is a form of language that does not necessarily require from someone to use a keyboard, mouse or computer, but the use and layout of physical objects, such as cubes and puzzles (Smith, 2007). Tangible programming makes programming an activity that is accessible to the hands and minds of students by making it more direct and less abstract. By combining computer programming and interaction, tangible programming allows students to manipulate physical code blocks directly, which makes learning and teaching programming more appealing (Sapounidis & Dimitriadis, 2013).

Two are the main categories of tangible programming. The first includes tangible programming blocks which are inexpensive and durable cubes with no embedded electronics or power supplies. An apparent paradigm is the use of familiar objects (wooden cubes) to transform an unfamiliar and potentially intimidating activity like computer programming into an enjoyable and playful experience. For example, *Tern* is a tangible programming language for middle school and late elementary school students. *Figure 3-2* below depicts *Tern* that is consisted of wooden blocks shaped like jigsaw puzzle pieces. Students can connect wooden blocks to form physical computer programs, which include action commands, loops, branches, and subroutines (Horn et al., 2007).

Figure 3-2: A collection of wooden tangible programming blocks using Tern (Horn et al., 2007)

The second category encompasses tangible cube blocks which integrate embedded electronic devices or power supplies. A suggestive paradigm is *AlgoBlock* (Suzuki & Kato, 1993). It is tangible programming that includes a collection of physical cubes that can be linked together to form a program using electronic supplies (*Figure 3-3*). These cubes are then linked in a way that a computer can run each program that is created since each one corresponds similarly to the LOGO commands (Sapounidis & Dimitriadis, 2013).

Figure 3-3: A collection of natural tangible programming blocks with electronic supplies using AlgoBlock (Suzuki & Kato, 1993)

3.3.1.1. Advantages and disadvantages

Most tangible systems have been designed and proposed for children in order to connect activities with the physical world. Tangible programming has a number of advantages. First, the attractiveness of the natural interface shows a trend toward a physical interface that is haptic. This feature may allow the use of tangible exploratory activities, in which users can gain greater experimental knowledge through instructive-guided approaches. A set of programming constructs is provided using cube codes having natural user interfaces that require kinesthetic interaction with those cubes which can be enriched in natural spaces (Sapounidis & Dimitriadis, 2013).

Second, tangible interfaces provide richer learning experiences so as to increase reflection and understanding in regard to students' actions in specific spatial contexts. For example, tangible programming blocks can be combined with material properties such as size, weight, texture, and temperature in order to help students to learn how to use programming knowledge to other areas such as physics, mathematics or chemistry (Marshall, 2007).

Third, the innovative tasks that can be achieved using tangible interfaces create a real-world programming environment in which everyday objects are converted into both input and output devices at the same time and can display any information. Appropriate representations on the interface may be proved as useful to reduce the complexity of problems and provide an easier way to decompose a program (Schneider et al., 2011).

Fourth, the tangible interfaces can support collaboration among students (face-to-face). In a collaborative learning activity using tangible programming blocks, students can increase their visibility in the work of other peers, and they can easily exchange ideas or opinions about their solution plans. Students can also watch kinesthetic gestures (e.g. hands, eyes) of other peers, thus achieving a richer collaboration within specific spatial school contexts (Suzuki & Kato, 1993).

However, tangible interfaces have also a number of disadvantages. The main reason for these disadvantages appears to be lack of systems with different features, the high cost, and construction of such systems that hosted only in research centers or in a small amount in school laboratories (Horn & Jacob, 2006). In spite of various studies that have proposed several tangible systems, there is a lack of tangible programming tools, and thus the international literature has referred several restrictions. The most indicative are the following (Suzuki & Kato, 1993; Xie et al., 2008; Zuckerman et al., 2005):

- several tangible systems do not have a sufficient number of commands and parameters that may restrict students' learning on how to use programming constructs sufficiently.
- the lack of real-time control prevents the smooth interaction between the programmer and the program itself.

- some tangible systems are not easy to move or moving its units can cause unusable learning conditions in actual school classrooms.
- some physical properties, such as shape or temperature can provide advantages to tangible systems programming; however, such properties have not yet been investigated.
- the storage and reuse of tangible code blocks are not supported in any system.

A brief summary of the advantages and disadvantages is presented in *Table 3-1* below:

Table 3-1: Advantages and disadvantages of tangible programming

Tangible programming	
Advantages	+ Attractiveness of natural interfaces + Richer learning experiences so as to increase students' reflection in specific spatial contexts + Creation of real-world programming environments in which everyday objects are utilized as input and output devices + Student collaboration
Disadvantages	- Lack of systems with different features, high cost, and construction of toolkits that hosted only in research centers or in a small amount in school laboratories - Lack of a sufficient number of commands and parameters - Lack of real-time control prevents the smooth interaction between the programmer and the program itself - School or laboratory conditions sometimes prevent the movement of units of a tangible programming environment - Lack of objects' manipulation supporting only the use of specific conditions and concepts in programming tasks

3.3.2. Educational robotics in programming

Educational school contexts have today provided new instructional approaches which can rely on innovative actions and demands of students using educational tools. Educational robotics is a rapidly expanding industry at all levels of education worldwide that can be used in different STEM concepts. The use of robotics in programming courses is an innovative learning approach. It combines elements of basic sciences (physics, engineering), new information technologies (software development, artificial intelligence) and the study of the interaction between humans and robots. Robotics are widely used for observation, analysis, modeling, and control of various physical processes (Miglino et al., 1999).

Educational robotics is a broad term that refers to a collection of activities in specific instructional programs with educational resources having physical robot models. Such an instructional program generally includes the following:

a) the physical/natural section that includes objects made from simpler units (e.g. cubes, bricks) for processing of information, with an additional connectivity, suitable motors and sensors in order to learn someone how to program, and

b) the graphical section in which are included a physical object (robot) can be programmed and its inputs for the information transmitted by the sensors (e.g. ambient or ambient sound information) light and drives out motors to give motion-behavior.

Within such contexts, students are engaged in tasks which require the design and construction of robots to involve actively them in learning programming in order to develop skills related to problem-solving, logical reasoning, and tasks to support collaborative learning tasks for the following two reasons (Afari & Khine, 2017; Detsikas & Alimisis, 2011):

a) to gain knowledge regarding the use of robots that contain specific units and toolkits for learning how to use fundamental programming concepts and constructs in a physical environment for experimentation, and
b) to develop logical and critical thinking in collaborative, innovative and project-based learning tasks for the active participation of students.

With the creation of integrated robotics packages in combination with suitable programming environments, the integration of robotics into schools has gained much attention (Klassner & Anderson, 2003). Two relevant technologies that have been designed to assist students' participation in programming courses. These are *Lego Mindstorms* and *Pico-Crickets kits* created by the MIT's Media Lab (Resnick et al, 1996). In addition, Carnegie Mellon University and Lego worked together to design educational tools that promote mathematical and programming skills. Today, a lot of high and primary schools use Mindstorms and other robots, beyond NXT, are essential to introduce control concepts. Lego Mindstorms are designed for activities that require the completion of a project with the goal of solving a problem (Klassner & Anderson, 2003). The main Lego Mindstorms educational systems are:

1. *WeDo*: It is an educational robot with a complete set of instructions and kits that allows students to design, construct simple models on their computer, download the program on their model, and confirm its operation using a robot. *LEGO WeDo* offers a simpler robotic kit than *LEGO Mindstorms* (*Figure 3-4*), it is less costly and cannot produce an autonomous robot since the robot's functions required to be attached to a computer with a USB cable. This kit is being produced since 2008 and utilized mostly from primary schools (Kabátová et al., 2012).

Figure 3-4: Components of a robotic Bee-Bot (Kabátová et al., 2012)

2. *NXT*: It is an educational robot for learning the basic principles of programming for young students aged 8 years and over (*Figure 3-5*). It combines the basic principles of robotics with colorful blocks and programming principles and they all form a fun educational process. Its software has a drag and drops physical interface and a graphical programming environment making any application accessible to all (novices and experts) programmers (Kim & Jeon, 2007).

Figure 3-5: A LEGO Mindstorms programming environment (Kim & Jeon, 2007)

One of the most distinctive functions is the compass sensor. It presents an additional sensor for Lego Mindstorms NXT construction set. The digital compass operates with 1° azimuth accuracy, representing values from 0° to 359°, which enables its own definition of the four cardinal points for a room to any direction. The color sensor. In other words, this is an optical sensor making the color detection of the scan surface much easier. The sensor is able to distinguish six colors (red, blue, green, yellow, red and white) marking them with numbers or selected color range. The Ultrasonic sensor is based on the sonar principle and serves for distance measurement in 0-250 cm or 0-100 inches range with ± 3 cm accuracy. The accuracy is influenced by the size, surface, material, and the shape of the object which reflects the wave motion back to the sensor (Bickford, 2011).

3. *EV3*: It is an educational robot and it contains a package with specific robotic kits that is proposed for classroom use. It allows students to build, plan and test their own solutions to real problems with robotic technology. It includes the EV3 Intelligent Brick, which is a small computer that enables users to control

the motor and collect data from sensors (*Figure 3-6*). Bluetooth and Wi-Fi communication for data collection and schedules with specific instructions about the robot's movements are also provided. This type of robot is used to collect, view, analyze and manage data from sensors and observe data in interactive graphs. Students are encouraged to think so as to express creative solutions to problems, and then apply to observe the consequences of those instructions for the robot's movements (Chatty, 2015).

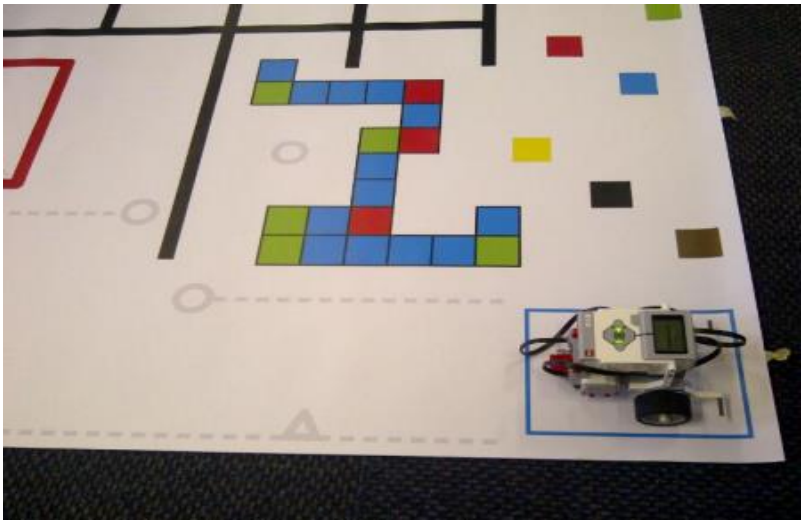


Figure 3-6: An EV3 Lego Mindstorms robot (Chetty, 2015)

3.3.2.1. Advantages and disadvantages

There is a common belief among educators and CS teachers' circles that the use of computer-controlled models is becoming important "tools" for teaching programming (Barnes, 2002). Engaging students with "smart" robotic constructions, such as Lego Mindstorms, which are similar LOGO (*"object-to-think-with"*) turtle can change the way that students think and learn before starting to code. Therefore, a learning approach using educational robotics have advantages.

First, it offers students the opportunity to understand programming methods, but also to acquire knowledge through practice-based tasks as being real scientists (Kabátová et al., 2012).

Second, students within these contexts observe, assume, apply, and verify several hypotheses based on programming constructs and concepts that utilize robots in specific spatial contexts (Klassner & Anderson, 2003).

Third, it provides an easy way to debug code. Natural models of robots offer direct feedback to students about the effectiveness of their programs. This may possibly assist students gradually to give more effective and precise instructions based on their solution plans (Chetty, 2015).

Despite the above advantages, several are also the disadvantages arising from the use of educational robotics. These are the following (Hamrick & Hensel, 2013; Kabátová et al., 2012; Kantor et al., 1996):

- the localization issues since it is sometimes observed uncertainty in sensing and actuation that impose several difficulties to provide the robot's pose accurately.
- the physical limitations of toolkits and units which are utilized. The movements of a physical robot model are not always accurate. This inaccuracy is caused due both to the environment, in which a physical model (robot) is controlled and operated, in addition to the difficulty in programming correctly its right movements on it. For example, two engines that control two different wheels are unlikely to produce exactly the same result, so the model deviates to a spatial context. Even a reasonably correct program may not bring the desired results due to external factors such as friction.
- the time constraints that really exist. Loading the program to the processor includes a process of "translation - load – execution" of the program that is more time-consuming than a digital simulator.
- the cost of robots or units and kits are sometimes high enough for educational sectors and schools, in contrary to other digital environments which are free of charge.

A brief summary of the advantages and disadvantages of educational robotics is presented in *Table 3-2* below:

Table 3-2: Advantages and disadvantages of educational robotics

Educational robotics	
Advantages	+ Attractiveness of a natural robotic interface + Rich learning experiences so as to increase students' reflection through practice-based tasks + A variety of learning outcomes based on the observation, assumption, and verification of several hypotheses which can be applied using programming constructs and concepts
Disadvantages	- Localization issues cause uncertainty and difficulty to provide the robot's pose accurately in specific spatial contexts - Physical limitations of toolkits and units cause an inaccuracy to control a robot's movements for a long period of time - Loading the program to the processor includes a process of "translation - load – execution" that is more time-consuming

3.3.3. Visual programming environments

Visual programming environments (VPEs) are interactive environments that provide visual elements and objects which can be manipulated and programmed with a limited set of simple or nested programming constructs and commands coupled with metaphors to aid to a problem description (Maloney et al., 2008). Additionally, VPEs attempt to introduce users to object-oriented programming by simulating actual computer-supported tasks. Users start becoming software designers and they start learning how to program by providing a visual overview of their progress inside "window-based" digital-oriented environments (Resnick et al., 2009).

With the rapid and extensive proliferation of VPEs, programmers (novices and experts) seemed to have a positive picture in regard to their first introduction to computation and programming. VPEs are the most worthwhile options for computer programming because of the user-friendly graphical design interface various features, elements and a visual palette that contains colored code blocks to provide programming constructs, easily accessible for beginners and advanced developers, or even for CS instructors without the appropriate background in programming (Maloney et al., 2008). 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 formed to the creation of interactive animations, games, or artistic expressions. By using a visual palette in which are available fundamental programming constructs, users can construct scripts by dragging-and-dropping the language blocks. This palette provides visual feedback showing the execution of scripts for users to comprehend how they work (Koorsse et al., 2015). Colored code blocks in a visual palette are resembled as jigsaw puzzle pieces with specific logical instructions (control flow blocks nesting) to avoid syntax errors (Chao, 2016; Werner et al., 2015). Users try to understand how to use programming solutions by integrating behavior to predict movements or program expressions for tracking characters or objects in a visual and/or animated environment. Such a process can support their understanding of programming knowledge and assist them to develop and use programming skills (Garneli et al., 2015). To this notion, users are focused on a problem-solving process than in syntax complexity and propose solutions as design patterns.

Notable results from past efforts (Mouza et al., 2016; Repenning et al., 2015) have advocated that visualization of programming constructs can support students' understanding on abstract concepts and make programming courses more interesting and applicable. VPEs are widely being utilized in programming for the following two reasons. First, programmers and specifical novices can develop and code using colored blocks a program using a visual palette. Such a process gives feedback to users so as to understand and correct (debug) optically and/or acoustically errors into code. Second, users can develop interactive games or stories that support their self-study understanding on how to use programming constructs and commands properly (Myers et al., 2004). Therefore, the most noteworthy features of VPEs are the applicability and visualization of algorithmic control flow (code tracing) can provide more insights into the behavioral patterns and design strategies of code blocks exhibited by programmers.

Although the manipulation can be successfully achieved by using visual elements from a menu in which users can configure or construct a program to develop an executable solution through code blocks from a visual palette, logical errors may still exist (Brennan & Resnick, 2012; Denner et al., 2012). While various studies (Repenning et al., 2015; Werner et al., 2015) have reported the increased satisfaction and motivation of students in learning how to program, other studies have mentioned that the results from the use of specific constructs which may not differentiate from their own previous or other peers projects can

cause misunderstandings about the appropriate use of VPEs for programming courses (Grover et al., 2013; Koorsse et al., 2015). In this perspective, it may be imperative to mention that a VPE to become successful in its use and assist users to learn how to program requires supplementary explanations/instructions from CS teachers (Webb & Rosson, 2013).

The use of VPEs has today shown considerable promise in languages which aim to give specifically novice programmers a good first introduction in computing literacy and mainly in coding. Also, a significant number of VPEs have been proposed for programming courses. During the last decade, literature reviews in this educational field (Lye & Koh, 2014; Vhaldick et al., 2014) have proposed VPEs for programming courses, such as Scratch, Alice, Kodu, and Greenfoot and Web-based simulation authoring tools such as Agentsheets and Agentcubes. Nevertheless, due to the on-growing number of VPEs, it is imperative to refer only those which have been mostly utilized in the majority of research studies, have similarly user interface design features and furthermore are acceptable (or well-documented) by many curricula around the globe.

Scratch

The first and most well-known VPE is Scratch[1]. It is a VPE developed to allow programmers to manipulate and program visual elements in a window-based "stage" to create different interactive tasks, media sources and stories (Maloney et al., 2008), with a primary audience to be between the ages of 8 to 16 years old. Scratch is a visual programming language designed by the MIT Media Lab and released in 2005. The user interface design features and elements of Scratch include a visual palette, on the left side, with different colored blocks with programming constructs on the right side to a "window-based" stage that can be programmed into different sprites (Resnick et al., 2009). By using a visual palette, users can drag and drop graphical blocks in order to compose simple or nesting code blocks with variables and/or to create more complicated programming constructs in favor of developing and programming at the beginning interactive games or storytelling. Code blocks are designed in order to be combined together so that assist users to create programs with logical reasoning and the code's shape to be considered as appropriate for the good operation of these programs. For instance, an *"If...else"* block will fit with a set of commands and cause without unlimited execution of these commands. For this reason, Scratch's visual palette with code building blocks has widely been recognized as really useful "tool" to the initial introduction of students to programming (Maloney et al., 2010). Nonetheless, it allows the creation of more complex programs by embedding code blocks from a visual palette in a digital environment that includes "sprites" (i.e. iconic

[1] http://scratch.mit.edu

characters) on a “stage” (scene background) with built-in graphics creation and sound editing capabilities (*Figure 3-7*).

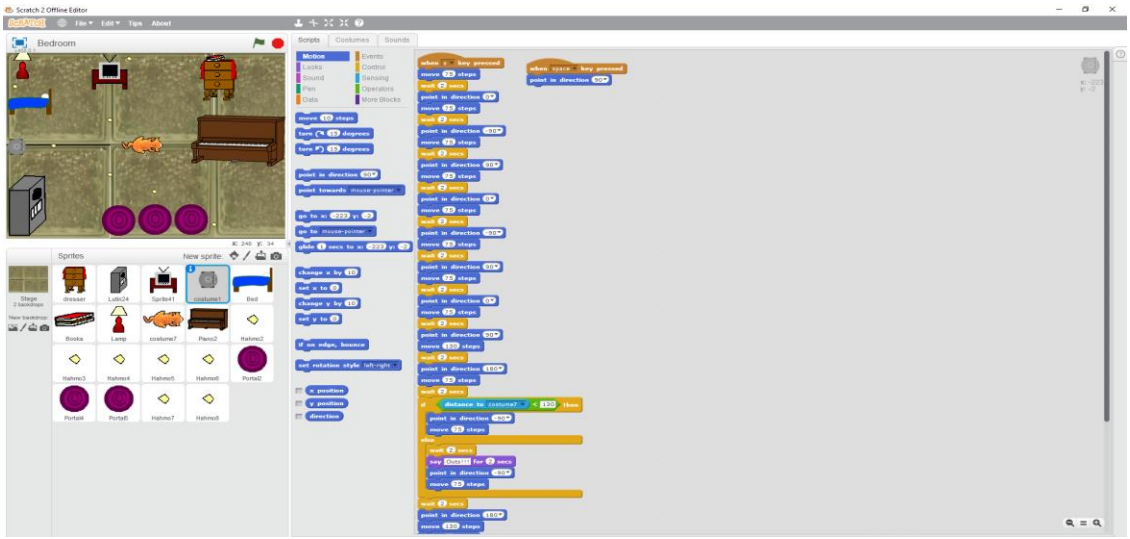


Figure 3-7: A screenshot of a game created in Scratch

Scratch is currently the most popular VPE and it is used worldwide by many high schools. It helps users learn to think creatively, synthesize logical steps of an algorithm stacked on a palette with colorful jigsaw pieces, and/or collaborate for designing their own interactive stories, games, and cartoons, raising from common standards of previous creations (Maloney et al., 2008). In specific, Scratch supports programmers, and especially novices to create animated stories, multimedia presentations, games, simulations and other interactive projects (Xinogalos et al., 2015). Such creations and works can then be shared in an online community that has more than 27 million registered users and their projects[2] accessible free to other users.

AgentCubes

The second most well-known VPE is AgentCubes[3]. It is a VPE that allows users to create their own games and agent-based simulations and upload their creations on the Web through a user-friendly interface following a drag and drop process (Repenning et al., 2010). It is an end-user game making and simulation prototyping tool for building a domain-oriented dynamic and visual environment that can help users to create 3D games or simulations (*Figure 3-8*).

[2] See Scratch’s statistics were retrieved 4/12/2017 from https://scratch.mit.edu/statistics/.
[3] https://www.agentcubesonline.com

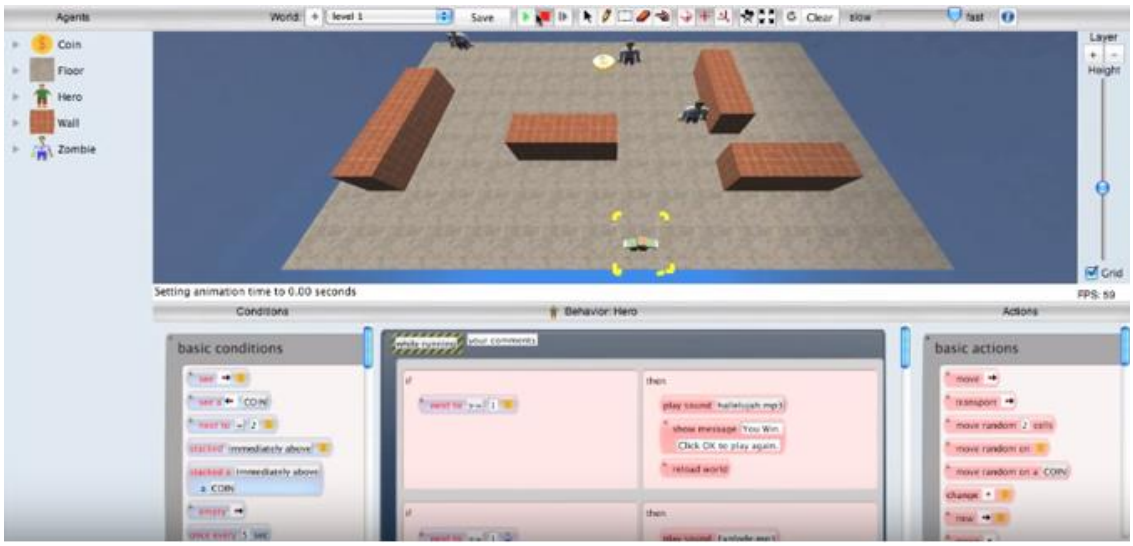


Figure 3-8: A screenshot of a game created in Agentcubes[4]

Interactive agent-based simulations can help students to understand new ideas, test theories, and explore complex processes in various scientific fields. Each agent contains a depiction of how it looks, and what behaviors covered by a set of rules that dictate its action when the game is running based on a variety of communication modalities such as animation, sound, and speech. Using AgentSheets, students can develop and create games based on the concepts of information technology, logic, and algorithmic thinking (Repenning et al., 2015). The simulation toolset includes the following parts (Repenning & Ioannidou, 2006):

a) the gallery where all agents and their shapes are presented,
b) the digital world where the simulation or game unfolds,
c) an inflatable icons editor for the creation of 3D objects,
d) the rule-based agent behaviors that are defined using a 3D visual agent in which each user can integrate specific conditions, and
e) the actions.

Alice

The third well-referred VPE is Alice[5]. It is a programming environment that is designed to assist the student to learn how to program through the construction of 3D visual objects. It is recognized as the most well-known VPE for storytelling and 3D animations (Cooper et al., 2003). Alice (or Alice 3, a newer version) is a digital programming environment in which scripts are composed by code blocks with snippets

[4] *Figure 3-8* was retrieved 23 May 2017 from http://www.agentsheets.com/agentcubes/index.html
[5] http://www.alice.org

of pseudocode. Alice supports object-oriented programming since it provides a visual palette with code blocks, in which users can transfer to a window-based *stage* their program tiles to a word processor and customize them if it is necessary (Werner et al., 2015). Once a program has been built, it runs as a 3D animation which allows users to quickly see if the program is correctly executed with the desired behavior or not (*Figure 3-9*).

Figure 3-9: A screenshot of a game created in Alice[6]

Alice also gives a very tight loop of visual feedback since it is very clear the way that all characters in the environment behaved (or not) according to the program that is integrated and produced as animation. It can be used by students from high school (usually 11 years old and older) to the university level, as it can support the development of logically meaningful programs by developing and programming storytelling expressions. It also gives to novices a list of predefined events in a digital world, the lower right-hand window, the core processor and, in the small middle-sized 3D window-based digital environment, which hosts storytelling tasks if the code is executed correctly (Kelleher et al., 2007).

3.3.3.1. Advantages and disadvantages

The utilization of VPEs still today remains as an alternative and worthwhile option for learning computer programming because of the easy to use graphical user interface (GUI) design features and elements alongside with a visual palette that includes colored code blocks (Maloney et al., 2008). Interactive environments provide many visualization techniques, such as the integration of behavior where someone can copy and paste code blocks into visual elements without worrying about the code syntax as with the use of general-purpose languages (Vahldick et al., 2014). This makes such programming environments

[6] *Figure 3-9* was retrieved 23 May 2017 from https://www.alice.org/wp-content/uploads/2017/04/Scene-Editor-Overview.pdf

accessible to all programmers or even to those CS instructors who have not got an advanced background in programming. VPEs such as Alice (Dann et al., 2000), Agentsheets/AgentCubes (Repenning et al., 2010), Scratch (Resnick et al., 2009) are widely utilized to be formed and created with fully fledged integrated development of visually-rich contexts, thus providing digital contexts for the development of interactive games, digital artifacts, animations and storytelling expressions.

VPEs offer to users the opportunity to practice and to understand fundamental programming constructs and concepts, with some of the most distinguished characteristics and features that provide various advantages.

First, problem-solving and strategy analysis with code's execution indicate that users should not create solutions depending on its operability when planning a solution in an effort to transform their knowledge from an algorithm described from a natural language into code. VPEs assist users to apply a program solution using a visual palette with code blocks. All programming constructs and commands are described as small phrases of codes and commands can be selected and executed as a program (Koorsse et al., 2015).

Second, the execution of code blocks can assist users to learn how to trace a program and evaluate the consequent results of the chosen constructs and commands variables at different points when are integrated into digital elements in a window-based *stage* (Repenning et al., 2010).

Third, code comprehension can improve the abilities of reading and understanding code blocks adequately with a purpose to find and fix errors (debugging). Such a process facilitates especially novices to understand common algorithms or programming constructs in order to adapt and utilize different problems, i.e. "know how" to solve similar problems (Dann et al., 2000). The error messages can allow users to fix code errors and debug their program by giving visual feedback if the code blocks are not resembled correctly without having a logical and operational order (Brennan & Resnick, 2012).

However, a wide range of previous studies has also mentioned some disadvantages. These are the following (Brennan & Resnick, 2012; Howland & Good, 2015; Repenning et al., 2010):

- the use of visual languages are a good starting point especially for novices to learn how to use fundamental programming constructs and to engage in coding tasks. Nevertheless, user interface design features and elements of visual programming environments have not been designed to encourage the development of a more general understanding in regard to even more complex computational problems.
- the user interface design features and elements are easy to use, but students alone tend to create and program interactive games or artifacts that are simple and without purpose.

- the users require not only the instructor's support but also features and tools which can assist them not only understand the visualized problem and its subparts but also to create workable plans and programs in an effort that leads from a problem's description to its solution.
- the iterative use of creating artifacts and projects using specific programming constructs or design patterns (code blocks) which can be found on the Internet in order to be used for the same or similar projects. As a result, they may be merely played or create artifacts, games and projects only by using code blocks in the *trial-and-error* process rather than thinking before practicing and assessing the correctness of their thinking solution plans into code.

While results from previous studies have shown a general improvement on students' engagement and participation, the use of visual languages as coding assistance tools, specifically for creating games, artistic expressions or animations, does not alone improve students' learning performance in long-term use (Grover et al., 2015; Koorsee et al., 2015). Scholars may not have the appropriate experience to know all potentials and capabilities of VPEs so as to provide as many as possible different learning tasks which may have an impact on the effectiveness of Scratch to assist students' understanding about the use of programming concepts to a larger extent (Vahldick et al., 2014). For all those reasons, it is really arguable if students alone can fully understand the cognitive aspects of using coding in several tasks without guidance and have a programming experience that may reflect on their overall learning performance (Howland & Good, 2015).

A brief summary of the advantages and disadvantages is presented in *Table 3-3* below:

Table 3-3: Advantages and disadvantages of visual programming

Visual programming	
Advantages	+ Problem-solving and strategy analysis with code's execution is easier using a visual palette with colored code blocks + Execution of code blocks without worrying about code syntax issues + Easy code blocks comprehension and organization + Debug with visual feedback if all code blocks are (or not) in a logical order
Disadvantages	- Easy coding tasks are usually focused more on novices' creations which are sometimes are simple or without purpose - Lack of features and tools that may assist further a visualized problem and its parts - Use of specific programming constructs or design patterns (combination of code blocks) similar to those that can be found on the Internet or from previous similar projects

3.3.4. Three-dimensional virtual worlds

The use of 3D virtual worlds (or 3D VWs) is increasingly becoming a potential task of the modern global culture and in fact, there is a common conviction that is provided as another social phenomenon (Schroeder, 2008). A 3D VW is a computer-based simulated environment that is accessible by many users who can create a personal avatar (digital figure which are alike as a humans' representation) so as to interact and explore its features using various visual objects, participate in activities, and communicate with other

(or not) peers (Bell, 2008; Girvan, 2018). It can also offer a sense of realistic representation of problem-solving situations due to the high representational fidelity that some in-world objects have. Users can provide solutions in simulated problems that resembled as those of a real-world and track their errors optically or acoustically in a specific grid (spatialized) in order to understand better the consequences of their actions during the execution time (Esteves et al., 2011).

3D VWs can assist users to participate in various learning disciplines/domains due to their inherent features that make such platforms to differentiate from others. These are the following:

- The *sense of (co-)presence* that most users can "feel" when they are immersed in a virtual grid allows their co-existence in a common virtual environment to (re-) construct metaphorical representations (metaphors) with other avatars to exchange and/or apply their ideas without spatial-temporal physical (or digital) constraints (Dalgarno & Lee, 2010).
- The different types of *communication* using verbal (VoIP) calls or non-verbal channels (e.g. gestures or facial expressions that compose each user's emotional state, IM and chat text). These tools can facilitate interaction among users in a common virtual environment. It is important to be referred that communication is spatialized available only in the specific grid where avatars are online permitting them to communicate freely with others to understand the effects of a learning situation in a collaborative climate (Esteves et al., 2011).
- The *embodiment representations* of users as cyber entities (avatars) can allow efficient interaction with other peers in a common 3D virtual environment (grid). At the same time, users can also use representational functions or artifacts with high fidelity (Okutsu et al., 2013).
- The *expressiveness* of animated and interactive 3D graphical representations of users (avatars) or virtual places (grids) can be used for the presentation of interactive concepts that are difficult to comprehend in digital or textual forms. With virtual metaphors, users are able to construct meaningful artifacts or projects within a persistent 3D environment. Such an environment continues to exist and develop even if no avatar to interact with it (Girvan et al., 2018).
- The *real-time simulation using interactive* visual objects and their combinations that are created as artifacts for the implementation of different learning scenarios. Students can create and use their own tools or artifacts in experiential and problem-solving learning activities. The creation of 3D simulations and microworlds can enhance knowledge representation of the explored domain (Rico et al., 2011).

Beyond the above, two remarkable features of 3D VWs which differentiate them from other interactive environments are as follows. The first is navigation and it is achieved using two types of navigation techniques: joystick-based input devices and steering metaphors based on movements of the user's body as an avatar in order to walk, speak and fly inside a 3D VW. A second, equally important for

enhancing interactivity, is the opportunity that a user has to manipulate the visual objects and integrate behaviors to those all those objects by simply clicking on them, moving them or rotating them to one, two or three dimensions (Rico et al., 2011). In other cases, users are simultaneously connected to a *"world"* in which they can communicate via chat text or voice call. Nonetheless, contrary to other social media platforms, conversation and chatting in 3D VWs is only spatialized. Beyond the projection of a visually-rich environment with content and objects that mimic those of a real-world, 3D VWs give the possibility of viewing and exploring even abstract or hypothetical constructs by taking advantage of intuitive, natural modality contexts for user-interaction tasks. Users can observe and explore in an intuitive way even data that do not come from the real world. Visual elements and objects for scientific purposes are presented in a 3D window-based virtual environment with user interface features and elements, constructs or processes to be visualized using the appropriate metaphors, complex systems or processes in simpler forms, and/or even in a hypothetical version (Dalgarno & Lee, 2010).

Instructional designers, scholars, and educators need to consider the utilization of the inherent technological capabilities of 3D VWs as important for creating and programming learning platforms for different subjects through (in-) formal instructional contexts. 3D VWs can be utilized with respect of facilitating practice-based learning exercises which can lead to the development of enhanced spatial knowledge representation of an explored domain, because they are well suited to such physical simulations permitting the full physical behavior of objects that are modeled, without restriction (Howland & Good, 2015; Robertson, 2012). In their review, Dalgarno and Lee (2010) have already noticed several educational affordances using 3D VWs, such as the formation of spatial knowledge representations that support learning tasks, greater opportunities for experiential learning, increased motivation/engagement, and improved contextualization of learning.

Categorization of three-dimensional virtual worlds

A great number of different 3D VWs have been developed and utilized not only for the socialization or collaboration among users who are spatially separated (or not) across the globe but also for educational purposes in different learning subjects and/or domains. Two are the categories of 3D VWs that can be separated according to their technological capabilities, characteristics, and features. The first category is social virtual worlds (SVWs). In SVWs, users can co-exist in multiple 3D persistent environments without having specific purposes, but with an easy production of progressive or interactive storytelling expressions and imaginary (or not) game-based environments or on the part of upgrading their avatars' appearance. The most well-known virtual world is Second Life[7] (SL) and it is created in 2003 by Linden Lab. SL is the most widely known 3D VW with more than 700 educational institutions to have a grid inside it (Linden Lab,

[7] https://secondlife.com/

2011). Companies and universities have already used SL to test ideas and products, organize workshops, seminars, lessons, staff recruitment, and advertisements (Warburton, 2009). For many years, universities (e.g. the Open University of the UK and Ohio University) and organizations (e.g. ISTE and New Media Consortium) have utilized SL as an alternative learning platform for online or blended learning courses, like course lectures, design-based activities or experimental problem-based tasks (*Figure 3-10*).

Figure 3-10: An educational region inside Second Life

Linden Lab and its founder Rosedale Philip have imagined the development of a 3D VW, in which users can interact, play, work, and/or communicate. Each user can change the size, shape, color, texture of the objects and give them physical properties (e.g. elasticity, gravity, movement). In addition, the Linden Scripting Language (LSL) is used to deliver greater interactivity either among objects or among avatars with in-world objects. Users can also place their objects only in a specific grid, or others' objects from different grids if they have the appropriate rights by the owners. Nevertheless, there are some limitations, such as maintenance cost, and/or support, allocation of functional or learning resources with the appropriate management of student activities which may prevent some educators and scholars to use 3D VWs as learning platforms (Dalgarno & Lee, 2010).

The second category is the 3D open source VWs (OSVWs). Users can become administrators (owners) of a "world" having access to the open-ended core of programming language that is provided in different server modes (networked or standalone) in order to develop their own virtual environments (grids). OpenSimulator[8] (OpenSim) is the most popular 3D OSVW. It appears in 2007, and it has the same features and characteristics with SL, allowing users to interact with their avatars using a-/synchronous communication tools. OpenSim is a 3D server-based platform, open source, and free of charge. It is also

[8] http://opensimulator.org/wiki/Main_Page

interactive, visually-rich having a persistent environment accessible simultaneously by many (distributed spatial or not) users. OpenSim is written in C and is based on Microsoft.NET. Users can create one or more grids of virtual land and allow even other authorized users to manipulate and configure in-world objects and elements (Rico et al., 2011). In addition, since the source code of the OpenSim server is "open" and relatively easy to modify its programming scripting language, users can make the necessary changes, depending on their needs and demands (*Figure 3-11*).

Figure 3-11: A region for creating a house prototype inside OpenSim

To "upload" a 3D OSVW, users need to have an Internet connection to download a client-viewer and to create a standalone virtual environment to "run" it in a personal server or locally in a user's personal computer hard disk, so as to have the control over it as administrators. Also, beyond the server's main program itself that is available for download on its official webpage, there are some distributions that give extra functionality and several preinstalled items, allowing a direct installation of a server with minimum possible configuration, such as Diva[9], Sim-On-A-Stick[10], and New World Studio[11].

3.3.4.1. Advantages and disadvantages

Among various platforms that have been used in the past, such as Learning Management Systems (LMS) or Massive Open Online Courses (MOOCs) for various learning disciplines and domains, 3D VWs, such as Second Life, OpenSim have been considered as also appropriate platforms which can affect positively students' motivation and participation. The participation inside 3D VWs is a powerful magnet for spatially (or not) distributed users, giving them incentives for socialization and collaboration due to the

[9] http://metaverseink.com/Downloads.html
[10] http://simonastick.com/
[11] http://www.hypergridbusiness.com/tag/new-world-studio/

technological capabilities and instructional affordances that offer in blended or (fully) online instructional formats. According to Dalgarno and Lee (2010), 3D VWs have various "affordances" that represent the theoretical learning benefits. This term was mostly preferred over 'benefits' or 'advantages' in favor of referring learning tasks, activities, theoretical underpinnings or pedagogical strategies supported by 3D VWs and labeled as "educational potential". Thus, 3D VWs can provide various potential advantages from both instructional-educational and technological-operational perspective. The most worth noting that need to be denoted are the following (Dalgarno & Lee, 2010; Girvan et al., 2018; Topu et al., 2018; Okutsu et al., 2013):

- to develop and program realistic situations using a 3D simulation environment using a wide range of several constructions with built-in tools and geometric objects. More specifically, visual objects (primitives) of a 3D VW are similar as those of a real-world and obey in certain rules such as laws of physics which are already existed from the system or can be integrated using programming to configure them properly;
- to manipulate rules of the spatial proximity of visual objects and elements with high representational fidelity for various teaching and learning subjects. Therefore, users should specify rules governing on how objects need to manipulate and create with other visual entities (avatars) or other similar objects artifacts in order to be achieved several tasks, such as simulation, artificial intelligence (AI), animation, modeling of natural laws to impart plausible behavior inside a 3D environment;
- to interact and/or move visual objects/elements by integrating behavior using a 3D VW's own programming language or handle those objects/elements using a keyboard and a mouse. Users (spatially distributed or not) can also communicate with a−/synchronous communication channels with other avatars through different instructional formats in blended (face-to-face and online) or fully online settings, in order to be improved students' learning achievements and outcomes;
- to (co-)construct, (co-)manipulate and examine in-world metaphorical representations, artifacts or primitives to design practice-based a knowledge domain. Users have also the ability to access and experiment with simulation-based learning tasks, without having significant technological literacy background. Learning content and design standards for a wide variety of learning subjects can become more realistic and encouraging to relevant standards through implications for theory and practice related to scientific domains, by following, trial-and-error, inquiry-based or problem-based learning approaches.

Despite the growing interest in the use of 3D VWs, several studies have also noticed some disadvantages. From a technological and functional perspective, 3D VWs have also high demands on computer hardware requirements and on the processing power, particularly to the graphics subsystem and

random-access memory (RAM). Also, due to the fact that 3D VWs were not designed for educational purposes, the difficulty of creating learning materials and teaching environments is an issue that should be taken into serious account from instructional designers and scholars. For example, and first of all, the procedure for purchasing a virtual grid (island) in SL is sometimes complex, and time-consuming. The computer equipment must possess the required technical features so that the environment can work properly. Second, the development process for the multimedia objects takes a long time, and sometimes users may lose or cannot control their personal objects/data when the SL's servers crash (Coban et al., 2015). Concerning the platforms that require fees, certain problems will likely be experienced during the purchasing process. Nevertheless, there are many websites available that someone can download and customize visual objects or elements according to their needs and interests. In such a case, users can also upload/download visual objects from other grids if they have the appropriate rights or pay money to use ready-made items from the SL Marketplace[12].

In the case of a 3D OSVW, the maintenance cost of a computer server for the development of such a VW may also prevent the use of this technology. The technical features that someone needs from a server can cause data loss. If some computers have inefficient technical features or even if the server is installed in a computer with low specs, the 3D OSVW will have several "freezing" problems. In this perspective, the deleted visual objects and elements are also difficult to be retrieved inside a 3D open-source "world" (Coban et al., 2015).

A brief summary of the advantages and disadvantages is presented in *Table 3-4* below:

Table 3-4: Advantages and disadvantages of 3D virtual worlds

3D VWs	
Advantages	+ Simulated realistic problem-solving situations/tasks + Geometric objects and primitives for the development and programming of realistically simulated constructions + Interaction of users with visual objects/elements that have realistic simulated representational fidelity + Spatialized communication among other avatars using a−/synchronous channels can support blended (face-to-face and online) or fully online instructional formats + Collaborative construction and manipulation of 3D visual objects in a common and persistent environment
Disadvantages	- High demands on computer hardware requirements in the graphics subsystem and RAM - The development process for the multimedia objects takes a long time - The maintenance cost is sometimes high - Servers that host a 3D environment have crashed and "freezing" issues

From a research methodology and instructional perspective, while many studies have suggested methodologies or educational models, there is lack of comparative studies to investigate the effectiveness

[12] https://marketplace.secondlife.com

of 3D VWs in contrast to other platforms (MOOCs or LMS) with the purpose to present substantial evidence about the learning outcomes in several learning subjects and disciplines. According to the literature review of Hew and Cheung (2014), there is an ever-increasing use of innovative applications in the educational process and their integration into curricula. Additionally, there is an imperative need for conducting empirical studies with respect to explore the effects on learning subjects and disciplines that can ultimately gain by using VWs as technological means (Warburton, 2009).

3.4. The use of three-dimensional virtual worlds in programming courses

3D VWs have become very popular for the development of various applications from interactive games to simulations with high representational fidelity. In this perspective, 3D VWs have been widely utilized in different learning subjects of STEM education and specifically in programming courses. To increase students' engagement and participation, several educators and scholars have proposed their instructional approaches using 3D VWs. For example, Lim and Edirisinghe (2007) have presented results from a pilot project exploring the use of SL for CS and programming through GBL tasks. The results indicated an increased level of student engagement without previous experience, evidence of peer teaching among avatars. Nonetheless, elements and tools inside SL such as notecards were an ineffective medium to provide instructions, and thus further explorations and evaluations will be necessary to evaluate the effectiveness of meeting the learning outcomes.

Rico et al. (2011) have utilized OpenSim for teaching introductory programming to high-school students to measure their subjective experience when they used the *V-LeaF* environment. In their initial empirical evaluation, Rico et al. (2011) have observed that there exists a students' interest to interact with a 3D VW. OpenSim assisted students to have higher levels of attention, and interest in learning programming. Students had a feeling that learning the (scripting) programming language of OpenSim was more interesting by interacting with visual objects and by collaborating with other avatars.

Esteves et al. (2011) have conducted action research to analyze if teaching and learning computer programming could be developed within SL. Results supported the appropriateness of SL as a potential platform for educational purposes in teaching/learning computer programming. The main results are the identification of problems hindering the CS instructor's intervention in SL and the detection of solutions for those problems that were found effective to the success to use SL. However, some students who already had contact with programming and specifically with the C language have presented many faults to understand the basic programming concepts in LSL.

In their study of Jakos and Verber (2016) have investigated the effectiveness of using educational games for learning basic programming skills by developing a 3D game via OpenSim called *''Aladdin and his flying carpet''*. The results have demonstrated that most 6th-grade students achieved all the learning

objectives. While students have achieved the biggest progress in ‘‘complete a program’’ objective, the less was observed with the tasks where ‘‘create a program’’ and ‘‘divide a problem’’ objectives. Lastly, there was no significant difference observed in the results between girls and boys using OpenSim.

3D VWs as platforms for the implementation of learning scenarios in programming courses have gained the researchers’ interest. Also, some other positive learning outcomes according to previous studies highlighted several potential benefits. These are the following (Esteves et al., 2011; Rico et al., 2011):

a) high representation fidelity of visual objects and elements can improve the simulated problem-solving contexts corresponding to real-world problems;
b) better-understanding use and analysis of programming constructs in collaborative tasks;
c) communication with remote a/-synchronous tools among students with their peers or among students with the CS instructor can give prompt feedback to their in-world actions, and fix errors into code; and
d) active participation of students in creating and programming visual tools that can help the implementation of interactive experimentations.

In addition to the above, several are the most noticeable characteristics to support further programming courses. First, self-evaluation and reflection upon students’ cognitive thinking process are achieved visually or acoustically either by integrating behaviors in visual objects or by creating artifacts to link abstract-concept formation to a more concrete game experience (Esteves et al., 2011). Second, students can find more challenging a *‘‘divide and conquer”* problem to achieve the learning objectives in a 3D environment due to a variety of metaphors that can be developed and programmed (Jakos & Verber, 2016). Third, participation in tasks is accessible to all users giving CS instructors opportunities to evaluate their computing skills and competencies during the learning process or providing feedback using a/-synchronous communication tools (Lim & Edirisinghe, 2007).

By taking advantage of 3D VWs, users can improve their cognitive thinking skills through engaging game-based learning tasks (Jakos & Verber, 2016; Rico et al., 2011). Even if the creation of interactive games in 3D VWs is still promising, there is no additional information on how students try to write and execute correctly the code in order to integrate programming constructs as behaviors in visual objects by using 3D VWs’ own scripting language, which is similar to C. Dickey (2005) has already noticed that the built-in tools of 3D VWs can create a high-floor hurdle (“steep learning curve”) that school-age students need to overcome. This situation cannot facilitate students’ engagement in problem-based tasks and eliminate possible obstacles to understanding the correct use of programming constructs. Notwithstanding the general acceptance of 3D VWs in different learning subjects/domains, students’ first-time entry has become the most crucial parameter that might hinder their participation and engagement.

To be addressed the above issue in the most popular 3D VWs, such as OpenSim and Second Life, Scratch for Second Life (aka S4SL) was designed and created by Rosenbaum (2008) in order to facilitate students to write syntactically correct code and integrate behaviors into visual objects. It is an easy way for users to integrate new behaviors into virtual objects (primitives) and predict their interactions inside a 3D VW. S4SL (version 0.1) is a visual palette outside of a 3D VW, in which graphical blocks are snapped together to create a program. Eric Rosenbaum with the Scratch team and the creator of S4SL modified an internal build of Scratch (version 1.1). It comprises a visual palette with control flow statements and command blocks, similar to Scratch's palette, in place of being proposed the design patterns without financial cost (Rosenbaum, 2008). The simple approach to *"copy-and-paste"* programming constructs can help users to transfer colored code blocks of different colored as graphical puzzles that include loops, conditional, motion or behavior into virtual objects' notecards to integrate and incorporate those behaviors and interactions. The combination of the S4SL visual palette with a 3D VW can determine a wide range of high-ceiling/visually-rich applications to be enhanced users' technological literacy that can lead to the active production of dynamic interactions or behaviors in geometric solid objects or complex shapes (artifacts). Thereupon, the use of a 3D VW like OpenSim with S4SL may satisfy the triplet of *"low-floor/high-ceiling/wide-walls"* and it can become really useful for the reduction of the "steep learning curve" that is created when students are involved in complex learning tasks via a 3D VW (Girvan et al., 2013).

Chapter 4: Game-based learning to support computational thinking

This chapter presents the basic characteristic of game-based learning in educational settings generally, and more specifically the potentials of using games created via VPEs and 3D VWs in programming courses. The analysis is focused on the use of game-making and game-playing approaches which can support CT instructional contexts. This chapter also gives information about the related works which have identified drawbacks and difficulties from previous works which have followed either game-making or game-playing approaches. In particular, it gives a pathway to be recognized which user interface design features and elements of games can foster students' cognitive thinking skills related to CT in order to inform educators and game developers how to design and develop simulation games (SGs) using interactive environments. To this notion, this chapter informs how SGs can adequately address gender inequalities and influence boys' and girls' learning performance.

4.1. Game-based learning

The utilization of computer games in different instructional formats known as game-based learning (GBL) is becoming a recognizable term inside school contexts (Maloney, 2008). GBL is a learning approach in which students can use computer games in order to practice or gain knowledge inside (or not) school contexts (Killi, 2005). *"Play"* is a significant facet of GBL because through it people learn how to connect with and/or interpret their physical and social worlds (Gee, 2007). Many efforts have been undertaken to develop digital environments in order to integrate educational content and materials into games so as to increase students' participation (Maloney, 2008). GBL in many learning subjects can greatly improve students' engagement and participation. Such an approach can also provide teachers with instant feedback and tools that can support or even improve learning conditions through (in-) formal instructional settings (Papastergiou, 2009).

With the emergence of digital games in 1970, various efforts have tried to integrate educational content into computer games (Bodrova & Leong, 2003). A computer game is an emulation and a subtractive version of a real or imagined world that has well-defined rules, targets, and limits in which players can interact by playing. Following specific rules and instructions of a game, players can acquire knowledge with appropriate guidance from the instructor (Squire, 2003). Computer games can be used as *"bait"* for learning, vehicles for content, "tools" for engagement, and evaluation of users' strategies for gaining knowledge (Steinkuehler & Squire, 2014). Computer games have provided significant effects on computer-assisted instruction and students' attitudes on knowledge acquisition in different scientific domains and disciplines. As more schools and educational sectors have brought computers into classrooms, computers

games have become an easy way to assist teachers and scholars to participate most students and more those who are getting bored in lectures or traditional instructional approaches which are related to any teacher's instructions.

Computer games cannot only bring to users the opportunity to learn through enjoyable and playable settings with clear goals but also provide immediate feedback to their actions affecting their performance if specific goals are properly achieved (Dickey, 2005). Immediate feedback is also prominent in good formative assessment processes (Sitzmann, 2011). For instance, it is hard for instructors to give constructive feedback and a set of plans for their lessons to incorporate probing questions and subsequent players' actions. For this reason, when in-game feedback is integrated to a game, it can assist instructors to take information about students' performance and progress when they can achieve specific learning goals (Garris et al., 2002). Computer games can also assist players to think systemically and consider the relationships instead of isolated events or visual elements because within a game they can apply and adapt knowledge into various situations (Gee, 2007). In this point of view, players need to think how to accomplish specific in-game activities/missions and give answers to simulated problem-solving tasks with specific learning goals through discovery problem-solving activities without spatiotemporal constraints to overcome challenges that may have in a real-world (Garris et al., 2002; Papastergiou, 2009).

However, the use of computer games alone cannot give reasonable solutions to any problem that teachers face today. In-game learning purposes and goals which are reflected on virtual characters' abilities and opportunities need to be announced from instructors in order to understand what problem-solving tasks each player will face avoiding any possible constraints or difficulties (Gee, 2007). GBL brings to light another aspect of learning, where players are encouraged to explore which in-game elements and objects pave a pathway to gain knowledge. Such a finding can promote the construction of knowledge as a process, in which players interact inside a digital environment to identify and gather information from visual learning materials/elements so that propose solutions to problem-solving activities (Ke, 2009). Students in K-12 education when playing games can gain a variety of skills that are essential for their careers in the professional sector, their personal development, and their well-being. There are appeared many examples of games as indispensable "tools" for conventional (or not) instructional formats. Furthermore, a wide range of skills such as critical thinking and problem-solving, communication and information management and interpersonal and self-management players can gain by playing games (Partnership for 21st-century skills, 2009). Therefore, computer games can greatly fulfill students' learning needs and experiences by supporting various learning tasks which correspond to an imitation of operation, a process or a system consisted of specific simulated problem-solving situations of the real world.

To integrate GBL approaches and thus computer games successfully inside (or not) school contexts, students need to have opportunities that allow them to be educated and entertained within playable contexts.

Various design features and elements need to be referred and integrated into gameplay for certain game mechanics which determine the overall characteristics of the game itself. *"Gameplay"* and *"game mechanics"* are two terms that play a key role in a game and on how well it can satisfy users' preferences, needs, and expectations. According to Salen and Zimmerman (2004), *"gameplay"* is the process in which players interact through a (computer) game. It entails specific patterns defined through game rules, instructions, and challenges that players need to overcome in order to achieve specific learning goals. The same authors have pointed out that playing a game requires from someone to know:

a) what s-/he needs to do inside a game in order to win by achieving specific objectives or what s/-he loses if cannot achieve those objectives properly, and
b) what visual elements that help her/his actions to play and have an immersive game experience, such as mystery, challenge, anticipated outcomes and game features. This comes in align with in-game components or objects which can assist players to consider for which purposes a game is developed and created, such as the use of a visual palette for learning how to program in order to interact and respond to her/his actions.

From a design perspective, gameplay is developed to reveal constituent *"game mechanics"* (Salen & Zimmerman, 2004). *"Game mechanics"* are designed to support each player's interactivity through several in-game activities. In other words, game mechanisms are the "black boxes" which may (or not) be visible in the game; however, such mechanisms can allow the interaction among in-game elements and players (Alessi & Trollip, 2001). Also, players need to understand in-game interactions among all elements and objects, which are capable of receiving inputs and reacting to events made by producing outputs arising from game mechanisms (Fabricatore, 2007). To this notion, players need to be focused on specific objects or elements that they have to deal with in order to identify interactions that can happen inside a computer game (*Figure 4-1*).

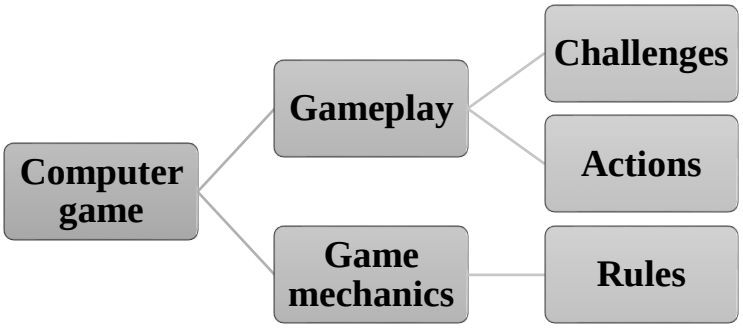


Figure 4-1: Components of a computer game

To understand the use of in-game mechanics through a concrete example, below it is presented a case in which a player when pressing a button, the light will turn on. Assume that a player is required to walk

inside a dark room and s-/he needs to push a specific button in order to see what it is inside. A comprehensive schematic representation of in-game mechanics to turn on this room's light is formulated in *Figure 4-2*.

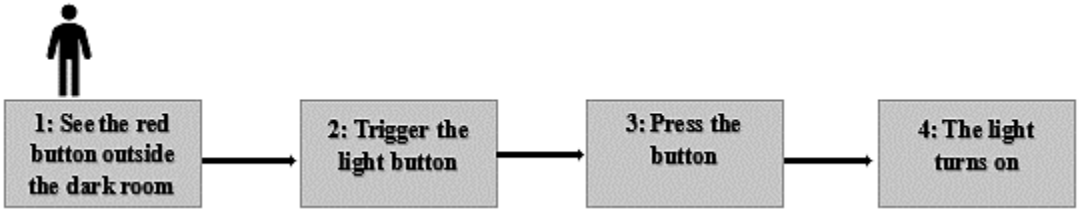


Figure 4-2: A specific example of interaction among game mechanics

A wide range of characteristics corresponding to the gameplay in computer games have to do with an effort to support new types of instruction is summarized as follows (Alessi & Trollip, 2001; Prensky, 2007; Squire, 2011):

- *Learning objectives*: Educational games are designed to have specific purposes and learning goals for one or more (interdisciplinary) learning subjects.
- *Specific instructions and rules*: A clear set of rules that can facilitate the player's interaction with the game is necessary in order to avoid misconceptions about its use for (educational) purposes.
- *Interactivity*: The active role of players and the achievement of specific in-game goals are depended solely on a player's decisions and actions. Without the active participation of each player, the concept of the game cannot exist.
- *Feedback*: The game should have rewarding mechanisms for a correct decision and "punishment" for a wrong one. In this demand, players will be able to distinguish successful from failed actions and concentrate to succeed properly specific goals of the game.
- *Challenge*: Every challenge has to do with uncertainty on specific goal achievements, hidden information, and multiple levels of difficulty. The degree of challenges should be proportional to the level and potential of players which support (or not) directly their actions.

Computer games can also become candidate platforms due to integrated technological and functional features that can be useful in education. Some of the most important benefits which have been well-documented are the following (de Freitas & Oliver, 2006; Dondi & Moretti, 2007):

- Manipulation of in-game's spatial or time conditions and/or digital character's awareness according to user needs and interests.
- Simulation of real or imaginary situations with (or not) rules or behaviors that resemble (or not) those of reality.

- Definition of specific goals and actions with specific results that are visually or acoustically defined as positive or negative in regard to each player's actions.
- The total cost of simulating or assimilating a learning situation that is applied in a digital environment is lower than it is required with human resources in the real world.

To have the appropriate learning conditions inside a computer game, it is necessary to provide a game with a wide range of tasks with deep level of interactivity so that to be engaged easily players on solving certain learning goals with tasks and test how the outcomes inside the game are generated based on their decisions and actions (Squire, 2003). Significant works in the domain of GBL (e.g. Iacovides et al., 2014; Prensky, 2004; Gee, 2007) have stated the importance in regard to the contextualization of gaming focused on the quality of gameplay, when it is explicitly designed to support learning. To summarize, designing learning tasks in computer games requires a multi-dimensional approach in which students need to develop cognitive thinking and practical skills so that improve their learning outcomes.

4.2. Design features to foster computational thinking through game-based learning

GBL approaches are widely utilized in various scientific theoretical to applied subjects/domains. The integration of GBL in formal disciplines such as programming courses has gained much attention for many educators and scholars in recent years. More specifically, previous efforts have proposed the use of GBL approaches to foster CT instruction through programming courses (Werner et al., 2014; Witherspoon et al., 2017). For example, a computer game can fulfill the requirements in programming, since it presents embodied problem-based contexts which can foster students' problem-solving abilities to experience within a scientific discovery process interacting with digital elements and objects (Repenning et al., 2010). In such a learning situation, students need to analyze a problem or a situation and take decisions using skills related to logical and algorithmic thinking for solving problems prior to the writing of a program so as to choose the most appropriate programming constructs for proposing and executing their own solution plans (Brennan & Resnick, 2012). This process may lead students to learn how to think before starting to program by integrating interactions and rules inside objects/elements so that develop and observe game situations and generalize those tasks later. Such learning approaches come in contrast to the most common exercises, in which students tend to formulate and write correctly instructions combined with programming constructs in order to observe the consequences of executing those constructs or use certain constructs corresponding simply to specific problem-solving contexts (Liu et al., 2017; Theodoropoulos et al., 2016).

Various GBL approaches have been proposed by using interactive environments during the last decade not only to foster the development of fundamental of CS concepts but also to influence students' computational practices for solving simulated real-world problems (Grover & Pea, 2013). Newfangled CS curriculums worldwide (ACM Education Policy Committee, 2014; Tuomi et al., 2017) and previous efforts

(Garneli et al., 2015; Grover et al., 2015) have indicated GBL approaches as more appropriate to fulfill students' learning needs and experiences, either with exercises for learning how to program by creating a game (game making) or by playing a game (game playing). Firstly, "game making" aimed at facilitating students to develop skills related to CT by following a scaffolding instruction so as to design and create playable game-based applications with specific storylines and challenges. Programming in such an approach is introduced as part of a wide-ranging activity, in which students are involved by making all in-game contexts from characters design to game mechanics programming (Brennan & Resnick, 2012; Mouza et al., 2016). Secondly, "game playing" aimed at assisting students to develop skills related to CT in a specific game playing context. Programming is getting introduced as part of limited-ranging activities, in which users are involved only by playing activities having a specific character that needs to make substantial progress when specific constructs and instructions of a programming language are used to achieve specific goals (Webb & Rosson, 2013; Witherspoon et al., 2017).

It is of great importance to mention that an interactive environment needs to provide simulated real-world contexts where students can inherently their abstract thinking with logical reasoning and formalize their thoughts into code through gameplay. In this perspective, students need to initially conceptualize a process of using their skills related to CT in gameplay modes to concretize logically abstract concepts without considering any in-game unnecessary information to describe and apply their strategies for simulated problem-solving tasks (Kafai & Burke, 2015). Students as players in GBL contexts are encouraged to take part in activities which can assist them to gain knowledge with a more general understanding about the use of computational concepts so as to articulate skills related to CT and applying their computational problem-solving strategies (Chao, 2016; Werner et al., 2015). In addition, they use games as "tools" in order to explain their approaches on how to solve simulated (real-world) problems.

User interface design features play an important role in game-based learning and instruction. In particular, prior studies (Grover & Pea, 2013; Lye & Koh, 2014) have reported that user interface design features and elements can help K-12 students to understand computational concepts with the visualization of 2D or 3D output so that solve simulated problem-solving tasks logically and methodologically using CT. Additionally, there is a need for appropriate means such as visual tools and user interface design features and/or elements combined with programming tools, such as a visual palette with code blocks that can be used in order to be applied users' computational problem-solving strategies into programs. Such features are significant for testing and debugging those computational problem-solving strategies in an effort to have visual feedback about the correctness of their programs. Other studies (Liu et al., 2017; Witherspoon et al., 2017) have argued that a proposed interactive environment to support programmers needs to provide user interface design features and elements in which creating and/or playing a game-based learning situation can

be closed on what they can understand in a simulated problem-solving situation by taking advantage of intuitive, natural modality contexts for user-interaction tasks.

Many interactive environments from VPEs or 3D VWs have been extensively utilized in programming courses using various features/characteristics with visual tools to foster and support CT instruction (Grover & Pea, 2013; Lye & Koh, 2014). On the one side, VPEs provide several features to support and foster CT in K-12 education. The graphical code blocks have shown considerable promise in programming languages syntax, aimed at giving students a first introduction to coding tasks. To this notion, users are focused more on a problem-solving process than in syntax complexity. The applicability and visualization of algorithmic control flow (code tracing) can facilitate the organization and documentation of code blocks. Thus, users can write, trace their code, find logical errors into their code, turn back to fix issues and observe the consequences of their constructs and commands either in a stage screen by integrating behavior into visual elements or when code blocks are not allowed by the system (visual palette) to be combined together.

On the other side, 3D VWs which have a significant number of characteristics and features to support and foster CT in K-12 education. A 3D VW offers a realistic representation of a virtual environment, in which users can provide solutions to simulated problems, tracking their errors visually and acoustically in order to understand better the consequences of their actions at the execution time (Esteves et al. 2011). 3D VWs can be appropriate for the creation of interactive learning activities allowing users (Dalgarno & Lee 2010; Good et al. 2008):

a) to construct problem-solving contexts using content creation tools and practice competencies;
b) to identify the spatial association of visual objects' rules so that provide prompt feedback on users' actions in high representational fidelity virtual contexts; and
c) to understand metaphorical representations (metaphors) of their ideas without spatial-temporal physical constraints through embodied actions, like view control, navigation or object manipulation.

User interface design features which can lead to the improvement of users' learning experience through game playing in 3D VWs than in VPEs should consider the following two significant issues. The first is the *"flow"* state. It describes a state of enjoyment and psychological immersion referring to the optimal experience through in-game challenges without matter the challenges that someone will face in order to succeed in specific goals when s/-he is fully immersed through challenging and engaging activities (Csikszentmihalyi, 1991). The second is the sense of *"presence"* that refers to a human's feeling when s-/he is somewhere else than truly is his/her location by taking part in computer-generated activities (Topu et al. 2018). The intuitive modality of a 3D VW offers a realistic display of a digital environment displayed in three-dimensions to provide visual objects and elements with high representational fidelity and a view

of changes on elements/objects' motion. This feature can lead to a greater perception and subjective sense of being each user in a place (sense of presence). Also, the immediacy of controlling events and objects/elements in a 3D environment can assist in-world interaction among users and objects. Both representational fidelity and immediacy of control can allow users to interact and predict behaviors by integrating a 3D VW's programming language into elements/objects for solving problems to execute and assess the consequent results of those instructions/commands in problem-solving contexts which are resembled as those in a real world (Dalgarno & Lee, 2010).

4.3. Learning to program through game making

4.3.1. Game-making learning approaches

The increasingly ubiquitous and frequent use of instructive guided game-making approaches using interactive environments in order to assist students' learning to think and practice "computationally" has been largely extended and documented by previous studies (Good & Howland, 2016; Lye & Koh, 2014; Werner et al., 2015). Within specific school contexts, the role of CS instructors is also very important in the learning process since students need to learn how to program in an effort to demonstrate their computational competencies and skills related to CT following game making approaches. This means that another factor which can impact negatively students' performance is the way of using an interactive environment under specific instructional conditions (Mouza et al., 2016). In order to understand the use of CS concepts with CT instruction through programming courses, users need to have the appropriate CS teacher's guidance, otherwise, they may find it hard even to participate (Grover et al., 2015). In this perspective, students are almost invariably intimidated and frustrated in game making because they may find difficult to program and present entirely a computer game without being experts or without the substantial assistance from other experts or CS instructors (Koorsee et al, 2015; Howland & Good, 2015).

Various learning activities following instructive guided game-making approaches have not only significantly influenced the motivation and involvement, particularly younger students in K-12 education (primary and secondary) to participate in programming courses, but also their learning performance. Indicative results from previous studies with regard to students' learning performance have provided significant evidence. More specifically, a game making approach aims to facilitate students to develop skills related to CT in programming courses by following a scaffolding instruction so as to design and create playable game-based applications with specific storylines and challenges. Programming in such an approach is introduced as part of a wide-ranging activity, in which students firstly are involved by choosing from several pre-defined elements, role-playing characters backgrounds, and objects to design and create interactive-playable games. Secondly, students need to specify in-game core mechanics in order to start

programming via visual palettes comprised colored code blocks like a puzzle's pieces and integrate interactions among any chosen objects and elements. Several examples have extensively utilized VPEs.

The first most indicative is the use of Alice. Werner et al. (2012) have reviewed student-created games to identify the CS concepts that are accessible, by counting the frequency of inclusion and successful execution of programming constructs which students have utilized for their game creations. Students needed to learn how to program using specific programming constructs by creating in-game challenges via Alice. The same authors have also found that students' learning performance to be at a higher level by measuring game comprehension tasks which were simpler and lower on more complex to debug and present their programs. The results from the same study have shown that many games created by middle school students exhibited successful uses of high-level CS concepts such as student-created abstractions, concurrent execution, and event handlers. To this end, Werner et al. (2012) have explored at the students' games for evidence about the appropriateness of programming design patterns (i.e., combinations of programming constructs) which integrated inside game mechanics. The same authors have identified a number of non-contiguous sequences in programming constructs over the presentation of students' game creations in Alice indicating lack of high abstraction levels. These findings provide a major difference on what students would like to create from the final creations as executed by the programming constructs that were necessary to apply in-game mechanics and patterns for games that want to develop (Werner et al., 2012).

The second indicative VPE is the use of Scratch. Mouza et al. (2016) have examined how equitable pedagogical practices can be applied in the design of computing programs and how students' participation via Scratch following game-making approaches can influence them to learn better how to program. Students seemed to use CS concepts, computational practices, and attitudes toward computing with the use of certain CS concepts, such as loops, conditionals or data within or across objects to present more advanced computational in-game concepts. Further, the majority of students were able to exhibit good computational practices associated with code organization and documentation and to develop user-friendly programs with smooth functionality. The results, however, indicated that most students utilized certain CS concepts, such as loops, more than others, such as conditionals and data. Even fewer students utilized parallelism within or across objects or more advanced concepts associated with operators.

Another game-making approach is the combination of Massive Open Online Courses (MOOCs) with Scratch. The focus of Grover et al.'s (2015) study was to create and test programming courses for middle school. As *"Foundations for Advancing Computational Thinking"* (FACT) titled all courses which were aimed at preparing and motivating school students of secondary education for future engagement with algorithmic problem-solving using Stanford's OpenEdX MOOC platform in blended in-class for game-making instruction. By assessing students' final projects, it appeared that the FACT courses helped them

to build a substantial understanding beyond the revealed basic algorithmic flow of control in computational solutions. Nonetheless, Grover et al. (2015) have identified that students had difficulties in proposing algorithms as pseudocode in "semi-English" language and transform those algorithms into workable consisted of the most appropriate programming constructs, such as loops, in an effort to apply their computational problem-solving practices.

All the above game making approaches are instructive guided by one or more CS instructors. Instructive guided game-making approaches can support the representation and visualization of problems require predefined scenarios. In game making approaches, CS instructors can measure the students' learning performance by taking under serious consideration the following three aspects: a) the operability and adequacy of programming constructs which are generated (or not) properly inside design patterns and if such patterns are (re-)used extensively to be programmed other in-game objects or elements, b) the frequency of applying problem-solving strategies which include how many times students repeat and reuse (or not) programming constructs and instructions, and c) the description and appropriateness of integrating interactions using design patterns into objects or elements and what is produced in their final creations.

There is good evidence that many instructive guided game-making approaches discussed above can offer a way for novice programmers to engage in coding tasks. Nonetheless, other studies (e.g. Denner et al., 2012; Howland & Good, 2015) have argued that visual programming may not be designed to encourage the development of a more general understanding about the appropriateness of using computational concepts in various problem-solving learning situations. Even though it is syntactically easier learning how to code by combining programs via a block-based palette, the conceptual difficulties in understanding and using code blocks such as variables and loops may still exist for solving simulated problems (Mouza et al., 2016). Thus, the evidence is somewhat tenuous in terms of the sheer number of studies which have the tendency to focus exclusively on the assessment of how correct design patterns are "running" based on a code tracing analysis. Despite the fact that students seemed to participate in engaging tasks to master how to code correctly a variety of programming concepts using only a visual palette with naturally-express phrases (or words) to apply code blocks so as to avoid programming with a general-purpose language, it cannot be guaranteed that they have learned the correct way how to think and practice correctly computational concepts (Grover et al., 2015; Howland & Good, 2015).

4.3.2. Drawbacks and difficulties

Notwithstanding the foregoing potentials and benefits of using VPEs that have been described from the above-mentioned analysis of prior works, several imminent difficulties to overcome and understand what students finally learn with the integration of CT as a cognitive thinking process in programming courses requires to have its own answer. In this respect, there is a dearth of recent evidence on whether

VPEs can engage students in a way of thinking how to solve simulated problem-solving tasks and prepare them for more advanced programming activities. In this perspective, the utilization of interactive environments has become a target of negative criticism from a growing body of literature for two reasons.

First, *"Use–Modify–Create"* approaches are focused on code block commands to be sequentially and syntactically correct coding tasks, in which students get to use only certain CS concepts, such as loops, more than others, conditionals and data by remixing or adding new code blocks to already existed inside previous design patterns (Brennan & Resnick, 2012; Mouza et al., 2016). Nevertheless, with a specific solution resulting from the use of frequently similar and/or commands/instructions with programming constructs and data representation, students tend to create games that sometimes seemed to be similar to others. Computational practices, therefore, cannot guarantee why students start using specific instructions and programming constructs to solve problems.

Second, *"Do-It-Yourself"* as project-based ("bottom-up") learning approaches are the most common-in-use for learning computer programming. Several studies have presented results where students tend to create and use ambiguous "trial and error" approaches to create their own computer games, either by copying and pasting code blocks of other projects or by adopting only some programming constructs from other design patterns, rather than creating patterns arising from a thinking before coding process which can be considered as proposed solutions to problems (Grover et al., 2015; Werner et al., 2012). Nonetheless, even when code blocks are correctly written and executed by synthesizing and/or copying-pasting parts from the use of specific programming constructs, which are mainly observed by creating design solutions of code blocks are sometimes related to the incomplete or non-project parts from the online system of Scratch, such as the repeating or sequence programming constructs, then students mostly may provide insufficient game applications. The results from game-making approaches regarding CT instruction through programming courses seemed to become a process in which students can:

a) develop superficial knowledge that includes a limited understanding of the code's purpose and fail to apply their problem-solving strategies for proposing solutions to a problem. It seemed that they cannot understand the main problem and its subparts in order to use and execute correctly specific programming constructs to solve each one of those parts (Brennan & Resnick, 2012);
b) understand how to use CT skills into a cognitive thinking process in which they cannot apply their thinking solution plans for solving a problem into the code to create in-game mechanics which is needed during game playing. Therefore, students fail to inherently conceptualize their cognitive thinking process in playable modes and concretize logically abstract concepts (Werner et al., 2012);
c) use only algorithm instruction by starting and ending with the construction of pseudocode or flowcharts as "vehicles" in programming courses to create algorithms, students can comprehend

programming logic at a low level, and they fail to transfer this knowledge to other (general-purpose) programming languages (Grover et al., 2015).

From the above analysis, the main concern is whether students can develop skills related to CT to solve a problem when they also try to comprehend source code that implies in a *"programming as activity''* perspective. Furthermore, it is arguable if a set of skills related to problem-solving, logical and abstract thinking are associated with a more general understanding of computational concepts and practices. Therefore, students need to learn how to utilize innovative technological devices, acquire skills related to CT in order to understand how to think before start coding and how to combine proper cognitive thinking strategies using interactive environments.

4.4. Learning to program through game playing

4.4.1. Game-playing learning approaches

An alternative learning approach that gains ground in recent years is learning computer programming by playing computer games. Computer games are also appropriate for instructive guided game playing approaches in programming courses. Playing games can support CT instruction through problem-solving tasks ranging from tightly constrained to *"drill and practice"* (Liu et al., 2017) to more open-ended simulations (Lye & Koh, 2014). A game playing approach can become another option that can support students to describe and practice their solution plans arising from their intuitive understanding of events in different gameplay settings in favor of debugging and understanding the correctness of their thinking about solution plans into code (Liu et al., 2017). It aims to facilitate students to develop skills related to CT in specific game playing contexts which are exclusively pre-defined by game developers or CS instructors, and many times such games are related to the most well-known (see for example *"Minecraft"* or *"Angry Birds"*) that students tend to play in their daily life. Programming is getting introduced as part of limited-ranging activities, in which students as players can learn how to program by playing specific in-game tasks having specific characters, roles and goals in order to achieve certain goals by making substantial progress when specific programming constructs and instructions.

During the last five years, many scholars and education researchers have admitted that students can develop a variety of skills related to CT by playing games using VPEs, prototypes and web platforms.

Webb and Rosson (2013) have utilized semi-structured projects that could be modified with code blocks errors via Scratch and Alice to introduce and support interaction among students and visual objects/elements with CT concepts, including problem-solving, abstraction and basic computational vocabulary. The findings from the same study have suggested that learning tasks in which scaffolding instruction followed by using Scratch created an effective way to convey CT concepts and skills in a short amount of time while serving as a funny and engaging learning activity. The same authors identified

considerable success on students' overall problem-solving process by testing and debugging their workable programs. Nevertheless, there was the only one study and it was not found any other related study to utilize VPEs.

"Code.org" (or *"Hour of Code"*) is a nationwide initiative by the CS Education Week. It was created to introduce millions of students to one hour of using computers for learning computer programming with more than 154.145 events to be successfully made[13]. This website has also gained educators' and researchers' interest. Theodoropoulos et al.'s (2016) study aimed at assessing the learning effectiveness and motivational appeal of digital games for learning fundamental programming concepts, involving high school students who have used games from the *"Hour of Code"* website. The same study investigated students' attitudes from gaming activities to reveal the quality of their learning experience based on correlation analysis of their profiles with a twofold purpose. The first was to identify potential differences in computer games that can promote algorithmic thinking and basic programming skills. The second was to be measured students' performance by investigating possible correlations with their cognitive styles and any possible biases arising from the use of specific games. The results have suggested that specific games utilization is an affecting factor that might produce different results regarding students' preferences. For example, some students might be better at puzzle games, whereas others might prefer adventure games.

In another study, Román-Gonzalez et al. (2017) have suggested that skills related to CT by playing games from the *"Hour of Code"* website assisted students' learning in different coding tasks through logical and visual-spatial problems including those for solving mazes or designing geometric patterns. Thus, in their study, the same authors aimed at promoting and validating a new instrument called *"Computational Thinking Test"* for measuring CT, and additionally the same authors have tried to give evidence in regard to the correlations between skills related to CT, including other well-established psychological constructs related to students' cognitive abilities. The use of games from *"Hour of Code"* seemed to assist high school students to understand several computational competencies. Nevertheless, the same authors have raised concerns since such a process indicated clear biases on the development and use of specific cognitive thinking skills, thus on what students tried to solve by playing games in specific problem-solving tasks. Almost all those tasks were focused only on modeling scientific simulations and algorithmic composition of code blocks which are integrated into visual elements.

In game playing approaches, CS instructors can measure the students' learning performance by taking under serious consideration the following two aspects:

a) the operability and adequacy of programming constructs which are generated (or not) properly inside design patterns and if such patterns are (re-)used extensively by using programming

[13] https://csedweek.org/

constructs and instructions in order to integrate behaviors inside visual characters having specific in-game goals other in-game objects or elements, and

b) the frequency of problem-solving strategies which include and repeat (or not) only specific programming constructs and instructions.

4.4.2. Drawbacks and difficulties

By following game playing approaches through semi-finished or simple pre-defined concepts of well-known games using VPEs or *"Hour of Code"*[14], students can play computer games that promote algorithmic thinking and basic knowledge about programming. Although in recent years the growth of CS curricula at online venues such as *"Khan Academy[15]"* and *"Hour of Code"* is being extended, their success for the development of deeper, transferable CT skills is yet to be empirically validated, and so far, lacking rigorous assessments (Grover et al., 2015). Playing with artistic expression tasks to learn how to think and practice "computationally" using well-known interactive games is remaining a respectable starting point that can enhance students' technological literacy. For instance, in *"Hour of Code"*, students try learning how to program and understand the use of CT principles within the context of experiments using simulation models from real-world phenomena, like "*StarLogo Nova"* (agent-based modeling paradigms) and within well-known computer games, like "*Minecraft"* or "*Angry Birds"*.

However, other researchers have the opposite view. In particular, previous studies (Román-Gonzalez et al., 2017; Theodoropoulos et al., 2016) have pointed out the following negative critics in regard to the *"Hour of Code"* website:

a) it hosts games that do not fruitfully support all possible programming phases, but only specific ones, such as problem-posing, coding, debugging, and pre-defined selected solutions without the choice of proposing alternative solutions.

b) it causes possible biases, especially for design solution-thinking with skills related to CT that can be restricted due to the pre-defined tasks which are accomplished by using specific coding design patterns.

c) it cannot largely encourage students to develop higher-order thinking skills that can be conceptualized as a set of cognitive thinking and abstract reasoning in order to generalize adequately alternative solutions to problems. If students cannot understand and clarify problem-solving thinking on how to apply computational rules and programming constructs into the code, they may not properly use and express relevant basic computational concepts correctly to propose their solutions to a problem encountered.

[14] www.code.org
[15] http://www.khanacademy.org

4.5. Addressing gender inequalities in programming using interactive environments

Over the last ten years, addressing gender inequalities in programming courses, especially in school-age contexts, has gained considerable interest (Grover & Pea, 2013; Kafai & Burke, 2015). Persistent concerns about the underrepresentation of girls arising also from the use of interactive environments, particularly in the light of the encouraging elimination of the gender "gap" (Howland & Good, 2015; Werner et al., 2015). Especially, lack of interest and participation of girls cause usually their overall negative attitude towards learning computer programming (Cohoon & Aspray, 2006; Denner et al., 2012).

To engage all students in learning activities, an interactive environment needs to include a set of tools that can allow boys and girls to create or play through problem-solving tasks, in which they are engaged to learn how to use specific CT concepts including procedural, data abstraction, logical thinking, and debugging (Grover & Pea, 2013). Various studies have paid attention on how to use interactive environments and visual tools regularly for novices in favor of learning how to program as an effort to bridge the gender "gap" in computing education. Over the last years, the field of CS that is related to CT integration has already utilized various interactive environments which contain different user interface elements and features. Users can start learning how to program using a visual palette to drag and drop code blocks by playing (or by creating) which can be integrated inside visual objects of an interactive "world" (Maloney et al., 2008; Resnick et al., 2009).

A variety of previous studies (Howland & Good, 2015; Mouza et al., 2016) have suggested some learning approaches which can promote students to a greater understanding of CT in furtherance of avoiding possible gender biases, and gradually to encourage their interest in more advancing programming languages. On the one side, the experience with interactive environments to support CT instruction in programming courses has significantly influenced not only boys' and girls' motivation and participation but also the experience with interactive environments but also their learning performance through game-making approaches. In their study, Kelleher et al. (2007) have found that girls could learn how to use fundamental programming constructs with Storytelling Alice easier than for boys since storytelling as an instructional approach for girls seemed to be more appealing.

Denner et al. (2012) have focused on girls' computer competencies in programming through game making. The alignment of storytelling and game design for the description of the correct use of programming concepts is an important issue that can also influence girls' participation. The results indicated moderate levels of complex programming activities when girls have created games using Stagecast Creator. Nevertheless, the same authors have found that girls have enhanced easier their computing skills due to their better perceptions about computers as supporting tools, but with moderate levels to use or compose the programming constructs.

Howland and Good (2015) have described the design and assessment of *Flip*, a visual programming language aimed at helping middle school boys and girls to develop skills related to CT by creating their own 3D role-playing games using *Neverwinter Nights 2*, in which players explore a large fantasy world and take part in dramatic interactive stories. Specifically, a majority of girls were able to use *Flip* palette for writing small programs and provide pseudocodes for storytelling creations and integration of code through visual in-game behaviors to their visual creations at a larger extent than boys did. Thus, girls seemed to write more complex scripts than did boys showing greater learning gains relative to the boys. Nonetheless, even if girls succeed to write pseudocodes greater, the findings indicated a relatively small number of conditional statements that were used in regard to the correctness of the proposed coding scripts which cannot convince CS teachers about a broader understanding of CT and support their progress.

On the other side, the experience with interactive environments to support CT instruction in programming courses has significantly influenced boys' and girls' learning performance through the game-playing approaches. For example, Webb and Rosson (2013) have evaluated a set of computing activities that illustrated in Alice, Scratch or Lego RCX and required by applying CT concepts, such as problem-solving to finish some semi-structured tasks. All tasks seemed to be engaging and motivating for girls as working on their own for computation problem analysis and solution expression. In the same study, girls learned and reused successfully better computational concepts through fading scaffolding tasks such as problem decomposition and mapping into computational solution steps than boys. The same authors have paid certain attention to tasks that novices can become overwhelmed if asked to start from scratch when using a computing tool for the first time; starting from a working example that may offer several opportunities to explore and build confidence in design thinking and programming skills usage.

Román-Gonzalez et al. (2017) have intended to provide an instrument for the measurement of skills related to CT and give evidence through association of CT with key related psychological constructs using *"Hour of Code".* The same authors argued that the projection of logical and visual-spatial problems, such as solving mazes or designing geometric shapes can assist the development of CT. Also, it appears a greater spatial ability of boys with higher values in the computational complexity of scripts written as algorithmic solutions which applied into more correct programs than girls had through in-game tasks including mentally logical and visual-spatial problems. The results supported the opinion that CT is associated with general mental and cognitive aptitudes, such as inductive reasoning, spatial and verbal abilities. This corroborates that spatialized problem-solving activities are a remarkable option for the conceptualization of skills related to CT.

Many interactive environments have provided supportive features and elements assisting students to create their own artifacts and link abstract-concept formation to a more concrete game experience for measuring boys' and girls' learning performance. Nonetheless, project content analysis through artifacts

reflected as a means of assessing CT but it quickly revealed limitations. More specifically, existing works have mostly focused on the assessment of students' final creations in order to understand how they tried to develop and use skills related to CT from problem formulation to solution expression (Howland & Good, 2015). Such an effort refers to a code analysis of design patterns based on the applicability and visualization of control flow (code tracing) created by (simple or nested) programming constructs (Denner et al. 2012). As shown by analyzing previous studies, the measurement of students' learning performance was based on design patterns and game mechanics which were created by combining specific programming constructs to understand if those patterns (or mechanics) that have a rationale to be included in gameplay modes (Kelleher et al., 2007). The game mechanics include specific rules for actions, behaviors, and control mechanisms that can be available to each player in order to provide specific actions when each one needs to take and program his/her decisions for specific gameplay modes (Werner et al., 2014). Even if developing and programming gameplay using correctly programs, and this is proved through a code tracing analysis; however, it is arguable whether the use of programming constructs can also cause (or not) abnormal program execution for problem-solving tasks (Webb & Rosson, 2013).

A brief summary of results and general outcomes from the above-mentioned studies are presented below in *Table 4-1*.

Table 4-1: A summary of results from previous studies which have tried to address gender inequalities

Studies following game making approaches	**Programming environment**	**Results**	**General outcomes**
Kelleher et al. (2007)	Storytelling Alice	Girls performed better in learning how to program with storytelling tasks than boys.	+ Visual palette can assist boys and girls to learn how to program + Game-making activities motivate and engage boys and girls. - Previous studies were focused on project content analysis of students' final creations. - Less attention was given on what finally students learn in computer programming following game making approaches.
Denner et al. (2012)	Stagecast Creator	Girls have enhanced easier their computing skills but also achieved moderate levels of complex programming in game making activities.	
Howland and Good (2015)	Flip palette combined with Neverwinter Nights 2	Girls have written more complex scripts than did boys showing to have greater learning gains relative to the boys.	
Studies following game playing approaches	**Programming environment**	**Results**	**General outcomes**
Webb and Rosson (2013)	Alice, Scratch or Lego RCX	Girls learned and reused successfully better computational concepts in simulated problem-solving tasks.	+ Game playing approaches can lead students to develop and use skills related to CT from problem formulation to solution expression. + VPEs are also utilized to be manipulated into
Román-Gonzalez et al. (2017)	Games in "Hour of Code"	Boys had more great spatial ability with higher values in the	

		computational complexity of scripts written as algorithmic solutions which applied into correct programs than girls through mentally logical and visuospatial problems.	simulated and game-based problem-solving contexts. - A limited number of works have tried to investigate gender inequalities through game playing. - Lack of evidence to be investigated how interactive environments can support students to think about how to use more advanced programming activities in problem-solving tasks.

Concerning all the above, a growing interest is still today existed on how girls and boys learn computer programming following GBL approaches. Several works were conducted in order to measure students' learning performance based on the implementation of coding tasks and specifically on the code tracing analysis through game making. Nonetheless, there is a lack of recent evidence on whether interactive environments can engage students in a way of thinking more advanced programming activities by understanding why specific programming constructs need to be utilized in specific problem-solving tasks. Recent studies (Denner et al., 2012; Howland & Good, 2015) have already advocated that programming is motivating for boys and girls either through game making or game playing (Webb & Rosson, 2013; Román-Gonzalez et al., 2017) to eliminate any potential gender inequalities; however, few have presented findings on what they finally learn. In other words, there are widespread concerns over the lack of computational understanding and its effects in solving problems, specifically of girls in programming and how to address this imbalance needs further investigation. Foremost, there is not yet identified any study to investigate whether the use of interactive environments by playing to learn how to program has an impact on boys' and girls' learning performance.

4.6. Recent trends and challenges

The widespread acceptance of GBL in programming courses to support CT instruction is inevitably reliable and well-founded by using interactive environments. As described in previous sections, a substantial amount of interactive games has been proposed by following game making approaches, ranging from simple simulated problem-solving learning tasks in developing and programming adventure games (Denner et al., 2012), role-playing creations (Howland & Good, 2015), albeit less studies have followed game playing approaches focused on problem-solving tasks through maze games (Román-Gonzalez et al., 2017) or simulations such as feeding a fish (Webb & Rosson, 2013). Computer games and specifically simulation games (SGs) have received great attention and rapid growth so as to assist users to become more active in several tasks because they can connect adequately theory and practice in a knowledge acquisition process. In specific, users can develop problem-solving and computing skills in CS courses, because they

can develop and apply their strategies as solution plans (Lye & Koh, 2014). Likewise, in regard to the integration of CT into programming courses and specifically in K-12 education, previous studies (Fluck et al., 2016; Webb et al., 2017) have stressed to the importance of establishing visual or/and symbolic representations which can be used to introduce and explain computational concepts related to abstraction, algorithm, automation, decomposition, debugging and generalization. Without using one of the appropriate forms of notation, students at the age of 13-16 years-old may really strive to develop cognitive abilities for spotting and solving problems (Kalelioglu et al., 2014; Mouza et al., 2016).

The integration of GBL in programming courses for CT instruction has been extensively utilized in K-12 education providing many good learning tasks. Cooper et al. (2003), for instance, have noted that although students may originally be attracted in programming due to their previous experience with computer games and multimedia applications, they can quickly be discouraged as they may find extremely difficult and time-consuming to create their own. In this perspective, students may pay more attention to design games aesthetically, without having to learn how to transform specific algorithmic steps into the source code. In their meta-review, Costa and Miranda (2016) have provided serious evidence to overcome some of these problems and facilitate the learning process in programming learning at an initial stage. The same authors disclosed that students should first acquire the programming's logic of a programming language and after that its syntax. This relies on understanding the creation of a game and *"know how"* so as to solve a problem that students face (Cooper, 2010; Kafai et al., 2014). Also, the superficial use of learning practices such as those reflected on "*drill and practice*" through game making approaches can lead to insufficient computational practices, where players tend to experiment with actions with no reflection on learning, but simply experimenting and programming artifacts until their scores can be improved (Brennan & Resnick, 2012). Such a process requires only on *"trial and error"* coding tasks, thus it cannot impact students' learning performance (Denner et al., 2012; Garneli & Chorianopoulos, 2017). For instance, an easy or a simple game using only "*drill and practice"* can assist them at an initial problem-solving learning stage to practice without worrying about the syntax complexity. Nonetheless, even in this case, they may struggle to rationalize and apply similar code blocks for more complex or larger problem-solving tasks with logical reasoning in order to propose a solution (Hong & Liu, 2003; Liu et al., 2017).

To overcome the aforementioned design challenges, there is a broad agreement arising in which it is converged on the game playing in order to prevent students from creating games without specific purposes or with very simple problem-solving tasks for CT instruction in programming courses. During the last four years, prior efforts have been appeared to suggest the creation of interactive environments following design features and guidelines. Furthermore, prior literature reviews (Burke & Kafai, 2015; Lye & Koh, 2014) have argued that computational problem-solving strategies require the development and connection of skills related to CT combined with programming skills for presenting design patterns as solutions to a problem.

In specific, Lye and Koh (2014) have proposed a constructivist (thinking-doing) problem-solving learning approach through game playing, including the use of a simulated (authentic) real-world problem, the adoption of information processing strategies, the scaffolding of the program construction with the instructor's feedback to more complex activities so that assist students' reflection. In their review, Kafai and Burke (2015) have suggested the connection of serious gaming opportunities, like the well-known game *SimCity* with the newly released *Scratch 2.0*, where students will both know how playing a game can contribute to a better understanding of a simulated problem-solving situation and propose a solution through programming. Key features that can enhance a more in-depth learning process to explore with a high-level of freedom, possible prediction of actions, analysis, and testing of any ideas/hypothesis based on the abstract and analytical reasoning in an effort of planning and applying solutions based on their problem-solving strategies (Good et al., 2008). Thence, educators need to propose the appropriate learning conditions through interactive environments for having all students able to define clear and unambiguous instructions for carrying out a process by developing skills related to CT and the expression of a solution into code (Good et al., 2010).

Many learning tasks arise from the idea of *"low floor, high ceiling"* through simulated problem-solving tasks using interactive environments. It is one of the most important issues which have been widely utilized for the creation of environments to foster CT (Lye & Koh, 2014). Using such principles means that students learn program and mostly novices try to create workable but easy to create programs (low floor), and tools which can be used in order to assist them leverage such tools to create more advanced programs (high ceiling) (Maloney et al., 2008; Repenning et al., 2010). Previous studies have disclosed that to become effective and promote the development and use of CT an interactive environment, it needs to provide various tools where students must have a low threshold and high ceiling tools that can support gender equality. For example, proposed activities to foster CT are those of abstracting the data information, integrating pertinent behaviours into visual agents, and applying rules or instructions need to be combined with programming constructs so that evaluate the consequences of those instructions and constructs via modeling and simulations (Kafai & Burke, 2015; Repenning et al., 2015). Other studies have argued that computer simulations need to support gender equality. For instance, discovery learning tasks are more preferable rather than tasks focusing on the creation and programming of a specific storyline that characters are included or other fast-paced actions with conditions that demand to fight with other digital characters (Robertson, 2012). An indicative example will be a game when players have specific roles, storyline, simulated problem-solving tasks, and goals with the right toolset to produce different coding tasks in well-designed instructional contexts. Within these contexts, boys and girls can easier master abstract computational concepts, construct meaningful computational artifacts and apply their solution plans (Carbonaro et al., 2010; Good et al., 2010; Grover et al., 2015).

In spite the growing popularity of CT into computing curriculums for promoting many 21st century competencies in K–12, GBL approaches related to game making and game playing are still recommended as noticeable approaches which have been utilized for CT instruction among school contexts. Literature reviews in the field of CT instruction (Grover & Pea, 2013; Lye & Koh, 2014) and previous studies (Repenning et al., 2010; Werner et al., 2015) have suggested that GBL activities can be developed through the demonstration of computational competencies such as conditional logic, iterative and parallel thinking, and/or data abstraction. Although recent research has recognized the appropriateness of using interactive environments, others have an opposite view. For example, previous works which have already utilized VPEs, such as Scratch (Grover & Pea, 2015), Alice (Kelleher et al., 2007) or AgentCubes (Repenning et al., 2010) have supported the opinion that such environments are lacking appropriate means to provide abstract functionality into functions and procedures including a design scaffold for teachers and students to transparently map out and observe subparts of problem with a view of encouraging the development of a more general understanding of computational concepts and express more properly a solution plan. Accordingly, a substantial body of relevant literature reviews about teaching CT through computer programming courses (Burke & Kafai, 2015; Grover & Pea, 2013; Lye & Koh, 2014) have come to the statement that there is an overt "gap" concerning either the creation and use of alternative platforms or the combination of already known tools for game-playing tasks in such an interactive environment with the purpose to assist computational understanding and learning in this research area for students in K-12 education. For this reason, previous works (Repenning et al., 2015; Hsu et al., 2018) have admitted that the development of new interactive game-based environments not only influence the "flow" experience in learning processes built expressly to foster CT among school-age children, but also to develop and use cognitive thinking skills such as problem-solving, abstract, logical reasoning and programming.

A brief summary of game design trends and challenge is provided in *Table 4-2* below.

Table 4-2: Recent trends and challenges in game design to support computational thinking instruction

Issues related to CT instruction	Related works	Trends	Challenges
Learning approaches	Carbonaro et al., 2010; Good et al., 2010; Grover et al., 2015; Repenning et al., 2015	a) Avoid difficult and time-consuming creations. b) Evade "*drill and practice*" approaches through simplified and gamified tasks. c) Dodge any *"trial and error"* coding tasks.	a) Connect theory and practice in a knowledge acquisition process to develop problem-solving and coding skills. b) Establish visual or/and symbolic representations to assist students explaining their computational concepts. c) Support students' understanding on using skills related to CT through instructive guided approaches

			each simulated problem-solving tasks.
Learning tasks	Grover & Pea, 2013; Repenning et al., 2010; Kafai et al., 2014; Witherspoon et al., 2017	a) Offer in-depth learning tasks to explore all users with a high-level of freedom, to predict their in-game actions, to analyze and test their ideas/hypothesis. b) Require well-defined problem-solving tasks. c) Execute programming commands and workable programs in *"low floor, high ceiling"* simulated problem-solving tasks.	a) Provide game playing conditions for the development and connection of skills related to CT combined with programming skills. b) Present different design patterns as solutions to each subpart of the main problem. c) Achieve in-game goals with the right toolset to produce different coding tasks.
User interface design	Burke & Kafai, 2015; Liu et al., 2017; Lye & Koh, 2014; Webb & Rosson, 2013	a) Provide in-game visual objects/elements which assist users to abstract the data information. b) Design game where players have specific roles, storyline, simulated problem-solving tasks. c) Develop programming tasks into game-based environments to foster the students' flow state to enhance their learning experience. d) Allow students to apply rules or instructions combined with programming constructs to integrate pertinent behaviours into visual agents using programming tools like a visual palette with code blocks.	a) Simulated (authentic) real-world problem connection of serious gaming opportunities. b) The creation of SGs using VPEs or other platforms which can provide problem-solving tasks with realistic simulated representational fidelity. c) Analyze the demonstration of core concepts by observing realistic simulated objects and elements in order to program finally design patterns to apply any proposed solution.

In addition to the above, the lack of theoretical design frameworks is revealed in regard to the correct use of specific requirements and guidelines for the creation of a computer game, and specifically for SGs in order to support CT instruction through programming courses (Lye & Koh, 2014). Without having specific design guidelines, instructional technologists, scholars and educational designers cannot be informed on how a SG should be designed to enhance the learning experience and to assist players to link abstract concepts with more concrete game experience (Grover & Pea, 2013). Many educators and researchers (Chao, 2016; Liu et al., 2011; Witherspoon et al., 2017) have asserted that students' computational problem-solving strategies can be applied via SGs; thus, influencing to a large extent their learning performance. This means that students need to practice more effectively CT concepts such as abstraction, logic reasoning and algorithmic thinking in a simulated real-world context (Grover et al., 2015; Liu et al., 2017). Therefore, an interactive environment needs to foster CT and support the development of SGs having all those design features and elements which can increase students' engagement in pursuit of explaining and proposing their solution plans for simulated problem-solving situations.

4.7. Computer simulation games to support computational thinking

GBL in programming can provide engaging exercises/tasks in which players can participate and facilitate their flow experience, regardless of gender or socio-cognitive background (Liu et al., 2011). An indicative example of games that can facilitate players' flow experience is the utilization of SGs. Generally, SGs are increasingly being applied to foster higher-level abilities in educational contexts, as they may facilitate an active learning experience. A SG covers a wide range of simulated real-world activities in which students can participate in various learning tasks, such as training, analysis or prediction of in-game conditions. Users have specific roles and well-defined responsibilities or constraints in simulated (real-world) activities can create a visual-rich and engaging digital-oriented environment (Garris et al., 2002).

More specifically, the use of a SG in programming courses can present embodied problem situations fostering students' problem-solving ability, and thus experience to learn how to use fundamental programming constructs within a scientific discovery process (Liu et al., 2011). It encompasses several embodied simulated real-world problem-solving situations that foster students' abstract thinking and logic ability, when they are in *"flow"* state since they are more likely to demonstrate in-depth learning experience when applying their own computational problem-solving strategies. Players can study through several exercises in *learning-by-example* perspectives and develop skills, such as higher-order, analytical reasoning and problem-solving (Liu et al., 2017). Also, players can address problems arising from specific problem-solving situations and trying to recognize the consequences of their decisions by using several programming constructs in order to propose solution plans to several problem-solving situations (Witherspoon et al., 2017).

Simulation prototype games created in VPEs and 3D environments have gained an increased momentum, especially in high school programming courses. Following game-making approaches, several studies have controversial results about students' learning performance. For example, in their study, Brennan and Resnick (2012) have assessed students' performance based on strategies followed to create interactive games using Scratch. In particular, programming interactive media, such as the creation of simulations about virtual countries, with the player's making decisions to support and control trigonometry in physics simulations were utilized for the development of CT.

Repenning et al. (2015) have proposed a strategy that gives opportunities for students to design and program STEM simulations by leveraging CT skills acquired from game design simulations using AgentSheets and AgentCubes. The same authors have supported the opinion that the use of interactive environments beyond programming is also to explain the idea of CT into gameplay. Their findings indicated that students' learning abilities and problem-solving skills can be extended and transformed to the next level of problem domains; that is ranging from SG's formation to its sufficient implementation.

Garneli and Chorianopoulos (2017) have conducted an empirical study to investigate CT skills development and student motivation under two diverse approaches. Two middle school student groups were taught computer programming in two different ways; one group represented certain physics concepts by creating a simulation, while the other group copied the same physics concepts on a video game using Scratch. The results from their study unveiled that a video-game construction approach could be challenging since students had a higher performance creating "realistic" digital applications based on advanced graphics, sounds, and user interfaces for learning coding and science concepts.

Recent research on learning computer programming showed that an active and constructive process through the creation of 3D game prototypes can become more effective when it comes to problems in simulated real-world contexts. To this notion, studies have reported that problem-solving, game-based, activity-led exploratory learning tasks can support a student's analytical and logical reasoning thinking skills. For example, in their paper, Liu et al. (2017) have presented debugging exercises to middle school students and analyzed problem-solving behaviors that integrated into visual elements/robots of a 3D prototype game called *"BOTS"*. The same authors have identified behaviors in relation to problem-solving stages and correlated these behaviors with the student prior programming experience and performance. Nonetheless, learning how to program by playing games and debugging programs are two of the most significant issues that require a deeper understanding for problem-solving than writing extensively so many lines of code. The results indicated that problem-solving behaviors were significantly correlated with students' self-explanation quality, a number of code edits, and prior programming experience.

Witherspoon et al. (2017) have conducted a study to evaluate the effectiveness of a programming curriculum for developing knowledge and skills related to CT using 3D visual robotics. This curriculum is designed to scaffold the use of technologies such as graphical programming languages and 3D virtual robotic simulations to produce optimal conditions for developing skills related to CT. The visual robotics was related to significant gains in pre- to post-test scores, with larger gains for students who participate in a scaffolding programming approach, within the context of virtual robotics. The simulations supported the development of generalizable CT concepts and skills that are associated with the increased problem-solving performance of students.

A brief summary of results and general outcomes from the above-mentioned studies which have utilized SGs is presented below in *Table 4-3*.

Table 4-3: A summary of results from previous studies which have utilized simulation games to support computational thinking instruction

Studies following game making approaches	Programming environment	Learning tasks	General outcomes for designing a SG to support CT instruction

Brennan and Resnick (2012)	Scratch	Designing and programming several interactive media such as the creation of simulations.	a) SGs need to not provide superficial gameplay and problem-solving contexts very easy and without purpose. b) When students are in the "flow" state via SGs, they are more likely to demonstrate in-depth learning on how to apply computational problem-solving strategies. c) With the use of SGs, students can utilize their skills related to analytical reasoning and critical thinking. d) Well-designed learning tasks can assist students not only in spotting and solving a problem but also on applying efficient and effective problem-solving design patterns.
Repenning et al. (2015)	AgentSheets and AgentCubes	Designing and programming SGs (e.g. a town with traffic etc.).	
Garneli and Chorianopoulos (2017)	Scratch and simulations	Designing and programming the function of a basic electric circuit by creating a simulation and by creating a video game in which players need to achieve specific scores in order to win.	
Studies following game playing approaches	**Programming environment**	**Learning tasks**	
Liu et al. (2017)	BOTS (3D prototype)	Programming and integrating behaviors in relation to a 3D robot's movements in problem-solving stages using students' programming experience and performance (debugging exercises).	
Witherspoon et al. (2017)	Games in *"Hour of Code"*	Programming and integrating behaviors of 3D virtual robotic simulations to produce optimal conditions for developing skills related to CT.	

Based on the above, the use of SGs can benefit players when interacting with in-game virtual elements which are not so simple but not also so difficult, and thus without having superficial gameplay. Within these contexts, learning is arising from users' active participation and engagement through interactive and immersive tasks in simulated problem-solving contexts. In general, students are able to learn by participating in simulated problem-solving learning situations and activities in a "constructivist" approach through instructive guided examples, scaffolding instruction, and reflection to their actions (Brennan & Resnick, 2012; Lye & Koh, 2014; Witherspoon et al., 2017). In specific, when students are in "flow" via a SG, they are more likely to demonstrate in-depth learning on how to apply computational problem-solving strategies such as analytical reasoning and learning-by-example. Therefore, there is a need to have a better understanding about the effects of SGs on students' problem-solving strategies not only in spotting and solving a problem but also on applying efficient and effective problem-solving design patterns (Liu et al., 2017; Repenning et al., 2015).

Chapter 5: *PIVB* - A proposed theoretical design framework

This chapter outlines widely referenced serious game design frameworks in terms of choosing, rationalizing and using the most appropriate one for the development of a SG to support CT instruction in programming courses. Since less are today known about how game playing can be associated with the development of CT and how fundamental programming concepts are supported, this chapter gives main reasons of using a SG to support the development of students' computational problem-solving strategies. Such a design framework derives mainly carried out on related works, thus aiming at addressing the "gap" identified by suggesting several promising features from the use of contemporary interactive environments which can support the development of a SG. Additional information is provided regarding the development of game playing conditions of a SG prototype with its architecture and illustrations in regard to its functionalities. The threefold purpose of this chapter is: (a) to propose a theoretical design framework called *"PIVB: Programming for Interactive Visual Behavior"* for the development of a SG; (b) to suggest design decisions made and criteria with design guidelines considered to understand someone how a SG can benefit students to think "computationally" in order to express and apply a logical way of thinking to a solution using fundamental programming constructs; and (c) to describe a design rationale on how in-game elements/features should be mapped in the direction of assisting students to use their problem-solving, logical and abstract skills so that solve real-world simulated (computational) problems.

5.1. Rationale

A theoretical framework provides a general representation of relationships among distinctive characteristics and key concepts which are resulted by previous theories and models. It can assist researchers to explore a phenomenon permitting them to intellectually transit from simply describing with a view of giving specific guidance with a set of principles that embodies a specific direction by which a chosen research approach for a topic will have to be undertaken (Rocco & Plakhotnik, 2009). Within such a context, a theoretical design framework for programming courses needs to include the following four steps:

a) the analysis of problem statement in relation to the determination of the learning objectives that programming courses require to be achieved,
b) the design principles and guidelines which can outline the development of a game prototype,

c) the utilization of elements and features selected in a design process to make the necessary modifications that considers students' pre-existing knowledge as well as their needs or demands, and
d) the implementation and prototyping process based on the in-game learning goals that someone can achieve using the learning content and the capabilities that such a game is developed in order to respond to the requirements of programming courses.

Up until now, a significant number of previous studies have widely proposed several game-making approaches for the development of games in which students start learning how to program their gameplay and core mechanics (Brennan & Resnick, 2012; Repenning et al., 2015; Werner et al., 2012). Particularly interesting are the results from those studies which have presented design frameworks associated with the correlation of cognitive thinking CT skills and programming relevant to game design and simulations. The increased interest to explore alternative ways in which design-based learning activities can have an impact and particularly in programming interactive media applications to support CT instruction have been broadly proposed. Assessing learning through game design is thoughtfully elaborated by several related works. To this notion, several frameworks were based on a strategic analysis description focused on how to correct students have tried to program their games creations. The most indicative are the following:

a) The *"three-dimensional framework"* is presented by Brennan and Resnick (2012) in order to assist students to articulate a design framework concerning computational concepts, practices, and perspectives via Scratch. This framework aimed at describing the processes of construction, and thinking design practices based on gaming creations of middle school students so that give CS instructors the opportunity to assess the development of CT.
b) The *"fairy performance assessment"* is proposed by Werner et al. (2012) in order to present students a way to perform well on a thinking design process via Alice. Such a framework was created to assess if students tried to understand their own narrative framework of stories by underlying their own programs and to elaborate on how accurate the existing programs are combined with instructions as design patterns. Students' thinking design process is related with the way that they do this correctly articulate their main narrative framework of a storyline associated with the correct place of instructions within a sequence of instructions (workable algorithms).
c) The *"scalable game design"* is suggested by Repenning et al. (2015) in order to provide a theoretical framework to be conceptualized students' object interactions related to CT design patterns via AgentCubes. Such a framework allowed students to dissect game descriptions and to articulate CT patterns they found how to apply. Each CT design pattern that was applied would not only describe the phenomenon of simulations that students need to describe but such a simulation

need also to include the appropriate programming constructs combined with instructions to be clearly identified how those patterns could be operationalized properly

To all the above studies, a project content analysis through artifacts is reflected more as a means to develop and evaluate skills related to CT through programming. Furthermore, all these previous studies have developed and suggested their own CT framework arising from their findings using different programming environments and learning activities in which students were usually the main software designers of their interactive games. For this reason, code documentation, information, and organization of programming constructs which can be integrated into the gameplay of students' creations seemed to be critical parts of learning. Such a process requires from students to think in a computational way so as to modify parts and features of interactive games on future use understanding their good code operation to their similar (or not) game-making creations.

However, game-making frameworks have quickly revealed several limitations about the appropriateness of games since there is a dearth of evidence on what specific features and elements should be integrated and how such features can be provided in a theoretical design framework (Grover & Pea, 2013; Lye & Koh, 2014). Computer game programming for CT integration especially for compulsory education through the game making interactive design applications has received considerable attention over the past five years, albeit there is little agreement on how students have properly tried to use their skills related to CT and programming concepts to encompass them inside their creations. Game-making approaches are entirely product-oriented, and thus there is provided less evidence in regard to the design process or design decisions taken by developing and programming different game projects, game mechanics and anything about what and why particular computational practices have been employed (Werner et al., 2014). For instance, 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 tracing code through exhibits with correct output for presenting functionality and readability of code commands and constructs correct sequentially or syntactically, students are focused explicitly on the declarative aspects of programming knowledge without perspectives on providing specific guidelines or features which seemed to assist them in designing and programming computer games (Denner et al., 2012).

Beyond the aforementioned difficulties, the lack of information on how a game is created and what components or design criteria are necessary to be taken under serious consideration on its design and development may have additionally an impact on the assessment of students' computational understanding based only on their final creations/concepts (Repenning et al., 2015; Werner et al., 2012). In other words, it is unclear what students as game developers of interactive applications inside games were able to do on their own (as opposed by getting help from other people or other projects), the extent to which they have

tried to understand the concepts that they utilized to complete their creations which many times are associated with particular code blocks usage, and lastly if they were able to articulate his/her computational problem-solving strategies (Brennan & Resnick, 2012; Werner et al., 2014). Thus, even less agreement about what problem-solving strategies can be properly applied into the code for assessing students' learning performance and which of those strategies are associated with the development of games in which users can think and practice "computationally".

Due to the surge of game-making approaches as the most "mainstream", an alternative and certainly less explored to support CT instruction is learning how to program through game playing. In specific, existing works either by using VPEs (e.g. Garneli & Chorianopoulos, 2017; Repenning et al., 2015; Webb & Rosson, 2013) or 3D prototype games (e.g. Liu et al, 2017; Witherspoon et al., 2017) have connoted that students' learning performance is associated with problem-solving patterns and behaviors integrated into visual elements in which students try to develop and apply their computational problem-solving strategies. In their meta-synthesis review about game-making learning approaches, Denner et al. (2019) have advocated that are existed conflicting findings from previous studies which cannot provide any serious evidence in regard to their generalizability. More specifically, the generalizability of findings is limited because of lacking data to investigate whether any potential benefits can be extended within school contexts. Also, a lack of data is revealed which cannot thoroughly indicate more properly the conclusions about game mechanics using different programming knowledge in order to be more useful for students to learn how to use CT skills before starting to code. The same authors have provided two important reasons. The first is a lack of studies to describe instructional conditions and means with no conclusions that can be drawn about the benefits of game making approaches. The second is the lack of additional detail about the methods and procedures made in K-12 education that indicated by few studies' findings.

A substantial number of previous studies (Chao 2016; Kafai & Burke, 2015; Werner et al., 2015) has suggested that skills related to CT and programming can be transferable using computer games. Literature reviews in the field of CT for K-12 curriculum have also come to the statement that it is still unclear the effect of computer games and more specifically of SGs to support CT instruction in programming courses (Burke & Kafai, 2015; Lye & Koh, 2014). This statement is still intensifying more due to the lack of design frameworks and requirements for the creation of a SG. Lack of essential guidelines, characteristics, and features that a theoretical design framework for the development and creation of a SG may prevent game educators and developers to justify their claims whether a computer game has (or not) an impact on students' learning performance and outcomes (Grover et al., 2015; Lye & Koh, 2014). Moreover, previous reviews in the field of CT instruction through programming courses have come to the statement that a computer game needs to be developed by using a theoretical framework having specific design guidelines and criteria in order to assist students develop and demonstrate a wide range of CT skills related to cognitive

thinking and programming. In their review analysis, Grover and Pea (2013) have mentioned that a theoretical design framework needs to be proposed in order to inform computer game designers, educators and scholars on how to develop and program computer simulated problem-solving tasks using SGs either by using new interactive environments or by combining already known design features and characteristics of the most well-known interactive environments. Also, Lye and Koh (2014) have noticed the need to propose directions towards the use of a "constructivist" framework for the creation of a SG to support the demonstration of skills related to CT and programming.

In addition to the above, prior works in the field of CT instruction (Howland & Good, 2015; Liu et al., 2017; Witherspoon et al., 2017) have concluded that there has been relatively little research showing how a game playing framework can be associated with the alignment of skills related to CT and fundamental programming concepts and constructs in an effort to support the expression and implementation of students' computational problem-solving strategies. Consequently, there is a need to have a better understanding about the effects of SGs on students' problem-solving strategies not only in spotting and solving a problem but also in applying efficient and effective their problem-solving design patterns (Liu et al., 2017; Repenning et al., 2015). For this reason, it is appropriate to propose a theoretical framework with specific design features and characteristics which can facilitate the creation of a SG and support the development of students' computational problem-solving strategies to contribute to the field of CT integration through game playing in programming courses.

To address the aforementioned "gap", the current chapter suggests a theoretical design framework called *"PIVB: Programming for Interactive Visual Behavior"* in order to propose and present specific design guidelines and criteria for the development of a SG. Such a SG can include several simulated real-world activities with various learning purposes, such as training, analysis or prediction either of specific digital objects or in-game conditions that students can handle and/or manipulate using fundamental programming constructs. A proposed SG can become an effective "tool" for learning computer programming as it can support how fundamental programming constructs can be associated with skills related to CT and more importantly to be presented as a valuable solution for game playing approaches. Therefore, the twofold purpose of this chapter is:

a) to describe a theoretical design framework with specific characteristics and guidelines that can be utilized for the development of a problem-solving environment displayed via a SG.
b) to elaborate a design rationale on how in-game user design features and elements need to be mapped in the direction of helping students to use their problem-solving, logical and abstract skills for the analysis of subparts of a simulated (real-world) problem.

The proposed theoretical design framework can inform instructional technologists or educators and game software developers on how design features and elements should be used in order to support and

assist players to link abstract concepts with the concrete game-based learning experience. Such an effort can give a better understanding of the impact of SGs in programming courses and CT instruction.

5.2. Computer game design frameworks

During the last ten years, a significant number of theoretical design frameworks for the development of computer games to support different learning subjects have been proposed. The most suggestive and well-documented are briefly presented as follows. First, Garris et al. (2002) have developed an instructional game-based model that illustrates how players can be engaged when they play SGs. Players need to make judgments based on evaluations and modification of their behavior within a game cycle that is resulted inside gameplay that continues to exist within a repeated "*judgments – behavior – feedback"* cycle, as they can observe and manipulate in-game conditions. Such a process is achieved by separating instructional content from the game characteristics.

Second, Kiili (2005) has provided a framework in favor of connecting gameplay with empirical learning, thus having a relationship with the Kolb's cycle called the "*experiential gaming framework*". His framework emphasizes the importance of examining flow experience before its final design and creation of an educational game. The focus was on the challenges responding to the player's skills based on the feedback that they can receive, and the sense of controlling in-game events, having specific contexts with clear and achievable goals. This model describes learning as a circular process through direct experience in a digital environment that someone can play and practice.

Third, de Freitas and Oliver (2006) have proposed a four-dimensional framework that entails pedagogical considerations, learner specifications, context and model of representation for helping instructors and educators to evaluate the potential of games within different instructional formats focused on four dimensions. The first is the *context* in which learning by playing tasks take place. The second is *learner or learner group* taking under consideration their learning background, styles and preferences. The third is the *internal representational world* – or "diegesis", including the mode of presentation, the interactivity, the levels of immersion and fidelity to practice players' tasks using serious games and simulations. The fourth is the *process of learning* that promotes players' reflection upon methods, theories, and models are used to support learning practice. This framework is considered as an extended methodology that could be used to evaluate computer games and their appropriateness for learning purposes.

Fourth, Good and Robertson (2006) have presented "*CARSS"*, a framework for carrying out learner-centered design with children. It is used to suggest design intelligent and "non-intelligent" learning environments alike specifying the initial parameters and constraints of the project in such a way so that someone can determine the level of child involvement which may be more suitable. It attempts to provide a fully inclusive design framework comprises five components: context, activities, roles, stakeholders, and

skills. It also offers a comprehensive set of issues to consider when planning to use a child-centered design approach in a fully-fledged participatory design approach.

Fifth, Ryan and Siegel (2009) have analyzed the process of embodied learning by observing the phenomenon of a breakdown in players' use of video games for examining gameplay. The *"Breakdowns of Interaction"* framework is focused on the implications gained from the player's experience when they usually fail to apply strategies, and reasons are only focused on missing characteristics that a gaming environment may have and thus not assisting them to take on several decision-making steps. The four-part framework is constituted by dimensions, such as perceiving the environment, developing a strategy, taking action, and meaning-making. In particular, the same authors present four main dimensions of breakdown, though they do not make a point of indicating which those are that can impact interaction or illusion.

All the above frameworks have paid their attention either on the emergence of specific elements/features that a game needs to have or in the different aspects of educational design. What really seems to be lacking is an educational approach to game design in regard to key game principles, design criteria and educational goals that a game can provide within educational contexts since only proposing game guidelines cannot alone lead to use most of such frameworks to design and create instructional game prototypes. For example, de Freitas's and Oliver's (2006) framework has given more systematic effort to deepen the balance between game design and target education; however, it cannot support instructors to identify which type of games would be applicable for specific learning objectives (Robertson & Howell, 2008). Regarding the use of the "*experiential gaming framework"*, Kiili (2005) has noticed that it does not provide a means for a whole game design project but only it links educational theory and game design. Additionally, *"Breakdowns of Interaction"* framework (Ryan & Siegel, 2009) has been implied to the most breakdowns stem from interaction issues which can lead to further breakdowns in illusion; however, it is not clear why some breakdowns end up affecting involvement and others do not (Iacovides et al., 2014). Another significant point of view is that frameworks such as the "*CARSS"* (Good & Robertson, 2006) have addressed several constraints about their appropriate use for game design since many times game designers at a younger age may not on their own develop and create computer games or may not always be possible to have enough time and budget to develop their games properly. Furthermore, Robertson and Howell (2008) have relied on having game designers the appropriate background for the development of computer games following some of the above frameworks. This issue may prevent some other instructors and designers in identifying and proposing which games would be applicable to different learning subjects/domains and how to create such educational games.

Despite the fact that previous efforts provide guidance and assist the work of game designers, most of the above design frameworks are focused on theoretical underpinnings with general principles which have not been widely well-founded and have not provided any empirical evidence to investigate their

appropriate use in specific learning subjects. Thus, a lack of a clear demonstration of how to produce motivational and pedagogically effective games is arising (Good & Robertson, 2006; Robertson & Howell, 2008). Although there are many other serious game models in the literature, research in GBL often reference the Garris et al.'s (2002) framework as an ideal one to show how the development and creation of educational SGs can assist students' participation as a way to illustrate their learning outcomes. It is an instructional game-based model focus specifically on the development of SGs and it still remains as the most widely referenced and accepted work in the literature. In addition, the vision of creating a SG such as the popular *SimCity* to support CT instruction that Kafai and Burke (2015) comes in align with the game design framework proposed by Garris et al. (2012), as a quite instructive and appropriate example of an environment for someone who wants to learn by playing in simulated problem-solving tasks.

According to all the above, Garris et al.'s (2002) framework can be considered as appropriate since its design principles and features behind the development of problem-solving tasks can make game designers think about how players can try:

a) to identify learning objectives for the main problem, handle its subparts and propose a solution;
b) to recognize a way of understanding how they to think before start coding based on their judgments and behaviors, and
c) to achieve specific learning objectives that may support students' outcomes that need to plan in order to solve sub-tasks of a simulated problem using cognitive thinking skills, such as critical and logical thinking.

For this reason, an attempt is proposed by outlining and describing on how design guidelines with specific features should be reflected on Garris et al.'s (2002) framework in order to manipulate another a theoretical design one that can be more essential for the development of a SG supporting the CT instruction.

5.3. Design decisions

Over the last few years, various computer games and specifically SGs have been developed following game-making approaches in programming courses, but limited evidence is provided in regard to which characteristics and features are the most important for any potential improvement on students' learning performance (Garneli & Chorianopoulos, 2017; Werner et al., 2015; Witherspoon et al., 2017). SGs can provide more engaging tasks for the introduction of students inside a digital environment for knowledge acquisition with the appropriate functions that can be familiar to players. It is important to mention that delivering and organizing any learning material into in-game stages, is a process that accommodates students' needs and meets their demands (Garris et al., 2002). Providing students with specific problem-solving learning tasks inside a SG can be crucial to support also informal (in-class) or informal (outside the class) instructional approaches. Students can use of SGs for learning how to think "computationally" and

practice into code their solution plans in simulated problem-solving (real-world) tasks (Chao, 2016; Liu et al., 2017).

A set of important features and design decisions that game designers should consider for the development of SGs to be included the following (Garris et al., 2002; Prensky, 2007):

- the association of learning objectives with in-game goals in order to provide all players with the anticipated outcomes.
- the relevant learning materials can assist players to achieve in-game learning goals and increase (if it is possible) their learning performance.
- a specific scenario with specific learning goals needs to include visual characters/elements that all users can choose in order to achieve in-game goals.
- the awards and punishments to all in-game tasks need to be based exclusively on players' outcomes and achievements so as to accomplish specific learning goals.

Regarding the requirements to support CT instruction, since SGs are increasingly utilized in programming courses, knowing how game designers can take design decisions to develop such computer games is becoming one of the most imperative issues. Thus, it is beneficial to propose a theoretical game-based framework for CT instruction, with the purpose of considering (Kafai & Burke, 2015; Liu et al., 2011):

- what are the game characteristics that need to be integrated to support students' engagement and participation;
- under what instructional contexts students need to have the instructor's assistance when playing a computer game in favor of recognizing if they really tried to develop and use skills related to CT; and
- how students can develop and/or use skills related to CT so as to solve simulated (real-world) problems built into workable plans and algorithms with precise instructions.

In addition to the above, there are specific requirements and design decisions that imply on understanding why and how certain design decisions with specific game principles can add several prominent learning conditions to support CT instruction. Thus, it is first of all necessary to investigate how principles should be mapped to design a SG that can facilitate flow learning experience through problem-solving in-game tasks. To this notion, the development of a SG should support players not only to understand the syntax and semantics of a programming language but also to observe and recognize its effects and consequences for solving (real-world) problems (Davies, 2008). Therefore, the following three design decisions need to be taken into consideration:

1) *Decomposition and formulation of the main problem (abstraction):* Decomposing and formulating a solution to a problem are associated with "abstract conceptualization". Abstract conceptualization

is everything that makes sense inside (digital) environments including an understanding of the relationships between events and humans made without unnecessary information. For example, if students can understand how to use learning material (e.g. objects, elements or programming tools) in order to achieve a learning objective, then it is recognized that such an environment integrates successfully the appropriate materials to extend what they have already know about a learning topic and what they can gain if achieve certain learning in-game goals (Garris et al., 2002). It is the first decision that designers should consider since it is crucial the use of a SG can allow students to conceptualize their actions. Such a process can be supported either verbally, e.g., by trying to formulate a question such as *"How can I solve this problem using in-game elements to work for this effort?"* and/or visually, e.g., by analyzing their innate thinking solution plans as diagrammatic representations or in texts written in natural language (pseudocodes) in place of identifying the in-game objects' behaviors or relationships between users and in-game elements. Two are the most appropriate characteristics that can support "abstract conceptualization" and assist students to achieve their cognitive thinking process that leads from the problem formulation to solution expression. The first is *visual thinking* so that students can organize their thoughts, describe object interactions and improve their ability not only to think but also to communicate them (Lye & Koh, 2014; Repenning et al., 2017). The second is the use of *spatial metaphors* which can support critical thinking and problem-solving skills, giving to all users the opportunity to organize information visually (Repenning et al., 2017; Roman-Gonzalez et al., 2017);

2) *Description and expression of a solution (automation):* The second design decision refers to the expression of solution plans that include the alignment between (correct) computational rules and concepts with programming constructs. To be considered as successful and assist students such an effort, an understanding of in-game events and their association with the entire storyline, concepts, elements and goals with logical reasoning is required in order to use and communicate their solution plans more effectively. An example is on how students propose a solution based on their natural perceptions and if its rules/concepts related to programming (e.g., *"if an element moves…then…"*) can be transferred into code using programming constructs to observe the consequences of those instructions. Players need to focus on in-game visualized behaviors that are created correctly by programming code blocks consisted of motion commands (command blocks) and programming constructs (control flow blocks) that can be integrated into objects by composing programming constructs as design patterns in order to propose executive solutions to a problem. For this reason, previous studies (Good & Howland, 2016; Grover et al., 2015) have addressed several syntactic challenges of end-user programming. To know how to apply solution plans into the code, students

need both to understand the syntax and semantics of programming concepts into executable programs (programming knowledge);

3) *Execution and evaluation of design patterns (computational problem-solving strategy analysis):* The third decision is reflected in the assessment of how to correct students' problem-solving thinking strategies can become. It is worth noting to evaluate whether such strategies can be transferred with the expression of unambiguous instructions for solving problems in natural language or diagrammatic representations into workable plans and programs using fundamental programming constructs (Grover et al., 2015; Lye & Koh, 2014). To assess the correctness of students' thinking strategies into the code, a SG needs to be designed in order to support a scaffolding instructional approach. Therefore, from the students' side, each stage with different levels of difficulty needs to have the appropriate elements and integrated materials so as to support them thinking before starting to practice into code their solution plans using programming constructs as design patterns. From the instructor's side, a SG needs to have simulated problem-solving tasks to evaluate the appropriateness of those patterns as essential for solving problems and examine so that correct (or if it is not necessary to not make any actions) students' solution plans to ensure their appropriateness (Repenning et al., 2010). For example, if the CS instructor can provide explicit educational instructions and extra feedback on the composition of programming plans with visualized program tracking in gameplay mechanisms, students can more deeply understand how nested control flow blocks work and what the subsequent effects of the chosen actions are (Werner et al., 2014).

According to the above, the following three-goal examples are regarded as appropriate to design and create a SG for students to articulate and transfer their thinking solutions into workable plans and algorithms:

a) *Integration of the learning material within the game:* A way of formalizing knowledge in simulated problem-solving contexts during gameplay with a more natural intuitive modality for user interaction within a game is imperative. The representational fidelity of in-game visual metaphors that can be projected can infer and predetermine a game designer to specify algorithmic rules corresponding to specific movements that are the most appropriate to be done by each player (Witherspoon et al., 2017). For example, the visual metaphors of geometric shapes can support students to learn how to program with tasks related to the conceptualization of algorithmic rules through logical reasoning in align with programming constructs that can be projected in simulated (real-world) problem-solving contexts (Papert, 1980; Roman-Gonzalez et al., 2017).

b) *Transfer from tacit to concrete thoughts using computational concepts:* Understanding of game events and having the ability to describe events can be a good starting point that would allow

students to be engaged with basic computational concepts (Good & Howland, 2016). For instance, evidence from previous works (Chao, 2016; Grover & Pea, 2013; Werner et al., 2015) has mentioned that students need to understand first of all conceptually what problem(s) they will solve using a computer game in order to propose and present their solution plans. Students may be able to transfer and use a game's user interface design features into their own contexts by recognizing that problem-solving within such contexts is regarded as an activity that can be meaningfully and seriously approached in a playful attitude. Therefore, in-game activities should allow users to describe the learning situation in which they attend and explicitly link their actions during gameplay with the development of skills and concepts related to CT. The reflective observation of the concrete experience assimilates abstract conceptualization without remaining tacit so as to facilitate students' understanding or how and why can use specific computational concepts and constructs in two ways (Brennan & Resnick, 2012; Repenning et al., 2010):

i. by decomposing abstract representations of the main problem to articulate a way in an effort of formalizing tacit knowledge within specific and reliable contexts, and
ii. by conceptualizing abstract logical thinking during gameplay to invent and formulate an idea or a concept so as to provide design patterns for testing and debugging a solution plan.

c) *Transform students' concrete thoughts into formal logic and analysis of a solution into code:* The student's progress through in-game activities requires the concreteness of solutions by transforming a cognitive thinking process for solving a problem into code. For example, a SG can provide an intuitive-natural modality on its GUI design features and elements for user-interaction tasks (Liu et al., 2017; Witherspoon et al., 2017). Thus, in association with the above, users can articulate and transfer from tacit thinking to more concrete that can be transformed into the code so as to develop and apply their computational problem-solving practices (Mouza et al., 2016; Werner et al., 2014). A suggestive way to support such a process is the use of a visual palette such as those of Scratch or S4SL which can eliminate split attention of code syntax and users can focus on goals of solutions that are applied as results of computational problem-solving practices (design patterns that can include code blocks).

5.4. Design principles and guidelines

The current sub-section provides information regarding the design process of the proposed *PIVB* theoretical design framework focusing on CT instruction through computer programming (Pellas & Vosinakis, 2017a). An important step that needs to be realized is the establishment of a SG's infrastructure, to support such an effort. It is essential to initiate the design of the game by studying its characteristics and features that need to be instantiated following specific design guidelines from an instructional game

framework. Garris et al.'s (2002) framework are suggested as one of the most appropriate to be explicitly illustrated a SG's design features and/or elements for active learning processes. It emphasizes both players' motivation and process aspects which are associated with skill-based learning outcomes providing several motivating and challenging goals. Garris et al. (2002) have also proposed the use of SGs which can present embodied problem situations fostering players' problem-solving ability and thus provide conditions to experience within a scientific discovery process. Players are engaged in simulated (real-world) tasks with features that include rules and strategies allowing the exploration of a game environment for achieving specific goals and protecting them from the more severe consequences of mistakes (Garris et al., 2002). Such a SG can promote in-depth learning on users' actions while they can interact with its elements and objects through problem-solving tasks.

Garris et al. (2002) have tried to categorize specific game characteristics to support such a SG, such as fantasy, rules/goals, sensory stimuli, challenge, mystery, and control. The same authors have also proposed specific 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 presented below:

a) the user's motivation and persistent engagement (P1)
b) the clear and challenging goals (P2)
c) the system's feedback on user's actions (P3)
d) the scaffolding process (P4)
e) the debriefing process based on students' skill-based learning outcomes (P5).

An indicative way to support CT instruction using a SG is by extending the instructional game framework of Garris et al. (2002) so as to propose a theoretical design framework with specific design guidelines and features/elements that can assist students in developing and using skills related to CT. The design guidelines that are proposed by Pellas and Vosinakis (2017a) are the following.

- 1st guideline (G1): *Motivating students to participate in active learning tasks*- While every computer game can be motivating per se, there should be existed several subparts of the main problem with clear and challenging tasks through interactive game-based conditions. Players need to achieve specific in-game goals in order to start analyzing, creating, applying and evaluating (debugging) their proposed solutions. Such a process will allow them more properly to think logically and critically about 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 when players start to play a game. Data visualization and representation need to support the operation of learning activities in which players can participate. If players are engaged and involved in several tasks knowing what precisely have to do, and they will also devote more time to actively pursuing to other 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 players' everyday life, but it should also provide prompt feedback during the run-time of their actions, visually and/or acoustically. Such a process will assist players to conceptualize better their problem-solving strategies into a concrete game learning experience (P3);
- 4th guideline (G4): *Facilitating the development of computational problem-solving strategies through a scaffolding process*- A game may allow students to develop their problem-solving strategies into programs. In other words, before finalizing a solution into the code, players need to think about how to program by combining relevant subprograms together and how all its components corresponding properly to each of the given simulated problem-solving tasks. For example, through a game-playing approach, students need to know how to integrate behaviors into objects by programming with the purpose of having interactivity with each other. In such a fading scaffolding teaching approach, CS instructors should demonstrate how such subprograms can be constructed. Even if frustrating tasks by playing a game are observed, the CS instructor needs to guide the students by prompting them with questions on their problem-solving processes further when they play such a game (e.g., "*what is the main reason of putting that command there?"*) (P4);
- 5th guideline (G5): *Applying design patterns to propose an answer for a problem question*- Players' embodied experiences/ideas need to be simulated through actions that are performed on the subparts of the main problem. Assisting players to understand how to transfer of behaviors in different objects is considered as a crucial process to recognize how these actions will (or not) solve a problem. In this perspective, they need to propose design patterns to execute their thinking solution plans into code by applying their own programs using programming constructs (e.g., repetition or selection). Such an approach can foster computational practices and perspectives because students need to think about how to use properly fundamental programming constructs and/or instructions to present programs and observe the consequences of those constructs and instructions inside the SG (P5).

Figure 5-1 illustrates the proposed framework and game guidelines, which can be designed as an integral part of game characteristics to support the design guidelines (G1-G5) that have been described above. According to Garris et al. (2002), the use of the following SG elements can support CT instruction:

a) *Fantasy*: A SG should offer visual metaphors from real-world processes that permit users to have experience of a process/system with phenomena or tasks which sometimes cannot be done in real-world settings. For example, students having specific roles can learn how to program by integrating behavior in visual objects responding to events or issues and actuate controlling of such objects without having spatial-temporal constraints or payments about technical equipment.

b) *Rules/Goals*: The rules describe goal structures of the SG. A game designer should predetermine specific game mechanics that would help users who lag while playing a game; it could include bonus/subsidies or for their poor performance to have a scoreboard with punishments. If players can understand and specify in-game rules, they may use and express relevant basic computational concepts correctly to propose their solution plans.

c) *Challenges*: A SG should include several stages which can have progressive difficulty levels, multiple goals, and appropriate information to ensure certain learning outcomes. Performance feedback and score-keeping game features can allow players to track progress toward desired goals. A challenging task is created by issues, like time pressure and opponent play in order to understand under which conditions players can win (or lose) points and take some awards. Players need to collect information from in-game digital objects/elements in order to understand what is correct to do or what is not. Using challenging tasks inside a SG, CS instructors need to assist students to use their virtual characters and then start to apply their computational practices using design patterns which are aligned with the pre-defined game rules and features.

d) *Mystery*: Simulations that incorporate these features become more game-like and allow players to explore in-game events/conditions. For example, in a game-based environment, players need to be engaged in specific simulated tasks having user design features and elements, such as role-playing and scoring that are not presented in real-world tasks.

e) *Control*: A sense of freedom using objects and elements inside a SG can allow players to select/refine their problem-solving strategies, manage their activities, and make decisions that can directly affect their outcomes and/or achievements. Such a sense, beyond the players' engagement in each learning task, gives the ability to explore, recognize the problem space, and propose alternative solutions.

f) *Sensory stimuli:* A computer SG should include sound effects, visual objects with representational fidelity, and media sources. Such an environment should not distract the stability of players' sensations and perceptions, but it should also allow the user to have a more reliable experience. A certain example is how a train can pass over a railway that may require the integration of behaviors by programming the former in order to understand some simulated phenomena, such as gravity, imitating its correct instructions/movements as in the reality.

According to all the above, the proposed theoretical design framework for game playing is developed specifically for designing SGs that can be associated with specific learning tasks to support CT instruction. *Figure 5-1* shows how players are engaged in in-game tasks, which will generate their desire to be engaged through attractive learning scenarios, such as role-playing (Stage 1). All those tasks will assist students to develop computational problem-solving strategies and will be able to produce a set of learning outcomes

in several learning tasks (Stage 2). The learning objectives will be achieved and evaluated if the in-game experience can support CT instruction (Stage 3). It is important to mention that players need to increase their cognitive thinking skills if in problem-solving tasks can provide *"abstract conceptualization"* comprised *visual thinking* and *visual metaphors* so as to provide their solution plans to each stage which can be different on their levels of difficulty, when they have a progress inside the game (Stage 4). In this demand, segmented four stages (S1-S4) are aligned with the proposed game design guidelines (G1-G5). The first stage (S1) is aligned with the 1st and 2nd guidelines and the second stage (S2) with the 4th guideline. The third stage (S3) is related to the 5th guideline. Also, the fourth stage (S4) is aligned to the 3rd guideline for the system's feedback cycle.

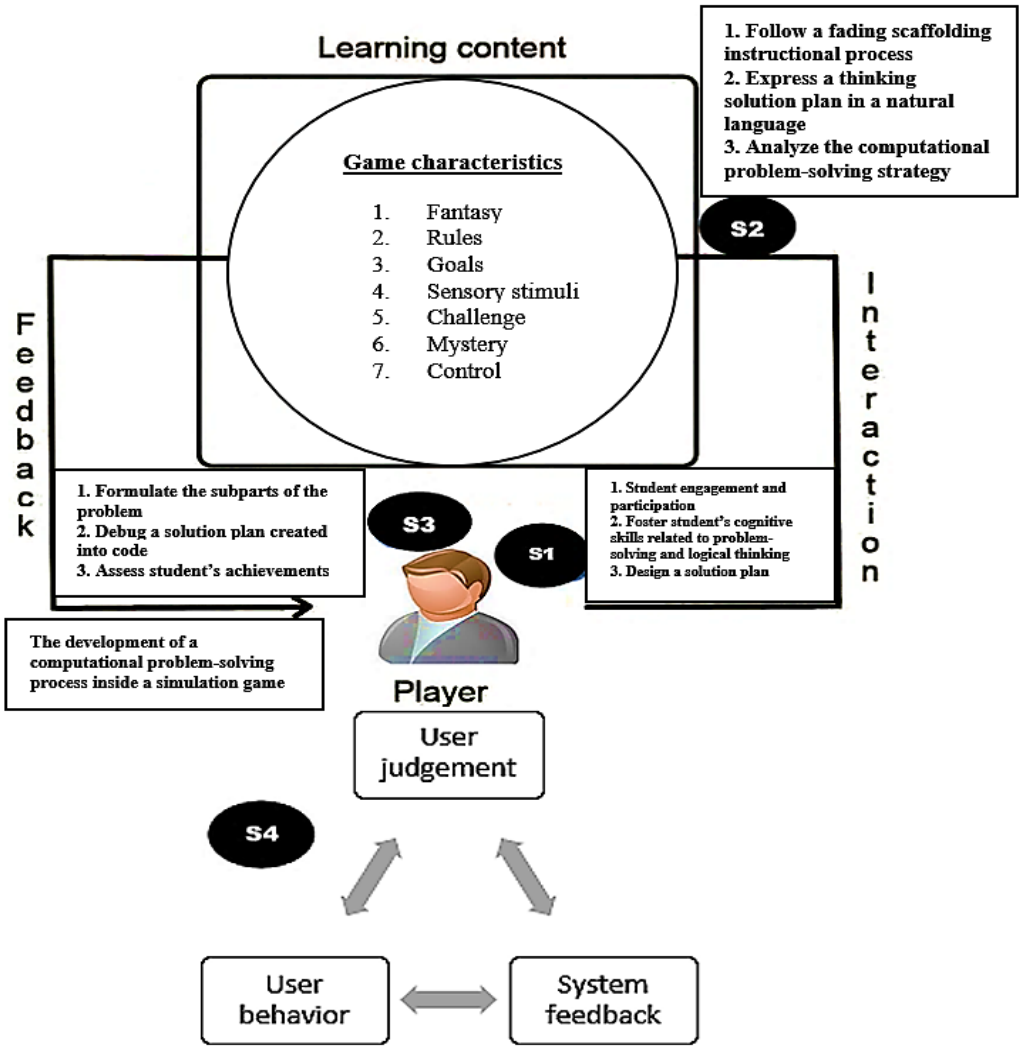


Figure 5-1: The illustration of the proposed framework

To summarize, a SG prototype to support CT instruction is composed of the following parts:

a) the simulation of a real-world problem-solving situation and functionalities in fading scaffolding processes for supporting users' roles (students and instructor),
b) the visual metaphors that players can conceptualize into algorithmic rules that may assist them to think logically and methodologically using CT before start programming their solution plans for each subpart of a problem, and
c) the use of a programming "tool" to eliminate split attention in code syntax, focusing on users' solutions that applied as results of their computational problem-solving strategies.

Based on all the above, a theoretical design framework is presented for learning how to program constructs related to CT through game playing. Further to this, this chapter establishes the premise that a SG can be designed to encourage players to think "computationally" from problem formulation to solution expression through specific problem-solving tasks. The proposed theoretical design framework with specific guidelines and principles is based not only on the operational level of abstraction and skill acquisition of skills related to CT but also on the appropriate use of fundamental programming constructs. The mentioned issues may improve the learning process in the following two aspects:

a) by using visualization or animation so that assist students to understand only the most important information (abstraction), and
b) by exposing students to playable conditions that allow the exploration of in-game objects/elements which have specific core mechanics so as to propose workable algorithms, and then execute those instructions and constructs into code (automation).

5.5. Essential components and design criteria

A wide range of studies has already proposed several design principles and characteristics for the development of a computer game using interactive environments that can support CT instruction. In specific, past efforts (Good & Howland, 2016; Repenning et al., 2017; Werner et al., 2014) have suggested a variety of design features and elements fostering CT skills development with *visual thinking* and supporting problem formulation in applications, such as evocative spatial metaphors (e.g., geometric shapes) which are offered to be an alternative and worth noting option for boys and girls inside game-playing contexts. To be considered as appropriate an interactive environment, it needs, first of all, to have some essential components that can assist *visual thinking* for information processing in order to be provided a better understanding of knowledge acquisition that is reflected from a problem's description to solution expression (Repenning et al., 2015). An apparent example is the use of a mind map tool that includes several geometric visual objects (Papert, 1980). Players, in this vein, are able to organize information inside visual contexts which can be more easily recognized. Thus, they can start thinking through a visual process arising by a sequence of steps about how visual elements and objects with the intension to be programmed properly

by following specific instructions (Kafai & Burke, 2015). Such a process is associated with *"abstract conceptualization"* and game mechanisms of a SG, including visual elements, interaction among objects with players, and rules that provided in align with specific goals considered as essential features (Garris et al., 2002). The use of visual objects and elements can be considered as essential in such a conceptualization approach to support CT instruction. For instance, prior works (Kafai & Burke, 2015; Lye & Koh, 2014) have advocated that such characteristics and features can assist players to have more concrete experience through spatial abstractions, which can predominately pave a pathway from problem formulation to solution expression. Such components are expected to support spatial reasoning, due to the fact that players need to understand the logical relations among visual objects/elements so as to use inductive and abstract reasoning thinking (Ambrosio et al., 2014; Román-Gonzalez et al., 2017).

Another significant point of view can be the use of "abstract simulations". An abstract simulation is related to the visual objects/elements and a variety of abstract icons (e.g., numerical domains or dots) which are integrated inside a game. It can be used to eliminate the complexity of any unnecessary information from the gaming system and can assist players to understand any projected relationships in order to succeed an active experimentation through a more concrete experience (Garris et al., 2002). Abstract simulations can assist players to understand the concepts by taking advantage of the formation of spatial knowledge representations which can support problem-solving learning tasks. For example, Román-Gonzalez et al. (2017) have also pointed out that visual-spatial abilities can be enhanced through various activities when students (boys and girls) can try to give commands/instructions and/or observe visually the consequences of their actions (outcomes). In such a game, instructional game designers need to consider as essential two things. The first is the spatial orientation that involves in-game contexts related to 2D or 3D objects/elements which can be visible to players. The rotation of mental representations is determined using visual objects or images from certain viewing angles (Ha & Fang, 2018). To this notion, spatial navigation and exploration on how features elements/objects are integrated inside a game by considering the player's awareness need to be provided in two aspects:

a) by providing visual clues for spatial navigation around a digital environment with specific game objectives, and
b) by giving to each player several opportunities to be engaged through in-game activities with a view to carrying out a set of quests and explore interesting areas so as to gather information for solving certain tasks.

From a theoretical and design perspective, it is imperative for instructional and game designers, firstly to answer a specific question: "*How can a SG meet the design criteria that involve a wide set exercises in order to acquire knowledge and skills related to CT?".* While most SGs seemed to be motivating and interesting for each player, one important topic is how to apply their knowledge inside the game and how

they can gain knowledge using their cognitive thinking skills. Four design criteria (C1-C4) that are important in meeting the aforementioned components and requirements are indicated below and depicted in *Figure 5-2*:

a) Learning content: The development of a SG needs to address important concepts or content related to computational problems separated in several stages with different levels of difficulty. Logical reasoning of players' actions needs to be assisted by in-game elements and objects. Also, unambiguous instructions from the CS instructor can be also important than a collection of random events without meaning. The interaction with in-game visual elements facilitating players to receive feedback about the consequences of choices that they have made (C1).

b) Gender equality: The development of in-game mechanics (e.g. colors or images) needs to fit on the socio-cognitive level of all players regardless of gender. Players need also to have the chance to choose their own visual representation inside the game and they should know exactly at the beginning and before start playing their specific roles inside it. Additionally, trace balancing among quests and goals to all in-game stages need to be connected from simple to more complicated tasks, in which each player needs to navigate inside it, and explore objects/elements so as to achieve certain learning goals to each stage properly without causing any gender biases (C2).

c) User interface design features and elements: The user design features and elements need to support a specific storyline and assist players to understand the spatial navigation inside a SG. Free exploration and accessibility to each stage should have the appropriate features and elements which may motivate players. Players' actions need to be aligned with learning outcomes in order to be accomplished certain in-game goals using specific tools. For example, the user interface features and elements can assist players not only to observe a problem-solving environment and its subparts but also to have a tool for programming their solution plans (C3).

d) Awards and punishment conditions: The in-game awards and punishments need to be based only on the demonstration of skill-based learning outcomes so players can understand how to achieve specific learning objectives. The alignment of in-game goals with the learning objectives can assist players to consider a clear indication about what they need to accomplish to receive awards or punishments in case of avoiding or not being able to complete the in-game goals. Within such an effort can be accessed effectively the knowledge gained from each game task in specific time-limited tasks based on students' skill-based learning outcomes in order to receive awards or punishments (C4).

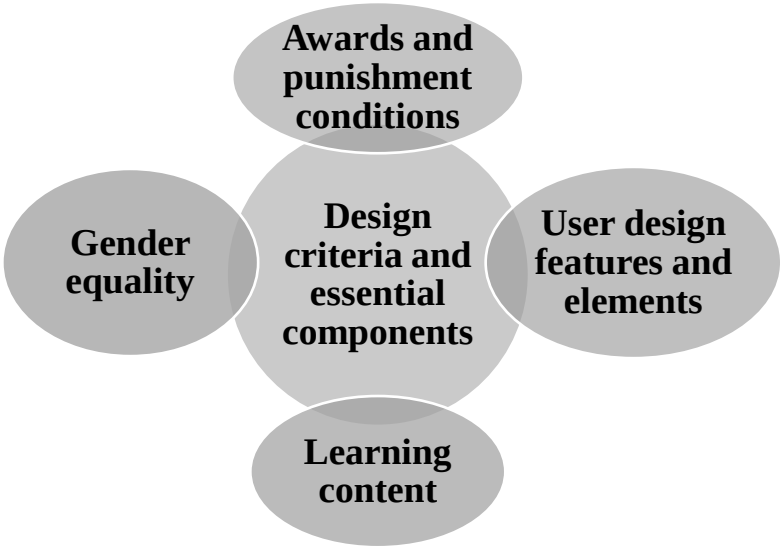


Figure 5-2: The alignment of game components and design criteria

From a practical and implementation perspective, considering that computational problem-solving strategies of boys and girls require the use of skills related to CT and programming for solving real-world simulated problem-solving tasks, it is crucial to suggest specific design requirements and tools that can assist students to apply their problem-solving strategies. For example, the analysis from results of previous literature reviews (Grover & Pea, 2013; Kafai & Burke, 2015; Lye & Koh, 2014) has recommended the development and use of simulated problem-solving tasks using SGs. More specifically, the analysis of the literature review from Grover and Pea (2013) has suggested the development and use computer simulated problem-solving tasks using SGs either by developing new interactive environments or by combining already known design features and characteristics of the most well-known interactive environments. Also, in their review study, Lye and Koh (2014) have proposed design guidelines and directions towards a constructivist (thinking-doing) problem-solving learning approaches in a SG created either in VPEs or in 3D VWs. Additionally, Kafai and Burke (2015) have recommended the connection of serious gaming opportunities in a simulated "world" that can be created in interactive environments, such as *SimCity* or *Scratch* to design and propose a new one in which can be created simulated problem-solving tasks that can be relevant to the needs and demands of boys and girls. Thus, it is appropriate to investigate if the design features and characteristics either from the category of VPEs or 3D VWs can facilitate the creation of a computer game to support the development of students' computational problem-solving strategies. Thence, it is crucial to propose a SG that can be created using interactive environments with specific design requirements and guidelines to support the demonstration of skills related to CT and programming in which gender equality is perceived to all learning stages. Such a process can allow educators and scholars to

understand better the effect of a computer game on gender equality in programming courses and on the possible improvement (or not) on their learning performance.

To date, many games created by using interactive environments either from the category of VPEs or 3D VWs have been extensively developed for programming courses. Due to a wide range of features/characteristics and tools, both platforms are indicated as the most relevant to foster CT instruction (Grover & Pea, 2013; Lye & Koh, 2014). On the one side, VPEs provide several features to support and foster CT in K-12 education. In particular, Scratch can be the most relevant and reliable VPE for the development of interactive games based on related literature. For instance, Koorsee et al. (2015) have tried to determine the impact of a programming assistance tools such as RoboMind, Scratch, and B# on IT scholar understanding of programming concepts and perception of the difficulty of learning how to program. Findings have indicated that Scratch was easy to use, influencing to a large extent the effectiveness of students' understanding of how to use properly fundamental programming concepts and constructs. Webb and Rosson (2013) have also utilized Scratch for fully fledged integrated development to create scaffolding game playing learning tasks. It seemed that students' learning was focused on key aspects of problem-solving, game testing and debugging their own programs. From a gamer's design perspective, all VPEs have one crucial limitation that Scratch does not have. In most programming environments, all code blocks can be observed from the users and this may not assist so ease in thinking before coding. Nevertheless, there is a notable option that Scratch has than other VPEs, since game designers can program and upload a game without other users/players have the permission to download, play, modify it or even explore the code inside the visual elements and objects. Owing to the positive perspectives and perceptions of gender equality to support CT instruction with good computational practices resulted from previous studies (Mouza et al., 2016; Webb & Rosson, 2013), there is another reason for choosing Scratch as the most appropriate VPEs to satisfy the purposes of this thesis.

On the other side, 3D VWs provide a significant number of characteristics and features to support and foster CT in K-12 education. A 3D VW offers a realistic representation of a virtual environment, in which users can provide solutions to simulated problems, tracking their errors visually and auditory to understand better the consequences of their actions during the execution time (Esteves et al. 2011). Nonetheless, taking under consideration the little evidence in regard to the use of 3D VWs' own programming language which is similar to C, thus it provides several difficulties to be utilized by boys and girls at a younger age, it is imperative to identify further tools that may impact positively their engagement and participation in coding tasks. Particularly interesting to meet the design guidelines can be the combination of the S4SL palette with a 3D VW, such as OpenSim for the following reasons (Pellas & Vosinakis, 2017b):

a) the emphasis on the design of algorithmic problem-solving activities by avoiding syntax errors from LSL;
b) the tools that students can use to create, edit and syntax multiple artifacts via S4SL and a-/synchronous communication tools in OpenSim to coordinate the learning procedure;
c) the direct feedback based on their actions in a 3D environment by copying and pasting the code blocks using the visual palette from S4SL as design patterns to an object's notecard to integrate behaviors and/or predict movements/instructions; and
d) the S4SL's palette has similar instruction commands and programming constructs in colored code blocks similar as those of Scratch. The S4SL's palette is frequently being used by students in high schools to university novices in programming. Such a feature can help students' motivation and participation in programming.

Based on the above, the development of a SG using either Scratch or OpenSim need to have a substantial number of different stages that have progressive difficulty levels, multiple goals, and appropriate information to ensure certain learning outcomes inside simulated problem-solving tasks that can support students to think and practice "computationally". It is expected that students regardless of their gender to express and apply efficiently and effectively their solution plans using a logical way of thinking and use some of the most appropriate programming constructs to apply their plans.

Chapter 6: The Robot Vacuum Cleaner (RVC) simulator

The current chapter introduces the implementation of the proposed theoretical design framework that is specifically designed to assist high school students' learning on how to use computer programming constructs to solve simulated problem-solving tasks while also developing skills related to CT. It designates the game design and gameplay overview of a SG called *"Robot vacuum cleaner"* (RVC) simulator following the design decisions and design criteria from *PIVB* design framework created in Scratch and OpenSim combined with the visual palette of Scratch4SL so as to support students develop and apply their computational problem-solving strategies in instructive guided settings (formal and informal). It highlights a detailed game design mapping to align a set of specific guidelines from the *PIVB* design framework with the essential components and elements. Thus, the most prominent alignment between design guidelines and criteria to draw a rationale with the purpose of giving an answer to describe the RVC simulator design are presented including gameplay overview (scenario, game mechanics and tasks), user interface design features and elements that can help students to learn how to think and practice "computationally" by playing such a game.

6.1. Game design

Game design is one of the most important issues that game developers need first of all to consider. It is the description of a game's process about the way it works, its features and components such as conceptual, functional, or artistic, and how someone can transmit any information to build it properly using such a game (Adams, 2009). The *PIVB* framework includes design guidelines and concepts that need to be represented within a SG to support CT instruction through programming courses. A significant number of guidelines and features that need to be presented are of great interest for those instructional and game designers who have not a strong socio-cognitive or programming background. For this reason, the current section highlights a detailed game design mapping to align a proposed set of specific guidelines from the *PIVB* framework with the essential components and design criteria which are finally utilized in order to be created a SG. Therefore, the most prominent alignment between design guidelines and criteria to draw a pathway with the purpose of giving an answer to the research "gap" that described in the previous chapter are depicted in *Figure 6-1* and presented more analytically below:

- *G1: Motivating students to participate in active learning tasks*. Players' motivation and persistent engagement (P1) will come from the exploration and identification of a problem from the real world and it may have some contemporary aspects since students live in such an era. For this reason, a computer SG needs to have a scenario, features, and elements which may reflect on students' real

life. At this point of view, the learning content (C1) needs to provide in-game visual objects and elements that all players can use in order to gain information. All those objects and elements need to be integrated inside the game and provide to a player some unambiguous information in relation to the main scenario. A suggestive scenario that can have an impact on students' life regarding, for example, their assistance and solidarity that they can give to other people. As a result, many learning scenarios can support a proposed game concept. One of the most indicative is the students' assignment having a specific role in which they may try to program a computing machine created via interactive environments so that solve realistic simulated problem-solving tasks.

- *G2: Simulating an authentic problem.* The clear and challenging goals (P2) of a gaming system need to allow players choosing a male or a female virtual representation and provide various learning materials material that cannot cause gender biases (C2). First of all, students need to observe and use visual objects/elements which are really crucial for them to get any information that is required to complete in-game learning tasks and goals. Second, since a game scenario contains several events and actions using a game environment project those events and actions, thence designing such a game should provide visually appealing objects. It is hypothesized that if students try to propose a solution for simulated real-world tasks, they should be also able to give them specific commands and constructs without being so ambiguous. In this perspective, the learning goals are also important and need to be mentioned. The learning material inside the game is represented through in-game elements and indicated as a part of many CS curricula around the globe to be focused on the following two key aspects. The first is the analysis of proposed solutions to a problem in a text form using algorithms or pseudocodes in natural language. The second is the implementation of students' computational problem-solving strategies that lead from problem formulation to solution expression into the code so that students can be able to apply their solution plans.
- *G3: System's feedback on the user's actions.* The system's feedback on the user's actions (P3) is associated with the user interface design features and elements (C3). Since a SG needs to provide problem-solving tasks reflected on simulated real-life events, the feedback that players may receive should be predefined and prompt according to their actions inside the game. For this reason, all visual objects and elements need to provide visual and auditory feedback on each players' actions in order to be easier understandable how correct (or not) they try to approach each subpart of the main problem.
- *G4: Facilitating the development of computational problem-solving strategies through a scaffolding process.* The scaffolding process (P4) refers to an instructional game that contains several stages with different levels of difficulty. This means that students need first to start with an

exercise that is included inside each stage from the easy to a more advanced in order to solve a diversity of problem-solving tasks. Such an effort may assist them to start thinking how easy parts of a solution for some subparts of a problem can be combined or can be extended in order to provide a more concise later.

- *G5*: *Applying design patterns to propose an answer to a problem question.* The debriefing process is based on students' skill-based learning outcomes (P5) which are reflected on awards or punishments concerning to the solutions that they can propose (C4). In this perspective, players cannot use ambiguous code blocks but only those which may give a solution in practice about each task of the problem. Players need to apply their solution plans as subparts of a program according to the given instructions and detect logically any potential errors by executing programming commands and constructs into their programs. For instance, the use of a visual palette can be proposed in an effort to avoid code complexity and focus more on problem-solving.

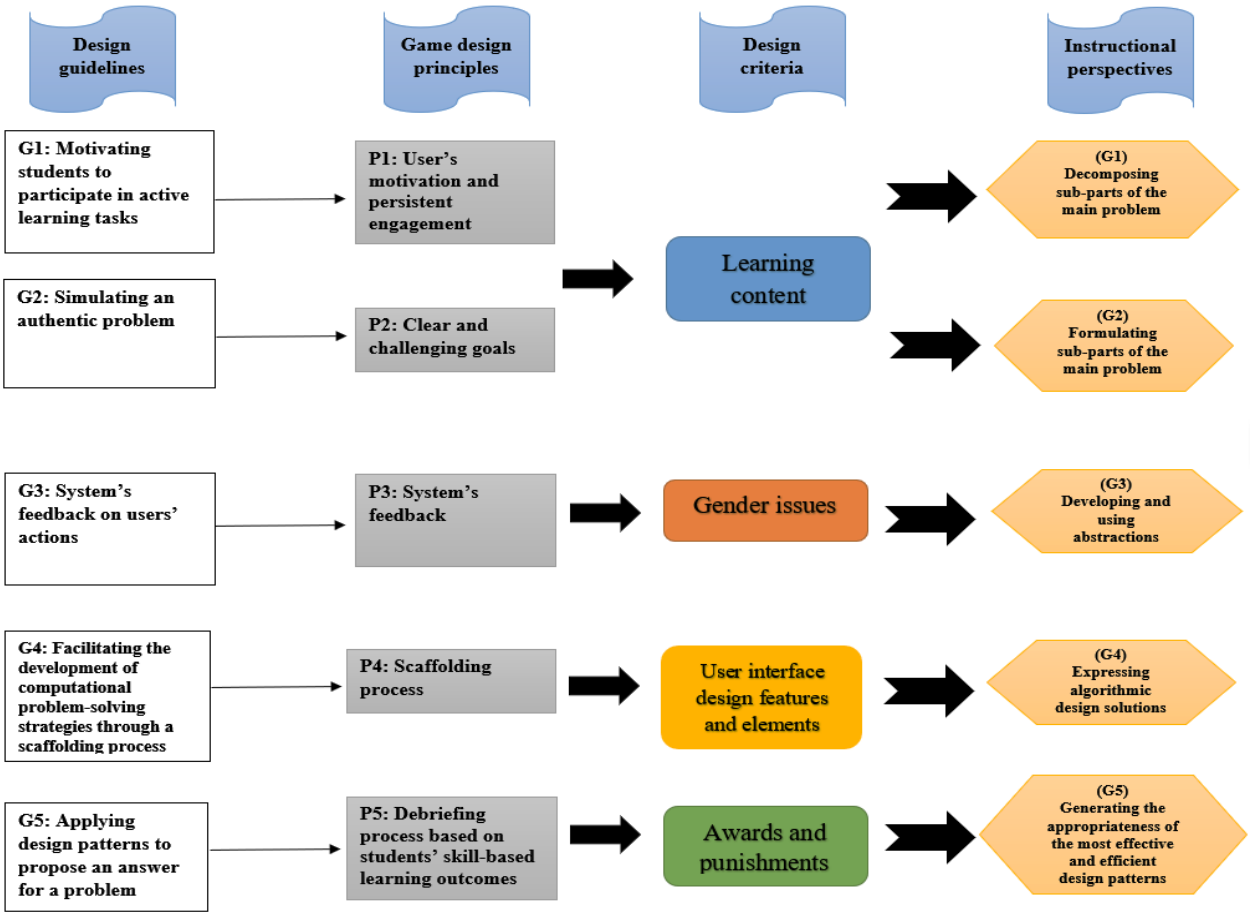


Figure 6-1: A design map constructed by following the game guidelines and principles of the PIVB framework

According to the proposed framework and design guidelines by Pellas and Vosinakis (2017a), the RVC simulator has the following characteristics:

a) the simulation of authentic problem situation and functionalities in fading scaffolding processes to support users' roles (students and instructor),
b) the visual metaphors of Scratch and OpenSim related to innate CT skills and conceptualize them into algorithmic rules through abstract thinking logic, and last but not least
c) the use of programming tools which can eliminate the split attention in code syntax assisting to this vein players to be focused on their solutions that applied as results of computational problem-solving strategies.

By using the proposed SG following such design guidelines, players can consider how specific actions will (or not) solve a problem. Thus, they can have the opportunity to understand the appropriate use of fundamental programming constructs by liaising to those constructs constantly with the appropriate computational problem-solving strategies to transform their innate thinking solutions into code.

6.2. Gameplay overview

6.2.1. Learning goals and scenario

The "*Robot vacuum cleaner"* (RVC) simulator is an interactive problem-solving environment that visualizes a simulation process in which players need to steer one visual object (a vacuum robot cleaner). The main learning goal that students need to complete is to program correctly a simulated vacuum robot in order to clean some rooms in a big house, by investigating and applying the most viable routes. Players need to think before start coding for expressing and applying the most efficient and effective solution plans using fundamental programming constructs and instructions. Inside a big house, 8 rooms are existed to determine all in-game stages. Players need to explore the entire house and then decide which of those rooms would like to play, having the role of embedded software engineers. For each room, players need to map out spatial orientation and layout of each room that is different from the others and they must program a RVC in order to clean the only the 4 chosen rooms. Each room has different levels of difficulties. This means that they start initially with stages (rooms) that have less evocative spatial metaphors of basic geometric shapes (e.g., triangle, square, and hexagon) so as to identify and apply a solution plan into code as pathfinding in a logical problem. If players have progress, they need to continue programming the RVC so that clean the other rooms which have more complicated geometric metaphors, and until completing all the chosen one, then the game can be terminated in the last stage. The main algorithmic problem that is projected inside the proposed SG comes in align with pathfinding is the "visual plotting" that refers to a computer application in which players need to identify and apply the shortest route between two or more points. Such an approach is useful in a more practical variant on problem-solving a mind trap maze.

The learning materials inside the RVC are represented through in-game elements and objects which are relevant to the needs and demands of high school students as indicate with specific instructions given by many CS curricula around the globe, focusing on the following two key aspects (Webb et al., 2017; Tuomi et al., 2017):

a) the analysis of expressed solutions to all problem-solving in-game tasks in a text form using algorithms or pseudocodes written in natural language, and
b) the implementation of students' computational problem-solving strategies that lead from problem formulation to solution expression into the code so that they can be able to apply their solution plans for problem-solving simulated tasks using fundamental programming constructs.

To all in-game tasks, specific guidelines from the Greek curricula were taken under serious consideration. Both in the Greek curriculum (Hellenic Pedagogical Institute 2003; Teaching Guidance from the Greek Ministry of Education, Research and Religious Affairs, 2017) and the Greek school book (Arapoglou et al., 2003) have been referred specific learning objectives that need to be completed inside school contexts regarding the way that computer programming needs to be taught and thus all those were considered as essential for the creation of the proposed SG. In particular, the learning goals that lead to the expected outcomes can be achieved by familiarizing students with specific elements and features regarding the use of interactive environments in order to solve various problem-solving tasks in simulated real-world contexts. Another point of view is that the researchers and/or the CS instructor(s) need to inform students at the beginning of a teaching intervention about how to use the proposed SG in order to achieve the following learning goals:

a) to investigate how a RVC needs to be moved into a house, taking into account the spatial layout of each room in which existed several simulated problem-solving contexts between the furniture and other house objects are provided;
b) to propose a solution with logical reasoning by expressing specific steps based on a computational problem-solving strategy and exploit different forms of constructs and commands such as REPEAT, *"From ... until ..."* or *"Until...repeat"*, SELECTION (*"If ... then"* or *"If"* then *"else"*) or the SEQUENCE in order to apply into code to each in-game task;
c) to explain the appropriateness of using specific programming constructs in order to propose solutions as design patterns that can be integrated as behaviors into the RVC so as to predict its control movements without causing damages inside the house.

The RVC simulator has a specific scenario. Having the role of embedded software engineers, players should assist an old woman with special needs who moves only with her wheelchair and struggles to clean all rooms of her house by programming correctly an autonomous RVC. House furniture and objects in square floors are seen as evocative spatial metaphors of basic geometric shapes (e.g., triangle, square, and

hexagon) so as to assist students to think and practice "computationally" following an abstract conceptualization approach as an effort to understand better a visualized problem-solving environment alongside with a pathfinding in a logical problem. Abstract spatial representations of geometric shapes that are created by three visual objects (a table and six small chairs) and were extensively used inside the SG, such as a triangle, for example, to prevent hitting a table, players need to determine arithmetic computation between chairs and table distance. More specifically, each side's square floor has side 5m in OpenSim (or 140 steps for a movement that executed inside Scratch) and/-or calculate degrees of turning correctly (e.g., 90^{o} for square or 45^{o} for equilateral triangle) to traverse the RVC in specific cleaning pathways down from the table, without dropping all books from the table (see *Figure 6-2*). Players need to take advantage of the environment's spatial layout comprising all of the rules for performing arithmetic computations for the distance of the robot between their virtual representation and house furniture. The RVC can move and clean each room that differentiates in spatial geometry layout, in terms of division among house furniture or objects and succeeds to this notion player who first need to calculate and determine arithmetically the distances between objects in each room differently without causing hits or damages. This process is becoming more compelling as players need to apply their computational strategies in practice so as to present the shortest path between the present location and the goal location of the robot by integrating behavior inside it.

6.2.2. User interface design features and elements

The design and creation of the RVC simulator were tried to be as similar as possible in both platforms (OpenSim with S4SL and Scratch). On the one side, the user interface design features and elements of the proposed SG constitute from a window-based environment as a 3D simulation via OpenSim and S4SL, a visual palette that was "outside" from OpenSim to program behaviors which need to be integrated inside the RVC (see *Figure 6-2*). Following are the main elements of this game created via OpenSim and S4SL:

- The *"client viewer"* where the entire game is displayed allowing users to dictate when the script is executed properly.
- The *"notecard of RVC"* as a visual object where the script for determining a cleaning path that needs to be followed by integrating specific code blocks inside it. The notecard contains specific instructions and programming constructs that are applied in the visual palette of S4SL, and then each player can copy and paste those instructions and constructs inside the RVC's notecard to run it inside OpenSim.
- The *"S4SL"* palette outside the client viewer. It is a visual palette contains the colored blocks used to create the design patterns (right side). Users can select a variety of blocks that are displayed in

different colors and provide programming constructs, instructions/movement, numbers, and variables similar as those that exist in the visual palette of Scratch (left side of the palette).

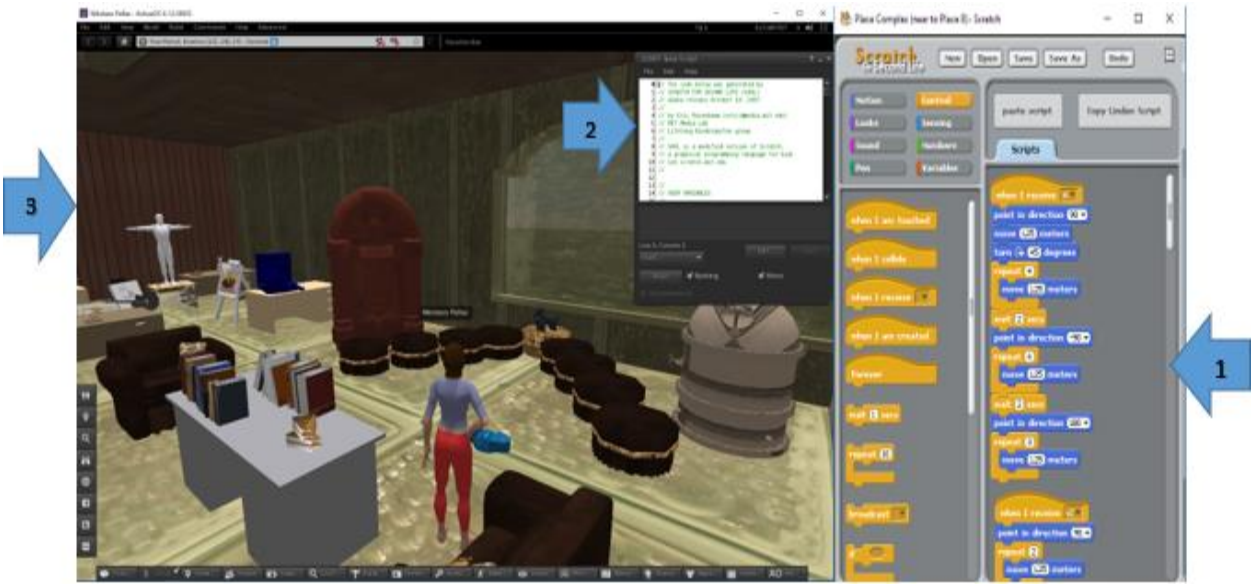


Figure 6-2: The graphical user interface of the RVC simulator created in OpenSim with Scratch4SL

On the other side, for the development of the proposed SG in Scratch, the following features and elements for the development of the RVC simulator required:

- Scratch includes both a visual palette and a "window-based" stage. The former contains several sprites (icons) that can be used by someone who wants to integrate behavior by programming them and using design patterns (right side). Users can select a variety of blocks that are displayed in different colors and provide programming constructs, instructions/movement, numbers and variables (left side of the palette).
- The *"Stage"* is where the entire game is displayed and allow users to dictate where each script can be executed.
- The *"Sprite"* of the RVC contains the script that players can integrate for determining and programming a cleaning path using the visual palette that can be visualized in Scratch's stage.

Figure 6-3 depicts a combination of "iteration" (repeat) code blocks inside Scratch. Once a player completes his/her design pattern, the visual object starts to run the main script. That is reflected only if there are more blocks underneath the under the *"when I receive…"* block in a script, they will run whether the condition placed in the *'If…Then'* block is true or not. Boolean blocks can be also used to make more complex checks on conditions.

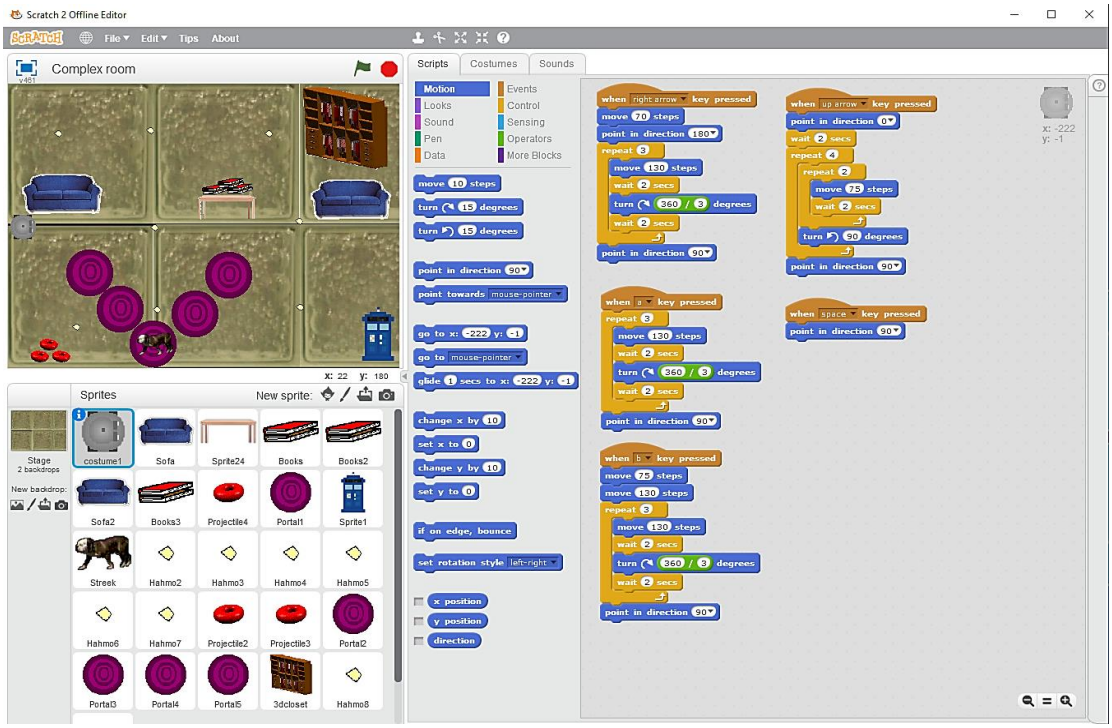


Figure 6-3: The graphical user interface of the RVC simulator created in Scratch

As it is depicted in *Figure 6-3*, there are also three types of Boolean blocks: The *"or"* block, the *"and block (+)"* and *"pick a random"*. Boolean blocks can be used in each design pattern as a standard condition block can be placed by adding one or more condition blocks, so that they can return a value of true or false that can be checked. Several are the notable code blocks which need to be referred. The event blocks are used to determine when a script will be triggered, such as a block titled *"when I clicked"* or *"when space pressed"*. All design patterns can be saved in the visual palette as instruction cards (or scripts) until it has been given an event.

6.2.3. Description of activities and learning challenges

The RVC simulator gives various kinds of visual feedback to help players comprehend if the computer instructions are correct in order to help the RVC's movements and change the states of the graphical objects (checkpoints) to gather the gray dust dots from the grids inside the house (see *Figure 6-4*). It also provides feedback on students' performance for solving computational problems in terms of the number of dust dots inside each grid that is cleaned, and the number of visualized instruction blocks used. The assembly of code blocks includes a drag-and-drop process present a chunk of computer instructions and programming constructs that can be used to help players to plan a solution by subdividing it into smaller parts. To accomplish all learning goals of the RVC simulator, players need to apply their computational strategies in practice beneficial to present the shortest path between the present location and the goal

location of the robot by integrating behavior from S4SL to OpenSim or in Scratch. Specifically, they need to execute and propose a solution as a set of design patterns by combining programming constructs (sequence, if/else statement or loop) and instruction/movement commands. Nevertheless, there are several distinctive similarities and differences which need to be identified. These are tabulated in *Table 6-1* below.

Table 6-1: Similarities and differences of the game interface design created in OpenSim and Scratch

Issues	Similarities	Differences
Learning tasks	Problem-solving tasks to support CT instruction and evaluation of students' learning performance	**OpenSim:** It gives a sense of presence on players' experience as avatars with the feeling of "being there". A 3D VW allows players to view all objects' motion to a greater perception and subjective sense of being within a realistic simulated digital context. **Scratch:** It gives flat drawings where players can view all in-game objects and elements in 2D sprites (images).
User interface design features	6 in-game stages (3 stages to play and another 1 for learning how to play)	**OpenSim:** In-game stages have realistic simulated representational fidelity that is displayed in a 3D digital persistent environment, where players can explore and observe everything inside it. A 3D RVC simulator portrays a visual realism. **Scratch:** In-game stages were separated and displayed as 2D sprites and are opened only when players choose them.
Functionality and playability	The RVC simulator's operationality	**OpenSim:** a) Viewing and exploring in-game stages and element/objects in OpenSim is achieved by taking advantage of intuitive, natural modality contexts for user-interaction tasks (length, width, height-*x*, *y*, *z*-axes). b) Movements in a 3D RVC simulator requires the exploration in a 3D world, in which players can move closer and deeper into realistic settings. **Scratch:** a) Viewing and exploring in-game stages and every feature/object were taken with a panoramic view using 2D sprites for user-interaction tasks. b) Movements to a 2D RVC simulator restrict each player's movements to a flat plane, but it includes various directions (length, width-*x*, *y*-axes).
Programming tasks	The programming tool is displayed as a visual palette and has the same fundamental programming constructs	**OpenSim:** The visual palette is outside OpenSim. **Scratch:** The visual palette is integrated inside Scratch's environment.

The description of in-game activities is also worth noting. The RVC simulator drives players to analyze, visualize and practice the correct use of computer programming constructs for achieving in-game learning goals. The conceptual integrity of the proposed SG is based on the use of skills related to CT from the game experience and not on teaching any general-purpose programming language. The RVC simulator is not designed to improve any operational refinement that assists students to describe their actions in terms of expressing pseudocodes, but it needs every solution plan to be applied with skills and strategies that are acquired from the game-experience to be transferred into programs. Such a SG is also concerned about scaffolding instructional approach as the whole idea behind constructing solution plans is to make each student think and practice "computationally". Furthermore, the proposed SG does not focus on a specific gender, and players do not need to have any programming knowledge to play the game. RVC is designed to respect gender equality and expertise neutral of high school students. Firstly, the proposed SG is not gender-oriented because its theme is a RVC that needs to be programmed correctly to clean all rooms, in which players should program and visualize several and alternative cleaning pathways. Secondly, players do not need to have prior or extensive experience in programming knowledge to play the game. Since a specific role is assigned to each player, a number of steps in order to complete his/her strategy need to make the following:

a) to explore any of the chosen rooms separately to identify drawbacks between visual objects and furniture creating visual and abstract simulation content,
b) to plan specific movements to pass all checkpoints the vacuum robot for optimum performance
c) to propose the shortest cleaning path in reasonable time, and locate any further points that should be avoided so as to clean all dusty dots the floor, without hit any object or furniture,
d) to program the shortest cleaning route that can be proposed for each room individually in order not to turn off the robot due to battery consumption after that cannot last up to one-hour time, and last but not least
e) each player needs to describe and apply algorithms that can calculate the most efficient and effective routes as cleaning paths.

All in all, 6 rooms designed with learning tasks lasted each for about 40 minutes. For each one, players were free to propose different solutions based on their design patterns as there was not a pre-defined one. They had the chance to choose and solve problems with only 4 rooms, with 1 to be chosen from each stage. Only the 3 chosen rooms counted for their final grades. The bedroom or the drawing room are developed to be chosen for introductory activities in order to learn players how to use some tools and another one room that each player could exclude.

Figure 6-4 depicts all in-game stages created in Scratch on the left side and stages created in OpenSim are on the right side. A presupposition is to use the same programming method and constructs (i.e., simple

or nested iteration, sequence or selection) at first stages including the bedroom (1.1.) and the drawing room (1.2.) to propose a solution for the other 3 chosen rooms (stages) only once more. This means that for the other two, players need to propose a combination of programming methods or nested with numbers and/or variables. When participants decide which of the 3 rooms from the three stages wanted to play, they had the chance to use one different method that can be combined with a proposed programming method in order to gain higher grades, e.g., a combination of selection (if…else) and/or iteration with a sequence of commands.

Except for the above two rooms, the rest four have different levels of difficulty. For example, the second stage includes the billiard room (2.1.) and cinema room (2.2.) have a medium level of difficulty due to a fewer number of objects and house furniture that is provided, in which players can use either one or a combination of more programming methods.

In another example, the relaxing room (3.1.) and sitting room (3.2.) are included in the third stage. Both have a higher level of difficulty, as at least optically house furniture and objects were significantly more than in other stages and this feature could assist (or not) players to create different the geometric shapes for cleaning pathways, and thence more programming methods need to be combined.

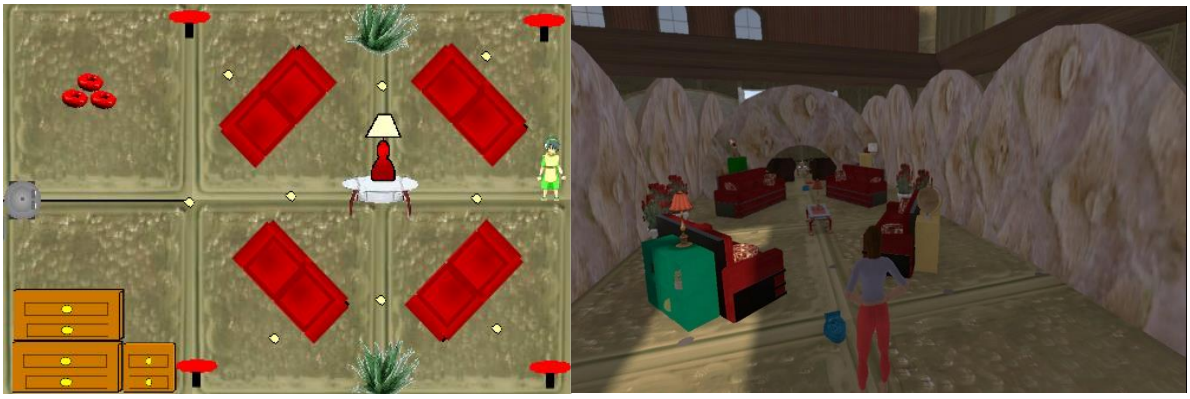
1.1. The bedroom

1.2. The drawing room

2.1. The billiard room

2.2. The cinema room

3.1. The relaxing room



3.2. The sitting room

Figure 6-4: The in-game stages created in Scratch and OpenSim with Scratch4SL

Learning challenges through the RVC simulator's gameplay require the analysis on how to plan a solution for a cleaning path problem. Players need to articulate a solution aimed at creating algorithms with logical and precise instructions and finally applying their solution plans for subparts of the main problem into code. Firstly, they need to navigate, determine movement positions and describe the best cleaning path

that an autonomous RVC can demonstrate in sufficient time. They need to subdivide the main spatial problem-solving task into smaller parts, apprehend hypothetical error situations for retrieving visual feedback for their actions inside OpenSim or Scratch. After that, they need to debug their cognitive thinking process by testing and figuring out possible misconceptions in computational practices through coding.

To identify and present a proposed solution by explaining a step-by-step solution before its execution, the core gameplay mechanics, basic rules, and functions of the RVC simulator were announced to all participants with specific instructions in hard copies (see Appendices H and G). The direct feedback is based on a player's actions by copying and pasting the code blocks from the palette of code blocks as design patterns to an object's note card that is integrated into a visual element created either in Scratch as a sprite or in OpenSim as a visual object. Players need to consider that the robot should not be moved for more than 10m, because for each square floor, it has to move 5m (or 140 steps for Scratch) distance in length and width from the owner in order to be controlled by a mobile smartphone. Stopping the RVC to pick up the dust only for 2 seconds for better cleaning is also needed. An indicative example is depicted in *Figure 6-5*. For example, a boy using Scratch and a girl using OpenSim with S4SL faced the same simulated problem-solving tasks. Both were needed to explore what movements the RVC should make in a cleaning pathway to be applied correctly their solution plans into code. The boy proposed an alternative solution that looks like being a square root spiral. In other words, he pointed out the center of each square to make the robot spiral movements based on the given instructions that need to be encoded. When the robot is moved under the table (root), the boy needed to use the same design patterns with iteration and commands blocks in relation to numbers or variables by changing its rotation spatially and correctly the RVC's movements to clean each room.

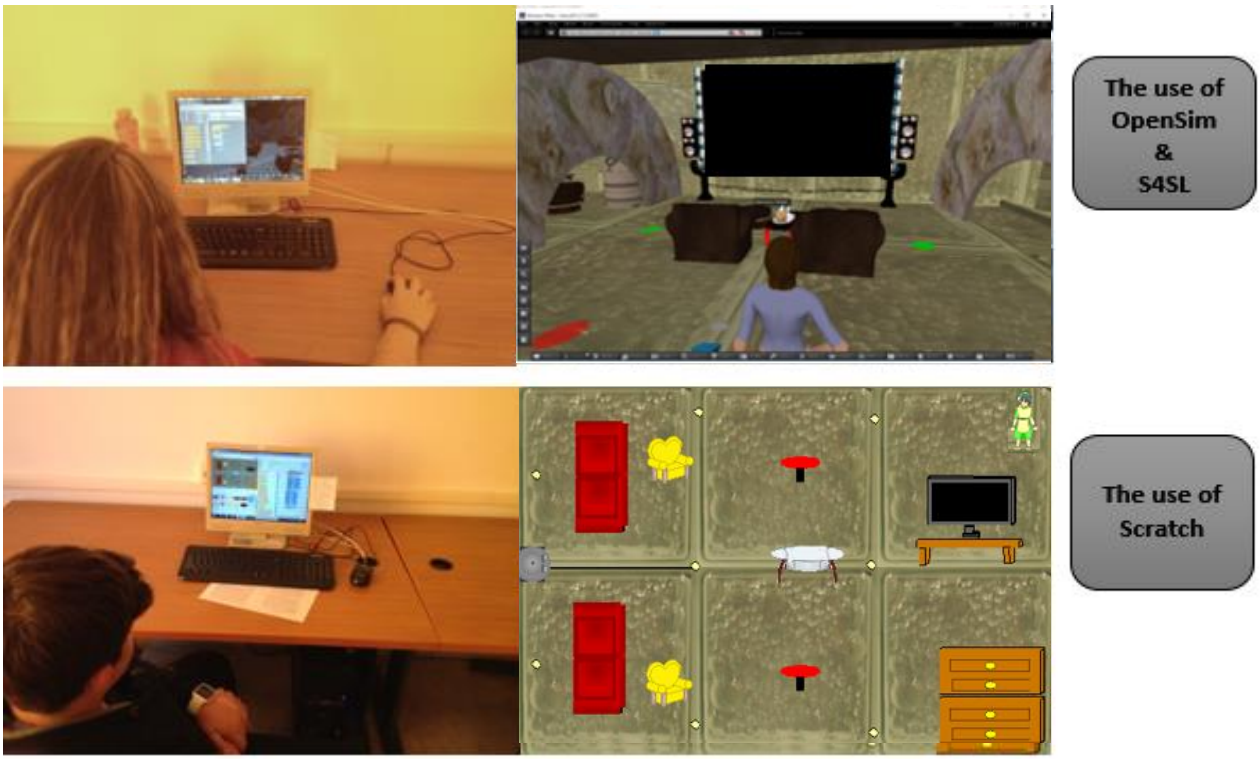


Figure 6-5: An illustration of the in-game learning process in the cinema room

The final scores encouraged a level of competition among players to be submitted in a high score list, when they succeed all in-game goals from the chosen rooms. Such an approach leads to a non-compulsory competition among those players who want to compete with each other and thus provides a limited interaction among players. As the competition in the game designed to respect any gender and expertise equality, since players had the chance to announce if they want (or not) to submit their scores to the final list and stay anonymous.

6.2.4. Game mechanics

Several core mechanisms are integrated inside the RVC simulator determining the pre-defined rules that are designed for the interaction of players with the game system, thus providing gameplay. First, six checkpoints inside each room allow the "mapping" process and allow players to start from such a point in case of hitting any house object. Also, each player had the chance to place another 3 checkpoints in order to visualize better his/her proposed cleaning pathway. If the robot is programmed correctly to pass above them, then the total time is not counted until the final solution is finished without losing the RVC battery's energy. Moreover, whenever the robot is programmed to pass and clean all dusty dots (gray signs) off the floor, it gains energy, giving grades to its battery life (award). Another import issue is to stop the robot for 2 seconds in order to clean each gray spot. Otherwise, penalty scores are excited every time that each player

applies his/her computational practices and hit an in-house visual object, losing for such an action 0.5 grade (punishment).

Second, for each of the 8-gray dust dots to every stage can give 0.5 grade which were visualized as a text message with the number "1" and a sound is played too. Therefore, each player can gain 12 grades at most from the 3 stages, since only 1 room was for practice. If gathering all dots with the smallest possible number of code blocks by applying efficient and effective programs that can be integrated inside the RVC to be cleaned each room based on resilient planning, execution time and fewer hits on the house furniture or objects, then such a player is declared as the winner.

Third, there are some in-game awards and punishments which are given. For instance, a good computational performance grade is announced when correct instructions and constructs in design patterns are integrated inside the robot, whilst in this SG, sketching geometric shapes have similar behavior patterns to the robot's movements as cleaning pathways inside a room. A bad one is provided if a player uses constructs and commands in which the robot's movement include only "zigzag" movements that may be correct. Additionally, the time to be finished and code blocks will be much more. Such an example is given in *Figure 6-6* that presents the visual palette of Scratch with 4 different design patterns as solutions to a computational problem inside the big house. Condition blocks' check is provided if a given condition is true or false. For example, the condition shown in *Figure 6-6* can be changed with the first code block checking if a statement of motion is taken to move it appropriately without hitting some objects or if a distance to the owner is larger than a proposed one. Second, the control blocks allow users to make more complex scripts that react to the player's choices and the current state of the proposed SG and introduce computational concepts that are likely to be of wider use.

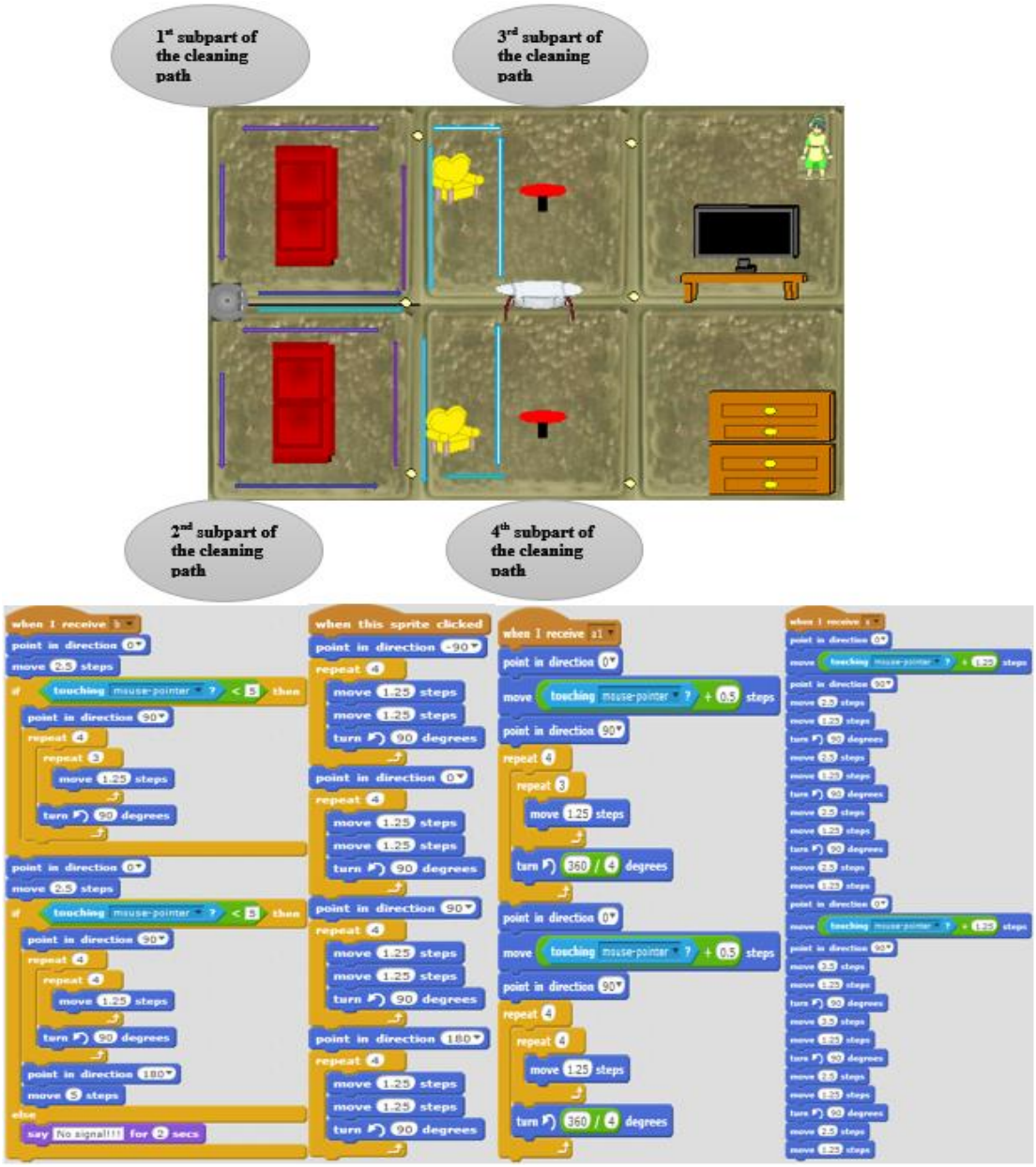


Figure 6-6: The four different design patterns as solutions to a computational problem

To sum up, the RVC simulator is developed to encourage players to think and practice "computationally", in an environment with simulated real-world problem-solving tasks in which they need to utilize CS programming concepts and constructs taught based on the guidelines that almost all CS curricula for high school programming courses have been across the globe proposed. By way of illustration a SG, a RVC simulator is developed. It is focused not only on the operational level of abstraction and skill acquisition related to CT, but also it gives to all students who have different gender and programming background to contextualize and use properly fundamental programming constructs (i.e. programming sequence, functions, decision making, loops) so as to apply their solution plan into the proposed SG.

Chapter 7: Experimental design

This chapter demonstrates the experimental design and data from the statistical analyses resulted by conducting two studies. It aims to present the effects of using the RVC simulator on teaching and learning computer programming. The effects of the RVC simulator are assessed through a preliminary and a quasi-experimental study. The former aims to examine the effects of using the first prototype RVC simulator created in OpenSim with S4SL to support CT instruction. It presents the first usage of RVC simulator and how well the proposed SG operates, to determine any problems and possible weaknesses that need to be solved by assessing the learning effectiveness, the learning procedure, and user experience of fifteen (n=15) high school students. The latter presents evidence about how the proposed SG could assist boys and girls to gain a greater understanding on skills related to CT for developing, implementing and transforming their solution plans into code in regard to their learning performance by assessing their computational problem-solving strategies (i.e. computational design, computational practices, and computational performance). A total of fifty (n=50) high school students who volunteered to participate in this second study divided into a control group (n=25) and an experimental (n=25) group using Scratch and OpenSim combined with the Scratch4SL palette, respectively.

7.1. Rationale and purpose

In recent times, education scholars, CS teachers, and researchers are increasingly turning to the use of interactive environments in order to identify and intervene with students at risk of underperformance or discontinuation in programming courses. Prior works following GBL approaches were focused either on the measurement of boys' and girls' engagement and participation using interactive environments (Costa & Miranda, 2016; Lye & Koh, 2014) or in the aspects of analyzing executive solutions built from the combination of blocks consisted of simple or nested programming constructs as design patterns in terms of using correct (or not) syntax or semantics of a programming language (Brennan & Resnick, 2012; Howland & Good, 2015; Werner et al., 2015). Literature in the field of CT instruction through programming courses (Denner et al., 2012; Mouza et al., 2016; Werner et al., 2015) has also advocated that measuring computational problem-solving strategies of students with different gender by applying integrated behaviors in visual elements using a SG can profoundly influence their learning performance.

Although recent studies (Kalelioglu et al., 2014; Mouza et al., 2016; Witherspoon, 2017) have provided empirical evidence on how students can develop and program their games using skills related CT so as to apply their solution plans into code through creative computing or artistic expression tasks, limited research demonstrated how a SG's features and elements can support CT instruction. Given the advances

in research about K-12 programming courses for CT instruction and in particular those that incorporate GBL approaches, a considerable limitation is the small number of empirical studies which have tested the appropriateness and the effects of SGs on students' learning performance in overall (Chao, 2016; Howland & Good, 2015; Liu et al., 2011; Liu et al., 2017). With that in mind, a substantial body of literature reviews has come to the statement that there is a "gap" concerning the creation and use of new interactive environments (Grover & Pea, 2013; Lye & Koh, 2014) or the combination of already known "tools" for game playing tasks (Kafai & Burke, 2015). Besides the widespread interest to use several interactive games, there was no evidence if a SG created either in VPEs or in 3D VWs which differ on user design features and elements can affect students' learning performance by solving simulated real-world problems.

To fill the above-mentioned research "gap", this thesis seeks to investigate whether a SG interface and elements created in OpenSim that has a more natural intuitive modality for user-interaction tasks than Scratch can significantly affect students' learning performance by assessing their computational problem-solving strategies (i.e. computational design, computational practices, and computational performance) to the same simulated real-world problem-solving situations. Having explained the rationale of proposing specific guidelines, characteristics and features of the RVC simulator and the reasons why it is designed, thereby a research approach and design needs to answer this thesis's hypothesis. In other words, it is required to assess whether or not such a game can offer an educationally effective solution for high school students on how to use fundamental programming constructs by thinking and applying their solution plans using skills related to CT. A suggestive way to give answers in such a hypothesis can be the use of the proposed SG gameplay created by combining the visual palette of S4SL to prevent programming syntax complexity and the realistic simulated representational fidelity of a 3D VW like OpenSim or by using Scratch's features and elements so that support greatly the development of students' computational problem-solving strategies. Therefore, two research questions (RQ) are arising:

RQ1: Can the RVC simulator created in two interactive environments with different GUI features and elements support the development of students' computational problem-solving strategies?

RQ2: Are there any significant differences in students' learning performance resulting from the description and expression of computational concepts and constructs into the code for proposing solutions to several simulated problem-solving tasks via the RVC simulator?

The present chapter describes the main research design method divided into a twofold experimental setup. Due to a lack of studies assessing a game playing framework, this thesis' experimental setup seeks:

a) to test a prototype SG so that support CT instruction through programming courses following the design guidelines of the *PIVB* theoretical framework by conducting a preliminary and an experimental study, and

b) to observe how and what features and characteristics of the RVC simulator can greatly support students' efforts in programming courses in order to develop and apply their computational problem-solving strategies.

To achieve the first objective, a mixed-methods preliminary study is conducted in order to investigate if the RVC simulator can support the development of students' computational problem-solving practices into code. Based on previous studies (Rubin & Chisnell, 2008; Tullis & Albert, 2013), a sample consisted of five and more participants are suited to detect the most important system issues since almost 80% of the usability deficiencies of a first prototype will be exposed by such a number of participants. In this preliminary study, students were familiar with technological and interactive environments and games, but they have not got any prior experience with other similar prototypes like the RVC simulator. Such a study can give initial evidence to discuss the potential reasons for using the proposed SG created in OpenSim with S4SL to identify any potential problems and then improve any design and/or usability issues by measuring learning experience and first perceptions of a total of fifteen (n=15) high school students (Pellas & Vosinakis, 2017b).

To achieve the second objective, in an effort to widen and generalize a more efficient way to foster computational problem-solving strategies of students, a quasi-experimental study is also conducted. The main purpose is to investigate if the RVC simulator can affect the learning performance of boys and girls in order to gain a greater understanding on the use of skills related to CT for developing, applying and transforming their solution plans into code by comparing and identifying any similarities or differences on the implementation of boys' and girls' solution plans. Therefore, in the experimental setup, a total of fifty (n=50) high school students who participated voluntarily in this study divided into a control group (n=25) and an experimental (n=25) group that used Scratch and OpenSim with the S4SL palette, respectively in favor of supporting and applying their solution plans into code for the same problem-solving tasks using the RVC simulator (Pellas & Vosinakis, 2018). Thence, an empirical study is conducted to analyze boys' and girls' computational problem-solving strategies focused on:

a) computational design to express their solution plans in natural language for all subparts of the main problem,
b) computational practices to apply those plans into code as design patterns, and finally,
c) computational performance to measure students' learning performance and outcomes by identifying the most effective and efficient design patterns which have been applied.

The assessment of students' learning performance requires not only the formulation and manipulation of a problem with skills related to CT, but also testing and debugging such a solution's correctness to a problem with design patterns integrated in visual programming elements e.g., the use of control flow blocks from a visual palette to propose and program solution plans. To measure any improvement in overall

rule/instruction specification ability, the mean scores of the worksheets from the two groups, an error analysis rubric is used in the direction of analyzing students' answers in response to the *RQ1*. An error analysis rubric was compromised to all in-game activities related to each one of the CT instruction through several sessions described in *Table 7-3* (see p.154) including examples of various thinking processes. The use of such a rubric is imperative for the description of a solution by writing short sentences in natural language (CT 1-4), then into algorithms/pseudocodes (CT5), and finally into code as design patterns (CT6). In addition, using descriptive statistics in regard to the accurate description and implementation of computational problem-solving strategies comparing students' computational design solution plans, as computational practices that are transformed into code using the visual palette of Scratch or S4SL. The main purpose is to be measured and to be identified students' computational performance by assessing the most efficient and effective design solutions. Also, self-reported students' answers regarding the effects of the RVC simulator focused on pre-and-post CTS questionnaires and post-tests in the direction of determining how they used skills related to CT in response to the *RQ2*.

7.2. Research methodology of the preliminary study

7.2.1. Sample

The sample comprised of 7 girls (M_{age}: 13.87, *SD*: 1.13) and 8 boys (M_{age}: 14.74, *SD*: 1.15) volunteered to participate from the local schools. All participants were recruited to attend after-school sessions. Also, participants were novices and all of them had previous experience with Scratch (100%). In regard to personal information about the sample, all participants had a personal computer (100%), albeit only two of them (13%) have also utilized in their free time other platforms to learn how to program by playing games via *"Hour of Code"*. Almost all have pointed out that Informatics and specifically programming courses are significant for their professional development (80%).

When all participants were selected, the main researcher contacted to their teachers and parents in order to obtain the necessary consent from both the student and the legal guardians (or parents) for the data collection.

7.2.2. Procedure

The preliminary study was conducted in an intensive 2-week period with 6 sessions (see *Table 7-1*). The first 4 sessions lasted 4 hours in face-to-face and the other 2 lasted 2 hours in supplementary online during the spring trimester 2017. In the RVC simulator, students tried to visualize their efforts by programming and integrating instructions combined with programming contexts inside the visual object of

the robot vacuum cleaner in order to predict its movements and proposed the most efficient and effective cleaning pathways (routes). *Figure 7-1* and *Figure 7-2* show students' efforts through blended instruction.

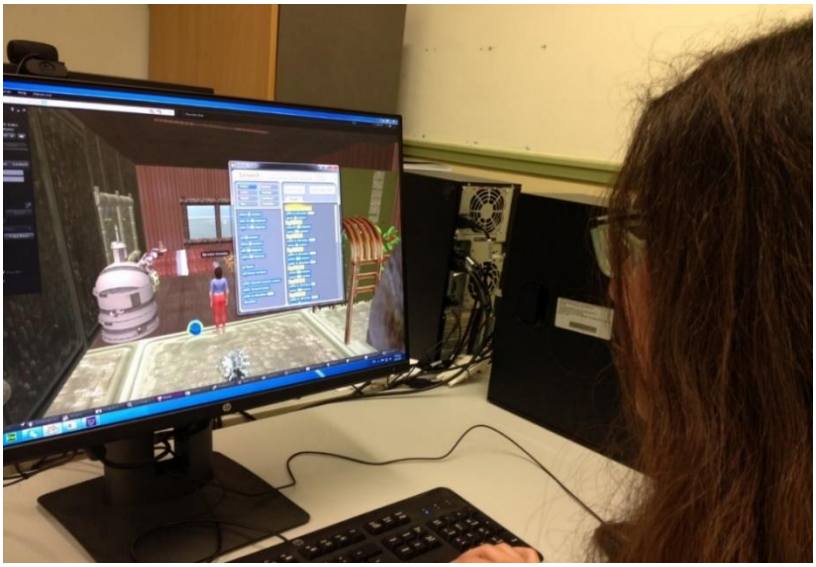


Figure 7-1: A girl proposes a solution via Scratch4SL for the first stage inside the RVC simulator

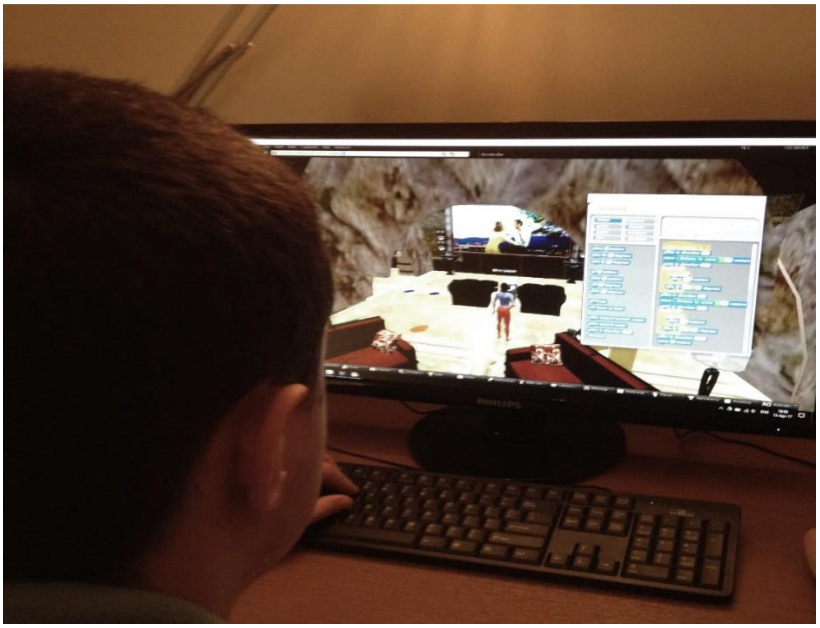


Figure 7-2: A boy proposes a solution via Scratch4SL for the second stage inside the RVC simulator

Table 7-1 outlines a process about how students can develop their skills in gameplay using CT skills so that support computational problem-solving development through in-game settings. This table also validates how cognitive thinking skills (e.g. logical or abstract thinking etc.) related to CT can be developed in the game playing modes in dwelling on problem-solving tasks, understanding problems, and formulating solution plans into code. The instructional approach was made according to the operational definition that

CSTA and ISTE (2011) that can be utilized for the development of the most essential skills related of CT in align with the proposed game design principles (G1-G5).

Table 7-1: Description of activities associated with game playing in the preliminary study

Sessions	**Learning tasks associated with CT concepts and CT skill definition (CSTA & ISTE, 2011)**	**In-game tasks and objectives**
1st session: Presenting the learning objectives and goals of the RVC simulator so as to use its functions.	Think about what the main problem is and its which elements. Students need to explore and utilize further all features in each in-game stage to propose are required to know in order to propose a solution.	**Decomposing subparts of the main problem:** Try to break into smaller pieces the main problem and describe what steps required to solve it properly. Possible solutions are seen as workable algorithms at the beginning a natural language/pseudocode writing in a text form (**G1**).
2nd session: Familiarizing students with the use of fundamental programming concepts. Link abstract thinking concepts through concrete game experience.	Problem identification and decomposition into a collection of intermediate sub-goals.	**Formulating subparts of the main problem that is visualized in the game:** Analyze alternative pathways which are followed. Students need to understand how the robot can move between other visual objects inside OpenSim (**G2**).
3rd session: Learning how to program using fundamental programming constructs such as sequence, iteration, and selection combined with several variables and the basic arithmetic operators using the S4SL palette.	a) Abstraction and data representation as steps to create algorithms. b) Design and implement a solution to all sub-goals of the main problem.	**Developing and using abstractions:** Designate the movements of an object, by exploring the spatial layout of each stage based on objects/elements. Try to transmit a solution into the code for the object's movements and observe the results during the run-time. Two are the most prominent questions that need to be answered: (i) *Can be applied correctly a solution that is expressed in natural language based on the proposed instructions and rules?* (ii) *Is it easy to transform a solution plan into the code to observe how the programming constructs are integrated and executed correctly into visual elements?* (**G3**)
4th session: Expressing proposed solution plans using programming constructs by creating reusable subprograms.	Automation requires practice in the run-time mode the proposed steps using programming constructs and specific instructions.	**Expressing algorithmic design solutions:** Develop step-by-step instructions that need to be followed for solving each of the subparts of in-game problems. Students need to express as pseudocode any potential solution using small pieces of instructions/movements and programming constructs (**G4**).
5th session: Applying students' solution plans into the code and integrate the most appropriate constructs combined with	Testing and debugging processes: Create efficient and repeatable design patterns as workable algorithms.	**Recognizing and defining the correctness of solution plans:** Students need to apply the entire solution plan according to the given

specific into the in-game visual objects (RVC).		instructions and detect any potential errors (debug) logically by executing programming commands and constructs blocks via S4SL (**G5**).
6th session: Examining students' solution plans (code tracing analysis) by identifying the most effective and efficient design patterns so as to announce the winner(s).	Simulation and parallelization: Problem generation and pattern generalization.	**Generating the appropriateness of the most effective and efficient design patterns:** The instructor needs to examine by benchmarking the proposed design patterns. What differences can be observed for scoring better in the game according to the proposed design patterns? Discuss with other peers and with the CS instructor (**G5**).

An instructor was attended to all sessions in the conventional computer laboratory and in OpenSim. Initially, even before the beginning of this study, the instructor needed to establish and ensure students' access in OpenSim and S4SL, in both computer laboratory and online courses, with the purpose of resolving any technical issues and allow them to participate seamlessly, like doing their homework. Therefore, the instructor has also the responsibility:

a) to attend all courses (face-to-face and/or supplementary online) and assist students' efforts in several coding tasks,
b) to give the appropriate feedback for the compilation or execution of any detected errors into code to syntax correctly their solution plans, and
c) to access on users' actions, either online via Open Sim or offline so that provide a general understanding of how students start thinking about solving sub-goals of the problem before starting to code.

7.2.3. Instrumentation and data analysis

A mixed-methods study was followed for assessing the experiential dimensions in the current preliminary study in favor of validating further its results. At the end of this experiment, quantitative data were gathered through close-ended self-reporting questionnaire responses of participants (Bargas-Avila & Hornbæk, 2011) given the option of writing short comments (Table 7), whilst maintaining their anonymity and confidentiality (see Appendix A, p. 219). Their answers analyzed according to the guidelines for user experience studies (Tullis & Albert, 2013). Supplementary, qualitative data were collected through open-ended interview questions to understand students' enchantment and engagement using the RVC simulator (see Appendix B, p. 222).

To assess the user experience, this study followed the research considerations by Bargas-Avila and Hornbæk (2011) who identified several aspects of experiential dimensions that should be utilized. All statements in this work are expressed and rated simply on a 5-point Likert scale (strongly disagree-1 to

strongly agree-5). The items about the procedure for measuring student learning experience was based on 16 questions, translated to Greek and separated in three subparts: learning effectiveness (LE), learning procedure (LP) and user experience (UX). Subparts about students' learning outcomes and experiences concerned with issues that are ubiquitous in respective work. More specifically, all identified aspects (aesthetics of interaction engagement, usability, usefulness, visual appeal) related to user experience (Bargas-Avila & Hornbæk, 2011). Cronbach's alpha (α) of the main questionnaire was 0.835, reflecting on a reasonable internal consistency of the variables to describe students' expectations. More specifically, data were analyzed using:

a) guidelines for usability metrics so as to evaluate the user experience (Tullis & Albert, 2013), including each user's response to the top-2-boxes (positive responses) or the bottom-2-boxes (negative responses),
b) probing questions from the instructor provided feedback by posing questions to each participant when s/he seemed to get confused helping them find an adequate direction to propose a solution, and
c) code tracing analysis via S4SL palette, the instructor evaluated the applicability of algorithmic control flow to identify whether the adoption of selection control flow blocks and the exploitation of nesting composition among programming constructs were achieved.

7.2.4. Results

Regarding the participants' background based on demographics information, almost more than half percent (55%) of them found really important their participation in CS courses with reasoning and learning capabilities to be the implementation of various tasks using programming environments. Most of them (60%) had previous experience with Scratch. Some of them (20%) answered that they knew about SGs, such as "*The Sims"* or "*Minecraft"* and some others (33%) who had utilized them.

Table 7-2: Short comments on how the proposed simulation game contributing to the learning effectiveness, learning procedure, and user experience

Learning effectiveness (LE)	(a) Roleplay scenario [*n*=8, 54%]	(b) Exploration and problem description [*n*=2, 13%]	(c) Learning objectives [*n*=2, 13%]	(d) Chat or voice communication [*n*=2, 13%]	(e) Visual feedback [*n*=1, 7%]
Learning procedure (LP)	(a) OpenSim and S4SL [*n*=5, 40%]	(b) Instructor's feedback [*n*=4, 30%]	(c) Game context [*n*=2, 10%]	(d) Understanding of user control in the game [*n*=2, 10%]	(e) In-game visual elements [*n*=2, 10%]
User experience (UX)	(a) The game setting (RVC, 5 rooms, visual objects, etc.) [*n*=5, 30%]	(b) In-game problem recognition accuracy [*n*=3, 20%]	(c) Interactivity with visual objects [*n*=3, 20%]	(d) The 3D graphical user interface [*n*=2, 15%]	(e) The anthropomorphic avatar [*n*=2, 15%]

The vast majority of participants reported on several points of view about the RVC simulator. In *Figure 7-3*, the top-2-box scores include responses to the two most favorable response options, i.e. ranking percentage based on their answers was e.g., from 87% (13 out of 15 students) about expressing and applying their solutions to 67% (10 out of 15 students) about decomposing in subparts the main problem. Slightly more than half of them (54%) referred that roleplay scenario and problem description contributing to LE (*Table 7-2*).

A student reported that *"some facts in the game are really represented well. This helped me not only to rationalize my decisions by applying and explaining my solution but also to know why I used some programming constructs without only proposing "zigzag" movements as cleaning pathways".* Another one said that *"S4SL helped me to apply a proposed solution, as I visually saw the results of the code inside OpenSim".*

In contrast, other users could not easily recognize the interaction between elements inside the house (Visual feedback: 7%) and one of them complained that *"I struggled sometimes to understand if the robot collided with house furniture or objects, when I was applied for my program",* albeit in the end their preference than Scratch or Alice was referred. The use of communication tools to succeed the learning objectives was mentioned less by a few users (13%), maybe due to the instructor's feedback in face-to-face tasks.

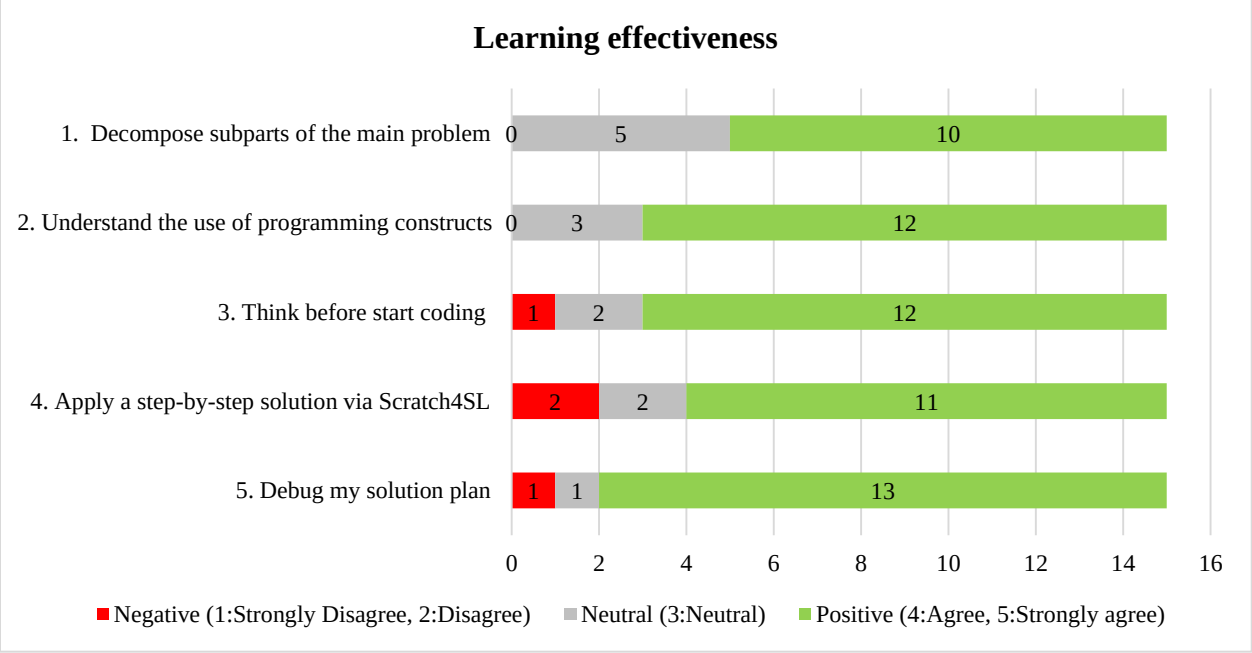


Figure 7-3: Horizontal stacked bar chart of top/bottom-2-boxes of users' responses about the learning effectiveness

In terms of LP, again many participants were at the top-2-box scores. The ranking percentage based on their answers was e.g., from 73% (11 out of 15 students) on understanding instructor's feedback to 53% (8 out of 15 students) for the effective communication and successful implementation of design patterns for proposing solutions to each subpart of the main problem (*Figure 7-4*). Others reported on several points of view in regard to the SG that contributed to the LP (Table 2) with the most notable to be the combination of OpenSim with S4SL (40%). After the game context, understanding of in-game user control and visual elements follow with 10% to each. The combination of OpenSim and S4SL was necessary for integrating behavior inside the robot to follow a cleaning path and getting responses of its movement, in an effort of proposing and applying visually solutions through design patterns.

The phase of programming to visualize a proposed solution was referred by others as an important feature, especially because it enables them to assess their thinking process: *"The S4SL palette enabled me to write correctly the code, while I was previously described and proposed a solution in natural language".* Another one participant referred that *"the instructor guided my practices and he helped me with the code responses in order to be applied my solution plans".*

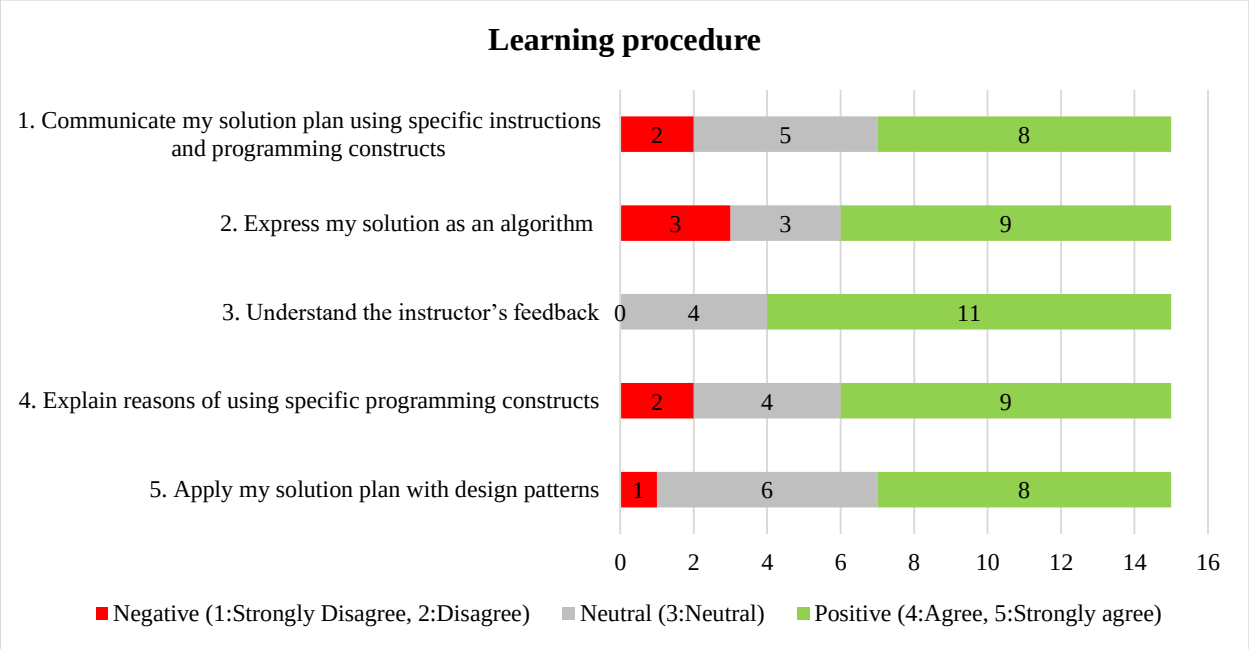


Figure 7-4: Horizontal stacked bar chart of top/bottom-2-boxes of users' responses about the learning procedure

With respect to the UX, most participants were at the top-2-box scores (*Figure 7-5*). For instance, the top-2-box score is 67% (10 out of 15 students) of students who felt engaged with the VRC simulator rating it favorably compared to their counterparts who have an opposite opinion according to a bottom-2-score of

13% (2 out of 15 students). Participants reported on several aspects of the SG, which contributed to positive user experience (Table 2) with the highest to be the game setting (30%).

The anthropomorphic avatar representation and the 3D GUI follow with 15%. A representative answer reported that "*It was a motivating setup of playing in-game tasks".* Other one said, *"In past, sometimes I did not have the opportunity to present my code and speak of why I used some programming constructs"*.

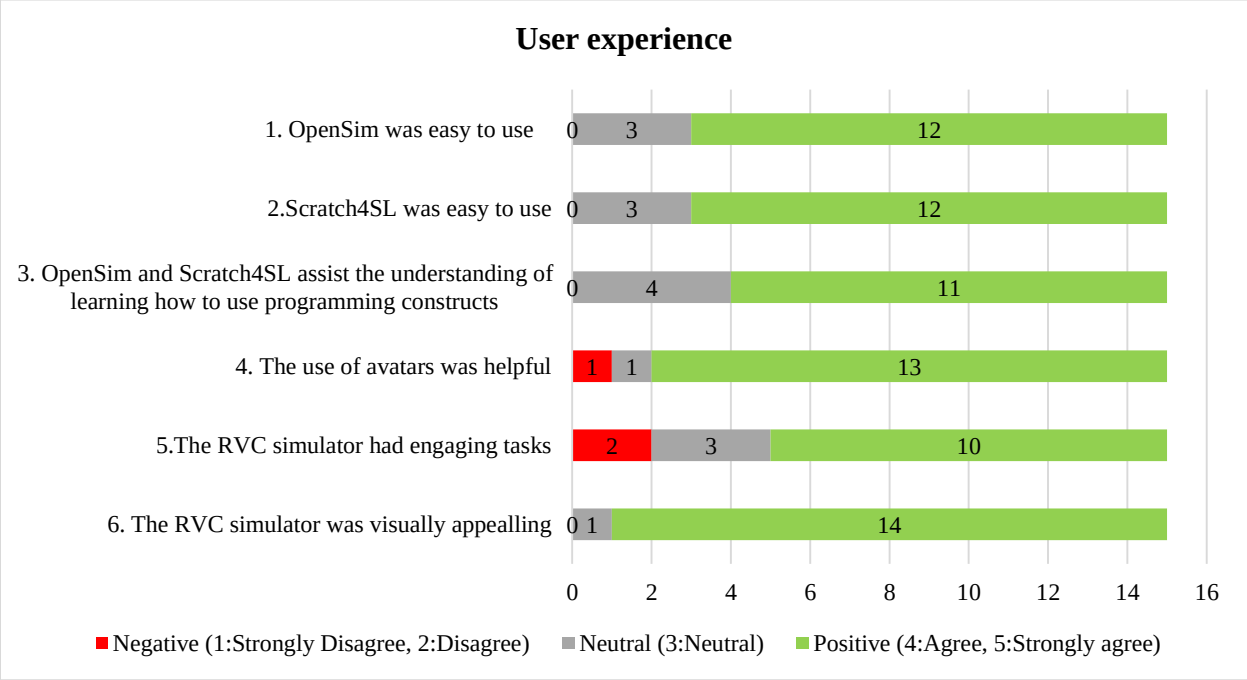


Figure 7-5: Horizontal stacked bar chart of top/bottom-2-boxes of responses about user experience

Negative aspects of the UX were also reported about interactivity among visual objects (15%), like *"When the robot stroked a table or a sofa, sometimes I did not recognize the error message, maybe because of the poor quality of graphics".* Few users struggled to log into OpenSim, said that *"I was observed slow loading times in my entrance"* at the beginning or others did not copy and paste correctly the code into the notecard of RVC.

7.2.5. Discussion

The main purpose of this preliminary study was to investigate the effectiveness of a 3D SG to programming high school course settings. The RVC simulator provides affordances with instructive guided support through informal blended instruction to CT teaching. Furthermore, it enables the free experimentation and reflection of students in a concrete problem-solving space by exploring and expressing solutions through design patterns. Their answers revealed the positive acceptance of how instruction using

S4SL and OpenSim engaged them in innovative and interactive learning situations since they had very satisfactory performance and user experience. Findings of this preliminary study unveiled that a great number of students found the proposed 3D SG interesting, fascinating and relevant to their previous experience with other SGs, like "*The Sims"* or "*Minecraft"*. Without so highly advanced, but with simple design patterns to be nested and presented as final solutions, students appeared not having any difficulties in producing some good computational problem-solving practices. Based on code tracing analysis, the applicability of selection control flow blocks and the exploitation of nesting composition among programming constructs, for instance, such as mastering if/else conditionals with numbers using S4SL, students were able to propose well-defined solutions and learning outcomes that could be easily visualized in OpenSim. Consistent with Howland's and Good's (2015) study findings, a block-based palette is regarded as a reliable tool for high school students to avoid syntax errors in programming and trigger more in problem-solving via 3D roleplay games by expressing and applying more succinct and precise rules with instructions in combination with programming constructs.

On the other side, contrary to the results of past efforts (Brennan & Resnick, 2012; Mouza et al., 2016), students of this study using a 3D SG seemed to have reasonable efforts by answering why they used specific programming constructs and/or instructions in computational practices, dodging the vague syntax of programming constructs and commands. Such a process can give valuable answers for assessing how students try to think and practice computationally before starting to code. This can also give evidence of a deeper understanding of the description of a cognitive thinking process to the comprehension and production of coded solutions.

Despite the small number of participants in this preliminary study, their answers from the close-ended questionnaire, interviews and code analysis can give important educational aspects. Therefore, as regards the LE:

a) The learning outcomes have been achieved with particularly encouraging evidence arising from the code tracing analysis via S4SL.
b) A few students seemed to face problems or report issues or report any issues according to the technological requirements of the SG created by using OpenSim and especially as regards their attempt to complete all in-game stages.
c) Any particular difficulty in compiling and applying their solution plans into code did not prevent all participants to complete successfully their activities required within specific time frames.

With regard to the LP:

a) The spatial presence of objects/elements in three-dimensions using OpenSim assisted participants to separate and explore easier all subparts within problem-solving context existed in each stage.

b) The natural-intuitive modality for user-interaction simulation tasks helped participants both to better analyze the components of a computational problem and propose effective solution plans to be applied their design patterns.

Regarding the UX:

a) The participants' satisfaction with the user interface features and overall enjoyment of the activity was at a high-level. It seemed that was positively associated with their engagement to learn by playing the RVC simulator and the technical characteristics (e.g., audio-motion quality and 3D visual in-game objects and elements).
b) The participants' navigation inside OpenSim was ease using a/-synchronous communication tools which are associated with the use of a keyboard and a mouse to play with and code;
c) Camera and object handling for the integration of visual behavior via coding was not considered as difficult, although some participants at the beginning have only mentioned in their comments that they were having some minor difficulties using specific tools;
d) Participants considered the presence of the main researcher as important, while in most cases they did not consider it necessary in online supplementary instructional formats.

In addition to the above, in online programming courses, participants were satisfied by the natural-intuitive modality for user-interaction simulation tasks inside OpenSim, but they did not find any possibilities of verbal communication except in case of communicating with the main researcher. Nonetheless, they found helpful both verbal and non-verbal communication tools to communicate with the instructor.

This preliminary study's findings may be of interest to instructional designers who want to take in advance a 3D SG and design (in-) formal introductory programming courses in blended instruction to foster students' computational problem-solving practices. The utilization of the proposed SG made students able:

a) to think critically and logically so as to organize code blocks design patterns and execute programs for a simulated real-world problem,
b) to understand easily all evocative spatial metaphors from the different spatial layout that room has inside OpenSim, referring from almost all of the different computational practices in coding, and
c) to succeed learning outcomes and achievements which have affected positively their overall performance in order to apply easily their thinking solution plans into code.

7.2.6. Limitations

The current preliminary study has the following four limitations:

a) The sample size of participants was too small (n=15).

b) The 6 time-intensive teaching intervention was completed in informal settings (after-school) sessions and into realistic school context conditions.
c) The researcher's feedback on participants' actions inside OpenSim, especially in the online sessions was daily.
d) All participants had personal computers and laptops which supported even the most advanced requirements for gaming. Therefore, any technical problems did not prevent any of them to attend to all sessions of this teaching intervention.

7.3. Research methodology of the quasi-experimental study

A mixed method study employed was an embedded approach with an experimental design. In this study, a quasi-experimental design was followed as a research method, with intervention and comparison groups to be tested their learning performance with pre-and-post-questionnaires and post-tests including worksheets and error analysis rubrics with specific criteria. Since a mixed methods approach was chosen, both quantitative and qualitative measures are employed in the present study, in addition with a semi-structured interview and a think-aloud protocol to be gathered data (Cohen et al., 2011). This approach was used as the majority of empirical studies following GBL approaches solely presented results from a quantitative approach. Therefore, as in their review, Lye and Koh (2014) have suggested that further studies need to give a more comprehensive picture of the topic and provide insights from the combination of quantitative and qualitative findings.

Based on the above, the current study used a nonequivalent control group design with pre-and-post questionnaires and post-tests. Firstly, it was important before conducting the experiment to identify the difficulties faced by students on how they use and apply basic programming constructs and concepts with the intention after that to create two groups. The measurement of students' learning performance was made by measuring:

a) computational understanding with the use of (pre-and-post questionnaire) questionnaire that is proposed by Korkmaz et al. (2017),
b) worksheets in relation to error rubric analysis criteria at the end of in-game tasks to describe their proposed solutions firstly in short sentences and in pseudocode for each stage superlatively (see CT 1-4 from *Table 7-3*),
c) on code tracing analysis for the applicability of selection control flow blocks and the exploitation of nesting composition among programming constructs, such as mastering if/else conditionals with numbers using S4SL or Scratch respecting to each group (see CT 5-6 from *Table 7-3*). Students should be able to propose well-defined solutions and learning outcomes that can be easily visualized in Scratch or S4SL.

Following Cohen et al.'s (2011) guidelines as a method research design, N represents non-randomization, O1 represents pre-questionnaires and pre-tests (i.e., questionnaires that participants are required to complete prior to the implementation of a treatment), X represents the implemented treatment (i.e., the OpenSim with S4SL adoption for one group and Scratch for the second group), and O2 represents the posttests (i.e. worksheets and error analysis rubrics). Both the control group (CG) and the experimental group (EG) completed pre-and-post-questionnaires and post-tests after the intervention; however, the experimental group was the only group that was received the research treatment. Nonequivalent control group has been described by Cohen et al. (2011) as "*one of the most commonly used quasi-experimental designs in educational research*" (p. 283) and it is represented below:

Experimental Group:	N	O1	X	O2
Control Group:	N	O1		O2

Participants were not randomly selected and not randomly divided; thus, the research method in this study is regarded as quasi-experimental. Using non-equivalent group designs, different groups receive different treatments and the effectiveness of a treatment is evaluated by comparing the performances of the two groups. Such a research design method requires pre-questionnaires, in furtherance of having an indication of how similar the two groups (control and the experimental) were before the intervention and post-tests for both groups after this teaching intervention.

Although a comparison group should be as alike as possible in as many dimensions as possible, the assignment of participants in the two groups was deliberately non-randomized. This decision was deemed necessary to be avoided any possible biases in this study's results, as it was difficult to randomly assign scholars to different schools about the control and treatment groups not only in general (Slavin et al., 2007), but in specific, it is needed a gender equality for CT instruction (Grover & Pea, 2013; Werner et al., 2015). The different CS instructors from the three different classrooms and programming environments used in their courses were crucial factors. Also, the assignment of participants to the two groups was non-randomized because it was needed the experimental group to be comprised of experienced to OpenSim with S4SL users for two reasons. The first is to understand whether high school students would be able to operate the OpenSim effortlessly, and the second is to minimize the novelty effect.

7.3.1. Setting and sample

This study was conducted in an intensive 4-weeks period with 6 sessions as described in Table 2. The first 2 sessions lasted 40 minutes inside the computer laboratories of the three high schools. The other 4 sessions lasted 40 minutes inside the computer laboratories on a University campus, and specifically Department of Product and Systems Design Engineering (DPSD), in which two computer laboratories were

formed to be alike as the conventional instructional conditions inside a school. Thence, each student had his/her own desktop computer in two different computer laboratories. One laboratory was used for each group, where either Scratch or OpenSim was installed in standalone mode in order to prevent any potential misconceptions among students' answers and evaluate the learning performance for each one separately. The conventional or similar to the aspects of regular instructional settings can give several potential benefits on how each interactive environment may be used by CS instructors in the future. Such instructional settings will be more valuable to CS instructors who may want to use the proposed interactive environments in the same instructional conditions rather than into conditions that any researcher wished to use the proposed SG which could be more appropriate to extract this study's results more widely in the educational community. An overview of using Scratch and OpenSim with S4SL and basic instructions and information about the RVC simulator were presented to each group.

The present evaluation study approved by the University of the Aegean Ethics Committee (No. Protocol: 7515/4-12-2017). In addition, before initiation of the research phases that described earlier, all necessary permissions were taken by the Greek Ministry of Education, Research and Religious Affairs (No. Protocol: 226058/D2/21-12-2017), and informed consent needed to be obtained from all participants and their parents (or their legal guardians).

After completing the questionnaire regarding the gained information from students' demographics and level of difficulties in CS concepts, they were split into two groups to be considered as similar as possible. The sample comprised of 24 girls (M_{age}=14.37, SD=1.55) and 26 boys (M_{age}=14.44, SD=1.48) who volunteered participate, and they were from three Greek local schools. Thence, a total of fifty (n=50) participants were recruited to attend in all formal (inside the class) and informal (inside the University campus) sessions. To potentially increase the diversity of the participants' opinions, it was imperative to ensure not only the heterogeneity on their gender and background about programming courses participation (demographics) but also the homogeneity of each group with participants who scored across all ranges in the pre-questionnaire adopted by Lahtinen et al. (2005). The two groups differed on the interactive environment that used, i.e., Scratch for the control group (CG), which consisted of 25 participants (boys, n=13, girls, n=12), and OpenSim with S4SL for the experimental group (EG), which consisted of 25 participants (boys, n=13, girls, n=12).

Since this study had a non-randomized sample, there were key concerns about methods of conscious control of implicit attitudes between male (boys) and female (girls) participants. Also, it was imperative to ensure the gender balance for both groups and so the same number of participants needed. For example, calling attention to gender may increase unconscious or implicit biases, even if the purpose of making participants' gender salient to avoid that gender influence (gender discrimination). Finally, before starting the experiment and without getting assigned randomly students in only one of the two groups, it was

appropriate to dodge any potential fellowships and friendships or to eschew perceptions about the level of difficulties on learning computer programming. The following figures depict an instructional process in-school and in the computer labs of the University campus (see *Figure 7-6, Figure 7-7, Figure 7-8, and Figure 7-9*).

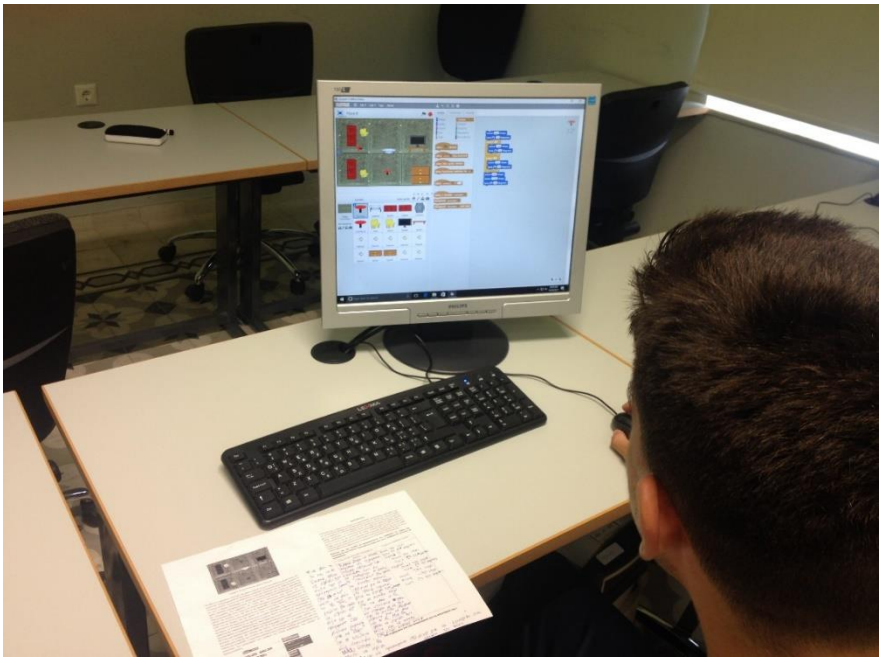


Figure 7-6: A boy from the control group plays the RVC simulator using Scratch

Figure 7-7: A girl from the control group plays the RVC simulator using Scratch

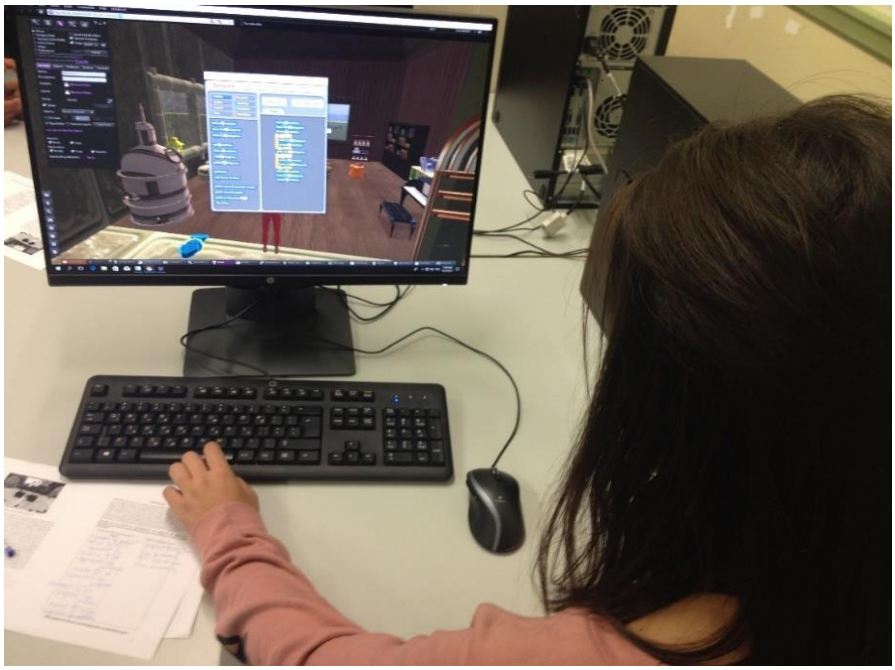


Figure 7-8: A girl from the experimental group plays the RVC simulator using OpenSim

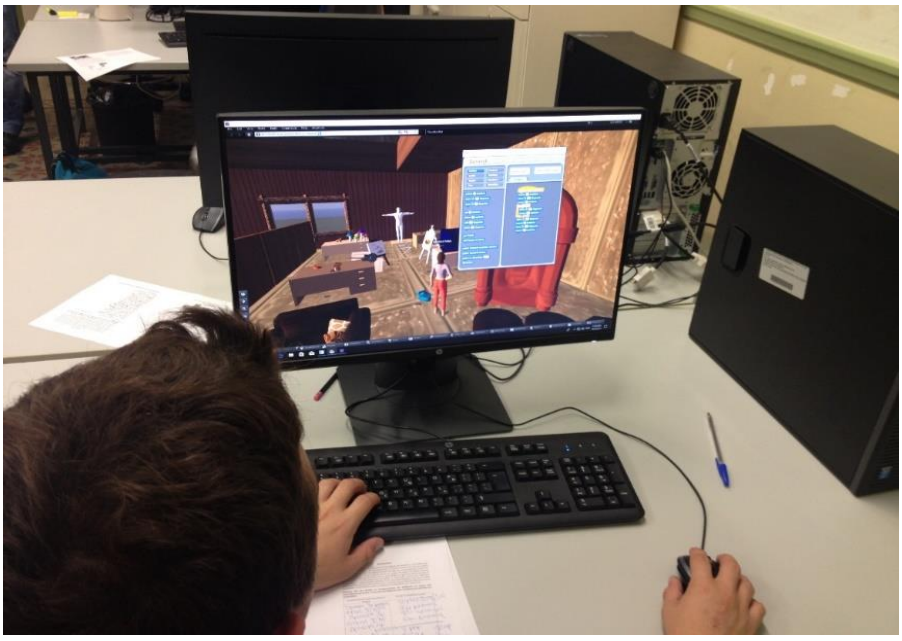


Figure 7-9: A boy from the experimental group plays the RVC simulator using OpenSim

In this study, a nonequivalent control group design with pre-and-post questionnaires and post-tests through worksheets were used. Thus, it was important before conducting the experiment to identify any potential difficulties that might students face regarding how they use and apply fundamental programming constructs and concepts. The purpose of such an effort was the creation of two groups respecting their different background on programming knowledge and gender to avoid possible biases.

All in all, the three CS instructors who had the responsibility for teaching the theory about the use of programming constructs were in collaboration with the supervising researcher in order to:

a) provide feedback by posing questions to each participant when s/he seemed to get confused and helping them find an adequate direction to propose a solution and
b) assess through code tracing from the palette of S4SL or Scratch focusing on the applicability of algorithmic control flow so that identify whether the adoption of selection control flow blocks and the exploitation of programming constructs and commands is achieved properly.

7.3.2. Experimental setup

The experimental setup of the quasi-experiment is shown in *Figure 7-10*. At the beginning of the learning activity, all participants took two pre-questionnaires adopted by Lahtinen et al. (2005) and Korkmaz et al. (2017) for gathering data about the difficulties existed on learning programming and about their self-report regarding the cultivation of skills related to CT based on their previous game playing experiences.

The pre-questionnaire aimed to understand the background information of the participants and assigned them to the two groups fairly by examining participants' demographic information, study habits, game experience, and prior programming knowledge. This stage was crucial to determine the homogeneity of the participants and to verify that they all had a similar science-related background before the experimental instruction. All integrated behaviors were recorded by the researcher for further analysis. Each part of the solution is represented by an instruction card which is downloadable as *.sb* and *.sb2* files from the palette of Scratch and S4SL respectively to investigate the correctness of programming behaviors through a code tracing analysis that integrated into visual elements. For each sub-goal, each novice created an instruction card and assembled the visualized instruction blocks to implement plans for the sub-goals.

During this teaching intervention, the CG used Scratch and the EG used OpenSim with S4SL to play the SG. After the learning activity, all students took the post-questionnaire of Korkmaz (2017) and completed as well as worksheets (post-tests) to propose in natural language and apply into code their solution plans to all subparts of the simulated computational problem.

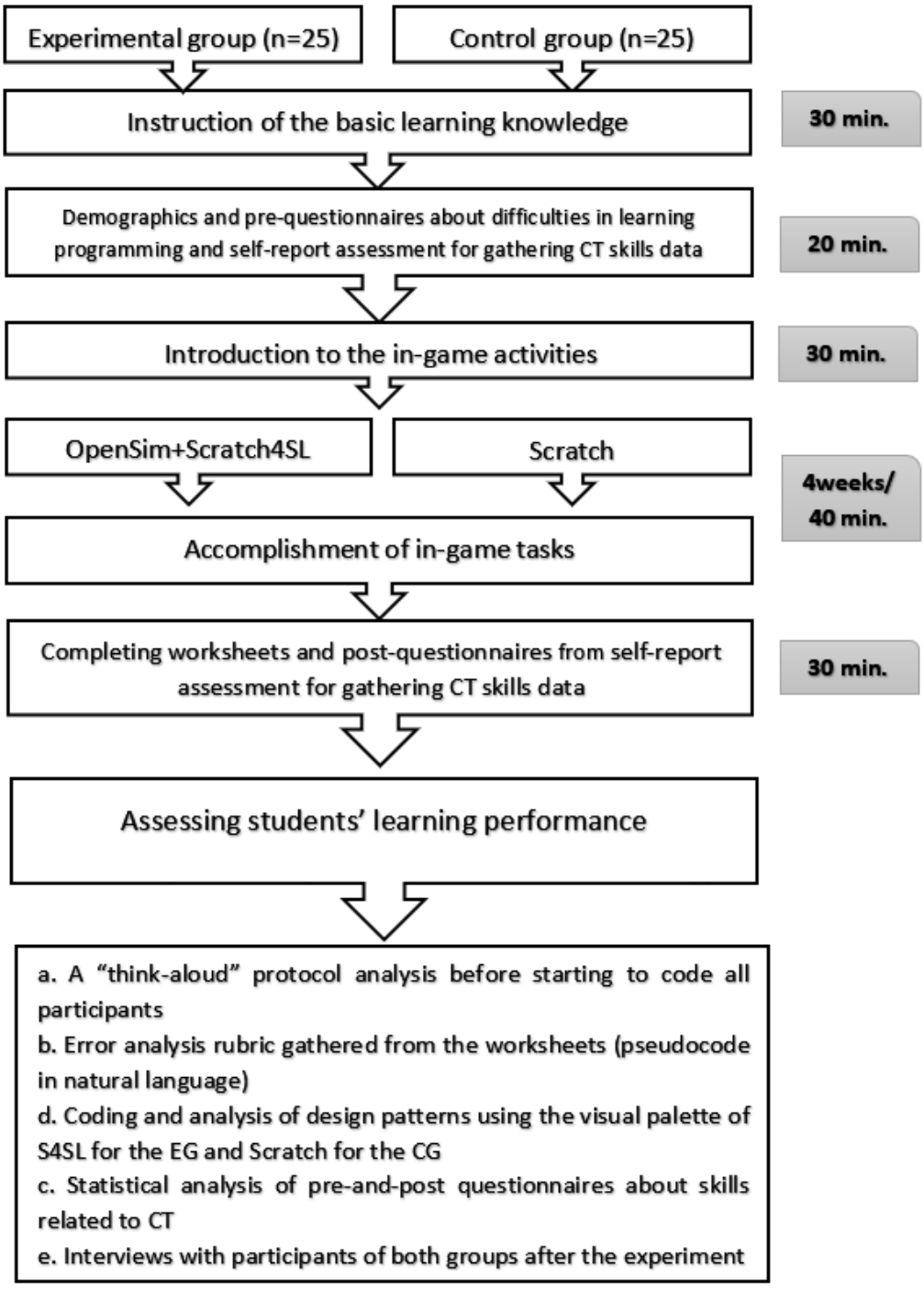


Figure 7-10: The quasi-experimental procedure

7.3.3. Procedure

To operationalize the CT instructional approach for the purposes of this teaching intervention, six sessions in *Table 7-3* are provided corresponding to the six core dimensions of the broader CT conceptual framework. The proposed in-game tasks are associated with concepts and skills related to CT may be predominantly helpful for instructors or educators who design (in-) formal instructional contexts using the RVC simulator to foster students' computational problem-solving strategies. The programming tasks took place inside formal (e.g. computer school labs) and informal (e.g. University campus) instructional settings. *Table 7-3* presents the design of tasks from this teaching intervention with the proposed SG consisted of:

a) the learning tasks associated with the operational definition of CT as a problem-solving process with specific learning objectives (CSTA & ISTE, 2011) combined with the skills related to CT such as problem-solving, algorithmic thinking, creativity and critical thinking (Korkmaz et al., 2017),
b) the proposed design guidelines (G1-G5) from Pellas and Vosinakis (2017a) about the creation of the RVC simulator, and
c) the in-game instructional contexts that can assist students to express and apply computational problem-solving practices. From the 1st to the 4th session [CT 1-4], the study was conducted in computer school labs and the final two sessions [CT5-6] at the DPSD campus (Pellas & Vosinakis, 2018).

Table 7-3: In-game activities associated with operational characteristics and skills related to computational thinking

Sessions	**1st session [CT1]**	**2nd session [CT2]**	**3rd session [CT3]**	**4th session [CT4]**	**5th session [CT5]**	**6th session [CT6]**
The operational definition of CT	Formulating problems	Logically organizing and analyzing the data	Representing data abstraction to become simplified the main problem	Automating solutions through algorithmic thinking	Identifying, analyzing and implementing possible solutions	Generalizing and transferring a problem-solving process to propose a solution
Skills related to CT	Problem-solving	Critical thinking	Abstraction	Algorithmic thinking	Design-based and creative thinking	Pattern Recognition
Proposed instructional guidelines	Student motivation (G1)	Student active participation (G1)	Simulation of an authentic problem (G2)	System's feedback on the user's actions (G3)	Development of computational practices (G4)	Applying design patterns (G5)
In-game activities (Students should be able to…)	Decompose in subparts the main problem	Analyze a cleaning path and describe the robot vacuum cleaner (RVC) movements	Designate the RVC's movements in the spatially-explicit context	Transform a solution to the algorithm and debug by finding errors preventing syntactic/semantic to make the	Proposing and creating a step-by-step algorithmic solution to coding	Implement and examine the effectiveness of the proposed design patterns

				refinement of problem-solving strategy an easier process.		
Students' learning outcomes	(a) Understan-ding how to separate in subparts the main problem to manage it and propose design patterns easier (b) Organizing the data	(a) Gathering appropriate information and selecting relevant information (b) Conceptua-lizing precise instructions and rules which students can use in order to propose an algorithm	(a) Describing common behaviors or programming constructs between different scripts. (b) Identifying abstractions in the digital environment	(a) Communica-ting a step-by-step algorithm. (b) Exemplifying why a proposed algorithm can become effective for a problem. (c) Discovering how effective is a proposed algorithm.	Developing and rationalize decision made to propose solutions through coding	(a) Creating simulations by executing design patterns, (b) Generalizing a proposed solution to a specific problem that was given and amplifying by the demonstration as a design pattern

Table 7-3 associates with a process on how students can develop their skills in gameplay with the previously defined CT skills from the aforementioned analysis so that supporting computational problem-solving development through in-game settings.

7.3.4. Instruments

The measurement of students' learning performance was made firstly through worksheets in relation to error rubric analysis criteria at the end of all in-game tasks to be expressed their proposed solutions, in short sentences, and secondly to be described as pseudocodes/algorithms (see CT 1-4 from *Table 7-3*), and thirdly to be applied as code their strategy for solving subparts of the main computational problem. A code tracing analysis related to the applicability of control flow code blocks to be exploited the appropriateness and correct execution of programming constructs and commands. This was a criterion about their correct (or not) use, such as mastering if/else conditionals with numbers using S4SL or Scratch respecting to each group in an effort to increase the validity of the conclusions referred and written in natural language (see CT 5-6 from *Table 7-3*).

To measure students' learning performance based on final design patterns, quantitatively, this study followed Chao's (2016) coding framework analysis. Also, pre-and-post questionnaires based on Korkmaz et al. (2017) were used to determine the level of skills related to CT. Since each student had his/her own PC and a specific nickname (user 1, user 2…etc.), all pre-and-post questionnaires and worksheets were answered anonymously protecting any confidential information. Also, all participants needed to give answers inside each school computer laboratory for completing the pre-questionnaires and inside the DPSD computer laboratory for any given post-test and post-questionnaire in order to be further ensured the

anonymity since it was unable to be identified any IP (Internet Protocol) address from someone's personal computer. All tests and questionnaires were translated into Greek. In particular, the instruments which were used for data collection are the following:

a) Student profiles and demographics in an individual questionnaire were administered at the beginning of the proposed teaching intervention. The questionnaire recorded some simple demographic data, such as student gender, background on computer use for example, like the frequency of computer use, computer experience and knowledge on creating and/or playing games in learning programming (see Appendix C, p. 223).

b) A closed-ended pre-questionnaire was adopted by Lahtinen et al. (2005) so as to understand the major difficulties that students face in how using and applying fundamental programming constructs and concepts before the experiment. It consists of 4 items evaluated using a 5-point semantic differential scale before the experiment (see Appendix D, p. 225). This questionnaire is the most appropriate to identify the perceived difficulty in programming courses and knowledge gained by using programming environments for students at the high school level (Koorsse et al., 2015). With this questionnaire, the separation of all participants based on their answers from the pre-questionnaire adopted by Lahtinen et al. (2005) was made. The second criterion for the separation of the two groups was the demographics of participants, their previous experience and difficulties regarding programming. With a view to increasing the diversity of their opinions, each group included participants who had not only different perceptions/opinions about programming but also those with different demographic characteristics. This decision was necessary to avoid the creation of any group of participants who may have the same gender and/or the same perceptions since their assignment was deliberately non-randomized.

c) A closed-ended (pre-and-post) questionnaire proposed by Korkmaz et al. (2017) was handed out from all participants from the two groups to fill it before and after the completion of the teaching, intervention to determine their personal opinion regarding the level of skills related to CT. The validity and reliability of a questionnaire named *"Computational Thinking Scales (CTS)"* that is proposed by Korkmaz et al. (2017). This questionnaire offers self-reported measures about participants' views on how they have tried to determine and use skills related to CT. There are appeared five components in regard to the CTS questionnaire. For the purposes of this study, the component of cooperativity was excluded since there was any activity to support collaboration among participants. Only four components comprised four questions about skills related to CT are used, for the following four components: critical thinking, algorithmic thinking, problem-solving and creativity (see Appendix E, p. 227). Nevertheless, three were the questions that used creativity. The adaption of this study's questionnaire was made according to Korkmaz et al.'s (2017)

suggestions and guidelines. In specific, the same authors have argued that the CTS questionnaire is relevant to participants who may come from different education levels and age groups. Since Korkmaz et al. (2017) have provided validity and reliability of each component, further studies have an opportunity to choose either to use each one of the five-factor components from CTS questionnaire separately or all of them as a whole (Korkmaz et al., 2017). The adopted CTS questionnaire was given to each participant of the two groups before and after the teaching intervention. For this study's purpose, the questionnaire consisted of 15 items with a 5-point Likert scale for four items that described earlier, ranging as *"(1) never", "(2) rarely", (3) sometimes", "(4) generally," and "(5) always"*. An indicative example of a question that has been adopted is the following: *"I have difficulties to demonstrate my proposed solution for a problem".*

d) A "think-aloud" protocol used to analyze and examine in more depth the computational practices and perspectives in which students verbalized their thought process while programming on-screen tasks in the interest of rationalizing their computational practices (Lye & Koh, 2014). Before starting to code, all participants were individually asked to describe the way that they would like to follow for solving each of the 3 chosen stages.

e) After the data gathering activity from the think-aloud protocol, a semi-structured interview at the end of the entire intervention was made in the University campus. Participants had the chance to express their opinion freely was used aimed to provide supplementary responses to the activities described in the previous questionnaires (see Appendix F, p. 229).

f) An error analysis rubric was given after this teaching intervention to all students individually in order to complete certain tasks. Each student was asked to describe and write in worksheets his/her proposed solutions for each subpart of the main problem (see *Table 7-4*), firstly by describing a solution in natural language, and thereafter in coding via Scratch for the CG (see Appendix G, p. 230) or via S4SL for the EG (see Appendix H, p. 236) and. The assessment of the student's proposed solution was based also on the same graded criterion instrument. The identification and interpretation of students' common error patterns due to the misconceptions about their achievements can provide diagnostic information about their strengths and weaknesses in expressing and/or implementing a proposed solution. The error analysis rubric items (see *Table 7-5 and Table 7*-6) challenged students to analyze, diagnose, and provide targeted instructional remediation. It intended to help them overcome common error patterns and misconceptions, i.e. logic errors through the expression of the algorithm as pseudocode in natural language. Such an instrument was also followed by Howland and Good (2015) and it is regarded as essential in order to be determined students' computational understanding and concepts in terms of expressing their solution plans for simulated problem-solving tasks.

g) During the teaching intervention, systematic monitoring of the students' work was applied by taking notes in a structured form (observation sheets). Both the supervising researcher and CS instructors filled in the sheets and then extensively discussed their observations to reach consent and decide on their importance.

7.3.5. Data analysis

An initial analysis of short sentences in natural language was conducted by looking at students' descriptions as computational rules and concepts for the creation of algorithms in natural language before starting to code. *Table 7-4* shows some indicative examples of the describing rules which are segmented into subsections according to the computational constructs that students need to represent as encoded solutions (design patterns). In this direction, for each rule and concepts section, an error rubric analysis in order to be identified the correct and incorrect variants of the computational rule sections was used. The proposed model below seeks to give some answers to inform and guide educators and researchers in regard to the alternative phrasings which preserved the semantic meaning of the rule section with a view of adding (or not) another phrase without changing the semantic meaning. A rule section can be accepted as correct if there is existed an event that could be described completely and unambiguously including the key phrases of the model answer.

Table 7-4: Error analysis rubric criteria

Category	Explanation [code]	Grades
0. Correct	Correct answers are described and implemented correctly without any errors to be identified not only in short sentences expressed in natural language or as algorithms/pseudocodes but also when applied with specific use of programming constructs and instructions into code [C]	0.5 grade for each correct task (CT 1-4) identified in each CT instructional session for each of the 3 in-game stages. Other 6 gained if expressing an algorithm (pseudocode) in the CT5 session can be applied properly into code using programming constructs in the final CT6 session (see all sessions described in *Table 7-3*). As a result, the maximum number of grades that someone can gain is 12.
E1. The errors of commission or errors of omission for the description and understanding of a proposed solution is based on the problem-solving situation that should be expressed	a. Errors of omissions: Some of the key elements for the description of a solution are missing, such as the following: (i) Goals, (ii) Instructions/events/rules, and (iii) Anticipated outcomes [E1.2] b. Errors of commission: Key elements for the description of a solution are totally missing or contain erroneous information [E1.2]	0.5 grade can be lost for any error that is identified in each key element of the first 4 CT instructional stage (see CT 1-4 from *Table 7-3*).

E2. The errors of commission or errors of omission for the description of the algorithm in a simple but rigorous form in natural language and its implementation into code (Testing and Debugging)	a. Errors of omission: Some rules, instructions or programming constructs that need to be used are missing from the algorithm expressed in text form as it is written in natural language [E2.1] b. Errors of commission: Some erroneous information about the rules, instructions or programming constructs description that need to be used are missing from the algorithm expressed in natural language [E2.2] c. Errors of omission: Some rules, instructions or programming constructs that need to be used are missing when a solution plan is applied do not finally exist [E2.3] d. Errors of commission: Key elements for the description of a solution are totally missing or contain erroneous information from the code that need to be finally applied [E2.4] e. Vague description: Description and/or implementation of ambiguous or vague descriptions of the basic elements corresponding to the algorithm and into code are identified in a solution plan [E2.5]	0.5 grade can be lost for errors identified in each key element of the last two CT instructional stage [see CT 5-6 from *Table 7-3*]

Table 7-5 and *Table 7-6* describe how the grading scheme is applied. Specifically, *Table 7-5* shows the marking scheme for a question which asks students to write a simple rule containing a goal and an anticipated outcome. It is described a model answer alongside with notes and scores in order to assist a coder determine any variations on students' answers which could be considered as acceptable. When all tasks completed, data were coded by the supervising researcher and specific guidelines were given to any CS instructor. The inter-rater reliability was determined by using Pearson's *r* in an effort to measure any possible correlation between the scores from the two raters (any CS instructor of each class and the supervising researcher), and *Cohen's Kappa* in regard to the agreement between their error coding. There was a correlation of 0.85 ($p < 0.001$) on scores and a *Kappa value* of 0.78 ($p < 0.001$) on the codes based on the post-tests from the worksheets written in natural language. Such scores indicate both high inter-rater reliability for the scores and high inter-rater agreement in coding tasks since categorical data up to 0.7 is regularly considered as satisfactory (Jonsson & Svingby, 2007).

Table 7-5: Example model answer

Question 4	Marks	Rules	Model rule	Notes
[CT 4]: *Can you briefly describe a step-by-step solution (rules, directions, programming structures and/or limitations) in natural language that the RVC has to follow?*	1	Goal	*"The RVC is placed under the table can make a movement to clean…"*	For example, accept 'Repeat'' (or "Iteration") when the RVC is placed somewhere in the room or another equivalent keyword
	1	Anticipated outcomes	*"Keeping as a root of the small table, the RVC can spin around it in a 4-spiral square cleaning path by turning as well in 180 or 90 degrees but not over 10m"*	Accept that *"the RVC will follow a cleaning path doing 4 squares with common root to be the small table"*

Table 7-6: Example of students' answers and grades

Indicative examples of proposed solutions in natural language [CT 4]	Grading scales (min. 0/2, max. 2/2)	Error code
"Since the robot moves only 5m, I can use an iteration method to break its motion into 4 x 1.25m for each side of the squares in OpenSim (or 4 x 35 for Scratch), depending on the direction that the RVC needs to move (0-180 degrees), without causing damages on its orientation in the floor.	2/2 – Correct description of programming constructs and instructions (Explanation: The spatial infrastructure of the room is considered and the numerical operations for the calculation of distances between objects or other visual elements with the avatar and the programming construct usage are adequately described).	E.0
"I propose to "split" the robot's movement into pieces 4 x 1.25m for each side of the squares in OpenSim (or 4 x 35 for Scratch). Also, each time depending on the direction I want to give behaviour to move without causing damage, I need to define its orientation in the space".	½ - Lack of clear instructions (Explanation: It is considered the spatial layout of the room and the numerical operations to calculate the distance that the robot cleaner has to move, but without proposing any programming construct that can be used).	E.1.1.
"I suggested using an iteration method in order to be rotated the robot around the square floor of the room."	½ - Lack of clear instructions (Explanation: The spatial infrastructure of the room and the numerical operations to calculate the distances in relation to the "cleaning path" that the robot has to follow were not taken into account. However, the programming construct that can be used is not).	E.1.2.
"I suggested that the robot need to be rotated around the 4-square floor and its' continuous movement with 90° (degrees) turning left or right when it is needed".	0/2 - Errors of commission with erroneous information (Explanation: It does not take into account the spatial layout of the room and the numerical operations to calculate the distances with respect to the "cleaning path" that the robot has to move, nor it is clear the programming constructs that can be used).	E.2.2

The data collected by retrieving log data about the students' computational problem-solving strategies. Three different types of assessment were utilized. For this reason, it was decided to allow students expressing their initial thinking about a proposed solution in natural language through short sentences and write the algorithm as pseudocode into worksheets.

Second, one pre-and-post CTS questionnaire as a self-assessment to determine students' opinions regarding the use of skills related to CT based on the components that Korkmaz et al. (2017) have proposed. In favor of planning and extracting this study's results, the Statistical Package for the Social Sciences (SPSS) was utilized to conduct and interpret, firstly, an internal consistency reliability analysis through Cronbach's alpha (*a*) and, secondly, *Kolmogorov-Smirnov* and *Shapiro-Wilk* normality tests for the homogeneity of the variance. Any statistical analysis and interpretation of the main findings have followed the guidelines from Privitera (2017). The Cronbach's alpha (*a*) for each component of the CTS pre-questionnaire from the EG are the following: a=0.81 for critical thinking, a=0.71 for algorithmic thinking, a=0.76 for problem-solving and a=0.73 for creativity. In the CTS post-questionnaire, a=0.86 for critical thinking, a=0.91 for algorithmic thinking, a=0.97 for problem-solving and a=0.93 for creativity. The Cronbach's alpha (*a*) for each component of the CTS pre-questionnaire from the CG is the following: a=0.93 for critical thinking, a=0.95 for algorithmic thinking, a=0.87 for problem-solving and a=0.94 for creativity. In the CTS post-questionnaire, a=0.77 for critical thinking, a=0.84 for algorithmic thinking, a=0.85 for problem-solving and a=0.81 for creativity. Therefore, Cronbach's alpha has a satisfying and high internal consistency for all the components of the CTS questionnaire ($a \geq 0.7$) for both groups, before and after the teaching intervention, according to the recommendations of Singh (2007).

Due to the non-normality and non-variance homogeneity of the data, non-parametric tests such as *Mann-Whitney U tests* were the most appropriate to be detected differences between the two groups. Also, *Wilcoxon signed rank tests* were used to detect differences between pre-and-post-questionnaires, split by gender from the participants' self-reported data analysis to be determined skills related to CT (Korkmaz et al., 2017). Supplementary, qualitative data were collected through semi-structured interview questions from participants' free comments and/or answers. For the best processing of the study analysis and reliability of qualitative data, the *Nvivo* (ver. 10) software was also used in an effort to be analyzed the content of participants' answers from the interview's questions.

Third, to measure students' learning performance, a coding framework analysis from Chao's (2016) study was utilized. It consists of 10 indicators related to computational practice (sequence, selection, simple iteration, nested iteration, and testing), computational design (problem decomposition, abutment composition, and nesting composition), and computational performance (goal attainment and program size). The entire debugging process seeks to investigate the consistency on how correct students' cognitive thinking as a solution plan expressed in natural language and if such a plan is applied properly into code.

7.3.6. Results

Descriptive sampling data analysis

After the data collection, the statistical analysis data of the profile questionnaire and the pre-questionnaire based on Lahtinen et al.'s (2005) questions in terms of difficulties in programming follows. The initial intention of this study is to provide some preliminary information about the perceptions of 50 participants regarding programming. Some of the most significant misinterpretations were largely concerned with the recognition in regard to the cognitive value of programming courses. Specifically, the understanding on how using programming constructs in real-world problems, either with simple or nested use of those constructs referring mostly to selection and iteration programming methods have been widely noticed by the majority of students (86%).

The acquisition of knowledge is usually either by reading theory or practically by solving exercises proposed by the formal textbook (65%), or outside of this in the context of learning how to program through proposed exercises in programming environments that CS instructors have chosen. During past programming courses, students mainly used "*Hour of Code"* and "*Scratch"* platform (95%), where many games are hosted in order to learn how to apply programming rules and constructs by programming through game playing small or semi-structured (10%), artistic expressions (80%) or storytelling creations (10%).

According to students' personal perspectives, difficulties and/or misconceptions are caused due to:

a) the lack of alignment on how to transform a solution from natural language to code (50%),
b) the inefficient attempts to unilaterally learn syntax or semantics of a programming language (40%), and
c) the use of interactive environments that often cannot simulate easily a design pattern that has value for implementation in solving a problem (10%).

Measuring computational concepts description and expression

Overall evaluation

To measure any improvement in overall rule/instruction specification ability, the mean scores of the worksheets from the two groups using error analysis rubric was indicated as appropriate to analyze students' answers in response to the *RQ1*. Also, the proposed rubric is comprised of specific grades was provided, but no more than 12. The error analysis rubric was compromised 6 in-game sessions for the 4 stages, 1 for the participants' personal training and other 3 to be counted for their final grades to each one from the CT instructional sessions described in *Table 7-3* includes the innate thinking of describing a solution in short sentences through text form in natural language (CT 1-4), to an algorithm (CT5) and finally apply into code every proposed solution plans (CT6). To this notion, 12 grades were the highest score that each participant

could gain that was calculated as follows: 0.5 grades gathered from each of the session CT1-4 described in *Table 7-3*, i.e. 3 (in-game stages) x 2 (0.5 grades x 4 for CT1-4 sessions) = 6 grades. In addition, other 6 grades could be gained. From the session CT5-6 described in *Table 7-3*, players could gather 3 grades, i.e. 1 grade by expressing pseudocodes (CT5) and another 1, when a solution plan was applied correctly into code (CT6) for every stage that they completed. This should be repeated 3 times since 3 were the stages that participants need to complete. Therefore, 3 grades from the 3 stages could be gained. If all participants from the EG or the CG achieved the maximum score by playing the proposed SG (RVC simulator), then their group could gather 300 grades in overall (i.e. 25 participants from each group x 12 grades that each one could gain). This indicates that completing the 3 stages, all boys and girls from the EG or the CG could have 156 and 144 grades, respectively. In other words, when boys either from the EG or CG completed all in-game stages, they could gain 78 grades for the sessions (CT1-4) and other 78 from the other two sessions (CT5-6). Also, girls could gain 72 grades for the sessions (CT1-4) and the other 72 for the sessions (CT5-6).

Figure 7-11 shows box plots of the grades between the scores of the two groups. The mean score on the EG was 9.7 (*SD*=1.56) and 8.5 (*SD*=1.45) on the CG. Such a difference had large effect between the two groups (*n*=50, *U*(1)=3.19, Z=-2.31, *p*=0.01, *r*=-.53).

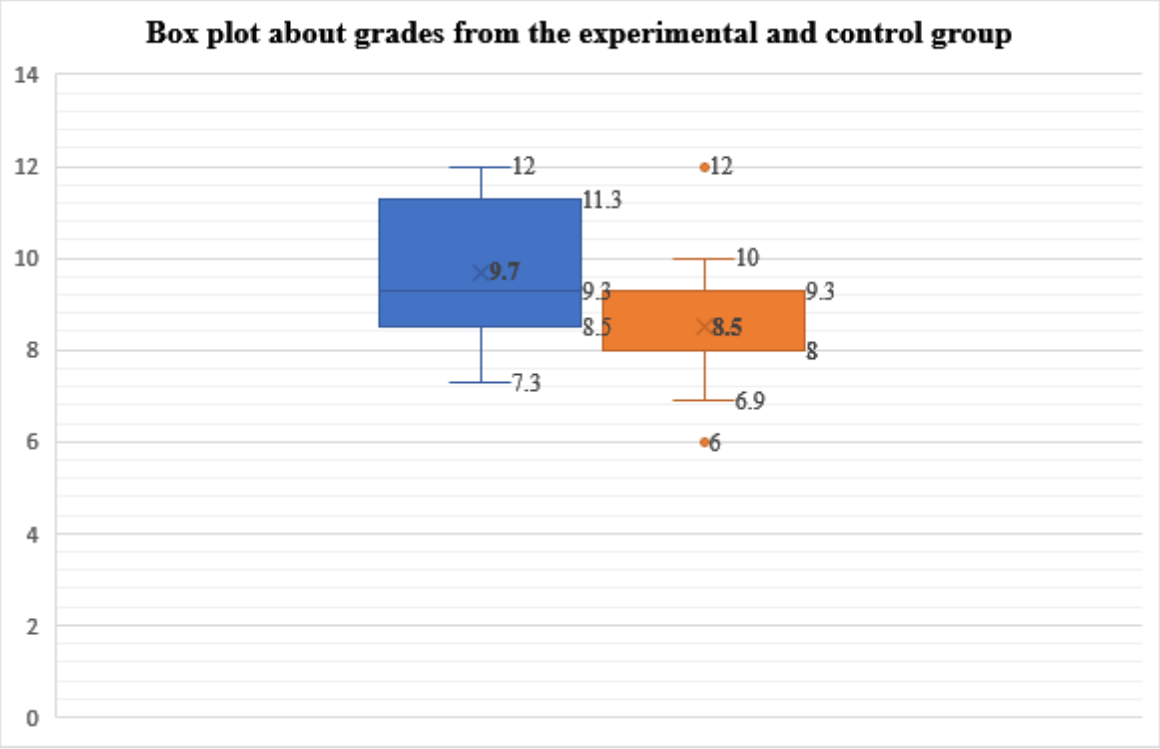


Figure 7-11: Box plot about grades from the experimental group and control group

The mean score of final grades on the EG was 9.7 (*SD*=1.56) and 8.5 (*SD*=1.45) on the CG. Based on the *Mann-Whitney U tests*, such a difference had large effect between the two groups (*n*=50, *U*(1)=3.19, *Z*=-2.31, *p*=0.01, *r*=-.53). In terms of overall measures of understanding and describing computational concepts, for boys, the mean score (final grade) was 9.91 (*SD*=1.38) while the mean of boys of the CG was 9.12 (*SD*=1.41). In specific, for girls, the mean score (final grade) of the CG was 7.82 (*SD*=0.99) while the mean score of the EG was 9.46 (*SD*=1.71). *Figure 7-12* displays the mean scores, split by gender for both groups.

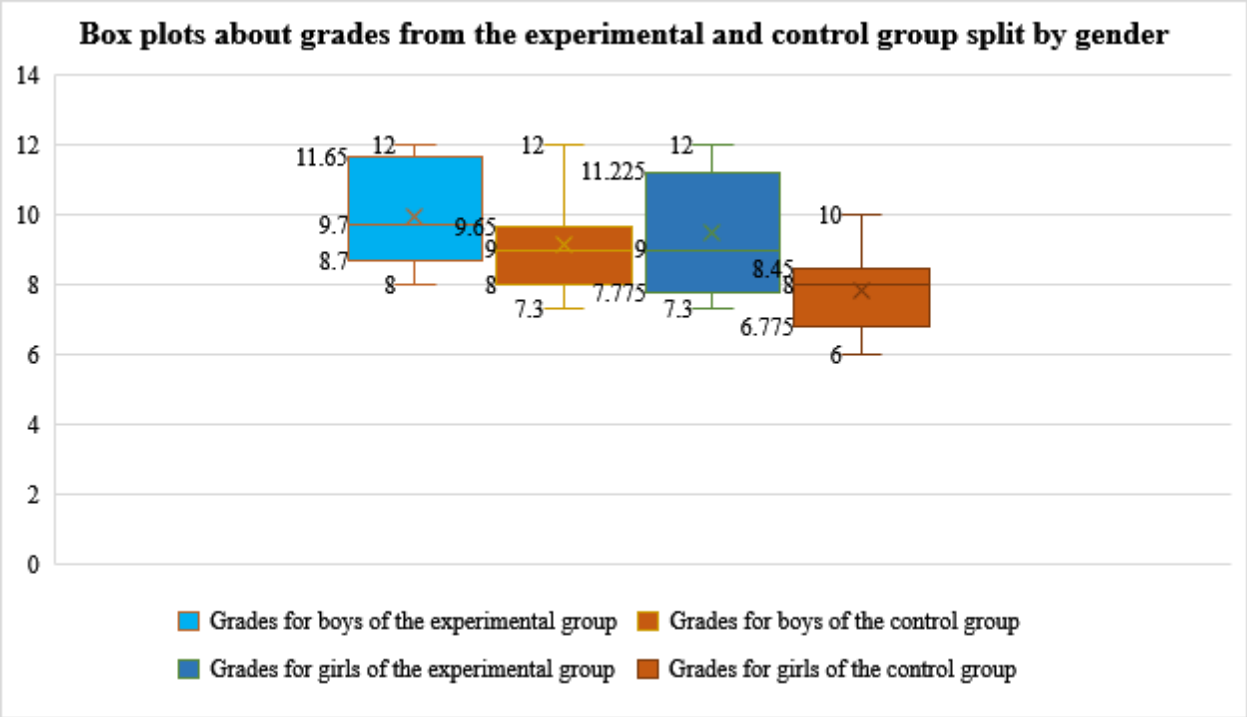


Figure 7-12: Box plots about grades from each group by gender

While in both groups two boys from each group achieved 12 grades, which was the highest-ranking score in this SG, two girls in the CG had minimum ranking score 6 grades.

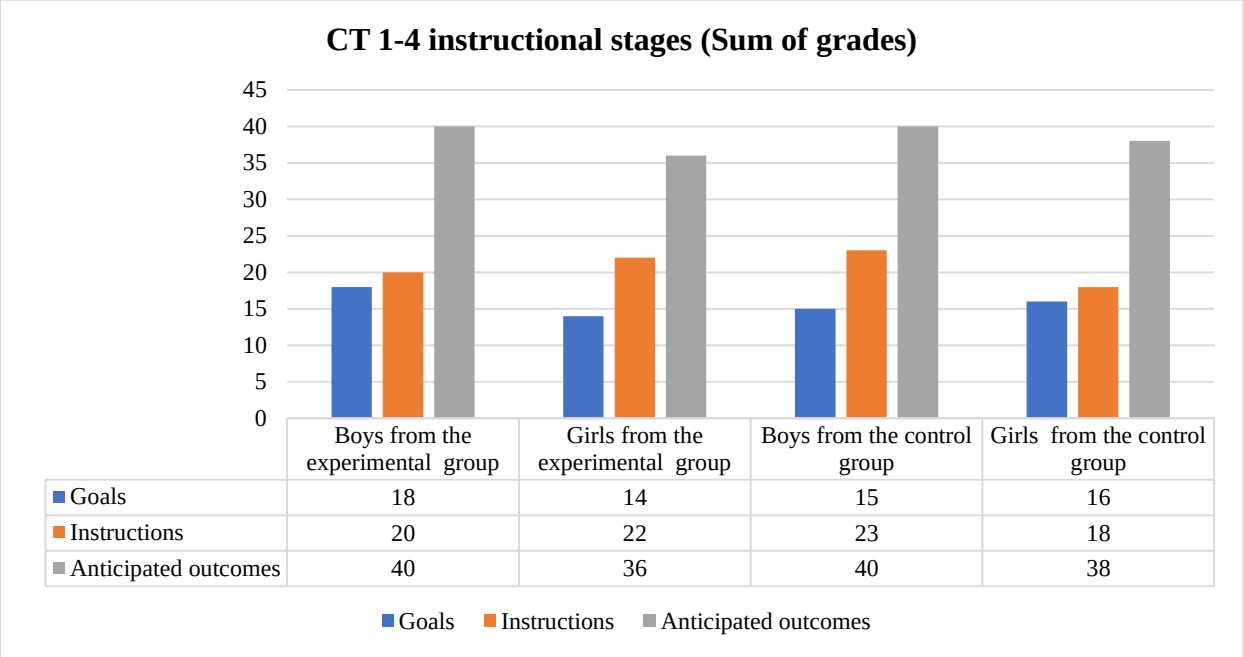
Measures of computational concepts

This work seeks to investigate any changes in boys' and girls' understanding regarding the different computational concepts, namely, enhanced understanding of goals, rules/instructions and anticipated outcomes. This implies an effort of presenting if the improvements were specific to certain computational concepts or occurred across all types. Three rule segments were categorized as goals, ten as rules/instructions, and anticipated outcomes respectively.

Using the *Mann-Whitney U* tests, the difference between the post-scores between the two groups was not significant for the goals (*n*=50, *U*(1)=3.34, *Z*=-2.11, *p*=0.14), significant for rules/instructions (*n*=50,

$U(1)=3.74$, $Z=-2.78$, $p=0.01$) and highly significant for anticipated outcomes ($n=50$, $U(1)=3.74$, $Z=-2.89$, $p=0.001$).

Below, *Figure 7-13* shows the sums of grades about correct for each concept for both groups, split by gender. To this notion, boys from EG and/or from CG can maximum gain in the session from CT1 to CT4 78 grades (13 boys x 6 grades=78 grades) and girls 72 (12 girls x 6 grades=72 grades).



	Boys from the experimental group	Girls from the experimental group	Boys from the control group	Girls from the control group
Goals	18	14	15	16
Instructions	20	22	23	18
Anticipated outcomes	40	36	40	38

Figure 7-13: Measures of understanding each computational concept

Based on grades gathered, all participants from both groups seemed to be really close. However, boys in both groups had better performance related to goals, instructions and anticipated outcomes in order to describe a solution. Also, in all stages, boys and girls of the EG from the sum of grades gathered are higher than the grades gathered from boys and girls from the CG.

Types of correct and incorrect computational concepts

Looking at the types of correct and incorrect computational concepts made in CT 1-4 instructional sessions from both groups, several are interesting findings. All in all, boys of the EG in rule segments have made fewer errors of omission (percentage difference was 19%) contrary to those who used Scratch. Girls of the former group have made fewer mistakes (percentage difference was 7%). Also, seeing errors of commission, boys and girls of the EG had fewer mistakes than to their CG counterparts. Figure 40 shows the sum of grades of incorrect answer segments by error type made from the two groups of participants.

Mann-Whitney U tests were carried out to examine whether the distribution of error codes changed significantly for the two groups. There were fewer missing rule segments mentioned in the EG as compared

to the CG, without any difference to be significant. Nevertheless, in total there was a highly significant difference in terms of vague fully erroneous instructions of the CG, with fewer vague rule segments of instructions/rules noticed by the EG. Another interesting point of view was a significant increase in the number of erroneous instructions and rules from girls of the CG. Concerning on the differences in error patterns between the two groups, design patterns largely reflected on the overall error patterns (shown in *Figure 7-14*), with the only significant difference to be the reduction in vague rule segments between the two groups (n=50, $U(1)$=3.66, Z=-3.25, p=0.03).

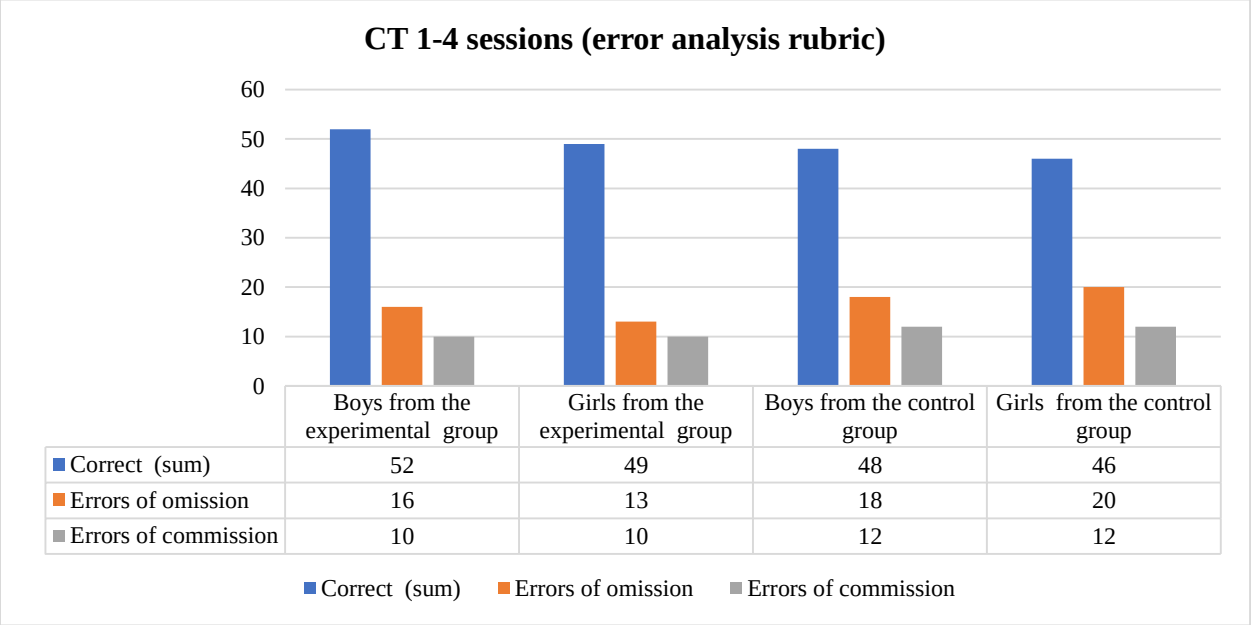


Figure 7-14: Types of correct and incorrect of computational concepts using an error analysis rubric

Based on the grades gathered, boys and girls of the EG made fewer errors in rules or instructions (errors of omission) than to their CG counterparts, while it is indicative that girls of latter groups made gave sometimes erroneous or vague information than girls (errors of commission) of the former group (see *Figure 7-14*).

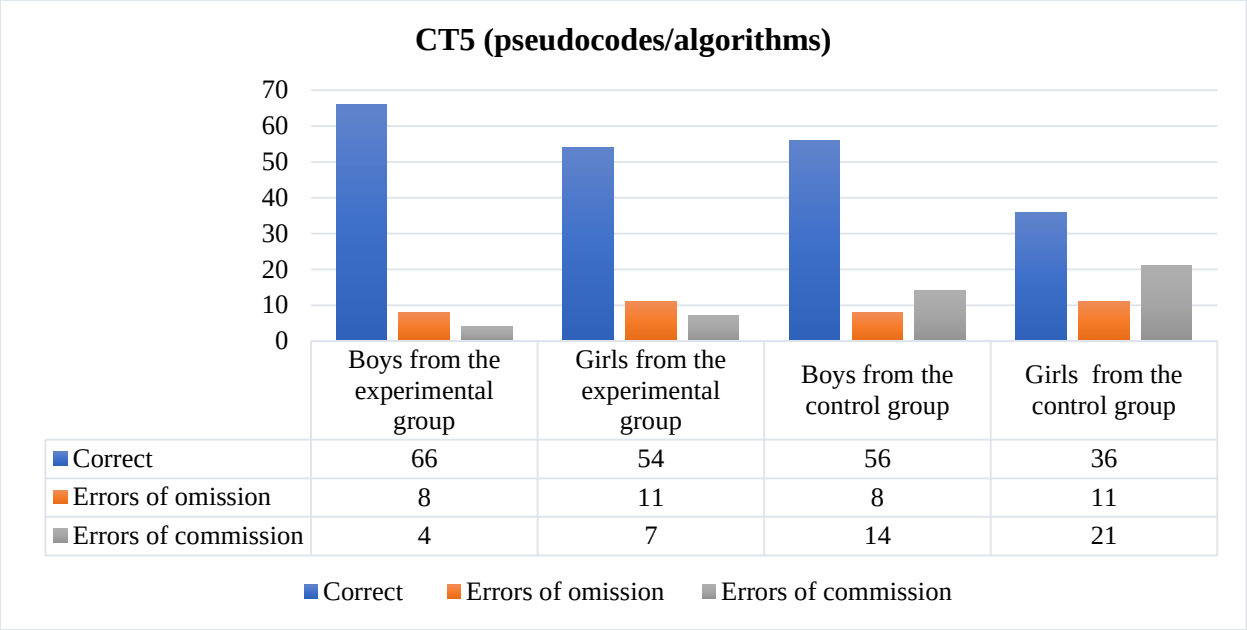


Figure 7-15: Types of errors in creating pseudocodes/algorithms

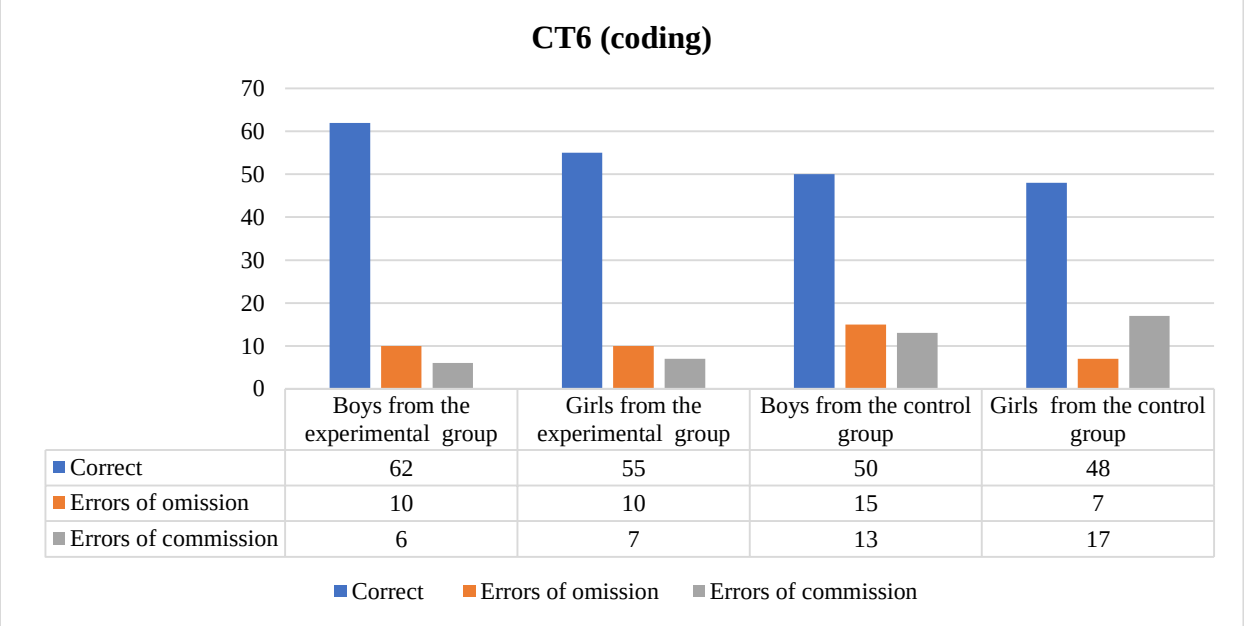


Figure 7-16: Types of errors in applying code

Figure 7-15 and *Figure 7-16* show the correct and incorrect answers in terms of expressing the algorithmic solution plans and applying them into code as design solutions. Since there was no single solution for using a specific programming construct, the choices had to be justified. In both groups, there was control and feedback from the supervising researcher, while recommendations were made for some changes and lapses. The scoring of pseudocode responses was made by taking into account the original

description proposed in a text form having short sentences in natural language and whether this as a thinking solution plan could be responded to an algorithm with concrete steps. If a lower value of the indicator "testing" identified, this suggested fewer tests on computer instructions composed by a participant. This may imply that participants tested their computer instructions based on chunks of the instructions rather than line-by-line or debugging by copying and pasting someone else's code. In other words, the participants who adopted the "trial approach" collected the least dusty dots and produced somewhat inefficient instructions, which implies relatively lower effective and efficient programs. "Testing" indicator disclosed that only one boy (4%) from the CG did not achieve to implement a script regarding of his 3 that he proposed in worksheets and the same case was also observed in the CG with a boy (4%) and a girl (4%).

For testing the consequence of generated computer instructions, the indicator of "testing" showed the average frequency that participants tested the consequence of executing a computer program immediately after generating or revising it. It shows the ratio in a number of rules and instructions that can be executed as computer programs in order to test the consequence of problem-solving depending on (visualized) control flow and command blocks. If a lower value of the indicator "testing" identified, this suggested fewer tests on computer instructions composed by a participant. This may imply that some participants tested their computer instructions based on chunks of the instructions rather than line-by-line or debugging by copying and pasting from someone else's code. In other words, the participants who adopted the Trial approach collected the least dusty dots and produced somewhat inefficient instructions, which implies relatively lower effectiveness and efficiency programs. "Testing" indicator disclosed that only one boy (4%) from the EG did not achieve to implement a script regarding of his 3 that he proposed in worksheets and the same case was also observed in the control group with a boy (4%) and a girl (4%).

Descriptive statistics of computational problem-solving indicators into code

In regard to *RQ2*, *Table 7-7* and *Table 7-8* reveal the descriptive statistics of 10 indicators concerning the implementation of computational problem-solving strategies from the EG. Regarding the dimension of computational practice, the results showed that the participants used more selection (*M*=2.51, *SD*=0.52) and nested iteration (*M*=1.89, *SD*=1.04) than sequence (*M*=1.67, *SD*=1.04) or simple iteration (*M*=1.51 *SD*=0.66) control flow blocks in solving the subparts of the main computational problem that consisted of 3 in-game stages. The results also showed that the participants, on average, tested all programmed instruction (see "testing" indicator) 2.84 times (*SD*=0.36). This may indicate that most participants tended to test their code by a chunk of instructions rather than by a single instruction.

Referring to computational design, in *Table 7-7*, the indicator of "problem decomposition" showed that the participants produced 2.76 (*SD*=0.42) subparts of solutions. This may indicate that the participants would generally divide one computational problem into two or more subparts of problems and formulate

corresponding solutions. The results also showed that the participants demonstrated more "abutment composition" (*M*=2.16, *SD*=0.73) than "nesting composition" (*M*=0.72, *SD*=0.77). The results suggest that the participants, in the RVC simulator created in OpenSim, were more likely to generate solutions to the subparts of the main problem by adjoining control flow code blocks rather than nesting the control flow blocks. With regards to their computational performance, the indicators of "goal attainment" and "program size" showed that the participants, on average, collected 18.56 (*SD*=3.33) dusty dots giving grades and used 12.32 (*SD*=2.11) command blocks to solve a computational problem.

Table 7-7: Statistical results of computational problem-solving strategies from the experimental group

Indicators	**Range (*n*=the number of times that each indicator was used)**	***M***	***SD***
Computational practice			
Sequence	0-3	1.67	1.04
Selection	0-3	2.51	0.52
Simple iteration	0-2	1.51	0.66
Nested iteration	0-2	1.89	1.04
Testing	0-3	2.84	0.36
Computational design			
Problem decomposition	0-3	2.76	0.42
Abutment composition	0-3	2.16	0.73
Nesting composition	0-2	0.72	0.77
Computational problem-solving performance			
Goal attainment	From 12 to 22	18.56	3.33
Program size	From 10 to 15	12.32	2.11

Table 7-8 reveals the descriptive statistics of 10 indicators regarding the implementation of computational problem-solving strategies from the CG. Regarding the dimension of computational practice, the results showed that the participants used more sequence (*M*=2.38, *SD*=0.99) and simple iteration (*M*=1.81, *SD*=0.71) than selection (*M*=1.21, *SD*=0.45) or nesting iteration (*M*=1.57, *SD*=0.49) control flow blocks in solving the subparts of the main computational problem that consisted of 3 in-game stages. The results also showed that the participants, on average, tested all programmed instruction (see "testing" indicator) 2.64 times (*SD*=0.48). In the opposite view, participants who utilized Scratch tended to test their code by a single instruction rather than by a chunk of instructions, for example, using nesting iteration.

Referring to computational design, in *Table 7-8*, the indicator of "problem decomposition" showed that the participants produced 2.76 (*SD*=0.42) subparts of solutions. This may indicate that the participants would generally divide one computational problem into two or more subparts and formulate corresponding solutions. The results also showed that the participants demonstrated more "abutment composition" (*M*=2.28, *SD*=0.77) than "nesting composition" (*M*=0.48, *SD*=0.75). The results suggest that the participants, who utilized the RVC simulator created in Scratch, were more likely to generate solutions to

the subparts problems by adjoining control flow code blocks rather than nesting the control flow blocks. Regarding their computational performance, the indicators of "goal attainment" and "program size" showed that the participants, on average, collected 18.04 (*SD*=2.66) dusty dots and used 13.76 (*SD*=2.37) command blocks to solve a computational problem.

Table 7-8: Statistical results of computational problem-solving strategies from the control group

Indicators	**Range (*n*=the number of times that each indicator was used)**	***M***	***SD***
Computational practice			
Sequence	0-3	2.38	0.99
Selection	0-2	1.21	0.45
Simple iteration	0-1	1.81	0.71
Nested iteration	0-2	1.57	0.49
Testing	0-3	2.64	0.48
Computational design			
Problem decomposition	0-3	2.76	0.42
Abutment composition	0-3	2.28	0.77
Nesting composition	0-2	0.48	0.75
Computational problem-solving performance			
Goal attainment	From 12 to 22	18.04	2.66
Program size	From 10 to 18	13.76	2.37

Summing up quantitative data to respond in *RQ2* based on code tracing analysis from the palettes of Scratch and S4SL, the results indicated that students of the EG have encoded more complex solutions by combining sequence aligned with selection and repetition programming constructs contrary to those of the CG who seemed to use in their design patterns either repetitive or sequential constructs. Taking into account the results from "program size" and "goal attainment", students of the former group were able not only to collect a great number of dusty dots to accomplish their goals but also to create more efficient and effective programs with a smaller number of code blocks. Therefore, almost all participants from the EG received higher grades than those from the CG.

Analysis of students created scripts in different instructional contexts related to computational thinking

Findings overall

All the proposed solutions written in short sentences through natural language (scripts) from students of both groups while playing a SG. Totally, of the 50 students who took part in this experiment, all of them created an algorithm and coding several proposed solutions regardless of the interactive environments that were used. Specifically, 22 out of 25 (88%) managed successfully the creation of a working script by

playing the game in OpenSim with S4SL, completed with goals alongside with one or more anticipated outcomes for at least 2 stages, in contrast to the CG where 20 out of 25 (80%). Overall, the sum of grades gathered by the EG were 254 for completing and correcting scripts that were written and saved from the CG were 237 grades enclosing revisions to already created scripts before the final proposal in coding ("testing" indicator). Considering distinct individual scripts alone, a total of 3 scripts for all the three stages need to be created for the CT 1-5, including the pseudocode and the CT6 was for the implementation of the final solution via Scratch or S4SL palette.

The examination of computational constructs presented in each script would give to this study's findings some important information. Basically, it was necessary for each script to contain, after the main goal, a single action that should include instructions and/or rules that the RVC needs to follow a cleaning pathway. One of the most distinctive characteristics that need to be referred was that more complex scripts contained more than one action, appropriately sequenced. For example, it was observed that further complexity is evidenced by the inclusion of conditionals either of simple "repeat" conditionals or of more complex *"repeat…until"* or/and some participants from the EG considered *Boolean* operators within the conditionals.

On the other side, participants from the CG had a different perception in solving such a problem, as they proposed simple or nested iteration methods or/and some of them included Boolean operators or variables. Besides, according to the creation of computational practices, 3 boys (23%) and 1 girl (8%) from the CG proposed solutions one boy (7%) and two girls (16%) from the CG proposed solutions which are created by scripts using a simple script and only, for example, sequence programming method. In addition, 31 students (74%) created one or more complex scripts, i.e. their scripts contained additional constructs beyond the basic requirements for some well-formed scripts. Lastly, 30 students (71%) created a sequence of two or more actions.

Findings from the use of programming constructs in computational practices split by gender

Of the 150 scripts which were created to describe the situation and instructions that a RVC should follow from all participants, 75 from each group were finally collected. A variety of different events were used by a student in furtherance of setting goals and anticipated outcomes in their scripts. In total, 22 participants from the CG (88%) have tried to use and combine more than one programming construct for the implementation of their programming method and 23 participants (92%) from the EG. In specific, 11 boys and 11 girls who utilized Scratch and 12 boys and 11 girls who utilized OpenSim with S4SL tried to combine another one programming construct with the chosen one that they would like to solve in the first

stage. However, there was not found an association between the EG and the CG ($U(1)=3.11$, $Z=-2.29$, $p=0.18$).

Figure 7-17 and *Figure 7-18* show specific events that are applied to the code, split by gender. Scripts specified participants' goals by conversation lines are easiest to implement, and the ones that students' innate thinking to solve a problem and after that to code via Scratch or S4SL palette. For instance, *Figure 7-17* depicts the cumulative percentage of boys and girls from the CG used to express and apply into the code a solution plan. Firstly, boys in their majority (41%) have used a sequence as the main programming construct, as the second choice was the combination of selection and simple iteration (29%) and as third the selection (18%). A variety of girls (40%) have used a combination of simple and nested interaction, secondly simple iteration (30%) and as the third choice was the sequence.

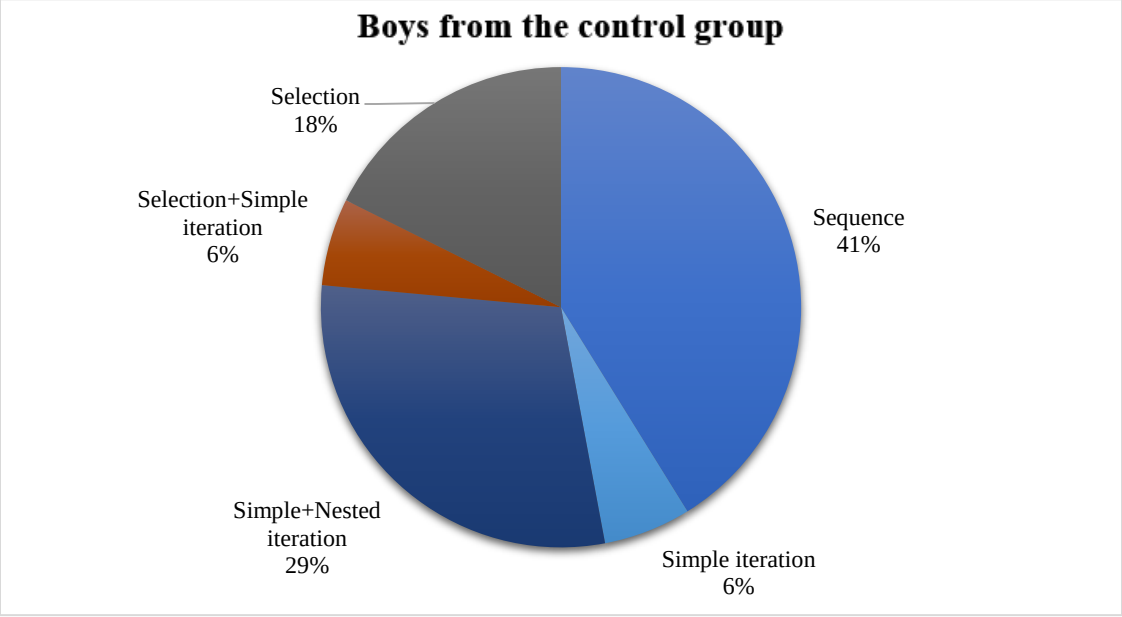


Figure 7-17: Computational concepts which are used from boys in the control group

A range of different types of actions was included in students' scripts with different methods to be also observed. These actions using programming constructs follow on from the goals until the anticipated outcomes shown in *Figure 7-19* below. It shows the cumulative percentage that participants from the EG have expressed and applied into code their scripts, split by gender. The use of programming constructs was not broadly similar between boys and girls.

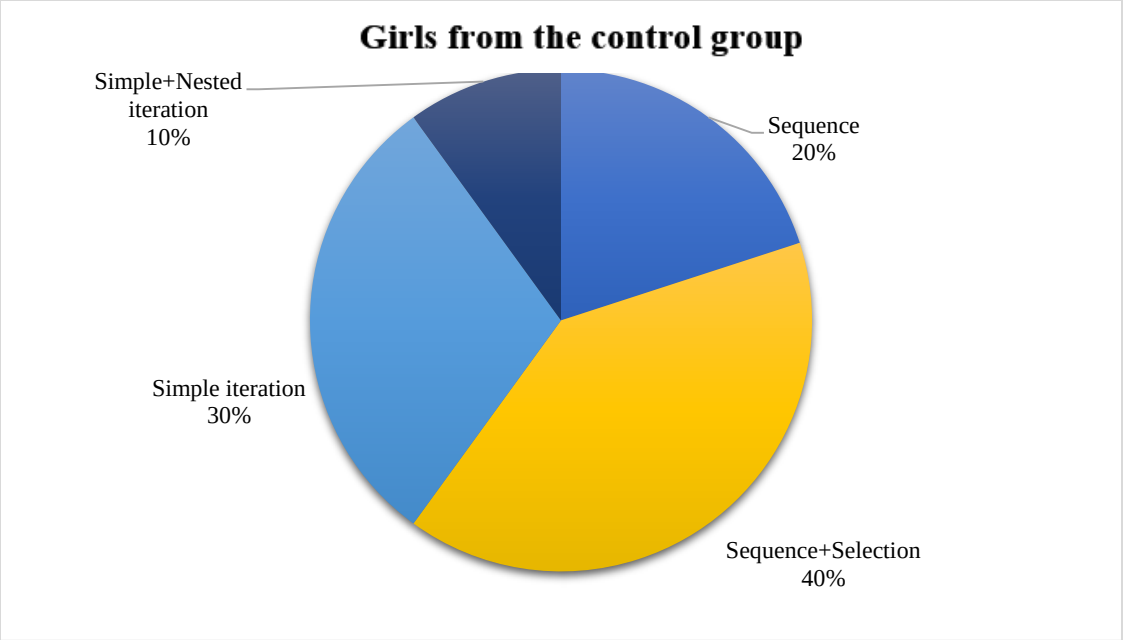


Figure 7-18: Computational concepts which are used from girls in the control group

Figure 7-20 depicts the cumulative percentage of boys and girls from the EG used to express and code a solution. Firstly, boys, in their majority, have used selection as the main programming construct (40%), as the second choice was the combination of simple and nested iteration (30%) and as third sequence and nested iteration (20%). Almost nearly, many girls have chosen to use a combination of sequence and nested iteration (34%) a combination of simple and nested interaction (33%), secondly sequence (22%) and as the third choice was the combination of sequence and selection (11%).

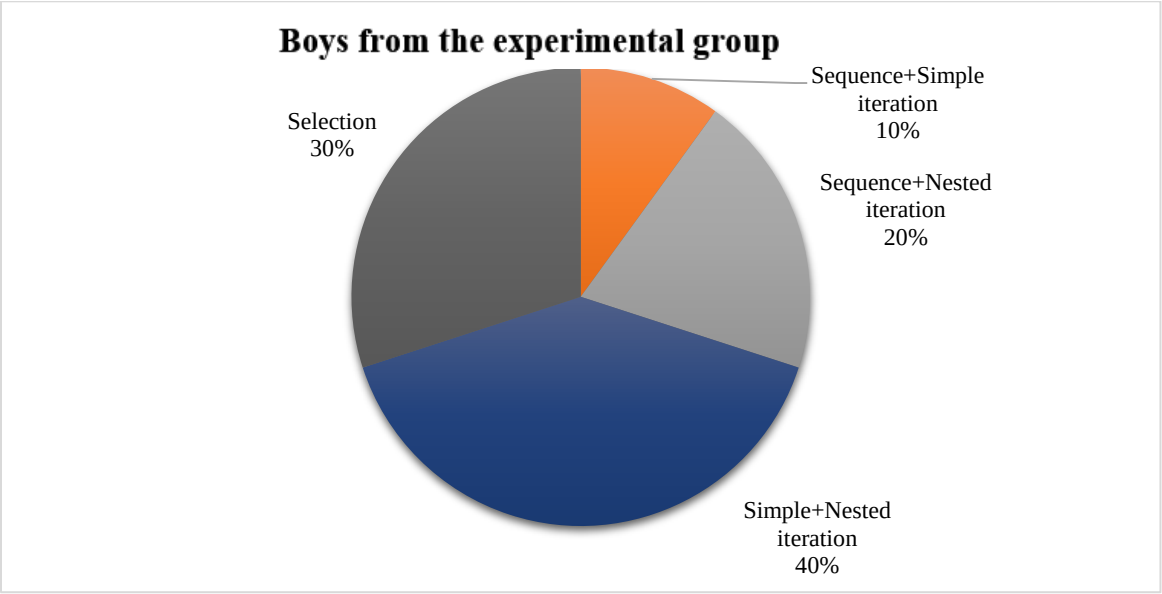


Figure 7-19: Computational concepts which are used from boys in the experimental group

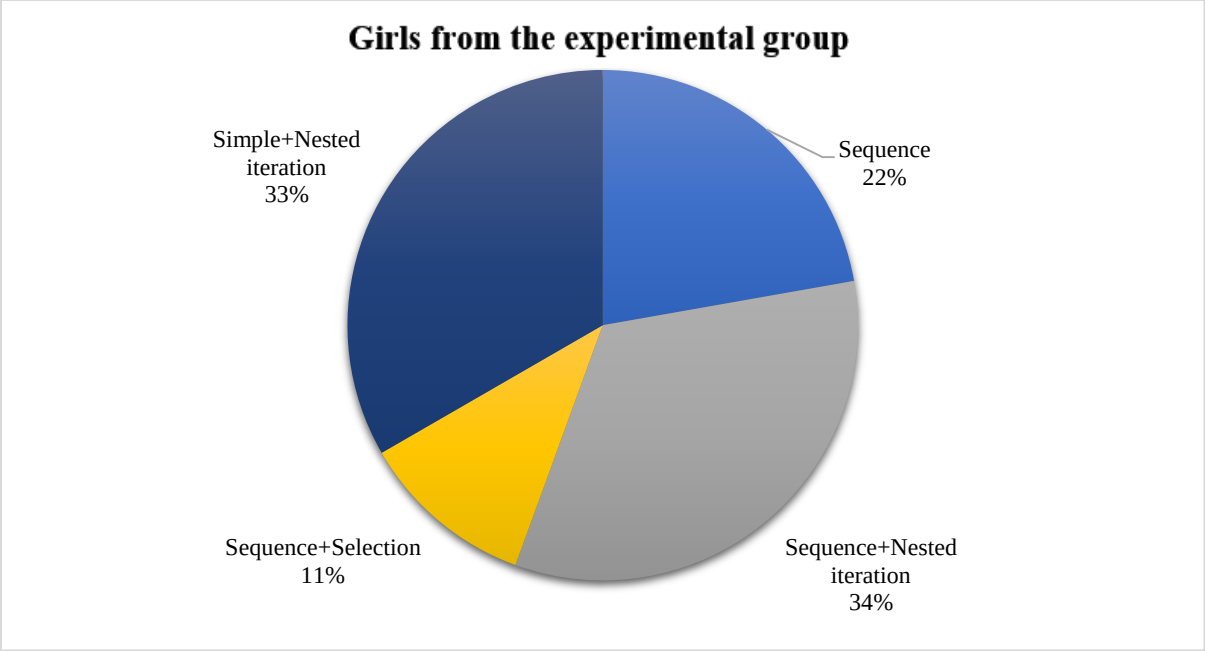


Figure 7-20: Computational concepts which are used from girls in the experimental group

Analysis of skills related to computational thinking

Results from the pre-and-post questionnaire to measure skills related to computational thinking split by groups

Self-reported students' answers in regard to the potential assistance of the RVC simulator for CT instruction needs to be investigated since it was difficult to extract answers only from the coding analysis process. Pre-and-post CTS questionnaires in the direction of determining the levels of skills related to CT (Korkmaz et al., 2017) were regarded as essential for that purpose. Therefore, to determine the mean score of skills related to the CT scale, it was imperative to find the average of all the scores. In total, the mean score of participants' answers from the CG in *Figure 7-21* is presented. Students have reported the highest scores not only in the demonstration of a solution (*Mpre*=3.06, *SD*=1.14; *Mpost*=3.64, *SD*=1.33) but also to the establishment of the equity that tends to give a step-by-step solution to a problem (*Mpre*=2.94, *SD*=1.14; *Mpost*=3.94, *SD*=1.33). Such an effort was accomplished either by developing genuine ideas different from the ordinary ones (*Mpre*=2.93, SD=1.04; *Mpost*=3.37, *SD*=1.12) or by using critical thinking and logical thinking focused on deciding what shall be done and believed that need to be done (*Mpre*=3.02, *SD*=1.34; *Mpost*=3.41, *SD*=1.37).

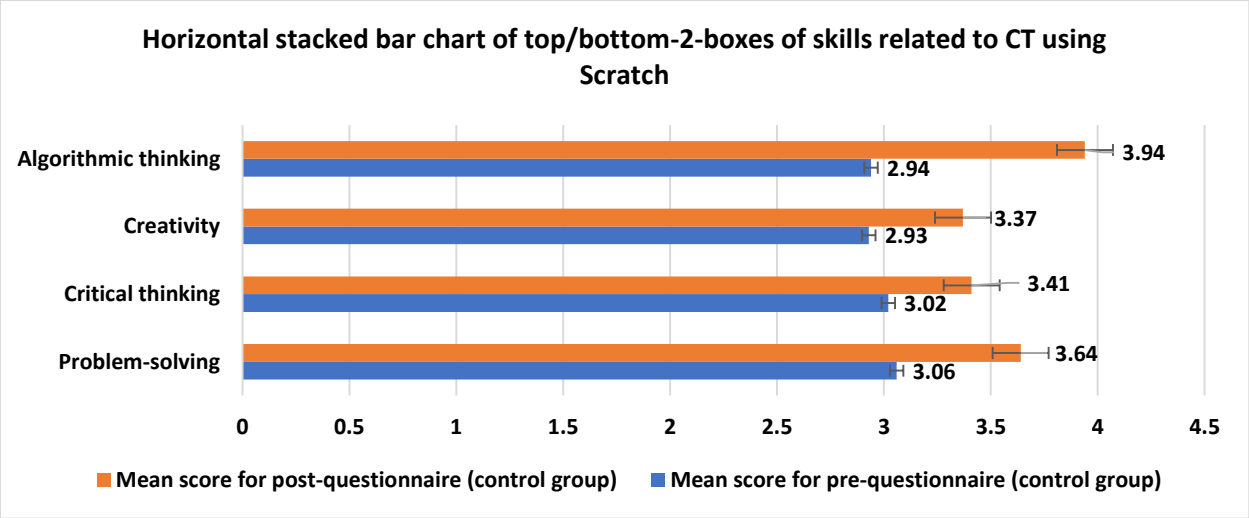


Figure 7-21: Determining the computational thinking skills of participants from the control group

Figure 7-22 presents totally the mean scores of participants' answers from the EG. Students reported the highest scores for shaping and assessing their own ideas, being able to make efficient use of code blocks critically (*Mpre*=3.04, *SD*=1.47; *Mpost*=4.24, *SD*=1.56). They also seemed to understand better the instructions that could be proposed as solutions to subparts of the simulated computational problem before the description of an algorithm (*Mpre*=2.96, *SD*=1.22; *Mpost*=4.22, *SD*=1.61). Participants from the same group, seemed to find essentially alternative solutions by generating different methods for presenting their thinking solution plan which can be different from the ordinary ones (*Mpre*=3.01, *SD*=1.27; *Mpost*=4.18, *SD*=1.87) or express by generating algorithmically a proposed step-by-step solution for solving subparts of the main problem (*Mpre*=3.01, *SD*=1.57; *Mpost*=4.18, *SD*=1.61).

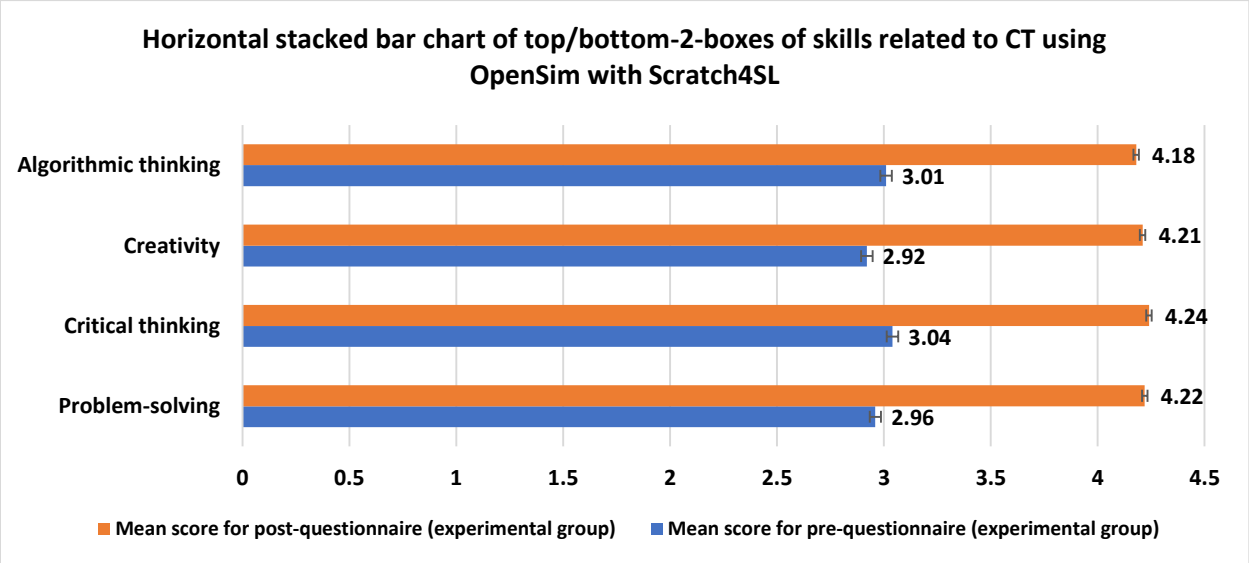


Figure 7-22: Determining the computational thinking skills of participants from the experimental group

Additionally, to investigate at absolute differences in CTS pre-questionnaire and post-questionnaire mean scores, the normalized learning change was also calculated by taking into account the maximum possible gain or loss given the pre-questionnaire scores (Marx & Cummings, 2007). The normalized learning change is defined as a variant on normalized learning gain (*<g>*) which is appropriate for situations in which there are instances of negative learning gain for a small number of the students. In specific, the normalized learning gain was calculated as *<g>=100x(post-pre)/(100-pre)*, and a modified calculation was used for students with negative learning gain: 100x(post-pre/pre) (Howland & Good, 2015; Knight, 2010). The purpose was to be measured the effectiveness of each intervention regarding the conceptual understanding for determining skills related to CT from each group separately. Overall, there was a positive normalized gain of the participants from the EG who utilized OpenSim with S4SL (41%) to determine their skills related to CT than their CG counterparts who used Scratch (20%).

To conclude, it is notable that while in both groups after using the two interactive environments, higher mean scores were achieved from participants who played the SG via OpenSim than those who played it via Scratch.

The mean score of CT questionnaires among all scales is higher for the EG. In addition to the above, some indicative responses to the semi-structured interview from the participants from the EG and CG are the following:

• *"In-game elements are really well-presented in OpenSim. This helped me not only to comprehend my decisions by applying and explaining my solutions using S4SL but also to know why I used some programming constructs. In contrast to other small parts of playable games such as Minecraft or Star Wars through Hour of Code website, the RVC simulator assisted me to understand reasons of using some programming methods for specific subparts inside the 3 rooms"* (a girl from the EG).

• *"I knew Scratch. I think that dragging and dropping programming constructs helped me really to apply my proposed solutions, as I visually saw the results of the code, save and present for each stage"* (a boy from the CG).

• *"The S4SL palette enabled me to write correctly the code, while I was previously tried to describe and propose a solution about what I observed in OpenSim"* (a boy from the EG).

• *"Because of knowing previously Scratch, I did not want the CS instructor's guidance to express and apply my solution plans"* (a girl from the CG).

Reflecting on researchers' observations, some of the most important parts of such a process need to be extracted. First, students have found interesting the entire process really because most of them tried to low the time of cleaning routes for each room and present alternative solutions as it was asked at the beginning to do. Second, students of the EG who explored using an avatar the entire house assimilated the intervention more easily and they explored it before the main activities starting often without any

intervention from the CS instructors. Third, students from both groups had similar difficulties in describing their algorithmic plans with clarity and accuracy in natural language to gather information verbalized in think-aloud, since they preferred to describe a process in general rather than being more concise in a step-by-step solution. This also had an effect on programming. It was at end crucial to mention that Scratch and the S4SL palette was an important factor affecting the proposed design patterns for avoiding any syntax errors and be focused on the problem statement simulated in the interactive environments.

Results from the pre-and-post questionnaire to measure skills related to computational thinking split by gender

Table 7-9 presents participants' answers on how they tried to determine skills related to CT before and after this teaching intervention, split by gender. Answers of participants from the EG were higher in the CTS post-questionnaire than their counterparts from the CG. In specific, boys presented higher mean scores in terms of problem-solving thinking, algorithmic thinking and critical thinking with 4.34, 4.32 and 4.47 respectively, in contrast to boys from the CG.

Table 7-9: Descriptive analysis and Wilcoxon signed-rank tests of computational thinking skills split by gender

CT skills	**Gender**	**Scratch**				**Wilcoxon signed rank test**	**OpenSim with S4SL**				**Wilcoxon signed rank test**
		M (pre)	*SD (pre)*	*M (post)*	*SD (post)*		*M (pre)*	*SD (pre)*	*M (post)*	*SD (post)*	
Problem-solving	Boys	2.81	0.79	3.81	1.17	*Z*=-4.09, p=0.11, *r*=-0.55	3.21	1.19	4.34	1.27	*Z*=-4.12, p=0.03, *r*=-62
	Girls	3.14	1.11	4.14	1.22	*Z*=-3.79, p=0.13, r=-0.67	2.87	1.37	4.12	1.07	*Z*=-4.02, p=0.02, *r*=-61
Creativity	Boys	2.83	1.02	4.03	1.14	*Z*=-3.67, p=0.14, *r*=-59	2.88	1.36	4.03	1.16	*Z*=-4.36, p=0.11, *r*=-46
	Girls	3.15	0.75	3.89	1.17	*Z*=-3.72, p=0.07, r=-45	3.43	1.61	4.12	1.14	*Z*=-4.49, p=0.12, *r*=-46
Critical thinking	Boys	2.88	0.73	3.88	1.04	Z=-3.96, p=0.11, *r*=-32	3.27	0.97	4.47	0.84	*Z*=-4.66, p=0.08, *r*=-45
	Girls	2.89	0.74	3.89	0.95	*Z*=-3.12, p=0.06, *r*=-38	2.78	0.74	4.15	0.85	*Z*=-3.89, p=0.15, *r*=-49
Algorithmic thinking	Boys	2.68	0.68	3.88	0.97	*Z*=-3.42, p=0.001, *r*=-66	3.52	0.87	4.32	0.97	*Z*=-4.03, p=0.01, *r*=-66
	Girls	2.97	0.99	3.78	1.08	*Z*=-3.65, p=0.001, *r*=-55	2.98	1.07	4.08	1.28	*Z*=-4.04, p=0.02, *r*=-58

In regard to problem-solving, the Wilcoxon Signed-rank test showed that there was a significant increase for boys from the EG resulted by the pre- (median=3.18) and post-questionnaire (median=4.28) to be determined skills related to CT, Z=-4.12, p=0.03, and the increase was large (r=-.62). This increase in problem-solving was also observed on girls from the EG by measuring the pre- (median=3.05) and the post-questionnaire (median=4.06) to be determined skills related to CT, Z=-4.02, p=0.02, and the increase was large (r=-.61).

With respect to algorithmic thinking, the Wilcoxon Signed-rank test showed that there was a significant increase of the boys from the EG resulted by the pre- (median=3.22) and post- (median=4.22)

questionnaire to be determined skills related to CT (Z=-4.03, p=0.01) and the increase was large (r=-.66). This increase in problem-solving was also observed on girls from the EG by measuring the pre- (median=3.55) and the post-questionnaire (median=4.02) to be determined skills related to CT (Z=-4.04, p=0.02) and the increase was large (r=-.58). In terms of critical thinking, however, there was not any significant difference to be mentioned either about boys or girls of the two groups.

As the data was skewed (not normally distributed), the most appropriate statistical test was the Mann-Whitney U in order to compare differences between the two groups. The statistical analysis showed that the EG of OpenSim (median=4.12; mean rank=4.06) scored higher on both problem-solving and algorithmic thinking. First, participants from the EG (median=4.87; mean rank=3.88) scored higher than participants from the CG (median=4.02; mean rank=3.67) in regard to problem-solving. Mann-Whitney U value was found to be statistically significant $U(1)$=4.33, Z=-5.43), p=0.02, and the difference between the EG and CG was large (r=-.62). Second, participants from the EG scored also greater (median=4.65; mean rank=3.56) than participants from the CG (median=6.88; mean rank=3.22) in terms of algorithmic thinking. Mann-Whitney U-value was found to be statistically significant $U(1)$=4.11 (Z=-5.61), p<0.01, and the difference between the EG and CG was large (r=-.56).

What follows is a discussion from transcribing and observing the think-aloud responses and retrospections from questions by asking the participants from both groups. One permanent example was for the rooms of the second stage and the main question was as follows: "*Why is this cleaning pathway that you decided to propose for the cinema room as the most appropriate in order to not lose battery energy the RVC?"* An indicative example was answered by a boy from the CG. It was interesting to mention that those who used mostly the selection programming constructs, were those who utilized Scratch as the user would see an only horizontal piece of floors in two-dimensions. His decision was justified saying that *"seeing those sits and sofas and one static avatar, I believed that by determining the RVC's actions calculating the distance between this avatar and the furniture, I would have the best solution using such a construct".*

In contrast, a girl from the EG stated the following: "*The iteration method use would better, since either walking in the first person or in three dimensions as an avatar, I understood the movements inside the room that need to be done like being under the small table and make spiral movements around it the robot performing to this notion better in order to reach the final goal"*. This implies that participants who used OpenSim seemed could understand better the spatial geometry layout for suggesting a particular movement that the RVC should follow to clean each room of the house as everything was formed in 3D realistic simulated problem-solving contexts.

Based on the above results and with a purpose to give an answer in the *RQ2*, participants from the EG seemed that had greater satisfaction by using OpenSim with S4SL as they seemed to determine in a

higher level in critical thinking and problem-solving skills. Also, the participants from the CG have achieved lower mean scores than those from the EG to all skills related to CT.

7.3.7. Discussion

The present quasi-experimental study seeks to investigate if a SG created in OpenSim with S4SL or in Scratch can affect the learning performance of boys and girls in order to gain a greater understanding on the use of skills related to CT for developing, applying and transforming their solution plans into code by comparing their computational problem-solving strategies. All grades were measured according to the proposed solutions into code from the participants in both groups in order to investigate the correctness of programming behaviors integrated into visual elements. In specific, the RVC simulator created in OpenSim seemed to have the potential to provide the appropriate teaching and learning contexts for instructive guided support through formal and informal instruction settings. While OpenSim allows the free experimentation and reflection of students inside a 3D problem-solving environment, its combination with S4SL enables users to express and apply their solution plans into as design patterns.

This study's findings indicate that a great number of participants from the EG appeared to not have any difficulties in producing some good computational problem-solving practices without being complicated, but with the combination of simple design patterns to be presented as final solutions. The participants from the CG attempted to provide relatively not so advanced computational design strategies and they appeared to have the most difficulties in producing a good computational performance. This may suggest that in the proposed SG created in Scratch with the adoption of selection control flow using nesting composition programming methods may be insufficient about a good performance for solving computational problems due to a large number of code blocks that were utilized. The participants from the CG tended to adopt the nesting method and show relatively frequent testing of solutions. The simple design strategies appeared to meet the quantitative requirement of computational problems; however, many solutions proposed by participants from the CG were relatively ineffective. In this regard, some participants seemed that had difficulties in applying their nesting programming methods to solve the subparts of the simulated problem. Therefore, it was reasonable to investigate errors or revisions made by players of both groups during the process of programming that could give answers about the difficulties or find other possibilities that might lead to frequent testing of solutions and good (or not) computational performance. To this notion, the participants' computational problem-solving strategies from the decomposition of subparts of the main problem to the combination of control flow code blocks seemed to affect their knowledge about why and how they can use those instructions and rules with fundamental programming constructs correctly so as to propose their solution plans (de Raat, 2007; Robins et al., 2003). Such a finding comes in line also with a substantial body of contemporary research (Brennan & Resnick, 2012; Lye &

Koh, 2014; Whiterspoon et al., 2017) have argued that students' computational practices and design are regarded as essential on measuring their performance in learning computer programming.

Consistent with Howland's and Good's (2015) study findings, a block-based palette is regarded as a reliable tool for high school students to avoid syntax errors in programming and it can facilitate them to trigger in problem-solving via 3D games by expressing and applying more succinct and precise rules with instructions combined with programming constructs. On the other side, contrary to the results of past efforts (Brennan & Resnick, 2012; Denner et al., 2012; Mouza et al., 2016), participants from this study using a 3D SG had reasonable efforts to answer why they utilized specific programming constructs and instructions in their computational practices, dodging the vague syntax of programming constructs. Such a process can give valuable answers for assessing how students try to think and practice computationally before they start coding in practice.

An instructional fading scaffolding process can be a crucial parameter for high school students on how learning can be applied into code with effective and efficient design patterns by understanding the use of skills related to CT and concepts as proposed solutions for simulated real-world problems. Moreover, through coding, students could critically review their solutions and adopt an analytical reasoning strategy during the problem-solving process as it was indicated according to their answers in the think-aloud protocol. Such a finding is consistent with the claims of previous studies (e.g., Liu et al., 2011; Liu et al., 2017) which have shown that students' critical thinking or analytical thinking is in relation with problem-solving skills fostered via a SG's gameplay.

Inevitably, the alternative computational design patterns in problem-solving contexts which were reflected on the creation of different computational practices and applied successfully as alternative solutions, have also influenced students' learning performance based on the indicators "program size" and "goal attainment". For example, the participants from the EG who applied their computational practices using selection programming, they also tended to use the more advanced design strategy of nesting different control flow code blocks. This may lead to the production of relatively effective and efficient computer programs. In essence, students are motivated due to the novelty of the 3D VW as a technology to be engaged in meaningful interaction with the visual elements and objects in order to develop more effective computational problem-solving strategies and then have a better performance. Therefore, the in-game use of evocative 3D visual objects of basic geometric shapes (e.g., triangle, square, and hexagon) can be considered as a powerful abstract conceptualization approach that can assist the development of skills related to CT. This can also give evidence of a deeper understanding of the description of a cognitive thinking process for the comprehension and production of the proposed solutions that applied in code. With this in mind, such a process can become appropriate and effective as well as learning gain on when and

how students tried to decompose problems so as to propose solutions by applying control flow code blocks, such as selection or nested iteration.

Another interesting point of view is that this study's results are encouraging from a gender perspective in terms of computational understanding. While previous studies (Brennan & Resnick, 2012; Denner et al., 2012) have focused on finding ways to motivate the views and perceptions of boys and girls related to programming in relation to computational practices using computer games, few were directly compared the relative performance across genders. In this study, boys and girls from the EG proposed more solution plans based on nesting and selection programming methods, thus using fewer code blocks than to their CG counterparts who used mostly simple selection and sequence.

7.3.8. Limitations

Inescapably, there were notable limitations in this study that should be referred. These are as follows:

a) The sample size was small to its number (n=50).
b) Non-equivalent groups design use to separate participants as similar as possible and compare fairly having lack of random assignment or any prior differences from participants from both groups may have an impact on this study's findings.
c) The convenience sampling that involved all participants, was up to a middle response from a part of the population from only three Greek high schools.
d) The three CS instructors and the supervising researcher during the entire teaching intervention gave sufficient support and feedback to each participant.
e) Even though the pre-questionnaire may indicate similarity in abilities, it was based on subjective self-reported data to separate participants into two groups.

Chapter 8: Educational implications for theory and practice

In response to limitations and in light of surge regarding the use of interactive environments which previous studies (e.g. Grover & Pea, 2013; Witherspoon et al., 2017) and literature reviews (Kafai & Burke, 2015; Lye & Koh, 2014) have been well-documented, the present thesis investigated the students' computational problem-solving strategies for solving simulated real-world problems created in OpenSim and in Scratch. It supports the opinion that a SG created in OpenSim that displays a more natural intuitive modality for user-interaction tasks can support greatly students' understanding in terms of problem-solving situations in simulated real-world contexts than in Scratch. The present thesis is also in the line of reasoning from future outlook or limitations which have been previously mentioned in terms of integrating SGs to CT instruction. In particular, this thesis has tried to give answers to a significant number of limitations that previous studies have noticed. First, it gives potential answers about the learning affordances of 3D VWs compared to other technologies such as VPEs in programming courses (Girvan et al., 2013). Second, it compares the learning gain between boys and girls in a controlled experimental design study (Liu et al., 2017). Third, it presents empirical evidence on how a SG can influence students' computational problem-solving strategies in programming courses at the high school level (Chao, 2016; Liu et al., 2011).

For this thesis's research aim and objectives, firstly, a theoretical design framework is proposed for the development and creation of a SG. Secondly, a preliminary and quasi-experimental (empirical) study were conducted. The findings from the preliminary study indicate that perceived learning support from the instructor combined with user interface design features and elements of a SG created in OpenSim with S4SL have positively affected students' learning involvement as well as their computational practices. Students were supported on learning how to think and practice "computationally" and achieved to analyze further how in-game elements should be mapped inside the RVC simulator. This process assisted students to develop skills related to CT skills in order to express their computational practices based on their own solution plans before start applying those plans into code. More specifically, such a process was regarded as essential for spotting and solving subparts of a computational problem inside the proposed 3D SG. This means that students were able:

a) to think critically and logically in order to communicate their solution plans by organizing correctly instructions and programming constructs in natural language in different tasks of a simulated real-world problem, and
b) to produce alternative computational practices with efficient and effective design patterns so as to apply successfully their solution plans into workable algorithms because they seemed to understand in-game visualized evocative spatial metaphors.

After the preliminary's study completion, a second empirical was conducted. A quasi-experiment was utilized to investigate the effects of a SG created in the 3D VW of OpenSim and in the VPE of Scratch on students' learning performance by assessing their computational problem-solving strategies for teaching and learning programming. Such a study was required to build more solid evidence based on the effectiveness and feasibility that a VPE such as Scratch and a 3D VW such as OpenSim combined with S4SL palette can offer. The findings revealed that instructive guided learning support alongside with a visual palette with code blocks from S4SL and natural intuitive modality for user interaction of a 3D SG has a significant and positive influence on students' learning outcomes based on expression and execution of their computational problem-solving strategies. Specifically, mean scores on pre-and-post questionnaires from the EG unveiled improvements higher than their CG counterparts in two aspects. First, participants from the former group created more complete computational instructions with unambiguous instructions and rules combined with programming constructs in order to program correctly using the proposed SG and be accomplished the learning goals. Second, participants from the EG proposed and expressed solutions not only with more correct computational concepts in natural language but also based on their practices into code than their counterparts who utilized Scratch.

To maximize further the students' learning performance in programming courses, the current thesis makes educational implications for theory and practice about the implementation and evaluation of scientifically-driven CT instruction using interactive environments. More specifically, the educational implications for practitioners and game designers are focused on the use of a SG that can enhance students' cognitive learning involvement for learning computer programming. Also, the implications of this thesis can inform scholars or educators about the use of the most potential user interface elements and features which can support a fading scaffolding instruction for students' achievements and outcomes. The theoretical implications are the following two. First, a theoretical design framework with specific guidelines and recommendation is proposed for designing a SG that can be developed by using interactive environments to support high school students' computational design, practice, and performance. Such a theoretical design framework can assist developers and educators to ensure that such a SG will provide the most appropriate features and elements to become the learning and teaching CT more effective.

Based on this thesis' studies, educators and scholars need to consider how to encourage the SG integration among girls and boys respecting gender equality in their in-game problem-solving tasks for learning how to program. The SG user interface design characteristics and features are considered as important on students' learning performance. More attention should be paid to in-game problem-solving tasks with a specific storyline with stages that include different levels of difficulty and objectives with characters in a digital environment that cannot cause any conflict of interest among students with a different gender. Students should learn how to formulate their thinking solution plans into abstract representations

using visual metaphors that can be projected using a SG in order to assist them to specify more precisely the algorithmic rules corresponding to fundamental programming constructs that need to be used in programs. The measurement of students' progress and learning performance through in-game activities follows a process that can allow them to apply a cognitive process by transforming their thinking solutions into the code for several problem-solving tasks. To this notion, a SG with an intuitive-natural modality for user-interaction can give to all players the opportunity to pay attention on the computational design of algorithmic problem-solving activities and more importantly to transform their computational practices as proposed solution plans into code to the given subparts of a main problem by avoiding syntax errors using a visual palette with code blocks. In addition, the reflective observation of the concrete visual experience assimilates abstract conceptualization without remaining tacit so as to facilitate students' understanding of how and why to use specific computational concepts to solve problems having two perspectives:

a) to create correct and complete computational instructions and rules specifying learning goals and
b) to develop an understanding of expressing and applying solution plans in terms of using cognitive thinking skills related to CT.

Second, this thesis suggests a teaching intervention with the use of a SG created in Scratch and OpenSim to foster CT instruction in high school programming courses within the operational-instructional context from CSTA and ISTE (2011). The proposed GBL teaching intervention emphasizes further to the important role of the instructor's support to all students' tasks for expressing sufficiently alternative and self-explanatory solutions through in/-formal instructional settings. The instructor's feedback and guidance also facilitated students to rationalize their decisions taken on the cognitive aspects of computational practices into code. Specifically, an instructional-guided approach that is accompanied with the use of a SG can be appropriate for understanding how the cognitive thinking process of solving a problem with skills related to CT such as problem-solving, logical and critical thinking. Such an approach is reflected inside their proposed computational practices for the execution and verification of students' thinking solution plans correctness into code.

The current thesis provides also practical implications. First, in terms of educational-instructional contexts, scholars and educators need to consider the realistic simulated representational fidelity of in-game elements and features in relation to the player's awareness and presence. User interface design features and elements of a 3D VW supported greatly players to map out in-game subparts of the main problem greater as they were able to configure grades by exploring and understanding the consequences arising from their choices made into the RVC simulator given the appropriate feedback to their actions. OpenSim seemed to assist players to think and transform alternative algorithms into workable solution plans and apply more accurate computational concepts and practices as design patterns via S4SL. The high representational fidelity of in-game elements and features in relation to the players' awareness allowed them to study

multiple traces of threads and consider several alternative choices. They have taken seriously into account for spotting and solving subparts of the main problem using skills related to CT such as algorithmic and critical thinking. This implies the transformation of a thinking process to be proposed a solution with accurate instructions or rules associated with the simulated problem-solving features and elements of the RVC simulator created in OpenSim. Moreover, based on the results from the experimental study, participants from the EG were focused on how to solve subparts of the computational problem via the proposed SG to think critically their solutions as workable algorithms and after that to start coding their computational practices using S4SL to propose solutions as design patterns than those who utilized Scratch.

Second, some practical implications need to refer for instructional design educators and scholars. A set of key recommendations about the design guidelines, design criteria, components and features to recommend building upon the experience gained from subsequent design and evaluation of the proposed SG for CT instruction are the following:

a) *Alignment of in-game learning goals and objectives of programming courses:* The alignment of in-game goals with the learning objectives of programming courses can help players to consider a clear indication about what they exactly need to accomplish in an effort to be assessed effectively any knowledge gained from each game task in specific time-limited tasks.
b) *Various quests and goals with different levels of difficulties:* Trace balancing among quests and goals to all in-game stages need to be connected from simple to more complicated tasks, in which each player (boys and girls) can navigate and explore fluidly as the time passing or if specific goals from each stage are accomplished properly.
c) *Logical in-game consequences of players' actions:* Logical reasoning of visual entities' actions combined with in-game elements/objects and unambiguous instructions from the CS instructor are more important than a collection of random events without meaning. The interaction with the visual elements should assist players to receive feedback about the consequences of each choice made.
d) *Exploration and accessibility:* In-game availability for free exploration and accessibility to each stage should have the appropriate features and elements to motivate players. It needs to be assumed that players' actions would have an increased level of efforts and outcomes in order to be accomplished certain in-game learning goals.
e) *A specific learning scenario that cannot cause gender biases:* A specific scenario with a rational structure can help players to think about essential choices for solving each subpart of the main problem without having to respect "gender equality". Both in terms of the choices inherent with a specific storyline and with respect to the constraints/limitations that are provided, a SG need to assist players to understand their in-game actions, outcomes and consequences based on the feedback received of his/her actions inside it.

f) *The player's awareness and presence:* Awareness and presence of players need to be visually appealing and distinctive. They need to have some choices to configure a virtual character to be ensured gender equity and ability so that can someone contribute based on his/her own willing.

g) *User design features and elements with natural intuitive modality:* In-game user interface design features and elements with a more natural intuitive modality for users' actions combined with simulations where various evocative spatial metaphors that have certain information can pave a pathway from problem formulation to solution expression supported by skills related to the logical reasoning and critical thinking on players' actions.

h) *Core mechanics for awards and punishments:* Pre-defined core mechanics to ensure possible solutions, mistakes and/or winning grades need to be announced at the beginning to each player. Pennant visual spotting via checkpoints can maintain players' interest as a sequence of logical reasoning steps which may be useful to understand their in-game progress when specific in-game learning goals are properly achieved.

i) *Game mechanics:* Visualized program tracking mechanisms or simultaneously selective processing of every target item (e.g., visual elements/objects) are important in the fading scaffolding CT instruction. In such a process, players can assess automatically the correctness of their own computational practices with the use of control flow blocks in terms of identifying the effectiveness and efficiency of the alternative design patterns as solutions to each part of the main problem.

j) *Simulation of embodied experiences/ideas:* Simulation of embodied experiences/ideas through guided discovery learning processes can foster players' problem-solving ability in spotting and in solving a computational problem. Such a process can assist players to experience and realize how programming knowledge is gained from the formulation of computer programs and evaluation of the consequent results using a visual palette of colored code blocks in regard to programming syntax so as to apply their solution plans to avoid any potential "cognitive overload".

According to the above, a revised game design map can be proposed by aligning design criteria, game guidelines and essential components with features from the *PIVB* framework which have empirically been investigated to support students' achievements and outcomes. *Figure 8-1* below depicts how specific colored frames of design criteria can be revised and aligned with the initial game design criteria and elements/features following the evaluation results from both studies. These are the following:

a) the *"Learning content"* can be aligned with *"the alignment of in-game learning goals and objectives of programming courses"* and *"simulation of embodied experiences/ideas"* (blue color frame),

b) the *"Gender issues"* can be aligned with *"various quests and goals with different levels of difficulties", "a specific learning scenario that cannot cause gender biases"* and *"the player's awareness and presence"* (orange color frame),

c) the *"Use interface design features and elements"* can be aligned with *"logical in-game consequences of players' actions", "the exploration and accessibility",* and *"the user design features and elements with natural intuitive modality",* (yellow color frame) and finally

d) the *"Awards and punishments"* can be aligned with *"the core mechanics for awards and punishments"* and *"game mechanics"* (green color frame).

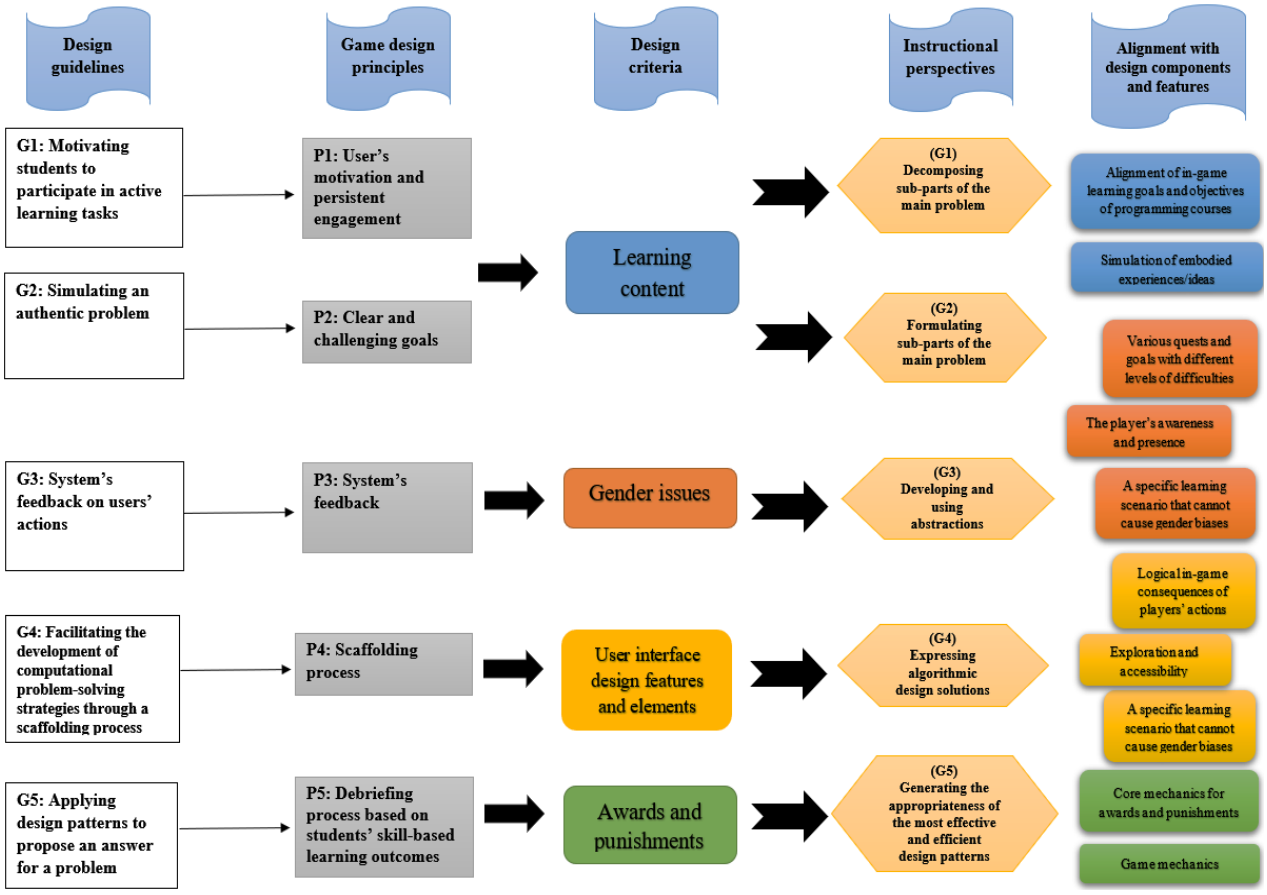


Figure 8-1: A revised design map constructed by following the game guidelines and principles of the PIVB framework

Chapter 9: Conclusions

The maturity and accessibility of computer technologies have prompted educators to harness the power of GBL in educational settings in favor of creating practical and highly interactive visual forms of learning for different learning subjects and domains such as those of CS and programming. Furthermore, GBL approaches using interactive environments has become a flourishing area for education research in computer programming that is quickly gaining momentum since it has the potential to enable new forms of CT instruction and transform the learning experience. So far, a significant number of literature reviews (Grover & Pea, 2013; Kafai & Burke, 2015; Lye & Koh, 2014) have suggested that further studies need to investigate the effects of using interactive environments for CT instruction in K-12 programming courses and their impact on students' understanding in terms of starting how to think before start coding. Indeed, it is arguable if the way of using VPEs and 3D VWs can support students to take advantage of intuitive, natural modality for user-interaction tasks in activities that required for the development and use of skills related to CT having a more general understanding about the use of computational concepts to solve problems (Howland & Good, 2015; Mouza et al., 2016). Therefore, there has been remarkably little research made to investigate if playing a computer game created in interactive environments have an impact on students' computational understanding to assess their learning performance.

To give answers on the above research challenge, this thesis provides empirical evidence from the exploitation of a SG following an instructive guided approach with exercises focus on programming and investigates its' impact on students' learning performance by assessing their computational problem-solving strategies (i.e. computational design, computational practices, and computational performance). The proposed SG was created in two interactive environments with a different user interface design features and elements to address the difficulties encountered in learning and teaching how to use fundamental programming constructs for solving simulated real-world problems. The first was the visual programming of Scratch and the second was the 3D VW of OpenSim, in which participants needed also to use the S4SL visual palette to apply their solution plans into code.

The aim of this thesis is twofold. The first is to propose a theoretical design framework for the development and creation of a SG. The second is to investigate and analyze the effects of a SG on high school students' learning performance in programming courses. To achieve the twofold aim and objectives of this thesis, an initial step was an overview of the research field and the impact of game playing approaches to support CT instruction and computational problem-solving through programming courses.

To achieve the first aim, the current thesis proposes a theoretical design framework with specific design guidelines and recommendations (Pellas & Vosinakis, 2017a). Such an effort was made to develop a simulated problem-solving environment that can assist students to support and understand how to use

skills related to CT by expressing in natural language (pseudocode) and by applying into code their computational problem-solving strategies. In particular, the RVC simulator has provided various visual features and elements for players to compose and test their programs. Such a SG seemed to assist especially high school students to propose different design patterns as solution plans and evaluate the consequence of the instructions relating to programming constructs that they would like to propose in simulated problem-solving contexts. Accordingly, it is of great importance to mention what makes the RVC simulator different than previous gaming prototypes. The main design features and elements that differ from the proposed SG from the rest of the existing educational games are as follows:

- Students had the chance to develop cognitive thinking skills related to CT and practice into code their solution plans with the same learning goals and stages with different levels of difficulty to accomplish by playing a SG created in Scratch and OpenSim with S4SL.
- In-game use of visual elements inside the proposed SG (RVC simulator) was displayed as evocative spatial metaphors of basic geometric shapes (e.g., triangle, square, and hexagon) assisting novices to think and practice "computationally".
- Abstract spatial representations of geometric shapes were extensive considering the different design features and characteristics of OpenSim or Scratch, which need to be used by participants to traverse the RVC a specific cleaning path taking into consideration the spatial layout in each room. Students needed to use critical and logical reasoning skills to propose their solution plans and after that started to code.
- The intuitive modality for user-interaction simulation tasks and realistic simulated representational fidelity of the proposed SG seemed that can assist players to analyze easier each problem-solving task better in order to propose effective and alternative solution plans that were applied or even being re-used with similar design patterns to other in-game stages later.

To achieve the second aim, a preliminary and an experimental study were conducted. Generally, both are in line with previous works (Chao, 2016; Grover et al., 2015; Werner et al., 2015), which have suggested that when students understand visually and conceptually how and what they need to program, they are also able for spotting and solving other subparts of the main problem within the game. Specifically, and more importantly was the measurement of boys' and girls' learning performance, and in specific the way that they have described and applied their computational problem-solving strategies. In an effort to understand the effects of a SG created in Scratch and OpenSim with S4SL to solve the same subparts of a computational problem, this thesis investigated students' design patterns as programming behaviors integrated into visual elements to be analyzed their computational problem-solving strategies. Thus, an empirical evaluation conducted to examine students' learning performance using the proposed SG with a view of supporting them:

a) to express not only by describing accurate rules, behaviors, situations which are combined as command/instructions from a natural language to workable algorithms but also by investigating any potential limitations, errors of commissions and/or omissions using programming constructs when they proposed different solutions to a computational problem,
b) to apply and execute computational practices into the code to be proposed solutions for the same computational problem-solving tasks in two interactive environments (OpenSim combined with S4SL and Scratch) which have different user interface design features and elements in order to compare their design patterns, and lastly
c) to identify the learning effect of a SG created in two interactive environments which have different user design features and characteristics in order to measure the learning performance of boys and girls before and after the teaching intervention.

To understand further the learning gain in terms of enhancing the learning outcomes of both groups, the findings from this experimental study identified the differences among rules/concepts, the use of programming constructs or instruction commands through logical steps and the expression of errors ranging from the correct expressions in natural language to the implementation of workable algorithms into code for the analysis of design patterns. Such an evaluation process was as one of the most indicative for testing and debugging thinking solution correctness of thinking solution plans for each of the subparts of the main problem. This thesis' findings have pointed out some important aspects to be considered in relation to the students' learning performance measurement in overall. Mean scores on the pre-and-post questionnaires and worksheets from participants from the EG who used OpenSim with S4SL has revealed reasonable improvements. The most important improvements which have been unveiled are the following:

a) the creation of correct and complete computational instructions with rules to specify learning goals fundamental to computing, and
b) the expression and implementation of computational concepts related to CT skills usage which were higher than their CG counterparts.

Reflecting on both studies' findings that are described in this thesis, a substantial number of instructional-pedagogical and technological-functional challenges for the successful integration of the proposed SG into formal and informal teaching contexts are clarified below.

- Using of the RVC simulator combined with the proper support of a CS teacher can assist high school students to develop and use cognitive thinking skills related to CT, such as problem-solving, critical thinking and creativity for the expression and implementation of their solution plans into workable algorithms. Nonetheless, such a SG alone cannot provide the expected learning outcomes, because it was observed by the preliminary and experimental study that students often needed explanations and clarifications regarding mainly the introduction to the first lectures until a

complete understanding of the proposed RVC simulator's functions. The use of the same command and programming constructs of the S4SL visual palette as those of Scratch has generally assisted students from EG to be more easily involved in coding tasks and more on how to use tools and camera of OpenSim to map out correctly.

- Since a SG cannot alone provide the anticipated learning outcomes, an instructional framework is also needed. In addition, one of the appropriate teaching and organizational frameworks for activities has been proposed by CSTA and ISTE (2011). This thesis provides specific problem-solving tasks which can be carried in formal (inside the school contexts) and informal (within the university's computer laboratories through workshops) to provide instructional support on students' learning performance. Such an instructional framework aimed at presenting a series of specific actions and a set of specifications, regulations with rules that define the actions and orders in which students tend to apply their solution plans in order to create favorable and efficient learning conditions for better use of the proposed game.
- The gradually fading scaffolding instruction from a CS teacher and the different levels of difficulties into the stages of a SG for players' progress is another important issue. Such a process has also assisted players to solve subparts of the main problem, maintaining their interest undisturbed to gain confidence in order to provide and communicate some good computational practices into code. Students who have clear instructions seemed to achieve all in-game goals in each teaching intervention more than those who showed little mood and perhaps did not want to continue in the experimental process.
- The ability of students to express and apply a set of solution plans in different in-game stages with a gradual difficulty to each one. Such a design decision seemed to assist them to be focused more on the problem- solving and the alignment between what they would like to solve and what a solution plan into code would contain. This was reflected by their answers to several questions and the data gathered by decoding quantitative and qualitative measurements in the research process that has positively influenced the development of skills related both to CT and programming.

This thesis advocates that a 3D VW such as OpenSim combined with a visual palette of S4SL can provide a digital environment to support GBL activities related to CT instruction in high school programming courses. Moreover, the proposed SG that is created in a 3D VW assisted students to have greater learning performance in terms of computational design, computational practices, and perspectives in contrast to a VPE, such as Scratch. The potential use of a 3D VW features and characteristics permit a wide use of computational problem-solving tasks in simulated real-world contexts with all of its benefits that boys and girls from the EG who finally achieved based on their grades than their counterparts who used Scratch.

There are several limitations that have to be noted in the preliminary and empirical study. To this notion, this thesis's results are limited by a number of factors described above, and thus there are some consequences. First, the non-random assignment of the CG and the correlational relationship between progress and gains do not address causality. Therefore, it is possible that other unobserved factors, such as supplemental materials developed by CS teachers that have been previously used in their programming courses or even other students' characteristics such as age or class attendance accounted for their abilities to move further into the curriculum and score higher on the post-questionnaires to achieve better scores as indicated by their computational performance. Second, the extent to which CS teachers have access and incorporated via SGs, specifically in the Greek curriculum can greatly vary, as each school regions may have differential access to other (or the same) computing devices. Third, feedback on students' actions from the three CS instructors and/or the supervising researchers during the entire teaching process may have to avoid any potential gender biases or misunderstandings about the use of a SG's features and elements may impact on their learning performance in overall.

This thesis suggests some key recommendation for further research. First, future studies in K-12 curriculum needs to investigate CS teachers' input for gathering information about what additional materials should be taken into consideration and which could contribute to the successful utilization of interactive environments in programming courses. Also, future works may investigate relevant issues with larger sample sizes and longer time experiments. In particular, longitudinal studies with long-term analysis of students' learning experiences in programming courses alongside with a larger sample to provide additional evidence based on their solutions to several real-life computational problems. Such an effort can also provide important insights regarding the suitability of interactive environments for interdisciplinary learning in STEM subjects.

Second, for the same game concept that is described in the RVC simulator, an empirical investigation needs to be conducted in order to measure students' learning performance. For instance, a comparative study can be also suggested between a group that can use a *LEGO Mindstorms NXT* accompanied by a wide range of sensors and an interactive environment from 3D VWs or VPEs. Such a study can give evidence if game design features with real and simulated natural intuitive modality can assist players to understand better certain information that may pave a pathway from problem formulation to solution expression using the former technology in a real environment and the latter in a computer simulation with the same problem-solving tasks.

Third, future research needs also to investigate further the proper exploitation and integration of a learning analytics subsystem in the RVC simulator in order to gather information data recorded from players' actions and how such information can be used for the re-design and monitor CS instructors more effectively in K-12 programming courses.

References

ACM Education Policy Committee. (2014). *Rebooting the pathway to success: Preparing students for computing workforce needs in the United States.* Retrieved 23 March 2017 from http://pathways.acm.org/ACM_pathways_report.pdf.

Adams, E. (2009). *Fundamentals of game design*. New Riders: Berkeley, CA.

Adler, R. & Kim, H. (2017). Enhancing future K-8 teachers' computational thinking skills through modeling and simulations. *Education & Information Technologies*, *23*(4), 1501–1514.

Alessi, S.M. & Trollip, S.P. (2001). *Multimedia for learning: Methods and development*. Boston, MA; Allyn and Bacon.

Afari, E. & Khine, M.S. (2017). Robotics as an educational tool: Impact of Lego Mindstorms. *International Journal of Information and Education Technology, 7*(6), 437–442.

Ambrosio, A., Almeida, L., Macedo, J., & Franco, A. (2014). *Exploring core cognitive skills of computational thinking.* PPIG 2014 - 25th Annual Workshop.

Anderson, N., Lankshear, C., Timms, C. & Courtney, L. (2008). 'Because it's boring, irrelevant and i don't like computers': why high school girls avoid professionally-oriented ICT subjects. *Computers & Education, 50*(3), 1304–1318.

AP Computer Science Principles (2017). *AP Computer Science Principles Including the Curriculum Framework.* New York, NY: College Board.

Arapoglou A., Makoglou, Economakos, N. & Fitrokos, G. (2003). Computer Science at Gymnasium. Organization of textbook editions: Athens (In Greek).

Bachu, E. & Bernard, M. (2014). Visualizing problem solving in a strategy game for teaching programming. *In Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)* (p. 1). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

Baldwin, D. (1996). Discovery learning in computer science. *SIGCSE Bulletin 28*(1), 222-226.

Barr, V. & Stephenson, C. (2011). Bringing computational thinking to K–12: What is involved and what is the role of the computer science education community? *ACM Inroads, 2*(1), 48–54.

Bargas-Avila, J. A. & Hornbæk, K. (2011). Old wine in new bottles or novel challenges: a critical analysis of empirical studies of user experience. In *SIGCHI* (pp. 2689–2698). New York, USA: ACM.

Bell, T., Alexander, J. & Freeman, I. & Grimley, M. (2008). *Computer science without computers: new outreach methods from old tricks.* In: 21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2008) (pp. 127-133). NACCQ: Auckland, NZ.

Bell, M. (2008). Toward a definition of virtual worlds. *Journal of Virtual Worlds Research, 1*(1). Retrieved December 1, 2008, from http://journals.tdl.org/jvwr/article/view/283

Ben-Ari, M. (2001). Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching, 20*(1), 45-73.

Bienkowski, M., Snow, E., Rutstein, D. W. & Grover, S. (2015). *Assessment design patterns for computational thinking practices in secondary computer science: A first look (SRI technical report)*. Menlo Park, CA: SRI International. Retrieved from http://pact.sri.com/resources.html.

Bickford, T. (2011). *Advanced programming with LEGO® NXT Mindstorms*. Retrieved 18 September 2018 from http://www.mainerobotics.org/uploads/8/3/4/4/8344007/advanced_programmingforprint.pdf

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A. & Engelhardt, K. (2016). *Developing computational thinking in compulsory education.* Spain: Joint Research centre.

Bodrova, E. & Leong, D. J. (2003). The importance of being playful. Educational Leadership, 60(7), 50–53.

Brennan, K. & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *In Proceedings of the 2012 Annual meeting of the American Educational Research Association, Vancouver, Canada*. Retrieved 29 March 2013 from http://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf

Burke, Q. (2012). The markings of a new pencil: Introducing programming-as-writing in the middle school classroom. *The Journal of Media Literacy Education, 4*(2), 121-135.

Carbonaro, M., Szafron, D., Cutumisu, M. & Schaeffer, J. (2010). Computer-game construction: a gender-neutral attractor to computing science. *Computers & Education, 55*(3), 1098–1111.

Carter, L. (2006). Why students with the apparent aptitude don't choose to major in Computer Science. *SIGCSE Bulletin, 38*(1), 27–31.

Chao, P.-Y. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Computers & Education, 95*(2), 202-215.

Chetty, J. (2015). Lego mindstorms: Merely a toy or a powerful pedagogical tool for learning computer programming? *In Proceedings of the 38th Australasian Computer Science Conference (ACSC 2015), vol. 27* (pp. 30-36). Sydney: Australia.

Coban, M., Karakus, T., Karaman, A., Gunay, F. & Goktas, Y. (2015). Technical problems experienced in the transformation of virtual worlds into an education environment and coping strategies. *Educational Technology & Society, 18*(1), 37–49.

Cohen, L., Manion, L. & Morrison, K. (2011). *Research methods in education*. Abingdon, Oxon: Routledge.

Computing at School. (2015). *Computational thinking: A guide for teachers*. Retrieved 5 February 2018 from http://computingatschool.org.uk/computationalthinking.

Cooper, S., Dann, W. & Pausch, R. (2003). Using animated 3D graphics to prepare novices for CS1. *Computer Science Education, 13*(1), 3-30.

Costa, J. & Miranda, G. (2016). Relation between Alice software and programming learning: A systematic review of the literature and meta-analysis. *British Journal of Educational Technology*. DOI:10.1111/bjet.12496.

Csikszentmihalyi, M. (1991). *Flow: The Psychology of Optimal Experience*. Harper Perennial, New York.

Culp, R.E. (1998). Adolescent girls and outdoor recreation: A case study examining constraints and effective programming. *Journal of Leisure Research, 30*(3), 356-379.

Dagdilelis, V., Satratzemi, M. & Evangelidis, G. (2004). Introducing secondary education students to algorithms and programming. *Education & Information Technologies, 9*(2), 159–173.

Dalgarno, B. & Lee, M. J. W. (2010). What are the learning affordances of 3-D virtual environments? *British Journal of Educational Technology, 41*(1), 10-32.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39*(2), 237-267.

Davies, S. (2008). The effects of emphasizing computational thinking in an introductory programming course. *38th Annual Frontiers in Education Conference (FIE 2008).* IEEE Saratoga Springs, New York, USA. DOI:10.1109/fie.2008.4720362.

Denner, J., Werner, L. & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education, 58*(1), 240-249.

Denner, J., Campe, S. & Werner, L. (2019). Does computer game design and programming benefit children? A meta-synthesis of research. *ACM Transaction in Computing Education, 19*(3), 1-35.

Denning, P.J. (2000). Computer science. *In Encyclopedia of Computer Science* (pp. 405–419). Chichester, UK: John Wiley & Sons Ltd.

de Freitas, S. & Oliver, M. (2006). How can exploratory learning with games and simulations within the curriculum be most effectively evaluated? *Computers & Education, 46*(3), 249–264.

de Raadt, M., Watson, R. & Toleman, M. (2006). Chick sexing and novice programmers: explicit instruction of problem solving strategies. *Eighth Australasian Computing Education Conference* (pp. 55-62). Hobart, Australia.

de Raadt, M. (2007). A review of Australasian investigations into problem solving and the novice programmer. *Computer Science Education, 17*(3), 201-213.

de Raadt, M., Watson, R. & Toleman, M. (2009). Teaching and assessing programming strategies explicitly. *In the 11th Australasian computing education conference (ACE)* (pp.45-54). Wellington, New Zealand.

Detsikas, N. & Alimisis, D. (2011). Status and trends in educational robotics worldwide with special consideration of educational experiences from Greek schools. *In Proceedings of the International Conference on Informatics in Schools: Situation, Evolution and Perspectives* (pp. 1-12). Bratislava: Comenius University.

Dickey, M. (2005). Three-dimensional virtual worlds and distance learning: Two case studies of active worlds as a medium for distance education. *British Journal of Educational Technology, 36*(3), 439–461.

Dondi, C. & Moretti, M. (2007). A methodological proposal for learning games selection and quality assessment. *British Journal of Educational Technology, 38*(3), 502–512.

Dumbleton, T. & Kirriemuir, J. (2006). Digital games and education. In J. Rutter & J. Bryce (Eds.), *Understanding digital games* (pp. 223–240). London: Sage.

European School net (2015). *Computing our future. Computer programming and coding Priorities, school curricula and initiatives across Europe*. Retrieved 12 August 2017 from http://fcl.eun.org/documents/10180/14689/Computing+our+future_final.pdf/746e36b1-e1a6-4bf1-8105-ea27c0d2bbe0

European Commission (2016). *Coding and computational thinking on the curriculum. Key messages of PLA#2. Helsinki: Education and Training*. Retrieved 25 March 2017 from https://ec.europa.eu/education/sites/education/files/2016-pla-coding-computational-thinking_en.pdf

European Commission (2016). *"She Figures 2015"*. Retrieved 20 March 2017 from https://ec.europa.eu/research/swafs/pdf/pub_gender_equality/she_figures_2015-final.pdf

Esteves, M., Fonseca, B., Morgado, L., & Martins, P. (2011). Improving teaching and learning of computer programming through the use of the Second Life virtual world. *British Journal of Educational Technology, 42*(4), 624–637.

Fabricatore, C. (2007). Gameplay and game mechanics: A key to quality in videogames. *In ENLACES (MINEDUC Chile) -OECD Expert meeting on videogames and education*, Santiago de Chile, Chile.

Feurzeig, W. & Papert, S. A. (2011). Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments, 19*(5), 487–501.

Fluck, A., Webb, M., Cox, M., Angeli, C., Malyn-Smith, J., Voogt, J. & Zagami, J. (2016). Arguing for Computer Science in the school curriculum. *Educational Technology & Society, 19*(3), 38–46.

Garris, R., Ahlers, R. & Driskell, J. (2002). Games, motivation and learning: a research and practice model. *Simulation & Gaming, 33*(4), 441-467.

Garneli, V., Giannakos, M. & Chorianopoulos, K. (2015). Computing Education in K-12 Schools: A Review of the Literature. *IEEE Global Engineering Education Conference (EDUCON)* (pp. 536-544). IEEE: Tallinn, Estonia.

Garneli, V. & Chorianopoulos, K. (2017). Programming video games and simulations in science education: exploring computational thinking through code analysis. *Interactive Learning Environments*. DOI: 10.1080/10494820.2017.1337036.

Gee, J. P. (2007). *Good video games and good learning: Collected essays on video games, learning, and literacy*. New York: Peter Lang

Girvan, C. (2018). What is a virtual world? Definition and classification. *Educational Technology Research and Development.* DOI: https://doi.org/10.1007/s11423-018-9577-y.

Girvan, C., Tangney, B. & Savage, T. (2013). SLurtles: Supporting constructionist learning in 'Second Life'. *Computers & Education 61*(2), 115-132.

Good, J., & Robertson, J. (2006). CASS a framework for learner centred design with children. In *International Journal of Artificial Intelligence in Education, 16*(4), 381–413.

Good, J. (2011). Learners at the wheel: novice programming environments come of age. International *Journal of People-Oriented Programming, 1*(1), 1-24.

Good, J., Howland, K. & Thackray, L. (2008). Problem-based learning spanning real and virtual worlds: A case study in Second Life. *ALT-J, Research in Learning Technology, 16*(3), 163–172.

Good, J., Howland, K. & Nicholson, K. (2010). Young people's descriptions of computational rules in role-playing games: an empirical study. *In 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).*

Good, J. & Howland, K. (2016). Programming language natural language? Supporting the diverse computational activities of novice programmers, *Journal of Visual Languages and Computing*. DOI: http://dx.doi.org/10.1016/j.jvlc.2016.10.008.

Graham, S. & Latulipe, C. (2003). Computer science girls rock: Sparking interest in computer science and debunking the stereotypes. *In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp.322-326). New York, NY: ACM Press.

Grover, S. & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38–43.

Grover, S., Pea, R. & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computers Science Education, 25*(2), 199-237.

Ha, O. & Fang, N. (2018). Interactive virtual and physical manipulatives for improving students' spatial skills. *Journal of Educational Computing Research, 55*(8), 1088–1110.

Haberman, B. & Kolikant, Y.B.D. (2001), Activating "Black Boxes" instead of opening "Zippers" - a method of teaching novices basic CS concepts. *Proceedings of the ACM ITiCSE '01 Conference* (pp. 41-44). Canterbury, UK.

Hamlen et al. (2018). Effects of teacher training in a computer science principles curriculum on teacher and student skills, confidence, and beliefs. *Proceeding SIGCSE '18 Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 741-746). Baltimore, Maryland, USA: ACM.

Hamrick, T. & Hensel, R. (2013). Putting the fun in programming fundamentals - Robots make programs tangible. *120th ASEE Annual conference and exposition*. Paper ID #6062. Atlanta: USA.

Hellenic Pedagogical Institute (2003). Diathematikon Programma: A Cross Thematic Curriculum Framework for Information and Communication Technology - Compulsory Education. Retrieved 15 March 2016 from http://www.pi-schools.gr/download/programs/depps/english/16th.pdf (in Greek).

Hew, K. F. & Cheung, W. S. (2014). Students' and instructors' use of Massive Open Online Courses (MOOCs): motivations and challenges. *Educational Research Review*, *12*(1), 45-58.

Howland, K. & Good, J. (2015). Learning to communicate computationally with Flip: A bi-modal programming language for game creation. *Computers & Education, 80*(2), 224–240.

Hong, J. C. & Liu, M. C. (2003). A study on thinking strategy between experts and novices of computer games. *Computers in Human Behavior, 19*(2), 245–258.

Horn, M. S. & Jacob, R. (2007). Tangible programming in the classroom with Tern. *In: CHI '07 Conference on Human factors in computing systems* (pp. 965-970). San Jose: USA.

Hsu, T. C., Chang, S. C. & Hung, Y. T. (2018). How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education, 126*(2), 296–310.

Iacovides I, Cox AL, McAndrew P, Aczel J. & Scanlon, E. (2014). Game-play breakdowns and breakthroughs: exploring the relationship between action, understanding, and involvement. *Human & Computer Interaction, 22-30*(3-4), 202–310.

International Society for Technology in Education (ISTE) and the Computer Science Teachers Association (CSTA). (2011). Operational definition of computational thinking for K-12 Education. Retrieved 26 August 2016 from http://www.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf?sfvrsn=2

Ismail, M., Nasir, N. & Naufal, U. (2010). Instructional strategy in the teaching of computer programming: A need assessment analyses. *The Turkish Online Journal of Education Technology, 9*(2), 125-131.

Jakos, F. & Verber, D. (2016). Learning basic programing skills with educational games: A case of primary schools in Slovenia. Journal of Educational Computing Research. DOI: 10.1177/0735633116680219.

Jonsson, A. & Svingby, G. (2007). The use of scoring rubrics: Reliability, validity and educational consequences. *Educational Research Review, 2*(2), 130–44.

Johnson, L., Smith, R., Willis, H., Levine, A., & Haywood, K. (2011). *The 2011 Horizon Report*. Austin, Texas: The New Media Consortium.

Kabátová, M., Jaskova, L., Lecky, P. & Lassakova, V. (2012). Robotic activities for visually impaired secondary school children. *Proceedings of 3rd International Workshop Teaching Robotics, Teaching with Robotics Integrating Robotics in School Curriculum* (pp. 22-31). Trento, Italy.

Kafai, Y. & Burke, Q. (2015). Constructionist Gaming: Understanding the benefits of making games for learning. *Educational Psychologist. Special Issue: Psychological perspectives on digital games and learning, 50*(4), 313-334.

Kalelioglu, F., Gülbahar, Y., Akçay, S. & Dogan, D. (2014). Curriculum integration ideas for improving the computational thinking skills of learners through programming via scratch. *In Proceedings of the 7th international conference on informatics in schools: Situation, evolution and perspectives* (pp. 101–112). Istanbul, Turkey.

Kalelioglu, F., Gülbahar, Y. & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing, 4*(3), 583-596.

Kafai, Y. B., Lee, E., Searle, K., Fields, D., Kaplan, E. & Lui, D. (2014). A crafts-oriented approach to computing in high school: Introducing computational concepts, practices, and perspectives with electronic textiles. *ACM Transactions on Computing Education* 14, 1, Article 1.

Kantor, G., Manikonda, V., Newman, A. & Tsakiris, D. (1996) Robotics for High School Students in a University Enviroment. *Computer Science Education, 7*(2), 257-278.

Klassner, F. & Anderson, S. (2003). Lego Mindstorms: Not just for K-12 anymore. *IEEE Robotics and Automation Magazine, 10*(2), 12–18.

Kay, J. et al. (2000). Problem-based learning for foundation computer science courses. *Computer Science Education 10*(2), 109–128.

Kelleher, C., Pausch, R. & Kiesler, S. (2007). Storytelling Alice motivates middle school girls to learn computer programming. *In Proceedings of CHI* (pp. 1455-1464). USA: ACM.

Kiesmüller, U. (2009). Diagnosing learners' problem-solving strategies using learning environments with algorithmic problems in secondary education. *ACM Transactions on Computing Education, 9*(3), 17-26.

Kiili, K. (2005). Digital game-based learning: Towards an experiential gaming model. *The Internet and Higher Education, 8*(2), 13–24.

Kim, S. & Jeon, J. (2007). Programming LEGO Mindstorms NXT with visual programming. *In International Conference on Control, Automation and Systems* (pp. 2468-2472). Seoul, Korea.

Kirkwood, M. (2000). Infusing higher-order thinking and learning to learn into content instruction: A case study of secondary computing studies in Scotland. *Journal of Curriculum Studies, 32*(4), 509-535.

Koenemann, J. & Robertson, S. P. (1991). Expert problem-solving strategies for program comprehension. In S. P. Robertson, G. M. Olson and J. S. Olson (Eds.) Reaching Through Technology, Proc. ACM Conf. on Human Factors in Computing Systems CHI'91 (pp 125-130). Reading, MA: AddisonWesley.

Kong S.-C., Chiu M.M. & Lai M. (2018). A study of primary school students' interest, collaboration attitude, and programming empowerment in computational thinking education. *Computers & Education*, *127*(2), 178-189.

Koorsse, M., Cilliers, C. & Calitz, A. (2015). Programming assistance tools to support the learning of IT programming in South African secondary schools. *Computers & Education*, *82*(2), 162–178.

Korkmaz, Ö., Çakir, R. & Özden, M. Y. (2017). A validity and reliability study of the Computational Thinking Scales (CTS). *Computers in Human Behavior*. DOI: http://dx.doi.org/10.1016/j.chb.2017.01.005.

Knight, J. K. (2010). *Biology concept assessment tools: Design and use*. Microbiology, 5.

Kurland, D. M., Pea, R., Clement, C. & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2*(4), 429–458.

Lahtinen, E., Ala-Mutka, K. and Järvinen, H. (2005). A study of the difficulties of novice programmers. *In: Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (pp. 14–18). ACM: Caparica, Portugal.

Li, F. W. B. & Watson, C. (2011). Game-based concept visualization for learning programming. *In Proceedings at the 3rd International ACM workshop on multimedia technologies for distance learning* (pp.37-42). Scottsdale, AZ, USA.

Lim, J.K.S. & Edirisinghe, E.M. (2007). Teaching computer science using Second Life as a learning environment. In ICT: Providing choices for learners and learning. *In Proceedings ascilite Singapore*. Retrieved 30 October 2016 from http://www.ascilite.org.au/conferences/singapore07/procs/lim.pdf.

Lindberg, R. Laine, T. & Haaranen, L. (2018). Gamifying programming education in K-12: A review of programming curricula in seven countries and programming games. *British Journal of Educational Technology*. Doi:10.1111/bjet.12685.

Linden Lab (2011). Second Life Education: The virtual learning advantage. Retrieved 23 October 2015 from http://lecs-static-secondlife-com.s3.amazonaws.com/work/SL-Edu-Brochure-010411.pdf

Liu, C. C., Cheng, Y. B. & Huang, C. W. (2011). The effect of simulation games on the learning of computational problem solving. *Computers & Education*, *57*(5), 1907–1918.

Liu, Z., Zhi, R., Hicks, Z. & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*. DOI: 10.1080/08993408.2017.1308651.

Lockwood, J. & Mooney, A. (2017). *Computational thinking in education: Where does it fit? A systematic literary review.* Retrieved from https://arxiv.org/ftp/arxiv/papers/1703/1703.07659.pdf.

Lye, S. Y. & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior, 41*(3), 51–61.

Luxton-Reilly, A., Ibrihim, S., Brett, A. Becker, A., Giannakos, M., Kumar, A.N., Ott, L., Paterson, J. Scott, M.J., Sheard, J. & Szabo, C. (2018). Introductory programming: A systematic literature review. *In Proceedings of the 2018 ITiCSE Conference on Working Group Reports* (ITiCSE-WGR '18) (pp. 284–289). ACM, New York, NY, USA.

Maddux, C. D. & Rhoda E. C. (1984). Logo is for all children: Learning with the turtle. *Exception Parent, 14*(3), 15-18.

Malone, T.W. (1980). *What makes things fun to learn? A study of intrinsically motivating computer games*. (Report CIS-7). Palo Altao, CA: Xerox Palo Alto Research Center.

Marshall, P. (2007). Do tangible interfaces enhance learning? *Paper presented at the Proceedings of the 1st International Conference on Tangible and Embedded Interaction* (pp. 163-170). Baton Rouge, Louisiana, USA.

Marx, J. D. & Cummings, K. (2007). Normalized change. *American Journal of Physics*, 75-87.

McGill, T.J. & Volet, S.E. (1997). A conceptual framework for analysing students' knowledge of the various components of programming. *Journal of Research on Computing in Education, 29*(3) 276-297.

Miglino, O., Lund, H. & Cardaci, M. (1999). Robotics as an educational tool. *International Journal of Interactive Learning Research, 10*(1), 25-47.

Mouza, C. Marzocchi, A., Pan, Y & Pollock, L. (2016). Development, implementation, and outcomes of an equitable computer science after-school program: Findings from middle-school students. *Journal of Research on Technology in Education, 48*(2), 84-104.

Myers, B., Pane, J. & Ko, A. (2004). Natural programming languages and environments. *CACM. 47*(9), 47-52.

Moorman, P. & Johnson, E. (2003). Still a stranger here: Attitudes among secondary school students towards computer science. *In Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 193-197). New York, NY: ACM Press.

National Research Council (NRC). (2010). *Committee for the Workshops on Computational Thinking: Report of a workshop on the scope and nature of computational thinking*. Washington, DC: National Academies Press.

Oddie, A., Hazlewood, P., Blakeway, S. & Whitfield, A. (2010). Introductory problem-solving and programming: Robotics versus traditional approaches. *Innovation in Teaching and Learning in Information and Computer Sciences, 9*(2), 1-11.

Pane, J., Ratanamahatana, J. & Myers, B. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies, 54*(2), 237-264.

Papastergiou, M. (2009). Digital game-based learning in high school computer science education. *Computers & Education, 52*(1), 1–12.

Papert, S. (1980). *Children, computers and powerful ideas*. New York: Basic Books publishers.

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning, 1*(1), 95-123.

Partnership for 21st Century Skills. (2009). *Framework for 21st century learning*. Retrieved from http://www.p21.org/our-work/p21-framework.

Pellas, N. & Vosinakis, S. (2017a). How can a simulation game support the development of computational problem-solving strategies? *In IEEE Global Engineering Education Conference* (pp. 1124-1131). IEEE: Greece, Athens.

Pellas, N. & Vosinakis, S. (2017b). Learning to think and practice computationally via a 3D simulation game. *In the 11th International Conference on Interactive Mobile Communication, Technologies and Learning - "Advances in Intelligent Systems and Computing"* (pp. 193-204). Springer: Thessaloniki, Greece.

Pellas, N. & Vosinakis, S. (2018). The effect of computer simulation games on learning introductory programming: A comparative study on high school students' learning performance by assessing computational problem-solving strategies. *Education & Information Technologies, 23*(6), 2423–2452.

Prensky, M. (2007). *Digital game-based learning*. USA: Paragon House Ed edition.

Privitera, G. (2017). *Student study guide with IBM SPSS workbook for research methods for the Behavioral Sciences* (2nd). SAGE Publications: Los Angeles.

Qian, Y. & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions in Computers Education, 18*(1).

Rapaport, W. J. (2005). Philosophy of Computer Science: An introductory course. *Teaching Philosophy 28*(4), 319- 341.

Repenning, A. & Ioannidou, A. (2006). What makes end-user development tick? 13 design guidelines. In End-User Development, H. Lieberman, F. Paterno, and V. Wulf, Eds., Human Computer ` Interaction Series (pp. 51–85). New York, USA: Springer.

Repenning, A., Webb, D., & Ioannidou, A. (2010). Scalable game design and the development of a checklist for getting computational thinking into public schools. *In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)* (pp. 265–269). New York: ACM Press

Repenning, A., Webb, K., Koh, E. et al. (2015). Scalable game design: A strategy to bring systemic computer science education to schools through game design and simulation creation. *ACM Transactions on Computing Education (TOCE), 15*(2), 11.

Repenning A., Basawapatna A.R. & Escherle N.A. (2017). Principles of Computational Thinking Tools. In: Rich P., Hodges C. (Eds) Emerging Research, Practice, and Policy on Computational Thinking. Educational Communications and Technology: Issues and Innovations (pp. 291-305). Springer, Cham.

Resnick, M., Martin, F., Sargent, R. & Silverman, B. (1996). Programmable bricks: Toys to think with. *IBM Systems Journal, 35*, 3&4.

Resnick, M. et al. (2009). Scratch: Programming for all. *Communications of the ACM, 52*(11), 60-67.

Rico, M., Martvnez-Muρoz, G., Alaman, X., Camacho, D. & Pulido, E. (2011). Improving the programming experience of high school students by means of virtual worlds. *International Journal of Engineering Education, 27*(1), 52–60.

Ring, B. A., Giordan, J. & Ransbottom, J. S. (2008). Problem solving through programming: motivating the non-programmer. *Journal of Computing Sciences in College, 23*(3), 61-67.

Robertson, J. & Howells, C. (2008). Computer game design: Opportunities for successful learning. *Computers & Education, 50*(2), 559 – 578.

Robertson, J. (2012). The influence of a game-making project on male and female learners' attitudes to computing. *Computer Science Education, 23*(1), 58-83.

Robins, A., Rountree, J. & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(3), 137–172.

Rocco, S. & Plakhotnik, S. (2009). Literature reviews, conceptual frameworks, and theoretical frameworks: Terms, functions, and distinctions. *Human Resource Development Review, 8(*1), 120–130.

Rodriguez, B., Kennicutt, S., Rader, C. & Camp. T. (2017). Assessing computational thinking in CS Unplugged activities. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'17) (pp. 501–506). Seattle, WA, USA.

Román-Gonzalez et al. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the computational thinking test. *Computers in Human Behavior, 72*(3) 678-691.

Rosenbaum, E. (2008). Scratch for Second Life. In S. Veeragoudar Harrell (Chair & Organizer). Virtually there: Emerging designs for STEM teaching and learning in immersive online 3D microworlds. *Symposium in proceedings of the international conference on learning sciences – ICLS 2008*. Utrecht, The Netherlands: ICLS.

Ryan, W. & Siegel. M.A. (2009). Evaluating interactive entertainment using breakdown: Understanding embodied learning in video games. *Proceedings of International Conference: Breaking New Ground:*

Innovation in Games, Play, Practice and Theory (DiGRA' 2009). London: United Kingdom. Retrieved 30 of March 2018 http://www.digra.org/dl/db/09287.38300.pdf.

Rubin, J. & Chisnell, D. (2008). *Handbook of usability testing: How to plan, design, and conduct effective tests* (2nd Ed.). Wiley, Indianapolis.

Schneider, B., Jermann, P., Zufferey, G. & Dillenbourg, P. (July-Sept. 2011). Benefits of a tangible interface for collaborative learning and interaction. *IEEE Transactions on Learning Technologies*, *4*(3), 222-232.

Slavin, R. E., Cheung, A., Groff, C. & Lake, C. (2007). *Effective reading programs for middle and high schools: A best-evidence synthesis.* Baltimore, MD: Johns Hopkins University, Center for Data-Driven Reform in Education.

Salen, K. & Zimmerman, E. (2004). *Rules of play: Game design fundamentals*. USA: MIT press.

Selby, C. C. (2015). Relationships: computational thinking, pedagogy of programming, and Bloom's taxonomy. *In Proceedings of the Workshop in Primary and Secondary Computing Education* (pp. 80-87). United Kingdom, UK: ACM.

Sentance, S. & Csizmadia, A. (2017) Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies*, *22*(2), 469–495.

Singh, K. (2007). *Quantitative social research methods.* Thousand Oaks, CA: Sage Publications.

Singh, S., Ribeiro, T. & Guestrin, C. (2016). *Programs as black-box explanations*. CoRR abs/1611.07579.

Soloway, E. (1986). Learning to program=learning to construct mechanisms and explanations. *Communication of the ACM*, *29*(9), 850-858.

Soloway, E. & Spohrer, J. (1989). Some difficulties of learning to program. *In E. Soloway & J.C. Spohrer, (Ed.). Studying the Novice Programmer* (pp. 283–299). Lawrence Erlbaum Associates, Hillsdale, NJ.

Steiner, C. M., Kickmeier-Rust, M. D. & Albert, D. (2009). Little big difference: gender aspects and gender – based adaptation in educational games. Lecture Notes on Computer Science. Vol. 5670: Learning by playing. Game-based education system design and development (pp. 150–161). Springer.

Steinkuehler, C. & Squire, K. (2014). *Videogames and Learning.* The Cambridge Handbook of the Learning Sciences, 377–394.

Suzuki, H. & Kato, H. (1993). AlgoBlock: A tangible programming language, a tool for collaborative learning. *In the Proceedings of 4th European Logo Conference* (pp. 297-303). Copenhagen, Denmark.

Squire, K.D. (2003). Video games in education. *International Journal of Intelligent Games & Simulation*. *2*(1), 49–62.

Taub, R., Armoni, M. & Ben-Ari, M. (2012). CS unplugged and middle-school students' views, attitudes, and intentions regarding CS. *ACM Transactions on Computing Education*, *12*(2), 1-29.

Teaching guidelines of the Greek Ministry of Education, Research and Religious Affairs. (2017). Retrieved 23 September 2017 https://www.minedu.gov.gr/publications/docs2016/%CE%9F%CE%94%CE%97%CE%93%CE%99%CE%95%CE%A3_%CE%A0%CE%9B%CE%97%CE%A1%CE%9F%CE%A6%CE%9F%CE%A1%CE%99%CE%9A%CE%97_%CE%93%CE%A5%CE%9C%CE%9D%CE%91%CE%A3%CE%99%CE%9F_2016_17.pdf (in Greek).

The Royal Society. (2012). *Shut down or restart? The way forward for computing in UK schools.* UK: The Royal academy of Engineering.

Theodoropoulos, A., Antoniou, A. & Lepouras, G. (2016). How do different cognitive styles affect learning programming? Insights from a game-based approach in Greek schools. *ACM Transactions on Computing Education, 17*(1), Article 3.

Topu, F.B., Reisoğlu, İ., Yılmaz, T.K. et al. (2018). Information retention's relationships with flow, presence and engagement in guided 3D virtual environments. *Education & Information Technologies*. DOI: https://doi.org/10.1007/s10639-017-9683-1

Tucker, A., Deek, F., Jones, J., McCowan, D., Stephenson, C., & Verno, A. (2003). *A model curriculum for K-12 Computer Science: Final Report of the ACM K–12 Task Force Curriculum Committee* (2nd ed.). New York, NY, USA: ACM.

Tullis, T. & Albert, W. (2013). *Measuring the user experience: collecting, analyzing, and presenting usability metrics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Tuomi, P., Multisilta, J., Saarikoski, P., & Suominen, J. (2017). Coding skills as a success factor for a society. Education and Information Technologies, 23, 419–434

Vallance, M. et al. (2009). Designing effective spaces, tasks and metrics for communication in second life within the context of programming LEGO NXT mindstorms™ robots. *International Journal of Virtual and Personal Learning Environments, 1*(1), 20-37.

Vahldick, A., Mendes, A.J. & Marcelino, MJ (2014). The review of games designed to improve introductory computer programming competencies. *In 44th Annual Frontiers in Education Conference* (pp. 781-787). Madrid, Spain.

Völkel, S., Wilkowska, W. & Ziefle, M. (2018). *Gender-specific motivation and expectations toward Computer Science. Proceedings of the 4th Conference on Gender & IT* (GenderIT '18) (pp. 123-134). Heilbronn, Germany: ACM Press.

Yusoff, Z., Kamsin, A. et al. (2018). A survey of educational games as interaction design tools for affective learning: Thematic analysis taxonomy. *Education & Information Technologies, 23*(1), 393-418.

Warburton, S. (2009). Second Life in higher education: Assessing the potential for and the barriers to deploying virtual worlds in learning and teaching. *British Journal of Educational Technology, 40*(3), 414-426.

Webb, N. M., Ender, P. & Lewis, S. (1986). Problem-solving strategies and group processes in small groups learning computer programming. *American Educational Research Journal, 23*(2), 243–261.

Webb, H. & Rosson, M. B. (2013). Using scaffolded examples to teach computational thinking concepts. *In Proceedings of the 44th ACM technical symposium on Computer science education* (pp. 95–100). ACM: USA.

Webb et al. (2017). Computer science in K-12 school curricula of the 2lst century: Why, what and when? *Education & Information Technologies*. DOI: 10.1007/s10639-016-9493-x

Werner, L., Denner, J. & Campe, S. (2015). Children programming games: A strategy for measuring computational learning. *ACM Transactions on Computing Education, 14*(24).

Werner, L., Denner, J., Campe, S. & Kawamoto, D. C. (2012). The fairy performance assessment: Measuring computational thinking in middle school. *In L. S. King & D. R. Musicant (Eds.), Proceedings of the 43rd ACM technical symposium on computer science education* (pp. 215–220). New York, NY: ACM.

Werner, L., Denner, J. & Campe, S. (2014). Using computer game programming to teach computational thinking skills. *In K. Schrier (Ed.), Learning, education and games: Volume 1, curricular and design considerations* (pp. 37–53). Pittsburgh, PA: ETC Press.

Wing. J. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35.

Wing, J. (2011). Research notebook: Computational thinking—What and why? *The Link Magazine, Spring.* Carnegie Mellon University, Pittsburgh.

Witherspoon, E.B., Higashi, R.M., Schunn, C.D., Baehr, E.C. & Shoop, R. (2017). Developing computational thinking practices through a virtual robotics programming curriculum. *ACM Transactions on Computing Education, 18*(1), 20.

Xie, L., Antle, A. N. & Motamedi, N. (2008). Are tangibles more fun? Comparing children's enjoyment and engagement using physical, graphical and tangible user interfaces. In the Proceedings of the 2nd International Conference on Tangible and Embedded Interaction (TEI '08) (pp. 191-198). Bonn, Germany.

Xinogalos, S. & Satratzemi, M. & Malliarakis, C. (2015). Microworlds, games, animations, mobile apps, puzzle editors and more: What is important for an introductory programming environment? *Education & Information Technologies, 22*(1), 145–176.

Zuckerman, O., Arida, S. & Resnick, M. (2005). Extending tangible interfaces for education: Digital montessori-inspired manipulatives. *In the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp.859-868). Portland, Oregon: USA.

Appendices

Appendix A: The questionnaire of the preliminary study

Dear participant,

I would like to welcome you to this study's questionnaire. This questionnaire intends to get grades of view about your personal opinion on computer programming and your attitude to the potential use of a simulation game created in OpenSim with Scratch4SL to support learning of computer programming constructs and skills. It consists of three different parts. These are the following:

1. Demographics and personal information
2. Background in computer programming
3. User learning experience

The current questionnaire is provided in order to be recognized any possible difficulties and constraints in terms of learning how to program. I would like to ask you to read the following items and put one of the following numbers as an answer:

(1) Strongly Disagree, (2) Disagree, (3) Neutral, (4) Agree, (5) Strongly agree.

All of your answers need to be written next to each question that is consistent with your personal experience after this teaching intervention.

Please be assured that this questionnaire is completely confidential, and no attempts will be done to identify you. All data will be kept only for the purpose of this research. For this reason, I will to you a unique nickname and you will be asked to enter this when filling it inside this questionnaire. Your nickname will be asked only if you want to complete this questionnaire after you have played the simulation game in order to extract your personal responses for this study's results. The questionnaire will take 15-20 minutes to complete.

Thanks for your participation. I really appreciate your contribution to this research!

Consent for participation

Based on the researcher's instructions and statements, I would like to declare the following:

- I understand the learning objectives and the aims of this teaching intervention.
- I am aware that my participation is completely voluntary.
- I have the right to withdraw from this study at any time, without any negative impact on me.
- I understand that the data gathered will be recorded only for this study's purposes and that the records will remain confidential.
- I am also aware that the interview will be modified to suit on the writing needs and that I will be able to review it, after the corrections and modifications have been applied, in order to approve it or not.
- I understand that the final outcomes either from my answers in the worksheets or from the interview will not be shared to anyone or published without my personal permission. Such a permission will be requested after the completion and emendation of the interview by the researcher as well as my own review, correction, modification and approval.
- I understand the aim of this study and the content provided to me above, and therefore, I agree to take part.

Nickname:

Age:

Gender:

In the first part of this questionnaire, personal information needs to be provided in the first part of the study. Please provide to me with some details about yourself that will enable me to evaluate your results statistically.

1. I have a personal computer

YES	
NO	

2. I believe that Informatics and specifically programming is

a) Strongly agree	
b) Agree	
c) I don't know	
d) Disagree	
e) Strongly disagree	

3. Frequently, I know and use the following programming environment

a) Scratch	
b) Alice	
c) AgentCubes	
d) Games from the Hour of Code	
e) Other	

4. I have previously utilized and played to learn how to code through online sites like Scratch or in the *"Hour of code"* (www.code.org)

a) Strongly agree	
b) Agree	
c) I don't know	
d) Disagree	
e) Strongly disagree	

In the second part, please provide some details about your learning experience and first perceptions by using OpenSim and Scratch4SL palette.

Learning Effectiveness (LE)	
1. I was able to decompose the main problem into subparts	
2. I was able to understand the use of programming constructs	
3. I started thinking before coding in order to assess the validity of my solution	
4. I understood how to apply a step-by-step solution with programming constructs and commands via Scratch4SL	
5. I had the chance to debug my solution by firstly expressing and then applying it to code blocks	

Learning procedure (LP)	
1. I could effectively communicate my solution plan using specific instructions and programming constructs	
2. I could effectively express a solution into the algorithm	
3. I was able to understand the instructor's feedback either in face-to-face or in-game context	
4. I was able to explain the reasons for using specific programming constructs	
5. I succeeded in applying my proposed solution with design patterns for each subpart of the main problem	

User experience (UX)	
1. OpenSim was easy to use	
2. The S4SL palette was easy to use	
3. OpenSim & S4SL are useful to understand how programming constructs can be used in a real-world problem	
4. I found the use of avatars helpful for exploration in order to gather information about the subparts of this problem	
5. I felt engaged by playing the RVC simulator	
6. The RVC simulator was visually appealing	

Appendix B: The interview questionnaire of the preliminary study

1. Can you briefly describe your experience using OpenSim and Scratch4SL?
2. Have you found helpful for learning to program the use of RVC simulator?
3. Can you refer to any potential advantages and disadvantages when you played this game?
4. Do you want to refer to any technical problems when playing the RVC simulator?

Appendix C: Demographics questionnaire for participants

Dear participant,

I would like to welcome you to this questionnaire to collect some demographics and personal information from you. This questionnaire consists of three different parts including seven questions. These are:

1. Demographics and personal information
2. Background in computer programming
3. The use of interactive environment to learn how to program

Please be assured that this questionnaire is completely confidential, and no attempts will be done to identify you. All data will be kept only for the purpose of this research. For this reason, I will to you a unique nickname and you will be asked to enter this when filling it inside this questionnaire. Your nickname will be asked only if you want to complete this questionnaire after you have played the simulation game in order to extract your personal responses for this study's results. The questionnaire will take 15-20 minutes to complete.

Thanks for your participation. I really appreciate your contribution to this research!

Consent for participation

Based on the researcher's instructions and statements, I would like to declare the following:

- I understand the learning objectives and the aims of this teaching intervention.
- I am aware that my participation is completely voluntary.
- I have the right to withdraw from this study at any time, without any negative impact on me.
- I understand that the data gathered will be recorded only for this study's purposes and that the records will remain confidential.
- I am also aware that the interview will be modified to suit on the writing needs and that I will be able to review it, after the corrections and modifications have been applied, in order to approve it or not.
- I understand that the final outcomes either from my answers in the worksheets or from the interview will not be shared to anyone or published without my personal permission. Such a permission will be requested after the completion and emendation of the interview by the researcher as well as my own review, correction, modification and approval.
- I understand the aim of this study and the content provided to me above, and therefore, I agree to take part.

Nickname:

Age:

Gender:

Personal Information is needed to be provided in the first part of the study. Please provide us with some details about yourself that will enable us to evaluate your results statistically.

1. I have a personal computer

YES	
NO	

2. I mostly use my personal computer for:

Games	
Internet	
Programming	
Exercises	
Other	

3. I believe that Informatics and specifically programming is

a) Strongly agree	
b) Agree	
c) I don't know	
d) Disagree	
e) Strongly disagree	

4. Frequently, I know and use the following programming environment

a) Scratch	
b) Alice	
c) AgentCubes	
d) Games from the Hour of Code	
e) Other	

5. I have previously utilized and played to learn how to code online sites like Scratch or in the "Hour of code" (www.code.org)

a) Strongly agree	
b) Agree	
c) I don't know	
d) Disagree	
e) Strongly disagree	

6. What kind of activities using interactive environments you are mostly involved in learning programming?

a) Learning how to code by game making in creative computing or artistic expression	
b) Learning how to code by making interactive stories	
c) Learning how to code by making interactive games	
d) Learning how to code by playing games	

7. I have previous experience with simulation games.

YES	
NO	

Appendix D: The questionnaire about students' difficulties in programming

Dear participant,

I would like to welcome you to this study's questionnaire. This questionnaire intends to get view your points on computer programming and your attitude to a potential use of a simulation game created in OpenSim (or Scratch) to support learning of computer programming constructs and skills.
This questionnaire consists of four different parts. These are:

A. Difficulties on understanding programming constructs and concepts usage
B. Main reasons for utilizing programming environments
C. Instructional setting and knowledge gained by using programming environments
D. Major difficulties and concerns on learning how to code using programming environments

The questionnaire is provided in the second part of this study in order to be recognized possible difficulties and constraints regarding introductory programming. Please read the following items and put a check mark (✓) next to those that are consistent with your personal experiences during programming courses. All of your answers need to be written next to each question that is consistent with your personal experiences after this teaching intervention.

Please be assured that this questionnaire is completely confidential, and no attempts will be done to identify you. All data will be kept only for the purpose of this research. For this reason, I will to you a unique nickname and you will be asked to enter this when filling it inside this questionnaire. Your nickname will be asked only if you want to complete this questionnaire after you have played the simulation game in order to extract your personal responses for this study's results. The questionnaire will take 15-20 minutes to complete.

Thanks for your participation. I really appreciate your contribution to this research!

Consent for participation

Based on the researcher's instructions and statements, I would like to declare the following:

- I understand the learning objectives and the aims of this teaching intervention.
- I am aware that my participation is completely voluntary.
- I have the right to withdraw from this study at any time, without any negative impact on me.
- I understand that the data gathered will be recorded only for this study's purposes and that the records will remain confidential.
- I am also aware that the interview will be modified to suit on the writing needs and that I will be able to review it, after the corrections and modifications have been applied, in order to approve it or not.
- I understand that the final outcomes either from my answers in the worksheets or from the interview will not be shared to anyone or published without my personal permission. Such a permission will be requested after the completion and emendation of the interview by the researcher as well as my own review, correction, modification and approval.
- I understand the aim of this study, and therefore, I agree to take part.

Nickname:

Age:

Gender:

A. The major degree of difficulty in understanding programming constructs and concepts usage (one choice)

a) Sequence	
b) Iteration	
c) Selection	
d) Nesting programming constructs	
e) Nesting programming constructs with variables and/or numbers	
f) Expression of a proposed solution in the natural language	
g) Describing an algorithm as pseudocode	
h) Applying a proposed strategy from natural language to code	

B. Main reasons for utilizing programming environments

a) Learning how to use fundamental programming constructs (e.g., sequence or selection) in general	
b) Learning how to apply programming constructs in specific problem-solving contexts	
c) Creating by coding interactive games	
d) Playing by coding interactive games	
e) Learning how to code for creating interactive stories	
f) Implementing pre-designed examples based on the school textbook	
g) Learning how to use fundamental programming constructs (e.g., sequence or selection) in general	

C. Instructional setting and knowledge gain by utilizing programming environments

a) By questioning the CS instructor only before starting the new course	
b) By questioning the CS instructor during the course's exercises	
c) Reading the theory and doing exercises on my own	
d) I rarely have questions	

D. Major difficulties and concerns on learning how to code using programming environments

a) Using interactive environments that do not facilitate the development of an algorithm	
b) Understanding why using programming constructs in a problem-solving situation is not well-defined	
c) Unilateral learning on how to compile using only either code syntax or semantics of a programming language	
d) Lack of features that can assist the description and execution of a program to solve a problem	

Appendix E: The pre-and-post questionnaire about the students' determination of skills related to computational thinking

Dear participant,

This questionnaire intends to get grades of view regarding the self-reported determination of your skills related to computational thinking in computer programming before and after this teaching intervention. The current questionnaire consists of four different parts in terms of your personal opinion for skills related to CT regarding the use of an interactive environment that you will use. These are:

A. Problem-solving
B. Critical thinking
C. Algorithmic thinking
D. Creativity

This questionnaire provided in the third part of this study in order to be recognized possible difficulties and constraints regarding introductory programming. Please read the following items and put one number as an answer **(1) never, (2) rarely, (3) sometimes, (4) generally, and (5) always** next to each question that is consistent with your personal experiences before and after this teaching intervention. All of your answers need to be written next to each question that is consistent with your personal experiences after this teaching intervention.

Please be assured that this questionnaire is completely confidential, and no attempts will be done to identify you. All data will be kept only for the purpose of this research. For this reason, I will to you a unique nickname and you will be asked to enter this when filling it inside this questionnaire. Your nickname will be asked only if you want to complete this questionnaire after you have played the simulation game in order to extract your personal responses for this study's results. The questionnaire will take 15-20 minutes to complete.

Thanks for your participation. I really appreciate your contribution to this research!

Consent for participation

Based on the researcher's instructions and statements, I would like to declare the following:

- I understand the learning objectives and the aims of this teaching intervention.
- I am aware that my participation is completely voluntary.
- I have the right to withdraw from this study at any time, without any negative impact on me.
- I understand that the data gathered will be recorded only for this study's purposes and that the records will remain confidential.
- I am also aware that the interview will be modified to suit on the writing needs and that I will be able to review it, after the corrections and modifications have been applied, in order to approve it or not.
- I understand that the final outcomes either from my answers in the worksheets or from the interview will not be shared to anyone or published without my personal permission. Such a permission will be requested after the completion and emendation of the interview by the researcher as well as my own review, correction, modification and approval.
- I understand the aim of this study, and therefore, I agree to take part.

Nickname:

Age:

Gender:

A. Problem-solving

a) I have problems with the demonstration of a proposed solution for a problem keeping it in my mind and expressing it adequately	
b) I struggle to apply a proposed solution the way I have planned it respectively and gradually.	
c) I cannot describe so many options while thinking of alternative ways to propose different solutions regarding a problem.	
d) I have problems to use where and how correctly the variables such as X and Y for proposing a solution to a problem	

B. Critical thinking

a) I like solving problems which are related to my previous knowledge gained from previous ones that I have solved	
b) I prepare regular plans regarding a solution for more complex problems	
c) I try being able to think with great precision in more challenging things.	
d) I try to think before practice systematically while comparing the options at my hand and while reaching a decision.	

C. Algorithmic thinking

a) I can immediately establish the equity that will give the solution to a problem	
b) I think that it is better to be provided instructions with mathematical symbols and concepts	
c) I believe that I can easily catch the relation between the figures and visual elements	
d) I can express alternatively a solution to a specific real-life, even if mathematical definitions are needed to become more accurate	

D. Creativity

a) I believe that by giving appropriate time and effort most of the problems can be solved	
b) I can apply my personal plan while making it solve a problem.	
c) I trust my intuitions and feelings on how wrong or correct they are when approaching a solution to a problem	

Appendix F: The interview questionnaire of the quasi-experimental study

1. Can you provide specific reasons on why the proposed simulation game helped you (or not) to express and apply your solution plans into code?
2. Which of the main in-game features helped you most to understand the simulated problem-solving context?
3. Do you think that the proposed SG really facilitated you to think before applying your strategy in a more creative way into code? Can you please justify your answer?
4. How do you think the use of basic gameplay features (e.g., the code palette or the graphical user interface features and elements) helped you in favor of expressing and applying a solution from an algorithm into code?

Appendix G: The worksheet about the learning activities using Scratch

THINKING ABOUT THE CONTROL MOVEMENT OF A ROBOT VACUUM CLEANER USING PROGRAMMING CONSTRUCTS

Proposed time duration: 4 teaching hours (40 min. for each session)
Requirements: Hardcopies to write pseudocode and instruction cards to write the encoded solution using Scratch
Technological means: Scratch

Learning goals
The learning goals can be achieved by familiarizing students with the simulation game and its potential contribution to facilitate the development and implementation of computational problem-solving strategies in simulated real-world contexts. In particular, students are expected to achieve the following:

- To explore how a robot vacuum cleaner can be moved into a big house, taking into account the spatial layout of each room that displays several simulated problem-solving contexts between the furniture and other house objects.
- To propose a solution with logical reasoning by expressing specific steps of a solution based on a computational problem-solving strategy and exploit different forms of constructs and commands such as REPEAT, "From ... until ..." or "Until...repeat", SELECTION ("If ... then" or "If" then "otherwise") or the SEQUENCE of in order to construct design patterns as a solution to each in-game task using the visual palette for coding tasks.
- To explain the appropriateness of using specific programming constructs in order to express your solution plans as design patterns that integrated as behaviors into the robot to predict its control movement without causing damages in the house.

Helpful tips

- By using specific programming constructs, a computer can execute the given instructions and actions (calculations, screen displays, etc.) precisely and faster than a human.
- Regarding the rotation and move of the robot around the home, please do not forget the basic concepts that you have learned in Geometry. In this case, it is imperative to remind you that 90° (degrees) is the right angle in a square with each side having a length and a width of 5m and 45° angle is equal half of the right angle. All in all, if you are thinking about how the robot needs to be moved into a square-shaped space; thus, turning 360º degrees in 4 steps or otherwise can turn 360°/4=90°.
- For the correct execution of the robot's control movements/instructions, there are notecards of Scratch and hardcopies/worksheets that can be used for proposing and describing through a text form in natural language your pseudocodes for each stage. Consequently, using a code block palette from Scratch to integrate behavior inside the robot (OpenSim) and assess the correctness of your solution plan (cleaning pathway) into code.

Basic guidelines

The research aim of this teaching intervention is the exploitation of a simulation game following an instructive guided approach with step-by-step programming exercises and the investigation of its' impact on students' learning outcomes depending on computational problem-solving strategies that can be applied intro code via Scratch. Having the role of embedded software engineer, you should assist an old woman with special needs, who moves only with her wheelchair and struggles to clean all rooms of her house. In

a gameplay context, you need to elaborate a solution aimed at creating algorithms with logically and precise instructions and finally to propose solution plans as design patterns into code. Firstly, you need to navigate, determine the robot's movement positions and describe the best cleaning path that an autonomous robot can follow in sufficient time. Thereupon, your solutions can be implemented by integrating behavior using Scratch in order to give specific directions to a robot vacuum cleaner that should move and clean 3 rooms that are differentiated in spatial geometry layout, in terms of division among house furniture and objects. Please try to calculate arithmetically distances without causing hits or damages.

According to the above, house furniture and objects in square floors are seen as evocative spatial metaphors of basic geometric shapes (e.g., triangle, square, and hexagon) to think and practice computationally with an abstract conceptualization approach alongside with pathfinding in a logical problem can be followed. To prevent hitting a table, you need to determine arithmetic computation between chairs and table distance (e.g., each side's square floor has side 5m) or-/and calculate degrees of turning correctly (e.g., 90^{o} for square or 45^{o} for equilateral triangle) to traverse the robot a specific cleaning pathway down from the table, without hitting the table lamp. This process is becoming more compelling as you need to apply a computational strategy via Scratch palette beneficial to be presented the shortest path between the present location and the goal location of the robot. Last but not least, it is also important to notice that the distance from the robot should be no more than 140 steps for movement (aligned to 5m), because then the signal will be lost, and the robot will not be controlled causing damage until the battery shutting down automatically. Since a specific role is assigned to you, a number of steps of your strategy need to be followed:

a) explore each room separately to identify any drawbacks among house objects and furniture,

b) plan specific movements to pass all checkpoints the vacuum robot for optimum performance in order to propose the shortest cleaning path in reasonable time, and locate any further grades that should be avoided so as to clean all dusty dots over the floor, without hit any object or furniture, and finally

c) program the shortest cleaning route that can be proposed for each room individually in order not to turn off the robot due to battery consumption after one hour. Whenever the robot is programmed to pass and clean all dusty dots (gray signs) off the floor, for rewarding, it gains energy, giving grades to its battery life. If gathering the smallest possible number of code blocks for cleaning each room based on resilient planning, execution time and fewer hits on the house furniture or objects, then such a player is declared as the winner.

The rooms

All in all, 6 rooms designed with learning tasks lasted only 40 minutes, but one is going to be used for your personal training. Therefore, you are free to propose different solutions based on your design patterns as there was not a pre-defined one. You have the chance to choose only 4 rooms, with 1 to be chosen from each stage. Only the chosen 3 rooms count to your final grades. The bedroom or the drawing room (Stage 1) are developed for introductory activities to learn how to use some tools and another one need to be excluded (see Figure 1).

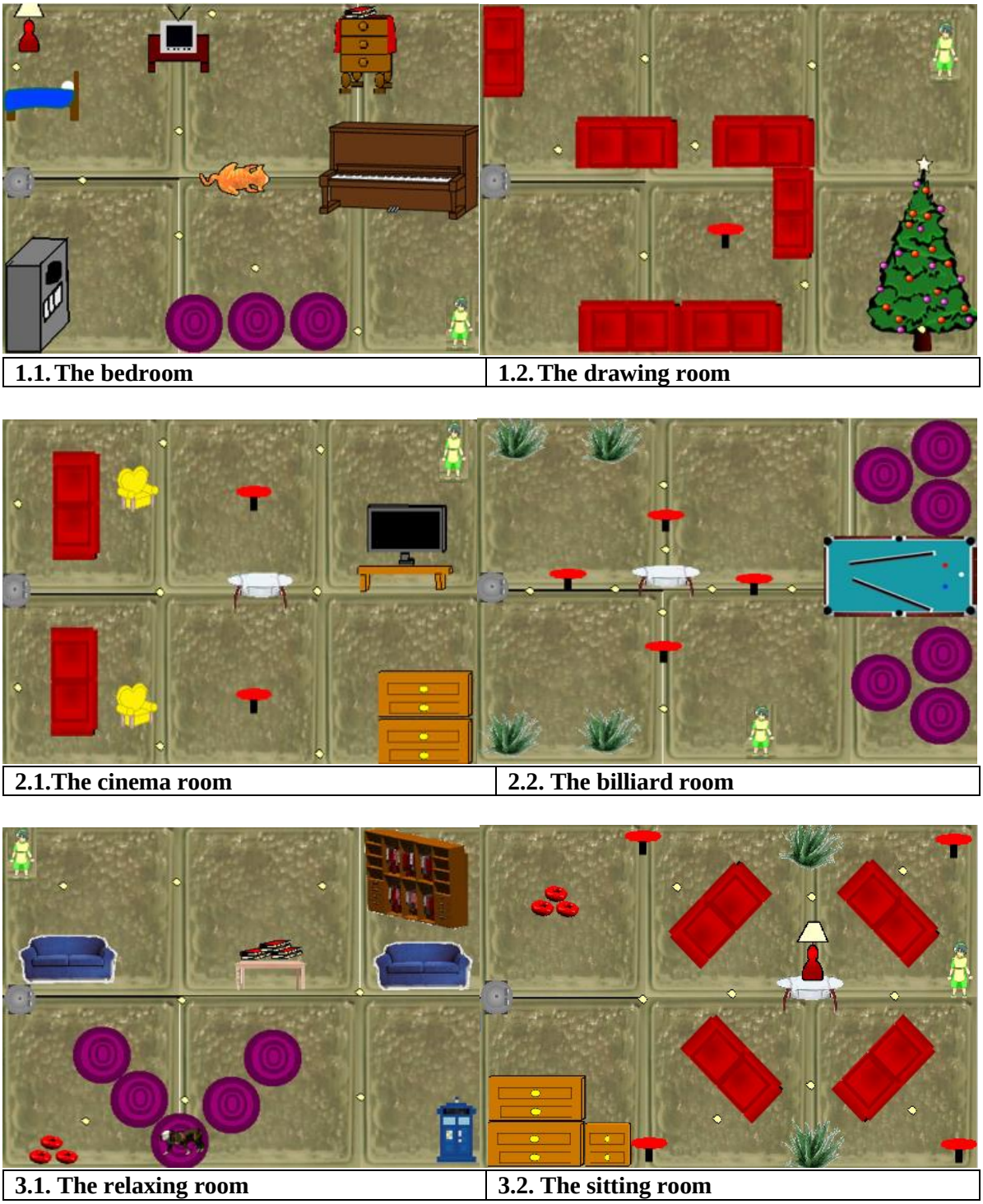
A presupposition is to use the same programming method and constructs (i.e., simple or nested iteration, sequence or selection) can be used at first stages including the bedroom (1.1) and the drawing room (1.2) to propose a solution for the other 3 chosen rooms again only once more. This means that for the other two, you should propose a combination of programming methods or other programming constructs nested with numbers and/or variables.

Except for the above two rooms, the rest four in-game rooms have different levels of difficulty. For example, the second stage includes the billiard room (2.1) and cinema room (2.2) have a medium level of difficulty due to the fewer objects and house furniture, in which players can use either one programming method.

The relaxing room (3.1) and sitting room (3.2) are included in the third stage. Both have a higher level of difficulty as at least optically house furniture and objects created to each one differentiates on the geometric shapes and thence more programming methods need to be combined. When you decide which of the 3 rooms from the three stages want to play, you have a chance to use one different method that can

be combined with the proposed programming method in order to gain higher grades, e.g., a combination of selection with the sequence.

In Figure 1, below 6 rooms are appeared. Please choose 3 of them so as to provide for each one a proposed solution to all subparts of this simulated real-life problem.



1.1. The bedroom	1.2. The drawing room
2.1.The cinema room	2.2. The billiard room
3.1. The relaxing room	3.2. The sitting room

Figure 1: The in-game stages created by Scratch

When all the above considerations can be addressed in a specific timeline, a thorough exploration of each room is appropriate to map out the possible grades from which the robot could pass to clean. Secondly, it is important for taking good grades not only to fully describe the robot's movement by controlling its steps in an algorithmic way but also to program it, utilizing programming constructs. At the end of the description and expression of the proposed solution in the worksheet (see the table below), I will be informed about your progress, and after that, I will allow you to present your described solution plan into code. Thus, any solution that you will give should not only be a description of commands and instructions by utilizing each programming construct that as an engineer should present in small sentences using natural language, but it should also be performed by using the Scratch palette in the code to prove the correctness and the degree of your applications. Helping you to apply the commands and instructions please use Scratch's palette, as Figure 2 depicts.

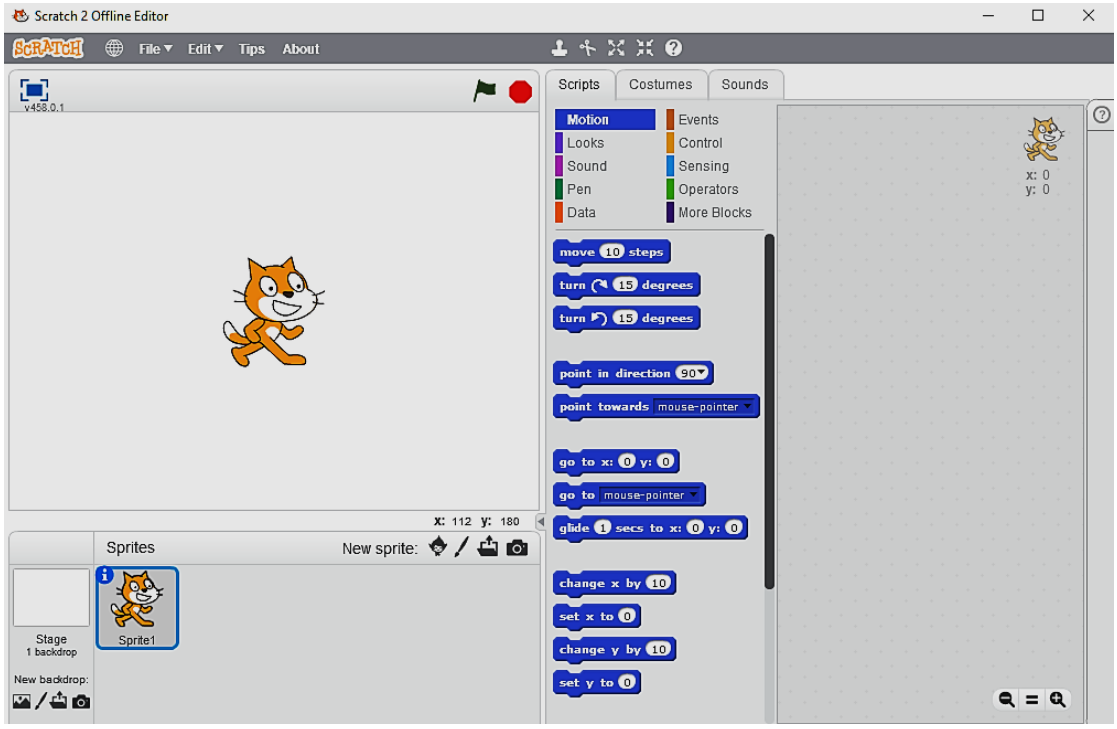


Figure 2: The programming constructs and commands in the palette of Scratch

Main activity

By following the structure of the scenario that was previously described, you must try to program the robot in each of the 3 rooms so that it can be cleaned in a specific period of time, without hitting objects or furniture that can change its direction. To achieve this goal, you need to write correctly programming constructs and commands to determine the movements and rotation of the robot in each room that has a specific layout geometry. A proposed solution for each of the 3 chosen rooms needs to be written in the table below. In this table, you must write both in the form of natural language short sentences that entail Goals, Rules/instructions, and Anticipated outcomes, pseudocode with simple step-by-step description. Lastly, you need to determine the extent to which the algorithm that you have proposed in natural language can be applied into code via Scratch.

Question: Can you describe which may be a preferable choice to demonstrate a cleaning pathway that a robot vacuum cleaner needs to follow for each of your 3 chosen rooms as depicted in Figure 1?

Description of a proposed solution in natural language with short sentences	Pseudocode

Important note: After the description of a proposed solution to the above Table, please use Scratch for coding and save it as a .sb file to your personal computer to gather data at the end of this experiment.

Thanks for your participation!

Appendix H: The worksheet about the learning activities using OpenSim with Scratch4SL

THINKING ABOUT THE CONTROL MOVEMENT OF A ROBOT VACUUM CLEANER USING PROGRAMMING CONSTRUCTS

Proposed time duration: 4 teaching hours (40 min. for each session)
Requirements: Hardcopies to write pseudocode and instruction cards to write the encoded solution using Scratch4SL
Technological means: OpenSim + Scratch4SL

Learning goals
The learning goals can be achieved by familiarizing students with the simulation game and its potential contribution to facilitate the development and implementation of computational problem-solving strategies in simulated real-world contexts. In particular, students are expected to achieve the following:

- To explore how a robot vacuum cleaner can be moved into a big house, taking into account the spatial layout of each room that displays several simulated problem-solving contexts between the furniture and other house objects.
- To propose a solution with logical reasoning by expressing specific steps of a solution based on a computational problem-solving strategy and exploit different forms of constructs and commands such as REPEAT, "From ... until ..." or "Until...repeat", SELECTION ("If ... then" or "If" then "otherwise") or the SEQUENCE of in order to construct design patterns as a solution to each in-game task using the visual palette for coding tasks.
- To explain the appropriateness of using specific programming constructs in order to express your solution plans as design patterns that integrated as behaviors into the robot to predict its control movement without causing damages in the house.

Helpful tips

- By using specific programming constructs, a computer can execute the given instructions and actions (calculations, screen displays, etc.) precisely and faster than a human.
- Regarding the rotation and move of the robot around the home, please do not forget the basic concepts that you have learned in Geometry. In this case, it is imperative to remind you that 90° (degrees) is the right angle in a square with each side having a length and a width of 5m and 45° angle is equal half of the right angle. All in all, if you are thinking about how the robot needs to be moved into a square-shaped space; thus, turning 360º degrees in 4 steps or otherwise can turn 360°/4=90°.
- For the correct execution of the robot's control movements/instructions, there are notecards of Scratch and hardcopies/worksheets that can be used for proposing and describing through a text form in natural language your pseudocodes for each stage. Consequently, using a code block palette from Scratch to integrate behavior inside the robot (OpenSim) and assess the correctness of your solution plan (cleaning pathway) into code.

Basic guidelines

The research aim of this teaching intervention is the exploitation of a simulation game following an instructive guided approach with step-by-step programming exercises and the investigation of its' impact on students' learning outcomes depending on computational problem-solving strategies that can be applied

intro code via Scratch. Having the role of embedded software engineer, you should assist an old woman with special needs, who moves only with her wheelchair and struggles to clean all rooms of her house. In a gameplay context, you need to elaborate a solution aimed at creating algorithms with logically and precise instructions and finally to propose solution plans as design patterns into code. Firstly, you need to navigate, determine the robot's movement positions and describe the best cleaning path that an autonomous robot can follow in sufficient time. Thereupon, your solutions can be implemented by integrating behavior using Scratch in order to give specific directions to a robot vacuum cleaner that should move and clean 3 rooms that are differentiated in spatial geometry layout, in terms of division among house furniture and objects. Please try to calculate arithmetically distances without causing hits or damages.

According to the above, house furniture and objects in square floors are seen as evocative spatial metaphors of basic geometric shapes (e.g., triangle, square, and hexagon) to think and practice computationally with an abstract conceptualization approach alongside with pathfinding in a logical problem can be followed. To prevent hitting a table, you need to determine arithmetic computation between chairs and table distance (e.g., each side's square floor has side 5m) or-/and calculate degrees of turning correctly (e.g., 90^o for square or 45^o for equilateral triangle) to traverse the robot a specific cleaning pathway down from the table, without hitting the table lamp. This process is becoming more compelling as you need to apply a computational strategy via Scratch palette beneficial to be presented the shortest path between the present location and the goal location of the robot. Last but not least, it is also important to notice that the distance from the robot should be no more than 140 steps for movement (aligned to 5m), because then the signal will be lost, and the robot will not be controlled causing damage until the battery shutting down automatically. Since a specific role is assigned to you, a number of steps of your strategy need to be followed:

a) explore each room separately to identify any drawbacks among house objects and furniture,

b) plan specific movements to pass all checkpoints the vacuum robot for optimum performance in order to propose the shortest cleaning path in reasonable time, and locate any further grades that should be avoided so as to clean all dusty dots over the floor, without hit any object or furniture, and finally

c) program the shortest cleaning route that can be proposed for each room individually in order not to turn off the robot due to battery consumption after one hour. Whenever the robot is programmed to pass and clean all dusty dots (gray signs) off the floor, for rewarding, it gains energy, giving grades to its battery life. If gathering the smallest possible number of code blocks for cleaning each room based on resilient planning, execution time and fewer hits on the house furniture or objects, then such a player is declared as the winner.

The rooms

All in all, 6 rooms designed with learning tasks lasted only 40 minutes, but one is going to be used for your personal training. Therefore, you are free to propose different solutions based on your design patterns as there was not a pre-defined one. You have the chance to choose only 4 rooms, with 1 to be chosen from each stage. Only the chosen 3 rooms count to your final grades. The bedroom or the drawing room (Stage 1) are developed for introductory activities to learn how to use some tools and another one need to be excluded (see Figure 1).

A presupposition is to use the same programming method and constructs (i.e., simple or nested iteration, sequence or selection) can be used at first stages including the bedroom (1.1) and the drawing room (1.2) to propose a solution for the other 3 chosen rooms again only once more. This means that for the other two, you should propose a combination of programming methods or other programming constructs nested with numbers and/or variables.

Except for the above two rooms, the rest four in-game rooms have different levels of difficulty. For example, the second stage includes the billiard room (2.1) and cinema room (2.2) have a medium level of difficulty due to the fewer objects and house furniture, in which players can use either one programming method.

The relaxing room (3.1) and sitting room (3.2) are included in the third stage. Both have a higher level of difficulty as at least optically house furniture and objects created to each one differentiates on the

geometric shapes and thence more programming methods need to be combined. When you decide which of the 3 rooms from the three stages want to play, you have a chance to use one different method that can be combined with the proposed programming method in order to gain higher grades, e.g., a combination of selection with the sequence.

In Figure 1, below 6 rooms are appeared. Please choose 3 of them so as to provide for each one a proposed solution to all subparts of this simulated real-life problem.

1.1. The bedroom

1.2. The drawing room

2.1. The relaxing room

2.2. The cinema room

3.1. The sitting room

3.2. The relaxing room

Figure 1: The in-game stages created in OpenSim

When all the above considerations can be addressed in a specific timeline, a thorough exploration of each room is appropriate to map out the possible grades from which the robot could pass to clean. Secondly, it is important for taking good grades not only to fully describe the robot's movement by controlling its steps in an algorithmic way but also to program it, utilizing programming constructs. At the end of the description and expression of the proposed solution in the worksheet (see the table below), I will be informed about your progress, and after that, I will allow you to present your described solution plan into code. Thus, any solution that you will give should not only be a description of commands and instructions by utilizing each programming construct that as an engineer should present in small sentences using natural language, but it should also be performed by using the Scratch palette in the code to prove the correctness and the degree of your applications. Helping to find the commands and instructions using from Scratch, the following Figure 2 was created to understand the alignment of code commands in the S4SL palette.

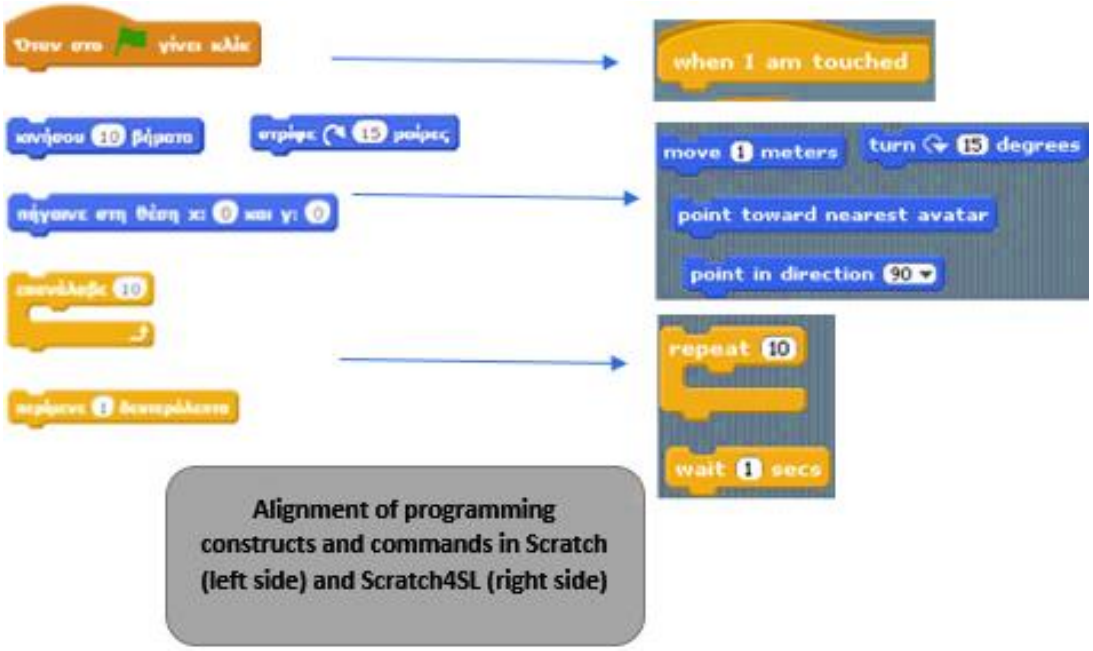


Figure 2: The programming constructs and commands in the palette of Scratch and Scratch4SL

Main activity

By following the structure of the scenario that was previously described, you must try to program the robot in each of the 3 rooms so that it can be cleaned in a specific period of time, without hitting objects or furniture that can change its direction. To achieve this goal, you need to write correctly programming constructs and commands to determine the movements and rotation of the robot in each room that has a specific layout geometry. A proposed solution for each of the 3 chosen rooms needs to be written in the table below. In this table, you must write both in the form of natural language short sentences that entail Goals, Rules/instructions, and Anticipated outcomes, pseudocode with simple step-by-step description. Lastly, you need to determine the extent to which the algorithm that you have proposed in natural language can be applied into code via Scratch4SL.

Question: Can you describe which may be a preferable choice to demonstrate a cleaning pathway that a robot vacuum cleaner needs to follow for each of your 3 chosen rooms as depicted in Figure 1?

Description of a proposed solution in natural language with short sentences	Pseudocode

Important note: After the description of a proposed solution to the above Table, please use Scratch4SL for coding and save it as a .sb2 file to your personal computer to gather data at the end of this experiment.

Thanks for your participation!