Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2009, Article ID 470590, 12 pages
doi:10.1155/2009/470590

Research Article

Towards a Serious Game to Help Students Learn Computer Programming

Mathieu Muratet,[1] Patrice Torguet,[1] Jean-Pierre Jessel,[1] and Fabienne Viallet[2]

[1] *VORTEX Group, IRIT, Paul Sabatier University, 118 Route de Narbonne, 31062 Toulouse Cedex 9, France*
[2] *DiDiST CREFI-T, Paul Sabatier University, 118 Route de Narbonne, 31062 Toulouse Cedex 9, France*

Correspondence should be addressed to Mathieu Muratet, muratet@irit.fr

Received 30 August 2008; Revised 17 December 2008; Accepted 24 February 2009

Recommended by Xiaopeng Zhang

Video games are part of our culture like TV, movies, and books. We believe that this kind of software can be used to increase students' interest in computer science. Video games with other goals than entertainment, serious games, are present, today, in several fields such as education, government, health, defence, industry, civil security, and science. This paper presents a study around a serious game dedicated to strengthening programming skills. Real-Time Strategy, which is a popular game genre, seems to be the most suitable kind of game to support such a serious game. From programming teaching features to video game characteristics, we define a teaching organisation to experiment if a serious game can be adapted to learn programming.

1. Introduction

Since the first boom of video games in the 80s, the gaming industry has held an important place in the world market. According to the Entertainment Software Association figures (http://www.theesa.com/facts/pdfs/ESA_EF_2008.pdf) accessed 26 August 2008, in 2007 the market of U.S. computer and video games amounts to $9.5 billion. This is almost equal to the U.S. movie market (http://www.the-numbers.com/market/2007.php) accessed 26 August 2008 ($9.6 billion in 2007). Students currently in university were born with video games, which are as much a part of their culture as TV, movies, or books.

However, to progress in video games, the player must at the same time play and learn. Serious games use this feature to interest players in specific topics, teach some specific educational content, or train workers to specific tasks. The idea is that entertainment can lead to learning if some specific constraints are respected.

On the other hand, all over the world, students are becoming less interested in science. In computer science, for example, according to Crenshaw et al. [1] and Kelleher [2], the number of students is shrinking. Moreover, "*colleges and universities routinely report that 50% or more of those students who initially choose computer science study soon decide to abandon it*" [3, page 39]. Our university experiences the same phenomenon with a decrease of 16.6% over the last four years in students studying computer science.

Therefore, in the computer science education research field, there is an important area directed to the recruitment and retention of students [4]. A promising way explored by this specific research is using games to teach and learn programming [5]. It allows students to better learn in a familiar and playful environment. Moreover, it promotes collaborative learning and spurs student to learn.

We propose to study if serious games, which can be collaborative learning games, could be of value in order to teach programming and to attract and keep computer science students. The question is: Is it interesting to use a serious game for teaching programming?

To achieve this goal, we propose the methodology of design experiments [6]: "*prototypically, design experiments entail both "engineering" particular forms of learning and systematically studying those forms of learning within the context defined by the means of supporting them. This designed context is subject to test and revision, and the successive iterations that result play a role similar to that of systematic variation in experiment.*" The intent of this methodology

in educational research is to investigate the possibilities for educational improvement by bringing about new forms of learning in order to study them. Because designs are typically test-beds for innovation, the nature of the methodology is highly interventionist, involving a research team, one or more teachers, at least one student, and eventually school administrators. Design contexts are conceptualized as interacting systems and are implemented with a hypothesized learning process and the means of supporting it. Although design experiments are conducted in a limited number of settings, they aim to develop a relatively humble theory that targets a domain specific learning process. To prepare a design experiment, the research team has to define a theoretical intent and specify disciplinary ideas and forms of teaching that constitute the prospective goals or endpoints to student learning. The challenge is to formulate a design that embodies testable conjectures about both significant shifts in student learning and the specific means of supporting those shifts. In our experiment, the theory we attempt to develop is the process of learning programming through serious games. In this paper, we discuss how to build a design context that will allow us to construct several conjectures to test our theory about an original form of programming teaching.

In the rest of this paper, we define briefly what a serious game is and what its learning aims are. After presenting programming teaching features and associated environment, we analyse some of them in reference to learning objectives and serious games features. Because there is currently no serious game dedicated to this field and suitable to design experiments, the rest of the paper presents the serious game we built. Section 4 presents how we chose the video game that supports our serious game. Section 5 details the implementation of our serious game. Section 6 describes how learning objectives are mapped into the game from the student, teacher, and knowledge points of view. Section 7 explains how we will conduct our first experiment.

2. Serious Games

2.1. Definitions. For Zyda [7], a serious game is "*a mental contest, played with a computer in accordance with specific rules, that uses entertainment to further government or corporate training, education, health, public policy, and strategic communication objectives*." Thus, any video game built to differ from pure entertainment can be considered as a serious game. Serious games represent, therefore, a wide range of digital games. Blackman [8] gives a synopsis of the gaming industry and its applications. Sophisticated video game graphic engines are nowadays used for nongame applications because they offer real-time rendering and physical models. Applications such as simulators can use such video game technologies. Serious games are not restricted to video games; they can also be based on simulators. Figure 1 illustrates the relationship between video games, simulators, and serious games.

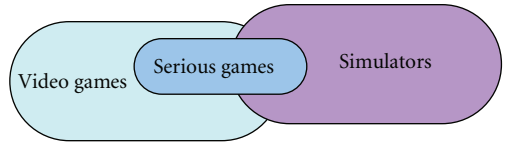

Figure 1: Relation between video games, simulators, and serious games.

2.2. Example of Serious Games. To highlight the relationship between the target public and serious game objectives we present three examples of serious games following different aims: "Darfur is Dying" (http://www.darfurisdying.com/) accessed 30 November 2008, "Tactical Language & Culture" (http://www.tacticallanguage.com/) accessed 30 November 2008, and "America's Army" (http://www.americasarmy.com/) accessed 30 November 2008; "Darfur is Dying" is a game developed in partnership with the Reebok Human Rights Foundation and the International Crisis Group. The purpose of this game is to increase public awareness of the crisis in Darfur. The player controls a Darfurian who forages for water and develops his/her camp. Because the objective was to reach a maximum of people, "Darfur is Dying" is a minigame based on a platform game genre easy to play even for nongamers. It is free and accessible by everyone.

"Tactical Language & Culture" is a game started in 2003 as a research project at the University of Southern California's Information Sciences Institute under funding from the Defence Advanced Research Projects Agency (DARPA). Its purpose is to teach foreign languages and cultural knowledge needed to conduct tasks effectively and safely during both daily life and military missions. Currently, it offers courses in Iraqi Arabic language and culture (*Tactical Iraqi*), Pashto language and culture for the Pashtun regions of Afghanistan (*Tactical Pashto*), and French as spoken in the countries of Sahel Africa (*Tactical French*). "Tactical Language & Culture" is a complex game targeted for servicemen. It is based on a role-playing game genre to enable an immersive communication with virtual avatars in the game. It is played in interaction with a virtual tutor who evaluates the learner's speech and gives feedback on errors.

"America's Army" is a game launched in July 2002 designed by the Modelling, Virtual Environments, and Simulation (MOVES) Institute at the Naval Postgraduate School in Monterey, Calif, USA. It was initially built as a recruiting tool for the United States of America's army. However it became the first really successful serious game and is currently one of the ten most popular PC action games played online. "America's Army" is a complex game based on a shooter game genre to immerse the player in action. It is a free multiplayer game requiring a team effort.

These serious games use entertainment to pursue different learning objectives: "Darfur is dying" tries to raise public awareness; "Tactical Language & Culture" aims to learn foreign languages and cultures; "America's Army" tries to attract young people to join the US Army.

2.3. Video Games. Serious games are mainly based on video games that define their usability. According to the aims

of the serious game, the video games characteristics of game genre, game mode, and game complexity define the target audience. The *game genre* is used to classify video games. Some examples are "shooters" like "Doom" series or "Counter Strike" (players combat several characters with projectile weapons, such as guns or missiles), "sports" like "Pro Evolution Soccer" or "Virtual Tennis" series (emulates the playing of traditional physical sports), "strategy" like "Age of Empires" or "Civilization" series (players control an army and command it to achieve objectives), or "role-playing" like "Final Fantasy" series or "World of Warcraft" (players are cast in the role of one or more "adventurers" to progress through a predetermined storyline).

We define the *game mode* as the networked nature of the game. Single player refers to a game, where only one player can interact with the game. The player plays against preprogrammed challenges and/or opponents controlled by Artificial Intelligences (AI). A multiplayer mode allows players to interact with each other. Partnerships, cooperation, competitions, or rivalries emerge to provide a form of social communication.

The *Game complexity* refers as in [9] to the duration of the game. Minigames or trivial games take less than one hour to complete, treat only one subject, puzzle, or game play type in a small way. Complex games take more than ten hours to complete, provide a sophisticated mixture of difficult challenges that typically intertwine and support each other. Main features of complex games are levelling-up, adaptability, clear and worthwhile goals, interaction with other players, and shared experiences.

The choice of a game genre, a game mode, and the complexity of the video game is a crucial point to be in agreement with the target audience and the serious game objectives.

2.4. Serious Games and Learning. The critical point of serious games is the relationship between the game and the educational content: Zyda [7] wrote that "*Pedagogy must be subordinate to story—the entertainment component comes first*." A hypothesis is that if the game is attractive, fun, and stimulating, and encourages the player to progress, then she/he will automatically learn skills from the game and will absorb a lot of information. What about serious games devoted to teaching how to program? Is there a need for such tools?

3. Programming Fundamentals Learning

In order to determine if serious games can be useful for teaching programming, we present educational features, bring to light problems encountered by students, and expose different solutions proposed by teachers. Among the developed tools devoted to this field, we chose to analyse some of them in reference to learning objectives and serious games features.

3.1. Features. An ACM/IEEE report [3] provides an overview of the different kinds of undergraduate degree curricula in computing. This report divides computing in five major disciplines: Computer Engineering (CE), Computer Science (CS), Information Systems (IS), Information Technology (IT), and Software Engineering (SE). It shows that the most important requirement for all these disciplines is the "*Programming Fundamentals*" topic, and the ability to "*Do small-scale programming*" is the most expected performance capability. The requirements of "Programming fundamentals" (PFs) are detailed in another report [10], which defines precisely the features of programming courses and outlines a set of recommendations for undergraduate curricula. It is divided in 5 core units: fundamental programming constructs (PF1), algorithms and problem solving (PF2), fundamental data structures (PF3), recursion (PF4), and event-driven programming (PF5). There are no recommendations on the programming language used for teaching. The main topics taught in PF are variables, types, expressions, assignments, simple Input/Output, conditional and iterative control structures, functions, parameter passing, arrays, and records.

3.2. Problems. Programming fundamentals are hard skills to learn, especially for novices, for several reasons. First, students encounter some unexpected epistemological obstacles, like learning looping constructs [11, 12], conditionals, or assembling programs out of base components: "*Data structure and algorithms* [...] *are often difficult issues, since capturing the dynamic nature of abstract algorithms is not a straightforward task*" [13]. Thus, "*the lack of student programming skill even after a year of undergraduate studies in computer science was noted and measured in the early 80's [14] and again in this decade [15]*" [4, page 127].

Second, the computer environment they use daily, to play or chat for example, is very different from the one they use for learning and they do not immediately see the connection between the two universes: "*People studying pedagogical techniques agree that students who are new to computer science typically find the field full of theoretical, technical, or even tedious concepts*" [16].

Third, learning how to program assumes lectures, classes, and practice sessions. To be able to program, students need to know programming skills and concepts, but to learn those skills and concepts they have to practice programming. Dealing with this paradox, Greitzer et al. [9] explain in particular that "*an effective approach is to encourage learners to work immediately on meaningful, realistic tasks*."

3.3. Some Solutions. To the question as to what makes programming easier for novices and what helps students to acquire programming fundamentals, a great many answers are proposed in the literature. For example, Stevenson and Wagner [17] analyse assignments from textbooks and historical usage to look for student's problems and propose a set of characteristics and assignments that should be a "good programming assignment" in CS1: (1) be based on a real-world problem; (2) allow the students to generate a realistic solution to that problem; (3) allow them to focus on current topic(s) from class within the context of larger programs;

(4) be challenging; (5) be interesting; (6) make use of one or more existing application programming interfaces (APIs); (7) have multiple levels of challenge and achievement, thus supporting possible refactoring; (8) allow some creativity and innovation. To implement this assignment, the authors propose a CS1 project based on a web crawler and a spam evaluator.

Another trend, studying how students relate to computer science and why they quit, has shown that a lack of meaning and relevance is key issues that create distaste for the discipline [4, page 150]. One answer is to develop specific interesting and relevant computational artefacts that have meaning for students. In this direction, three approaches exist: (1) building novice-programming environments, using (2) programming contests or (3) video games.

Many novice-programming environments have been built. Most of them use block-based graphical languages. This programming metaphor allows students to forget syntax and directly experiment with programming. Here are a few examples.

(i) StarLogo The Next Generation [18] uses computer game design as the motivation and theme to introduce programming to middle or high school students. It is a modelling and simulation software. Students and teachers use agents-based programming and 3D graphics to create and understand simulations and complex systems.

(ii) Scratch [19] is a programming language that makes it easy to create interactive stories, animations, games, music, and art and share them through the web. It is designed to help young people (ages 8 and up) develop learning skills from several disciplines. As they create Scratch projects, young people learn important mathematical and computational ideas, while also gaining a deeper understanding of the process of design.

(iii) Alice2 [20] is a programming environment designed for teaching programming while building 3D virtual worlds. This drag and drop programming system allows users to experiment with the logic and programming structures taught in introductory programming classes without making syntax errors. It allows users to experiment with conditionals, loops, variables, parameters, and procedures.

(iv) Cleogo [21] is a groupware environment that allows several users to simultaneously develop programs through any mixture of three alternative programming metaphors: a direct manipulation language for programming by demonstration an iconic language and a standard text-based language. Cleogo is motivated by the pedagogical values of peer learning and of collaborative problem solving, at home and at work.

The second approach consists in using competition to motivate students. Robocode (http://robocode.sourceforge.net/) accessed 17 April 2007, is a Java programming game, where the goal is to develop a robot battle tank to battle against other tanks programmed by other players. It is designed to help people learn Java programming. The robot battles are running in real-time and on-screen. It is suitable to all kind of programmers from beginners (a simple robot can be written in just a few minutes) to experts (perfecting an AI can take months).

The last approach uses video games in order to hook the player and bring him/her to programming. Two uses have been experimented: implementing new video games and playing video games. For example, in [22], students are required to implement in C++, through a collaborative project, a small-to-medium scale interactive computer game in one semester, making use of a game framework. In [23], a case study based on EEClone is proposed. EEClone is an arcade-style computer game implemented in Java: students analysed various design patterns within EEClone, and from this experience, learned how to apply design patterns in their own game software. In [5], a "Game First" approach is used to teach introductory programming. These authors believe that game programming motivates most new programmers. They use 2D game development as a unifying theme.

Another solution is to let students learn when they play a game. Two games use this approach: the Wireless Intelligent agent Simulation Environment (WISE) [24] and Colobot (http://www.ceebot.com/colobot/index-e.php) accessed 21 September 2007. WISE combines activities from virtual and physical versions of the Wumpus World game. It allows physically distributed agents to play an interactive game and provides a dynamic learning environment that can enhance a number of computer science courses: it can be used as a medium for demonstrating techniques in lectures; in the classes students can work on laboratory exercises that test, expand, or modify the simulator. The Wumpus World game can be played cooperatively or competitively. But WISE requires a great number of resources (e.g., space, robots, and so on) for the physical version.

Colobot is the only example that we know, of a complete video game, which mixes interactivity, storytelling, and programming. In this game, the user must colonize a planet using some robots that she/he is able to program in a specific object oriented programming language similar to C++. The only drawback in our opinion is that Colobot has no multiplayer mode.

3.4. Conclusion. "*Although there is a weak theoretical basis and few techniques for measuring learning in computer science*" [21, page 151], several environments for teaching programming have been developed. Stevenson and Wagner [17] define a set of characteristics for a "good assignment." All the studied environments agree with the criteria number two (generate a realistic solution), three (focus on current topics from class), four (be challenging), five (be interesting), seven (offers multiple levels of challenge), and eight (allow creativity and innovation). However none is based on real-world problems. But when students use Robocode, EEClone, and Colobot, they use complex APIs.

Novice programming environments (StarLogo, Scratch, Alice2, and Cleogo) are not games and cannot be considered

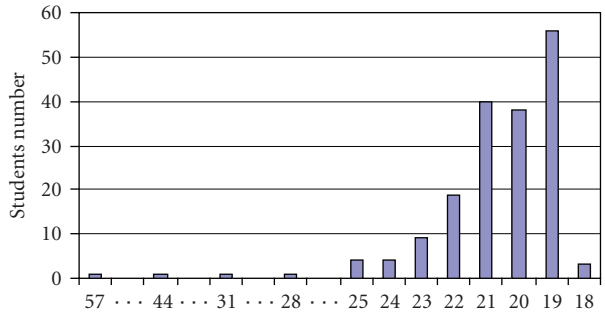

Figure 2: Age of students.

as serious games. In [5, 22, 23], since students have to build a game and not to play, these approaches cannot be considered as games. Among the others, only Robocode, WISE, and Colobot use a gaming activity to stimulate the player. However, only Colobot allows the player to interactively program in game. But it is not free and cannot be adapted to different teaching context. Indeed, it is devoted to a specific programming language and cannot be adapted to specific pedagogical choices. Since in [10] there is no recommendation about the programming language, there is no consensus about the choice of a language. Indeed for pedagogical reasons, some teachers develop their own programming languages. Moreover, it is not easy for a teacher to introduce new exercises into the game. Since our experiment lies on design experiments, we need to test a serious game dedicated to teaching programming in several contexts with different teachers and students. The existing games do not allow that, and thus we decided to build our own serious game.

4. What Kind of Video Game for Our Serious Game?

The usability of a serious game is based on the compatibility between the learning objectives and the target public. Since we want to build a serious game for students in computer science, we have to base it on one of their most played video games. Our first step was thus to ask what kind of video games our students practice, and to find the most suitable one to motivate students to program.

4.1. Students and Video Games. To check the interest of our students in the kind of game they practice, we submitted a survey: 181 students were questioned (154 males and 27 females) in three different curricula (two on computer science and one on civil engineering). The average age is 21 years old (see Figure 2 for distribution).

The first analysis verifies ESA's results about the student interest in games at our university. Figure 3 shows the percentage of students who play (males and females). More players are males but a small majority of females also play video games. We infer from these figures that *video games are widely played by our students*, even for females. These results corroborate the potential of serious games for these students.

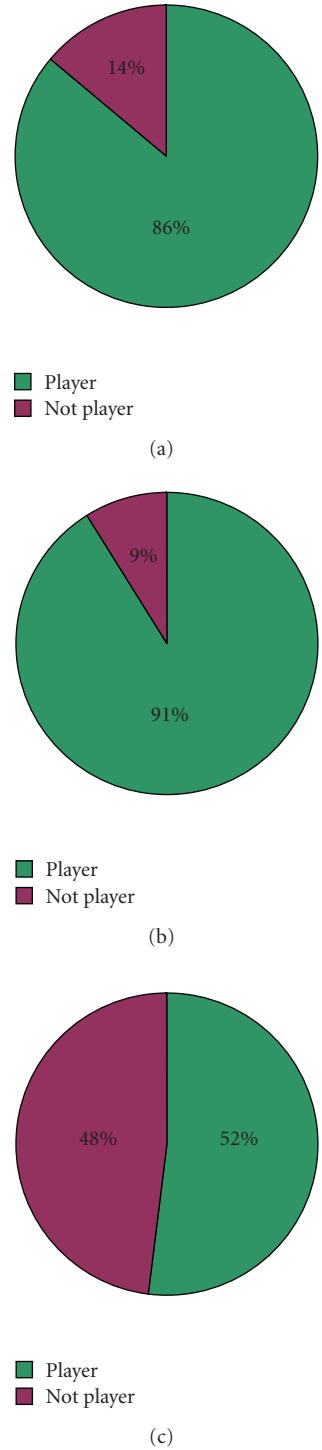

Figure 3: (a) Players' percentage for all participants, (b) males, and (c) females.

The second analysis identifies the most used game genre played by our students. Figure 4 shows the percentage of players who play each game genre. The *most played game genre is strategy games*. We notice that this game genre is also appreciated by females (57% of female players play strategy games).

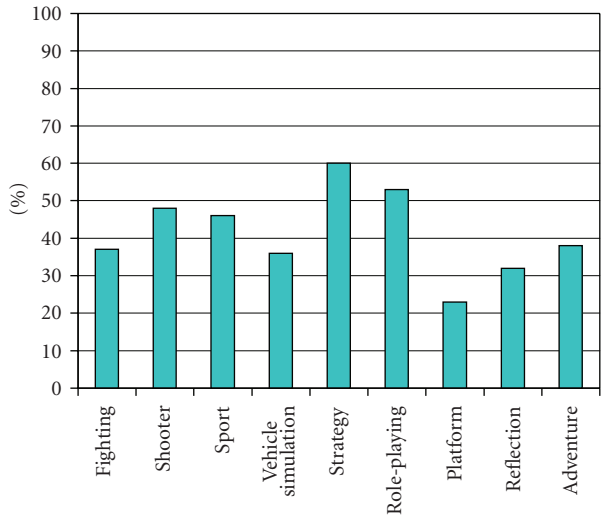

Figure 4: Percentage of players who play each game genre.

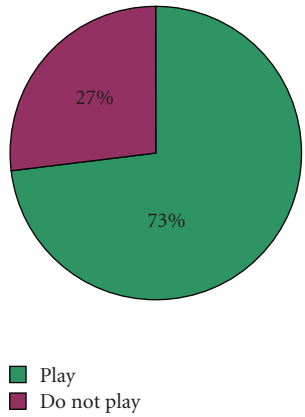

Figure 5: Percentage of players who play multiplayer games.

Figure 5 shows the percentage of students who used multiplayer games. As we can see, most of them use this type of game.

Thus, to be adapted to our target audience, our serious game should be based on a multiplayer strategy game.

4.2. What is a Strategy Game? Strategy games are, by and large, represented by Real-Time Strategy (RTS) games. In this game genre players evolve in a virtual environment, where resources are scattered on a map. RTS is traditionally structured around three main phases closely linked: harvesting resources, building structures and units, and fighting opponents. To win a game, the player has to destroy all structures and units of opponents or achieve a specific goal. To build a strong army, a good economy is required and protection of strategic areas is essential.

A strong player should show abilities in planning and know how to anticipate and react. She/he has to command hundreds of units, which leads to a large cognitive load. Moreover, RTSs have an important feature: the "fog of war." This hides the movements and actions of opponents until they come into the line of sight of one of the player troops. The player evolves in a virtual world with incomplete information, which increases the game interest.

Traditionally, RTSs provide two types of game: *Campaign* and *Skirmish*. Campaigns attract the player and teach him/her how to play, and skirmishes extend the life of the game. A campaign is divided in missions that gradually introduce game contents and complexity. Skirmishes require a better control of the game. The player fights against computer AIs or other players. Moreover, to increase the game challenge, it is always possible to find better or equivalent players on the Internet.

In RTS games, a player gives orders to his/her units to carry out operations (i.e., moving, building, etc.). Typically, these instructions are given by clicking on a map with the mouse. An RTS game, where such instructions can be given through programs, might be the answer to our serious game. The idea is to stimulate the player to give orders through programs. These programs will assist the student/player during the game and should increase his/her probability of winning, if they are efficient, relevant, and suitable to the game. Moreover, when they test their programs, students will still use the same environment (the game).

4.3. Are Strategy Games Compatible with Teaching Programming? As we have seen before, serious games for teaching programming already exist and are used. In particular, Colobot is based on a sort of RTS. In Colobot, teaching is provided through an interactive library available for consultation but a teacher using Colobot cannot modify or adapt it to his/her courses. And there is only one course level for novice programmers (PF).

Learning how to program requires writing programs. A priori, RTS and programming activities are incompatible: real-time games are dynamics and have a strong interactivity with the player, and programming tasks require time for design and implementation of programs. Integrating programming activities in an RTS should then modify foundations of the game. Colobot and Robocode found two different solutions to solve this problem.

(i) Colobot is based on a modified RTS to enable in-game programming. The common rules of the game are modified by this fact, and it demands a specific skill from the player. For example, the player cannot control several entities in the game at the same time.

(ii) Robocode distinguishes between programming and playing activities. First, the player writes an AI, and then she/he runs them. Thus the player is inactive during the simulation and is merely a spectator of his/her AI.

5. Implementation

RTSs are very complex programs, with more than tens of thousands of code lines. Because our goal was not to develop a new RTS, we decided to use an existing engine. This engine had to be open sourced to allow us to develop the specific features of the serious game.

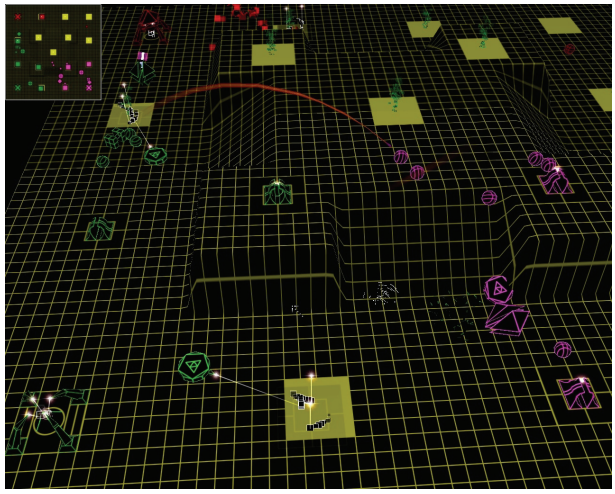

Figure 6: Kernel Panic.

5.1. Game Engine Choice. We found two open source multiplayer 3D RTS engines: the *Spring project* (http://spring.clan-sy.com/) accessed 2 February 2007, and *Open Real-Time Strategy* (ORTS). ORTS [25, 26] has been developed to provide a programming environment for discussing problems related to AI. This game is designed to allow the user to easily program and integrate his/her AIs. It is aimed at users who already know how to program. Spring is a project aiming to create a new and versatile RTS Engine which was built to reuse some game data from a commercial game called Total Annihilation. Currently, Spring is more successful than ORTS. A gamer community plays Spring everyday on the Internet. This community helps to discover bugs, which are fixed by a development group. This process is not present in ORTS which is experimental. We chose the Spring engine instead of ORTS because of this community.

5.2. Characteristics of Spring. Along with the Spring engine, several "mods" exist (http://spring.clan-sy .com/wiki/Mods) accessed 26 August 2008, (mods constitute additions to a game that change the way it works). For our experiment, we chose to use "Kernel Panic" (http://spring.clan-sy .com/wiki/Kernel_Panic) accessed 26 August 2008; Figure 6 presents a screenshot of Kernel Panic, where three players (red, green, and pink) fight on a map. Kernel Panic uses computer science metaphors, like bits and pointers, which is an asset for our training purposes. Moreover, Kernel Panic is a simplified RTS with the following features: there is no resource management except for time and space; all units are free to create; it has a small technology improvement tree with less than ten units; it uses low-end vectorial graphics, which match the universe. These characteristics emphasize strategy and tactics in an action-oriented game while always remaining user friendly.

5.3. Serious Game Implementation. To adapt the Spring engine to the serious game we wanted to build, we had to take into account some constraints: (i) allow players to write code plugged dynamically into the game; (ii) protect the game engine against player's code bugs; (iii) hide the complexity of the game engine; (iv) support different programming languages. The integration of the player's code in the engine must be interactive (without stopping the game) in order to maintain the progress and coherence of the game.

Table 1

GEI_Init	Create the shared memory
GEI_Quit	Close the shared memory
GEI_Update	Make an update if it is required
GEI_ExecPendingActions	Execute pending actions

Table 2

openGame	Connect the program to the shared memory
closeGame	Disconnect the program from the shared memory
refreshGame	Ask the game engine to update its data
getMapSize	Get the map size
numberEnemies	Get the number of enemies visible by the player
numberUnits	Get the number of units controlled by the player
getUnit	Get the nth unit of the player
giveOrder	Set an action to the unit on a target
unitStop	Command a unit to stop its current action

In some previous works [27] we used an implementation based on a dynamic library. Use of dynamic libraries turned out to be inadaptable to interpreted languages. Indeed, dynamic libraries solve problems in a single process: the game engine. This process controls student's computer programs. But interpreted languages also require an interpreter which is carried out in its own process. Thus, we discovered limits of the use of a dynamic library containing the player's code.

Considering this drawback, we designed a new system. Students' programs are not included in a dynamic library loaded and performed by the game but are running in an independent process and communicate with the game. This enables the use of compiled or interpreted languages. A set of techniques exist for exchanging data among processes. We needed a portable and fast solution designed for process communication and not just threads communication. We chose to use the Boost interprocess library (http://www.boost.org/doc/libs/1_37_0/doc/html /interprocess.html) accessed 26 November 2008, that provides shared memory functionality. Moreover, this library offers the possibility to use complex data, like vectors or maps, in the shared memory.

The UML component diagram in Figure 7 expresses the dependencies between the player's program and the game engine. These two components interact through the Game Engine Interface (GEI). The "Supplier GEI" is used by the game engine. We have integrated into the game engine some modifications. When the game starts, it creates the "Supplier GEI," and then the supplier interface can be used through the subroutines in Table 1.

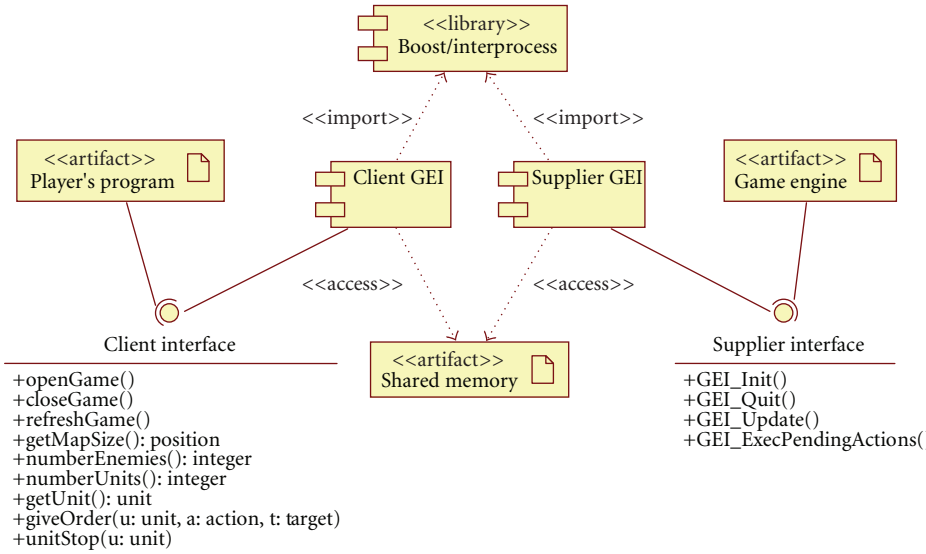

Figure 7: Architecture.

Table 3: Teaching organization schedule.

Session 1	Session 2	Two weeks	Session 3	Session 4
Presentation of the initial version of the RTS followed by a multiplayer playing session	Presentation of the API and carrying out the five missions of the campaign	Students develop their own programs by themselves with teachers remote tutoring and if they want with peers	Multiplayer playing using student's programs	Institutionalization

The "Client GEI" is used by the player's program. After creating the "Client GEI," it can use the client interface to interact with the game engine through the subroutines in Table 2.

GEIs use the Boost interprocess library to interact with the shared memory. GEIs hide the synchronization complexity of the shared memory and make communication with the game easier. At the request of the student program, pertinent data is copied into the shared memory. To avoid incoherent situations, students' programs work on this copy. In this way, at any time, the player can change his/her code and carry it out to use the shared memory and to communicate with the game.

All languages that are able to use a C library can use the "Client GEI" and communicate with the game engine. Currently, we propose interfaces for the "Client GEI" in C, C^{++}, Java, Visual Basic for Application (VBA), and an interpreted language called "Compaglo" (used in a specific course at our university).

6. Mapping Learning Objectives into the Game

Is the built serious game compatible with learning objectives? Different viewpoints can be envisaged: the student view, the teacher view, and the knowledge view.

6.1. The Student Point of View. The first step for a student is to learn how to use the serious game. Especially at the beginning, students have to understand what they can program and how they can do it through the "client GEI." A campaign seems to be very suitable for game appropriation where motivation is maintained by a story.

When players are over with all the missions, they should be able to control the serious game and develop their own AIs for skirmishes. To build these AIs, players need to call upon their skills learned during the campaign. They have to design and implement each AI. The developed AIs span from simpler ones for novices, to very complex ones for experts.

In skirmishes, a student can play against the computer or against his/her friends. Multiplayer sessions encourage them to carry out new challenges. The motivation is maintained by competition between players.

The player defines a strategy, composed by a set of tasks, to win. She/he can choose to carry out some of them through AI. If the developed AIs improve the game, students will be better when they play. They will then perhaps find interest in programming, spend time to perform it and so increase their abilities in programming practice.

6.2. The Teacher Point of View. The serious game we built can be adapted to the programming language chosen by

```
program mission1
glossary
    Unit u
    Position p
begin
    openGame()
    p.x ← 1056
    p.y ← 1792
    u ← getUnit(1)
    giveOrder(u, MOVE, p)
    closeGame()
end
```

Algorithm 1: Mission one algorithm.

the teacher. According to the language characteristics, she/he can build activities for different course levels from PF to complex AI algorithms. Moreover, if s/he chooses to use the multiplayer mode, she/he can use individual, competitive, and collaborative pedagogical methods.

6.3. The Knowledge Point of View. Because unlike Colobot, the tool has not an interactive library of programming skills, it cannot be used without an appropriate teaching environment: students need to be assisted by teachers and peers to write their AIs and, after playing, an institutionalization [28] stage is necessary to carry out collaborative learning. Simon [29] defines institutionalization as a phase where ideas "*constructed or modified during problem solving attain the status of knowledge in the classroom community*." PF skills are gradually introduced through the missions, which initial aims are to progress in the story.

GEI is a fairly complex Application Programming Interface (API) which is not simple for novices to use. Teachers can develop an overlay adapted to their own subroutines specifications.

7. First Experiment

Before conducting design experiments with different teachers and students, we decided to test our serious game on some of our students and first define a simplified design experiment. To comply with the traditional game mode of an RTS, we first propose to carry out a campaign and then to develop additional programs usable in multiplayer sessions. We first present an example of a campaign, then we show how to organise a skirmish to ensure learning.

7.1. Our Campaign. "Kernel Panic" is only a multiplayer game and does not provide campaigns. Therefore, we built a campaign to gradually introduce learning topics and enable students to learn how to play and to program AIs. We take advantage of the Kernel Panic universe and offer students the following scenario: "*For a certain number of years, a secret war is rife inside computers. Steady attacks are led against innocent victims. Today is your turn. Your aggressor captured your mouse controller. You must recover it. Your only solution: programming.*"

To achieve this objective, five missions are created.

(i) Mission 1. "*You lost a lot of units in the last attack. Units currently alive are dispersed on the map. You have only one BIT under control. You must go to the rally point at position (1056, 1792) to find other units*." To succeed, the player has to make a small program where she/he uses variables, types, assignments, functions, parameter passing, and records. Algorithm 1 shows a solution.

(ii) Mission 2. "*You just found a BYTE unit. It tells you that other units are reassembled not far. It gives you the position (479, 1825) of a BYTE group that it tries to rally. Moreover, it warns you that a group of BITS is forming at position (1400, 1371). To retrieve these units, command your two units to meet up with their respective groups*." In this mission the conditional control structure is introduced to give a target position to each unit according to their type (BYTE or BIT).

(iii) Mission 3. "*All units you control are weakened. You must repair them before starting a counter attack. A report indicates that an ASSEMBLER is posted at the position (256, 1024). Find it and it will help you*." In this mission the iterative control structure is introduced to iterate through each unit and move them on the right position. Algorithm 2 shows a solution.

(iv) Mission 4. "*You found an ASSEMBLER. Use it to repair your weakened units*." This mission is the most complicated and requires overlapping iterative and conditional control structures. The player has to iterate through each units and commands the ASSEMBLER to repair a unit if this unit is weakened and if the ASSEMBLER is inactive.

(v) Mission 5. "*All units are repaired. Now it is time to fight back. The mouse device is positioned at (1056, 256). Good luck commander*". This mission goal is to reward students with a simple fight.

7.2. Skirmishes. When students finish the campaign, they can write their own AIs and use them during skirmishes. Here are some examples of strategic AIs which could be written by students and give an "in-game" asset to the player: "Search for opponent" to quickly find the enemy in order to adapt one's strategy to the adversary's; "Create a mine field" which may slow down opponents' expansion in order to give more time to develop our own strategy; "Repair" using specific units to keep strategic units in good health; "Withdraw" to protect units when facing a stronger opponent. All these examples support a part of a player strategy and let him/her take charge of the rest, and therefore play the game at the same time.

Algorithm 3 shows the algorithm of "search for opponent" where units search for the enemy. Random target areas

```
program mission3
glossary
    Unit u
    Position p
    Counter c
begin
    openGame()
    p.x ← 256
    p.y ← 1024
    for c ← 1 to numberUnits() do
        u ← getUnit(c)
        giveOrder(u, MOVE, p)
    endFor
    closeGame()
end
```

Algorithm 2: Mission three algorithm.

are computed to move each unit until an enemy is found. It uses library subroutines, like "giveOrder(u, MOVE, pos)" to move the unit "u" to the position "pos." Usually, the player does this with the mouse and has to select each unit one by one, a long and tedious process. The loop allows the player to perform this operation automatically. Moreover, while the player is selecting the enemy units with the mouse, she/he cannot carry out other tasks. With this program she/he can, for example, develop his/her base while the program explores the map.

7.3. Organization of the Course. Table 3 shows the schedule of our teaching organization for our first experiment. Two instructors supervise each session: one game specialist and one computer science teacher. During the first session, *students play the game* to familiarize themselves with it. A discussion about what could be done to improve the game and which are the most efficient strategies for winning is initiated. The second session is a presentation of GEI. The computer science teacher proposes that all *students carry out missions*. The programming obstacles are dealt in concert with the teacher. During the two next weeks, students work autonomously but can call upon their instructors. They have to *develop their own AIs*. If they have no idea of what to program, a database of efficient algorithms is proposed such as the "Search for opponent" algorithm. The game specialist guides the students through different game strategies to improve the playing sessions. The computer science instructor deals with installations and programming problems. The students are allowed to communicate with each other. They can then elaborate alliances or cooperation strategies, or simply help each other with programming. When all the programs are completed and operational, the third session occurs: *students play using their own programs*. The game specialist teacher turns his attention to decipher what really happened during the game: the role of the programs, the activities of the students, and the strategies used. This observation is the base of the last session: students and teachers analyse games and try to find the reasons behind

```
program search-for-opponents
glossary
    Position pos, map
    Unit u
    Boolean found
    Counter c
begin
    found ← FALSE
    openGame()
    map ← getMapSize()
    while not found do
        refreshGame()
        // Check if an enemy is visible
        if numberEnemies() > 0 then
            // stop all units
            for c ← 1 to numberUnits() do
                unitStop(getUnit(c))
            endFor
            found ← TRUE
        else
            // give random target area to move for each unit
            for c ← 1 to numberUnits() do
                // choose a random target area
                u ← getUnit(c)
                if u.inactive then
                    // choose a random value between 0 and map.x
                    pos.x ← randomValue(map.x)
                    // choose a random value between 0 and map.y
                    pos.y ← randomValue(map.y)
                    giveOrder(u, MOVE, pos)
                endIf
            endFor
        endIf
    endWhile
    closeGame()
end
```

Algorithm 3: "Search for opponent" algorithm.

victories and defeats. A discussion about the importance of the programs is held. The learning objective is that our students continue to use by themselves this serious game and improve their programming skills.

This experiment will be conducted in our university this year with novice students. To evaluate the process we will use several indicators such as student investment, number, quality and pertinence of the written programs, student retention, gained skills and exam results. We also want to evaluate the "feelgood" factor as defined in [30].

8. Conclusion

This paper deals with the compatibility between a serious game and teaching programming. Serious games are more and more popular and can meet learning objectives. On the other hand, computer science students encounter a lot of difficulties while learning programming. Some researchers in computer science education develop programming environments to encourage and retain students. Some of these

environments can be considered as serious games but they are not adaptable enough to validate our hypothesis in regards to design experiments, which is why we decided to build an adaptable serious game dedicated to programming.

As a basis for our serious game we chose to use an RTS, because it is the most played game genre for our target audience. Because it was not possible to develop our own RTS engine, we decided to use the Spring game engine and the Kernel Panic game. The implementation of the serious game led to modifying the engine to enable an interactive and secure programming activity through an API. The students can command game entities with their own AIs and have contests with their friends in the multiplayer mode. The game can be adapted to specific programming languages, and teachers can adapt the API to their own specification subroutines. PF skills are mapped on the game through missions. In order to validate the game, we designed a first design experiment with our students.

The next step is to conduct this experimentation and to adapt it to several contexts with different instructors and students in order to apply the iterative process of design experiments. The possible evolution of the serious game is the introduction of teaching facilities, like Colobot, and in order to keep pace with the rapid evolution of video game standards, the use of another mod, or the integration of other RTS game engines.

We hope that these experiments will show us the breadth of teaching applications supported by our system as well as the range of potential audiences and teaching methodologies. Analysis of our experimentation will explore and resolve potential issues concerning usability and effectiveness of learning with serious games. At the same time, it will be important to determine which skills are learned by students when the campaign is finished and how users switch between game play and coding elements. It would also be interesting to evaluate this approach with another video game genre and to compare it with our RTS-based serious game.

References

[1] T. L. Crenshaw, E. W. Chambers, and H. Metcalf, "A case study of retention practices at the University of Illinois at Urbana-Champaign," in *Proceedings of the 39th ACM Technical Symposium on Computer Science Education (SIGCSE '08)*, pp. 412–416, Portland, Ore, USA, March 2008.

[2] C. Kelleher, "Alice and The Sims: the story from the Alice side of the fence," in *The Annual Serious Games Summit (DC '06)*, Washington, DC, USA, October 2006.

[3] ACM/IEEE-Curriculum 2005 Task Force, *Computing Curricula 2005, The Overview Report*, IEEE Computer Society Press and ACM Press, New York, NY, USA, September 2005.

[4] S. Fincher and M. Petre, "Mapping the territory," in *Computer Science Education Research*, RoutledgeFalmer, pp. 1–8, Taylor & Francis, Boca Raton, Fla, USA, 2004.

[5] S. Leutenegger and J. Edgington, "A games first approach to teaching introductory programming," in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07)*, pp. 115–118, Covington, Ky, USA, March 2007.

[6] P. Cobb, J. Confrey, A. DiSessa, R. Lehrer, and L. Schauble, "Design experiments in educational research," *Educational Researcher*, vol. 32, no. 1, pp. 9–13, 2003.

[7] M. Zyda, "From visual simulation to virtual reality to games," *Computer*, vol. 38, no. 9, pp. 25–32, 2005.

[8] S. Blackman, "Serious games...and less!," *Computer Graphics*, vol. 39, no. 1, pp. 12–16, 2005.

[9] F. L. Greitzer, O. A. Kuchar, and K. Huston, "Cognitive science implications for enhancing training effectiveness in a serious gaming context," *ACM Journal on Educational Resources in Computing*, vol. 7, no. 3, article no. 2, 2007.

[10] ACM/IEEE-Curriculum 2001 Task Force, *Computing Curricula 2001, Computer Science*, IEEE Computer Society Press and ACM Press, New York, NY, USA, December 2001.

[11] D. Ginat, "On novice loop boundaries and range conceptions," *Computer Science Education*, vol. 14, no. 3, pp. 165–181, 2004.

[12] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: an empirical study," *Communications of the ACM*, vol. 26, no. 11, pp. 853–860, 1983.

[13] O. Seppälä, L. Malmi, and A. Korhonen, "Observations on student misconceptions—a case study of the Build Heap Algorithm," *Computer Science Education*, vol. 16, no. 3, pp. 241–255, 2006.

[14] E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, "What do novices know about programming?" in *Directions in Human-Computer Interaction*, pp. 87–122, Ablex, New York, NY, USA, 1982.

[15] M. McCracken, V. Almstrum, D. Diaz, et al., "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," in *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '01)*, pp. 125–180, Canterbury, UK, June 2001.

[16] S. Stamm, "Mixed nuts: atypical classroom techniques for computer science courses," *Crossroads*, vol. 10, no. 4, p. 3, 2004.

[17] D. E. Stevenson and P. J. Wagner, "Developing real-world programming assignments for CS1," in *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '06)*, pp. 158–162, Bologna, Italy, June 2006.

[18] E. Klopfer and S. Yoon, "Developing games and simulations for today and tomorrow's tech savvy youth," *TechTrends*, vol. 49, no. 3, pp. 33–41, 2005.

[19] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: a sneak preview," in *Proceedings of the 2nd International Conference on Creating, Connecting and Collaborating through Computing*, pp. 104–109, Kyoto, Japan, January 2004.

[20] C. Kelleher, D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, and R. Pausch, "Alice2: programming without syntax errors," in *Proceedings of the 15th Annual Symposium on the User Interface Software and Technology*, Paris, France, October 2002.

[21] A. Cockburn and A. Bryant, "Cleogo: collaborative and multi-metaphor programming for kids," in *Proceedings of the 3rd Asian Pacific Computer and Human Interaction*, pp. 189–194, Shonan Village Center, Japan, July 1998.

[22] W.-K. Chen and Y. C. Cheng, "Teaching object-oriented programming laboratory with computer game programming," *IEEE Transactions on Education*, vol. 50, no. 3, pp. 197–203, 2007.

[23] P. Gestwicki and F.-S. Sun, "Teaching design patterns through computer game development," *ACM Journal on Educational Resources in Computing*, vol. 8, no. 1, article no. 2, pp. 1–22, 2008.

[24] D. J. Cook, M. Huber, R. Yerraballi, and L. B. Holder, "Enhancing computer science education with a wireless intelligent simulation environment," *Journal of Computing in Higher Education*, vol. 16, no. 1, pp. 106–127, 2004.

[25] M. Buro, "ORTS: a hack-free RTS game environment," in *Proceedings of the 3rd International Conference Computers and Games (CG '02)*, vol. 2883 of *Lecture Notes in Computer Science*, pp. 280–291, Edmonton, Canada, July 2002.

[26] M. Buro and T. Furtak, "On the development of a free RTS game engine," in *Proceedings of the 1st Annual North American Game-On Conference (GameOn'NA '05)*, pp. 1–5, Montreal, Canada, August 2005.

[27] M. Muratet, P. Torguet, and J.-P. Jessel, "Learning programming with an RTS based Serious Game," in *Serious Games on the Move International Conference*, Cambridge, UK, June 2008.

[28] G. Brousseau, *Theory of Didactical Situations in Mathematics*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[29] M. A. Simon, "Learning mathematics and learning to teach: learning cycles in mathematics teacher education," *Educational Studies in Mathematics*, vol. 26, no. 1, pp. 71–94, 1994.

[30] M. M. Muller and F. Padberg, "An empirical study about the feelgood factor in pair programming," in *Proceedings of the 10th International Software Metrics Symposium (METRICS '04)*, pp. 151–158, Chicago, Ill, USA, September 2004.