A Game Oriented Approach for Teaching Computer Science

Esteban Walter Gonzalez Clua[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
R. Passo da Pátria 156 - Bl E - 3º andar - Niterói – RJ / Brazil

esteban@ic.uff.br

Abstract. *Introductory computer science courses should be altered to use attractive, engaging contexts. After defending this claim, the paper provides evidence that the context of computer games qualifies as both attractive and engaging. The heart of the paper is a description of how certain logical, mathematical, and algorithmic concepts fundamental to computing may be taught and learned within a game-oriented context. Tools and languages adequate for this purpose are briefly discussed. The author describes his initial experiences and observations in using this approach.*

1. Introduction

Teaching the concepts required for introductory computer science education has always been a challenge. While there is much debate about methodologies and approaches best for this purpose, most educators agree that introductory computer science education should focus on conveying the required logical, mathematical and algorithmic concepts, rather than on specific languages or technologies. Computer scientists know and appreciate the math, logic, theory and practice of our discipline. We know that others who find and appreciate these qualities in the experience of programming might also have successful and satisfying careers in computer science. Often, however, introductory computer science education fails to convey this experience, and is instead reduced to tedium not by the theory itself, but because the tools, languages, examples and projects adopted for the courses lead to boredom. The traditional text-based "Hello World" is a canonical example. Even a brilliant and engaging university lecturer suffers if they do not have an interesting methodology and curriculum to present to an introductory computer science class.

In a recent informal study, the author of this paper posed a question to a traditional computer science class, given by a very good traditional "Hello World" professor, and asked the students before the first class: "How many of you think you will like this introduction to computer science?" Of 44 students, 32 said that they would. At the end of the semester, he asked the same students the question: "How many of you enjoyed this introduction to computer science?" Of the 41 remaining students, 11 said that they had. Citing published studies, Guzdial [6] has emphasized this point.

Computer science departments are not currently successful at reaching a wide range of students who are taking introductory computer science. The evidence for this statement includes international studies of programming performance [10], declining retention rates [7], and failure rates sometimes as high as 30% [13]. In particular,

participation of women in computer science is dropping. Studies suggest that computing courses are seen as overly-technical and avoiding relationships to real applications [9], and are frankly boring and lacking opportunity for creativity [1].

A recent Brazilian research study [2] showed that almost 60% of students who abandon their course of study do so in the first year. This specifically demonstrates that the level of motivation conveyed by a first-year introduction to the discipline is crucially important.

There is also a tendency for curriculum to become quickly dated by rapidly changing technology, and this dating can happen not only around the tools and technologies which are used to teach, but also simply around the user experience of the examples presented by the curriculum. On the first point, the author selected twenty computer science courses' curricula currently taught at the best Brazilian universities, based on Brazilian Computer Society [14] criteria. Thirteen of those courses are using old-fashioned languages (Pascal, Fortran and Basic) and seven still use MS DOS-based programming environments. On the second point, students in 2007 casually use the Internet, browsers, media players, cell phones with graphical displays, console games, etc. Their reaction to text-based programming curricula is far different from the reaction of those who grew up with only text- and keyboard-based computer interfaces.

Many institutions and associations, such as the ACM or the SBC (Brazilian Computer Society) have elaborated computer science fundamentals programs, which define the introductory CS disciplines that should be conveyed. Most of them agree with the following list of concepts:

•Algorithms: how to solve complex problems by using simpler concepts and operations. Some specifications also say: "and using math operations". [5]

•Variable declaration and usage

•Data Structures: there is debate about this topic, as many authors [3] feel this is not a recommended concept for an introductory course. The examples of data structures presented depend on the professor, the book being used or the university curriculum. Modern curricula generally agree that it is important to prepare students for understanding object-oriented data structures and object-oriented programming [3].

•Abstract Data Types

•Functions

•Iteration and Recursion: Again the recommended focus here is on teaching how to solve problems using loops, rather than teaching a specific language's implementation of "while" and "for" loops. Some teaching programs prefer to teach recursion first, in order to make this approach more abstract [11].

•Fundamentals of Software Engineering: Good programming practices and project collaboration must be taught at the beginning for the same reason that good penmanship should be taught early. Teaching and learning these early best ensure that the practitioner will move forward with good habits of practice rather than bad ones.

On the other hand, topics that are not considered essential for an introductory CS course are:

•Complex object-oriented concepts like inheritance and polymorphism.

•Window programming: It is difficult, boring and unhelpful to teach beginners the concepts and handling of windows, buttons and other graphical user interface elements.

•Complex data structures

•Memory allocation and manipulation

2. Related Work

The use of games in teaching introduction to CS is not new in academia, but today is still limited to a small number of professors and educators who are focused on or have passion about games. Many of this works only presents the usage of games only for a very preliminary conceptually teaching, such as [19] and [20] Most CS professors do not have a background in or experience with the game development process. This, combined with the scope and the complexity of game development tools and technologies, sets a high learning curve for a professor who is interested in using game development in support of his computer science classes. These factors can also make it difficult to convince that professor's institution to adopt such a curriculum officially. Kid's Programming Language and its new version, Phrogram, [8] are among the few examples of game-oriented programming tools that can introduce to beginners the core concepts listed above. Functionalities that are not part of those core beginner concepts, such as reading joystick input, drawing sprites on the screen, or playing sounds and music, are enabled in a easy-to-use way within the language libraries, allowing professors and students to focus on the algorithm itself. The author discussed KPL and other methodologies used for CS education in a SIGGRAPH 2006 panel [4] and a SIGGRAPH 2006 paper [15]. This paper is in large part a follow up to those presentations, including definition of specific methodologies, and presentation of the results of those methodologies.

Some of the advantages of using game-oriented teaching for an introduction to computer science include:

•Examples and solutions offer very immediate, pragmatic and visual feedback, helping to resolve the typical formalism difficulties encountered by beginners.

•Interesting, engaging and fun examples and applications motivate students to go further in their learning – including often encouraging students to explore and learn well beyond the limits of the course curriculum.

•Interactive environments that allow for immediate visual feedback encourage students to experiment and think more creatively about solving problems.

3. Methodology Proposed

The methodology elaborated here covers the topics considered essential by the standard curriculum of Introduction to Computer Science, as specified by the Brazilian Computer Society.

The methodology is based on a single high-level goal for the introductory class: step-by-step development of a complete 2D game, described here are the detailed steps of this development, with specific approaches for each topic. These topics convey the

required computer science concepts. Meanwhile, students also experience the pleasure of developing the complete game by the end of the class. This reward, of course, is important for their motivation and encouragement.

In the beginning course, it is not recommended that advanced topics such as complex artificial intelligence or physics be used. For a first course and first game, simplicity is best – in part because it is important that the students fully understand every instruction in the game they are creating. For this reason, it is not convenient to distribute large pieces of the functioning code of a larger game, with the students responsible for filling in only a few gaps with their own programming code.

It is worth emphasizing that the best possible motivation and reward for students is that they feel and understand that the game they created is entirely theirs, and that every element and instruction in it is understood by them. Good examples of games which are simple, fun, familiar, and possible for beginners to recreate include versions of Space Invaders, Pac Man, Pong and Asteroids.

3.1 Preliminary Concepts

The very first class session is the most important, not for its content, but for the first impression it creates in students, an impression which must motivate and encourage them as they launch into the course. Analyzing a complex AAA game[1] is a good strategy to start engaging class discussion, such as:

- Importance of clear programming: dozens of professionals are necessary to make a AAA game, so every one must be able to understand what others are doing
- Concept of resources required by a program (sprites, textures, 3D Models, sounds, etc.)
- Importance of performance optimization
- Concept of threads
- Software modularization
- Object-oriented thinking
- An exercise of listing and describing program requirements and features

3.2 Variables

When showing an example of a AAA game, it is easy to illustrate the concept of variables with many visual examples, such as the player's current health, remaining time, and weapon damage amounts. Showing these examples can specifically also introduce various variable types, such as integer, floating point, color, vector, Boolean and strings. It is also important to present and discuss the existence of other variables that are not so obviously visible, such as the position of the player within the game, or the number and difficulty of enemies at each level. This is also a good moment to present the concept of constants. The height and width of the screen can be presented as an example constant. It's a concept they are likely already to be familiar with, and which will help them understand the use of constants in a program.

[1] AAA is the highest rating given to computer games.

As a first exercise, it is possible to ask each student to identify their own favorite game, and then assemble a list of all the variables they can think of which are used in that game, asking also for the corresponding variable type.

After students are comfortable with the concept of variables and their types, a convenient first programming exercise will consist of the declaration of the complete list of suggested variables. Depending on the technology used, at this moment it is possible to teach how to save a header file and in the process explain one of the first software engineering concepts: the importance of modularization of a project. The students should also indicate which of the declared variables is best used as a constant. Other good programming practices which can be taught from this beginning include descriptive variable names, organizing variables and constants into logical groupings within the program, and adding comments explaining the meaning and use of variables.

If a header file was used, this exercise can be concluded by explaining the creation of a separate main program file, the inclusion of the header file which was just created, and the understanding that, when the header file is included, the variables and constants defined within it can subsequently be used inside the main program file.

3.3 Functions

For an adequate learning process that still allows beginning students to create their own complete game, the chosen programming language must offer a set of functions or class libraries that abstract many of the details of game implementation. Doing so removes much of the complexity of making a game work. Examples of important functions include displaying and manipulating sprites, sounds, or other drawing primitives on the screen. KPL or Phrogram [8] are good examples of this kind of language.

After presenting the concept of functions, one or more visual exercises should be presented that require the students to define and use their own functions, and, along the way, understand the concept of user-defined functions as a "black box". Appropriate examples should be based on the technology and students' interest and understanding, and could include, for instance, using functions to:

- Display differently-sized and -colored graphical primitives on the screen, such as circles
- Draw a simple, red thermometer display
- Display a game window with a specific size

3.4 Objects

Advanced object-oriented programming concepts such as polymorphism and inheritance are not appropriate for beginning programmers. However, helping beginners to think about objects,and how they can manipulate objects with their program instructions are important and practical concepts that beginners can learn at this stage.

Sprites are a good way to introduce object-based thinking early. Their immediate visual feedback and reality helps considerably with understanding the abstract representation of objects within a program. Sprites can be represented by variables, but they require many distinct properties, such as position, visibility, rotation,

etc. Sprites also should have many behaviors or methods which students can learn and use, such as displaying them, rotating them, or resizing them on the screen.

Depending on the language and curriculum being used, the complexity of the properties associated with a sprite object can also be a good opportunity for beginners to learn about, understand and experiment with classes and constructors.

The introduction of sprite objects also offers many opportunities for interesting and fun programming exercises. Sprites can be drawn, resized and rotated on the screen. Building on previous work, the initialization of variables can be changed, and the resulting changes in behavior as the program runs can be observed. Functions could be implemented that allow manipulation of sprites on the screen. Loops are still something unknown at this stage, so animating sprites will have to wait for the next class.

After students are familiar with sprites, other examples of objects can be presented, such as background elements of a game screen, background music, and sound effects. At the end of this lesson, abstraction of objects through program instructions will be something that is becoming familiar to the beginners, as is the idea of performing complex manipulations and producing visible results with very few commands.

3.5 Loops and interaction

Animating a sprite is an ideal introduction to 'for' looping. The movement of the sprite across the screen is a very visual way to help a beginner understand the syntax and functionality of a loop. The use and the impact of variables within a loop are easily demonstrated by changing the velocity of movement of the sprite through the use of a variable within the loop.

A high level understanding of 'while' loops and of game programming can be conveyed by explaining a game as nothing more than a big 'while' loop: the game reads information from the joystick, mouse or keyboard, it then makes the changes in the game to react to that player input, it then presents the results on the screen, and finally checks to see if the game is over. If not, we go back to the top of the game loop, read information from the joystick, mouse, or keyboard, and so on.

This concept can be turned into a fun exercise by implementing the main game loop, which controls the movement of the sprite on the screen based on player key presses. The useful new concept this requires beginners to learn, of course, is the reading of and reacting to keyboard input.

3.6 Conditional Expressions

Traditionally, conditional expressions are presented before interaction. However, after showing how to define and show sprites on the screen, it is very easy to comprehend how to manipulate them without conditionals - for this reason loops were presented before *if*.

An ideal presentation of conditional expressions that builds on these previous examples includes teaching students how to make a sprite bounce and change direction when it reaches the border of the game window. Collision sounds can also be added at this point, and are an easy as well as entertaining addition to the exercise.

A very entertaining exercise that builds on all of these previous concepts would allow the player to control a player sprite on the screen, while one or more 'enemy' sprites randomly bounce around the screen. When an enemy sprites collides with the player sprite – bang! Game Over! – with a sound effect and explosion sprite at the collision point, The collision would act as the end condition for the main loop. This exercise becomes the first – if very simple – fully functional game that the beginners have created. This is also a good point in the curriculum to demonstrate and teach about sequential as well as nested *if*s. Also, because, after reading keyboard input, many conditions must be tested, this is a good point to introduce the usefulness of the particular language *switch* statement. Handling keyboard input is an excellent example to help a beginner comprehend the use of a *switch*, and the comparison of an *if* implementation and *switch* implementation provides the instructor an opportunity to demonstrate and teach the importance of readable and easily comprehended code.

Advanced topics at this point could include the use of complex conditional expressions that require logic operators such as *AND* and *OR*. Another such topic is support for complex movement controls, for instance, enabling the user to use SHIFT+RIGHT to make the sprite move more quickly on the screen. Sophisticated artificial intelligence (AI) is beyond the scope of an Introduction to CS course, but simple AI examples can be very exciting for the beginning programmer, and can make for much more interesting games. One that might be taught in the context of a beginners' course is, rather than having the enemy movement be completely random or patterned, adding a very slight adjustment on each move, so that the enemy is moving slightly toward the player each time the two are moved on the screen. These actions will have the game-play effect of the enemy sprite seeming to "chase" and "curve toward" the player sprite as they move around the screen. The implementation of such an AI function can also be excellent reinforcement of the modular and functional programming we are teaching in the curriculum. For instance: at each pass through the main loop, a user-programmed AI_Move_Enemy function is called to calculate and move the enemy. The first implementation of this function can just be a simple random movement, or perhaps a movement continuing along a specific trajectory. But the point of modular programming can be conveyed by challenging students to implement that AI_Move_Enemy function in a way which is more intelligent, or more fun to play, or both.

An inevitable result of using a curriculum like this with a classroom full of students is that some of them will be so engaged by the fun of programming their own game that they will want to go far beyond the class curriculum and exercises. Offering them extra credit exercises throughout the course is an additional encouragement. An extra credit feature at this point in the curriculum might even include the implementation of enemies that "shoot" at the player sprite. This feature involves adding and controlling a third kind of sprite on the game field to represent the trajectory that the projectile follows across the screen. This projectile as well, of course, requires collision testing, probably with both the player and the enemy sprites.

This is also a good time to demonstrate and teach text and string manipulation and display, so that the game programmer can display scores or other messages on the game screen. Again, the fun of learning in the context of programming a game will be

excellent motivation and encouragement for a beginner to learn and to experiment with the manipulation and display of strings of information from their program.

3.7 Data Structures

Data structures are traditionally a curriculum item that is covered after an introduction to computer science course. However, most curricula recommend that at least some of the concepts of data structures be introduced to beginners.

For this purpose, a data structure which is relatively simple to understand is a graph: nodes in the graph might have a list of edges or a list of other nodes they are connected to. They might have a specific shape or location, as well. Depending on the capabilities of the specific programming language, graphs can again be very easy to display and visualize on the computer screen – always an important advantage for a beginner. It is possible to avoid pointer concepts while introducing node graphs in this way, but clearly this example prepares the beginner for going further with data structures in the future. A related exercise builds on previous work, and encourages further abstract thinking based on this example: a set of functions can be built for manipulation of the graph, with functions such as add node, delete node, add node connection, list connected nodes, etc. An advanced exercise might involve building a labyrinth and putting a sprite inside it on the screen, and using a node graph such as this to allow that sprite to "map" and explore the graph while moving through it. This technique could be used to guide enemy sprites' otherwise random movements in a game, but could also be an interesting exercise in programmatically "solving" a maze by finding the route through it.

3.8 Software Engineering

It is important to begin teaching sound principles of software engineering even during an introduction to CS because good habits established early are most likely to continue through students' future work. An approach that has had good results rests on the following task: in the middle of one of the class programming projects, everyone in the class trades programs with one of the others in the class, and continues the work that person has begun. An element of "surprise" is useful in this exercise. Doing this will bring up many questions for the students, including inevitable ones, such as "What is this function for?", "How do I use this function?", and "Why is the variable called wsw and what is it used for?"

Thus important principles of software engineering will come up and can be discussed, such as rules for good code comments, the principle of naming variables and functions well, and the importance of a list of requisites. The usefulness of diagrams of program logic or of game play may come up in the discussion. After this discussion, it is convenient for each student to take back his program example and update it based on the principles discussed. This and future programming assignments should include some portion of the grade based on how well the programming assignment follows the discussed principles of good software engineering.

4. Development Environments

As was said at the beginning, the use of a specific language or tool is something not relevant for the definition of an ideal learning curriculum. However, it is difficult to

elaborate a detailed proposed methodology and curriculum without considering the available technologies. The environment, engine, or language being used must allow for a high level abstraction of programming tasks, to minimize the amount of low level details that the beginner must learn in the process of studying the basic principles and logic of computer science as described here. Window handling and memory manipulation, specifically, are examples of the kind of low level detail that a beginner should not have to deal with in their introductory course. It is possible to use a basic C++, C# or JAVA environment, but a set of functions must be available, like create_sprite, show_sprite, move_sprite, create_window, read_keyboard, play_sound, play_music, etc. KPL and Phrogram are complete programming environments that have all these functionalities, as well as an easy to use compiler. Microsoft XNA [17] and Torque X [16] are also brand new technologies which may be good candidates for this kind of introductory course built around game programming.

5. Conclusion and Future Work

There is very little literature where a detailed methodology for a game-oriented computer science learning process is exposed. This paper's key contribution is a methodology that can be followed by any CS professor who is not familiar with game development, using tools, technologies and curricula that are currently free for academic use. Every pedagogic process, for any specific knowledge area, understands and acknowledges that motivation is a very important factor. Games are nearly-universally played and recognized as the most fun application of computer science as a skill and profession, so why do we continue to use data bases or statistics or textual "Hello world" examples for teaching CS? This paper also shows that a curriculum based on game programming goes further than simply being very motivating, as detailed in Section 3. Esteban Clua is developing complete curriculum materials, with examples, code, assignments, and a more detailed description of the learning process. This material will be ready before the publication of this paper, and will be free for academic use. Kid's Programming Language and Phrogram are also available for download from the Web, and can be used in their freeware version – and are specifically created to support curricula as described in this paper. A release of Phrogram will be available before the publication of this paper that adds support for its programs to also run atop XNA and, thus, on Xbox 360s as well as Windows computers.

6. Acknowledgments

The author acknowledge Jon Schwartz, creator of KPL and Phrogram, who had an important contribution for the process development and experimental validations.

7. References

[1] AAUW. Tech-Savvy: Educating Girls in the New Computer Age. American Association of University Women Education Foundation, New York, 2000.

[2] CAPES website - www.capes.gov.br

[3] Chin, D., Prins, P., Tenenberg, J. The Role of the Data Structures Course int the Computing Curriculum, Panel Discussion at CCSC: Northwestern Conference, December, 2003.

[4] Clua, E., Feijó, B., Rocca, J., Schwartz, J., Graça, M., Perlin, K., Barnes, T., Tori, R. Game and Interactivity in Computer Science Education. In Siggraph Educators Track Proceedings, Boston 2006.

[5] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. Introduction to Algorithms**.** The MIT Pressm, 2 edition (September 1, 2001).

[6] Guzdial, Mark. A Media Computation Course for Non-Majors. *SIGCSE Bulletin*, v. 35, n. 3, pp. 104-108, ACM Press, Sep. 2003.

[7] Guzdial, M. Summary: Retention rates in cs vs. institution. Message posted on ACM SIGCSE moderated members list, Georgia Tech, April 23 2002.

[8] Kids Programming Language - www.kidsproram-minglanguage.com

[9] Margolis, J., and Fisher, A. Unlocking the Clubhouse: Women in Computing. MIT Press, Cambridge, MA, 2002.

[10] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C, Thomas, L., Utting, I., and Wilusz, T. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. A CM SIGCSE Bulletin S3, 4 (2001), 125-140.

[11] http://mitpress.mit.edu/sicp/adopt-list.html

[12] Phrogram - www.phrogram.com

[13] Roumani, H. Design guidelines for the lab component of objects-first csl. In The Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education, 2002, D. Knox, Ed. ACM, New York, 2002, pp. 222-226.

[14] SBC website - www.sbc.org.br

[15] Schwartz, J., Morrison, W., Stagner. Kid's Programming Language (KPL). In Siggraph Educators Track Proceedings, Boston 2006.

[16] Torque Game Engine - www.garagegames.com

[17] XNA - www.microsoft.com/xna

[18] Wing, J.M., Computational Thinking, *CACM*, V. 49, No. 3 (March 2006) pp.33-35.

[19] Barnes, T.; Powell, E.; Chaffin, A.; Lipford, H. Game2Learn: Improving the Motivation of CS1 Students. In Third International Conference on Game Development in Computer Science Education, February, 2008, pp. 1-5.

[20] Rankin, Y.; Gooch, A.; Gooch, B. The Impact of Game Design on Students´ Interest in CS. In Third International Conference on Game Development in Computer Science Education, February, 2008, pp. 31-35.