Learning by Fixing and Extending Games

G. Costantini[1], G. Maggiore[1] and A. Cortesi[1]
[1] Ca' Foscari University, Venice, Italy

Abstract

This paper reports the results of experiencing computer graphics and videogames programming as a way to support the learning process of undergraduate courses on Programming and on Software Engineering, in a "fixing & extending" approach. In particular, we show how some XNA-based tools may provide a successful environment which enhances not only skills and abilities in programming (in the small and in the large, respectively), but also stimulates interest in the theoretical aspects covered in these courses.

Categories and Subject Descriptors: K.3.0 [Computers and Education]: General, K.8 [Personal Computing]: Games

1. Introduction

In this paper we analyze the use of computer graphics as a teaching tool in CS undergraduate courses. Even though this idea has already been explored [PM06], the concept of graphics as a practical application field for software-related courses is not as common as one would think. It is interesting to see which benefits might come from this approach, and how to avoid possible pitfalls, like overwhelming students with too many (or too difficult) concepts, or modifying too much the original aim of the course, or requiring too much work by lecturers and teaching assistants. A crucial role is played by the technologies employed: choosing computer graphics tools that are easy to use, that allow to quickly obtain good results, and that can be adapted to as many contexts as possible.

Graphics in computing is something extremely exciting that can dramatically reduce the feeling of boredom that so often accompanies students in traditional courses [GS02]. Many students enroll to a CS program mostly because they are used to seeing computers accomplish amazing features like 3d animations, media of all kinds, complex UIs and so on. The curiosity to deeply understand how such things are designed is usually what sparks the interest of a freshman student. Moreover students, as young adults, are starting to worry about what is "out there", behind the academia walls: they feel the importance of learning cutting edge technologies that will be used in the IT industry. To sum it up, a student wishes to study topics that will help her/him finding a job, and get the right skills needed to produce tools similar to the applications typically seen in fancy appliances.

What does the teacher gain from this approach? The introduction of computer graphics in a traditional CS course can really make the difference between students who attend lesson "*just because they have to*" and students engaged and passionate about the lessons and the course. Since we know the difference between talking to an interested audience and to an apathetic one, we believe this to be more than sufficient motivation to experiment something new.

Computer graphics is a very large area, and although any specific area of computer graphics could achieve the purpose we are setting, we choose videogames, since it's probably the area closer to the students' interests (and, perhaps more importantly, an area of CG we are familiar with). Students most likely play videogames, so the idea of learning more about these fascinating virtual worlds can make them feel empowered and skilled. Plus, there's another kind of reward they get: after having built a videogame, they can play it!

The structure of the paper is as follows. In the section "*Tools and technologies*" we will see the set of technologies required to put this new approach into practice. After that, we will review two case studies taken from our experiences: in "*Learning by fixing*" we see how broken videogames to be fixed can be the assignments in a first-year undergrad programming course; in "*Learning by extending*" we show how the production of a complete

videogame can be assigned to students in an introductory software engineering course. While exploring the two case studies, we will focus on both preserving the original structure of the considered course and also obtaining the most benefits for students and teachers. Finally, in "*Conclusions*" we will analyze our results for a final wrap up of "gains and losses".

2. Tools and technologies

Computer graphics and videogame development are very demanding and difficult tasks. This means that we risk adding some complexity to a course. We must avoid dealing with the risk to add too much difficulty to courses and with the risk to diverge the attention with respect to the main aim of them. Then, we need to find a tool that requires as less effort as possible to be learned and used, and we need to tune the level of difficulty with respect to the skills of the actual students: it has to be scalable to adapt to different courses.

In fact, consider a scenario in which the students do not know anything about programming yet: we cannot possibly ask them to develop an entire videogame, because they do not have the skills. The ideal would be to give them simple, extremely focused and narrow assignments without caring too much for the rest of the code that would constitute just noise for them. In other cases we could be working with more experienced and skilful students, sometimes even students who are close to their degree, able to program pretty well and capable of team-working. With this kind of preparation we can expect them (with the right support system and the right tools) to design and develop an entire videogame, in a collaborative team.

2.1 The XNA framework

An appropriate tool that suits our needs is Microsoft XNA (**X**NA's **N**ot **A**cronymed) [XNA], a library to create videogames first released in 2004. Since then, XNA has rapidly grown, and has now reached its 3.0 release. XNA supports various areas of game development like Audio, Input, main loop, time frame normalization and of course graphics. What is most interesting about XNA is that on one hand the framework contains classes that make it very easy for an unexperienced user to write her/his game, while on the other hand it supports advanced developers in writing code that requires to manage low level resources (graphics device memory, shaders, etc.). Finally, XNA is maybe the first framework that allows serious game development on both the PC and a next generation console like the XBox 360, with minimal to none code modifications.

Once chosen XNA as the reference framework, there are other important aspects that still need to be explored: which programming language to use, in what development environment, etc. We have to be careful because we cannot choose a language or an IDE that are too specific, and we wish to be able to swap both languages and IDEs depending on the topic of the course and the preferences of teachers or students.

2.2 The .NET Framework

XNA is part of the .Net framework, a framework that contains a massive set of libraries (collections, network communication, serialization, concurrence, parallelism, etc) and a garbage collecting virtual machine. This framework has its own assembly language in which high level languages are compiled. The assembly supports very advanced features, from generics to closures and anonymous functions that make it possible to implement lots of languages with any kind of paradigm, from purely functional to purely procedural/imperative language and any kind of hybrid. Among the supported languages we can find C#, F#, C++, Cobol, Haskell, APL, Eiffel, Fortran, and many more, often coming from Microsoft Research. All of this means that the students who learn to use the .NET framework (or at least who get acquainted with it) will have at their disposal a lot of interesting possibilities, among which that of learning language independent reasoning, that they can explore on their own. The .Net framework is typically used in Windows within the Visual Studio IDE (Integrated Development Environment), but is also supported in many other operating systems through Mono, an open source implementation of the .Net and the C# language standards. Mono and the .Net framework can be used in conjunction with many IDEs different from Visual Studio, some free and some open source, many of which are of amazing quality.

2.3 C# language

If we need an object oriented language, then we should probably use C# [C#LS]. C# is a multi-paradigm programming language defined by a standard that has been approved by ECMA and ISO. It has an object-oriented syntax based on C++ with heavy influences from other languages like Delphi and Java. The most recent version of the language is C# 3.0, released in 2007 in conjunction with the 3.5 version of the .NET Framework. In this version we have seen the emergence of very powerful constructs taken directly from functional programming. These constructs include lambda expressions to denote anonymous functions, anonymous types, currying, and so

on. In C# 3.0, through a set of features commonly referred to as LINQ [BH07], collections and enumerables have even been implemented as monadic combinators, offering the opportunity to use them in an extremely elegant and succinct way.

2.4 F# language

There may be other situations, like introductory programming courses using a functional programming approach, where C# would not be the right choice because of its deep rooting in OO programming. In such a case we can use F# ([SGC07], [Pic07]), a multi-paradigm programming language that allows both pure functional programming (the "F" stays for "functional") and imperative object oriented programming. It is a variant of ML and largely compatible with the OCaml implementation. The language is strongly typed, has a powerful type inference compiler and supports union types, tuples, language integrated lists, lambda expressions, pattern matching and in general anything you would expect from a functional language.

2.5 Team Foundation Server and SharePoint

As an optional (but really interesting) addition to the set of tools we have explored, we mention Team Foundation Server, a source control solution that integrates seamlessly in Visual Studio to offer students the possibility of collaboration in teams with enterprise grade tools but without any kind of mental or visual noise (which is the point of integration after all). Together with TFS comes SharePoint, a collaboration platform providing a centralized repository for shared documents and knowledge in general, that allows creating workspaces for students to meet their need for distributed interaction.

In conclusion, the general architecture of our development environment is the following:

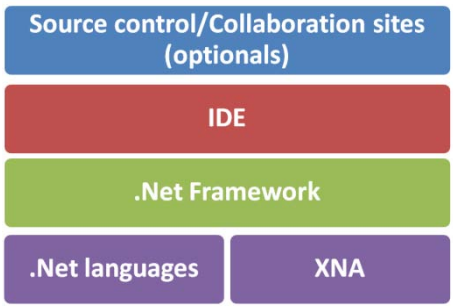


Figure 1: *General architecture.*

3. Learning by fixing for a first-year undergraduate Programming course

In this section we are going to consider the first of our two case studies.

3.1 A first-year Programming Course

The course we are talking about is a first-year undergraduate introductory Programming course. The course has a *functional programming* approach, so we choose F# as language to develop in XNA. The course structure is made of bi-weekly frontal lessons and weekly assignments (which can be solved in the next two weeks). Students can come weekly to the laboratory to be helped by the tutors with the assignments. The assignments are divided in two parts: the traditional one (with exercises run on the command line) and the advanced one (videogame related). Since the introduction of videogames is an experiment, students can choose between the two kinds of assignments.

3.2 Assignments description

As stated before, we cannot ask a first-year CS student to write a videogame alone, but we can ask her/him to *help* writing a videogame: we can give students *small videogames* that are intentionally *broken* by introducing bugs or removing core pieces of source code. The parts of the videogame to be fixed are chosen for each assignment according to the studied topic. The most important thing is that students have to complete a *very small* part of the videogame (something related to the topic studied) but fundamental to make it work. This gives them the satisfaction of solving a problem without having to own a deep knowledge of computer graphics or other difficult subject.

3.3 Assignments samples

Let us see two samples of assignments.

The very first assignment of the course, when the students knew almost nothing about the language, was about the *if-then-else* construct. We prepared a traditional set of exercises, an example of which would be: "*Given a price and a discount, you have to compute the discounted price, keeping in mind that the discount applies only if the price is above 10000$*". Along with this, we also prepared the advanced set of exercises: a simple videogame in which the player (a blue triangle in the centre of the screen) could shoot the enemies (red triangles all around the player) and

kill them (make them disappear). What was the catch? The game was completely broken: the player would shoot nonstop in only one direction and the enemies disappeared wrongly. All the tasks required to repair the game could be solved with the use of if-then-else (**if** the user presses S **then** shoot **else** don't shoot, etc). This way, the students who tried the advanced assignment had to understand the if-then-else construct as well as their colleagues who chose the traditional one, but at the end of their work they had a little videogame up and running, knowing that they contributed to make it right.

Below you can see a screenshot of the game:

Figure 2: *If-then-else game screenshot.*

An example of the code of the assignment can be seen here.
Exercise:

Consider the following code, corresponding to the bugged version of the game, where the function *updateSelf* updates the position and rotation of the player's triangle.

```
let updateSelf =
  fun ((px, py), (sx, sy), r) dt ->
    ((px, py), (sx, sy), r)
```

Fix it by using *Keyboard.GetState().IsKeyDown(Keys.Right)* to check if the right arrow key on the keyboard is pressed, and similarly *Keyboard.GetState().IsKeyDown(Keys.Left)* for the left arrow.

Corrected version of the code:

```
let updateSelf =
  fun ((px, py), (sx, sy), r) dt ->
    if Keyboard.IsKeyDown(Keys.Right) then
      ((px, py), (sx, sy), r + dt)
    else
      if Keyboard.IsKeyDown(Keys.Left) then
        ((px, py), (sx, sy), r - dt)
      else
        ((px, py), (sx, sy), r)
```

It is important to notice that we like to hide the XNA code from the students, since they are not able to understand it. To do this, we encapsulate this code in a precompiled assembly written in C#, so that the students are be distracted by it.

Let us see another example. One of the last assignments of the course was about lists. An excerpt from the traditional set of exercises would be: "*Find the product of all the odd numbers in this list*". For the graphical assignment we used a car game (in which the car physics had been partially built by the students during the previous assignments) with little 2d circles instead of car models. The purpose of the assignment was to implement the code to render the models.

In XNA, a Model can represent any kind of drawable entity. The Model class contains a collection of Meshes that represent single physical objects. Each ModelMesh can be moved independently, and some can be drawn while others are skipped. For instance, our car Model contained one ModelMesh for the body of the vehicle and one for the wheels. The ModelMesh class contains a collection of MeshParts that represent single graphics card draw calls and that contain sets of triangles that share the same material and vertex declaration. We preprocessed these for the students to make the access to this structure less object oriented and more functional, so that to render a model in XNA they only have to iterate the list of meshes in the model: for each mesh, they have to iterate the list of its effects and for each effect they have to set some parameters.

In the picture below, you can see how the game looked like after having correctly rendered the car models:

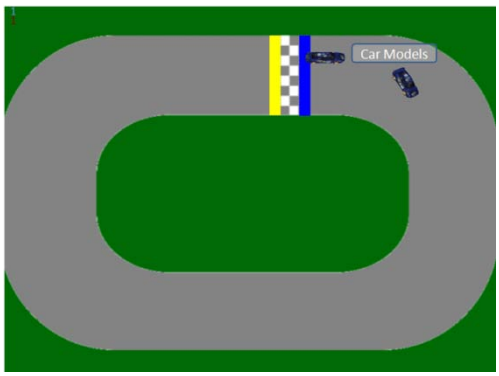


Figure 3: *Car game screenshot.*

The following is a sample of source code from the assignment:

```
let rec iterateMeshes modelMeshes =
  match modelMeshes with
  | [] -> ()
  | mesh::ms ->
        (iterateEffects mesh (meshEffects mesh))
        drawMesh mesh
        iterateMeshes ms
```

The function *iterateMeshes* iterates the list of the meshes contained in the model of the car and, for each mesh, iterates its effects (with the help of another function, iterateEffects) and draws it. Note the use of "pattern matching" to do elegant recursion by distinguishing between the empty list [] and a list with a head called "mesh" and a tail called "ms".

Again we encapsulated portions of XNA code in a precompiled assembly. This was especially useful when hiding the code that turned classes with a strong object oriented flavor into more functional constructs.

4. Learning by extending for an introductory course on Software Engineering

Let us consider the second case study, a Software Engineering introductory course at the third year of a CS bachelor program. The students, being seniors, mostly have the programming skills required to write an entire videogame; moreover, since the focus of the course is on software engineering methodologies and on team-working, we organized the class in teams of 4 or 5 students. Since the subject makes heavy use of design patterns and reusability constructs in general we have chosen the C# language because of its strong support of object orientation.

4.1 A structured project

In order to learn how to apply software engineering methodologies, we asked students to build a simple card game. Such a game typically does not have very high technological requirements but can be extended with interesting features like concurrency, parallelism, serialization to and from the disk (saving and loading the game) and extensibility.

We focused particularly on the concept of *extensibility*, as seen from two different points of view. On one hand, we offered students an AI for a specific card game (Machiavelli), and required them to build a system around this component. On the other hand, we required them to build their system so that it did not rely directly on our AI: their system had to be as reusable as possible with respect to the implementation of other card games. Menus, sounds, input management and drawing: all of these components must be independent from the type of card game played.

4.2 Technology background

Additional classes have been offered to introduce the technologies to be used: XNA, source control and web based collaboration sites. We started with basics on 2D rendering, input management, sound and code reusability in an XNA video game. We also explained the basics of source control and content management, and how having one's data in a remote server affects development.

To get students familiar with these notions we assigned a first task testing their degree of confidence with these technologies. We asked them to build a simple application focused on 2D rendering of textures and text and on the user's input management. The task had to be sent us through the source control system. Contrary to our expectations, some of the teams surprised us introducing features not explicitly requested by the task (and more importantly that we had not explained or showed or even mentioned): moving backgrounds, alpha blending, support for the XBox 360 gamepad, etc. Their efforts in the very first task showed us their degree of engagement.

To start tackling the true problem of the course, we decided that it was time to show how our AI worked and how to build a simple (non-XNA) application around it. Students were then fully ready to start analyzing the problem by knowing all the actors and the technologies involved. Without this knowledge they would have risked imagining a system too complex (or maybe even impossible) for them to build afterwards. After this phase of analysis they had clearly defined the problem they wanted to solve and they had produced a list of the requirements a good solution to this problem needed to fulfill.

Now that the students had defined what their system was supposed to do, it was time to focus on how the system was supposed to do it. In order to do so effectively, the students needed a deeper knowledge of how XNA could interact with design patterns and object orientation, and how to implement concurrency, parallelism and a plug-in system. In short, all those useful features that are never taught in class but always taken for granted when using many computer programs.

We started with code reuse: XNA offers a way to encapsulate a portion of the main loop of a game in a separate loop that runs alongside the main one. This secondary main loop is called a "Game Component", and it can expose its main functionalities through an interface called its "Service". This was more than enough for them to

achieve a very clean organization in subsystems and modules.

Secondly, we had to consider that the AI component of a card game could be slow, sometimes very slow (up to 10-15 seconds): no user can be expected to wait for an unresponsive program for so long. To prevent this unpleasant experience, we showed how they could solve the problem by computing the AI operations concurrently with the rest of the program. With C# and .Net concurrency almost becomes easy, thanks to the integration of locking constructs to make shared accesses simpler and thanks to threading libraries that implement many useful patterns.

Together with concurrency we showed how certain operations (mostly those done with LINQ, that is operations on collections) can be easily parallelized thus exploiting the multiple cores that so often are found in a modern PC or in the XBox 360.

Finally, we told our students about serialization, so that they could easily save and load complex data to disk without having to explicitly use files and IO.

Last but not least, we showed how reflection can allow us to load files (DLLs) that contain .Net assembly and classes. This is the core of a good plug-in system, because it allows the game to be extended to other card games without having to recompile anything beyond the new plug-ins.

In addition to defining the overall behavior and architecture our students were required to build a prototype for the user interface that their system will use. Here you can see a couple of UI prototypes produced by our students:

Figures 4 and 5: *User interface prototypes.*

In the last assignment the students have to fully implement the system they specified and designed in the previous tasks, and test it in according to the testing plan. Finally, the final grading is given on the basis of the quality of the whole documentation (analysis, design, test plan, code) and on the quality of the card game prototype.

4.3 Students' feedback

At the end of the project, the students were asked to answer a questionnaire about the course and the use of videogames as a case study. The results of the questionnaire are depicted in Table 1 (we show average scores based on the 44 students who answered the questions).

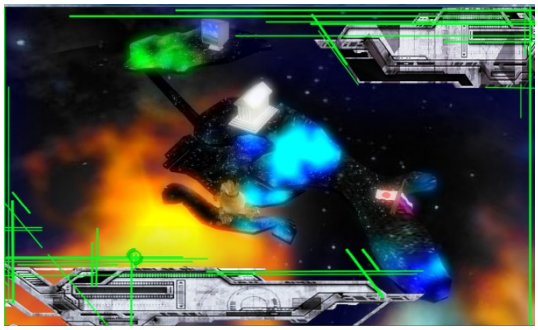
Table 1: *Questionnaire results*

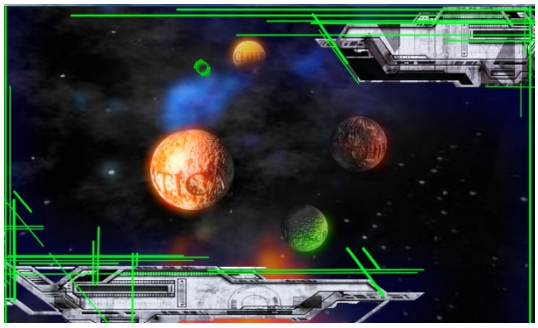
4.4 Possible follow-up

At the end of the course, we offer an interesting opportunity to the brightest students: to keep working on XNA and videogames for their final graduation report. This way, they can participate in the development of a larger videogame that allows the exploration of our degree in CS in a metaphorical virtual world. This is a project aiming at getting the player (usually a high school student) familiar with the CS curriculum like in a war, where each exam is a

battle that can be won by reading about the corresponding topics, thus conquering hordes of enemies each time.

Two screenshots of the current state of the game are here:





Figures 6 and 7: *Final project prototype.*

5. Conclusions

In this paper we analyzed the development of videogames as a useful tool to support the learning process about Programming and Software Engineering.

We explained what benefits both teachers and students can gain from this approach: students become more interested and engaged because videogames are exciting and the tools used are valuable knowledge in the IT industry; the teacher can keep focusing on the topics of the course but in a way that is quite more effective. The experience on the two courses involved an overall amount of 150 students.

Let us now consider the amount of work the teacher was required to do. In the first case, the teacher had to produce a small videogame; in the second one, the teacher needed to spend about three lessons talking about XNA and related concepts (concurrency, serialization, etc.). With a little background and preparation in XNA, these tasks can be accomplished with an effort that is quite small, especially compared to the quality of the results.

References

[BH07] BOX D., HEJLSBERG A.: LINQ: .NET Language-Integrated Query (February 2007) http://msdn.microsoft.com/en-us/library/bb308959.aspx

[CC01] Curriculum Guidelines for Undergraduate Degree Programs in Computer Science, http://www.acm.org/education/education/education/curric_vols/cc2001.pdf

[C#LS] C# Language Specification, http://msdn.microsoft.com/en-us/library/ms228593.aspx

[GS02] GUZDIAL M., SOLOWAY E.: Teaching the Nintendo generation to program. *Commun. ACM, 45(4)* (2002)

[MG07] MARAGOS K., GRIGORIADOU M.: Designing an Educational Online Multiplayer Game for learning Programming, *Proceedings of IEEII 2007, Thessaloniki, Greece* 322-331 (2007)

[Mos97] MOSER R.: A fantasy adventure game as a learning environment: Why learning to program is so difficult and what can be done about it. *Proc. Of ACM ITiCSE 97, Uppsala, Sweden* 114-116 (1997)

[Pic07] PICKERING R.: Foundations of F#. *Apress* (2007)

[PM06] PEDRONI M., MEYER B.: The Inverted Curriculum in Practice. *Proceedings of SIGCSE 2006*, *ACM, Houston, Texas* (March 2006)

[Pre03] PRENSKY M.: Digital Game-Based Learning. *Computers in Entertainment, vol. 1(1)* (2003)

[SGC07] SYME D., GRANICZ A., CISTERNINO A.: Expert F#. *Apress* (2007)

[Whi84] WHITE B.Y.: Designing computer games to help physics students understand Newton's laws of motion. *Cognition and instruction, 1*, 1, 69-108 (1984)

[XNA] XNA Official Site, http://creators.xna.com/en-US