Engaging High School Students in Computer Science via Challenging Applications

Giuseppe Maggiore Andrea Torsello Flavio Sartoretto
Agostino Cortesi
Università Ca' Foscari Venezia
Dipartimento di Scienze Ambientali,
Informatica e Statistica
{maggiore, torsello, sartoretto, cortesi}@dais.unive.it

ABSTRACT

In this paper we describe a general framework for building short-courses designed to engage student while presenting a sub-field of computer science. We also describe two of these short-courses centered around computer graphics and physical simulations.

We will discuss how even beginner students can participate in our short-courses; this is possible thanks to a careful choice of development environment, programming language and libraries that allow the students to focus on solving the problems and not thinking about low-level details.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Human factors, languages

Keywords

education, computer science, games

1. INTRODUCTION

Math is beauty, engineering is challenge, Computer Science is both...plus fun! However, it's becoming more and more difficult to get high school students enrolled in CS university programmes. One reason is that this scientific discipline is widely confused with its technological applications. However, we believe that there is also another reason: the way Computer Science, and in particular Computer Programming is presented to beginners. One of the greatest features of Computer Science is stressed too little: the fact that even the most complex problems can be expressed and solved by engaging, visual applications, and that most theory in Computer Science arises from intuitive and fascinating problems.

In this paper we report the results of a project run at our university with the final aim of getting high school students to look at Computer Science as "beautiful, challenging and fun". We describe how we built a template around an original pedagogical process for creating short-courses that can be taught in one day; these short-courses offer high school students the opportunity to experiment with a complex application of Computer Science in a visual and engaging environment. Our short-courses do not aim at making Computer Science appear as just "fun and games". Rather, we wish to offer students noise-free environments in which to do practical experiments (so no complex APIs or too technical details) while still maintaining rigor and theoretical soundness. In this paper we focus on two short-courses about computer graphics and physical simulations; in these courses students learn some of the equations that model the appearance and the motion of physical objects, and they learn how to translate those equations into (functional) code that the computer understands. The code that students write is used for rendering a scene where the students can immediately receive feedback to see if they have implemented everything correctly or not.

In Section 2 we describe the general template around which our short-courses are built. In Sections 3 and 4 we describe two of these short-courses: computer graphics and physics simulation; we will show a large portion of the actual code we used in the classroom. In Section 5 we report the feedback and the rating that we received from the first batch of students that experimented with these short-courses.

Related Work.

The study of how to better teach Computer Science is growing by the year. A very large part of these studies is centered on how to teach Computer Science by insisting heavily on visual, entertaining applications ([13], [10], [5] and many others) that more closely resemble the students' experience with modern applications [11]. These programs are all built to match the expectations that young people have when they start studying programming: they want to build web sites, games and interesting applications. Like these authors, we wish to offer beginner students an environment where they can learn complex tasks (like programming complex algorithms or equations) with the same visual appeal that they have come to expect from computer programs.

Most authors ([12], [9], [8], [14], [15]) have observed a

SIGITE'11, October 20–22, 2011, West Point, New York, USA.

large increase in the effort made by students to complete their game-themed assignments when compared with the lack of interest of traditional (shell-based) assignments; we have measured a similar outcome during our initiative.

In [7] and [6] the authors have experimented with building a curriculum that can be used from middle- to high-schools and even in a university introductory course. These studies, together with [4], have all used functional languages like we have done in our approach, believing that the syntactic and semantic cleanliness of these languages is a large bonus when compared to more complex imperative computational runtimes, especially when put in the hands of programming novices.

Downloading and Testing.

We have made our implementations available for download at the following URLs:

- the computer graphics source can be downloaded at `www.dsi.unive.it/~grafica/PLS`
- the physics simulation source can be downloaded at `www.dsi.unive.it/~sartoret/SistemaDinamico.zip`

2. A GENERAL TEMPLATE

In this section we discuss the general template around which we have structured our short-courses. Before presenting the template, though, we must briefly discuss the purpose of our initiative; in particular, our goals are:

- interesting students with engaging applications, not just theory
- showing students real CS tools and techniques for solving actual problems
- simplifying the problems (to make them solvable in a short time) but without dumbing them down; we *must not* give the impression that CS deals with trivial matters and is somewhat "inferior" to maths or engineering

Our template for short-courses requires following these steps:

1. we choose a concrete field of research or application, such as computer graphics, physics simulations, artificial intelligence, security, computer vision, and many others. Any field which has at least one interesting practical application is suitable for our framework;
2. we pick an aspect that is relevant to the field; this aspect can be anything from the accuracy of the simulation to the security of a computer network to the "smartness" of an AI, etc. as long as it is easy to visually compare different solutions to see which one is better;
3. we pick a sequence of algorithms, equations or techniques that are designed to improve the parameter (2) with various degrees of complexity and quality of the solution. Ideally, this sequence should be a chain where each algorithm improves the previous one by adding some complexity, that is each step should be strongly linked to its predecessor and successor;
4. we build a framework which hides all the implementation details that are not strictly relevant to building the various algorithms of (3); this framework should also implement the visualization system. The framework must be built so that the various algorithms of (3) can be plugged in it by the students, if possible even more than one at a time to directly compare them
5. we build the first algorithm of (3) in our system and we document it fully to offer a working starting point to the students
6. we discuss and explain (with a mix of frontal lessons and notes) the underlying theory of the various algorithms of (3) and let the students translate that theory into code that works with the framework (4)

The choice of the actual tools used to implement this template is an important one. There are many valid alternatives, but we believe that the environment used must fulfill the following requirements:

- it must support advanced rendering and visualization
- it must support syntactically simple programming languages
- it should offer auto-completion and helpful error messages (a need eloquently motivated by [6])

We have used Microsoft Visual Studio 2010 [2] with F# [1] and XNA [3]. F# is a functional programming language that belongs to the .Net framework and which can access an extremely large number of mature libraries that can be used for our purposes; F# is a dialect of the ML family, one of the three large families of functional languages (the other two being LISP and Haskell). It is important to take notice that functional languages are not a requirement of our template: scripting languages such as LUA or Python could arguably be used given their terseness. XNA is a computer graphics library that allows both high-level coding of games and simulations and which also allows in-depth access to the GPU to implement advanced rendering algorithms. XNA has been used in another experimental teaching initiative, as described in [14]: we report similar experiences to theirs, namely that XNA is a powerful framework for teaching that allows students to build interactive applications that can be at any point of the spectrum that goes from simple to program to very complex but visually stunning.

Using these powerful tools has a small downside: some time must be taken, before asking students to operate these tools, to learning the bases of the chosen programming language and environment. F# offers an immediate console which allows students to experiment with the basic language constructs as the instructor explains them, in order to quickly come to terms with the language.

3. COMPUTER GRAPHICS

The main objective of the Computer Graphics short-course is to show students how to compute a realistic appearance for some mesh (3D geometry composed of triangles) through shaders that simulate the visual properties of different materials.

The students are given a starting shader, which is a small program that is run by the GPU and which is responsible for

computing the position of the vertices of each triangle and for computing the color of each pixel of the various triangles.

The general shape of a shader is the following (PARAMETERS, VERTEX_SHADER and PIXEL_SHADER are just place holders for other code):

```
let shader =
<@@
  PARAMETERS

  VERTEX_SHADER

  PIXEL_SHADER

  in ()
@@>
```

The shader parameters are a set of global variables that are set to the GPU memory and which store properties of the scene such as the position and color of lights, transformation matrices, etc. The simplest parameters for a 3D scene are provided in the initial shader, and are three transformation matrices:

```
let World          = parameter() : Matrix
let View           = parameter() : Matrix
let Projection     = parameter() : Matrix
```

The students are also given a list of all the global variables that are supported by our system.

A vertex shader takes as input a vertex (which stores the position of the vertex plus optional additional attributes such as its color, its normal, etc.) and returns the transformed vertex (which again contains at least the transformed position plus optional additional attributes). The sample vertex shader simply transforms the input position from 3D space to screen space:

```
  let vertex_shader (InputPosition(pos)) =
    let worldPosition = pos * World
    let viewPosition  = worldPosition *
                        View
    let pos'          = viewPosition *
                        Projection

    in OutputPosition(pos')
```

A pixel shader takes as input a subset of the attributes of a transformed vertex and returns the color of its pixels; the sample pixel shader returns a uniform red color:

```
  let pixel_shader () =
    OutputColor(Vector4(1f,0f,0f,1f))
```

Running the initial configuration simply draws a red mesh. Starting from this the students add the normal to the vertex shader input and compute first Lambert lighting and then Phong lighting with a specular component. At this point a color texture is added and read with the input texture coordinates found in the vertex shader; by combining the texturing shader and the Phong shader results a shader capable of showing fine details and lighting (Figure 1). At this point the students replace the background texture with a cubemap which gives the illusion of a more detailed background. The final results are obtained by computing the reflected view direction with respect to the normal of the mesh in order to look up the background, thereby giving the illusion of a reflective surface; by perturbing the normals of the surface in an animated fashion the last shader can be used to show a very pleasant-looking reflective water surface (Figure 2).

4. PHYSICS SIMULATION

The main objective of the Physics Simulation short-course is to show students how to model simple discrete systems such as a bouncing ball.

The students start with an initial definition of two integrators. The first integrator is loaded from a precomputed file, and it offers an accurate simulation of the real behavior of the system. This first integrator acts as a benchmark. The second integrator simulates simple linear motion. An integrator is specified by giving an instance of the `Integrator` data-type:

```
// ball = velocity, position
type Ball = float * float

// time = total, delta
type Time = float * float

type Integrator =
  {
    // initial position of the ball y0
    y0 : float
    // initial velocity of the ball v0
    v0 : float
    // step-integrator function
    update : Time * Ball -> Ball
    // display name
    name : string
  }
```

The simulation environment takes as input a sequence of integrators to display side-by-side for comparison; the initial sequence the students have is composed of an instance of the precomputed integrator together with the simple linear motion integrator:

```
let actual = { y0 = FromFile.y0;
               v0 = FromFile.v0;
               update = FromFile.step;
               name = "Exact" }
let linear = { y0 = Linear.y0;
               v0 = Linear.v0;
               update = Linear.step;
               name = "Linear1" }

let integrators =
 seq{
   yield actual
   yield linear
 }
```

The students can inspect and modify the code from the linear integrator, which is:

```
let y0 = 10.0
let v0 = 15.0

let step ((t,h),(v,y)) = (v,y + v * h)
```

In particular from this integrator the students can see that the `step` function takes as input the current time `t`, the step length `h`, the ball velocity and position `(v,y)` at time `t` and returns the new velocity and position of the ball at time `t+h`.

Running the initial configuration clearly shows that the linear integrator is not an accurate simulation of the benchmark. The students then build a forward Euler integrator which is accurate only during the first seconds of the simulation and which soon starts diverging. They soon realize that the problem is that such a simple, intuitive integrator suffers from unsightly infinite oscillations (the so called *Zeno phenomenon*). Such oscillatory phenomena can be tamed (but never fully corrected) by either cutting the velocity after a while or when they become too small (a very ad-hoc solution) or by using a more refined integrator which suffers from much smaller oscillations. By implementing the Ralston integrator (also known as Second Order Runge-Kutta integrator) the simulation starts to look very accurate (indeed errors and oscillations are smaller than one pixel and cannot be perceived with the naked eye, as seen in the video linked at Figure 3).

5. FEEDBACK AND RESULTS

We have tried to assess the results of both programs according to the following indicators:

- perceived satisfaction
- timely completion of the exercises
- increase in enrollment

Given our original purpose of stimulating interest in Computer Science we are mostly interested in the first item, student satisfaction. Given that we also wished to offer students an active and engaging experience, we also set out to monitor how many students completed our exercises and to what degree. Finally, we wish to monitor the effectiveness of our approach in getting more students enrolled in our degree program.

Unfortunately, we still do not have data about the effectiveness of this year's initiative in terms of student enrollment, given that the next enrollment period has not started yet. We can report as preliminary results the fact that this year we witnessed a small increase in enrollment and we had experimented for the first time with an initiative that was similar to this one (indeed, last year we tried what was the pilot program for this year's initiative). Hopefully this trend will be confirmed in a few months, and in a few years we will have enough data to be of statistical significance.

Computer Graphics.

The computer graphics results have been quite good. As mentioned previously all the students completed their assignments in time. Being the computer graphics assignments both complex *and* time consuming, we were very surprised that all the students managed to work their way through all exercises, even the harder ones. Indeed, we had a "plan B" that consisted in showing the correct results after each exercise if too many students had not been able to complete it, in order to level the class so that it could keep moving at a steady pace; such plan was never needed!

The number of students that participated in this course was 31. A questionnaire was created that contained 8 true/-false questions about the topics; a few sample questions (we do not include them all for reasons of space) were:

- "normals must be transformed into world space before lighting"
- "Phong lighting is more accurate than Lambert lighting but is more computationally expensive"
- "texture filtering is needed for sampling in-between texels"

The resulting error percentages are:

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
13%	0%	7%	3%	7%	3%	7%	20%

Table 1: Comprehensions test

As we can easily see, on average questions scored more than 90% right answers. The students' perceived satisfaction was measured with another questionnaire, which asked questions about each their experience:

Glad of having participated	Clarity of teachers	Teaching method	Usefulness of notions	General satisfaction
83%	64%	75%	74%	75%

Table 2: Student satisfaction

Students are very satisfied of having participated in the experience, even though they found the frontal parts of the lesson too hard. In general the results are good, and we believe that by simplifying a bit the exercises and by spending more time on the background (vectors and trigonometry) these scores could become higher. It is interesting to notice that we also received enthusiastic comments from some students, who wrote further feedback such as *An amazing experience: it shows an aspect of Computer Science I did not believe existed*; many students observed that through this kind of course they came to understand that Computer Science is richer and more complex than they believed.

Physics Simulation.

In a manner similar to that used with the computer graphics short-course we have tried assessing the results of the second short-course. In this case we only administered a final questionnaire about the students satisfaction, and not about their understanding of the matters taught. All students completed all their tasks, and in *far less time* than we had anticipated.

11 students participated in this course. They reported their satisfaction as:

Interesting	Fun	Engaging	Boring	Uninteresting
82%	36%	27%	0%	0%

Table 3: General satisfaction

Too hard	Hard	Ok	Easy	Too easy
0%	9%	82%	0%	0%

Table 4: Difficulty

New concepts	Concepts I already knew	New way to study concepts I already knew	Nothing new for me
64%	0%	36%	0%

Table 5: Novelty

As we can see, this second course obtained very high ratings. All students described the course as either fun, interesting, engaging; all but one student found the difficulty appropriate (neither too easy nor too hard); finally, all students felt that we taught them something new, either new concepts or new approaches to concepts they knew already.

In addition two students wrote that they really enjoyed the introductory portions which had been somewhat lacking in the computer graphics short-course and that they really liked the opportunity to experiment theoretical concepts in a programming setting.

As a final notice, we wish to express satisfaction from the fact that we were able to teach differential equations without the usual level of students' stress. Differential equations are often considered a benchmark of obscurity, difficulty and boredom, and we are glad we were able to show them as beautiful mathematical concepts which elegantly capture the essence of important real-world phenomena.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a very applicative, hands-on approach for teaching complex topics in computer science. We have designed a set of short-courses that can be completed by high school students in at most 8 hours, each short-course focusing on an advanced application of computer science In the paper we have focused on two short-courses in particular, even though other similar short-courses achieved comparable results.

In all short-courses the first step consists in explaining the problem tackled in simple, intuitive (yet rigorous) terms. Then the required mathematical background is given, always focusing on its relationship with the problem and not as important "per se". A brief introduction to the development tools and languages is given; we used languages with little syntactic and semantic complexity such as Python or F#, in order to avoid explaining things like scope and brackets which can be confusing for beginners. The tools and frameworks offered to students must be tweaked to the point that there is virtually no distraction from the main task: the students are required to write only the minimum code necessary to solve the problem at hand and nothing more.

All of our students have solved all the exercises in the given time, even the most complex: the students' good feedback shows that even though there is room for improvement this kind of initiative leaves them happy from a pleasant learning experience; considering the fact that the topics are traditionally considered very hard and boring (algebra, geometry and differential equations) this is a very notable achievement.

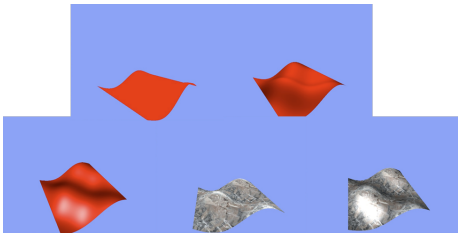

Figure 1: Uniform Shading, Lambert and Phong Lighting and Texturing

Figure 2: Reflection effects

This work is by no means complete. This paper has described the second step in a larger initiative that started last year and that we hope to continuously improve. Last year we created the first few short-courses, while this year we defined a more general template to create more short-courses all based on the same working formula. This initiative is resulting in stronger relationships with the high schools surrounding our university, and given the initial success we will try to expand our offering. On one hand we will create more and more short-courses on other topics such as game or mobile development, while on the other hand we will experiment with longer courses that take more than one day to complete.

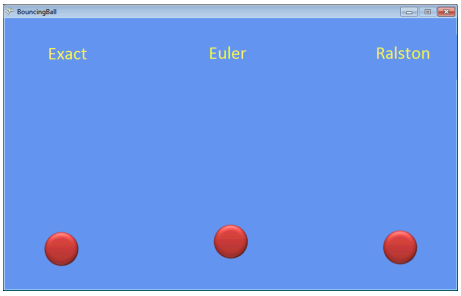

Figure 3: Video found at: http://www.dsi.unive.it/~orienta/simula/BouncingBall.wmv

References

[1] F#. http://msdn.microsoft.com/en-us/library/dd233154.aspx.

[2] Visual studio. http://msdn.microsoft.com/en-US/library/52f3sw5c.aspx.

[3] Xna on msdn. http://msdn.microsoft.com/en-us/library/bb200104.aspx.

[4] Manuel M. T. Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(01):113–123, 2004.

[5] M. Feldgen and O. Clua. Games as a motivation for freshman students learn programming. *Frontiers in Education, 2004. FIE 2004. 34th Annual.*

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(04):365–378, 2004.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional i/o system or, fun for freshman kids. *SIGPLAN Not.*, 44:47–58, August 2009.

[8] Michael Kölling and Poul Henriksen. Game programming in introductory courses with direct state manipulation. *SIGCSE Bull.*, 37:59–63, June 2005.

[9] Scott Leutenegger and Jeffrey Edgington. A games first approach to teaching introductory programming. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 115–118, New York, NY, USA, 2007. ACM.

[10] Maic Masuch and Lennart Nacke. Power and peril of teaching game programming. In Norman E. Gough and Quasim Mehdi, editors, *International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 347–351, Reading, UK, November 2004. University of Wolverhampton UK.

[11] M. Overmars. Teaching computer science through game design. *Computer*.

[12] Yolanda Rankin, Amy Gooch, and Bruce Gooch. The impact of game design on students' interest in cs. In *Proceedings of the 3rd international conference on Game development in computer science education*, GDCSE '08, pages 31–35, New York, NY, USA, 2008. ACM.

[13] Kelvin Sung. Computer games and traditional cs courses. *Commun. ACM*, 52:74–78, December 2009.

[14] Kelvin Sung, Michael Panitz, Scott Wallace, Ruth Anderson, and John Nordlinger. Game-themed programming assignments: the faculty perspective. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 300–304, New York, NY, USA, 2008. ACM.

[15] Kelvin Sung, Rebecca Rosenberg, Michael Panitz, and Ruth Anderson. Assessing game-themed programming assignments for cs1/2 courses. In *Proceedings of the 3rd international conference on Game development in computer science education*, GDCSE '08, pages 51–55, New York, NY, USA, 2008. ACM.