A Recipe for Game Development Assignments in CS2

Jude Allred
Under the direction of Professor Michael Main, Professor Clayton Lewis
Department of Computer Science, University of Colorado, Boulder
jude.allred@colorado.edu

Abstract

CS2 classes suffer from a trend of dropping enrollment, poor gender diversity, the challenge of students learning a secondary programming language, and difficulties related to inadequate (or varied) preparation provided by CS1. Surveying educational research has indicated that using game development as the focus for assignments can mitigate many of the challenges of teaching CS1. This paper has extracted many of the key beneficial components of game development assignments in CS1 and provides them as a recipe for beneficially augmenting CS2 with game development assignments. It is expected that all CS2 courses can be enhanced by applying the elements presented within this recipe.

1 Introduction

This paper provides a recipe for using game-oriented CS2 assignments. (section 3)

Research has shown that students in a CS1 course with a game development focus have significantly enhanced experiences relative to a non-game-oriented baseline: peer intimidation drops dramatically and they average a letter grade higher than control groups. (1) There are many studies which have applied game development towards CS education, and nearly all of them agree that game development provides for a concrete, accessible, and engaging context for computer science learning. (2) Although there have been many studies which have succeeded in using game development to revitalize CS1 courses (1)(2), there has been relatively little effort in applying game development to CS2. CS2 is still a bottleneck for many students, but game-oriented approaches have shown promising results. (3)

2 Background

Many papers have been written over the past several years which address topics within computer science education, the challenges it faces, and both hypothetical and well-tested methods of improving its prospects. There is sufficient research in the field to justify assorted conclusions about computer science education and to remark on specific educational needs. These data can be used to formulate categories for the challenges relating to CS education and make clear statements about simple changes which can be effected in order to produce substantial results.

2.1 General Difficulties

The quality of computer science education is suffering from a trend of dropping enrollment, poor gender diversity, and a great deal of infighting and inconsistencies relating to what pedagogical techniques are most appropriate. (2)(4)(5)

CS1 and CS2, as defined by the ACM(6), are the focus of the bulk of the educational research in the field. These foundation classes are experiencing severe dropout rates and gender diversity issues(2)(4), and are further complicated because they are frequently offered to computer science non-majors as their introduction and often entire education within computer science.

2.2 Drawing a Parallel

No Silver Bullet(7) is a software engineering article which defines the essential and accidental difficulties within software engineering. Essential difficulties are those inherent in the field, while accidental difficulties are implementation-bound. Attacking the essence is a challenge, but the accidents can be methodically minimized. A parallel can be drawn with computer science education. Essential difficulties within computer science education include the task of conveying knowledge and skills to a student, the amount of practice required by a student before they can master a skill, the process of conveying mental models and constructs which the students can use to understand the material, conveying the concepts of object orientation or imperative program flow, etc. Accidental difficulties are therefore bound to implementation and circumstance: Computer Science does not appeal to female students (2)(4)(5), the homework time is spent disproportionately on debugging(8), students feel intimidated by CS majors when taking CS classes(4), the chosen language has convoluted syntax, etc.

2.3 Defining CS1 and CS2

CS1 and CS2 derive from a set of curricular models proposed within the 2001 Computing Curricula Final Report, as set forth by the ACM(6). CS1 and CS2 are intended to function as the first two introductory courses for the field of computer science.

2.4 Implementation Strategies for CS1 and CS2

The Computing Curricula Final Report proposes six potential implementation strategies for these introductory courses: Imperative-first; Objects-first; Functional-first; Breadth-first; Algorithms-first; Hardware-first. (6) The Imperative-first strategy is the most common implementation choice, followed by Objects-first. The other implementation strategies listed were not frequently mentioned or addressed within the literature surveyed. In addition to these six basic strategies, some promising new strategies have been proposed: Components-first and Games-first.

2.4.1 Imperative-First

The imperative-first approach is the most traditional model for teaching CS1 and CS2. (6) ACM's computing curricula notes this approach as having significant disadvantages in the context of eventually teaching object orientation. The curricula further states that application of the imperative-first approach will require additional training for students in object-oriented programming at an intermediate level. A common argument for the imperative-first approach is that by avoiding the added complexity of object-oriented syntax, students are able to begin programming earlier. Since programming is a key skill which requires a lot of practice, introducing it early is beneficial. (6)

Stuart Reges provides evidence in favor of the imperative-first approach, but advocates using Java. (5) Reges found that teaching object-oriented principles early was proving to be too difficult for students in CS1, and CS2 suffered as a result. In an effort to stabilize his university's failing curriculum, Reges reverted CS1 to the imperative-first approach, but adopted Java as the course language. The primary motivation for using Java was to give students experience with the language prior to needing it in CS2. The Java code written in his CS1 class was primarily procedural Java code via static functions. Reges admits to the complexity overhead in using Java, but notes that his students did not seem to mind it. The procedural Java CS1 course resulted in a significant improvement in student feedback relative to both the previous object-oriented Java CS1 course and the procedural C CS1 course.

The imperative-first methodology has set the par for student experience and comprehension within CS1/CS2.

2.4.2 Objects-First

The Computing Curricula Final Report's objects-first model is intended to emphasize object-oriented programming and design immediately, and teach control structures and programming practices as secondary topics motivated by their need within object-oriented designs. The report cites the complexity of object-oriented languages, such as C++ and Java, as a chief disadvantage to this approach.

A particularly insightful case study of implementing an objects-first approach is documented in Object Orientation in CS1-CS2 by Design. (9) In iterating upon attempts to build an object-oriented CS1/CS2 sequence, they came to discover that many of the existing approaches and supporting textbooks for an object-oriented introductory class, while teaching object-oriented concepts, did not approach the material in an object-oriented way: procedural programming is taught, and then object-oriented concepts are built upon it. Iterating further, they found that introducing object orientation early was superior, however the point at which students transition from procedural to object-oriented methodologies persisted as a difficulty. They then moved to a substantially-more-successful method which involves postponing procedural programming topics until after object orientation had been presented. In spite of concerns that the transition from object-oriented to procedural code would be as difficult as the inverse, this method resulted in substantially increased student comprehension. They hypothesize that making object orientation a student's first experience with computer science causes them to have "objects on the brain" – object models become their intuitive structure for thinking about computer science, and other computer science topics, such as procedural programming, are motivated by their necessities within object-oriented code. They also hypothesize that when students learn procedural programming methods first, they have trouble understanding the motivation for object orientation and view the increased syntax complexity as an unjustified burden.

Goldwasser and Letscher also provide evidence in favor of an objects-first approach to CS1. (10) Their approach is noteworthy because they apply and recommend Python as a programming language for CS1.

2.4.3 Components-First

Components-first introductory computer science classes are intended to focus on the libraries, API's, and other common component infrastructures which are in common use by the software engineering profession.(11) Components-first approaches are highly pragmatic, and equip students with the ability to compose software applications from existing components.

2.4.3.1 Key Elements

Two independent components-first approaches are surveyed within Components-First Approaches to CS1/CS2: Principles and Practice(11), and some key elements to the components-first approach are established:

2.4.3.1.1 Client-View-First Pedagogy

Students are taught to understand components not by studying their implementations, but instead by studying their interfaces. Students are treated as clients seeking a necessary component, are provided said component, and must evaluate its usefulness and abilities based on how they can work with the interface. Only after the interface has been sufficiently motivated and applied do students switch roles to that of the implementers, and now must create the underlying structure which fulfills the interface.

2.4.3.1.2 Pointers

By delaying implementation of underlying component classes, the necessity of teaching pointers is also delayed. This delay allows students more time to become comfortable with their programming language and its debugging techniques prior to studying pointers. The delay can also be used to motivate the fact that the primitive pointer-based data structures are frequently available through component libraries.

2.4.3.1.3 Program Complexity

Because students are working with existing component libraries, their assignments can more easily be tailored to creating useful, complex software. This tends to dispel notions that the techniques they learn are only applied in "toy" programs.

2.4.3.1.4 Data Types

Introduction of the array data type is delayed. This provides students with the ability to learn data types at a higher level of abstraction and focus on common data type manipulation techniques such as iteration and recursion. Arrays are introduced later not as a commonly-used data type, but rather as a data type which is motivated by its performance properties.

2.4.3.2 Legitimacy and Implementation

Howe *et. al.* conclude that the components-first approach is a legitimate approach to CS1 and CS2. Reflecting on the difficulty and perceived resistance to switching an existing CS1/CS2 sequence to a components-first methodology, they assert adopting Client-View-First pedagogy would be of significant benefit for any object-oriented introductory computer science course.

2.4.4 Games-First

There has been a recent focus on game development as a pedagogical tool for Computer Science. This is perhaps because not only has everyone played games and can relate to them, but there are now more resources than ever before to facilitate game creation. (12) Some argue that motivation is key to learning, and that game development provides motivation in addition to promoting collaboration, socialization, and critical thinking. (13) Other arguments focus on the increased enthusiasm and interest provided by game-centered courses and how these augmentations have been found to improve the performance of low-end students. (14)

Scott Leutenegger and Jeffrey Edgington of the University of Denver argue for a *Games-First* approach to introductory programming classes. (2) Postulating that the concerns over imperative-first versus objects-first are less important than the types of assignments and examples provided to their students, they constructed a 2D-game-oriented CS1 course taught using Actionscript/Flash and C++/OpenGL. Having merely refocused the course content to game development, but still teaching and testing on their standard technical content, Leutenegger and Edgington report a comprehension increase in all of their course topics, as well as increased student retention and substantially increased positive feedback from their female students.

2.5 Accidents of CS1 versus CS2

Although they are commonly spoken of jointly, there are several specific differences between CS1 and CS2 which give rise to distinct accidental difficulties for each course.

2.5.1 Accidental Difficulties of CS1

CS1 was not constructed with the intention of functioning as a stand-alone course capable of preparing students to program professionally. (4) (6) (15) As computers become more and more a part of everyday life, the field of computer science is becoming ever more interdisciplinary. Unlike in most other engineering disciplines, there is now an expectation of achieving base-level proficiency after taking a single introductory computer science course. (15)

While it is desirable to construct a CS1 course which most benefits students who progress through the computer science major, it is now also a necessity to cater to students who expect to be able to produce meaningful software after having taken only CS1. It is thus a challenge to produce meaningful course material which caters to the diverse population of students who desire basic computer science training.

One solution is to provide different CS1 courses to different student demographics. (4) A simple division is to split CS1 based on whether or not the students in attendance are computer science majors. The non-majors section, then, would be able to focus on simple and pragmatic aspects of computer programming, such as writing simple scripts and applications, while the majors section could devote more time building skills that will be necessary in future computer science courses. Further division may also be relevant- provide CS1 courses for students specifically interested in image and video manipulation, web development, audio effects and manipulation, etc. While this approach is likely to be beneficial (4), it sidesteps the problem of students being dissatisfied with the content of existing CS1 courses.

Another approach is to attempt to teach CS1 in an inherently pragmatic fashion. The components-first methodology (section 2.4.3) is an example of an implementation technique which provides this. (11) A smaller change which can make CS1 inherently more pragmatic is to teach the course using an inherently more pragmatic language. Scripting languages, especially Python, are considered especially well-suited to this task because of their low overhead in creating simple applications. (4)(16)

2.5.2 Accidental Difficulties of CS2

As CS1 and CS2 are designed to be taught in a sequence (6), CS2 is inherently dependent on CS1 succeeding in conveying necessary prerequisites.

2.5.2.1 Language

CS2 courses usually incorporate more advanced language features than are covered in CS1. Further, it can be the case that the programming language students were taught in CS1 is different from the language being used in CS2. Unfortunately, students cannot learn a second language in a primarily independent fashion (15), and therefore the CS2 course must either ensure that its students are above a minimum skill level with the language to be used, or part of the CS2 course must be focused on teaching the language required.

While this may suggest a set of standards to which CS1 must conform (such as teaching the language that will be employed in CS2), it is not the case that the language chosen in CS1 affects student performance in CS2: The programming language used to teach CS1 does not have a statistically significant impact on the performance of students in CS2. (16) (17) Further, one study finds that the paradigm (procedural versus object-oriented) chosen for teaching CS1 does not have a statistically significant effect on student performance in an object-oriented CS2 class (17).

2.5.2.2 Inadequate Preparation

It can also be the case that students entering CS2 are not adequately prepared to be taught the material. This issue is unavoidable, as CS2 students may have taken different CS1 classes, or no CS1 class at all. One promising approach to this difficulty is to provide a bridge course between CS1 and CS2.

To address the issue of many of their students not possessing an adequate mastery of CS1 material in their CS2 courses, and fed by the common student complaint that the examples used in CS1 and CS2 were abstract and non-compelling, Scott Leutenegger of the University of Denver developed a game-oriented CS1 to CS2 bridge class. (18) The goals of this class were to "solidify CS1 concepts, provide concrete examples rather than abstractions, add some new topics, motivate the need for CS2, and offer a class that is fun for most students." The course was taught using Actionscript/Flash. Anecdotally, this class appeared to be of major benefit to CS students.

It may be that CS1 is simply insufficient preparation for a significant number of students who are entering CS2. If this is the case, teaching a bridge course, such as the one described above, could have a very substantial impact on student performance in CS2. It seems reasonable to claim that developing and offering a meaningful bridge course is a viable alternative to reconstructing and optimizing a CS2 course, especially considering that both CS1 and CS2 may be taught differently by different instructors, and therefore guaranteeing their compatibility would be impossible.

Aside: *As students are typically not allowed into CS2 without either CS1 or other programming experience, data on the performance of students who are introduced to computer science via CS2 is lacking. It would be a wonderful sanity check to be able to know how much of an effect CS1 has on student performance in CS2.*

2.6 The Role of Teaching Assistants, Recitations, and Labs

Another accidental difficulty of computer science education is quality control over the TA's and lecturers who interact with students. Many CS1 and CS2 courses include a computer lab or recitation component, and this component of the class is often neglected during curricular innovation of CS1/CS2. This neglect is odd, given that labs are a key source of hands-on exposure to course content and that an effective laboratory experience can free up lecture time to cover more advanced topics. (19) Indeed, only one of the papers surveyed attempted to integrate new lab and recitation techniques with their experimentation on course experience. (20) Presented here are two novel approaches toward enhancing the experience of labs and recitations, one of which is not a curricular development but rather a technique which is supportive of more rapid course innovation.

2.6.1 Students as Presenters

It is difficult to rapidly develop the content of CS1 and CS2 courses because as the courses develop and follow new approaches, so must all of the supporting faculty, staff, TA's, etc. To address this issue, Robbins *et. al.* propose the use of students as presenters within CS1 and CS2 laboratory sessions. (19)

Robbins *et. al.* feared for the quality of their students' laboratory experiences. As curricular changes were made and as the software used in labs became more sophisticated, teaching assistants have to spend increasing amounts of time troubleshooting software issues and otherwise managing lab affairs. An additional difficulty arises in assuring the quality and competency of TA's, especially as material evolves: it is too much to expect the TA's to grade, teach, run the lab, and also have to learn course material at a faster pace than the students. The proposed solution was to have TA's drop back to a supportive role of smoothing out the technical issues that arise during the laboratory sessions and working as graders. To replace the teaching role of the TA's, student presenters would be hired from the pool of students who excelled in a previous iteration of the course. These paid presenters would be responsible for teaching a laboratory session on a specific topic. Because these are lab sessions for CS1 and CS2, the student presenters who receive a 4-year degree in computer science will have the three years following their initial participation with the course to iterate on and enhance their presentations.

With this model, changing the course content does not require retraining TA's but just selecting new student presenters. Since a new pool of candidate student presenters is supplied after every course offering, new student presenters can quickly be recruited to cover new course topics.

After implementing student presenters for their CS2 class, Robbins *et. al.* surveyed students and asked them to compare their experiences with the CS2 labs with their past laboratory experiences (such as CS1 labs). The response to having the student presenters was overwhelmingly positive, and they were especially grateful for having two teachers available in the laboratory (the student presenter and the supporting TA) because it allowed the TA to answer questions on an individual basis while the presenter could progress with material.

A downside to this approach is the cost related to hiring the student presenters. For each presentation, the students were paid two hours of preparation. The total expenditure of Robbins *et. al.*'s experiment came to $12,000 per semester, but also included the salaries of student tutors who staffed the lab 105 hours per week (14 hours per day).

2.6.2 Think-Alouds

Inspired by the aforementioned work of Robbins *et. al.*, Naveed Arshad has created a recitation experience based on using Think-Alouds. (20) A Think-Aloud is a protocol that requires a subject to work through a process while verbally explaining all of the thoughts they have and methods they employ while solving the process. (20) Arshad's intent is to use Think-Alouds as a method of conveying computer science related problem solving skills during the recitations of his CS2 course.

High-quality TA's were selected to act as the Think-Aloud subjects. These TA's were exceptional graduate students who have had many years of programming experience and often also had experience in industry. After training the TA's on the Think-Aloud protocol, the TA's would hold Think-Aloud based recitations based on the preceding lecture's material. They were asked to select a significant problem within the domain currently being discussed, and would solve it during recitation using the skills that the students had been taught. This allowed for the students to observe the thought processes of the TA's as they decomposed and solved the problems.

The students reacted very positively to the Think-Aloud-based recitations, not only exhibiting superior problem solving skills, but also learning good code-writing practices based on the styles used by the TA's. At the end of the course, students were surveyed and asked to rank the effectiveness of the various aspects of the course. The Think-Aloud recitations were the most highly-ranked aspect of the course.

2.7 Choice of language

The choice of which programming language to use in a computer science course perhaps gives rise to the biggest accident in computer science education at the CS1 and CS2 level: The chosen language must be taught to students. Recall that "students cannot learn a second language in a primarily independent fashion". (15) While there is some relief in knowing that both the language and the pedagogical methodology chosen for CS1 appears to be independent of student performance in CS2(16)(17), the choice of language is still important: Language choice is known to impact student retention, perception of computer science, and overall performance within a given CS1 or CS2 class. (2) (4) (5) (16)

Assessing the Ripple Effect of CS1 Language Choice by Dingle and Zander provides an excellent overview of the strengths and weaknesses of the commonly employed programming languages for CS1 as of 2001(15). While their insights into C, C++, and Java maintain relevance, many of the languages that they survey are no longer mainstream. Further, since Dingle and Zander's article, many new languages have come into focus as candidate languages for teaching CS1 and CS2.

2.7.1 C

Usage of the C programming language is a topic of much contention. Some argue that C is both inappropriate and harmful to teach in an introductory setting (8), but many agree that having some exposure to C is still a necessity for modern computer scientists. (21) C is a significant introductory language in part because of its small yet powerful grammar (15) and the access that it provides to rudimentary pointer and memory operations. (22)

A major criticism of C is that students spend an inordinate amount of time debugging minor syntactic errors as well as convoluted memory errors, and that this debugging process is both demotivating and largely unproductive. (8) (15) Some argue that these are in fact positive traits of C: successful C programming requires careful coding practices and strong debugging abilities, and therefore teaching with C helps to convey these skills. (22)

C is the only programming language presented in this paper which is not object-oriented.

2.7.2 C++

As it is built upon C, C++ naturally shares most of the strengths and weaknesses of C. As with C, one of the most significant pedagogical reasons for choosing C++ is that it enables meaningful exploration of pointers and memory management. (2) C++ also tends to be more practical than C because in many industries, especially computer game development, C++ is still the primary language employed. (2)

Some meaningful enhancements which C++ provides include enhanced IO, superior access to libraries (via the STL), enumerations, pass-by-reference parameters, and object orientation. Unfortunately, C++ is criticized as way-too-complex, and still provides many of C's confusing components such as type casting, implicit type conversion, lack of error detection for array out-of-bounds errors, etc. (15) (23)

On the bright side, mastering C++ tends to make other languages seem easy by comparison.

2.7.3 Java

Java has been a major player in CS education for many years. This popularity exists partly because of Java being considered cutting-edge and "Cool". (24) Java is no longer a young language, and since its development, several new languages have emerged which have built upon and enhanced Java's ideals.

While Java is primarily considered as an alternative to C++, some CS1 courses have found benefit in using procedural Java in place of C. (5)

Garbage collection, superior String data types, better compiler and memory management errors, and a large body of libraries are among the chief reasons Java is selected. Because pointers are not practically accessible within Java, the ability to teach about pointers and memory management is greatly diminished. Java's differentiation between objects and primitive types also contributes to student confusion. (15)

2.7.4 C#

C# is a relatively young language which is currently in its fourth release iteration. Although built to be syntactically similar to C++, C# is, at the basic level, very similar to Java. In 2002, Reges postulated that C# (then in version 1.0) could be a viable candidate for replacing Java as a language for CS1/CS2. The language features he cited as advantages of C#, all of which still hold in C# 4.0, include simpler IO functionality, a simpler Main(), a consistent object model, iterators and foreach loops, properties, reference parameters, and closures. (24)

Another advantage of C# which may not have been available during the time of Reges' research is that C# supports pointers.(25) By using the *unsafe* keyword, C#'s garbage collector can be instructed to consider part of your program to be unmanaged code. Within this unmanaged region, C++ style pointers can be created and manipulated outside of the constraints of the garbage collector. Although the syntactic overhead for pointers is higher than that of C or C++, C#'s pointers still provide a pedagogical playground for pointer and memory-management based topics.

A point against C# is that it is not a truly cross-platform language. While efforts exist to create C# environments on non-Windows systems, Java is more compatible across platforms.

2.7.5 Python

Python is rapidly emerging as a viable choice for teaching CS1 and CS2, but it is especially receiving attention for its usefulness in CS1. Python's nature as an interpreted scripting language makes it ideal for students who are first learning to write code. There is little to no garbage-code overhead with Python: the language lends itself to very concise statements and syntax. The interpreted nature of Python means that students can run the Python interpreter, type code, and receive line-by-line feedback on the results of their input. Python is also a heavily object-oriented language, however, unlike Java and C#, the additional syntax imposed by the object orientation is basically nil. It can also be argued that Python is substantially motivating and practical for students to learn. (16)

2.7.6 Actionscript (with Flash)

Actionscript can be an immensely fun language to learn and work with, and this is a great reason to choose Actionscript for CS1 or CS2. There is very low overhead for getting a Flash project up and running, interactive, and with visual feedback. In some cases, the project can be composed entirely using the Flash IDE. Flash can also be very attractive to students because of the ease of creating and sharing their resulting Flash files. (18)

The object model within Flash is highly conducive to event-driven programming and, if Actionscript is used, it is necessary to instruct students on event-driven programming. Flash also has the "feature" of being slow- this can be put to good effect by using the speed limitations of Flash as motivators for using superior algorithms. (2)

Unfortunately, Actionscript can be very difficult to debug. The Flash compiler is notorious for generating code which can fail silently. Further, transitioning from Actionscript to a C++ style language has been shown to be unintuitive. (18)

2.8 Retention Efforts

A common variable measured in course development efforts for CS1 and CS2 is the effect of the course on student retention within CS. (4) (18) (26) (27) (28) While designing a single course as a way of solving retention problems might seem like a worthy goal, the potential impact of student outreach programs and cohesive departmental tutoring and outreach efforts is also quite substantial. (19)(29) Jeffrey Popyack argues that an ACM-Women's chapter is a strong benefit to women in computer science because it provides a community of same-sex peer support. (30) Orientation activities, such as the Scavenger Hunt (29), can be highly impactful.

A multi-faceted approach is necessary to improve retention and gender diversity issues within computer science; high-impact retention efforts such as supporting student groups and hosting orientation activities are likely to be much easier to implement than redesigning CS1 and CS2 into the ultimate collaborative, educational, and social experiences.

Although alterations to the teaching methodologies of CS1 and CS2 can improve retention, department-level involvement can be a more substantive channel through which to reach students.

2.8.1 Collaboration and Teamwork

Among the most consistently successful methods of improving student retention, morale, and community via classroom experiences is to have courses involve collaboration and teamwork.

McKinney and Denton experimented with using agile techniques to host project teams in CS1 (26), hypothesizing that the team aspect of the course would be of special benefit to the women and minorities in the class. They allowed students to form their own five to nine person teams, and then lead the teams through many of the practices of agile software development: pair programming, stand-up meetings, test-driven development, etc. After leading the teams through three project iterations, the teams were surveyed. Distinctive problems arose from allowing teams to self-form: specifically, skill levels were not appropriately matched and many teams suffered as a result. There were many situations where team members were rude and unethical; their article hints at several students who they felt were unfit to work with others. In spite of these disturbing behaviors, students in the class exhibited increased senses of comfort and belonging as a result of the team activities.

Agile methods may or may not be ideal for approaching teamwork in CS1, but it seems that even in what appear to be strenuous collaborative efforts, benefit arises.

2.9 Comfort, Intimidation, and Interest

Gail Chmura is a high school teacher in Vienna, Virginia. She teaches introductory computer science to 135 students every year. While it is rare for her students to have programming experience, many of them have experience with computer games. When she starts her course, her students who have had computer experience are, as she puts it, "ready to go". The students without computer experience are timid and anxious- though they are no less prepared for the material than the other students, the burden of intimidation weighs on them. Investigating her students further, she found all of her students could learn the material - they just required different amounts of support, time, and paces of work. She also found that there were no performance differences between males and females. (31)

Intimidation does not fade after high school: Intimidation and attitude are present factors in the performance of college-level students studying computer science. (4) (28) Methods of reducing intimidation include:

2.9.1 Comfortable Questions

Forte and Guzdial found that by teaching a CS1-equivalent course with only non-CS majors enrolled eased tension and made students feel more comfortable about asking questions and participating in class. (4) They also provided students with a web forum which they could use to communicate (anonymously, if desired) with each other about class topics. This forum successfully facilitated communication between students who were otherwise too shy to ask what they perceived as "stupid questions."

2.9.2 Comfortable material

Another group found that comfortable assignment material is a significant factor in student comfort and interest. Faced with a class of mixed-background students, they learned to omit mathematical topics from their computer science assignments and postpone math-dependent topics until later in the curriculum. In place of math, assignments were built around game simulations and simple software applications. As a bonus, students who were bad at math became motivated by their newfound abilities as algorithmists. (32)

2.9.3 Intriguing Content

Thomas Standish and Norman Jacobson of UC Irvine were disappointed with their students' lack of interest in theoretical computer science. Hypothesizing that computer science theory was not being sufficiently motivated by teaching standard algorithms, they decided to incorporate an O(n) sorting algorithm, ProxmapSort, into their CS2 class. They anecdotally exclaim "Cool algorithms really do show that theory is cool!" (33)

2.9.4 Media and Image Processing

An increasing trend in student engagement is linked to media and image processing. Some hypothesize that using media as a conveyance mechanism for computer science allows students to feel more artistic about programming, while others are content to realize that media is a domain within which most people are comfortable interacting. (4) (22) (34)

Forte and Guzdial launched the first significant effort in teaching CS1 through media and image manipulation, and their results from teaching the course to strictly non-CS majors were overwhelming positive. One year later, Wicentowski and Newhall developed and taught a similar course, but this time targeted as a true CS1 course. Their results, too, are overwhelmingly positive. Another year passes and another course succeeds: Matzko and Davis teach an image manipulation CS1 course using C. Their results are less overwhelmingly positive, as a substantial amount of their students seem to have struggled with implementations. Even so, student feedback was positive and student motivation was high.

Media processing is a powerful motivator for teaching CS1, and sufficient supporting materials now exist that new CS1 courses have many examples to draw upon.

2.9.5 Games

Game development has been found to be another highly motivating angle from which to approach computer science education. In addition to the visual feedback and creative outlets provided by media processing, game development allows for opportunities to illustrate, concretize, and motivate computer science topics as aspects of gameplay. (18)

As interest in this field grows, educational infrastructure is also growing for the purpose of assisting game-inexperienced professors with incorporating games into their lectures. Lewis and Massing provide an infrastructure for use in running a semester-long game development project (35), while Sung and Panitz are working to provide sets of modular game-oriented assignments which are designed to be selectively implemented by interested professors as they move toward game-oriented teaching. [1](27)

To dispel any myths on the subject: Games are not male-biased. While it is true that certain genres of games have been anecdotally known to express gender bias, most casual games are profoundly gender-neutral. "Women play games too." (2)

Further research suggests that game development (and multimedia in general) succeeds in setting a context for the course work, and that context is critical to engagement. Allowing students to collaborate, use their own assets (pictures, sound, etc.), and be able to easily share their work leads to massive gains in enthusiasm. (4)

3 Recipe

There are many aspects to using game development as a pedagogical tool. Many distinct aspects have specific known benefits, and therefore it is reasonable to choose only a subset of the proposed steps and still enhance CS2 assignments.

The type of games being discussed are specifically interactive, graphical games. The general goal of this recipe is to combine the known benefits of multimedia-driven assignments with game development's ability to concretize, illustrate, and motivate abstract topics. (4) (18) (22) (34)

3.1 Familiarity

Use familiar games and gameplay. A strength of teaching through a game development medium is that, as all students have been exposed to some type of game, they should be able to relate to the game being developed. (12) There are many game genres and styles, and in creating a game it is important to select a genre or style which students are likely to have been exposed to. It is unreasonable to expect that all students will be familiar with all of the genres presented, therefore a diversity of common genres and themes should be used to provide maximal coverage. An important factor to consider is the gender bias in gameplay styles. Implementing a clone of Valve's CounterStrike might be exhilarating for some, but alienating for others. Where possible, draw inspiration from popular games on the market. While games such as Taxman(36) might fit your learning goals, many students will have more difficulty relating to them than they would relating to games such as Activision's Guitar Hero.

3.2 Embedded Learning

Exhibit the instructional topic as a component of gameplay. Succeeding at a game can require a depth of understanding which can be used to convey an educational topic. With games, it is possible to ensure the message is conveyed by embedding it into gameplay. (37)

[1] These assignments are currently available at: http://depts.washington.edu/cmmr/Research/XNA_Games/index.php

It is unusual for there to be obvious steps toward constructing familiar, fun games in this manner, and so it is common to neglect this step of the recipe. To illustrate this concept, an example of failed embedded learning would be to use a maze navigation simulation as a visual aid for exhibiting the path taken by the A* algorithm. While this illustration might accurately model A*, there is no user interaction which causes the player to apply A* during play. This could be modified to achieve embedded learning by tasking a player to navigate the maze, and at each step compare their movement with the movement that an A* algorithm would have taken, provide visual feedback on the correlations and divergences, and then score them on how well they were able to emulate A* behavior. A critical design element is that the player should have to master the learning goals in order to master the game. This is important because the core gameplay will provide a context for the student to strive to master the material. Returning to our A* example, if the player is able to master the game by using a guess-and-check system and memorizing A*'s pattern, we have failed. If, instead, the player is able to construct a mental model of A*'s behavior and apply that to solving the maze, we have likely succeeded.

3.3 Fun

Use fun games. If the game is not fun, it is likely not to be as engaging or relatable. This is a difficult guideline to design for, as there is no silver bullet for creating a fun game. (38) In designing the exercises, it is recommended to take an iterative approach to the game development, gathering feedback wherever possible. (39) As a guideline, consider modeling your game off of an existing successful game and implementing the most fundamental gameplay possible: If bouncing a ball is fun, then creating juggling game with a compelling story and heavy graphical polish, though it might also (but not necessarily) be fun, is likely to be excessive and not worth the added development time. There's nothing wrong with making yet another Pac-Man clone.

3.4 Assets

Encourage and facilitate custom assets. Once the core gameplay is in place, there arises a question of how much artistic content to add to the game. Assets, such as audio, plot, graphical effects, models, and textures, may be imported into many games and tend to make the game appear more professional. For the scope of these game development exercises it is unreasonable to expect professional artistic content, however it may also be inappropriate to implement a game using only abstract figures, as such a design may break immersion and relatability for many games. The issue of supplying assets presents us with an excellent opportunity to engage students: allowing students to provide their own assets, such as images and audio, provides them with enhanced relatability, motivation, and pride in their work. (4) For game development assignments which are designed to be immediately playable, it will likely be necessary to provide placeholder assets.

3.5 Deployability

Make the game easy to share. When students are able to share and "show off" their work, they tend to exhibit substantially more enthusiasm in the course material, are driven to better performance, and have a more favorable outlook on Computer Science as a whole. (4) Many game development frameworks, such as Microsoft's XNA and Adobe's Flash, provide automated deployment in the form of a "publish" command. Regardless of the programming environment chosen, it would be wise to incorporate deployment steps into the assignment so that students are able to easily share their completed assignments with their friends or on their websites.

3.6 Playable on Startup

Start assignments by providing students with a playable game. This allows for students to be able to observe the game reflect the changes that they make in code. (18) It also lessens the learning curve associated with getting their game development environment up and running.

3.7 Concretize Abstract Concepts

Motivate concepts using games as a context. Researchers have found that games work well as a mechanism for providing a unifying context for abstract concepts and many other complicated computer science skills. (40) (41)

Many game components favor certain data structures and algorithms and can justify their use from an architectural standpoint.

It can also be common in for performance to be a motivator of algorithm and data structure choice, particularly as the game scales. For example, priority queues are commonly used in real-time strategy games in order to facilitate selectively updating a subset of the units on the playfield during each time-step of the game.

3.8 Extensibility

Choose games with obvious design extensions. Extensions should be available for students to tweak or otherwise personalize their game. Though not necessarily related to your learning goals, providing ways for students to personalize their code has been shown to exhibit beneficial ramifications in terms of confidence, enjoyment, and enthusiasm towards computer science. (1)(3)(4)(14)(42)

3.9 Mitigate Extraneous Code

Structure the assignment to avoid extraneous code. The implementation should require some thought, but should avoid requiring extraneous code wherever possible. The implications of this tenet vary depending on the language used in the given CS2 course, as well as the course's focus on teaching the language versus teaching the concepts. It is arguably fair to include extraneous code for the purpose of reinforcing basic concepts, but be aware that an excess of extraneous code may detract from the intended learning goals. (41) While the choice of framework may depend on the language chosen for the class (See section 2.6 for background on language choices), there are many common frameworks which can drastically reduce the "garbage code" overhead for game development assignments.

Some popular frameworks include Microsoft's XNA framework for .Net-driven game development[2], pygame for Python-driven development[3], the Simple Directmedia Layer for C++ and OpenGL development[4], and commercial options such as Flash[5] and Unity[6].

4 Conclusion

There is a strong case for using game development as the basis for assignments in CS2. (4) (12) (13) (14) (37) Incorporating components of this recipe into a CS2 class is

[2] http://creators.xna.com
[3] http://www.pygame.org/
[4] http://www.libsdl.org/
[5] http://www.adobe.com/products/flash/
[6] http://unity3d.com/

expected to provide increased student comprehension, lowered peer intimidation, performance improvements for low-end students, and increased comfort with and enthusiasm for computer science. (1) (2) (4) (14) (22) (34)

Many resources exist to facilitate game development assignments, including plentiful sample games such as those provided by pygame and sample game development assignments such as those presented by Sung and Panitz's XGA project (section 2.9.5).

If game development assignments are infeasible for a given CS2 course, some steps of the recipe can still be applied regardless of the overarching context of the assignment. Deployability and customizable assets, for instance, will likely generalize as beneficial to most types of assignments.

It is the hope of the author that the reader will introduce at least one of the recipe's components into their CS2 coursework.

5 Further Work

Controlled experiments on the results of implementing components of this recipe would be beneficial. While the recipe is built from research indicating that its steps are appropriate, it may be the case that there are additional necessary components for game development exercises which have not yet been identified and are missing from this recipe.

6 Works Cited

1. *Games as a "Flavor" of CS1.* **Bayliss, Jessica D and Strout, Sean.** Houston, Texas, USA : ACM, 2006. Proceedings of the 37th SIGCSE technical symposium on Computer science education. pp. 500-504.

2. *A games first approach to teaching introductory programming.* **Leutenegger, Scott and Edgington, Jeffrey.** 2007, ACM SIGCSE Bulletin Volume 39 , Issue 1 , pp. 115-118.

3. *Intra-Class Competitive Assignments in CS2: A One-Year Study.* **Garlick, Ryan and Akl, Robert.** 2006, 9th International Conference on Engineering Education.

4. *Computers for Communication, Not Calculation: Media as a Motivation and Context for Learning.* **Forte, Andrea and Guzdial, Mark.** Big Island, Hawai`i : IEEE Computer Society, 2004. Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4 - Volume 4. p. 40096.1.

5. *Back to Basics in CS1 and CS2.* **Reges, Stuart.** 2006, ACM SIGCSE Bulletin Volume 38 , Issue 1, pp. 293-297.

6. **Roberts, E., Ed.** *Computing Curricula 2001: Computer Science Final Report.* New York : IEEE Computer Society, 2002.

7. *No Silver Bullet: Essence and Accidents of Software Engineering.* **Brooks, Frederick Jr. P.** s.l. : ACM, 1987, Computer Volume 20 , Issue 4, pp. 10-19.

8. *C in the first course considered harmful.* **Johnson, L F.** 1995, Communications of the ACM Volume 38 , Issue 5, pp. 99-101.

9. *Object Orientation in CS1-CS2 by Design.* **Alponce, Carl and Ventura, Phil.** Aarhus, Denwark : ACM, 2002. Proceedings of the 7th annual conference on Innovation and technology in computer science education. pp. 70-74.

10. *Teaching an object-oriented CS1 - with Python.* **Goldwasser, Michael H and Letscher, David.** Madrid, Spain : ACM, 2008. Proceedings of the 13th annual conference on Innovation and technology in computer science education table of contents. pp. 42-46.

11. *Components-First Approaches to CS1/CS2: Principles and Practice.* **Howe, Emily, Thornton, Matthew and Weide, Bruce W.** Norfolk, Virginia, USA : ACM, 2004. Proceedings of the 35th SIGCSE technical symposium on Computer science education. pp. 291-295.

12. **Nordlinger, John.** *Computer Gaming To Enhance CS Curriculum.* s.l. : Microsoft Research, 2005.

13. **Prensky, Marc.** Digital Game-Based Learning. *ACM Computers in Entertainment.* s.l. : ACM, 2003, Vol. 1.

14. *Evaluating the Effectiveness of a New Instructional Approach.* **Moskal, Barb, Lurie, Deborah and Cooper, Stephen.** Norfolk, Virginia, USA : ACM, 2004. Proceedings of the 35th SIGCE technical symposium on Computer science education. pp. 75-79.

15. *Assessing the ripple effect of CS1 langiage chouce.* **Dingle, Adair and Zander, Carol.** Oregon Graduate Institute, Beaverton, Oregon, United State : Consortium for Computing Sciences in Colleges, 2001. Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference. pp. 85-93.

16. *Python CS1 as preparation for C++ CS2.* **Enbody, Richard J, Punch, William F and McCullen, Mark.** Chattanooga, TN, USA : ACM, 2009. Proceedings of the 40th ACM technical symposium on Computer science education. pp. 116-120.

17. *Has the paradigm shift in CS1 a harmful effect on data structures courses: a case study.* **Gal-Ezer, Judith, Vilner, Tamar and Zur, Ela.** Chattanooga, TN, USA : ACM, 2009. Proceedings of the 40th ACM technical symposium on Computer science education. pp. 126-130.

18. *A CS1 to CS2 bridge class using 2D game programming.* **Leutenegger, Scott T.** 2006, Journal of Computing Sciences in Colleges Volume 21 , Issue 5, pp. 76-83.

19. *Solving the CS1/CS2 lab dilemma: students as presenters in CS1/CS2 laboratories.* **Robbins, Kay A, et al.** 2001, ACM SIGCSE Bulletin Volume 33 , Issue 1, pp. 164-186.

20. *Teaching programming and problem solving to CS2 students using think-alouds.* **Arshad, Naveed.** Chattanooga, TN, USA : ACM, 2009. Proceedings of the 40th ACM technical symposium on Computer science education table of contents. pp. 372-376.

21. **Spolsky, Joel.** Advice for Computer Science College Students. *Joel On Software.* [Online] 1 2, 2005. [Cited: 3 18, 2009.] www.joelonsoftware.com/articles/CollegeAdvice.html.

22. *Teaching CS1 with Graphics and C.* **Matzko, Sarah and Davis, Timothy A.** 2006, ACM SIGCSE Bulletin Volume 38 , Issue 3 , pp. 168-172.

23. *Some Deficiencies of C++ in Teaching CS1 and CS2.* **Agarwal, Achla and Agarwal, Krishna.** 2003, ACM SIGPLAN Notices Volume 38 , Issue 6 , pp. 9-13.

24. *Can C# Replace Java in CS1 and CS2?* **Reges, Stuart.** Aarhus, Denmark : ACM, 2002. Proceedings of the 7th annual conference on Innovation and technology in computer science education. pp. 4-8.

25. **Liberty, Jesse and Xie, Donald.** *Programming C# 3.0.* Sebastopol, CA : O'Reilly, 2008. 0-596-52743-8.

26. *Affective assessment of team skills in agile CS1 labs: the good, the bad, and the ugly.* **McKinney, Dawn and Denton, Leo F.** 2005, ACM SIGCSE Bulletin Volume 37 , Issue 1, pp. 465 - 469.

27. *Assessing game-themed programming assignments for CS1/2 courses.* **Sung, Kelvin, et al.** Miami, Florida : ACM, 2008. Proceedings of the 3rd international conference on Game development in computer science education. pp. 51-55 .

28. *CS Minors in a CS1 Course.* **Kinnunen, Päivi and Malmi, Lauri.** Sydney, Australia : ACM, 2008. Proceeding of the fourth international workshop on Computing education research. pp. 79-90.

29. *Scavenger hunt: computer science retention through orientation.* **Talton, Jerry O, et al.** 2006, ACM SIGCSE Bulletin Volume 38 , Issue 1, pp. 443-447.

30. *Take your daughters (and sons) to work: and leave them there.* **Popyack, Jeffrey.** 2008, ACM SIGCSE Bulletin Volume 40 , Issue 2, pp. 22-23.

31. *What abilities are necessary for success in computer science.* **Chmura, Gail A.** 1998, ACM SIGCSE Bulletin Volume 30 , Issue 4, pp. 55-58.

32. *Computer Science at Staten Island Community College: Teaching Computer Science in an open admissions environment.* **Chi, Emile C.** s.l. : ACM, 1974. Proceedings of the fourth SIGCSE technical symposium on Computer science education. pp. 48-52.

33. *Using O(n) ProxmapSort and O(1) ProxmapSearch to motivate CS2 students (Part I).* **Standish, Thomas A and Norman, Jacobson.** 2005, ACM SIGCSE Bulletin Volume 37 , Issue 4, pp. 41 - 44.

34. *Using image processing projects to teach CS1 topics.* **Wicentowski, Richard and Newhall, Tia.** 2005, ACM SIGCSE Bulletin Volume 37 , Issue 1 , pp. 287-291.

35. *Graphical game development in CS2: a flexible infrastructure for a semester long project.* **Lewis, Mark C and Massingill, Berna.** Houston, Texas, USA : ACM, 2006. Proceedings of the 37th SIGCSE technical symposium on Computer science education. pp. 505-509.

36. *Taxman revisited.* **Trono, John A.** 4, s.l. : ACM, December 1994, ACM SIGCSE Bulletin, Vol. 26, pp. 56-58.

37. *Video games in education.* **Squire, Kurt.** 2003, International Journal of Intelligent Simulations and Gaming.

38. *What makes things fun to learn? heuristics for designing instructional computer games.* **Malone, Thomas W.** Palo Alto, California : ACM, 1980. Symposium on Small Systems. pp. 162-169.

39. **Gibson, Andrea.** *Agile Game Development And Fun.* Computer Science, University of Colorado at Boulder. Undergraduate Thesis.

40. *Design and implementation of computer games: a capstone course for undergraduate computer science education.* **Jones, Randolph M.** 1, s.l. : ACM, March 2000, ACM SIGCSE Bulletin, Vol. 32, pp. 260-264.

41. **Overmars, Mark.** Teaching Computer Science through Game Design. *Entertainment Computing.* April 2004, pp. 81-83.

42. *The BlueJ system and its pedagogy.* **Kölling, Michael, et al.** 4, December 2003, Journal of Computer Science Education, Vol. 13.

43. *A comprehensive project for CS2: combining key data structures and algorithms into an integrated web browser and search engine.* **Newhall, Tia and Meeden, Lisa.** 2002, ACM SIGCSE Bulletin Volume 34 , Issue 1 , pp. 386-390.

44. *Scaffolding for Multiple Assignment Projects in CS 1 & CS 2.* **Kussmaul, Clifton L.** Nashville, TN, USA : ACM, 2008. Conference on Object Oriented Programming Systems Languages and Applications. pp. 873-876.