Functional Video Games in the CS1 Classroom

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu

Abstract. Over the past decade enrollments in Computer Science undergraduate programs have drastically dropped while simultaneously seeing demand for computer scientists in the job market increase. The reason for this disconnect is, in part, due to the perception new potential students have of programming as a dull activity requiring no creativity, very little social interaction, and endless hours of coding in front of a monitor. The question then is how can we capture the imagination of new students and perk their interest in a way that gets them excited while at the same time giving them a solid foundation in computer programming and Computer Science. This article puts forth the thesis that developing video games using functional programming should be a new trend in the CS1 classroom. The article describes the approach implemented at Seton Hall University using video game programming and Felleisen et al.'s textbook *How to Design Programs.* The first-year programming curriculum is briefly described and how to get students interested in programming through the development of a Space-Invaders-like game is illustrated. The presented development gives the reader a clear sense of how to use functional video games in the first semester classroom.

1 Introduction

Over the past decade enrollments in Computer Science programs have drastically dropped up to 70% in some countries [13]. According to CRA's most recent Taulbee Survey in the United States and Canada, the number of Computer Science and Computer Engineering newly declared majors has dropped from a high around 24,000 in the year 2000 to under 14,000 in the year 2008 [14]. In addition, the production of Bachelor's dropped from a high of over 20,000 in 2002 to under 12,000 in 2009. The Taulbee Survey also suggests that retention rates need to be improved. For example, in 2004 there were about 16,000 newly declared majors and, four years later, in 2008 there were under 12,000 Bachelor's produced.

The drop in enrollment is occurring while seeing demand for computer scientists in the job market increase. According to recent occupational employment projections for 2008-2018, computer and mathematical occupations are expected to grow by 22.2% [8]. This rate of growth is over twice as high as the average for all occupations. Among the fastest growing occupations are computer software engineers with demand for application developers expected to increase by 34% and demand for systems software developers to increase by 30.4%. The data clearly suggests that there is and there will continue to be a high demand for Bachelor's in Computer Science. In addition to the expected demand, trends indicate that Computer Science majors are expected to be amongst the best paid professionals (e.g., software architects rank 8^{th} with a median salary of US$117,000), and amongst the professionals with the best quality of life (e.g., software developers rank 4^{th} with 59% stating that there job has low stress) [10].

Being a field projected to remain in high demand and promising the potential for obtaining a high-paying low-stress job is not enough to attract students to and retain students in Computer Science. This seems counter-intuitive at first glance and can not solely be explained by the negative outlook caused by the dot com bust and the recent down turn in the economy. It is necessary to assess the perspective of students that enroll in the beginning courses. To this end, students enrolling in the introductory Computer Science course at Seton Hall University (the home institution of the author) have been interviewed over the past 8 years. From 2002 to 2007 this course was taught using Java as the language of instruction following the typical syntax-based approach of most textbooks with little emphasis on design and problem solving techniques. Uniformly across students, regardless of whether or not they continued as Computer Science majors, the sentiment was that Computer Science and programming were boring and required little or no creativity and social interaction. Programming was characterized as spending endless hours in front of a monitor debugging code. These sentiments were stronger in women which also exhibited lower retention rates. In 2008, the introductory course was taken over by the author and taught based on Felleisen's et. al's textbook *How To Design Programs* (HtDP) [4]. The outlook of students improved as well as retention rates, but students still characterized most of what they did as boring. Despite focusing on design and problem solving (instead of syntax), students felt that there was nothing really interesting or special about, for example, searching a list, computing the value of an integral, or sorting. The bottom line was that students felt it required no creativity and everyone was doing exactly the same thing and producing the same code. This sentiment to some degree is not unlike what students in other disciplines like, for example, Mathematics and Engineering face: the solution to a problem is the same for all students. There is, however, a difference with Computer Science that may signal why retention is harder. The typical assignment in Computer Science has a component that assignments seen by students in other disciplines do not have. Computer Science students must design, write, debug, and produce a working piece of software. That is, they must build an artifact of their discipline. It is a time-consuming process that beginning students in other disciplines do not have to face. This is not to say that other disciplines do not offer challenging and enlightening exercises to their students, but rarely, if ever, are beginning students in other disciplines asked to build an artifact of their discipline like beginning Computer Science students are asked to do on a regular basis. Given that beginning students can easily shop around and switch majors (at least in the USA), this represents a challenge that must be faced creatively by Computer Science departments to attract students to the major and to increase their retention.

The interviews with students at Seton Hall University identified one element that can help attract and retain students. Students across the board, regardless of whether or not they continued as Computer Science majors, qualified the design and implementation of video games (using the DrScheme's[1] universe teachpack [3]) as very interesting, as requiring creativity, and as fun to work in groups. In addition, students felt that requiring the design and implementation of a large video game by the end of the semester truly brought everything that they had to learn into focus which provided a sense of accomplishment and a sense of satisfaction with majoring in Computer Science. In 2009, the delivery of the introductory course was redesigned to incorporate more development of video games as motivation.

This article advocates that the design of functional video games should be a new trend in introductory Computer Science courses. Having beginning students develop *functional* video games means that they are liberated from reasoning about state and the sequencing of statements, because the code that they develop is assignment-free. Thus, students focus on how to design and implement a solution without having to focus on the overhead and dangers of using assignment. Our experience suggests that this approach facilitates the introduction and the understanding of recursion which usually is a fundamental topic in Computer Science that students struggle with in introductory courses. The use of video games has the added benefit that it has a built-in creative outlet. Students are able to customize their solutions to their personal preferences. The choice of graphics used, the level of difficulty preferred for the game, and the speed at which the game advances, for example, can vary from student to student. This provides students with the sense that not all solutions to a problem are the same and that they can creatively inject their own personality in the development and implementation of a solution. The reader can contrast this with the typical word problem found in a Mathematics or Engineering textbook. This ability to offer students problems with a creative outlet ought to be leveraged to engage, attract, and retain beginning students in Computer Science. The built-in creative outlet that video game development and implementation offers, for example, has proven an especially effective tool to make Computer Science and programming interesting to female students. Among young female students, the opportunity to be creative was the highest ranked characteristic. In contrast, male students ranked the ability to create competitive games the highest with creativity closely ranked behind it. Finally, the development of functional video games provides the opportunity to make core lessons in Computer Science and programming (e.g., design, recursion, sorting, and searching) relevant to the pop culture students are an integral part of. Much of what they learn ceases to be purely theoretical and can directly be applied to create something that not only are they interested in, but are also excited about.

The article first outlines the topics taught in the introductory courses at Seton Hall University and why the use of a functional language is ideal. The article then demonstrates how the design and implementation of a functional

[1] DrScheme has recently been renamed DrRacket.

video game, specifically a Space-Invaders-like game, can be used to motivate and teach students in CS1. The presentation aims to illustrate how functional video games can be used in the first-semester classroom and to serve as a road map that others can follow and adapt to their particular environment and students. The presentation also aims to demonstrate how relatively easy it is to develop a functional video game and to integrate functional video game development into the CS1 classroom. Finally, the article concludes with a discussion of related approaches and some conclusions.

2 Introduction to Computer Science and Programming

Introduction to Computer Science courses tend to focus on providing students with a solid foundation in programming [13]. This characteristic is justified, because teaching students about programming prepares them for the job market, programming tends to attract more students (both those majoring and those not majoring in Computer Science), and programming is a prerequisite for many upper-level Computer Science courses [12]. The debate of what should and what should not be included in an introduction to Computer Science and programming rages on. Instead of engaging in the futile exercise of systematically analyzing the list of potential topics to gain converts, the solution adopted at Seton Hall University is outlined below. The reader can decide decide if the choices made make sense for her institution and her environment.

It is noteworthy that this article is not advocating the presented methodology as absolute or rigid. As Computer Science evolves, so will the technologies, like video games, used to motivate students in introductory courses. The topics (e.g., structures, lists, and sorting) covered in such courses are also subject to change as Computer Science evolves, but at a much slower pace than vogue technologies. The primary lesson that should be drawn is that an interesting domain can be used to make the delivery of a solid foundation in programming fun and interesting for beginning students. Video game programming is such a domain for the foreseeable future.

2.1 Topics Covered in CS1 and CS2 at Seton Hall University

At Seton Hall University, all students must complete four years of study to earn a Bachelor's degree. During this time, students must fulfill general requirements as well as the requirements for their major. The Computer Science major requires 53-54 credits with the typical course being worth 3 credits and some courses being worth 4 credits. During their freshman year (i.e., the first year), students are expected to pass CS1 and CS2 which allows them to move on in their sophomore year (i.e., their second year) to courses focusing on designing classes. During their junior and senior year (i.e., their third and fourth years), students take upper-level Computer Science requirements as well as Computer Science electives most of which require programming.

It is our perspective that introductory Computer Science courses ought to focus on problem solving. Students should be empowered by helping them develop skills that take them from a problem statement to a *well-designed* solution. The emphasis is much more on designing the solution to a problem than the actual implementation of the solution. Although being able to follow through with the implementation of a solution is an important skill, it is the design of the solution that makes the implementation possible. Furthermore, it is the ability to design a solution to a problem that makes a Computer Science education relevant to other aspects of a student's life. Stated simply, solution design skills can be applied to problems beyond those solved using a computer and a programming language, because they make the thinking process explicit.

In addition to developing problem solving skills, students must also learn the rudimentary nomenclature of programming. At Seton Hall, there are two courses, CS1 and CS2[2], that serve as the introduction to Computer Science and Programming. Broadly speaking, CS1 covers the following topics (listed to make the connection with HtDP's Parts I-IV easy):

- Programming with primitive data (e.g., symbols, numbers, and pictures) and primitive functions (e.g., symbol equality, addition, and geometric drawing functions).
- Programmer defined functions and variables.
- Processing finite compound data (e.g., structures).
- Processing arbitrarily large compound data (e.g., structural recursion on lists, trees, and natural numbers).
- Abstraction (e.g., elimination of code repetition and functions as values)

Broadly speaking, CS2 covers the following topics (listed to make the connection with HtDP's Parts V-VIII easy):

- Generative recursion (e.g., quicksort).
- Iteration (i.e., accumulative recursion and loops).
- State-based computations (i.e., design using assignment).
- Distributed Computing (not a topic in HtDP).

Readers interested in a rationale for including the above topics in the curriculum for CS1 and CS2 are referred to the appropriate sections in HtDP. The abstraction techniques studied are specific to functional languages, but the focus is the reduction of errors by reducing code duplication. Students learn that common programming patterns can be captured as functions to make code more readable and less bug-prone. The distributed computing component introduces students to networks, a pervasive technology today, and provides the opportunity to design and implement a distributed application using the same language and software students have used throughout their first year.

These introductory courses aim to provide the foundation needed for students to go on and learn how to design solutions and write programs using any programming language. In fact, the skills acquired are directly transferable to designing programs using object-oriented languages such as Java. Although teaching languages with Scheme-like syntax are used in these courses, the goal is

[2] These courses are actually called Design of Programs I and Design of Programs II.

not teach students Scheme nor is the goal to make them functional programmers. In the interest of absolute clarity, we are not teaching our students Scheme nor do we advocate teaching beginning students Scheme. Scheme is a mature and powerful programming language with native support for many advanced features (e.g., continuations and hygienic macros) that are not addressed nor used in CS1 and CS2. Equally noteworthy is the fact that the emphasis is not on the syntax of any particular programming language although, of course, students must learn some Scheme-like syntax in order to implement solutions. Scheme-like syntax may not seem natural to students on the first day of class (e.g., prefix instead of infix notation), but it is useful in distinguishing Scheme from mathematics. Students may analyze a problem off-screen using mathematics written using infix notation, but must translate it into a programming language's syntax to implement a program. This is a process that is common to program development in general. One of the advantages of using Scheme-like syntax is that this translation is simple enough that it quickly becomes natural to beginning students using HtDP. Other reasons for using Scheme as the core behind the employed teaching languages are given in the preface of HtDP [4]. It is our estimation that the foundation we provide enables students to go on to learn about powerful abstractions provided by other languages (regardless of the syntax used) such as, for example, monads in Haskell, objects and inheritance in Java, and continuations and hygienic macros in Scheme.

In addition to the topics above, emphasis is placed on *iterative refinement*. It is important for students to understand that designs, solutions, and implementations evolve through a continuous cycle of enhancements. This lesson is a difficult one to convey especially when the programs students are asked to develop are small. Large projects, like the design and implementation of a video game, provide an excellent vehicle with which to emphasize iterative refinement.

3 The Functional and HtDP Advantages

The choice of a functional language for introductory courses can be controversial for some faculty members and for some students. This article will not digress too much into the objections raised by faculty members. These objections mostly boil down to not teaching a language used in industry and not focusing on teaching state-based problem solving. Teaching a particular language, even one used in industry, should not be the goal of an introductory course. Mostly focusing on teaching state-based problem solving fails to expose students enough to easy-to-use skills in the design of solutions and programs. In fact, assignment is harmful at the beginning. In our experience, students that start with state-based problem solving find it very hard to design solutions or to understand solutions that fail to mutate variables at every step. The sharp reader will have detected the concept of step (and sequencing) introduced into this text all of the sudden. This is precisely how students think of computation if they start with state-based problem solving: programs are a collection of sequenced assignments.

As any functional programmer knows, nothing can be farther from the truth and statements to this effect by students should not go unchallenged[3].

3.1 The Functional Advantage

Liberating students from reasoning about state and the machine, as mentioned before, is a formidable advantage offered by functional languages. Students are allowed to think about how to solve problems and do not have to reason about how to sequence mutations to solve a problem. They can build on their knowledge of high school algebra to design functions which brings problem solving into a domain that seems familiar to them. This approach has the added benefit that it makes mathematics relevant for students [6], improves the grades of students in mathematics courses [5], and builds on a natural synergy that more and more looks like an endangered species in the CS curriculum.

Functional languages can also–but not always–present students with a minimal amount of syntax that needs to be learned in order to solve interesting problems. Dynamically typed functional languages, for example, remove all syntax requirements associated with types which are required by statically typed languages. The observation is simple: the less time we spend discussing syntax the more problem solving and design principles we can actually teach.

Finally, as pointed out by Felleisen et. al [4], if an interpreted functional language is used, then Byzantine discussions about input and output are not necessary. Students do not have to be bogged down with how to input and output data–which has little or nothing to do with the solution to the problem they are implementing. Once again, students are liberated from side issues and allowed to focus on problem solving. Learning how to do I/O should not be a prerequisite to learn the basics of programming nor to take your first steps into the world of Computer Science.

3.2 The HtDP Advantage

An HtDP-based curriculum presents two major advantages for teaching introductory Computer Science and programming courses. The first is that it gives students a road map to follow from a blank screen to a working solution. This road map is based on what Felleisen et al. have coined the *design recipe*. A design recipe is a series of steps a student can follow in the design of a solution. In fact, there are several different design recipes all of which are variations on a theme depending on the type of problem being solved or the type of data being processed. The basic skeleton for developing a function for all the variations of design recipes is:

1. Problem analysis and data definitions.
2. Stating the contract, the purpose, and writing the function header.
3. Defining tests showing how a function should work.

[3] The author does not recommend challenging or trying to convince faculty members that express such a view in open debate. Let your results speak for themselves.

4. Development of a function template (derived from the data being processed) and an inventory of expressions that can be used to implement the function.
5. Defining the function.
6. Running the tests and making corrections if necessary.

At the beginning, students find the use of the design recipe cumbersome especially when the programs/functions being designed are small. In fact, many students feel it is overkill. It is important, however, to encourage them to develop good habits by following the steps in the design recipe even if they can see the solution before going through all the steps. The assignment of a non-trivial problem as homework and grading how well students follow the design recipe go a long way to bringing the point home.

The second major advantage an HtDP-based curriculum presents is that it is tightly-coupled with the DrScheme programming environment. This environment comes with a series a successively richer subsets of Scheme-like languages called the teaching languages. Each part of HtDP is associated with a teaching language. The teaching languages make available just enough syntax for students to learn to design solutions to the types of problems that they are being asked to solve. This hierarchy of teaching languages allows for meaningful error messages to be generated for mistakes that would otherwise be hard to decipher by a beginning student [4]. Our experience is that students suffer through much less frustration when compared to Seton Hall's old Java-based approach. In addition, DrScheme also comes with a rich set of libraries/teachpacks that simplify the implementation of solutions for different kinds of problems. One such teachpack is *universe* which defines an interface for writing animations (both interactive and non-interactive). Universe envisions an animation as a series of snapshots of an evolving world. There is a clock that at every tick displays the next snapshot of the world. Students must define the elements of the world and define functions for computing the next snapshot of the world when the clock ticks or when an external event, such as a keystroke or a mouse movement, occurs. Students must also define functions for drawing the world and for detecting the end of the animation. The code students develop can be functional (i.e., assignment-free) and free of any concerns about coordinating the display of snapshots. Readers interested in more details about the universe teachpack are referred to *How to Design Worlds* [3].

4 Video Games in CS1

Armed with the design recipe and with DrScheme's universe teachpack, instructors and (first-year) students can be ambitious and start developing a video game starting on the first day of class. At the beginning, of course, the video game is, shall we say, less than interesting. It lacks any real features video games have, because students still do not know how to do very much. The promise of developing a video game, however, is used to keep students motivated and students are encouraged as the process of iterative refinement adds dimensions to the game.

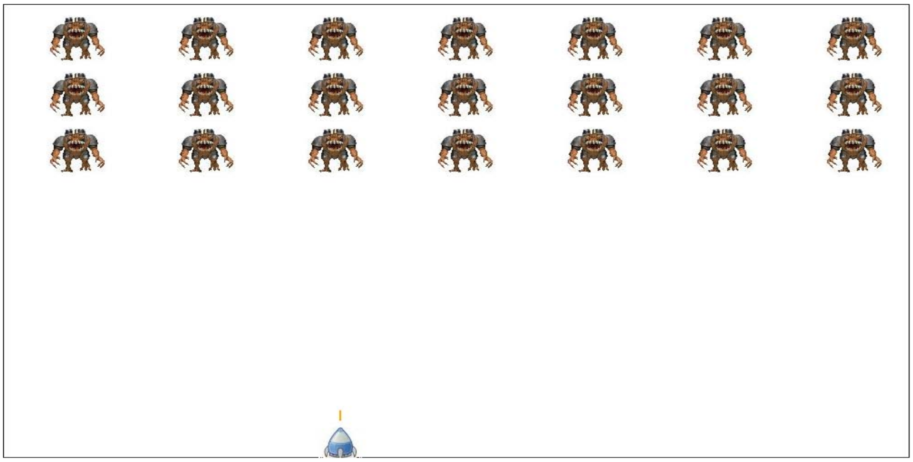

Fig. 1. A snapshot illustrating an implementation of Aliens Attack

Our attention will now focus on illustrating how to motivate topics in the CS1 curriculum by tying them in with the development of a Space-Invaders-like video game that we shall refer to as *Aliens Attack*. The presentation will display a series of different incarnations implemented during the iterative refinement process. In the game there is a defender at the bottom of the screen that the player may move left and right to shoot aliens. There are also one or more aliens in a grid-like formation that are trying to reach the bottom of the screen–presumably to conquer earth. All aliens move in the same direction–either left or right–and when an alien reaches the edge of the screen all aliens move down and start moving in the opposite direction. The game ends when either all aliens have been destroyed by the defender or an alien reaches the bottom of the screen. Figure 1 displays a visual representation of Aliens Attack.

4.1 Aliens Attack v0.0

On the first day of class, the assumption is made that students have no background in programming. Therefore, they are stumped by the task of creating a video game despite their enthusiasm to do so. They are told that the game will be developed using iterative refinement–not all at once, but little by little as they learn how to design programs. Nonetheless, the first version of aliens attack is developed. It is simply an empty scene of `HEIGHTxWIDTH` computer graphics coordinates where the game is to be drawn and played. A sample is generated by the code in Figure 2.

Students may be a little disappointed with this first version, but they are motivated to learn about defining constants, about primitive data, and about how to place images in a scene. In addition, students are encouraged to read the documentation to learn more about how place-image works. By the second class, most students are proud to show how they have modified the color

```
(define HEIGHT 650)
(define WIDTH 900)
(define E-SCENE
  (place-image (rectangle (* 2 WIDTH) (* 2 HEIGHT) 'solid 'yellow)
               0
               0
               (empty-scene WIDTH HEIGHT)))
```

Fig. 2. The code for Aliens Attack v0.0

and the size of the canvas to their liking. The stage is now set to learn about primitive data and primitive functions.

4.2 Aliens Attack v0.1

After gaining some programming experience with primitive data, students are brought back to the video game. The first enhancements tackled are drawing the defender in a scene and creating a defender in a new position. During problem analysis, students quickly realize that the defender can be represented by a natural number, n, such that $0 \leq n \leq \texttt{WIDTH}-1$. This natural number represents the x coordinate of the defender. There is no need to represent the y coordinate nor the image of the defender as variable, because they can be defined as constants.

To draw a defender, students realize they need as input a defender and a scene and they need to return a scene in which the defender has been drawn in the given scene. This analysis leads to their contract and drawing function which may look as follows:

```
;;;   DATA DEFINITION: A defender is a natural number, n,
;;;                    such that 0 <= n <= WIDTH - 1
; EXAMPLE
(define OUR-HERO (/ WIDTH 2))

;;; draw-defender: defender scene --> scene
(define (draw-defender a-defender scn)
  (place-image DEF-IMG a-defender DEF-HEIGHT scn))
```

`DEF-IMG` and `DEF-HEIGHT` are the constants for the image and the y coordinate of the defender.

Naturally, the next task that students wish to tackle is getting the defender to move using keystrokes. This provides an opportunity to introduce students to conditional statements and booleans as the direction the defender moves in, if at all, depends on keystrokes. Data analysis reveals that computing a moved defender requires a defender and a direction[4] leading to a function to move the defender:

[4] Represented as a string corresponding to a keystroke in DrScheme.

```
;;; move-defender: defender string --> defender
(define (move-defender a-defender direction)
  (cond [(string=? direction "right") (+ a-defender DEF-DELTA-X)]
        [(string=? direction "left")  (- a-defender DEF-DELTA-X)]
        [else a-defender]))
```

DEF-DELTA-X is a constant representing by how much to move the defender with each keystroke.

Testing the above function, however, reveals a bug. The defender can move off the scene driving home early the importance of testing in software development. Iterative refinement yields an improved function to move the defender:

```
;;; move-defender: defender string --> defender
(define (move-defender a-defender direction)
  (cond [(and (symbol=? direction 'right)
              (<= (+ a-defender DEF-DELTA-X) (sub1 WIDTH)))
         (+ a-defender DEF-DELTA-X)]
        [(and (symbol=? direction 'left)
              (>= (- a-defender DEF-DELTA-X) 0))
         (- a-defender DEF-DELTA-X)]
        [else a-defender])).
```

4.3 Aliens Attack v0.2

The next task is to introduce aliens each requiring an x and a y coordinate to represent their position which motivates the need to represent finite compound data. Such data is represented using structures. In this part of the course, some students may stumble given that it is unlikely that they have studied functions on compound data in any other course. Students start by studying a built-in structure in DrScheme called a posn to represent a position in a scene. After posn, students study how to define their own structures and how to design functions for such structures.

A student's first attempt to represent an alien will typically define a structure that only contains posn. The experienced programmer will notice that a structure definition is unnecessary, but its elimination is an optimization that can pursued as a future refinement. There is nothing inherently wrong with defining an alien as a structure that contains a posn.

Using compound data requires the development of a function template and an inventory of expressions that can be used to access and manipulate the components of the compound data. For an alien in our video game, for example, the results may be:

```
(define-struct alien (position)) ; where position is a posn
; EXAMPLE
(define ALIEN1 (make-alien (make-posn (/ WIDTH 2) (/ HEIGHT 2))))
; f-on-alien: alien --> ???
```

```
(define (f-on-alien an-alien)
  ; inventory
  ; (alien-position an-alien) = the posn stored in an-alien
  ; (posn-x (alien-position an-alien)) = the x-coordinate of
  ;                                       an-alien
  ; (posn-y (alien-position an-alien)) = the y-coordinate of
  ;                                       an-alien
  <the body of f-on-alien> )
```

This template can then be specialized by students to write functions to manipulate aliens akin to moving and drawing the defender. It is noteworthy to point out that students are not hacking code nor are they developing code using a blind trial and error strategy. Instead, they must think explicitly about the structures they are manipulating and understand that their structures influence the shape of the code they must develop.

Once students have some experience with structures and have written functions to draw and move the defender as well as the alien, they are ready to define a structure for the world and write the handlers for the first animation. The world is a structure that captures the elements that can change. In our video game there are three changing elements (so far): the defender, the alien, and the direction (left, right, or down) the alien is traveling. This leads to the following definition for `world` and its function template:

```
(define-struct world (def al dir))
; where def is a defender, al is an alien, and dir is a string
; EXAMPLE
(define INIT-WORLD (make-world OUR-HERO ALIEN1 "right"))

;f-on-world: world --> ???
(define (f-on-world w)
  ; inventory
  ; (world-def w) = the defender in w
  ; (world-al w) = the alien in w
  ; (world-dir w) = the string for the direction in w
  ; (alien-position (world-al w)) = the posn of the alien in w
  ; (posn-x (alien-position (world-al w)))
  ;    = the x coordinate of the alien in w
  ; (posn-y (alien-position (world-al w)))
  ;    = the y coordinate of the alien in w
  <BODY OF f-on-world>)
```

The game requires four event handlers for the animation: one to draw the world, one to process key strokes, one to compute the next world every time the clock ticks, and one to detect the end of the game. The above template is specialized by students to create the functions displayed in Figure 3 that serve as the event handlers. The code displayed is fairly easy to understand and uses auxiliary functions `at-edge?` to detect if the alien is at either the left or the right edge of

```
; draw-world: world --> scene
; Purpose: To draw the world
(define (draw-world w)
  (draw-alien (world-al w)
              (draw-defender (world-def w) E-SCENE)))

; process-key: world string --> world
; Purpose: To create a new world based on a keystroke
(define (process-key w k)
  (make-world (move-defender (world-def w) k)
              (world-al w)
              (world-dir w)))

; next-world: world --> world
; Purpose: To compute the next world (after a clock tick)
(define (next-world w)
  (make-world (world-def w)
              (move-alien (world-al w) (world-dir w))
              (cond [(and (at-edge? (world-al w))
                          (not (string=? (world-dir w) "down"))) "down"]
                    [(and (over-r-edge? (world-al w))
                          (string=? (world-dir w) "down")) "left"]
                    [(and (over-l-edge? (world-al w))
                          (string=? (world-dir w) "down")) "right"]
                    [else (world-dir w)])))

; game-over?: world --> boolean
; Purpose: To determine if the game is over (i.e., the alien has landed)
(define (game-over? w)
  (> (+ (posn-y (alien-position (world-al w))) ALIEN-DELTA-Y)
     HEIGHT))
```

Fig. 3. Functions to manipulate the world in Aliens Attack v0.2

the scene, over-r-edge? to detect if the alien is at the right edge of the scene, and over-l-edge? to detect if the alien is at the left edge of the scene.

Finally, students must provide the handlers to the universe interface to run the game. The syntax to do so is not cumbersome and easy to follow for students:

```
(big-bang INIT-WORLD
          (on-draw draw-world)
          (on-key process-key)
          (on-tick next-world)
          (stop-when game-over?))
```

4.4 Aliens Attack v0.3

Once students have a running video game with a moving alien and a defender that responds to keystrokes, the desire to add multiple aliens and shooting

```
; DATA DEFINITION
; A list of aliens, loa, is either
;  1. empty
;  2. (cons a l), where a is an alien and l is a loa.

; f-on-loa: (listof alien) --> ???
(define (f-on-loa a-loa)
  ; inventory
  ; (first a-loa) = the first alien in a-loa
  ; (rest a-loa) = a-loa minus its first alien
  ; (f-on-alien (first a-loa)) = the ??? from applying f-on-alien
  ;                              to the first alien in a-loa
  ; (f-on-loa (rest a-loa)) = the ??? from applying f-on-loa to
  ;                           (rest a-loa)
  (cond [(empty? a-loa) ...]
        [else (...(f-on-alien (first a-loa))
               ...(f-on-loa (rest a-loa)))]))

; move-loa: (listof alien) string --> (listof alien)
(define (move-loa a-loa direction)
  ; inventory
  ; (first a-loa) = the first alien in a-loa
  ; (rest a-loa) = a-loa minus its first alien
  ; (move-alien (first a-loa)) = the first alien moved
  ; (move-loa (rest a-loa)) = the moved (rest loa)
  (cond [(empty? a-loa) empty]
        [else (cons (move-alien (first a-loa) direction)
                    (move-loa (rest a-loa)))]))
```

Fig. 4. Recursive data definition for a list of aliens, recursive template for a list of aliens, and a specialization of the template to move a list of aliens in Aliens Attack v0.3

capabilities quickly arises. Student analysis reveals that there can be zero aliens, if all have been destroyed by the defender, or there can be one or more aliens that still need to be destroyed. Thus, the introduction of multiple aliens motivates the need for data of arbitrary size and leads to the study of lists (and other recursively defined data definitions like trees). During this study, students design and implement, for example, searching, sorting, accumulating (e.g., summing the elements of a list), and filtering algorithms. Throughout, it is emphasized that students exploit the structure of their data to determine the structure of their code. For example, a self-reference in the data definition translates to a recursive call in their function. In this manner, students learn quite naturally how to exploit structural recursion.

Armed with some experience processing data of arbitrary size, students return to the design of the video game and create the recursive data definition and function template for a list of aliens displayed in Figure 4. It is noteworthy that students realize that a function that consumes an alien must be applied to the first alien in the list and that a function that consumes a list of aliens must be applied to the rest of the list. Furthermore, the reason for a recursive call is not a mystery–the self-reference in the data definition for `loa` translates to a recursive call–and students know in advance of writing any code that such will be the case. Figure 4 also displays a specialization of the function template to move a list of aliens. Students can now be charged with changing their definition of the `world` structure to incorporate the list of aliens and to incorporate shots. Such an exercise reinforces the lessons on designing structures as well as filtering and list processing in general given that shots and aliens that are hit and shots that go off the screen must be eliminated from the game.

4.5 Aliens Attack v0.4

The final component of CS1 introduces students to abstraction. At this point in the course, students have added shots to their video games and will have functions to move a list of aliens and to move a list of shots. Typically, a function to move a list of shots will look as follows:

```
; move-los: (listof shot) --> (listof shot)
(define (move-los a-los)
   (cond [(empty? a-los) empty]
         [else (cons (move-shot (first a-loa))
                     (move-los (rest a-los)))]))
```

Structurally, this function is similar to `move-loa` in Figure 4 and most students grow tired of having to write similar code as this over and over. This presents the opportunity to introduce students to abstraction using elimination of code duplication and code reuse as motivation to create shorter programs. After an introduction to abstraction, students return to the design of the video game and re-implement as follows:

```
; move-loa: (listof alien) string --> (listof alien)
(define (move-loa a-loa direction)
   (map (lambda (a) (move-alien a direction)) a-loa))

; move-los: (listof shot) --> (listof shot)
(define (move-los a-los) (map move-shot a-los)).
```

When seen side-by-side, students realize that these new functions are structurally similar and apply the design recipe for abstraction to them. This process yields the code in Figure 5. Students realize that the first function in Figure 5 is an abstract function to move a list of anything which can be used in the development of other video games and appreciate that it is short (i.e., one line of code), that it is not recursive, and that it is easy to use. In fact, most students can not believe how easy moving a list of anything is made through abstraction.

```
; move-list: (X --> X) (listof X) --> (listof X)
(define (move-list f a-list) (map f a-list))

; move-loa: (listof alien) string --> (listof alien)
(define (move-loa a-loa direction)
   (move-list (lambda (a) (move-alien a direction)) a-loa))

; move-los: (listof shot) --> (listof shot)
(define (move-los a-los) (move-list move-shot a-los))
```

Fig. 5. Abstract function to move a list of `X` and concrete functions to move a list of aliens and a list of shots

5 Related Approaches

There have been a several approaches to the use of video game programming in conjunction with functional languages to motivate beginning students. The developers of DrScheme and HtDP have described the technical implementation of I/O in the universe teachpack and have outlined how to implement, both non-distributed and distributed, small simulations based on that description [5]. Naturally, the work described in this article builds on the work done by the developers of HtDP and the universe teachpack. In contrast, the work presented in this article sets aside the technical discussion of I/O and presents a more detailed road map for the actual use of video games in the CS1 classroom. In essence, the work described in this article is for educators "in the trenches" focusing on the actual deployment of a functional video game strategy in the classroom. In addition to describing a larger more realistic application in a CS1 setting, the work described here closely knits together the use of video games in conjunction with CS1 topics.

Soccer-Fun, developed using Clean, aims to motivate students by having them write programs to play soccer games [1]. It has successfully been used in a sophomore-level course aimed to teach functional programming to students with imperative and object-oriented programming experience and in a high school setting to attract students to Computer Science. The developers of Soccer-Fun report no experience with it in CS1. Although soccer is the most popular sport on the planet, it is unclear if such a platform is effective with students that are not fans of the sport.

Yampa is a language embedded in Haskell used to program reactive systems such as video games [2]. Yampa, in fact, has been used to implement a Space-Invaders-like game. As Soccer-Fun, Yampa is mostly intended to help those already familiar with imperative/OO programming to learn functional programming techniques. Both Soccer-Fun and Yampa, nonetheless, have been effectively used to motivate students.

The use of Haskell itself to program a video game, inspired in the classical game *Asteroids*, has also been reported successful at motivating students [9]. The authors report that the popularity of their approach was due, in part, to the use of animated graphics. Furthermore, the authors report that students made great efforts to embellish their solutions with fancy graphics. This may be the earliest indicator that providing students a creative outlet to personalize solutions to problems is an important pedagogic technique in Computer Science education. As with Soccer-Fun and Yampa, the scope of the efforts was to teach functional programming.

Outside the realm of functional programming, Python is poised amongst the most popular languages used to motivate students using games. Python presents students with an interpreter for easy interaction, but is an object-oriented language that naturally carries all the difficulties of designing and implementing programs using assignment. Furthermore, textbooks using Python require almost immediately the use of assignment and looping constructs (e.g., see [7,11]). Thus, programming quickly moves away from the familiar domain of high school algebra.

6 Concluding Remarks

This article puts forth the thesis that programming functional video games should become a trend in the CS1 classroom. The strongest proof that can be presented for why this should be a new trend is two-fold. On one side, the reader hopefully agrees that the development of functional video games is an imaginative approach that is not beyond the scope of beginning students as evidenced by the development presented in this article. On the other side, although not quantified, we have the enthusiasm and interest in programming that developing video games sparks in students. It is the belief of the author that functional video games can be an effective tool to once again make Computer Science an attractive and popular major for beginning college students.

Unlike previous efforts in the classroom to use functional languages to program video games, the goal is not restricted to teaching functional programming to students with programming experience. Instead, the goals of using functional video games are to motivate a student's interest in programming and to provide a sound vehicle for the dissemination of a solid foundation in programming. Essential to such an effort in the CS1 classroom is providing an interface with minimal syntax and an easy to understand semantics. It is the expectation of the author that the described development of a functional video game, using HtDP and DrScheme's universe teachpack, has demonstrated how easily a solid programming foundation can be imparted to students using a domain they consider fun and interesting.

Future work includes demonstrating how functional video games can be an effective pedagogical tool for motivating and teaching generative recursion, accumulative recursion (i.e., iteration), state-based computations, and distributed programming. The approach will assume that students have a foundation using structural recursion as well as abstraction as outlined in this article.

Acknowledgements

The author thanks the plt-scheme and the plt-edu mailing list community for the many frank and eye-opening discussions about teaching programming, about HtDP, and about interesting programming projects for students. Special thanks are extended to Matthias Felleisen and Shriram Krishnamurthi for frequently and kindly engaging me in frank discussions about teaching programming to beginning students. I trust that our public discussions on the mentioned mailing lists have been mutually beneficial.

References

1. Achten, P.: Teaching Functional Programming with Soccer-Fun. In: FDPE 2008: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education, pp. 61–72. ACM, New York (2008)
2. Courtney, A., Nilsson, H., Peterson, J.: The Yampa Arcade. In: Haskell 2003: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell, pp. 7–18. ACM, New York (2003)
3. Felleisen, M., Findler, R.B., Fisler, K., Flatt, M., Krishnamurthi, S.: How to Design Worlds (2008), http://world.cs.brown.edu/1/
4. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge (2001)
5. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: A functional i/o system or, fun for freshman kids. In: Hutton, G., Tolmach, A.P. (eds.) ICFP, pp. 47–58. ACM, New York (2009)
6. Felleisen, M., Krishnamurthi, S.: Viewpoint: Why Computer Science Doesn't Matter. Communications of the ACM 52(7), 37–40 (2009)
7. Harris, A.: The L Line, The Express Line to Learning. In: Game Programming, Wiley Publishing, Inc., Hoboken (2007)
8. Lacey, T.A., Wright, B.: Occupational Employment Projections to 2018. Monthly Labor Review, 82–123 (November 2009)
9. Lüth, C.: Haskell in Space: An Interactive Game as a Functional Programming Exercise. J. Funct. Program 13(6), 1077–1085 (2003)
10. Money Magazine and Salary.com. Best Jobs in America. Money Magazine (2009)
11. McGugan, W.: Beginning Game Development with Python and Pygame: From Novice to Professional. Apress, Berkeley (2007)
12. The Joint Task Force on Computing Curricula. Computing Curricula 2001 Computer Science (December 2001), http://www.acm.org/education/education/education/curric_vols/cc2001.pdf
13. CS2008 Review Taskforce. Computer Science Curriculum 2008: An Interim Revision of CS 2001 (December 2008), http://www.acm.org//education/curricula/ComputerScience2008.pdf
14. Zweben, S.: 2007-2008 Taulbee Survey. Computing Research News (May 2009)