Abstract. The use of video games to teach introduction courses to programming and Computer Science is a trend that is currently flourishing. One of the most successful and promising approaches uses functional video games to get students interested and engaged in programming. This approach is successful, in part, because functional video games provide a domain of interest to most Computer Science undergraduates and remove the need to reason about designing state-based programs. A plethora of examples exist that have students develop games exploiting structural recursion which resemble such classics as Space Invaders and Snake. Once students master the basics of structural recursion the time comes to move beyond structural recursion to generative and accumulative recursion. It is up to the instructor to harness the enthusiasm and appetite that students have to develop more video games. This requires finding games that require the generation of subproblems in the same class as the input problem or that require accumulators to be successfully played or solved. This article presents a road map to make the transition from structural recursion to accumulative recursion using the N-puzzle problem as motivation to capture student enthusiasm and exploit what they have learned about program design. The N-Puzzle was also chosen to demonstrate that informed heuristic search strategies, traditionally the domain of undergraduate courses in Artificial Intelligence, are within the grasp of CS1 students. With proper guidance, CS1 students can reason such an algorithm into existence instead of simply using a textbook to study such algorithms. If the work described in this article is replicated elsewhere, there is no doubt that it will be an exciting time for Computer Science education and it will elevate the relevance of functional programming in the minds of future CS professionals.

1 Introduction

Based on the teaching philosophy of program by design (PBD) put forth in the textbook How to Design Programs (HtDP) [2], the use of functional video games to teach introduction courses to programming and Computer Science is a trend that is currently flourishing. At the heart of the PBD philosophy is the design recipe—a series of steps that students can follow to design and write programs.
These steps include the development of data definitions based on problem analysis, the development of contracts and function headers, the development of function templates for all data definitions, the specialization of function templates to create functions, and the development and running of tests. One of the most successful and promising implementation approaches to a PBD-based course uses a hierarchy of successively richer student languages and functional video games to get students interested and engaged in programming. PLT’s Dr-Racket integrates such a hierarchy of student languages for use in conjunction with HtDP. The reader should note that students are not taught Racket, but do learn Racket-like syntax on a need-to-know basis. This approach is successful, in part, because the student languages allow for the generation of error messages that are meaningful for beginners. This approach is also successful, in part, because functional video games provide a domain of interest to most Computer Science undergraduates and remove the need to reason about designing state-based programs. A plethora of examples exist that have students develop games exploiting structural recursion which resemble such classics as Space Invaders and Snake.

At the beginning of an introduction course, the focus is on solving problems using primitive data, structures, and structural recursion. Once students master the basics of structural recursion, the time comes to explore other forms of recursion such as generative and accumulative recursion. In generative recursion, the subproblems generated are not derived from the data structure employed and are in the same class as the original problem (a typical example is quicksort). One of the important consequences of this that beginners must realize is that programs using generative recursion are not guaranteed to terminate like programs that employ structural recursion. Thus, generative recursion requires the development of termination arguments. In accumulative recursion, one or more accumulators are added as parameters to a function designed using structural or generative recursion to capture information that, otherwise, would be lost between recursive calls (a typical example is finding a path between two nodes in a cyclic graph). An important consequence of this for beginners is that they must realize that for each accumulator an accumulator invariant must developed to describe the value of the accumulator. The code students write must guarantee that the accumulator invariant holds for every recursive call. It is up to the instructor to harness the enthusiasm and appetite that students have to develop more video games to motivate these topics. This requires identifying games that can not be played nor solved by only using structural recursion. It is important to note, however, that the goal is not to make students masters at developing video games. Instead, the goal is to make students interested in generative and accumulative recursion by showing them how they are needed and/or used in a video game. Surprisingly, there are not many examples in an HtDP-based curriculum of video games that require students to go beyond structural recursion.

This article advocates the position that video games ought to be used to motivate the need to study generative and accumulative recursion in the CS1 classroom. It presents an example on how to make the transition from structural