

The Quest for Excellence in Designing CS1/CS2 Assignments

Todd J. Feldman and Julie D. Zelenski
Department of Computer Science
Stanford University

ABSTRACT

We identify the principles involved in designing effective programming assignments for CS1/CS2. Through a case study of a particularly successful recursion assignment, we establish several prerequisites that must be present in the foundation of a potential assignment and discuss techniques for engineering exceptional assignments through changes to their more malleable components.

INTRODUCTION

Providing students with first-rate homework assignments is integral to the success of any course, and nowhere is this relationship so pronounced as in CS1/CS2, where the lion's share of what students really learn comes from their completion of several programming projects. Furthermore, the course material constantly builds upon itself and students must live with each assignment for one or two weeks at a time. Clearly, students cannot afford to have an inferior assignment deprive them of the educational rewards it was intended to deliver.

Our aim is to illustrate what it takes to create quality CS1/CS2 assignments by following an individual assignment through its genesis and refinement. Because recursion tends to be conceptually troublesome for students, it is especially important to have a focused and penetrating assignment for this material. We saw the word game Boggle[®] as an excellent basis for a recursion assignment within the framework of our library-based approach to programming that makes extensive use of tools for handling strings, I/O, and graphics [Roberts95a].

Boggle is played on a 4 x 4 board on which 16 lettered dice are randomly arranged. The object is to find sequences of four or more adjoining letters that form words; two letters adjoin if they are horizontal, vertical, or diagonal neighbors. No die can be used more than once in forming any one word. All players simultaneously write down whatever words they can find until time is called; each player receives credit only for those words not found by

This paper has been submitted to the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education in February 1996. Copyright is retained by the authors prior to publication release.

Boggle is a registered trademark of Parker Brothers.

anyone else.

Our adaptation of the game pits a single human player against the computer. Using an unlimited amount of time, the user enters all of the words that she can find, after which the computer searches the board to find and display any remaining words.

PREREQUISITES FOR EFFECTIVE ASSIGNMENTS

We first assigned Boggle in the fall of 1993. We required the students' programs to generate and draw the board, assemble a dictionary out of a provided abstract data type (ADT) and a file of words, check the validity of the user's words as they are entered, and perform the computer player's exhaustive search. Establishing the validity of a user's word entails confirming that it appears in the dictionary, verifying that it has not yet been used, and locating and highlighting the word on the board. In the following snapshot, the user has just entered the word "trails":



In coding the solution ourselves, we were pleased to discover that Boggle met our expectations as an especially illustrative exercise in recursion. Our students tackled the assignment with enthusiasm, embracing the challenges in both programming and playing the game, and related that they found the experience rewarding and enjoyable.

What made Boggle such a good choice? What can it teach us about crafting other good assignments? A careful analysis enabled us to answer these questions and establish the following prerequisites for effective assignments:

- *The material that an assignment is intended to teach must lie at the heart of the problem it poses.*

The selection of appropriate examples plays an important role in successful teaching. In a lecture setting, one can fashion isolated functions and code fragments into canonical examples that incorporate and illustrate the new

material with clarity and focus. On a larger scale, programming assignments, too, must prominently employ the constructs and techniques they are designed to teach. Peeling back the exterior of the Boggle program — its interface, I/O, etc. — reveals fundamentally recursive algorithms that use backtracking search to finding the user's and computer's words. The formation of words is so characteristically recursive that even students who are just becoming acquainted with recursion do not experience the usual temptation to apply more familiar iterative techniques. In fact, iterative implementations are practically impossible and therefore preclude serious consideration.

A common drawback of most recursion assignments is the fact that they exhibit only one of the two classical types of recursive problems. Some assignments involve the production of a single solution (e.g., a path through a maze), while others involve the production of all solutions (e.g., the permutations of a set). Boggle, however, exemplifies both types of recursion, one in finding the individual words entered by the user and the other in producing the collection of words that the user overlooked. As a result, students are able to explore both recursive paradigms as well as the differences between them.

- *An assignment must only be challenging with respect to the material it is intended to teach.*

It is counterproductive for students to squander their efforts on conceptual difficulties unrelated to the specific learning experience for which an assignment is designed. Instead, students must be able to focus their attention on the challenges of the new material. Free from unnecessary distractions, students will be able to channel their energy towards the proper goal; they will also be better able to appreciate how their solution to a particular problem is characteristic of the solutions to other problems of the same class.

The two recursive algorithms of Boggle are the only elements of the problem that are in any way challenging. Everything else is simple: the remaining functionality is essentially I/O, in the form of library-based string and graphics routines; the only data structures required are a two-dimensional array of records to represent the board and a provided dictionary ADT to store words.

- *An assignment must be engaging.*

In this age of Sega and Nintendo, it takes a lot more for a program to be 'cool' than it used to. With instructors taking the care to make assignments enticing and interesting, students look forward to working on their programs and want to get their code to work. As a result, students achieve far greater results than they otherwise would [Roberts95b].

We have been very successful in this regard by designing assignments with a strong audio-visual component (most or all of which is provided for the students through library

code) and a high degree of interactivity; students are excited by flashy programs in which they can get involved. Not surprisingly, games work particularly well, as Mary Poppins would agree [Sherman64]:

In ev'ry job that must be done
There is an element of fun;
You find the fun and — snap! —
The job's a game.

In Boggle, graphics and sound¹ combine with a challenging and addictive game to produce a program that students want to write because they enjoy running it themselves:

Just a spoonful of sugar helps the medicine go down
In a most delightful way.

- *An assignment must be accessible to all students.*

Since the true value of an assignment lies in how effective it is at fostering students' mastery of certain skills, it is imperative for all students to have a fair shot at attempting all assignments. Programming assignments should not be unnecessarily difficult in terms of either the complexity of the code that students must write or the approachability of the problems they must solve. Furthermore, assignments should be without bias towards students of any particular culture, gender, or academic discipline²; the problems involved should not require specialized knowledge of unfamiliar topics or attention to overwhelming details and special cases.

Boggle fits these criteria well: it is a straightforward and popular word game with clear objectives and concise rules.

- *The end result of an assignment must be worth the time and effort required to achieve it.*

When students see the end result of a programming assignment as something especially impressive, useful, or fun — a program they would like to have for themselves — they will approach the project with a heightened sense of interest and motivation. Writing a program that is worthy of being shown off to others does wonders for a beginning programmer's sense of accomplishment. These positive results will not only leave a student with good memories of the relevant assignment, but they will inspire the student on subsequent assignments as well.

On completion of their Boggle programs, students were amazed that they were able to write a program to solve a problem that seemed so difficult to them. The large branching factor associated with the formation of words is enough to confound most humans, yet students witnessed their programs' ability to find an incredible number of words with speed and completeness. Students accepted

¹We gave our students a sound-playing routine and lots of fun sounds with which to embellish their programs.

²Including computer science and engineering, since many students in CS1/CS2 have nontechnical backgrounds.

with good-natured humility the fact that the computer player consistently defeated them, and took much-deserved pride in the rare situations where the tables were turned.³

ENGINEERING EFFECTIVE ASSIGNMENTS

Although the first version of our assignment had important strengths, it was not without its flaws. On successive iterations of the assignment, we sought to clear up the aspects of the original version that were unnecessarily problematic. Our experiments in revising the assignment have yielded this formula for engineering an exceptional assignment from a strong foundation:

- *Focus the assignment.*

Clearing away nonessential components allows students to focus their energy on the more interesting aspects of the assignment. Students can be given the code for features of the program that would otherwise be eliminated. This works best when the functionality naturally separates into a cohesive and independent module. Ideally the code only loosely couples with the students' work and they use it only through its public interface.

This approach acquaints students with the techniques involved in working with code written by others: reading an interface, integrating its use into their own code, and debugging a program which has diverse contributions. These are useful skills for students to cultivate. We recommend distributing the module as source code if it would be accessible to the students at their current skill level. Source code provides a valuable debugging aid and furnishes a strong example of well-commented, cleanly written code.

The initial version of the Boggle assignment was too long. Giving students more time was not the right answer; we needed to eliminate the extraneous tasks that were crowding an already complex assignment. For example, drawing and highlighting the board introduced no new concepts and required tedious fussing to tune to perfection. In our revised version, we provided a module of graphics routines that the students could call at appropriate times, substantially reducing their time spent on the project's graphical aspects.

We gave our students the source code for the graphics module and encouraged them to adapt or embellish the routines. In truth, few students did, and most assignments were visually identical, lacking the creativity and individuality of the first batch (which was, perhaps, a minor downside of this change). Also, the sense of pride and accomplishment was diminished for some students, who felt they had only written "half" of the assignment. We countered this by assuring students that our code did nothing mystical or complex — that they could certainly write it if they were asked to do so — and encouraging

them to examine the source code to bolster their sense of proficiency.

- *Don't cut corners on materials provided to students.*

Students will not be able to fix any deficiencies in a compiled library, thus its functionality must be complete and efficient enough to properly support their work. The quality of any supporting data files is equally important. Admittedly, constructing data files with more than the barest bones is time-consuming, yet having access to interesting and comprehensive data often makes the difference between a "toy" result and a program that accomplishes something wonderful and impressive. An investment in exceptional materials can greatly enrich the students' experience without any change to the assignment itself.

Our original assignment suffered from inadequate and incomplete dictionary support. We provided a simple hash-table dictionary ADT in library form that supported the `Enter` and `Lookup` operations. In order to recognize when the exhaustive search of the computer's turn was futilely exploring a dead-end path, a prefix lookup was sorely needed. Unfortunately, this operation is not easily supported by a hash table, and in the original assignment we had to impose an artificial limit on the maximum word length to curtail the time-consuming and fruitless search.

In revising the assignment, we sought and implemented a dictionary better tuned for our needs. [Appel88] describes a highly efficient dictionary built from a directed acyclic word-graph, or *dawg*, a trie with unified suffixes as well as prefixes. The desired prefix lookup operation was trivial to implement. Now that students could identify and terminate pointless searches, we removed the unnecessary upper bound on word length.

The dictionary data file we first used was the standard UNIX `/usr/dict/words` file which contains about 25,000 of the most common English words. Unfortunately, it includes only stem words and none of their common derived forms such as plurals, past tenses, and gerunds. While playing the game, the dictionary would report that many simple and familiar words (e.g., "boats", "running", "happiest") were not in the dictionary. The students' programs seemed toy-like and the students themselves felt less accomplished, although this shortcoming was in no way reflective of their efforts.

In our later assignment, with a more space- and time-efficient dictionary, we were able to quadruple the size of the dictionary's vocabulary, with no increase in memory requirements and no degradation in speed.⁴ With these improvements to the dictionary and its data files, the end result was a much more realistic and playable game with virtually no changes to the students' part of the assignment.

³Students reportedly posted screen snapshots in their dormitories and computer clusters as testimonials to their triumphs.

⁴In fact our new dictionary was almost too large — it included obscure words that caused more than a few students to wonder if their program had a bug when it found one of the more unusual words!

Unfortunately, increasing the sophistication of the library code or the size and intricacy of the data files creates more work for already busy instructors. It also raises the possibility of introducing bugs, necessitating an extremely thorough testing phase to shake out errors before students stumble over them.

- *Don't add features indiscriminately.*

When the basic assignment concept seems a bit too simple, various extensions can be added to spice it up or to provide extra challenges for the more motivated students. However, many small details often make an assignment only longer, not harder. The best enhancements are those which complement the main issues of the assignment and help to further the students' mastery of the key concepts. The consequences of any addition to the assignment must be carefully weighed, for it might contribute more complexity than intended.

In place of the standard 4 x 4 Boggle arrangement, we asked students to accommodate variable-sized boards in the original assignment. This small extension introduced not only additional generality but also the need for dynamic memory allocation, creating an unfortunate mire of troubles for our students who were not yet firmly in control of the related issues. Although they did need to solidify their memory allocation skills, asking them to do so while also learning recursion reflected poor timing. We backed out of that decision and stuck with the usual fixed-size board in the revised assignment.

In contrast, we developed a few extensions that entailed interesting thought on the part of the students with only minor impact on their code: handling any occurrence of "q" on the board as "qu"⁵ or allowing dice to be used repeatedly in forming a single word. Both of these additions required students to wrestle further with the recursive code rather than a fringe detail.

- *Make the assignment easier for students to test and debug.*

CS1/CS2 students are still fairly new to programming, and they often aren't experienced with the gamut of debugging and testing techniques. To clarify the goals the students must attain, a compiled sample application can demonstrate the behavior of a working solution. Even better, a program can offer its own feedback, making it immediately obvious to the student whether it is working correctly, not only on the main tasks, but on the small details as well. A "back door" or "cheat mode" may be needed to allow students to directly construct the situations they need to debug and overrule decisions usually made at random.

The first version of Boggle did reasonably well at offering feedback. For example, when the user enters a word, the program must highlight it visually on the board; students can easily notice if their programs fail to find the word or highlight the wrong letters. However, other flaws can be harder to detect. No word can be used more than once, but in the first version of the assignment, there was no visible indication as to which words had been used; to verify that repetitions were caught, the students had to remember the words they had used. In the revised version, we provided a

⁵To make "q" more usable, the commercial version of Boggle combines it with "u" on a single die face as "qu".

score-keeping facility that displayed the players' word lists alongside the board. Now that it was clear which words had been used, students were better able to test whether their programs properly enforced the repeated word restriction.

Students also found it difficult to establish the veracity of their algorithms for the computer's exhaustive search. Something is obviously wrong when this search finds words that do not exist. However, since there are so many words to be found, it's much harder to tell whether any have been omitted. In the revised assignment, we added an option to the sample application that allowed the students to assign letters to locations manually. By forcing the board in the sample application to match a board generated by their own program, a student could observe the two programs, side by side, searching for words on the same board. If the two programs found the same words, the student could be more confident about the correctness of their code.

SUMMARY

Boggle has blossomed into one of our top assignments. It consistently rates very highly with our students and continues to defy our attempts to come up with a comparable alternative.

Our work with Boggle has convinced us to restrict our consideration of potential assignments to those that share its initial strengths. We labeled these qualities *prerequisites* because they are difficult or impossible to impart if they are missing from an assignment's fundamental concept. For example, one would be hard put to contort Boggle into an appropriate string assignment, because its peripheral use of a small subset of string operations cannot be shifted to the heart of the game. Similarly, computing to 500 decimal places is neither compelling nor engaging to the average student, and little can be done to make it so.

Boggle also demonstrates that even good assignments can be made better. Each time we prepare a new version of an old assignment, we are surprised at how much we are able to improve it; we evolve our assignments as we learn from our own experiences as well as the successes and failures of every new group of students. For example, our students have exposed several aspects of Boggle that can still be refined: we'd like to find a way to help them debug their code without getting lost in the depths of recursion and avoid the memory allocation problems that accompany recursive string processing. Although our students have not complained about the single (human) player limitation of Boggle in its present form, we are excited by the prospect of someday devising a competitive networked implementation of the game for multiple players.

Designing assignments is one of the most personalized aspects of teaching, and therefore also one of the most gratifying. We receive heartfelt joy and pride from

watching our students — many of whom have never touched a computer before — learn and mature from the programming experiences that we craft for them. We hope that in sharing our own discoveries and ideas in this paper we may inspire others to invest the extra time and effort in creating assignments that will return great rewards to both their students and themselves.

REFERENCES

[Appel88] Andrew W. Appel and Guy J. Jacobsen, "The World's Fastest Scrabble Program," *Communications of the ACM*, Volume 31, No. 5, May 1988.

[Roberts95a] Eric S. Roberts, *The Art and Science of C: A Library-Based Approach*, Reading, MA: Addison-Wesley, 1995.

[Roberts95b] Eric S. Roberts, "A C-Based Graphics Library for CS1," *SIGCSE Bulletin*, March 1995.

[Sherman64] Richard M. Sherman and Robert B. Sherman, "A Spoonful of Sugar", *Mary Poppins*, The Walt Disney Company, 1964.