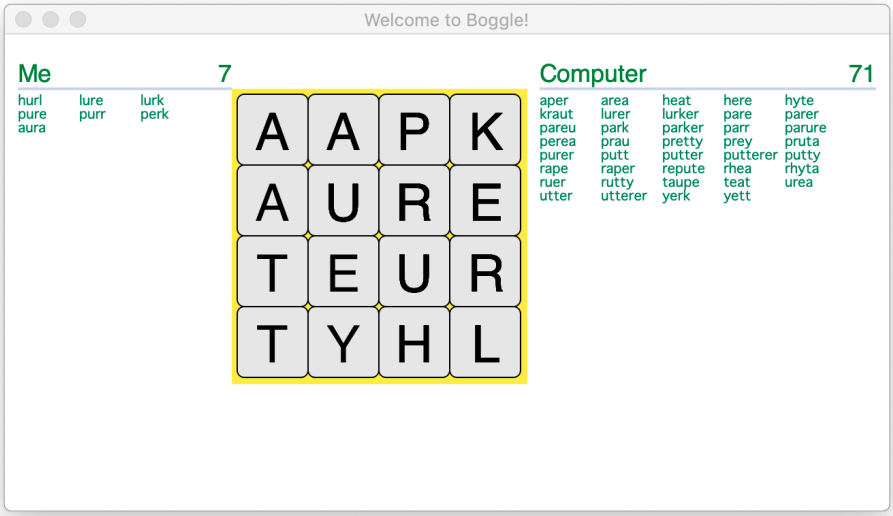
Assignment 3: Boggle

Thanks to Todd Feldman for the original assignment idea. And thanks to Julie Zelenski and Eric Roberts for the handout.



The Game of Boggle

Those of you fortunate enough to have spent summers seeing the world from the back of the family station wagon with the 'rents and sibs may be familiar with Boggle, the little word game that travels so well, and those who didn't will soon become acquainted with this vocabulary-building favorite. The Boggle board is a 4x4 or 5x5 grid over which you shake and randomly distribute 16 or 25 dice. These 6-sided dice have letters rather than numbers, creating a grid of letters from which you can form words. In the original version, the players start simultaneously and write down all the words they can find by tracing by a path through adjoining letters. Two letters adjoin if they are next to each other horizontally, vertically, or diagonally. There are up to eight letters adjoining a cube. A letter can only be used once in the word. When time is called, duplicates are removed from the players' lists and the players receive points for their remaining words based on the word lengths.

Due: Wednesday, January 30th at 11:59 pm

Assignment 3 asks that you write a program that plays a fun, graphical rendition of this little charmer, adapted to allow the human and machine to play one another. As you can see from the screen shot above, the computer generally pummels you into the ground, but it's fun to play anyway, and the assignment frames it as an interesting recursion problem.

The main focus of this part of the assignment is designing and implementing the recursive algorithm required to find all the words that appear in the Boggle board.

How's this going to work?

You set up the letter cubes, shake them up, and lay them out on the board. The application precomputes all words that can be formed and pairs them with information about where they appear. The human player gets to go first (nothing like trying to give yourself the advantage). The player enters words one by one. After verifying that a word is legitimate, you highlight the word's letters, add it to the player's word list, and award the player some points.

Once the player has found as many words as he or she can, the computer gets its turn. The computer identifies (and takes credit for finding) all the remaining words and awards itself all points. The computer typically beats the player mercilessly, but the player is free to try again and again and again until he or she feels inferior. ☺

The letter cubes

The letters in Boggle are not simply chosen at random. Instead, the letter cubes are designed in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. To recreate this, our starter code declares an array of the cubes from the original Boggle. Each cube is described using a string of 6 letters, as shown below:

```
static const string kStandardCubes[16] = {
   "AAEEGN", "ABBJOO", "ACHOPS", "AFFKPS",
   "AOOTTW", "CIMOTU", "DEILRX", "DELRVY",
   "DISTTY", "EEGHNW", "EEINSU", "EHRTVW",
   "EIOSST", "ELRTTY", "HIMNQU", "HLNNRZ"
};

static const string kBigBoggleCubes[25] = {
   "AAAFRS", "AAEEEE", "AAFIRS", "ADENNN", "AEEEEM",
   "AEEGMU", "AEGMNN", "AFIRSY", "BJKQXZ", "CCNSTW",
   "CEIILT", "CEILPT", "CEIPST", "DDLNOR", "DDHNOT",
   "DHHLOR", "DHLNOR", "EIIITT", "EMOTTT", "ENSSSU",
   "FIPRSY", "GORRVW", "HIPRRY", "NOOTUW", "OOOTTU"
};
```

These strings are used to initialize the cubes on the board. At the beginning of each game, "shake" the board cubes. There are two different random aspects to consider. First, the cubes themselves need to be shuffled so that the same cube is not always in the same location on the board. Second, a random side from each cube needs to be chosen to be the letter facing up.

To rearrange the cubes on the board, you should use the following shuffling algorithm, presented here in pseudo-code form:

Copy the constant array into a vector **vec** *so you can modify it.*
Shuffle **vec** *using the following approach:*
```
for (int i = 0; i < vec.size(); i++) {
```
Choose a random index **r** *between* **i** *and the last element position, inclusive.*
Swap the element at positions **i** *and* **r**.
```
}
```
Fill the Boggle grid by choosing the elements of **vec** *in order.*

This code makes sure that the cubes are randomly distributed across the grid. Choosing a random side to put face-up is straightforward. Put these two together and you can shake the cubes into many different board combinations.

Alternatively, the user can choose to enter a custom board configuration. In this case, you still use your same board data structure. The only difference is where the letters come from. The user enters a string of characters, representing the cubes from left to right, top to bottom. Verify the string is purely alphabetic and of the length required to fill the board and re-prompt if it is too short.

Precomputing all formable words

Once the board has been populated with letters, you should recursively search the board to find all words. A word is considered legitimate if:

- It is at least four letters long.
- It is contained in the English lexicon.
- It can be formed on the board (i.e., it is composed of adjoining letters and each cube is used at most once).

When a word can be formed in multiple ways, you ignore any possibilities beyond the first, retain information about all possible formations, or flip a coin to decide whether to discard a previously saved option with the one you just found. Because a word can only be counted once during play, it's only important you present some path on the board when the word is player. Words that differ only in capitalization scheme are considered the same (e.g. **"tree"**, **"TREE"**, and **"TrEe"** are all the same word).

As with any search algorithm with a high branch factor, it's important to find ways to limit the search to ensure that the process can be completed in a reasonable amount of time. One of the most important Boggle strategies is to prune dead end searches. For example, if you have a path starting **zx**, the **Lexicon**'s **containsPrefix** method should inform you there are no English words down that path. So, you should stop right there and move on to more promising search paths. If you ignore this suggestion, you'll find yourself taking long coffee breaks while the computer is looking for words like **zxgub**, **zxaep**, etc.

The human's turn

Once everything's been precomputed, you allow the user to enter as many words as he or she can find. When they enter something bogus, your program should reject their response with a helpful message saying why it was rejected (too short, not an English word, not on the board, been used before, etc.) using whatever message you want to use. If all is good, you add the word to the player's word list and score. In addition, you graphically show the word's path by temporarily highlighting the relevant cubes. You can use the graphics function **`pause`** to implement the delay. The length of the word determines the score: one point for a 4-letter word, two points for 5 letters, and so on. The functions from the **`gboggle.h`** interface provide helpful routines for highlighting cubes, displaying word lists, and handling scores.

The player enters a blank line when done finding words, which signals the end of the human's turn.

The computer's turn

The computer then searches the entire board to find the words not found by the human. The computer's already done all the precomputation, so it can take credit for all the words it precompiled and claim them as its own. The only words it can't post as its own finds are those the human player found during his or her turn.

Our provided code

We have written all the fancy graphics functions for you. The functions exported by the **`gboggle.h`** interface are used to manage the appearance of the game window. It includes functions for initializing the display, labeling the cubes with letters, highlighting cubes, and displaying the word lists. The implementation is provided to you in source form so you can extend this code in your own novel ways if you are so inclined.

The **`Grid`** and **`Lexicon`** classes you've already seen will come in handy again. Our English **`"dictionary.txt"`** file contains over 125,000 words, which is about four times as large as the average person's vocabulary, so it's no wonder the computer is so good.