Jerry Cain
CS 106AX

Handout #08
October 16, 2023

Assignment #3—Wordle

This assignment and assignment handout was written by Jonathan Kula, with help from Jerry Cain, Ryan Guan, Sophie Andrews. It has been modified by Avi Gupta.

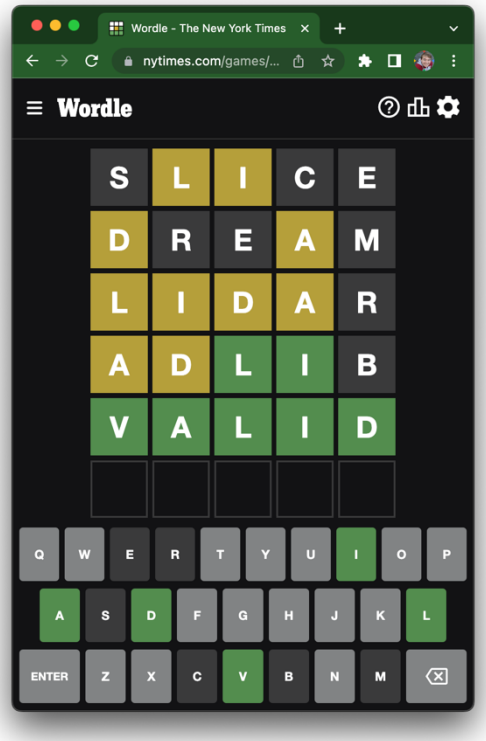
Due: October 23rd, 2023, at 5:00 P.M.

Your job for this assignment is to implement the word game by Josh Wardle that took the internet by storm—the one and only Wordle. This assignment will extend the skills you honed with Breakout and take them to the next level, while giving you some introductory practice with strings, arrays, and aggregates.

This assignment is also large. The primary challenge will be in developing a clean architecture for the **state** of your game, using that to update the **view** of your game (i.e., what the user sees), and handling user **actions** (e.g., clicking, or typing a letter). This is totally in your wheelhouse, and we'll help you structure your implementation so that everything stays manageable.

What Is Wordle?

Wordle is a deceptively simple word-guessing game created by Josh Wardle as a small game for he and his partner to play. The game went viral, and it was quickly picked up by the New York Times. In fact, you can play it on the NYT's website if you'd like to get a sense for how it's played.



In each game, there is a secret 5-letter word that is selected. It is the player's job to try and guess the word in 6 tries. Here's an example – each time I guess a word, some information is revealed:

- Some letters are highlighted in dark gray. This means those letters *don't* appear in the word at all.
- Some letters are highlighted in yellow. This means those letters *do* appear in the word, but *not at that location*.
- Some letters are highlighted in green. This means those letters are *correctly placed*.

Note that every guess must also be an English word! For ease of bookkeeping, as you make guesses, the keyboard at the bottom of the screen updates with the latest information.

Table of Contents

Learning Goals

Here's why we're giving this assignment, and what you can expect to take away from it:

- Students know how to and have practice managing more complex program state.
    - Students have practice designing their program's state.
    - Students have practice using that state to update a display shown to the user.
    - Students have practice updating that state given some user action.
- Students have shown mastery with use of event listeners and callback functions.
- Students are comfortable with basic string manipulation, including: taking substrings, determining character-by-character equality, etc.
- Students can access elements of, iterate over, and append to arrays.
- Students have successfully designed and implemented a complex game they can show off to their friends.

Getting Started: Tips & Tricks

Start by just playing a few different games of Wordle! (hello wordl is a good place to do this, as it will give you a new word each page refresh).

- Get a sense for how it plays. What happens when you win? What happens when you lose? What happens in edge cases– like, for example, if you press "enter" before typing in a full word, or type in an invalid word, or continue typing once all 5 spots are filled? What about if there are two of the same letter in a guess?
- As you're trying out Wordle, consider what the program needs to keep track of. For example, it probably needs to store and remember the secret word; what else might it need to keep track of?
- Thinking about these questions will kickstart ideas for things you'll need to keep in mind while you're designing your own version.
- Use your main **`Wordle`** function for debugging. Several times throughout this handout, we'll ask you to check your work so far by adding some throwaway test code to your **`Wordle`** function (e.g. to quickly add something to the **`GWindow`**– just to make sure it looks right!). This, combined with **`console.log`** can be a powerful tool to test and debug your code while you work on it!
- **Important Decomposition Tip:** We recommend keeping all the functions you create (for example, a function to create a guess square) *as inner functions within* the main **`Wordle`** function. In our solution, we add several functions inside **`Wordle`**, and no other functions inside of those.
    - This allows you to easily access the "state" (ex: the secret word) without passing those variables as arguments into all your helper functions.

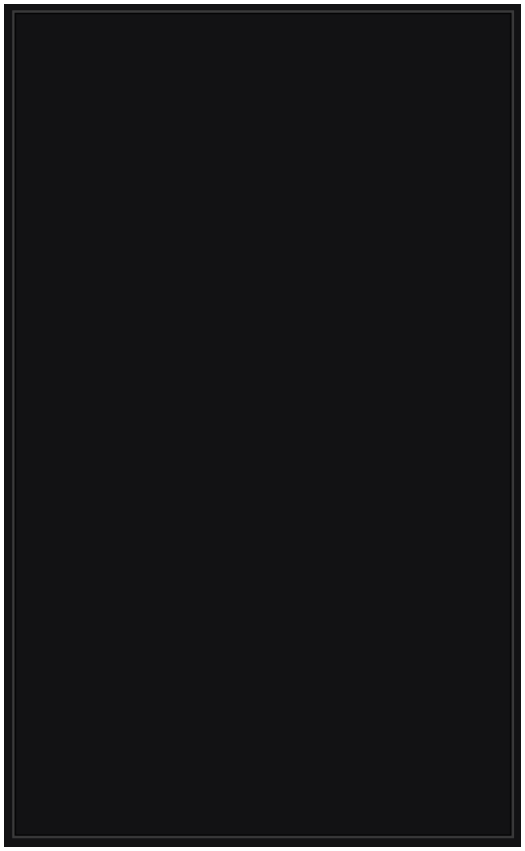
Once you have a good sense for what Wordle is, I recommend reading through the entire handout before getting started on Part 1. Knowing where you're going and letting those

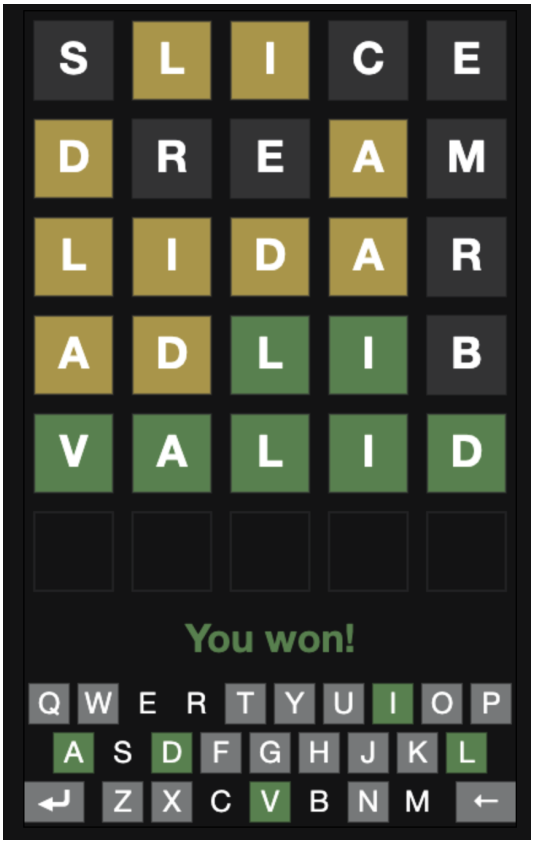
ideas marinate while you work will help provide direction as you work on earlier parts, and you'll come to later parts feeling more prepared.

Part I: The View

"The view" refers to the display: This is what the user sees and interacts with. We're going to sketch out (in code!) what your game of Wordle will look like; then, later, we're going to start thinking about (1) how to store what's currently going on in a game of wordle (the state), and (2) connecting that state (i.e. what's going on) to the view (i.e. making it so when you type letters or guess a word, that's all reflected in what a user sees!)

Here is where you'll start, and what your completed game of Wordle will look like**:**





We'll build up to it one piece at a time!

Object 1: The "Guess Square"

Given a letter guess, a provided color, and a position on the GWindow, write a function that produces a "guess square."

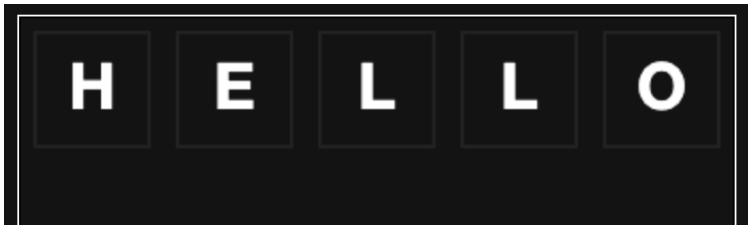
Tips:

- Once again, we recommend writing all helper functions in this assignment as *inner functions* within the `Wordle` function.
    - Style: Please write brief comments above each function describing the functionality. This will help you revisit your code and also will help the SLs understand your code!
- Do not assume that the provided letter will be capitalized.
- A "guess square" should be sized using the **`GUESS_SQUARE_SIZE`** constant.
- You'll need to package together two **`GObject`**s into a compound placed at the provided position. They are:
    - **`GRect`** for the background square. Note in the completed game image that although each guess square might be filled with a different color, each square always has a light gray border color, making it visible even if no guess is present.
    - **`GLabel`** for the letter. You'll find the **`GLabel`**'s **`setTextAlign`** and **`setBaseline`** methods particularly helpful in centering the text. You can also set the guess font and text default color using provided constants.

Decomposition check! Edit **`Wordle()`** to call the function you made to produce a guess square, and add it to the **`GWindow`**. You should be able to draw any of the following individual squares at any position on the **`GWindow`** by simply calling the function with different arguments.

Object 2: A "Guess Row"

Now that we're easily able to create a square that will hold a letter guess, it's time to leverage that function to create a row of squares at any vertical position that could be used to hold a player's word guess. It will look like this:

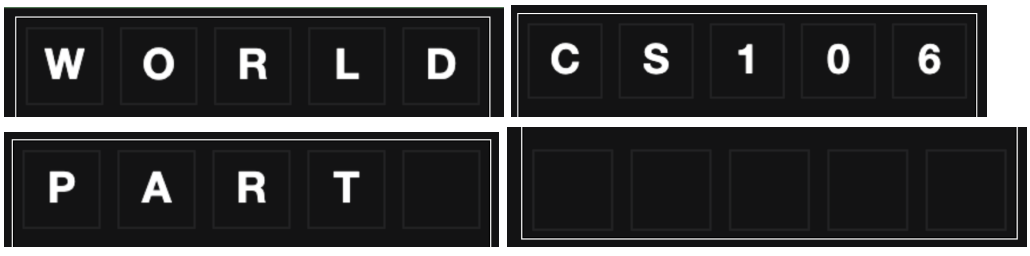


Tips:

- You'll find the **`NUM_LETTERS`** constant useful here.
- Note the spacing between each guess square; each of them is surrounded on all sides by a distance of **`GUESS_MARGIN`**. (This means that if **`GUESS_MARGIN`** is 8, there is an 8px gap on either side, and 16px between each guess square).
- Also note that the margin and **`GUESS_SQUARE_SIZE`** are calculated such that it will exactly fit in the window width if done correctly; this also means you can always assume that we start at x = 0, i.e., the far-left side of the **`GWindow`**.

As you're coding up your guess row, you might be thinking about how you would highlight each individual square. If you are, that's great! Continue thinking about this, but for now, go ahead and keep them all some default color. We'll connect colors to squares later when we start thinking about state.

Decomposition check! Call the function you created to produce a guess row, and add it to the GWindow. You should be able to go between any of the following by changing the parameters to that function call. You should also be able to freely reposition the row anywhere in the window by modifying parameters of that same function call.



Object 3: Guess Grid

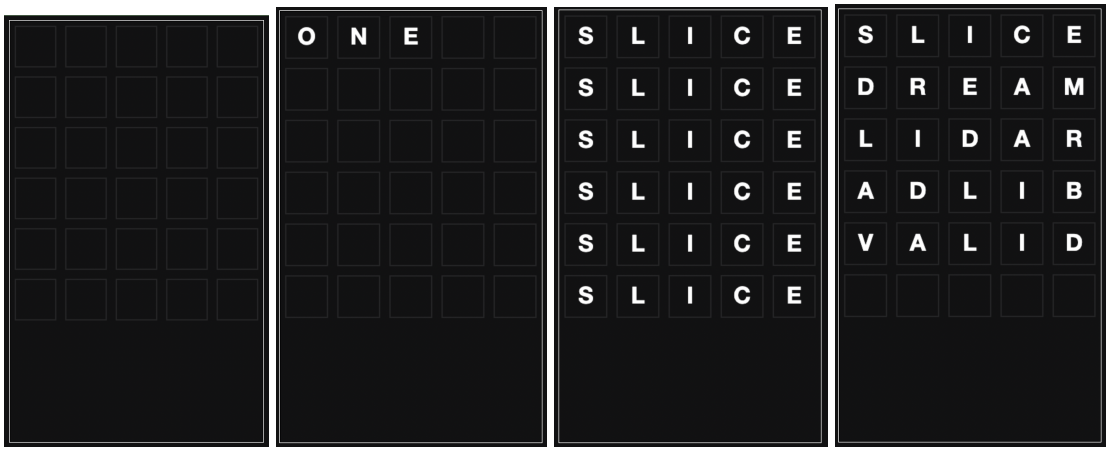
You've got squares. You've got rows! Now, it's time to put them all together:

Tips:

- You'll find the **`NUM_GUESSES`** constant useful here.
- Note, again, the use of the margin: there is only one **`GUESS_MARGIN`**'s worth of space next to the top, but there are two **`GUESS_MARGIN`**'s worth of space between each row.
- Again, don't worry about highlighting right now! We'll take care of this in the next part.

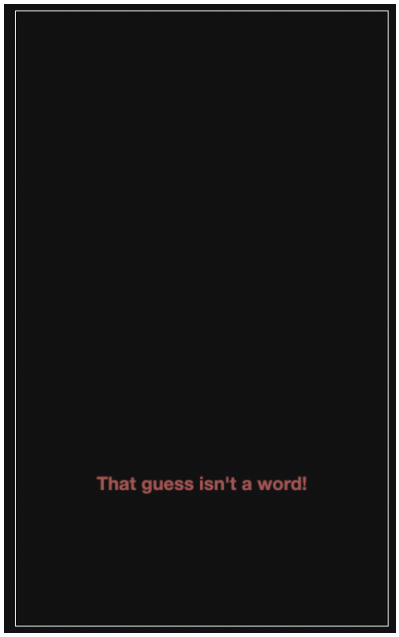
Decomposition check! For now (until Part II), hard-code a data structure (hint: array) of fake word guesses and use this for testing. Syntax reminder: If you want to write an array yourself, you can write something like **`["hello", "world"]`** – this will create an array with 2 entries, being the strings **`"hello"`** and **`"world".`**

Call your function to produce a guess grid, and add it to the GWindow; by changing the parameters to that function, you should be able to produce the following:



Object 4: Alerts

Now that we have the grid established, we need a way to tell the user about things that happen (in various colors):



- We've precomputed where the alert should go; check out the **`ALERT_X`** and **`ALERT_Y`** constants.
- Again, you'll find the **`GLabel`**'s **`setTextAlign`** and **`setBaseline`** methods particularly helpful in centering the text.

Decomposition check! Call your function to produce an alert and add it to the GWindow; by changing the parameters to that function, you should be able to produce the following:

Object 5: The Keyboard

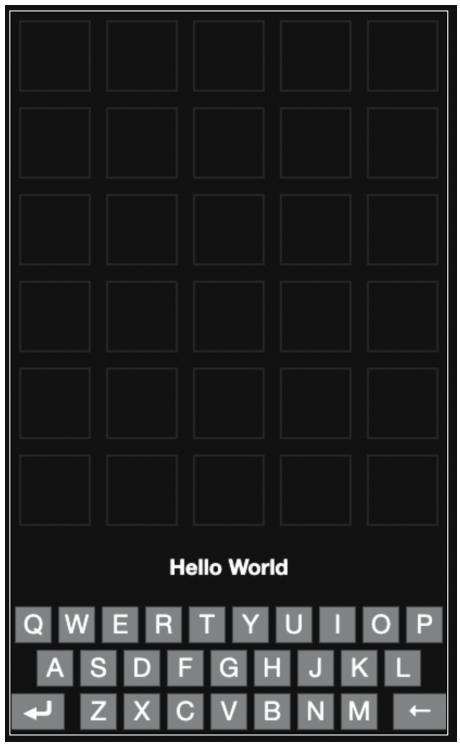
The final individual step is to place the keyboard down on the grid; we provide a new **`GKeyboard`** object. Take a look at the **`GKeyboard.js`** file and read the header comment to understand how to use it.

Tips:

- We've pre-computed the location to place this keyboard at: check out **`KEYBOARD_X`** and **`KEYBOARD_Y`**.
- It should be as wide as the **`GWindow`**.

And now: Putting it all together

Finally, create a function called **`draw`** that (1) clears the **`GWindow`**, and then (2) draws the **`GuessGrid`**, an **`Alert`**, and a **`GKeyboard`**. Verify that you can change **`NUM_GUESSES`** and **`NUM_LETTERS`** and your code handles it gracefully. (It's OK if the letters get too big; you can adjust the **`GWINDOW_WIDTH`** to accommodate high numbers of letters).



Part II: Program State

You have a swanky view all set up now. Great job!! However, right now, the view doesn't really *do* or *respond* to anything. The brain of our program doesn't really function yet.

We're going to fix that now!

We're going to start with **designing our program state** and **making the view use this state**. Then in Part III, we'll link together player actions to our state.

Task 1: Designing State

We need to tell the "brain" of our program *what it needs to keep track of*. This accounting makes up the **state** (also known as the **model**) of our program. You can think of the state simply as variables that all of your inner functions can use. State should be minimal to make our code as simple as possible; we shouldn't add top-level variables that are redundant.

Consider the game of Wordle. What does it need to keep track of in order to work?

- One such example is the secret word. What other information do you **need** to store?
    - For example, do you need to store the text the user has typed in?
    - What about whether or not a *particular letter* should be colored gray, yellow, or green? Do we need to store that, or can that be calculated on the fly from the other information you're storing?
    - What about whether or not you should be showing an alert?
    - How does your state keep track of if a player has simply typed in 5 letters, versus having submitted a guess? (Does it need to?)
- You can use any types you wish for your state; some things may make sense as strings; others may make sense as arrays; etc.

Once you have figured out what you need to store, these can all become **local variables** placed at the start of the **top-level Wordle function**.

This may be an iterative process. Make your best-informed decisions now, and if you discover that some state is no longer necessary, refactor your code to remove it; or if you discover you need more state, feel free to add it.

Task 2: Linking State & View

Now that you've designed the brain of your program, it's time to use that brain to update our view.

Update your **draw** function, and then your guess grid and guess row functions, to use the **current state** of your program. Between the arguments to the functions you've already designed and the state of the program, you should be able to add the following functionality:

1. The text in your view should match the state in your program.
    a. Make sure this is working before moving on to highlighting! You should be able to reproduce the guess grid on page 7 just by editing your state.
2. The squares should have colors that are accurate to the rules of Wordle:
    a. In each guess, a letter should be highlighted in **gray** (`BACKGROUND_WRONG_COLOR`) if that letter does not appear in the word at all.
    b. A letter should be highlighted in **green** (`BACKGROUND_CORRECT_COLOR`) if that letter appears in the word at that location.
    c. A letter should be highlighted in **yellow** (`BACKGROUND_FOUND_COLOR`) if that letter appears in the word, but at a different location. A few nuances to think about:
        i. If I guess `"HELLO"` for the secret word `"WORLD"`, the second `"L"` should be green and the first `"L"` should be gray, not yellow.
        ii. As another example of this behavior, if I guess `"REARS"` for the secret word is `"TRAIN"`, the first `"R"` should be yellow, but the second one should be gray.
        iii. What do you expect to happen if the secret word is `"LLAMA"`and the guess is `"TROLL"`?
3. Your keyboard should update to the correct colors as well (hint: check out the documentation for **`setKeyColor`** in **`GKeyboard.js`**)
    a. Consider that unlike the letter highlighting in the squares, keyboard highlighting partly depends on previous guesses.
    b. An edge case to think about: If a letter is correct (green) in the first guess, but in the wrong place (yellow) in the second guess, your keyboard should still show the color green after the second guess.

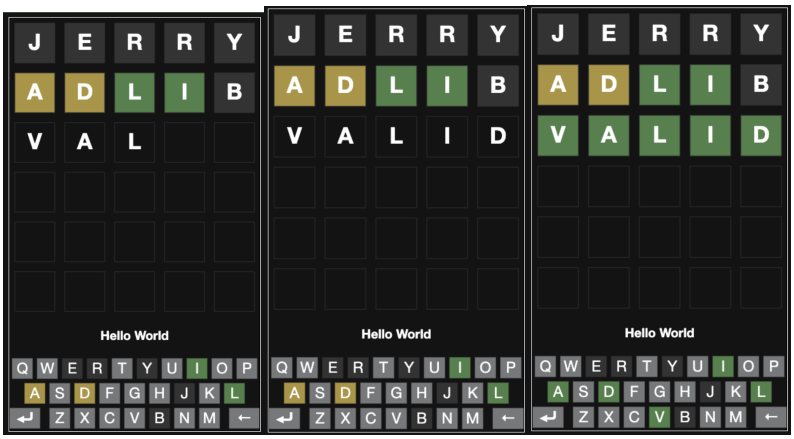
Keep decomposition in mind as you work towards accomplishing this task; as just a single example, you might create a function that, given a guess and the secret word, returns an array of indices in the guess that match the secret word. (e.g. a guess of `"HEL`**`L`**`O"` for the secret word `"WOR`**`L`**`D"` could return `[3]`), and you know that these letters should be highlit green. By decomposing in this way, you can also quickly test

your functions right inside **`Wordle()`** (for example, by just adding a couple extra **`console.log`** statements!) without relying on all your other code.

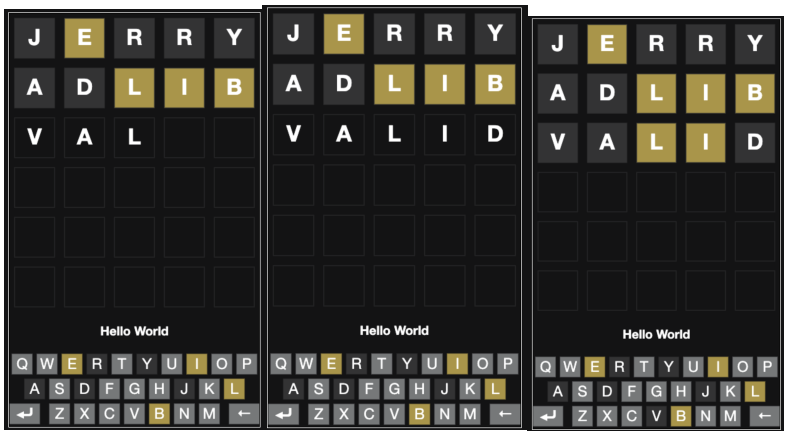
When you're done testing, you can use the **`getRandomWord()`** function to retrieve a random common word from the dictionary to use as your secret word.

See the next page for some examples of what you should be able to do by **manually editing your state** (i.e., preloading your state variables with certain values; no usage of input from keyboard yet).

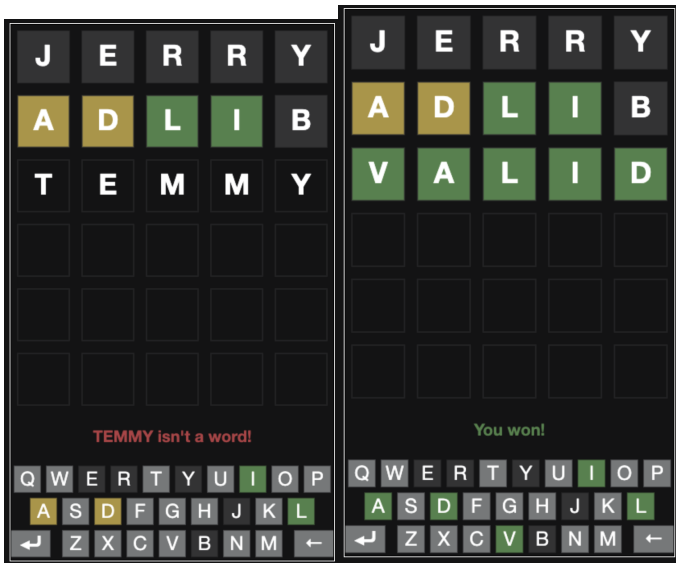
At this point, if you hand-edit your state in the `Wordle()` function, you should be able to produce any of the screenshots below: (For this first set, the secret word is "valid"):



If you change the secret word to "libel," it should look like this:



And you should also be able to set and clear alerts by hand-editing your state:



Part III: Interactions

You now have your **state** connected to your **view**! You're almost to the end of this road; now, in order to change what's displayed, all you have to do is change those state variables and call **`draw`**.

The high-level procedure is: whenever a user types (or clicks) a key, you update the relevant state and re-draw the board.

This is a common pattern called MVC ("Model-View-Controller") that you'll see all the time: you *model* your application in some way (this is the state you created!). The **view** (i.e., the **`GWindow`**!) is created from the model (i.e., program state). Then, finally, when the user interacts with **controls**, that will manipulate the state (which updates the view).

Task 1: On-Screen Keyboard

Your first task is to get the on-screen keyboard working (this is 90% of the work in this part, as the code you write here should be easily reusable for your physical keyboard).

If we take a look at the **`Gkeyboard.js`** file's documentation, we'll find that we can use **`addEventListener`** to listen to a few different **keyboard** events. In particular:

- We can use the **`keyclick`** event to trigger a function when a user presses a letter. (What is the parameter(s) that the callback function accepts? How do you know which letter was clicked?)
- We can use the **`enter`** event to trigger a function when a user presses the enter button on the on-screen keyboard.
- We can use the **`backspace`** event to trigger a function when a user presses the backspace button on the on-screen keyboard.

Pay special attention to the letter casing you expect. We want to display all text uppercase, but the **`keyclick`** event gives you letters in lowercase; etc.

You will need to implement the following interactions while considering the following questions:

- When a user presses a letter, you should add that letter to their current guess-in-progress.
    - What happens if a user presses a letter when they already have the maximum number of letters?
- If there is an alert present (e.g., about a word not being valid), you should clear it when the user presses a letter.
- When a user presses backspace, you should remove a letter from their current guess-in-progress.
    - What happens if a user presses backspace when their guess-in-progress is already empty?
- When a user presses enter, you should submit their guess if it is valid.
    - You should notify the user and not accept their guess if it is not a valid English word. (Check out the provided function, **`isEnglishWord`**)
    - You should not accept their guess if it is too short.
    - After you accept their guess, if they have won the Wordle, you should notify them (and likewise if they have lost).
- If the game is over, you shouldn't accept any new events.

Task 2: Connecting to the real keyboard

The last thing you need to do is connect the real keyboard to your program, so you don't have to find-and-click each of the letters!

The **`GWindow`** responds to **`keydown`** events that allow you to intercept when a user presses a key. You can use the provided **`getKeystrokeLetter`**, **`isEnterKeystroke`**, and **`isBackspaceKeystroke`** with the event object to make decisions about what to do. You should be able to easily connect this to the callback functions you wrote for Task 1.

And with that…

Part IV: Bask in your success!

If everything has gone smoothly, **you have just implemented a fully operational version of Wordle!** Give yourself a pat on the back– this is not a trivial piece of software, and you've done it within just the first 4 weeks of the class. Share it with your friends and your classmates!!

General Advice

- *Start early*. This is a large assignment, and we are happy to help you at each stage of the process. The assignment is also brand new, so we don't have a history to rely on to anticipate what your pain points will be.
- *Be thoughtful about design*. Bad design tends to snowball. Having cleanly decomposed code at each stage will make the next stage significantly easier. We're happy to help you in office hours!!
    - When you make design decisions, ask yourself why you chose to design it that way, and write it down somewhere for yourself (for example, in an inline comment). You'll thank yourself later when you wonder why you made those decisions in the first place!
    - Don't be afraid to iterate either. You may decide that some design choice that sounded good at one point eventually turns out to be a bad one. It's OK—in fact, it's perfectly normal—to reconsider previous choices. If you documented why you made the decisions you did, you'll be able to decide whether you should stick with the original decision or change it.
- Don't be afraid to ask for help!
- *Test often*! If you follow the order of instructions in this handout, you should be able to test each individual component, object, and task as you write them.

Possible Extensions

- *Animations*. In the NYT's Wordle, their guess squares bounce when a letter is typed, and squares flip over when the correct word is found.
- *Implement hard mode*. Require the user to use letters revealed in prior guesses. If a guess correctly places a letter, then all subsequent guesses must place that same letter in the correct location. And if a guess identifies a letter that belongs somewhere else in the word, all subsequent guesses must include that letter somewhere as well.
- *Multi-wordle*. You could add additional concurrent Wordle games, like what Quordle does.
- *Allow replay.* Prompt the player to play again and reset the state of your application to allow that.
- *Keep score.* Keep track of how well a player is doing and give them a score. There are many ways you could do this, of varying complexity
- *Something else?* There are plenty of ways to extend Wordle– we're excited to see what you come up with!