Assignment III: CodeWord Breaker

Objective

It is time to venture off and build your own application. This application may be very familiar to you! It's also extremely like Code Breaker, so hopefully you'll be able to heavily leverage what we've done in lecture so far this quarter.

You must build a version of Code Breaker (you can name this application anything you want) with codes (both the master code and all guesses) that are *words in the English language* (instead of the codes being some colored circles or some emoji). This means that there are 26 peg choices (all 26 letters in the English language). That would make for a relatively difficult game, so instead of using your `MatchMarkers` to show the matches of a given attempt, you can show the user directly which pegs are exact matches and which are inexact matches.

Do not modify your Code Breaker from A2. Start a new application.

Be sure to review the Hints section below!

Due

This assignment is due before the start of lecture 9.

Materials

You are welcome to use all of the code (i.e. the text inside .swift files) from all lectures and from your solutions to Assignments 1 and 2. But you must use it all in the context of a **new application** that you create from scratch.

Be sure to check out the posted code from lecture 6 onwards.

You'll probably find the `Words.swift` file included in the lecture 6 code upload helpful.

Do **not** use AI. The code in your application must be entirely your own work (other than what was shown in lecture). **Do not import code from the internet or AI.**

Required Tasks

1. You must start this application as a **new application** (i.e. not a copy of an existing one). Think carefully about what you want to name it (read through the rest of the Required Tasks before you decide). Do NOT call it CodeBreaker. You can copy and paste *code* from your A1 and A2 and from all lectures into this new application as much as you need to, but don't copy *an entire project*, just *code* from files.
2. Create a new word-based Code Breaker application. It should have all of the features in Code Breaker thru at least lecture 6 with two exceptions:
   a. All codes (the master code and all attempts) **must be valid words in the English language** (this means colored circles and emojis are no longer valid codes and that there are 26 peg choices, the letters in the English language).
   b. The **match state** of each "peg" must be shown **on the peg** (i.e. not off to the side in `MatchMarkers`). You can use whatever colors/adornments you want to differentiate between an `.exact`, `.inexact` or `.nomatch` match.
3. Since it must be possible to guess any code, your "peg chooser" will have to contain all the letters in the English language. You must lay out the buttons in this "peg chooser" in the arrangement of a **QWERTY keyboard** (i.e. a typical English-language keyboard).
4. Required Task 2 means that you still must support restarting the game and must support words varying in length from 3 to 6 letters.
5. Make your application animate nicely. Much of this will involve incorporating all *applicable* animation work from lectures 7 and 8 into your application (i.e. all animation from lectures 7 and 8 that makes sense in your app's UI). It might involve a little bit of extra animation work that is specific to your UI too. It all depends on your UI.

Hints

1. Here are some simple changes you could make to L6's code to jump-start your solution. It should get you playing a skeletal version of the application, albeit with plenty of work left to do. Using this code is absolutely, positively NOT a Required Task. We're just trying to give you a boost here if you're struggling to know where to start. Feel free to ignore it all if you want.

   In `CodeBreaker.swift` (`Peg` is now a `String`, just like in A2; choices are all letters) …

```
typealias Peg = String

struct CodeBreaker {
    var masterCode: Code = Code(kind: .master(isHidden: true))
    var guess: Code = Code(kind: .guess)
    var attempts: [Code] = []
    let pegChoices: [Peg]

    init(~~pegChoices: [Peg]~~) {
        pegChoices = "QWERTYUIOPASDFGHJKLZXCVBNM".map { String($0) }
        masterCode.randomize(from: pegChoices)
    }
```

   In `Code.swift` (a convenience for setting the `pegs` in a `Code` to a word) …

```
struct Code {
    var kind: Kind
    var pegs: [Peg] = Array(repeating: missing, count: 5)

    static let missing: Peg = ""

    var word: String {
        get { pegs.joined() }
        set { pegs = newValue.map { String($0) } }
    }
```

   In `PegView.swift` (pegs aren't `Colors` anymore)…

```
    var body: some View {
        pegShape
            ~~.foregroundStyle(peg)~~
            .stroke()
            .overlay { Text(peg) }
```

In `PegChooser.swift` (You'll have to do something to be able to choose letters until you get your Qwerty keyboard working. The code change below will make 26 super tiny buttons, so it's better than nothing, though not that useful. Maybe an early step in your development is to at least update the peg chooser to show 3 rows of letters even if they are not strictly in a Qwerty arrangement?)

```
Button {
    onChoose(peg)
} label: {
    Text(peg)
}
```

2. The above Hints only get you started with 5 letter words and there's no restart button, so you'll have to apply what you learned in A2 to A3.

3. You can get English-language words for the master code for your game from anywhere you choose, but the easiest source is probably the provided `Words.swift` (found in the L6 demo code download). It creates an `@Environment var` called `words` which you can use to look up random words by length. **Only access these words in your UI (not in your Model!)—this means you'll need to pass the master code for a given game into your Model from your UI code.** If you do want to use `Words.swift`, make sure you have this at the top of any `View` that wants to use `words`.

```
struct CodeBreakerView: View {
   @Environment(\.words) var words
```

Then simply use the `random(length:)` `func` in `Words.swift` to generate random words.

4. The words are loaded from the internet, so if you have a bad connection, it might take a little while for them to be available. Here's a cool view modifier that might help with that initial startup "lag" possibility. You can apply this view modifier to pretty much any `View` inside of the `var body` of your main game-playing `View` (e.g. the top-most `VStack`). It will react to when the `words` `@Environment` variable actually gets updated to have some words in it (after it finishes downloading them from the network, which could take anywhere from milliseconds to seconds to forever if the network is down). This also shows you how to get a random word of a given length out of `words`.

```
.onChange(of: words.count, initial: true) {
    if game.attempts.count == 0 { // don't disrupt a game in progress
        if words.count == 0 { // no words (yet)
            game.masterCode.word = "AWAIT"
        } else {
            game.masterCode.word = words.random(length: 5) ?? "ERROR"
        }
    }
}
```

Note that if `onChange` notices the words arriving after the game has already started (`game.attempts.count > 0`), you're stuck playing with `AWAIT` as the word. Also note that once the words have arrived, if a word of length 5 cannot be found (should never happen, but you never know!), the user will be playing with the word `ERROR`.

This `onChange(of:)` only activates twice: when the game `View` first appears (because of the `initial: true`) and when `Words.swift` finishes loading some words (`words.count > 0`). So all other times a master code is needed (e.g., when a game *restarts*), you'll have to get it out of `words` yourself (the code above will not do it for you).

Hopefully by the time the user hits restart, the `words` will have been loaded from the network, but you'll have to do something sensible if not (kind of like `onChange` is trying to do above at startup).

Finally, note that this `onChange(of:)` assumes 5 letter words. You'll have to adjust it when you support 3, 4 and 6 letter words.

5. The file `Words.swift` also adds a `func` called `isAWord` to SwiftUI's `UITextChecker`. It returns whether a given `String` is a word in the English language or not. You can feel free to use this to test whether a given user *guess* is a valid word. `isAWord` is case-sensitive (since it can tell the difference between proper nouns or not), so you'll probably want to make the words you're testing with it be all lowercase (you can use `.lowercased()` for that). You could use the `words` `var` mentioned above to validate user guesses, but `words` only contains "common words" (which makes some sense for choosing the hidden master code word, but might be too restrictive when it comes to validating the user's guesses). This is all up to you. As long as you are meeting the Required Tasks above, you can choose master code words and validate guesses however you think is best. Here's an example of using `UITextChecker` to validate your user's guesses as English words …

```
@State private var checker = UITextChecker()

if checker.isAWord(guess) { … }
```

6. Summary of Hints 3 and 5 above: use `words.random(length:)` to chooser your master code and `UITextChecker`'s `isAWord()` to test that your guesses are valid English words. Again, this is only a Hint, not a Required Task.
7. While working on your application, it might help to specify `.master(isHidden: false)` in your `CodeBreaker` code temporarily so you can see what the master code is. Once you have things working, you can set it back to `true`.
8. Just to be clear, you will **not** be using `MatchMarkers.swift` in this assignment.
9. Since you no longer have match markers, your UI is now very much about how you draw each peg (since you have to show the match status there). You might want to enhance your `PegView` to take some more arguments to specify how to draw the different kinds of "peg looks" that are involved.
10. It's actually quite possible that your code for drawing `Codes` (in `CodeView`) might get *simpler* because of this, not more complex. The selection, for example, might just be a different way of drawing a peg (including how to draw a `.missing` peg). Maybe that's even true for the hidden master code? Of course, it all depends on how you choose to represent matches.
11. It is generally recommended that you do **not** bring any Challenge Mode work you did in A2 over to A3 (just to make your A3 code simpler). Most of them don't make sense anyway. Again, this is just a Hint, not a Required Task.
12. The Qwerty keyboard task is mostly about mastering layout. Layout can be a bit tricky, so don't be frustrated if your first attempts don't come out how you expect.
13. Aspect ratio can be a very good friend in layout. For example, your Qwerty keyboard has a well-known aspect ratio (i.e. the ratio of the number of keys in the top row (10) to the number of rows (3)). Similarly, you probably want each key to have the same aspect ratio. Enforcing both of these might help SwiftUI better understand what you want when it is laying things out.

14. The last Required Task (animation) requires Lectures 7 and 8, so save this for the end. If you really do an awesome job with animation (i.e. do something really cool above and beyond the sorts of things shown in lecture), there is some room in the Challenge Modes below for us to give you a little bit of extra credit for doing so.

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Project does not build without warnings.
- Code is visually sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, long methods, etc.

Often students ask "how much commenting of my code do I need to do?" The answer is that your code must be easily and completely understandable by anyone reading it. You can assume that the reader knows the iOS API and knows how the Code Breaker game code from the lectures works, but should <u>not</u> assume that they already know your (or any) solution to the assignment.

Challenge Mode

Challenge Mode is an opportunity to expand on what you've learned this week and push yourself with more difficult exercises. Attempting one or more of these each week is highly recommended to get the most out of this course.

1. 2 pts. On each key of your Qwerty keyboard, give the user feedback showing the *best result* (`exact`, `inexact`, `nomatch` or "not tried yet") that the user has ever gotten with that key *in **any** attempt* so far in this game. You'll probably want to use the same (or similar) visual appearances as you use in the game itself to indicate these things on each key.
2. 1 pt. Instead of choosing a *random* code length (i.e. between 3 and 6), add UI to your application that lets the user **choose** what code length to play when restarting a game.
3. 1 pt. Integrate your guess button and a "backspace" key into your Qwerty keyboard somewhere. The backspace key backs up the selection one space and removes any letter in the new space it reaches. This is mostly a layout challenge.
4. 1 pt. Meaningfully use a tuple and/or an `extension` in your solution. We haven't talked about tuples in lecture (except mentioning `TupleView`, but that's not related to this), so this is about understanding your reading assignment and applying it. As for the `extension`, you'll only earn this point if the code you add in the `extension` would not be better placed somewhere else.
5. 1 pt. We might give a point if you do some extra animation work. How much? It would have to be something that really demonstrates a thorough understanding of some of the animation mechanisms. Make us say "wow, cool, nice animation(s) there." We're not going to "pre-approve" your animation choices, so this Challenge Mode is a bit of a dice roll as to whether you'll get the extra credit, but the journey is the reward, so, at worst, you'll learn a bit more about animation!