iOS: Staff's Choice

Evil Hangman*

due by Wed 8/10, noon

Ingredients.

- Cocoa Touch
- Evil
- MVC

* Inspired by Keith Schwarz of Stanford at SIGCSE'11.

Academic Honesty

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed by some project. Viewing, requesting, or copying another individual's work or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available your or other students' solutions to projects to individuals who take or may take this course (or CSCI E-76) in the future.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. You may also turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on projects. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

If in doubt as to the appropriateness of some discussion or action, contact the staff.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is disciplinary action, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Grades.

Your work on this project will be evaluated along four primary axes.

Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Scope. To what extent does your code implement the features required by our specification?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

Hangman.

☐ Hello, human. I am thinking of a seven-letter word:

```
_______
```

Okay, what is it? Hm, no idea? Okay, I'll allow you to guess letter by letter. Guess a letter, and I'll tell you where, if anywhere, it appears in the word. You can guess incorrectly up to, oh, six times. If you can figure out the word, you win; if you can't (and you're out of guesses), I win. Ready? Okay, guess a letter.

`A`, you say? You are so smart. That letter appears twice:

```
-A---A-
```

Recognize the word now? No? S'ok, you still have six chances left, since your first guess was a good one. Guess again.

`E`, you say? Nope, sorry, not so smart after all. You've now used up one of your six chances, so you have five chances left. Guess again.

`I`, you say? Nope, double fail. You have four chances left. Guess again.

`Z`, you say? Really? No. You now have three chances left. Guess again.

`N`, you say? Nice! Finally. That letter also appears twice:

```
-AN--AN
```

You still have three chances left. Guess again.

`E`, you say? Um, you kinda guessed that already. I'll pretend I didn't hear that. You still have three chances left. Guess again.

`H`, you say? Nice! That letter appears once:

```
HAN--AN
```

Recognize the word yet? `G`, you say? `M`, you say? That's right! It's `HANGMAN`!

☐ So that is how the age-old game of Hangman is played. But for this project, you will not implement Hangman. (Hangman is boring.) You will implement….

Wait for it…

Evil Hangman.

Evil Hangman.

☐ Evil Hangman is quite like Hangman, except that the computer cheats. Rather than pick, say, a seven-letter word at the game's start, the computer instead starts off with a mental list (well, maybe a set or array) of all possible seven-letter English words. Each time the human guesses a letter, the computer checks whether there are more words in its list with the letter than without and then whittles the list down to the largest such subset. If there were more words with the letter than without, the computer congratulates the user and reveals the letter's location(s). If there were more words without the letter than with, the computer just laughs.

Put more simply, the computer constantly tries to dodge the user's guesses by changing the word it's (supposedly) thinking of. Pretty evil, huh? The funny thing is most humans wouldn't catch on to this scheme. They'd instead conclude they're pretty bad at Hangman. Mwah hah hah.

Suffice it to say that the challenge ahead if you is to implement…

Wait for it…

Evil Hangman.

☐ But what is the algorithm for evil? Well, let's consider what the computer needs to do anytime the user guesses a letter. Suppose that the user needs to guess a four-letter word and that there are only a few such words in the whole English language: `BEAR`, `BOAR`, `DEER`, `DUCK`, and `HARE`. And so the computer starts off with a list of five words. Next suppose that the user guesses `E`. A few of those words contain `E`, so the computer had best decide how to dodge this guess best. Let's partition those words into "equivalence classes" based on whether and where they contain `E`. It turns out there are four such classes in this case:

`----`, which contains `BOAR` and `DUCK`
`-E--`, which contains `BEAR`
`-EE-`, which contains `DEER`
`---E`, which contains `HARE`

Note that `HARE` is not in the same equivalence class as `BEAR`. Even though both have just one `E`, each's `E` is in a different location. If the computer is forced to admit that the word that it's "thinking of" contains an `E` (because it just so happens to have discarded all words that lack `E` in response to the user's past guesses), it will be forced to announce (and thus commit to) that letter's location.

So, back to our story: the user has guessed `E`. What should the computer now do? Well, it could certainly declare that the word it's "thinking of" does not contain `E`, the implication of which is that the list of five words becomes two (`BOAR` and `DUCK`). That does feel optimal: armed with two words, the computer might still have a chance to "change its mind" yet again later. Then again, `DUCK` seems pretty easy to guess, whereas most people might not even think of `HARE`. On the other hand, I don't remember the last time I used `BOAR` in a sentence. But let's keep it simple: you

may assume that the computer will always react to a user's guess by whittling its list down to the largest equivalence class, with pseudorandomness breaking any ties.

Realize, though, that the largest equivalence class might not always correspond to `----`. For instance, suppose that the user had guessed `R` instead of `E`. The equivalence classes would thus be as follows:

`----`, which contains `DUCK`
`---R`, which contains `BEAR`, `BOAR`, and `DEER`
`--R-`, which contains `HARE`

In this case, the computer should go ahead and admit that the word that it's "thinking of" contains `R` at its end, since that leaves three possible words (`BEAR`, `BOAR`, and `DEER`) and thus the maximum amount of maneuverability down the road in reaction to subsequent guesses. Of course, if the user has never heard of a `DUCK`, then `----` could very well be a superior strategy. But, again, lest you drive yourself nuts with overanalysis, you may assume that the largest equivalence class is optimal, even though it might not be in reality.

Indeed, that strategy might sometimes backfire, at least in a sense. Suppose that, in a new version of the story at hand, there are only three four-letter words in the English language: `BOAR`, `DEER`, and `HARE`. Suppose now that the user guesses `E`. In this case, our equivalence classes are:

`----`, which contains `BOAR`
`-EE-`, which contains `DEER`
`---E`, which contains `HARE`

Because each has one word, these classes are, of course, of the same size. But if we happen to select `-EE-` pseudorandomly, thereby whittling our list down to just `DEER`, we'll have revealed to the user two of the word's letters, whereas we could have revealed zero (had we selected `----` instead) or one (had we selected `---E` instead). But, again, that's okay. You are welcome, but not required, to implement a more sophisticated algorithm than that prescribed here; just make clear in some comments how yours happens to work.

As for how to divide a list of words into equivalence classes programmatically, well, we'll leave that as a challenge for you!

Required Reading.

☐ Okay, time to curl up with some reading. Read through the *iOS Human Interface Guidelines* so that you get off on the right foot with iOS apps:

`http://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/Introduction/Introduction.html`

☐ Next peruse the *iOS Application Programming Guide* to get a sense of what you can do on the platform:

`http://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html`

This guide goes into more detail than you need for this project, so don't fret if you start to feel overwhelmed. Consider it a teaser for what else is out there.

☐ Finally, read the *Memory Management Programming Guide* carefully:

`http://developer.apple.com/library/mac/documentation/cocoa/conceptual/MemoryMgmt/index.html`

That stuff's important, lest you start writing bad code!

Getting Started.

☐ Alright, here we go. Launch Xcode 4 and create a new project, as by clicking **Create a new Xcode project…** in the splash screen that displays upon launch (if you didn't disable) or by selecting **New > New Project…** from Xcode's **File** menu. When prompted to choose a template for your new project, select **Application** under **iOS** (not **Mac OS X**), select **Utility Application**, then click **Next**. When prompted for a **Product Name** and **Company Identifier**, input **Hangman** and **edu.harvard**, respectively; you should see that your **Bundle Identifier** will be **edu.harvard.Hangman**. Leave **Use Core Data** and **Include Unit Tests** unchecked. Then click **Next**. Choose a location for the project when prompted, then click **Create**. (Best not to save anything in `/Developer`, lest you upgrade Xcode in the future.)

A window named **Hangman – Hangman.xcodeproj** should then appear. Go ahead and expand each of the triangles at left (except for **UIKit.framework**, **Foundation.framework**, and **CoreGraphics.framework**), and you should see `HangmanAppDelegate.{h,m}`, `MainWindow.xib`, `MainViewController.{h,m}`, `MainView.xib`, `FlipsideViewController.{h,m}`, and `FlipsideView.xib` among their contents. We'll take a tour through each of those files, but first a quick demo. Ensure that **iPhoneApp | iPhone 4.3 Simulator** is selected in the drop-down to the right of the **Run** button in Xcode's top-left corner. Then click **Run** (or hit ⌘-R on your keyboard). If all goes well, the iOS Simulator should launch with this app. How fun!

Okay, it doesn't do all that much yet, but do click that ⓘ button in the app's bottom-right corner. Notice how the app flips around to its flipside. Well that's kind of neat. Click **Done** in the flipside's top-left corner, and you should be returned to the front side. It's okay if you'd like to do that again.

Anyhow, once ready for that tour of the code, quit iOS Simulator, return to Xcode, and open up `HangmanAppDelegate.h`. Notice how this file declares a class called `HangmanAppDelegate`, which implements the protocol called `UIApplicationDelegate`. That's good, because the "owner" of `MainWindow.xib` (*i.e.*, of your application) is a `UIApplication` object that just so happens to have a property called `delegate`.[†] Let's see how the two are connected. Go ahead and open `MainWindow.xib`. If, to the left of the "graph paper" that appears, you don't see any mention of **Placeholders** but do see a column of five icons (three cubes, one square, and a square in a circle), click the small arrow within a rectangle that's below that column of icons; you should then see **Placeholders** (as well as names next to those icons). Click **File's Owner** under **Placeholders**. Then, if not open already, open Xcode's **Utilities** by clicking the rightmost icon above **View** in Xcode's top-right corner. Below **View** are six smaller icons; click the third from the left to reveal Xcode's **Identity inspector**. Because **File's Owner** is still highlighted, you should see that its class is indeed UIApplication. Now ctrl- or right-click **File's Owner**, and you should see in a little black window that property called `delegate`, conveniently exposed as an "outlet" (*i.e.*, `IBOutlet`) for the sake of Xcode's interface builder. Notice that `delegate` is wired to **Hangman Application Delegate**, which is simply the (slightly more user-friendly) name for the instance of `HangmanAppDelegate` that will be instantiated when this nib is awakened. In fact, ctrl- or right-click **Hangman Application Delegate**, and you'll see that same wiring in reverse: the little black window's mention of "referencing outlet" implies that this object is wired to another object's outlet. Make sense? Oh good.

Now click on **Window** below **Hangman App Delegate**. Hm, not much to see. Okay, click on **Main View Controller** below **Window** instead. Aha! If you click on Xcode's **Identity Inspector** (remember how?), you should see that this object belongs to a class called `MainViewController`. If you then click on Xcode's **Attribute Inspector** (one icon over from that **Identity Inspector**), you'll see that this view controller's **NIB Name** is **MainView**, which implies that this view controller's UI (*i.e.*, view) is defined in `MainView.xib`. You know where we're headed next!

Open up `MainView.xib`. Ah, a slightly more interesting UI. (There's that ⓘ button.) Go ahead and click **File's Owner**. Can you guess what its class will be? Open Xcode's **Identity Inspector** to confirm or deny your prediction. Now click **View** under **Objects** (which itself is below **Placeholders**). Notice that this object's class is **UIView**, and it looks like it will fill an iPhone's whole screen. To confirm as much, open Xcode's **Attributes Inspector**, and change **Background** to something other than gray. See the view now? That's where your game's UI will soon be! If you ctrl- or right-click on **File's Owner**, you'll see how this `MainViewController` is wired to that view. If you ctrl- or right-click **View** under **Objects** (which itself is under **Placeholders**), you'll see the same in reverse.

[†] You can confirm as much at
`http://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/Reference/Reference.html`.

But not yet. Open up `MainViewController.h`, which declares the class that owns the nib you just closed. Notice how `MainViewController` descends from `UIViewController`. That's handy, because `UIViewController` comes with a whole lot of features:

`http://developer.apple.com/library/ios/documentation/uikit/reference/UIViewController_Class/Reference/Reference.html`

Notice, too, that `MainViewController` implements a protocol called `FlipsideViewControllerDelegate`. (It turns out that protocol is defined in `FlipsideViewController.h`, but more on that in a moment.) Notice that `MainViewController` has no instance variables and one instance method (`showInfo:`).

Now open up `MainViewController.m`. Hm, lots more in this file. Let's look at each method in turn. Defined first is `flipsideViewControllerDidFinish:`, which belongs to that protocol called `FlipsideViewController`, so more on that in a bit. Next up is `showInfo:`, whose declaration was in `MainViewController.h`. Notice how this method allocates a `FlipsideViewController`, thereafter initializing it with `FlipsideView.xib`. It then defines itself as the delegate for the flipside's controller, sets the transition from front side to flipside, and presents that flipside. As for the rest of the methods defined in that file, they're just placeholders for now that you might want to remove or enhance later.

Alright, almost done with the tour. Open up `FlipsideView.xib`. Ah, there's the flipside's UI. If you ctrl- or right-click **File's Owner**, you'll see how this nib's owner (a `FlipsideViewController`) is wired up to a view. You'll also see how that **Done** button is wired to a method (*i.e.*, `UIAction`) called `done:`. If you expand the triangle next to **View** (and in turn everything below), you'll see how the flipside's navigation bar and title relate to that button. In fact, if you ctrl- or right-click **Bar Button Item – Done**, you'll see that same wiring in reverse.

Okay, now open up `FlipsideViewController.h`. Not only does this file declare a class called `FlipsideViewController` (which descends from `UIViewController`), it declares a property called `delegate`, which must apparently point to an object (implied by `id`) that implements the protocol called `FlipsideViewControllerDelegate`. It also declares that instance method called `done:`. And, as promised, it declares that protocol, which apparently prescribes a method called `flipsideViewControllerDidFinish:`. The implication, of course, is that any class that implements `FlipsideViewControllerDelegate` (*e.g.*, `MainViewController`) should implement a method called `flipsideViewControllerDidFinish:`. Just as we've seen!

Finally, open up `FlipsideViewController.m`. In here are some placeholders, just like those we saw in `MainViewController.m`. But this file also implements `done:`, whose sole line of code apparently informs the flipside's delegate (*i.e.*, a `MainViewController` object) when the flipside is done flipping back.

Phew, time for a break.

☐ Welcome back. Feel free now to poke around the files in the **Supporting Files** group, but odds are you won't need to touch any of them for this project. But let's add one additional file to that group, a dictionary with 233,614 English words![‡] Download the property list (*i.e.*, XML file) at

`http://cdn.cs76.net/2011/summer/projects/ios/words.plist?download`

and then drag `words.plist` into into that **Supporting Files** group within Xcode. When prompted to choose options for adding the file, check **Copy items into destination group's folder (if needed)**, ensure that **Hangman** is checked to the right of **Add to targets**, and then click **Finish**. (Since `words.plist` isn't a folder, it doesn't matter what's selected next to **Folders** in that window.) You should now see `words.plist` among your app's **Supporting Files**. Click it, and you'll see a huge list of strings, a big ol' array of English words.[§]

Alright, that's it! It's time to delegate control of this project to you! (Get it?)

Specification.

☐ Your mission for this project is to implement Evil Hangman as a native iOS app. The overall design and aesthetics of this app are ultimately up to you, but we require that your app meet some requirements. All other details are left to your own creativity and interpretation.

Features.

- ☐ Immediately upon launch, gameplay must start (unless the app was simply backgrounded, in which case gameplay, if in progress prior to backgrounding, should resume).
- ☐ Your app's front side must display placeholders (*e.g.*, hyphens) for yet-unguessed letters that make clear the word's length.
- ☐ Your app's front side must inform the user (either numerically or graphically) how many incorrect guesses he or she can still make before losing.
- ☐ Your app's front side must somehow indicate to the user which letters he or she has (or, if you prefer, hasn't) guessed yet.
- ☐ The user must be able to input guesses via an on-screen keyboard.
- ☐ Your app must only accept as valid input single alphabetical characters (case-insensitively). Invalid input (*e.g.*, multiple characters, no characters, characters already inputted, punctuation, *etc.*) should be ignored (silently or with some sort of alert) but not penalized.
- ☐ Your app's front side must have a title (*e.g.*, **Hangman**) or logo as well as two buttons: one that flips the UI around to the app's flipside, the other of which starts a new game.

[‡] The words are taken from `/usr/share/dict/words` on Mac OS 10.6.7, with all forced to uppercase and duplicates removed.
[§] Even though Xcode presents the contents of `words.plist` with three columns (**Key**, **Type**, and **Value**) as though the file contained a dictionary, it indeed contains an array (implemented in XML with an `array` element, which you can see if you open `words.plist` with a text editor).

- ☐ If the user guesses every letter in some word before running out of chances, he or she should be somehow congratulated, and gameplay should end (*i.e.*, the game should ignore any subsequent keyboard input). If the user fails to guess every letter in some word before running out of chances, he or she should be somehow consoled, and gameplay should end. The front side's two buttons should continue to operate.
- ☐ On your app's flipside, a user must be able to configure two settings: the length of words to be guessed, and the maximum number of incorrect guesses allowed. The allowed range for the former must be [1, *n*], where *n* is the length of the longest word in `words.plist`; the allowed range of the latter must be [1, 26].
- ☐ When settings are changed, they should only take effect for new games, not one already in progress, if any.

Implementation Details.

- ☐ Your app's UI should be sized for an iPhone or iPod touch (*i.e.*, 320 × 480 points) with support for, at least, `UIInterfaceOrientationPortrait`. However, if you own an iPad and would prefer to optimize your app for it (*i.e.*, 768 × 1024 points), you may, so long as you inform your TF prior to this project's deadline.
- ☐ Your app must come with default values for the flipside's two settings; those defaults should be set in `NSUserDefaults` with `registerDefaults:`. Anytime the user changes those settings, the new values should be stored immediately in `NSUserDefaults` (so that changes are not lost if the application is terminated).
- ☐ You must implement each of the flipside's settings with a `UISlider`. Each slider should be accompanied by at least one `UILabel` that reports its current value (as an integer).
- ☐ You are welcome to implement your UI with Xcode's interface builder in `MainView.xib` and `FlipsideView.xib`, or you may implement your UI in code in `MainViewController.{h,m}` and `FlipsideViewController.{h,m}`.
- ☐ You must obtain a user's guesses via a `UITextField` (and the on-screen keyboard that accompanies it). For the sake of aesthetics, you are welcome, but not required, to keep that `UITextField` hidden (so long as the on-screen keyboard works). You are also welcome, but not required, to respond to user's keypresses instantly, without waiting for them to hit **return** or the like, in which case `textField:shouldChangeCharactersInRange:replacementString` in the `UITextFieldDelegate` protocol might be of some interest.
- ☐ Your app must not leak memory.[**]

How to Submit.

- ☐ *To be announced at* `https://www.cs76.net/` *prior to this project's deadline.*

[**] Do not assume that Xcode's static analysis or Instruments will find all; rely ultimately on your own eyes and mind.