(This document is written with the primary intention of describing the progress I've made so far to someone at RCOS who may take over this job in the future.)

My primary objective is to repeat the last callout that occurred when a user shakes their device. This involves two things:

1. Being able to trigger an event when the device is shaken
2. Repeating the last callout

Part 1 was pretty easy to implement. The program already has the relevant information to do something when the device is shaken. This most importantly involves the motionEnded function which is overridden (pretty standard from tutorials I've seen). On XCode, it's really easy to look for a particular keyword. On the left menu, which is commonly used for files, there is a tab which contains a search bar, where you can search for any keyword. Putting in motionEnded quickly led me to the appropriate section to put the code. I then put code to detect a motion shake and made a message get logged to the console whenever the device was shaken. Part 1 is completed - an event is triggered when the device is shaken.

Part 2 has been much trickier to implement, and, as of writing this, is currently not implemented. Implementing this part has required me to establish a thorough understanding of how sounds are processed in soundscape. While there are functions in Swift that can call tts audio, like .speak, Soundscape uses a far more complex approach to sounds, though it's this that allows it to broadcast more complex sounds, like 3D sounds. This 3D sound is significant - especially to someone who is blind. For example, when using headphones, using the app in Amos Eaton when Lally Hall is on the right will play the sound primarily on the right headphone. Moving the phone in real-time (for example, flipping the phone so Lally Hall is now on the left) primarily plays the sound on the left headphone.

The callout system also takes advantage of a queue, which ensures that only one sound is played at a time. All the text is put into a CalloutGroup, an object which is then passed to an AudioEngine at the last moment for rendering. The CalloutGroup has the potential to change while sounds are rendering, so by rendering the sound right before playing it, the most up to date information is presented to the user. Although many locations are presented, there appears to be a cap on how many locations are called out based on count and distance (count seems to be around 5, distance.. around a mile?) I don't know exactly where these numbers are stored.

The code also seems to utilize 'generators' as a part of the sound playback process, of which there are automatic and manual generators. Automatic generators generally queue up instead of interrupting and are updated based on things like user location (I'd guess that the sound moving to the other headphone based on how I adjusted my phone earlier was also a product of an automatic generator). Manual generators, represent user actions and tend to interrupt automatic generators. These generators are utilized in classes, which contain CalloutGroup objects.

My interpretation of the above request is to repeat the last callout even if current callouts are being called. For example, if callout1 was “Lally Hall around 100 feet”, callout2 was “Sage Lab around 200 feet”, and callout2 was speaking, shaking the device would interrupt callout2 and instead play callout1 (which played just before callout2), before playing callout2 from the beginning. Shaking while callout1 was playing would just restart callout1. If callout2 was the last callout, shaking should play callout2. Now note that callout1 is (I assume) based on an automatic generator. But it needs to interrupt callout2 before playing. In the AutomaticGenerator, there is canInterrupt variable, so before playing callout1, that should be set to true. This should allow callout1 to properly interrupt callout2.

How would I implement this? To start, CalloutGroup objects are generally located in classes that utilize generators, so by convention we should store callouts there. We could create a new variable that will store the information of the last CalloutGroup played that gets updated whenever a callout is finished playing. We then need to tell the audio generator to stop playing the current callout (if any) and play the last callout upon shaking the device.

While I haven’t been able to really implement something so far in this part, I do have a few ideas on things that might work. We could create a new variable that will store the information of the last CalloutGroup played. We then need to tell the audio generator that, when the device is shaken, to play the callout stored in that variable. The part that I’m struggling with is where to place that code and how to write that code.

Swift coding, despite being fairly high-level, is unlike traditional languages like C++, Java, or Python in its syntax and the things you can do with it, so stuff doesn’t stick out as easily. There are also lots of custom-defined objects which, most likely because of Swift and their complexity, doesn’t make it very clear to me what each line of code does.

However, one thing that has helped has been communicating with people in the Soundscape Slack. Here, RCOS students can communicate with other active developers. If Wes Turner is still at RCOS, ask him to add you and he should be able to honor that request. The expert on the sound side of Soundscape is Daniel Steinbrook, so if you work on this issue, send him a DM if you have any problems. I’ve been exchanging messages with him and he’s given me great advice on how the sound system works, much more information than me just poking around the code. If there’s one thing you should do after reading this, it should be joining the Soundscape Slack.

## Update 3/26/24

Since last week, I have gotten a better understanding of how some of the program code works. Last friday (3/22), I expanded on my attempt to store the callouts in a variable.

My attempts on Friday were pretty fruitless, perhaps trying to write a line of code here and there, but ultimately to no verdict. Today, I got a slightly different approach. I had remembered that

Daniel had told me to look at the storyboards and see how the code was implemented for the four buttons on the bottom of the main menu. I made some decent traction here. I watched a YouTube video (https://www.youtube.com/watch?v=oZGAicT2zbg) to get an idea of how storyboards interacted with code. I eventually discovered the @IBAction keyword, which linked up buttons with actual code. Right clicking on the buttons revealed the relevant function that they linked up to. From there, I used cmd + f to search the entire project (as explained above) and quickly found the file in which they were located. One line started with AppContext.process(ExplorationModeToggled(... and I right clicked on ExplorationModeToggled, which took me to the top of the ExplorationGenerator.swift file. This taught me that the buttons were pretty closely related to the ExplorationGenerator, and that this is where I should probably be focused.

As I looked through the class of ExplorationGenerator, I started to find more evidence of these buttons. Right after the declaration, I found a Mode enumeration, having cases of locate, aroundMe, aheadOfMe, and nearbyMarkers. As I searched through the file, I found answers to a question I was wondering earlier. A variable titled maxAheadOfMeCallouts is initialized to 5, confirming my suspicion that up to 5 callouts are listed. While I haven't looked into markers yet, there is a maxNearbyMarkerCallouts that is initialized to 4, so that seems to answer that question.

But after that, I quickly got confused again. Line 157 (as of writing this) says var callouts: [CalloutProtocol]. This made me quite confused because lots of documentation I was reading about queued audio (https://github.com/soundscape-community/soundscape/blob/main/docs/ios-client/overview.md#discrete-queued-audio) discussed Callout*Group*. Protocols, from what I understand, are a level above classes. They create a framework that is inherited by a class or an object (CalloutGroup, for example, is an object). CalloutProtocol is mentioned in CalloutGroup's class declaration, but I don't know if it's necessarily inherited by it. Looking through the code doesn't explicitly mention anything about the explicit *text* that is rendered. But ultimately, I'm most confused about how CalloutGroup and CalloutProtocol are related and when to use which. I've sent Daniel Steinbrook a message concerning some of my issues and I await his response.

## Update 4/12/24

I've made some more progress on understanding the code and have even started making some slight contributions. The biggest thing I discovered was where the sounds get passed to the Audio Engine. This is done in CalloutStateMachine.swift - the particular line is

```
strongSelf.audioEngine.play(sounds) { (success) in
                callnoutGroup.delegate?.calloutFinished(callout, completed: success)
```

I don't entirely understand the whole line, but it calls .play on the audioEngine, which is a big sign. It also calls sounds, which is an array of particular sound objects that are grouped into a callout.

Sounds appear to be short phrases - things like “Nearest road” and “Star One Credit Union”,  (a weird example, yes, but the simulator oddly kept calling this out). These are all strung together to form a callout. The compilation of such objects is done in POICallout in the sounds function. A sounds array is created, which is an array of sounds. Individual sounds are then appended and, when every sound is appended, is passed back to the caller of sounds (which might be CalloutStateMachine). I have tried appending sounds, like..

```
sounds.append(TTSSound("Hello world."))
```

Appending this sound causes “Hello world” to be added to the callout, which I’m able to hear. This isn’t the first time I’ve tried running this line - I tried this in the past but I put it in a spot where the sound wouldn’t be added to an array that would get properly returned, meaning the sound never gets played.

Back to the code I’ve been trying to write - I’ve tried to obtain the previous sound, but have struggled to do so. The big issue is where to declare this “previous sound” variable. Declaring it in the same function as where the sound is won’t work. I’ve attempted to expand the play sound code so it utilizes the previous sound variable. Declaring this variable after the play sound code results in scope issues. Declaring it before resets the previousSound variable, therefore causing the sound to not be preserved. This means it needs to be declared outside of this function. I tried declaring it in the class declaration where several other variables are declared, but this results in problems.

I sent a message to Daniel explaining the previous sound variable I was trying to create and how to declare variables in CalloutStateMachine. He sent me a message back saying that the variable I’m describing already exists. I looked at the variable and immediately got confused as its type is [POI] rather than Sounds. So instead of having an array of sounds, there seems to be an array of POIs. A POI object contains several useful things, like the name of an object, a localized name, an address, a street, longitude, latitude, and even functions stating the closest location and its distance (I don’t entirely know the purpose for the functions). Part of the reasoning may be that the callout might need to be updated (i.e. if we’re 100 feet from a location and we move 50 feet away, the callout should now say that we’re 150 feet away). I’ve asked Daniel on how to utilize this POI object within the context of the CalloutStateMachine (which utilizes sounds), and I await his response.