**ARchitecture App Developer Documentation v1.0**

Authors: An Nguyen, Lam Pham

**Table of Contents**

[**I. Introduction:**](#_gjdgxs)

[**II. Basic System Requirements:**](#_30j0zll)

[**2.1 Hardware Requirement**](#_ru1veukqaqnn)

[**2.2 Software Requirement**](#_62l8qeusrtdz)

[**III. Getting Started:**](#_1fob9te)

[**3.1 Developer’s Prerequisite**](#_7wxc6ah34k85)

[**3.2 App Installing**](#_3znysh7)

[**IV. Code:**](#_rnglowi4t2ah)

[**4.1 Project Structure**](#_t2lj3srdvas7)

[**4.2 Classes and Functions**](#_98fyels2ur38)

[**4.2.a *Main.storyboard***](#_uwfwyvr2lesb)

[**4.2.b *ViewController.swift***](#_wnke453gz07e)

[**4.2.c *CustomCellView.swift***](#_gsekyo94kc65)

[**4.2.d *ModelSelectionController.swift***](#_uo1uy1qa2d3q)

[**V. Concerns**](#_bxuvwssromul)

[**VI. Final Words**](#_gfey6zqik9hh)

[**VII. Reference**](#_12t1rmvwmhbd)

# **I. Introduction:**

ARchitecture utilizes augmented reality technology to display a 3D computer-generated model into reality through the mobile device’s camera. ARchitecture utilized one of the most popular augmented development platform known as ARKit which is developed and maintained by Apple. With ARKit, ARchitecture is capable of detecting a horizontal plane, placing 3D architectural models into reality, interacting with the models that have been placed, and detecting physical images.

# **II. Basic System Requirements:**

# 2.1 Hardware Requirement:

Since ARKit is a development platform exclusively for iOS mobile devices, the only hardware requirement is building the project onto iOS devices that are compatible with ARKit. Here is a list of recommended iOS devices:

* *iPhone Devices*: iPhone 8, iPhone 8 Plus, iPhone X, iPhone 7, Iphone 7 Plus, iPhone 6s, Iphone 6s Plus, iPhone SE, iPhone 6.
* *iPad Devices:* iPad Pro (12.9-inch), iPad Pro (9.7-inch), iPad Pro 12.9-inch (2nd generation), iPad (5th generation).

*Note:* The ideal iOS device for this project is the latest iPhone model which is iPhone X because it offers better and more stable tracking capability.

# 2.2 Software Requirement:

*ARchitecture* requires Xcode to be built onto a compatible device. Here are the recommended version of iOS devices’ firmware and Xcode’s firmware version:

* iOS (Apple mobile devices operating system) must be 11.3 + version.
* The latest version of Xcode (Xcode 9.4.1) is recommended because it comes with the latest version of ARKit (1.5). Note: Xcode is exclusively available for macOS.

# **III. Getting Started:**

Getting started with our app is simple. An instruction on how to get our repository and install the app is provided in section 3.1 below.

# 3.1 Developer’s Prerequisite:

*ARchitecture* is an iOS mobile app built using Xcode and Swift. It is recommended that developers should have a basic understanding of Swift and how to use Xcode. Future developers should also understand how ARKit works and how to use ARKit within Xcode. Fortunately, Apple provides a really good documentation that can be found here: <https://developer.apple.com/documentation/arkit>.

# 3.2 App Installing:

Step 1: Project’s repository can be cloned or downloaded as a ZIP file via

<https://github.com/nguyenAn1201/ARchitecture>

Step 2: Edit your *provisioning profile* that corresponds with your *Apple Developer* account.

Step 3: Start building the app through Xcode.

# IV. Code:

# 4.1 Project Structure:

In order to assist future developers who might want to develops ARchitecture, a structure overview of the project’s files is provided below.

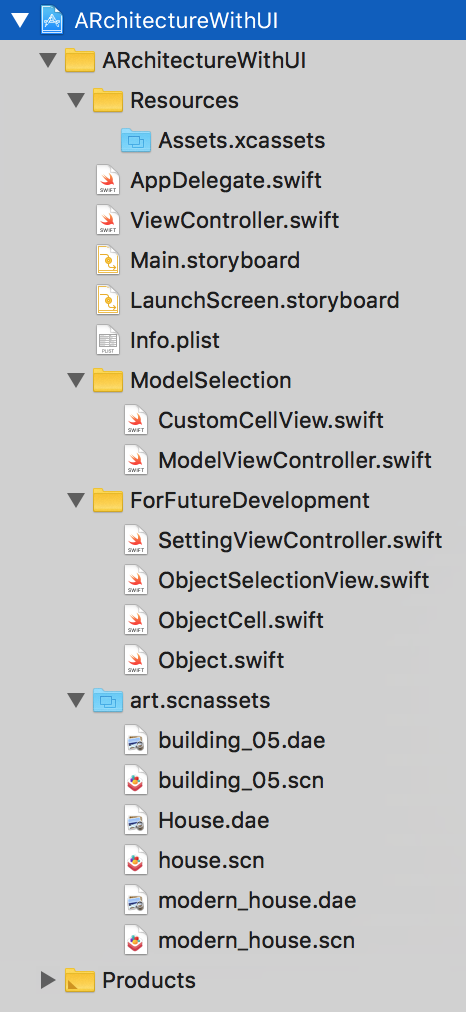


Figure 1.1: Overview of *ARchitecture’s* structure

All the classes, code, images, and models will reside inside *ARchitectureWithUI* folder. Sub-folders under Master folder provide components that are essential to *ARchitecture.* Here is a list of in-depth overview of every folders and their classes:

* *ARchitectureWithUI*: This folder contains everything that is required to successfully build the app. It contains ‘ViewController.swift’ and *AppDelegate.swift*. ‘ViewController.swift’ is one of the most important component because it contains all the functions that create and manage an augmented reality session. It also contains functions to add 3D models into reality, interact with the models, and it regulates actions based on user’s input. ‘Main.storyboard’ is also really crucial because it manages the UI designs and elements of *ARchitecture.*
* *Resources*: This folder contains a sub-folder *Assets.xcassets* where all the UI elements, model images, and image used for image detection. *Figure 1.2* below shows what is contained in *Assets.xcassets*.

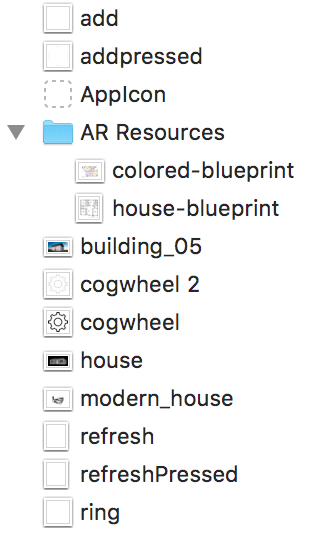


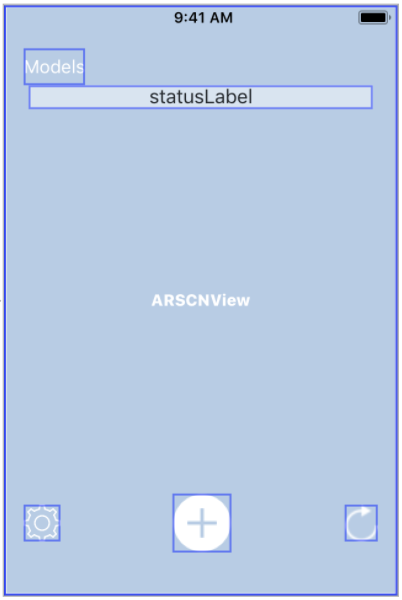
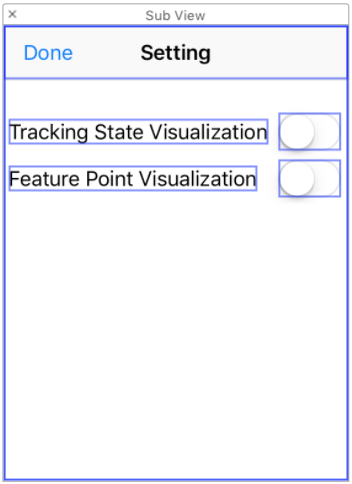
Figure 1.2: Overview of *Assets.xcassets* folder

* *ModelSelection*: All four classes in this folder create and update a custom table view that would display rows of 3D model with its image and its name.
* *ForFutureDeveloper*: This folder contains classes and functions that were being developed but were not considered to be used for the complete build. There are plans to use these classes and its functionalities in the future. Therefore, future developers can ignore this folder.
* *art.scnassets*: as *Figure 1.1* shown above, this folder contains all the 3D models that the app will try to display into reality. Most model have file format as *.dae* which is the best model format for ARKit to transform into a *.scn* file format. Methods to add new model will be explained in [section] below.
* ‘Product’: is a folder automatically generated by Xcode. It does not contain any components created by the original developers.

# 4[.](#_3znysh7)2 Classes and Functions:

With a general overview of all the folders and do they contain in *Section 4.1,* *Section 4.2* will go more in-depth with the classes and their functions that are crucial to the app. There are three important classes/files which are *ViewController.swift*, *ModelViewController.swift*, and *Main.storyboard*. Here are the breakdowns:

# 4.2.a Main.Storyboard

Figure 2.1: Three main views of *ARchitecture.* From left to right: *View Controller Scene, Model Selection Scene, and Sub View* 

When *ARchitecture* launches, *View Controller Scene* will be shown first. When user tap on the *Models* button on top left, the app will display *Model Selection Scene.* Lastly, the *Sub View* will be displayed when user taps on the *Setting* icon on bottom left.

The UI buttons and elements in *View Controller Scene and Sub View* are handled by *ViewController.swift*

# 4.2.b ViewController.swift

‘ViewController.swift’ controls many things, from managing user’s interaction to generating augmented reality functionalities.



Figure 2.2: Codes at the beginning of *ViewController.swift*

*Viewcontroller.swift* imports *UIKit* for configuring the UI elements of the app, *SceneKit* to handle the scene that would show augmented reality, and *ARKit* to handle all functionalities such as plane detection, model placement, and image recognition.

First two outlets *ARSCNView, sceneView* are variables that are attached to the app scene and *statusLabel* is the label that would display tracking states of the current AR session.

There are five important variables:

1. *textValue:* a type String variable that is initialized with “house”. This variable will be changed through *modelSelectionDelegate*. More info in
2. *modelNode:* an array of type *SCNNode* which is the type that contains information about the model in form of a node.
3. *currentNode:* variable of type *SCNNode* that holds the currently selected node. This will be set to the node of the model selected by the user.
4. *currentAngleY:* variable of type Float initialized to 0.0 that will be changed by *rotate* function.
5. *imageHighlightAction:* variable of type *SCNAction.* It is the animation that will be called when the app detected an image from *“AR Resources”* to indicate a successful image recognition.



Figure 2.2: Screenshot of code that set up an AR scene

Function *viewDidLoad()* is used to set up the view’s delegate create a scene that would later run the an AR session and it also configes the UI of *statusLabel* seen in Figure 2.1. Note that *statusLabel* is initially hidden until user turns them in the ‘Setting’ popup. Function *viewWillAppear(\_ animated: Bool)* creates a configuration for the AR scene on line 64 and it set the configuration to enable horizontal plane detection as seen on line 65. The code in line 68 enables image recognition which the app will be looking at “AR Resources” folder. “AR Resources” is a sub-folder of ‘Assets.xcassets’ containing images that the app uses to detect. The code on line 70 run the configuration that is previously created.

Code from line 73 to line 78 set up the app for user’s interaction with the app screen. Variable *rotateGestureRecognizer* will call function *rotate* to rotate an anchored model when a user does a rotating gesture on the app screen. Variable *scalingGestureRecognizer* will call *scale* function to scale an anchored models when user does a pinch gesture. Lastly, *tapGesture* will call function *tap* that would select an anchored model that the user taps on. All three variable are initialized with *sceneView.addGestureRecognizer().*

*ViewWillDisappear(\_ animated: Bool)* will pause the current scene when user switch away from the app.

*didReceiveMemoryWarning()* is called when the amount of memory is low.

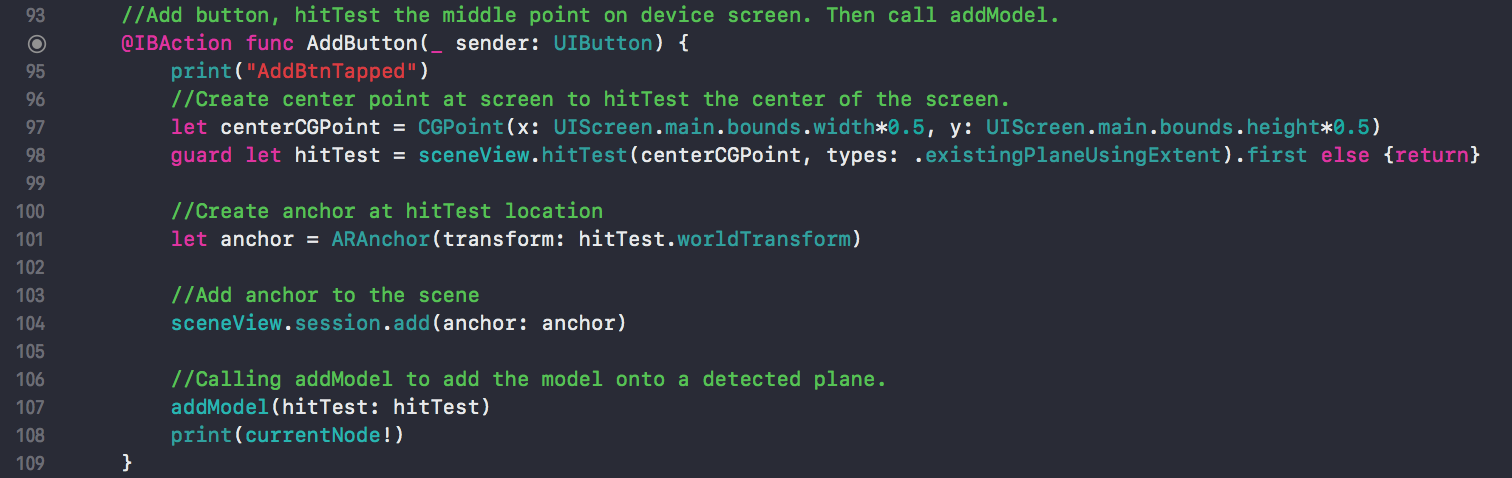
Figure 2.3: Screenshot of *AddButton* function

Figure 2.4: Screenshot of *addModel* function

*AddButton* is called when user tap on the Add button . This function will perform hit-testing at the center of the screen. Successful hit-test will add an anchor with the code on line 104. Now with the anchor created, *addModel* function is called to get the corresponding model which is define by *textValue.* Variable *textValue* contains the name of the model such as “house” which will allow the codes on line 248 and 249 to get and create a node of the model “house”. The codes from lines 252 to 254 will rotate the model to face the user before it will be anchored. On line 257, the code set the position of the model based on the hit-test result. Then the current node will be appended to the *modelNode* array and added to root node. This is where the model is placed into reality.

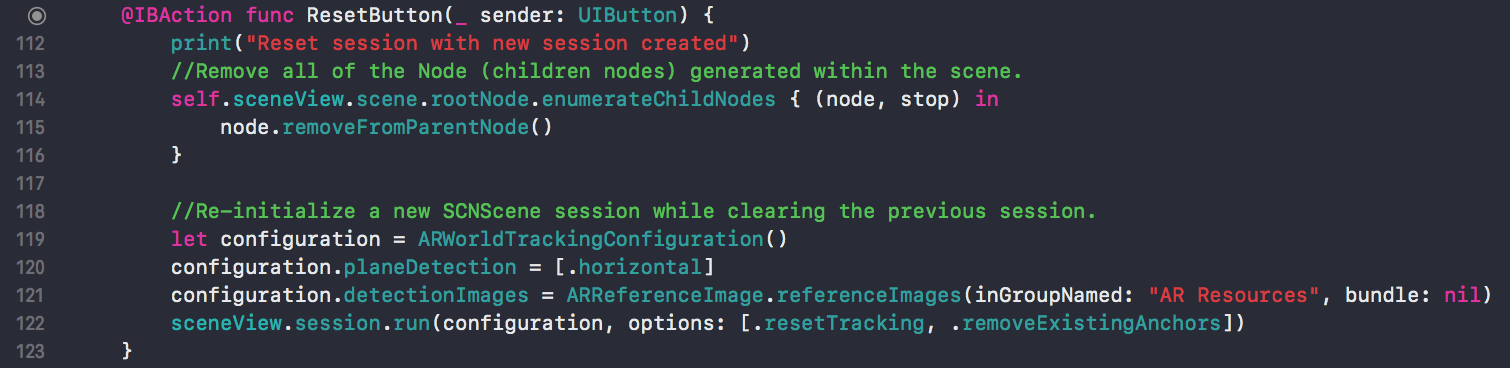
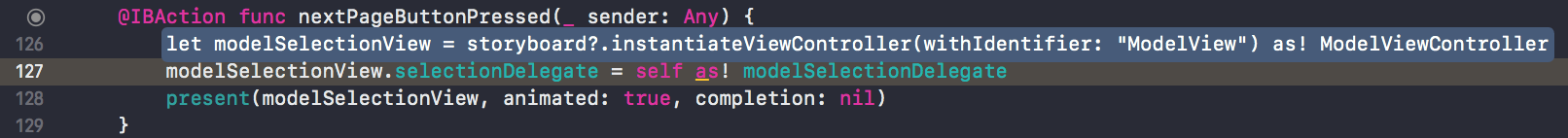


Figure 2.5: Screenshot of *ResetButton* function

*ResetButton* is called when user tap on . This function contains code that will remove all nodes in the scene. Then it will initialize a new scene while setting the same configurations similar to the codes shown in figure 2.2.



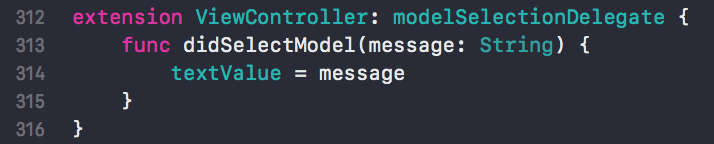


Figure 2.6: Screenshot of *nextPageButtonPressed* function and extension for *modelSelectionDelegate*

This function is called when user tap on ‘Models’ button in *View Controller Scene* that will display the *Model Selection Scene.* Line 126 will set *modelSelectionView* as the variable linked to the *Model Selection Scene* with “ModelView” identifier . Line 127 will set up connection between *View Controller Scene* and *Model Selection Scene* that allows both scenes to send and receive data. Line 128 will display *Model Selection View.* An extension is needed for the *modelSlectionDelegate* to pass data back to *ViewController.swift* from *ModelSelectionController.swift.* This is how user can select different models. Once the user selected a model from the *ModelSelectionView, ModelViewController* will send the name of the model as a string back to *ViewController* and *textValue* is set to that string. Remember from *AddModel* function, *textValue* is the name of the model used by *AddModel* function to get the correct model from *art.scnassets.*

**

Figure 2.7: Screenshot of functions that handle the ‘Setting’ popup

First, we created an UIView as *subView.* Function *SettingButton* will display *subView* through the code on line 136. Line 137 will center the *subView* in the center of *ViewController. subViewDone* is called when “Done” button in *Sub View* is pressed.

*trackingStateSwitch and featurePointSwitch* handles when to hide tracking states and when to show feature points.

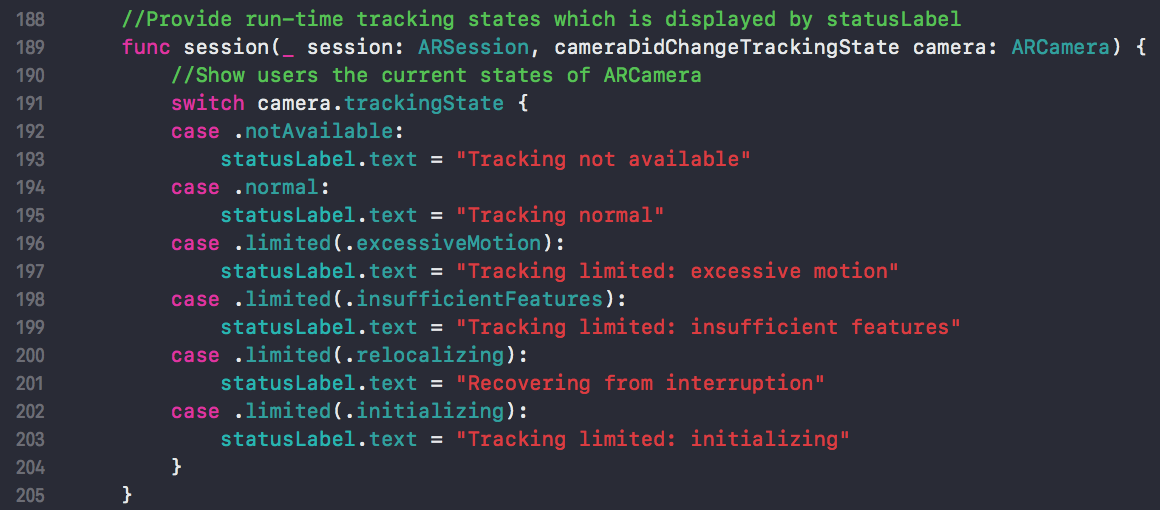


Figure 2.8: Screenshot of function that handle tracking states

This function is called every time the tracking state of the camera changes. A switch was used to replace *statusLabel* with the corresponding string that reflect the tracking state of the camera.

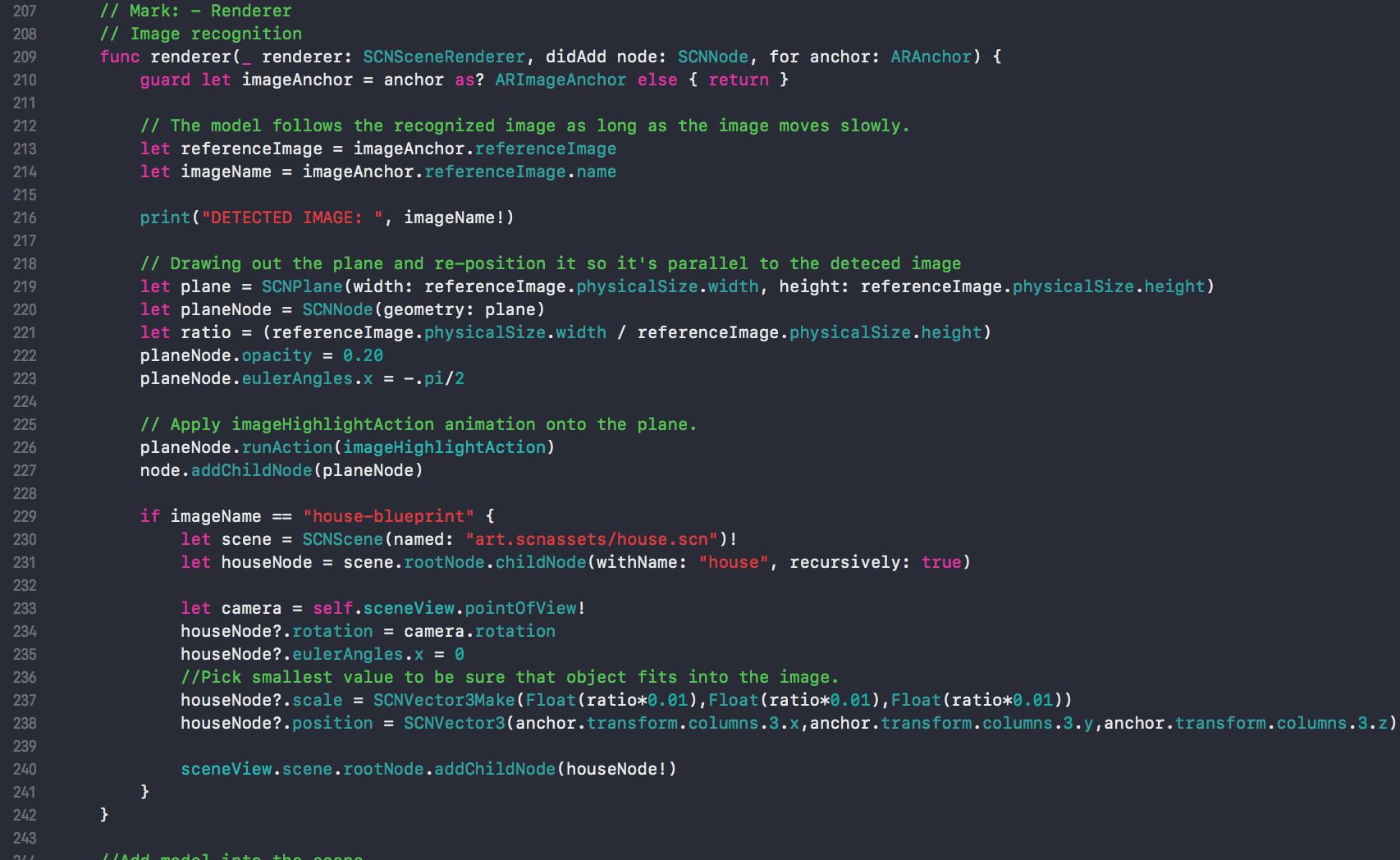


Figure 2.9: Screenshot of image recognition function

This function will initialize an anchor as *imageAnchor* where the model will be placed once image is detected. *referenceImage* will contain the detected image and *imageName* will contain the name of the image a string. Line from 219 to 223 will create a plane that has height and width equal to the detected image. Line 226 adds an animation indicating that the image is recognized. Then line 227 will add the image node onto the recently created plane. Code from line 229 to 240 will run when the correct image is recognized which will set the position of the model to face the user and scale that model so that it fits inside the image then it will display the model into reality by adding the model node to root node on line 240.

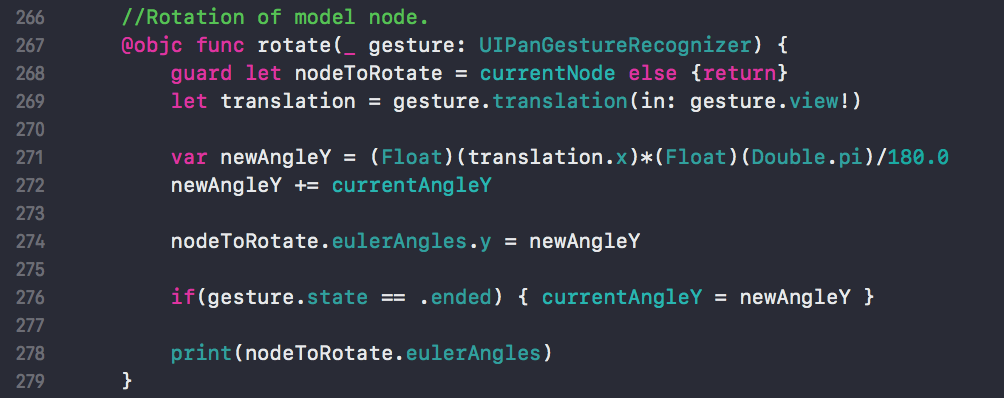


Figure 2.10: Screenshot of *rotate* function

Initially set the current node as the node to be rotated. Then create a variable to handle the amount translation based on user’s gesture. The variable *newAngleY* will calculate the rotation angle based on the *translation* variable. *newAngleY* then will be the sum between itself and the *currentAngleY* set at the beginning of the class. The code on line 276 will set *currentAngleY* with the final *newAngeY* once user stops the rotating gesture.

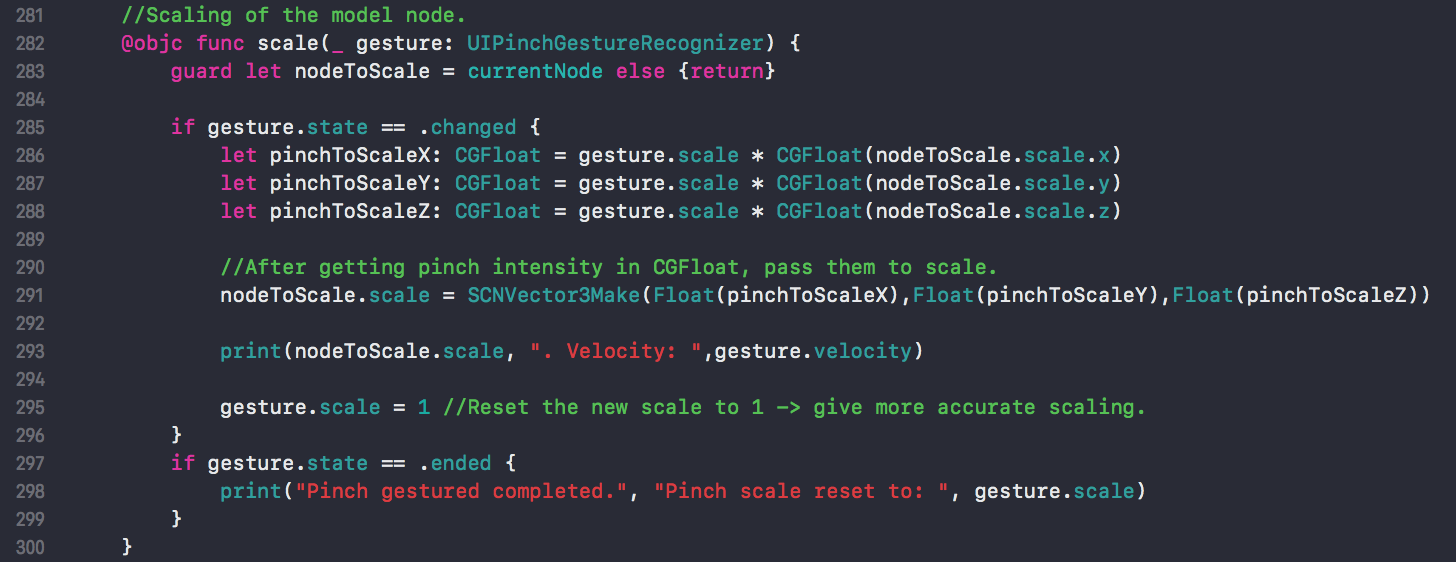


Figure 2.11: Screenshot of *scale* function

Initially set the current node as the node to be scaled. Code from line 285 to 288 will get pinch intensity from the gesture of the user. Multiply it with the current scale of the model then passes them to *pinchToScaleX, pinchToScaleY, pinchToScaleZ* variables. Line 291 will begin to scale the model using SCNVector3Make to set the scales using three variables above. Finally, reset the gesture.scale back to 1 so that when user pinch again, the function can still scale the model accordingly. There are two *print()* function used to output the gesture velocity to terminal.

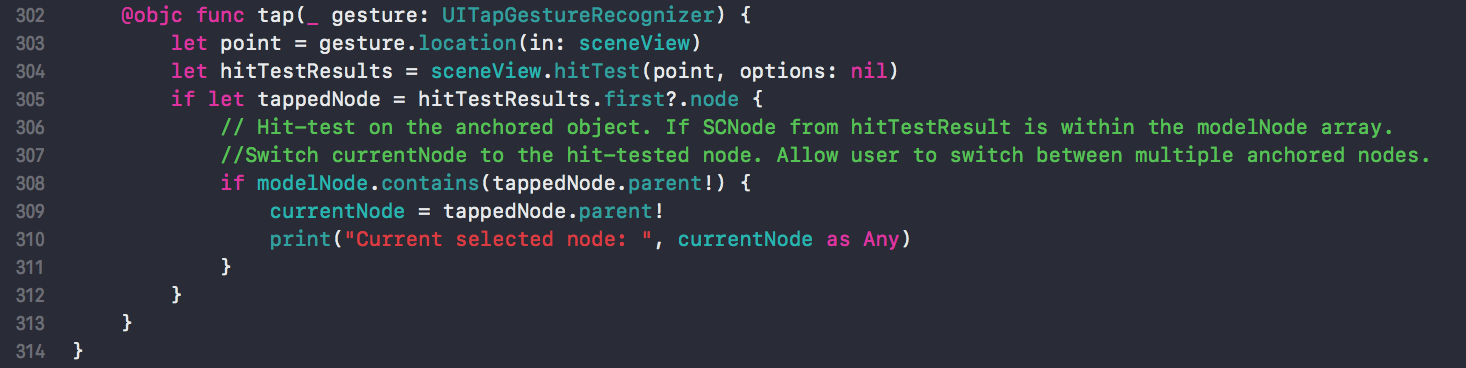


Figure 2.12: Screenshot of *tap* function

This function tries to select different anchored models and is called when user tap on the anchored model. First, it will get the user’s tap location and perform a hit-test on that tap location as you can see on line 303 and 304. When the hit-test result successfully return one of the anchored node, compare whether or not *modelNode* contains that node. Since *modelNode* is an array of all the anchored nodes and if it contains the tapped node then set *currentNode* to the tapped node. Note that *currentNode* has to be set to the parent of the tapped node since the tapped node return one of the child node.

# 4.2.c CustomCellView.swift

This class purpose is to configure how the table view should look like. All codes here set up the constraint of the table view and its rows.

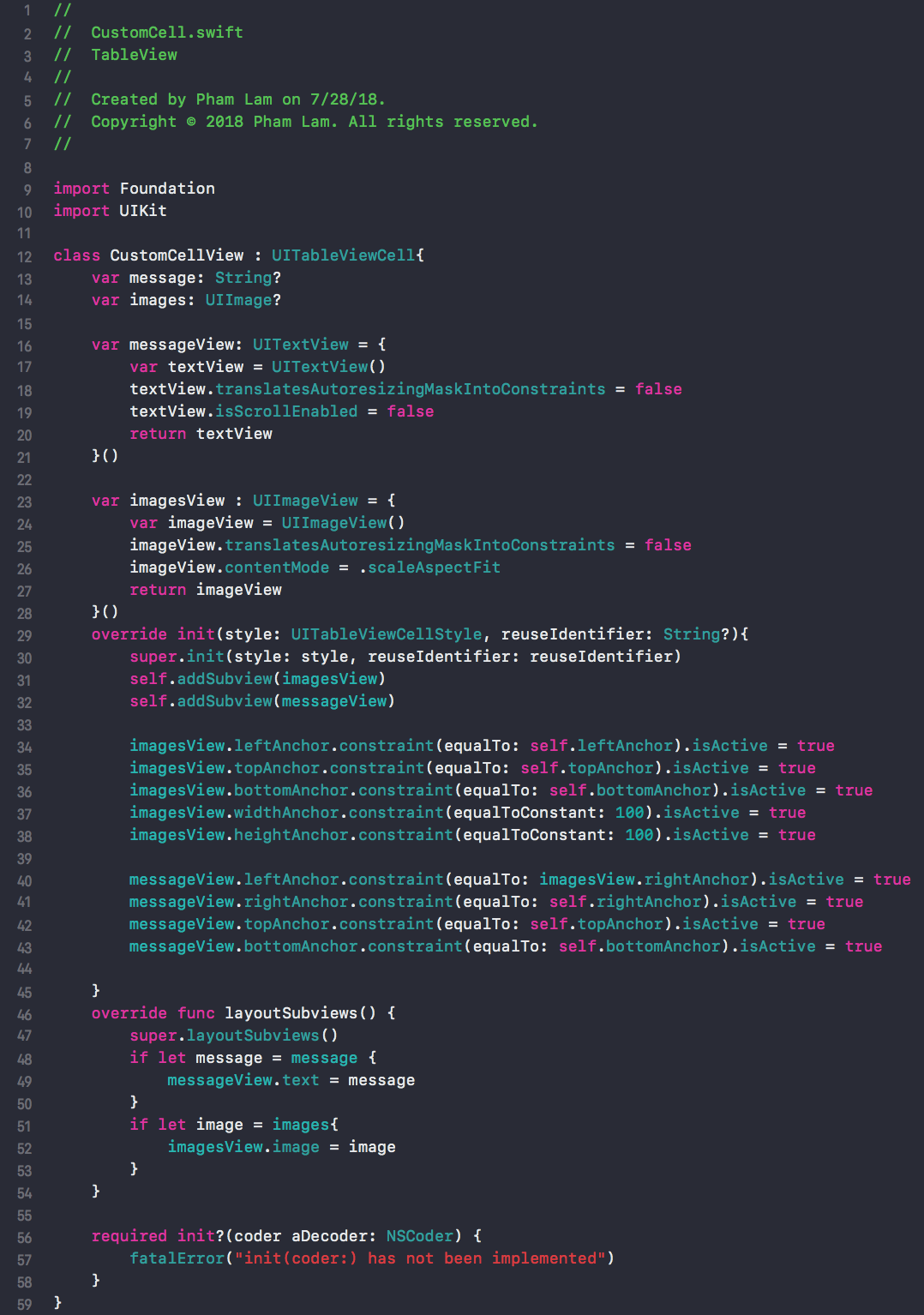


Figure 3.1: Screenshot of *CustomCellView*

# 4.2.d ModelSelectionController.swift

This class is crucial in handling the *Model Selection Scene* as it generate rows of model’s image and string name. This will allow user to select different models to be placed in the AR scene. Here is a part of the code:

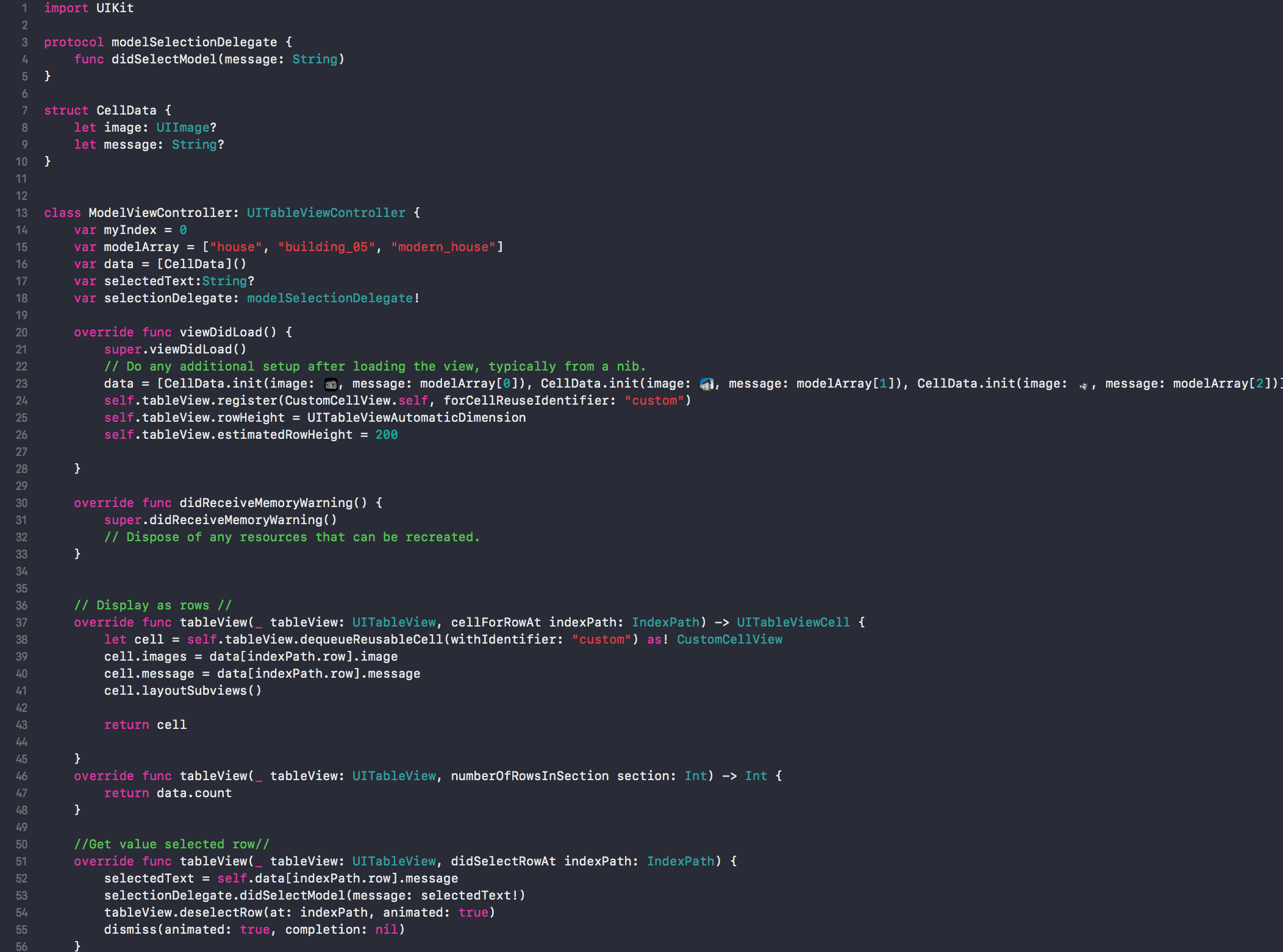


Figure 4.1: Screenshot of *ModelViewController.swift*

In order to pass data back to *ViewController,* a protocol is created as shown in code from line 3 to line 5. Line 14 to line 19 create variables that will be used in *viewDidLoad()* to generate the table view. *myIndex* is for keeping track of the current array index and is initialized to 0. *modelArray* will be an array of type String that will contain all the name of the models. *data* variable will be an array of type *CellData* that we defined on line 7 which will contains all the model image and model string name. *selectedText* is a variable of type String that would holds the string name of the model after the user select a model from the table view. Lastly, we have *selectionDelegate* that will be used to pass data back to *ViewController.*

*viewDidLoad()* function will populate *data* array with the model image (from Assets folder) and model string. Then it will create the table view and set the table view’s row height.

The code from line 26 to line 45 will populate the table rows with elements from *data* array. The code between line 46 and line 48 will return the number of row.

Finally, the last function will set *selectedText* to the string name of the model selected by user. Then it calls *didSelectModel(message: selectedText)* to pass the *selectText* back to *ViewController* where the *ViewController’s* extension will handle. Then the *Model Selection Scene* will be dismissed.

# V. Concerns

Currently, *ARchitecture* has a bug that when user selects ‘Models’ button to switch to *Model Selection Scene* view and selects a model that they want to display next. After selecting one of the row from table view in *Model Selection Scene* view, the app will return to *View Controller Scene* view again and this caused the previously anchored model to move from its original anchored position. We suspect that the present and dismiss between two views causes this and we are still trying to look for a resolution.

*ARchitecture* at its current state does not have the best UI design and we are trying to improve our UI layouts and design. We are planning to redesign our *Sub View* popup in the future.

*ARchitecture* is incapable of handling occlusion which is one of the most common problem with developing augmented reality application. In the future, we will try to resolve this occlusion problem. For more information about occlusion: <https://xinreality.com/wiki/Occlusion>.

# VI. Final Words

*ARchitecture* has accomplished its first stage of development which was to detect a horizontal plane, place a 3D model into reality, and recognize an image. As augmented reality technology is gaining popularity when Apple released its ARKit platform and Google released its ARcore development platform, *ARchitecture* was a successful attempt to explore and experience with developing augmented reality application for mobile device.

# VII. References

Documentation

* <https://developer.apple.com/xcode/>
* <https://developer.apple.com/documentation/swift>
* <https://developer.apple.com/documentation/arkit>
* <https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/>

Videos

* <https://developer.apple.com/videos/play/wwdc2017/602/>

3D Models

* <https://free3d.com/3d-model/house-11303.html>
* <https://free3d.com/3d-model/buildinghouse-05-3979.html>
* <https://free3d.com/3d-model/casa-moderna-white-rock-version-completa-4767.html>