AR Home Builder - Developer Guide

Introduction

AR Home Builder (project name ARHomeDesigner) is an app that runs on iOS devices 18.0 and newer. This app heavily relies on Apple's AR Kit and Reality Kit libraries to function, including core Swift UI/UI Kit functionality, providing gesture recognition and event handling when specific views change data in various parts of the app. This guide will give the reader a brief overview of some of the main techniques and algorithms used in this app and how views and published variables are connected throughout the app. This guide assumes the reader is already familiar with how Swift and Xcode function, thus, this guide will primarily focus on how the AR libraries function and how the app is structured overall.

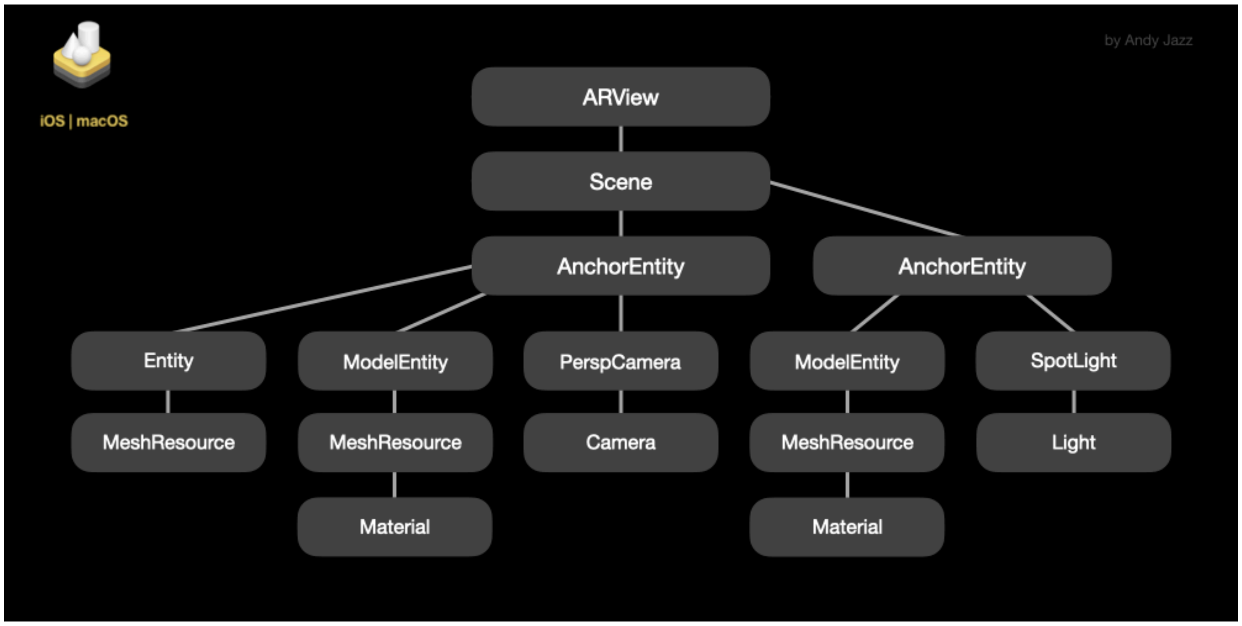
Table of Contents

Setting Up

For this project, you will need a Mac device with Xcode installed with the proper SDKs installed to develop iOS apps. You will also need a physical iOS device to test the program on, since AR Kit requires a physical camera and, as of the time of writing, cannot be tested using Xcode's iOS simulator. The project files can be found on compsci04. From there, you can find the download link on this project's website. If you need assistance with how to set up the app in Xcode to run on a physical iOS device, you can refer to the guide in the user manual in the "Installing The App" section on page 2.

ARView - How Data is Structured

At the app's core lies the central class that handles AR functionality: ARView. This class can be found within the Reality Kit library, which it's main responsibility is to render the AR display that contains virtual 3D graphics. This display is contained within a scene, which is made up of several anchor entities that tether the rendered entities, such as 3D model entities or light entities, to the scene. The flow chart below (credit to Andy Jazz) gives an example of what this looks like. Each component in the nodes of the chart needs to be linked together so that ARView can properly render entities in the correct location with the proper child entities (i.e. things that make up each entity/element in the AR scene)



Core Functionality

This section briefly goes over how key parts of this app work, mainly the parts that are essentially required for this app to function, and why. These are some of the key concepts that I, the one who designed/devolped this project, believe to be necessary to keep going forward.

World Tracking

A key part of how placing and managing the location of virtual object in a real world environment is done through world tracking. There are several types of pre-built configurations in AR Kit that each are aimed at general common use cases for AR, such as body and face tracking. World tracking sets up the AR View to track the device's movement with six degrees of freedom (up-down, left-right, forward-back, yaw, pitch, and roll). This is done using motion sensor data from the camara and through the use of a gyroscope.

Another key element of world tracking is that it can detect planes, images, and 3D objects in the environment according to the developer's specification. These kinds of detection can be enabled when first configuring the AR View. The main one of discussion here is plane detection.

Plane Detection

Plane detection is an optional feature that comes as part of the ARWorldTracking configuration. It enables AR View to scan the environment for flat surfaces, and it will place a special type of plane anchor on that surface. Each plane anchor has additional properties compared to a normal anchor entity. Plane anchors keep track of the information regarding the plane itself, i.e. how large the plane is and where the center of the plane lies. The documentation in the source code mentions this, but to explain it in more detail, this information is helpful because it allows for automatic sizing of shapes so that the app can resize the shape for the user and move it to the correct position. This information can be accessed via the planeExtent var, which is done after obtaining the plane anchor from a raycast.

Raycasting

The ability to tap on surfaces displayed onscreen to place objects is largely thanks to raycasting. Raycasting is a technique that converts a 2D coordinate into a 3D coordinate through a mathematical ray, whose direction and origin point in this case are determined by where the device is currently located in physical space. The direction is also influenced by where the user taps on the screen. This ray will shoot outwards until it intersects with another entity or specified geometry. AR Kit provides a raycast function that takes care of the ray calculations for the developer. In the current state, raycasting is used to find detected planes via a tap gesture recognizer:

```
arView.addGestureRecognizer(UITapGestureRecognizer(target:
    self, action: #selector(handleTap(recognizer:))))
```

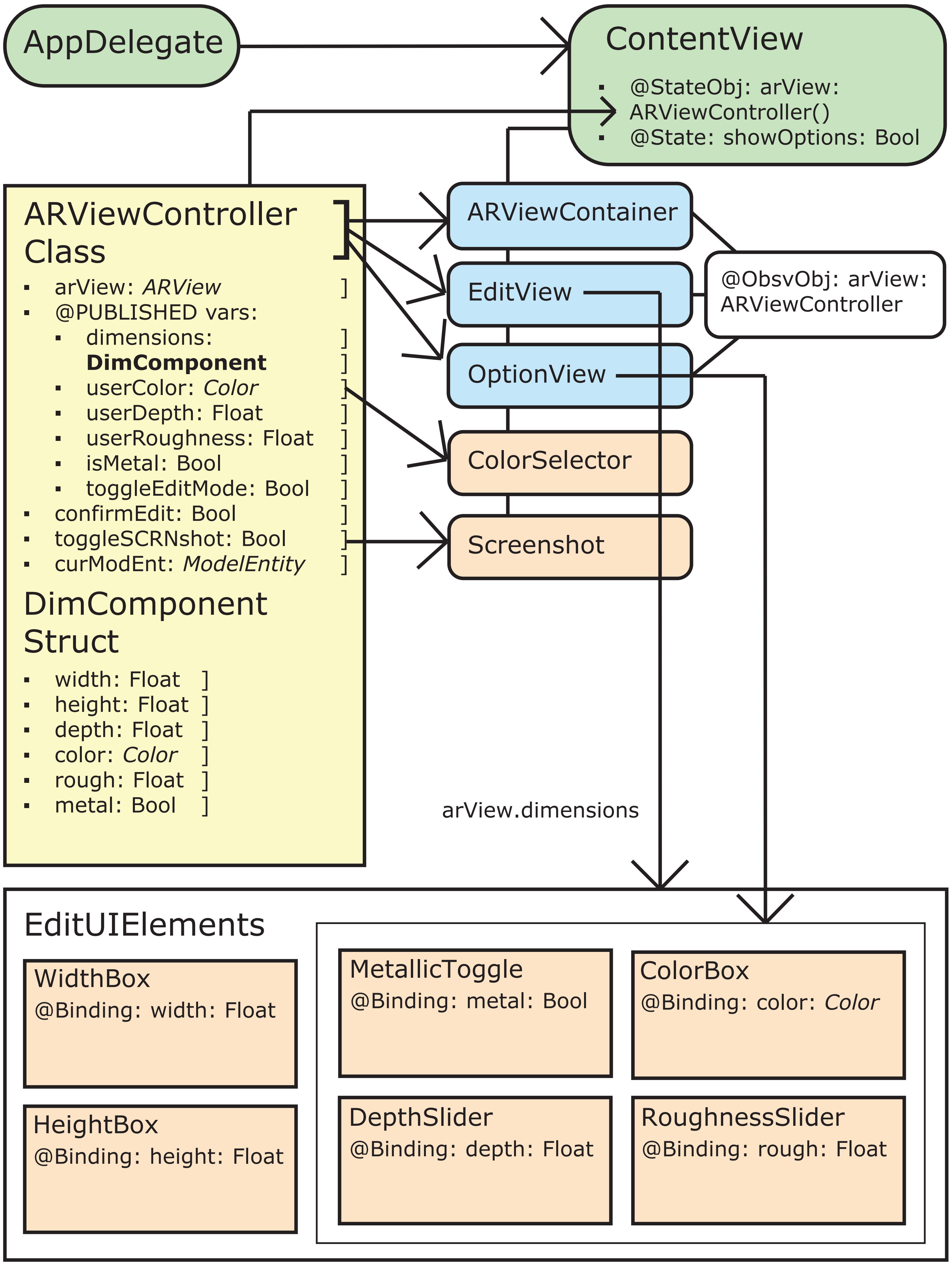
When the ARViewController detects a tap from the gesture recognizer, a handler function is called that takes the 2D coordinate of where the user tapped on the screen. From there, the function checks for a pre-existing model entity first by checking if the 2D point makes a collision with the model entity before performing a raycast on the location. This is due to each model entity being given a collision component so that they can be interacted with by the user.

```
@objc
private func handleTap(recognizer: UITapGestureRecognizer) {
    // Touch Location
    let tapLocation = recognizer.location(in: arView)
```

Once it finds a plane that intersects with the ray, the plane anchor can then be used to create the dimensions of the mesh and will serve as a blueprint for the newly created anchor to tether the model entity to the scene.

Overall App Structure

The next page shows a data flow diagram that illustrates how variables are connected throughout the app.


AppDelegate
ContentView
@StateObj: arView: ARViewController()
@State: showOptions: Bool
ARViewController Class
arView: ARView
@PUBLISHED vars:
dimensions: DimComponent
userColor: Color
userDepth: Float
userRoughness: Float
isMetal: Bool
toggleEditMode: Bool
confirmEdit: Bool
toggleSCRNshot: Bool
curModEnt: ModelEntity
DimComponent Struct
width: Float
height: Float
depth: Float
color: Color
rough: Float
metal: Bool
ARViewContainer
EditView
OptionView
ColorSelector
Screenshot
@ObsvObj: arView: ARViewController
arView.dimensions
EditUIElements
WidthBox
@Binding: width: Float
HeightBox
@Binding: height: Float
MetallicToggle
@Binding: metal: Bool
ColorBox
@Binding: color: Color
DepthSlider
@Binding: depth: Float
RoughnessSlider
@Binding: rough: Float

Other Notes:

ARViewContainer:

When displaying the AR View to the user, the view itself must conform to a SwiftUI view hierarchy, however, ARView conforms to a UIView instead. The ARViewContainer struct acts as a bridge between SwiftUI and UIKit, making UIViews compatible with SwiftUI views. When the app first launches, the struct will call the makeUIView function, which will initialize the AR configuration. The other function, updateUIView, is a function that gets called when data associated with any bindings or observed objects is changed. Since the majority of my AR functionality resides in a custom-made class, this function was left unused, but is required for the UIViewRepresentable protocol in the struct.