```swift
//
//  ARViewController.swift
//  ARHomeDesigner
//
//  Purpose:
//  The main class which controls several of the events that take place within the AR Scene.
//  This class is responsible for creating and running the AR View, as well as creating event handlers that listen and respond to user interaction, such as when the user taps on the screen or when they change a setting in the UI. Variables that are marked as published
//
//  Sources:
//  Nikhil Jacob - RealityKit tutorial: Plane Detection and Raycasting - https://youtu.be/T1u1tyMlMLM?si=qEeqGG4_Dumbdlej
//      Explains how to set up plane detection the placing of models using raycasting.
//      Code for functions like handleTap, createPlane and placeObject were modified based off of this example.
//  Nikhil Jacob - RealityKit tutorial: Rotate and Scale an Object With Gestures! - https://youtu.be/0yiF6I0JtiM?si=bL1pTO1y8U73vXFA
//      Explains how to give objects gesture recognition to rotate and scale objects
//      The code for addObjGestures function is heavily borrowed from this example
//
//  Created by Kyle McFadden on 3/22/25.
//

import ARKit
import RealityKit
import SwiftUICore

final class ARViewController: ObservableObject {
    // MARK: - Variables & Initialization

    var arView: ARView

    @Published var dimensions: DimComponent
    @Published var userColor: Color
    @Published var userDepth: Float
    @Published var userRoughness: Float
    @Published var isMetal: Bool
    @Published var toggleEditMode: Bool

    var confirmEdit: Bool {
        didSet {
            if confirmEdit {
```

```
            editObject(object: curModEnt, dimensions: dimensions)
            confirmEdit = false
        }
    }
}

var toggleSCRNshot: Bool {
    didSet {
        if toggleSCRNshot {
            toggleSCRNshot = false
            takeScreenshot()
        }
    }
}

private var curModEnt: ModelEntity

init() {
    arView = ARView(frame: .zero)
    toggleEditMode = false
    toggleSCRNshot = false
    dimensions = DimComponent()
    confirmEdit = false
    curModEnt = ModelEntity()
    userColor = .yellow
    userDepth = 0.02
    userRoughness = 0.0
    isMetal = false
}

// MARK: - DimComponent Struct

// Struct that contains the information for a model. Gets tied to
 the entity as a component.
struct DimComponent: Component {
    var width: Float
    var height: Float
    var depth: Float
    var color: Color
    var rough: Float
    var metal: Bool

    init(width: Float, height: Float, depth: Float, color: Color,
     rough: Float, metal: Bool) {
        self.width = width
        self.height = height
        self.depth = depth
        self.color = color
```

```
            self.rough = rough
            self.metal = metal
        }

        init() {
            width = 1.0
            height = 1.0
            depth = 0.02
            color = .yellow
            rough = 0.0
            metal = false
        }
    }

    // MARK: - Public Functions

    public func BeginARSession() {
        startARConfiguration()
        startGestureRecognition()
    }

    // MARK: - Private Functions

    private func startARConfiguration() {
        arView.automaticallyConfigureSession = false

        let configuration = ARWorldTrackingConfiguration()

        configuration.planeDetection = [.vertical, .horizontal]
        if
         ARWorldTrackingConfiguration.supportsSceneReconstruction(
         .meshWithClassification) {
            configuration.sceneReconstruction = .meshWithClassification
            configuration.frameSemantics.insert(.smoothedSceneDepth)
        }

        configuration.environmentTexturing = .automatic
        configuration.worldAlignment = .gravity
        configuration.frameSemantics.insert(
         .personSegmentationWithDepth)

        arView.environment.sceneUnderstanding.options.insert(.occlusion)

        #if DEBUG
//            arView.debugOptions.insert(.showSceneUnderstanding)
//            arView.debugOptions.insert(.showAnchorOrigins)
//            arView.debugOptions.insert(.showFeaturePoints)
//            arView.debugOptions.insert(.showWorldOrigin)
```

```
//            arView.debugOptions.insert(.showStatistics)
        #endif

        arView.session.run(configuration)
    }

    private func startGestureRecognition() {
        arView.addGestureRecognizer(UITapGestureRecognizer(target:
         self, action: #selector(handleTap(recognizer:))))
        arView
         .addGestureRecognizer(UILongPressGestureRecognizer(target:
         self, action: #selector(handleLongPress(recognizer:))))
    }

    // handleTap
    // Handles when the user taps on the screen when they wish to add a
     new object to the scene
    //
    @objc
    private func handleTap(recognizer: UITapGestureRecognizer) {
        // Touch Location
        let tapLocation = recognizer.location(in: arView)

        if let entity = arView.entity(at: tapLocation) as? ModelEntity
         { // Find the nearest entity at the point of the tap
            for component in entity.components {
                guard let dimComp = component as? DimComponent else {
                 continue } // Finds the component with the
                 DimComponent type (only one component of a given type
                 can exist within the sequence)
                dimensions = dimComp // Grab the dimensions of the
                 current object/model for the UI views to display
                curModEnt = entity // Grab the object/model that the
                 user wishes to edit

                dimensions.width *= curModEnt.scale.x
                dimensions.height *= curModEnt.scale.z
                toggleEditMode = true // Update will trigger a change
                 in the UI view
                break
            }
        } else {
            // Ray casting - Converting a 2D point on the app view to a
             3D point on the virtual plane
            let results = arView.raycast(from: tapLocation, allowing:
             .existingPlaneGeometry, alignment: .any)

            if let firstResult = results.first {
```

```
            // Get the anchor of the plane that the raycast
             intersected
            guard let planeAnchor = firstResult.anchor as?
             ARPlaneAnchor else { return }

            // Gets the dimensions of the plane / sets up the
             components that will be used to create the plane
            let objDims = DimComponent(width:
             planeAnchor.planeExtent.width,
                                       height:
                                        planeAnchor.planeExtent
                                        .height,
                                       depth: userDepth,
                                       color: userColor,
                                       rough: userRoughness,
                                       metal: isMetal)

            // Make the object/model entity
            let plane = createPlane(dimensions: objDims) // Returns
             a Model Entity object

            //            #if DEBUG
            //                let component =
             ModelDebugOptionsComponent(visualizationMode: .normal)
            //                plane.modelDebugOptions = component
            //            #endif

            // Place object at the center of the plane relative to
             the location of the plane anchor of the estimated
             plane that the raycast intersected
            placeObject(object: plane, at: planeAnchor.transform,
             center: planeAnchor.center, with:
             planeAnchor.planeExtent, with: objDims.depth)

            // Install gestures to the object - Rotation,
             translation, and scale
            // Basic Version - will be working on making a more
             user friendly and robust version -- Essentially
             allowing the user to tap on an object and given them
             unlimited range to scale and rotate and translate the
             object without needing to interact directly with said
             object
            addObjGestures(on: plane)
        }
    }
}

@objc
```

```
private func handleLongPress(recognizer: UITapGestureRecognizer) {
    // Touch location
    let tapLocation = recognizer.location(in: arView)

    // Find the entity at the point of the long press
    if let entity = arView.entity(at: tapLocation) as? ModelEntity {
        if let anchorEntity = entity.anchor {
            anchorEntity.removeFromParent()
        }
    }
}

private func createPlane(dimensions: DimComponent) -> ModelEntity {
    // Create the mesh: give the plane a proper width and depth
    let objectMesh = MeshResource.generatePlane(width:
     dimensions.width, depth: dimensions.height)

    // Create material
    let objectMaterial = SimpleMaterial(color:
     UIColor(dimensions.color), roughness:
     MaterialScalarParameter(floatLiteral: dimensions.rough),
     isMetallic: dimensions.metal)

    // Model Entity
    let objectEntity = ModelEntity(mesh: objectMesh, materials:
     [objectMaterial])

    // Add dimension component to the entity (for storing size and
     color)
    objectEntity.components.set<DimComponent>(dimensions)

    return objectEntity
}

private func placeObject(object: ModelEntity, at location:
 simd_float4x4, center: SIMD3<Float>, with extent: ARPlaneExtent,
 with depth: Float) {
    // Make anchor entity at the location of the existing plane
     anchor
    let entityAnchor = AnchorEntity(world: location)

    // Translate the modelEntity to the center of the plane.
    object.transform.translation = center + [0.0, depth, 0.0]

    // Rotate anchor to match plane orientation
    entityAnchor.transform = Transform(pitch: 0, yaw:
     extent.rotationOnYAxis, roll: 0)
```

```
        // Tie model to anchor
        entityAnchor.addChild(object)

        // Add anchor (that has the model entity tied to it) to the
         scene
        arView.scene.addAnchor(entityAnchor)
    }

    private func editObject(object: ModelEntity, dimensions:
     DimComponent) {
        // Find the anchor that the existing model is tied to
        let entityAnchor = object.parent as! AnchorEntity

        let updPlane = createPlane(dimensions: dimensions)

        // Give the new plane the existing rotation/position relative
         to anchor from the old plane
        updPlane.transform.translation = [object.position.x,
         dimensions.depth, object.position.z]
        updPlane.transform.rotation = object.orientation

        object.removeFromParent()

        // Restore the model entity with the new model and give it
         gesture collision
        entityAnchor.addChild(updPlane)
        addObjGestures(on: updPlane)

        // Update the current model in scope for making further edits
        curModEnt = updPlane
    }

    private func addObjGestures(on object: ModelEntity) {
        // Give the entities collision which are needed for gesture
         recognition
        object.generateCollisionShapes(recursive: true)
        // Put gestures onto the model to allow the user to rotate,
         scale and translate the object by using global gestures while
         interacting with said object
        arView.installGestures([.rotation, .scale, .translation], for:
         object)
    }

    private func takeScreenshot() {
        arView.snapshot(saveToHDR: false) { image in
            let compressedImage = UIImage(data: (image?.pngData())!)
```

```
            UIImageWriteToSavedPhotosAlbum(compressedImage!, nil, nil,
             nil)
        }
    }
}
```