HomeViewer

Developer Guide

An Apple Vision Pro app for visualising interior paint on real walls

Hunter Thiel

Computer Science & Mathematics, St. Norbert College

May 3, 2026

Introduction

HomeViewer is a mixed-immersion visionOS app that runs on Apple Vision Pro (visionOS 2.0 and newer). The app lets a user walk through a real room while wearing the headset, see every detected wall highlighted in cyan, and "paint" individual walls with any colour the user chooses. The result is rendered as a physically-based overlay that respects the actual room lighting, can be made glossy or matte, and stays correctly anchored even as the user moves around. The same scan also drives an occlusion mesh layer so virtual content tucks behind real geometry — including furniture, ceilings, and openings such as doors and windows — instead of floating in front of it.

Built end-to-end in Swift with SwiftUI for the 2D dashboard and RealityKit + ARKit for the immersive scene, the project leans heavily on visionOS-native APIs introduced in the `ARKitSession` / `DataProvider` model. Two providers run concurrently in a single session: a `PlaneDetectionProvider` filtered to vertical planes (walls, doors, windows), and a `SceneReconstructionProvider` that emits a continuously refined triangle mesh of the room. This guide walks through how those providers feed the RealityKit scene graph, how the five Swift source files share state, and what every variable, gesture, and method in the codebase is responsible for.

This guide assumes the reader is already comfortable with Swift, SwiftUI, and Xcode. Familiarity with traditional iOS ARKit is helpful but not required — visionOS uses a different, fully-immersive scene model and the relevant pieces are introduced as they appear. The focus here is the AR architecture and how the components interact.

Table of Contents

Introduction 2
Table of Contents 2
Setting Up 4
visionOS Foundations 5
- **ARKitSession and Data Providers** 5
- **Plane Detection** 5
- **Scene Reconstruction** 5
- **The RealityKit Scene Graph** 5
- **Physically-Based Rendering and Ambient Occlusion** 6
- **Gestures** 7

Overall App Structure 8
The ARKit Anchor-Update Pipeline 9
Selection and Editing 10
Per-File Reference 12
- **HomeViewerApp.swift** 12
- **AppModel.swift** 12

ContentView.swift **14**
ImmersiveView.swift **15**
Stored properties **15**
body — wiring summary **15**
Gestures **15**
Methods **16**
MeshAnchorGenerator.swift **17**
Notes and Conventions **19**

Setting Up

To build and run HomeViewer you need a Mac with Xcode 16 or newer (visionOS 2 SDK installed) and a physical Apple Vision Pro device. The simulator can render the windowed UI but cannot test the ARKit features — scene reconstruction and plane detection both depend on the device cameras and LiDAR, so a real headset is required for any real validation work.

Required capabilities. The app's Info.plist must request world-sensing authorization. visionOS prompts the user the first time `ARKitSession.requestAuthorization(for: [.worldSensing])` is called, which the app does inside the `.task` modifier on `ImmersiveView`. If the user denies authorization, both providers silently stop emitting updates and the immersive space will appear empty.

Project layout. All five source files live under the `HomeViewer/` target. There are no third-party dependencies; the project compiles against `SwiftUI`, `RealityKit`, `RealityKitContent` (the auto-generated package), and `ARKit`.

Running on device. Pair the headset to Xcode (Window → Devices and Simulators), then build with the headset selected as the run destination. The first run will install a development provisioning profile; subsequent runs deploy the app immediately. The 2D dashboard window opens first; tapping *Start Scanning* opens the immersive space and begins the ARKit session.

visionOS Foundations

HomeViewer is built on top of four visionOS APIs: `ARKitSession`, `PlaneDetectionProvider`, `SceneReconstructionProvider`, and `RealityView`. The sections below give just enough background on each to make the rest of this document make sense.

ARKitSession and Data Providers

On visionOS, ARKit is no longer organised around a single `ARView` subclass with a configuration object. Instead, an `ARKitSession` is a lightweight coordinator that runs one or more `DataProvider` instances concurrently. Each provider exposes its results through an async sequence — typically `.anchorUpdates` — that the app consumes inside a Swift `Task`. HomeViewer runs both providers in a single session call, `session.run([sceneReconstruction, planeProvider])`, and reads them in two parallel `.task` modifiers attached to `ImmersiveView`'s body.

Plane Detection

`PlaneDetectionProvider` emits `PlaneAnchor` updates whenever the system finds, refines, or loses a flat surface. HomeViewer initialises it with `alignments: [.vertical]` so it only receives vertical planes — that filter alone is what makes this a wall-painting app rather than a generic plane visualiser. Each `PlaneAnchor` carries:

- a `UUID` used as the lookup key in `planeEntities` and most of `AppModel`'s dictionaries;
- an `originFromAnchorTransform` — the world-space pose of the plane's local origin;
- a `geometry.extent` containing detected width, height, and an `anchorFromExtentTransform` that re-centres the plane on its true geometric centre (the anchor origin and the centre rarely match);
- a `surfaceClassification` enum: `.wall`, `.floor`, `.ceiling`, `.door`, `.window`, `.table`, and others. HomeViewer branches on this to decide whether to paint the plane or merely occlude it.

Scene Reconstruction

`SceneReconstructionProvider` emits `MeshAnchor` updates that together approximate the entire room as a continuously updated triangle mesh. Walls, sofas, plants, the user's own arm — everything visible to the cameras and LiDAR is in there. HomeViewer doesn't render this mesh visibly. Instead, `MeshAnchorGenerator` wraps every mesh anchor in a `ModelEntity` with an `OcclusionMaterial`, which writes to the depth buffer but skips the colour buffer. The effect is invisible geometry that occludes anything behind it — so when the user moves a virtual paint chip behind a real chair, the chair correctly hides it.

The RealityKit Scene Graph

Everything the user sees in the immersive space is an `Entity` attached, directly or transitively, to `RealityView`'s `content`. HomeViewer organises the scene into two parallel subtrees, both attached to the content root at startup:

- `rootEntity` — the parent of every painted wall overlay (and the door/window cut-outs).
- `occlusionRoot` — the parent of every scene-reconstruction occlusion mesh.

Inside `rootEntity`, each detected plane gets its own three-level mini-hierarchy so that the world transform, the float-off-the-wall offset, and the mesh itself can be manipulated independently. The full layout is shown in the diagram below.

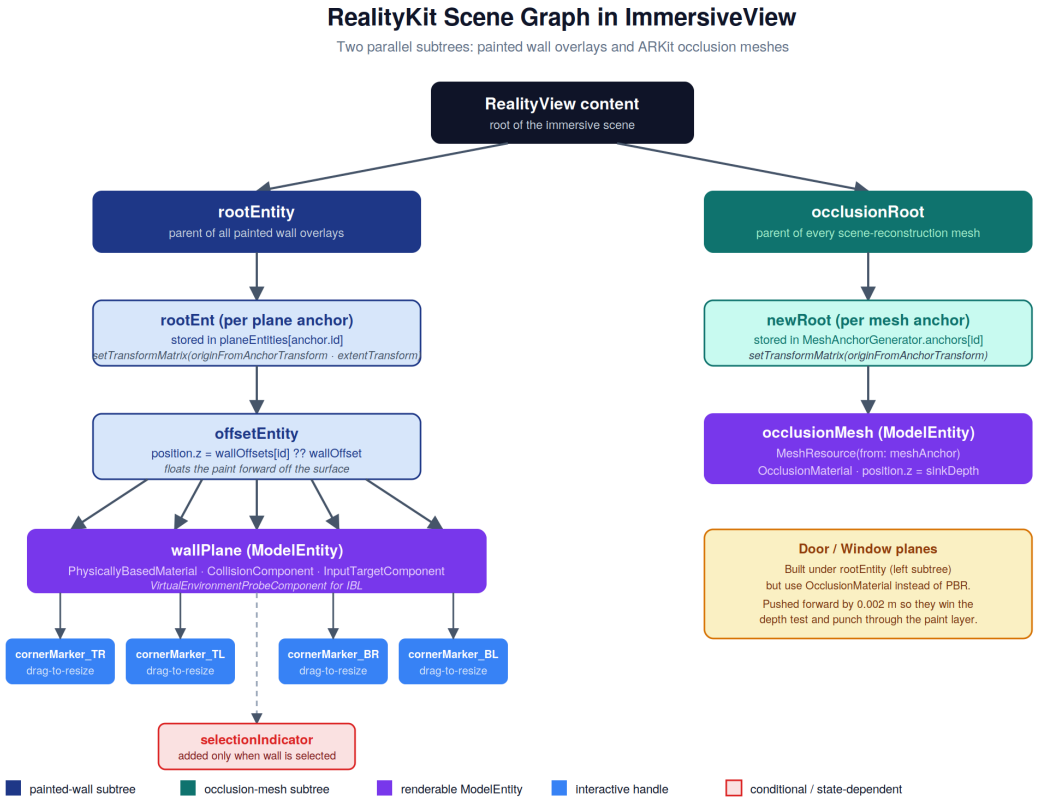

Figure 1. RealityKit scene graph used by ImmersiveView.

Physically-Based Rendering and Ambient Occlusion

Wall paint is drawn with a `PhysicallyBasedMaterial` — RealityKit's full PBR shader — rather than a flat colour. The material's `baseColor`, `roughness`, `clearcoat`, `specular`, and `blending.opacity` are all driven by sliders in the dashboard, so the user can dial in anything from chalky matte to wet-look gloss. A `VirtualEnvironmentProbeComponent` attached to each wall tells RealityKit to sample the actual room's image-based lighting (IBL) when shading that surface, which is what makes the painted overlay react to real light and shadow.

The painted overlay also gets a procedurally generated ambient-occlusion (AO) texture: a 256×256 grayscale image that's white in the centre and gently darkens toward the edges. Because the falloff is computed in physical metres rather than UV space, narrow walls and wide walls show shadow bands of the same physical thickness. The AO texture is applied through `PhysicallyBasedMaterial.ambientOcclusion` and is regenerated whenever a wall is resized so the falloff always matches the current geometry.

Gestures

All interaction inside the immersive space goes through three gestures attached to `ImmersiveView`'s body. Each is targeted to entities by name, walking up the parent chain to find the anchor's bookkeeping. `grabWallGesture` is registered with `simultaneousGesture(...)` because two `DragGestures` would otherwise compete for priority with `dragCornerGesture`. The state machine is described in the section *Selection and Editing*.

Overall App Structure

HomeViewer is split across five source files. **`HomeViewerApp.swift`** is the entry point and creates the single `AppModel` instance that all other views see via SwiftUI's `@Environment`. **`ContentView.swift`** renders the 2D dashboard window and writes back to the model through `@Bindable` property accessors. **`ImmersiveView.swift`** owns the ARKit session and the RealityKit scene; it reads sliders from the model on every `.onChange` and writes per-wall state (colors, dimensions, offsets) back into the model so it survives anchor updates. **`MeshAnchorGenerator.swift`** is a small helper class that consumes scene-reconstruction updates and maintains the occlusion mesh layer, reading `sinkDepth` and `isUpdatingWalls` from the same shared model. **`AppModel.swift`** itself is a single `@Observable` class that holds every piece of cross-component state.

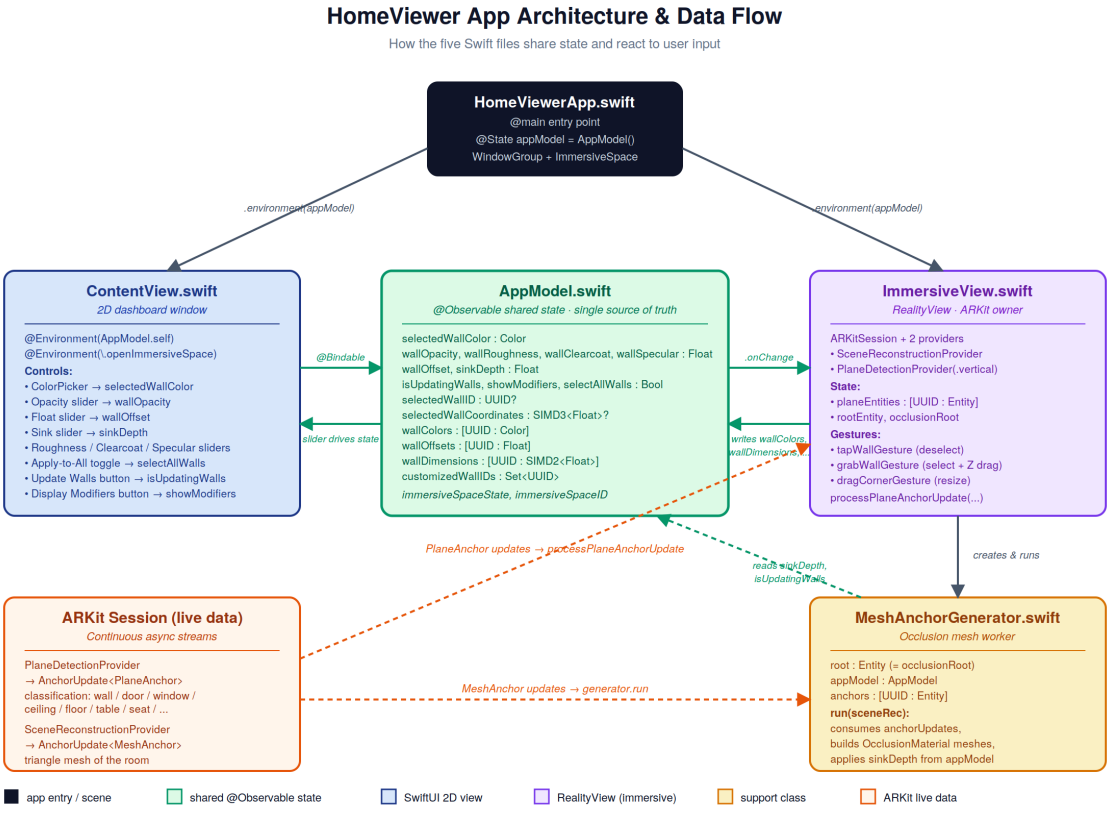

Figure 2. Data flow between the five source files. Solid arrows mark direct construction or environment injection; dashed arrows mark async streams from ARKit; green arrows mark observable-state read/write.

The ARKit Anchor-Update Pipeline

Both providers run for the lifetime of the immersive space. Every plane and mesh update flows through the `isUpdatingWalls` gate first — when the user toggles *Stop Updating* from the dashboard, only `.removed` events are processed, which freezes the scan but still cleans up disappeared anchors. Plane updates that pass the gate are dispatched in `processPlaneAnchorUpdate` by classification: walls get a paintable PBR mesh with corner handles; doors and windows get an occlusion punch-through; everything else gets a thin occlusion box.

Mesh updates take a simpler path: `MeshAnchorGenerator` turns each `MeshAnchor` into a `MeshResource` with `OcclusionMaterial`, positions it by the anchor's world transform, and applies `sinkDepth` along the local Z axis so the occlusion mesh sits slightly behind the painted overlay rather than fighting it for depth.

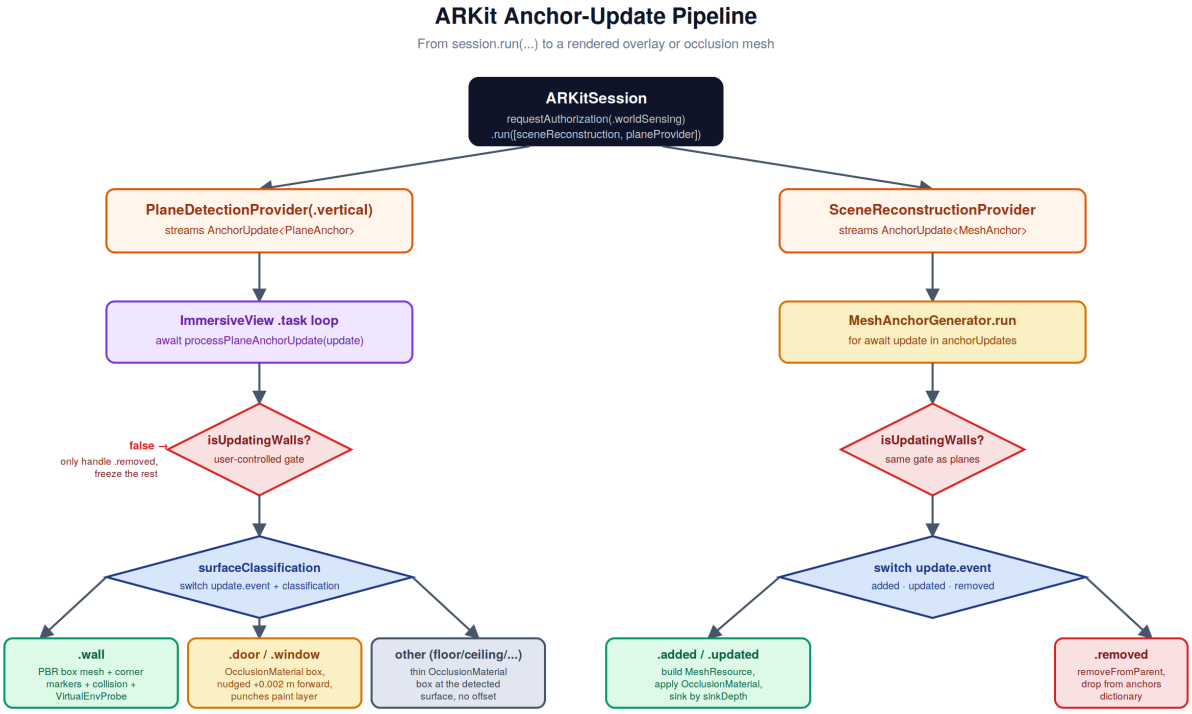

Figure 3. The two anchor-update streams and how each is dispatched.

Selection and Editing

Wall selection is encoded by a single optional UUID, `appModel.selectedWallID`. The selection state machine has only two states — selected and not-selected — but the transitions involve all three gestures and a few persistent pieces of state.

Grabbing a wall with `grabWallGesture` both selects it (on the first frame of the drag) and adjusts that wall's individual Z-offset live (on every subsequent frame). The Z-offset is computed by converting the live drag location into `rootEnt`-local space, where the local Z axis is the wall's outward normal. The result is clamped to the same range as the global *Wall Float* slider and persisted in `wallOffsets[anchorID]` so future ARKit anchor updates restore the user's adjustment instead of snapping back to the global default.

Tapping a wall with `tapWallGesture` only does anything if that wall is *already* selected — in which case it deselects. Selecting a different wall is the grab-gesture's job. This split makes the headset's pinch behave as a smart, modal action: pinching empty space changes nothing, pinching a wall picks it up, and pinching the picked-up wall again drops it.

Dragging a corner marker with `dragCornerGesture` resizes the wall. As soon as a corner is touched, the wall's UUID is added to `customizedWallIDs`, which acts as a flag telling `processPlaneAnchorUpdate` to leave that wall's geometry alone on subsequent ARKit refreshes — only the world transform is still applied, so the wall stays attached to the right physical surface even though its dimensions are now under user control. The drag itself updates the dragged corner plus its two adjacent corners on the shared axis, then rebuilds the box mesh, the collision shape, and the AO texture from the new dimensions.

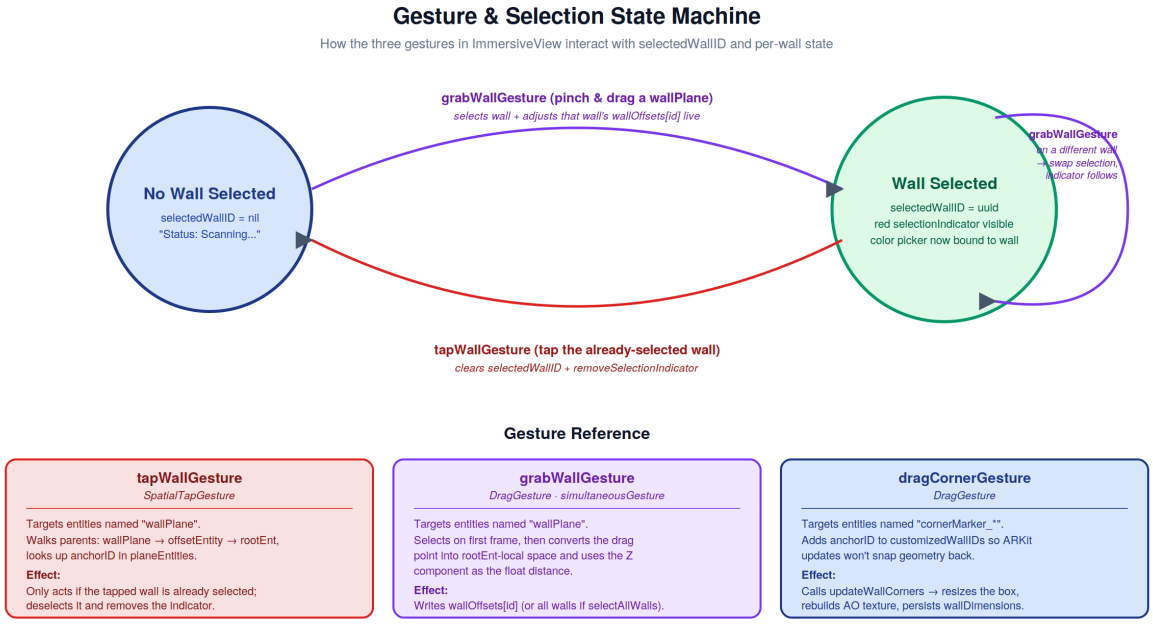

Figure 4. Selection state machine and what each gesture does to it.

Per-File Reference

This section walks through every public property, gesture, and method in the codebase, organised by source file. Tables are kept tight; longer explanations sit in prose between them.

HomeViewerApp.swift

The `@main` entry point. It owns the only `AppModel` instance, exposes a `WindowGroup` containing `ContentView`, and registers an `ImmersiveSpace` scene whose ID matches `appModel.immersiveSpaceID`. Both scenes inject the app model into their environment, which is how every downstream view picks it up.

The two lifecycle hooks attached to `ImmersiveView` — `.onAppear` and `.onDisappear` — drive `appModel.immersiveSpaceState` between `.open` and `.closed`, which `ContentView` reads to switch between the welcome screen and the dashboard. The scene declaration ends with `.immersionStyle(selection: .constant(.mixed), in: .mixed)`, forcing mixed mode so virtual content composites over the real passthrough video rather than replacing it.

AppModel.swift

Single `@Observable` class, single instance per app run. Every other file mutates it directly. The `ImmersiveSpaceState` enum models the three-phase lifecycle of opening the space (`.closed`, `.inTransition`, `.open`) so the dashboard can show a spinner while the immersive space is being requested.

Property	Type	Purpose
`immersiveSpaceState`	`ImmersiveSpaceState`	Tracks whether the immersive scene is closed, opening, or open. Drives the welcome / loading / dashboard switch in ContentView.
`immersiveSpaceID`	`String`	Constant ID used both when registering the scene in HomeViewerApp and when calling openImmersiveSpace from ContentView.
`selectedWallColor`	`Color`	The colour shown in the dashboard ColorPicker. Writing to it triggers ImmersiveView's onChange, which applies the colour to the selected wall (or every wall, if selectAllWalls is on).
`wallOpacity`	`Double`	Global paint transparency, 0.0 to 1.0. Fed into PhysicallyBasedMaterial.blending so the painted overlay can be made see-through.

Property	Type	Purpose
isUpdatingWalls	Bool	Master gate for both anchor-update streams. While false, only .removed events are honoured — geometry is frozen but disappeared anchors still get cleaned up.
showModifiers	Bool	Toggles visibility of the blue corner-marker spheres on every wall. Hidden by default to keep the scene uncluttered until the user wants to resize.
wallColors	[UUID: Color]	Per-wall saved colour. Survives ARKit anchor updates, so a re-detected wall comes back painted instead of resetting to cyan.
customizedWallIDs	Set<UUID>	Walls whose geometry the user has manually resized. processPlaneAnchorUpdate will not overwrite their mesh on subsequent ARKit updates.
wallOffset	Float	Global Z float distance in metres — how far the painted overlay sits in front of the physical wall surface. Default 2 cm.
sinkDepth	Float	Global Z displacement applied to every occlusion mesh. Negative values push the mesh into the wall to prevent z-fighting with the paint layer. Default −2 cm.
wallRoughness	Float	PBR roughness, 0 (mirror) to 1 (matte). Drives the Roughness slider.
wallClearcoat	Float	PBR clearcoat layer intensity. Adds a wet/varnish sheen on top of the base paint.
wallSpecular	Float	PBR specular intensity. 0.5 is the standard non-metal default.
selectedWallID	UUID?	Anchor UUID of the currently-selected wall, or nil. The single source of truth for the selection state machine.
selectedWallCoordinates	SIMD3<Float>?	World-space position of the selected wall, captured at selection time. Currently consumed only as a debug hook; useful for future world-space UI.
selectAllWalls	Bool	Apply-to-all toggle. When true, colour and Z-offset changes broadcast to every detected wall instead of just the selected one.
wallOffsets	[UUID: Float]	Per-wall Z-offset overrides set by grabWallGesture. A wall with an entry here ignores the global wallOffset slider on subsequent updates.

Property	Type	Purpose
`wallDimensions`	`[UUID: SIMD2<Float>]`	Detected or user-resized width/height in metres. Read by createWallMaterial to scale the AO texture's edge falloff in physical units.

ContentView.swift

The 2D dashboard window. It branches on `immersiveSpaceState` three ways: a welcome screen with a *Start Scanning* button when the space is `.closed`; a spinner when it's `.inTransition`; and the full two-column control panel when it's `.open`. The *Start Scanning* button calls `openImmersiveSpace(id:)` inside a `Task` and resets the state to `.closed` if the user cancels or the open fails.

Control	What it does
`Status text`	Shows “Scanning...” when no wall is selected and “Wall Selected” once one is. The selected-state branch renders the colour picker and the apply-to-all toggle; the unselected branch shows a hint to pinch a highlighted wall.
`ColorPicker`	Bound to selectedWallColor. ImmersiveView's onChange handler applies the new colour to the selected wall (or all walls).
`Apply Color to All Walls`	Toggle bound to selectAllWalls. Active immediately — the next colour or grab broadcasts.
`Global Wall Opacity`	Slider 0.0 to 1.0, bound to wallOpacity. Triggers a full PBR rebuild via updateAllWallMaterials.
`Wall Float`	Slider 0.005 to 0.05 m, bound to wallOffset. Walls with a per-wall override (set via grab) are intentionally skipped by updateAllWallOffsets.
`Mesh Sink`	Slider −0.10 to 0.05 m, bound to sinkDepth. Repositions every occlusion mesh's local Z so the occluder sits behind the paint without z-fighting.
`Roughness / Clearcoat / Specular`	Three PBR sliders. Each writes its value into AppModel and triggers updateAllWallMaterials, which rebuilds the PhysicallyBasedMaterial on every wall.

Control	What it does
`Stop / Resume Updating`	Toggles isUpdatingWalls. The button label and tint flip to reflect the current state. Useful once the room is fully scanned: freezing prevents subtle anchor refinements from snapping the paint around.
`Display / Hide Modifiers`	Toggles showModifiers, which calls updateModifiersVisibility in ImmersiveView and enables/disables every cornerMarker entity.

ImmersiveView.swift

The largest and most central file. It owns the `ARKitSession`, both providers, and the two scene-graph roots, and it wires up every gesture and `.onChange` handler. Two parallel `.task` loops consume the provider streams continuously for the lifetime of the immersive space.

Stored properties

Name	Type	Purpose
`session`	`ARKitSession`	The single session that runs both providers concurrently.
`sceneReconstruction`	`SceneReconstructionProvider`	Streams MeshAnchor updates that MeshAnchorGenerator turns into occlusion meshes.
`planeProvider`	`PlaneDetectionProvider`	Initialised with [.vertical] so only walls, doors, and windows are emitted.
`appModel`	`@Environment AppModel`	Shared observable state. Read on every onChange and gesture frame; written when persisting per-wall colour, dimensions, and offsets.
`planeEntities`	`@State [UUID: Entity]`	Maps each plane anchor's UUID to the rootEnt entity at the top of that wall's three-level hierarchy.
`rootEntity`	`@State Entity`	Parent of every painted wall overlay (and of door/window punch-throughs).
`occlusionRoot`	`@State Entity`	Parent of every scene-reconstruction occlusion mesh, owned by MeshAnchorGenerator.

body — wiring summary

The body sets up the `RealityView` with both root entities, attaches the three gestures (one as `simultaneousGesture` to coexist with the corner-drag), declares an `.onChange` for every reactive slider, and starts three `.task` loops: one for ARKit authorization plus `session.run`, one for the mesh anchor pipeline, and one for the plane anchor pipeline.

Gestures

Gesture	Behaviour
`tapWallGesture`	SpatialTapGesture, targeted to any entity. Only acts on entities named “wallPlane” and only if that wall is already selected. Walks wallPlane → offsetEntity → rootEnt to recover the anchor UUID, then clears selectedWallID and removes the red selection indicator.
`dragCornerGesture`	DragGesture, targeted to any entity. Filters to entities whose names start with “cornerMarker_”. On change, marks the wall as user-customized, converts the drag

Gesture	Behaviour
	location into offsetEntity-local space (with Z forced to 0 so dragging only resizes width/height), and calls updateWallCorners.
`grabWallGesture`	DragGesture, registered with simultaneousGesture so it doesn't compete with dragCornerGesture. Fires on entities named “wallPlane”. On the first frame, if the grabbed wall isn't already selected, removes any previous indicator, registers the new selection, syncs the colour picker to the wall's saved colour, and adds the indicator. Every frame, converts the live drag location into rootEnt-local space, clamps the Z to 0.005–0.05 m, and writes the result into wallOffsets[id] (or every wall, if selectAllWalls is on).

Methods

Method	What it does
`processPlaneAnchorUpdate`	Async @MainActor. The dispatcher for every PlaneDetectionProvider event. While isUpdatingWalls is false, it short-circuits to handle only .removed. For .added/.updated, it skips geometry mutation on user-customized walls (re-applying just the world transform), then branches by surfaceClassification: walls get a paintable PBR box, four corner markers, a CollisionComponent, an InputTargetComponent, and a VirtualEnvironmentProbeComponent; doors and windows get a coplanar OcclusionMaterial box nudged 2 mm forward to win the depth test; everything else gets a thin OcclusionMaterial box at the detected surface.
`createWallMaterial`	Builds a fresh PhysicallyBasedMaterial from the current AppModel sliders plus the supplied colour and physical dimensions. Sets baseColor, roughness, clearcoat, specular, and transparent blending; never sets metallic (walls are not metal); adds the AO texture generated by generateAOTexture.
`generateAOTexture`	Builds a 256×256 grayscale TextureResource whose centre is full white and whose edges fall off to 60% brightness over a 30 cm physical falloff distance. Multiplies the two axis falloffs so corners are darkest, then smoothsteps the gradient. Returned as .raw semantic so RealityKit doesn't gamma-correct the data.
`updateAllWallMaterials`	Iterates every wall in planeEntities and rebuilds its PBR material with the wall's saved colour and dimensions. Called from the roughness, clearcoat, specular, and opacity onChange handlers.
`updateAllWallOffsets`	Slides every offsetEntity's local Z to the new global wallOffset. Walls with a per-wall override in appModel.wallOffsets are intentionally skipped so the user's grab-adjustment isn't clobbered.

Method	What it does
`updateAllSinkDepths`	Adjusts the local Z of every occlusionMesh under occlusionRoot to the new sinkDepth. Mirrors what MeshAnchorGenerator does inline on each new anchor update, but runs immediately when the slider changes.
`updateWallCorners`	The geometry-resize core. Moves the dragged corner to the new position, then nudges its two neighbours so the wall stays a rectangle (horizontal neighbour shares Y, vertical neighbour shares X). Recomputes width, height, and centre from the four corner positions; rebuilds the box mesh and the collision shape; persists the new dimensions into wallDimensions; and regenerates the PBR material so the AO falloff matches the new physical size.
`updateModifiersVisibility`	Walks every wall's offsetEntity and toggles isEnabled on each child whose name starts with “cornerMarker”. Driven by the Display / Hide Modifiers button.
`addSelectionIndicator`	Attaches a small red unlit sphere 2 cm in front of the given wallPlane to mark it as selected. Unlit so it stays visible regardless of room lighting.
`removeSelectionIndicator`	Finds and removes the entity named “selectionIndicator” from the given parent.

MeshAnchorGenerator.swift

A small helper class that owns the occlusion mesh layer. ImmersiveView constructs it inside a `.task` and calls `run(_:)`, which never returns until the immersive space closes.

Member	Type	Purpose
`root`	`Entity?`	Parent for every newly created occlusion mesh. Set to ImmersiveView's occlusionRoot at construction. Optional purely for safety; in practice it lives the whole session.
`appModel`	`AppModel`	Held by reference so the run loop can read sinkDepth and isUpdatingWalls every iteration.
`anchors`	`[UUID: Entity]`	Maps each MeshAnchor UUID to the rootEnt that wraps its occlusionMesh child. Used to update existing entities on .updated events and to remove them on .removed.
`init(root:appModel:)`	`init`	Wires the parent entity and the shared model.
`run(_:)`	`async @MainActor`	Awaits every anchorUpdate from the SceneReconstructionProvider. Honours isUpdatingWalls (handles only .removed when the

Member	Type	Purpose
		gate is closed). For .added/.updated, finds or creates a two-level entity (newRoot → occlusionMesh), builds a fresh MeshResource from the anchor, applies an OcclusionMaterial, sets the parent transform from originFromAnchorTransform, and shifts the mesh entity's local Z to sinkDepth. .removed simply detaches the entity and drops it from the dictionary.

Notes and Conventions

Naming. Entities are identified by name strings (“wallPlane”, “offsetEntity”, “cornerMarker_TR”, etc.) so gestures and helpers can recover them via `findEntity(named:)`. Gestures filter on these names before doing anything; touching this convention requires updating every filter.

Coordinate spaces. Three local frames matter throughout the file. `rootEnt`'s frame has its Z axis pointing along the wall's outward normal, so a positive Z translation floats things forward. `offsetEntity`'s frame is identical except shifted along Z by the wall-float amount; corner markers and the wall mesh live in this frame. `wallPlane`'s frame is the same again with no further shift. Most code that converts between SwiftUI drag locations and entity positions does so through `value.convert(_:from:to:)` and is explicit about which space it wants.

Persistence across anchor updates. ARKit can refine, replace, or briefly drop an anchor at any time. The four dictionaries on `AppModel` — `wallColors`, `wallOffsets`, `wallDimensions`, and the `customizedWallIDs` set — exist precisely so user choices outlive these refinements. Whenever you add a new piece of per-wall state, persist it through the same pattern: store it under `anchor.id`, read it back at the top of `processPlaneAnchorUpdate`'s `.added/.updated` branch, and remove it in `.removed` alongside the others.

Future hooks. A few obvious extension points are already implied by the code. `selectedWallCoordinates` is captured but not yet rendered into world-space UI. `Metallic` is hard-pinned to 0 inside `createWallMaterial` but could become another slider for metallic accent walls. The classification switch in `processPlaneAnchorUpdate` intentionally has only three branches; richer behaviour for `.table` or `.seat` surfaces would slot in there.