Article

Adopting best practices for persistent UI

Create persistent and contextually relevant spatial experiences by managing scene restoration, customizing window behaviors, and surface snapping data.

Overview

In shared and mixed immersive spaces on visionOS, people can snap and lock app content directly to their physical environment. This persistence allows apps to remain in place even when the person leaves and returns, or restarts their device. You can use this capability to create deeply integrated spatial experiences that people will naturally return to.

> **Note**
>
> This behavior is unavailable in full or progressive immersive spaces.

People can anchor apps to their space in a few different ways:

- **Snapping** — When a person moves a window near a vertical surface, like a wall, or a volume near a horizontal surface, like a table or floor, your app can automatically align with or snap to that surface.
- **Locking** — When an app snaps to a surface, the system places a lock icon in the window bar to indicate it's locked in place if the application supports restoration of that window.
- **Persistence** — After locking the app in place, your content remains fixed to its physical location. It persists even if the person moves to a different room, takes the device off or on, or recenters their view. When moving between rooms, locked content from the previous room fades out, and locked content in the new room appears. Content that isn't locked in place always launches relative to the person's position.

Locking and persisting content in visionOS makes the user experience more natural and integrated by letting people attach apps to specific places in their real world. This means people can easily return to what they were doing, creating a smoother and more immersive experience. Knowing the kind of surface the app snaps to allows for more useful and personalized apps, giving developers the ability to show content that's relevant to a given situation.

> **Note**
>
> Watch the WWDC25 session Set the scene with SwiftUI in visionOS for more information on system behavior and API usage.

Handle scene restoration

Scene restoration is an important part of a seamless experience on visionOS. People expect content to persist where they've placed it, and they expect to be able to pick up where they left off. To ensure the system correctly restores windows and maintains a person's desired layout, you'll need to handle your app's scene restoration.

> **Note**
>
> To learn more, download the sample code project in Restoring your app's state with SwiftUI.

However, there are some cases in which you might not want to restore scenes, especially for those that serve a temporary or context-specific purpose, including:

- **Transient elements** — Welcome screens, onboarding flows, or temporary alerts.
- **Context-dependent UI** — Tool palettes or inspectors whose relevance is tied to a specific state or document that might not exist upon relaunch.
- **Completed one-time actions** — Login prompts or import and export dialogs.

You can opt out of scene restoration on a per-scene basis by applying `.restorationBehavior(.disabled)` to any `Window` or `WindowGroup`. The default behavior is `automatic`, which restores your content.

```
WindowGroup(id: "to-do") {
    // The main body of a to-do list app.
    ToDoList()
}

Window("Onboarding", id: "onboarding") {
```

```
    // The flow that takes the person through how the app works.
    OnboardingView()
}
.restorationBehavior(.disabled)
```

Customize window launch behavior

It's important to launch only essential windows to prevent redundancy and confusion. You can use `.defaultLaunchBehavior` to fine-tune which windows appear when a person relaunches your app. By default, your app presents the first window declared in your `Scene` matching the "Preferred Default Scene Session Role" key in your `Info.plist`.

To prioritize presenting a specific window on launch, use `.presented`:

```
@AppStorage("needsOnboarding") private var needsOnboarding = true

var body: some Scene {

    Window("Onboarding", id: "onboarding") {
        // The flow that takes the person through how the app works.
        OnboardingView()
    }
    // Prevent state restoration for this window.
    .restorationBehavior(.disabled)

    WindowGroup(id: "to-do") {
        // The main body of a to-do list app.
        ToDoList()
    }
    // Presents the main body if the person doesn't need the onboarding experience.
    .defaultLaunchBehavior(needsOnboarding ? .automatic : .presented)
}
```

For secondary windows, use `.suppressed` to ensure it doesn't open when the app relaunches. This is ideal for transient windows like onboarding screens, toolbars, or other UI elements that shouldn't reappear every time a person opens the app.

```
@AppStorage("hasOnboarded") private var needsOnboarding = true

var body: some Scene {
    Window("Onboarding", id: "onboarding") {
```

```
        // The flow that takes the person through how the app works.
        OnboardingView()
    }
    // Prevent state restoration for this window.
    .restorationBehavior(.disabled)

    WindowGroup(id: "to-do") {
        // The main body of a to-do list app.
        ToDoList()
    }
    // Presents the main body if the person doesn't need the onboarding experience.
    .defaultLaunchBehavior(needsOnboarding ? .automatic : .presented)

    WindowGroup(id: "settings") {
        SettingsView()
    }
    // Suppresses launch of this window when the app is re-opened.
    .defaultLaunchBehavior(.suppressed)
}
```

Present content dynamically using surface snapping

You can use a scene's physical placement to show content dynamically in an immersive app. Knowing where a person has placed content in their physical space allows you to integrate your app more deeply with their real world and make your app more contextually relevant.

To ensure your app has access to the physical placement:

- **Request ARKit data access** — Follow the steps outlined in Setting up access to ARKit data.
- **Enable detailed surface info** — Add the `UIWantsDetailedSurfaceInfo` key to your app's `Info.plist` file and set its value to YES.

To check and use the snapped status:

- **Access snapping information** — Use the `@Environment(\.surfaceSnappingInfo)` property wrapper within your SwiftUI view to observe the scene's snapping state.
- **Check snapped state** — Read the `isSnapped` Boolean property from the environment value (`snappingInfo.isSnapped`) to determine if the system has currently snapped the scene to a physical surface.
- **Verify authorization** — Confirm the person has granted permission by checking that the `authorizationStatus` property (`snappingInfo.authorizationStatus`) equals

`.authorized`.

- **Retrieve the surface type** — If the scene is snapped and authorization is granted, access the `classification` property (`snappingInfo.classification`). This returns an ARKit `SurfaceClassification` value, such as `.wall`, `.floor`, or `.table`.
- **Conditionally show content** — Use the retrieved surface classification to dynamically show content based on the detected surface type.

```
import SwiftUI
import ARKit

struct ContentView: View {
    @Environment(\.surfaceSnappingInfo) private var snappedStatus
    @State private var showWhiteboard = false


    var body: some View {
        Group {
            // If it's snapped to the wall, show a whiteboard to-do list.
            if showWhiteboard {
                 WhiteboardView()
            } else {
                // Otherwise, show a paper to-do list.
                PaperView()
            }
        }
        .onChange(of: snappedStatus) {
            if snappedStatus.isSnapped {
                switch SurfaceSnappingInfo.authorizationStatus {
                case .authorized:
                    switch snappedStatus.classification {
                    case .some(.table):
                        showWhiteboard = true
                    default:
                        showWhiteboard = false
                    }
                case .denied:
                    // The authorization prompt was denied.
                    print("Authorization denied.")
                case .notDetermined:
                    // The authorization prompt has not been answered.
                    print("Please accept the authorization prompt.")
```

```
                case .restricted:
                    print("Access to surface snapping is restricted.")
                @unknown default:
                    print("Please accept the authorization prompt.")
                }
            } else {
                showWhiteboard = false
            }
        }
    }
}
```

See Also

SwiftUI

{} Canyon Crosser: Building a volumetric hike-planning app

Create a hike planning app using SwiftUI and RealityKit.

{} Hello World

Use windows, volumes, and immersive spaces to teach people about the Earth.

Presenting windows and spaces

Open and close the scenes that make up your app's interface.

Positioning and sizing windows

Influence the initial geometry of windows that your app presents.