Sample Code

Canyon Crosser: Building a volumetric hike-planning app

Create a hike planning app using SwiftUI and RealityKit.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

Canyon Crosser is a volumetric app that allows people to plan a hike through a historic trail in Grand Canyon National Park. As hiking in the desert revolves around temperature, the app allows people to see the predicted temperatures and plan rest stops and departure time accordingly.

Canyon Crosser shows off a number of RealityKit and visionOS features, including spatial layout, RealityKit and SwiftUI interoperability, and dynamic bounds extensions for volumes.

Play

Canyon Crosser uses RealityKit to present 3D models and SwiftUI to present standard user interface elements like buttons and pickers. The app uses Reality Composer Pro to manage the models and lighting effects.

Canyon Crosser demonstrates the use of several visionOS APIs:

- **Spatial Layout** — Demonstrates techniques for positioning 2D and 3D content in a volume.
- **Framework Interoperability** — Illustrates how to add SwiftUI views to entities in a `RealityView`.
- **Observable Entities** — Shows how to dynamically update and observe changes on RealityKit entities.
- **Dynamic Bounds Restriction** — Shows how to render content outside the bounds of the volumetric window.

Use spatial layout to create a carousel

After launching Canyon Crosser, the app displays a volumetric window featuring a carousel of landmarks. This carousel allows people to select a virtual hiking destination. While all landmarks are tappable in this example, they currently all lead to the Grand Canyon experience.

Play ⊙

Canyon Crosser takes advantage of spatial layout to implement the carousel by using a combination of depth layouts and 3D rotations. To position the landmark models and the carousel base at the bottom of the volume, it uses a `VStackLayout` with a depth alignment of `.center` and a `Spacer`.

The app uses a custom `Layout` implementation, named `RadialLayout`, to arrange the individual landmark items in a circle. To present the carousel horizontally, the entire `RadialLayout` is rotated 90 degrees around the x-axis using

```
.rotation3DLayout(Rotation3D(angle: .degrees(90), axis: .x))
```

Because the models initially load upright, they counter-rotate with `rotation3DLayout(_:)` by -90 degrees to compensate.

Canyon Crosser uses the following to create the carousel:

```
VStackLayout(spacing: 0).depthAlignment(.center) {
    // Pushes the content to the baseplate of the volume.
    Spacer()
    // A radial custom layout.
    RadialLayout(angleOffset: angleOffset) {
        ForEach(Array(zip(0..., $carouselModel.items)), id: \.1.id) { (index, $item)
            // The view that contains each landmark.
            LandmarkItemView(item: item)
                // Set the opacity based on the z position of the item.
                // The closer to the front of the carousel, the more opaque it is.
                .opacity(1 - carouselModel.normalizedZPosition[index])

                // Rotate the item by -90 degrees over the x-axis to account for the
                .rotation3DLayout(Rotation3D(angle: .degrees(-90), axis: .x))

                .onGeometryChange3D(for: Rect3D.self) { proxy in
                    proxy.frame(in: .global)
                } action: { newValue in
                    localZPosition = backOfVolume.z - newValue.origin.z
                    item.zPosition = localZPosition
                }
        }
    }
    // Rotates the radial layout to be horizontal instead of vertical.
    .rotation3DLayout(Rotation3D(angle: .degrees(90), axis: .x))

    CarouselPlatter()
}
```

> **Note**
>
> For a deeper dive into spatial layouts, watch the WWDC25 session, Meet SwiftUI spatial layout. For more information on building custom layouts, watch the WWDC22 session, Compose custom layouts with SwiftUI.

The app uses a front depth alignment to place the label at the front of the carousel.

```
VStackLayout(spacing: 20).depthAlignment(.front) {
    // The rotational layout and bottom platter.
   CarouselBodyView(angleOffset: angleOffset, backOfVolume: backOfVolume)
        .environment(carouselModel)
```

```
    CarouselLabelView()
        .environment(carouselModel)
}
```

Present SwiftUI views from a volume

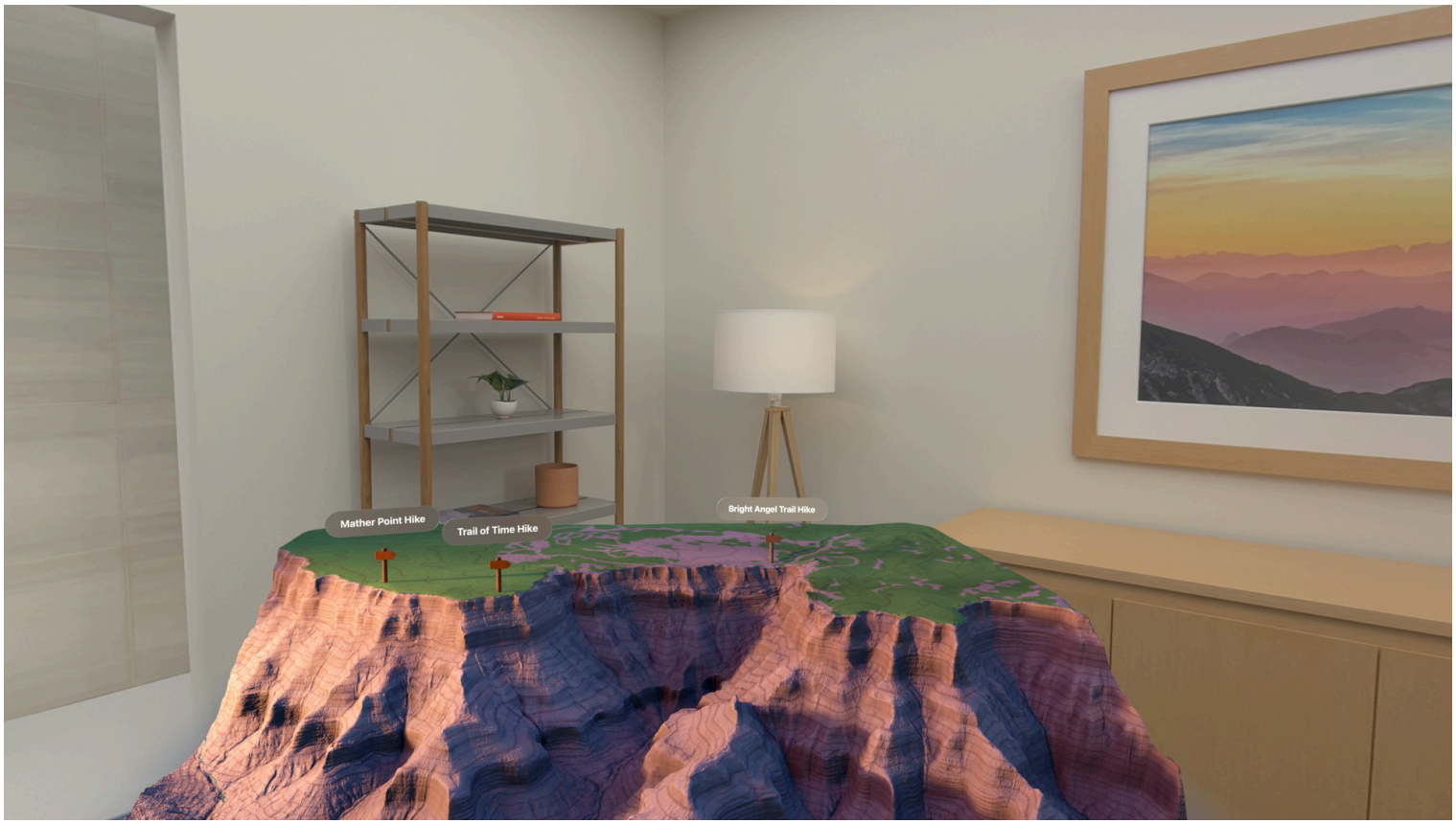
After selecting a landmark, three named trailhead markers appear for the Grand Canyon. The trailhead names appear above the marker at the top of the trail. Each trailhead has a sub-entity that has a `ViewAttachmentComponent`. The `ViewAttachmentComponent` attaches SwiftUI views to entities.

It often makes sense to present views directly from an entity. However, sometimes it may be better to offset the presentation of views from the presenting entity. Canyon Crosser uses a descendant entity to present the view from the desired location:

```
private func addBillboardingPositioningEntity(
    for viewComponent: Component,
    offset: SIMD3<Float>,
    relativeTo entity: Entity
) -> Entity {
    let positioningEntity = Entity()
    positioningEntity.position = offset
    positioningEntity.name = entity.name + ".positioningEntity"
    positioningEntity.components.set([viewComponent, BillboardComponent()])
    entity.addChild(positioningEntity)
    return positioningEntity
}
```

`ViewComponentAttachment` also respects the scale of its parent entity. If the parent entity doesn't inherit a large scale factor this is the desired behavior. If the entity's hierarchy has a scale factor other than `1.0`, invert the scale factor to make sure the attachment is legible:

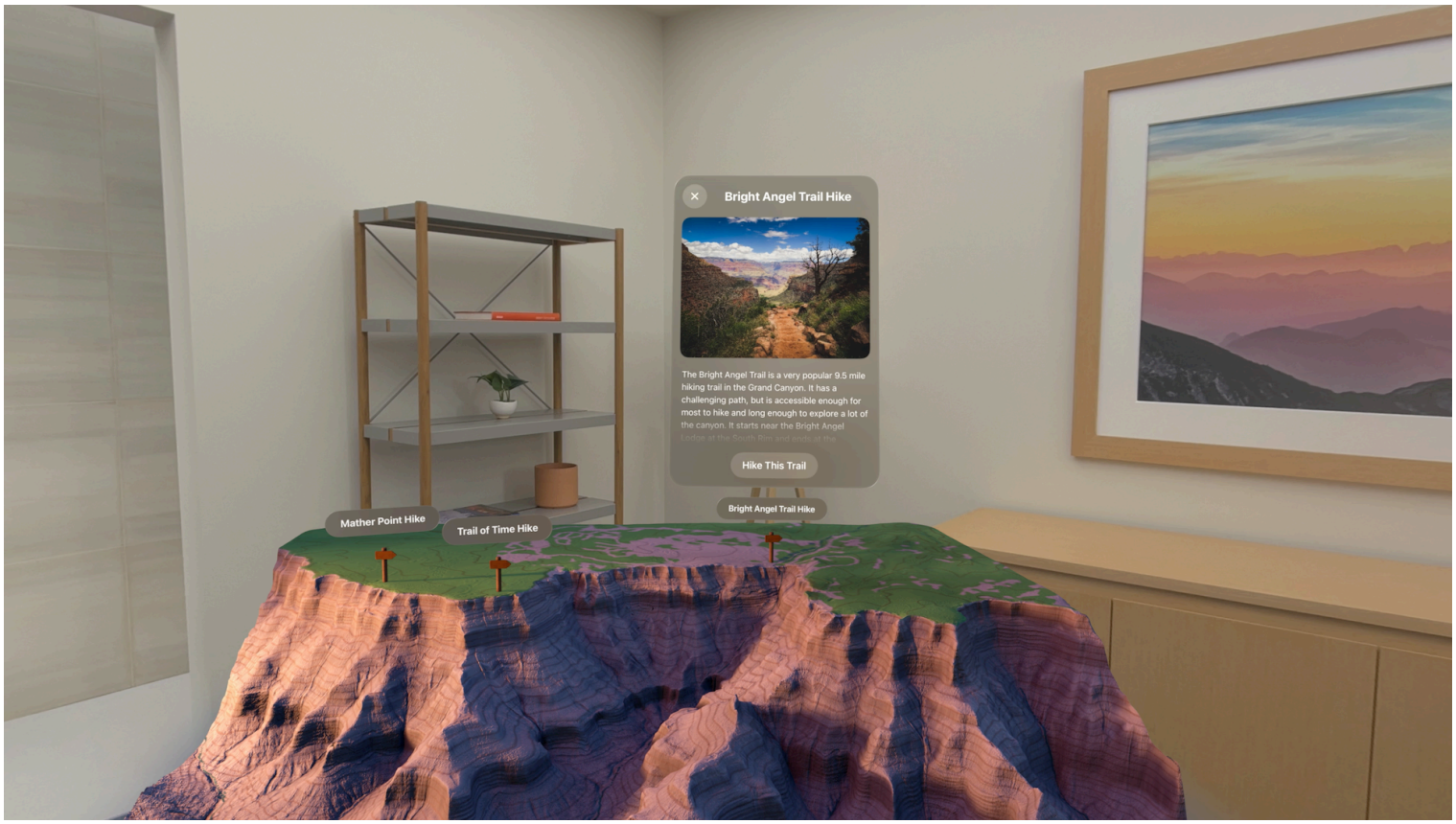
```
let trailheadNamePositioningEntity = addBillboardingPositioningEntity(
    for: trailNameAttachment,
    offset: SIMD3<Float>(x: 0, y: 1.1, z: 0),
    relativeTo: trailheadEntity
)
trailheadNamePositioningEntity.scale = 1.0 / trailheadEntity.scale(relativeTo: nil)
```



Use `PresentationComponent` to present a SwiftUI view as a popover. The component has an `isPresented` property that controls the popover's visibility. The popover presents modally, similar to popovers on other platforms.

```
var trailHeadPopover = PresentationComponent(
    configuration: .popover(arrowEdge: .bottom),
    content: HikeDetailView(hike: hike)
)
trailHeadPopover.isPresented = false

let trailheadPopoverPositioningEntity = addBillboardingPositioningEntity(
    for: trailHeadPopover,
    offset: SIMD3<Float>(0.0, 1.4, 0.0),
    relativeTo: trailheadEntity
)
```



> **Note**
>
> There's no need to apply an inverse scale to this entity because popovers use the dynamic scaling built into visionOS.

Choosing "Hike This Trail" activates the UI to allow people to interact with the hike. Each hike consists of the route for that hike and one entity for each rest stop along the trail. After selecting a hike, the entities activate. Conversely, the entities deactivate when the person deselects a hike. After selecting a hike, Canyon Crosser displays a toolbar that includes elements for modifying the hike. For example, Canyon Crosser lets people set the hike's start and stop time, and they can drag a slider to locations along the hike's path.

Use observable entities and components for dynamic updates

SwiftUI views can react to changes in components attached to an `Observable` entity to drive dynamic updates throughout the app.

> **Note**
>
> For a comprehensive introduction to observable entities, watch Better together: SwiftUI and RealityKit.

Canyon Crosser observes the hiker entity to keep several SwiftUI views up to date. There are four key components on the entity that update different parts of the UI:

- `HikeTimingComponent` — Manages hike speed, arrival, and departure times.
- `HikerProgressComponent` — Tracks the hiker's progress along the trail.
- `HikeDragStateComponent` — Controls the hiker's drag state during user interaction.
- `HikePlaybackStateComponent` — Manages the hike's playback state for playing and pausing a hike.

The `HikerProgressComponent` is central to updating many parts of the app as the hiker moves. Whenever a property within this component changes, all views observing it are automatically updated. For example, the `HikeProgress` system updates the hiker's progress:

```
progressComponent.hikeProgress = min(1.0, progressComponent.hikeProgress + Float(de
```

This modification to `hikeProgress` triggers updates in any view observing this component. The `SliderThumb` view, for example, observes this property to update its position along the slider track and adjust its visual effects based on the hiker's location.

The `GrandCanyonView` also observes the `hikeProgress` value. As the observed value changes, RealityKit notifies the `update:` block on the contained `RealityView` of the change and updates the entities related to the lighting of the canyon:

```
setSunlight(
    for: calculateTimeOfDay(from: appModel.hikerProgressComponent.hikeProgress),
    shouldAnimateChange: appModel.shouldAnimateSunlightChange
)
```

Using an observable entity property in the `update:` block registers the `RealityView` to update when the property changes. The implementation of `appModel.hikerProgressComponent` enables observation on the entity:

```
var hikerProgressComponent: HikerProgressComponent {
    get {
        guard
            let component = hikerEntity.observable.components[HikerProgressComponent
        else { fatalError() }

        return component
    }
    set { hikerEntity.observable.components[HikerProgressComponent.self] = newValue
```

```
}
```

The getter uses the `observable` property on the entity so any reference to this property registers the observation. Canyon Crosser relies on observable entities to synchronize the hiker's movement with various aspects of the app, including UI updates and environmental changes.

> **Important**
>
> Modifying any variable within an observable component causes the entire component to update. Carefully consider the scope of your components to optimize performance. Also avoid observing and modifying the same variable within a component in the same view to prevent infinite loops.

The best places to modify observable properties are in a custom system, a gesture closure, or the `make:` closure of your `RealityView`. Avoid modifying observable properties in view bodies or the `update:` closure. The `make:` closure of a RealityView differs from the `update:` closure in that it doesn't create a dependency when you access an observable property and doesn't re-run on changes to the property.

Request additional margins for drawing beyond the volume's bounds

Use the `preferredWindowClippingMargins(_:_:)` view modifier to request additional margins for drawing beyond the bounds of the window. This modifies the system behavior allowing you to display content that extends beyond the volume's bounds.

Canyon Crosser uses this API to request additional space at the leading and trailing edges to draw the clouds as they come into the volume. To read the value of the window clipping margins, use the `windowClippingMargins` environment variable. Monitor for changes of this environment value to update the app as needed:

```
.onChange(of: windowClippingMargins) { _, newValue in
    appModel.clippingMarginEnvironment.clippingMargins = physicalMetrics.convert(edg
}
```

In Canyon Crosser, the clipping margins are saved to the `ClippingMarginPercentage Component`. The `FeatherSystem` reads these values and applies them to all entities that have a

`FadingCloudComponent`. The system provides these properties to a shader graph material that uses them to fade out the clouds as the reach the edge of the clipping margins.

> **Note**
>
> For more details, watch the WWDC25 session, Set the scene with SwiftUI in visionOS.

Related videos

Better together: SwiftUI and RealityKit

Compose custom layouts with SwiftUI

Meet SwiftUI spatial layout

Set the scene with SwiftUI in visionOS

See Also

SwiftUI

- {} 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.

Adopting best practices for persistent UI

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