Sample Code

Swift Splash

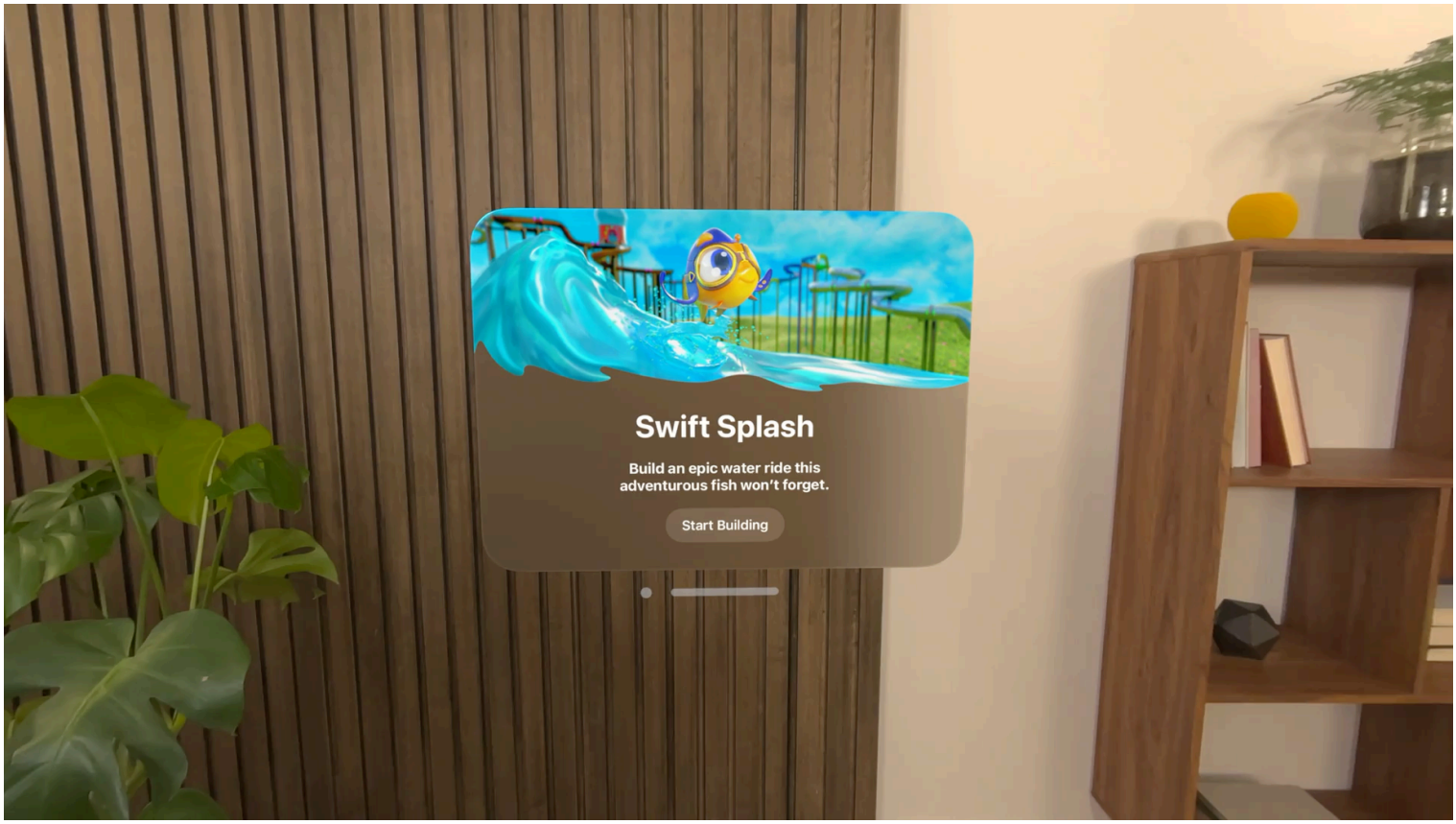
Use RealityKit to create an interactive ride in visionOS.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

Apple Vision Pro's ability to combine virtual content seamlessly with the real world allows for many kinds of interactive virtual experiences. Swift Splash leverages RealityKit and Reality Composer Pro to create a virtual water slide by combining modular slide pieces. When the builder finishes their ride, they can release an adventurous goldfish to try it out.



Play

Swift Splash uses multiple Reality Composer Scenes to create prepackaged entity hierarchies that represent each of the slide pieces the player connects to construct their ride. It demonstrates how to hide and reveal sections of the entity hierarchy based on the current state of the app. For example, each slide piece contains an animated fish entity that's hidden until the ride runs and the fish arrives at that particular piece. While Swift Splash is a fun, game-like experience, the core idea of assembling virtual objects out of predefined parts can also be used as the basis for a productivity or creation app.

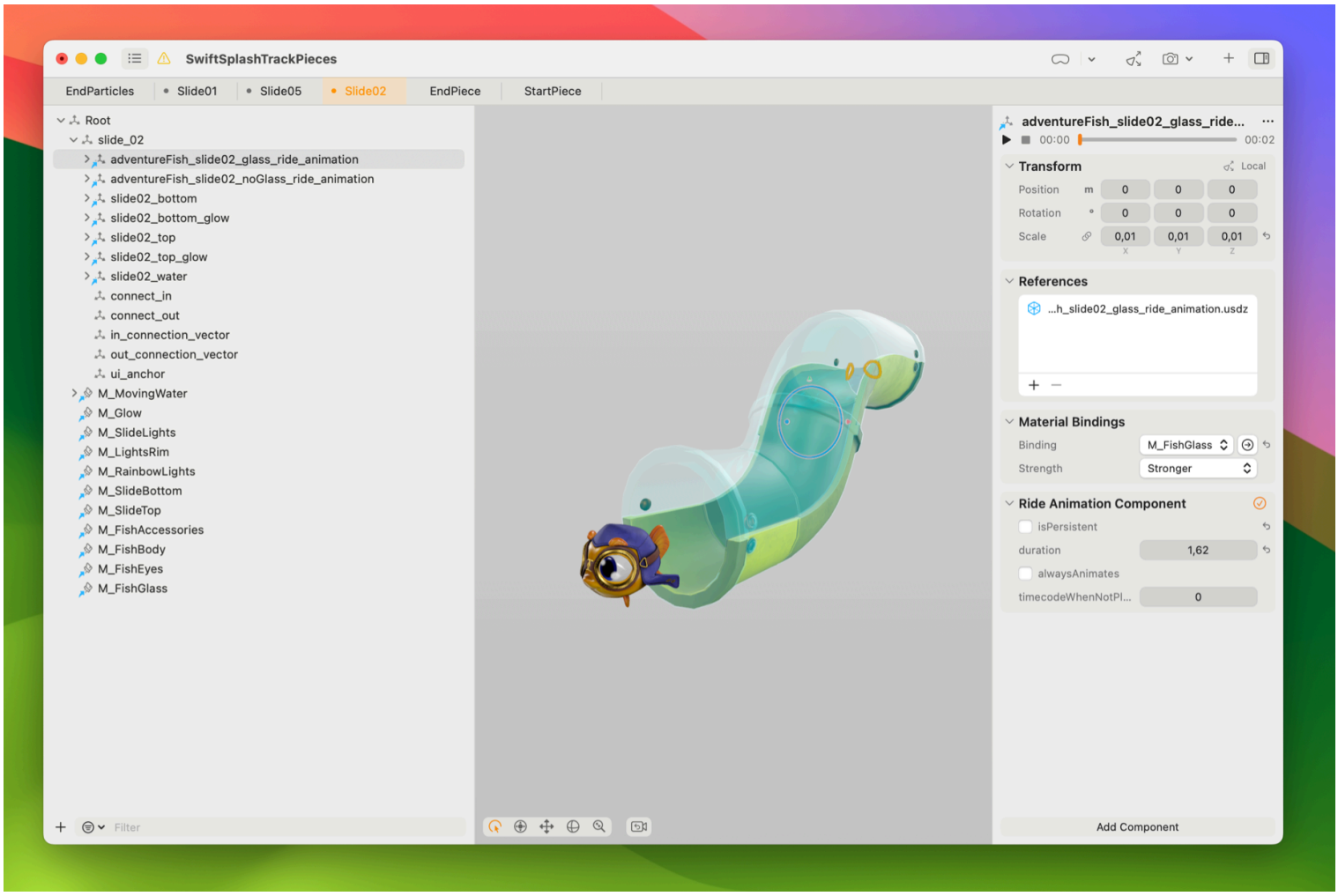
Swift Splash scenes include Shader Graph materials built in Reality Composer Pro to change the appearance of the ride at runtime. Each piece can be configured to display in one of three materials: metal, wood, or plastic. Other Shader Graph materials create special effects, such as the movement of the water and the flashing lights on the start and end pieces. Even particle effects are included in some of these prepackaged entities, such as the fireworks that play when the goldfish crosses the finish line.

Play

Play

Build slide pieces in Reality Composer Pro

Slide pieces are the building blocks of Swift Splash. The Reality Composer project contains a separate scene for each one. In addition to the 3D models that make up the slide piece, each scene contains a number of other entities the app uses to animate and place the slide piece.



In the hierarchy viewer on the left side of the screenshot above, there are two transform entities called `connect_in` and `connect_out`. These transforms mark the points where the slide piece connects to the next or previous piece. Swift Splash uses these transforms to place new pieces at the end of the existing slide, as well as to snap pieces to other slide pieces when you manually move them near each other.

Slide pieces demonstrate the two primary mechanisms Swift Splash uses to find entities at runtime. For some entities, such as `connect_in`, Swift Splash uses a naming convention and retrieves the entities by name or suffix when it needs to use them. In other cases, such as when names aren't unique or the retrieving code needs configuration values, Swift Splash uses a custom component to mark and retrieve entities.

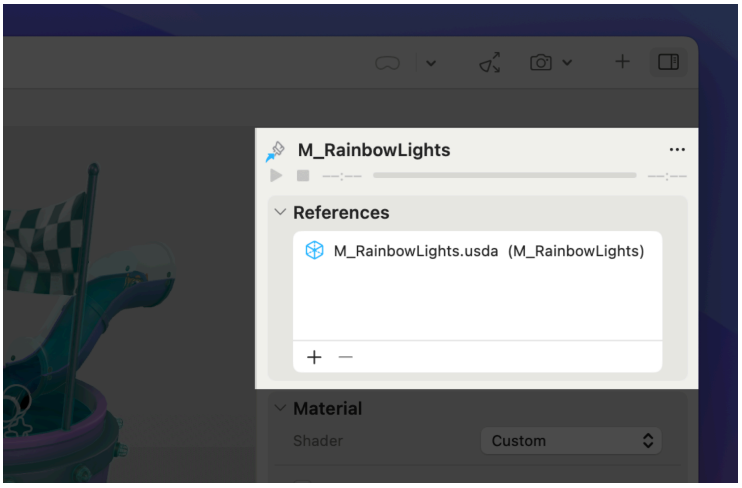
For example, animated entities that appear when the ride runs contain a component called `RideAnimationComponent`. The app uses this component to determine if the entity is an animation that plays while the ride is running. The component also stores additional state the app needs to implement the ride animation, such as a property called `duration` that specifies when to start the animations on the next connected slide piece.

`RideAnimationComponent` also includes a property called `isPersistent`. Persistent ride animations stay visible at all times but only animate when the ride is running, such as the animated door on the start piece. Nonpersistent ride animations, such as the fish swimming through a slide piece, display only while the ride is running and the fish swims through that particular piece.

Avoid duplicate materials with material references

Many of Swift Splash's slide pieces use the same materials. For example, the shader graph material that changes pieces from metal to wood to plastic is shared by all but one of the slide pieces. To avoid having duplicate copies of each material, Swift Splash leverages USD *material references* to share materials between multiple entities in multiple scenes.

The Reality Composer Pro project contains a separate scene for each shared material, containing only that one material. Other track pieces create references to that material. If you change the original material, it affects all of the entities that reference it. For example, a scene called `M_RainbowLights.usda` contains the material `M_RainbowLights`, and both `StartPiece.usda` and `EndPiece.usda` reference that material.



Parallelize the asset load

To maximize load speed and make the most efficient use of available compute resources, Swift Splash parallelizes loading scenes from the Reality Composer project using a `TaskGroup`. The app creates a separate `Task` for each of the scenes it needs to load.

```
await withTaskGroup(of: LoadResult.self) { taskGroup in
    // Load the regular slide pieces and ride animations.
    logger.info("Loading slide pieces.")
    for piece in pieces {
        taskGroup.addTask {
            do {
                guard let pieceEntity = try await self.loadFromRCPro(named: piece.ke
                                                                     fromSceneNamed: piece.sceneNa
                    fatalError("Attempted to load piece entity \(piece.name) but fai
                }
                return LoadResult(entity: pieceEntity, key: piece.key.rawValue)
            } catch {
                fatalError("Attempted to load \(piece.name) but failed: \(error.loca
            }
        }
    }
    // Continue adding asset load jobs.
    // ...
```

```
}
```

The app then uses an async iterator to wait for and receive the results.

```
for await result in taskGroup {
    if let pieceKey = pieces.filter({ piece in
        piece.key.rawValue == result.key
    }).first {
        self.add(template: result.entity, for: pieceKey.key)
        setupConnectible(entity: result.entity)
        result.entity.generateCollisionShapes(recursive: true)
        result.entity.setUpAnimationVisibility()
    }
    // ...
}
```

For more information on task groups, see Concurrency in The Swift Programming Language.

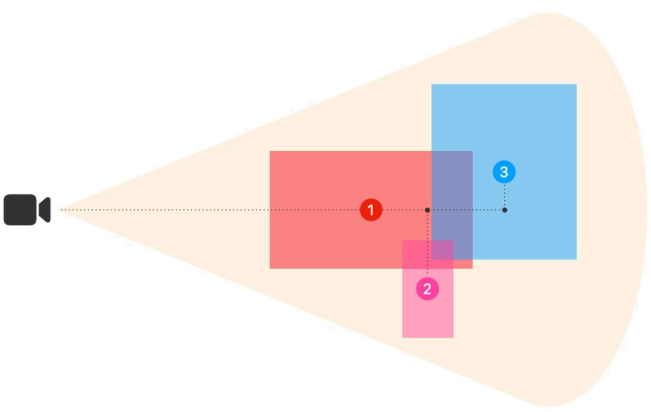
Each of these loaded pieces acts as a template. When the player adds a new piece of that type, the app clones the piece loaded from Reality Composer Pro and adds the clone to the scene.

Specify sort ordering for transparent entities

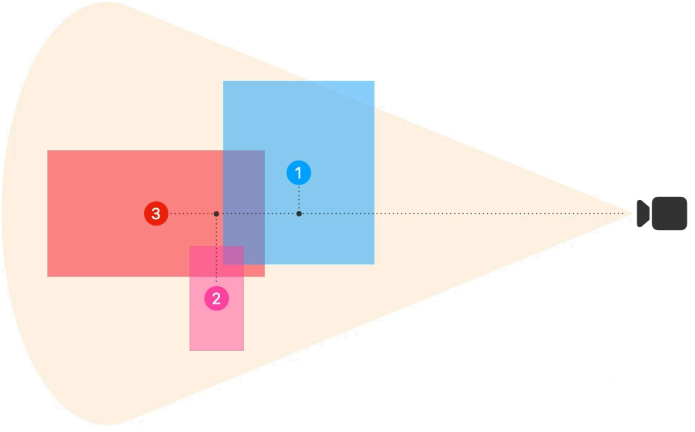
When multiple entities have more than one overlapping, nonopaque material, RealityKit's default depth-sorting can cause it to draw those entities in the wrong order. As a result, some entities may not be visible from certain angles or in certain positions relative to other transparent entities. The default depth sorting is based on the center of the entity's bounding box, which may result in the incorrect drawing order when there are multiple overlapping materials with any amount of transparency. You can see an example of this by looking at the start piece in Reality Composer Pro, or by watching the video below.

Play

The following video demonstrates the problem. If the three boxes are the bounding boxes for three different transparent entities, and the small spheres are the box centers, the sphere that's closest to the camera changes as the camera moves around the boxes, which changes the order that RealityKit's default depth sorting algorithm draws them.



Play



Play

Swift Splash assigns a `ModelSortGroupComponent` to each of the transparent entities to manually specify the relative depth sorting. To fix the transparency issues in the start piece in the

video above, Swift Splash instructs RealityKit to draw the opaque parts of the fish first, its transparent goggles second, the water third, the glass globe fourth, and the selection glow shell last. Swift Splash does this by assigning a `ModelSortGroupComponent` to each of the overlapping entities using the same `ModelSortGroup`, but with a different order specified.

```
fileprivate func setEntityDrawOrder(_ entity: Entity, _ sortOrder: Int32, _ sortGrou
    entity.forEachDescendant(withComponent: ModelComponent.self) { modelEntity, mode
        logger.info("Setting sort order of \(sortOrder) of \(entity.name), child ent
        let component = ModelSortGroupComponent(group: sortGroup, order: sortOrder)
        modelEntity.components.set(component)
    }
}

/// Manually specify sort ordering for the transparent start piece meshes.
func handleStartPieceTransparency(_ startPiece: Entity) {
    let group = ModelSortGroup()

    // Opaque fish parts.
    if let entity = startPiece.findEntity(named: fishIdleAnimModelName) {
        setEntityDrawOrder(entity, 1, group)
    }
    if let entity = startPiece.findEntity(named: fishRideAnimModelName) {
        setEntityDrawOrder(entity, 2, group)
    }

    // Transparent fish parts.
    if let entity = startPiece.findEntity(named: fishGlassIdleAnimModelName) {
        setEntityDrawOrder(entity, 3, group)
    }
    if let entity = startPiece.findEntity(named: fishGlassRideAnimModelName) {
        setEntityDrawOrder(entity, 4, group)
    }

    // Water.
    if let entity = startPiece.findEntity(named: sortOrderWaterName) {
        setEntityDrawOrder(entity, 5, group)
    }

    // Glass globe.
    if let entity = startPiece.findEntity(named: sortOrderGlassGlobeName) {
        setEntityDrawOrder(entity, 6, group)
    }
```

```
    // Selection glow.
    if let entity = startPiece.findEntity(named: startGlowName) {
        setEntityDrawOrder(entity, 7, group)
    }
}
```

Traverse connected track pieces

The root entity for all of the individual slide pieces has a `ConnectableComponent`. This custom component marks the entity as one that can be connected or snapped to other connectable entities. At runtime, the app adds a `ConnectableStateComponent` to each slide piece it adds. The component stores state information for the track piece that doesn't need to be edited in Reality Composer Pro. Among the state information that this component stores is a reference to the next and previous piece.

Play ⊙

To iterate through the entire ride, ignoring any disconnected pieces, the app gets a reference to the start piece and then iterates until `nextPiece` is `nil`. This iteration, similar to iterating a linked list, repeats many times throughout the app. One example is the function that calculates the duration of the built ride by iterating through the individual pieces and adding up the duration of their animations.

```
/// Calculates the duration of the built ride by summing up the individual durations
public func calculateRideDuration() {
    guard let startPiece = startPiece else { fatalError("No start piece found.") }
    var nextPiece: Entity? = startPiece
    var duration: TimeInterval = 0
    while nextPiece != nil {
        // Some pieces have more than one ride animation. Use the longest one to cal
        var longestAnimation: TimeInterval = 0
        nextPiece?.forEachDescendant(withComponent: RideAnimationComponent.self) { e
            longestAnimation = max(component.duration, longestAnimation)
        }
        duration += longestAnimation
        nextPiece = nextPiece?.connectableStateComponent?.nextPiece
    }
    // Remove the part of the animation after the goal post.
    rideDuration = duration  / animationSpeedMultiplier + 1.0
}
```

Interact with the ride

To build and edit the ride, players interact with Swift Splash in two different ways. They interact with SwiftUI windows to perform certain tasks, such as adding a new piece or deleting an existing piece of the ride. They also manipulate slide pieces using standard visionOS gestures, including taps, double taps, drags, and rotates. The player taps on a piece to select or deselect it. When a player double taps a piece, they select that piece without deselecting any other selected pieces. When someone drags a piece, it moves around the immersive space, snapping together with other pieces if placed near one. A two-finger rotate gesture spins the selected track piece or pieces on the Z-axis.

Play ▷

Add a ride piece

Play ▷

Delete a ride piece

Play ⊙

Rotate a ride piece

Play ⊙

Drag a ride piece

Swift Splash handles all of these interactions using standard SwiftUI gestures targeted to an entity. To support any of these gestures at any time, the app declares them using `Simultaneous Gesture`. The code for all of the gestures are contained in `TrackBuildingView`, which controls the app's immersive space. Here's how the app defines the rotation gesture:

```
.simultaneousGesture(
    RotateGesture()
        .targetedToAnyEntity()
        .onChanged({ value in
            guard appState.phase == .buildingTrack || appState.phase == .placingStar
            handleRotationChanged(value)
        })
        .onEnded({ value in
            guard appState.phase == .buildingTrack || appState.phase == .placingStar
            handleRotationChanged(value, isEnded: true)
        })
)
```

Because multiple tap gestures on the same `RealityView` execute with a different number of taps, multiple gestures may be called at once. If a player double taps an entity, for example, both the single tap and the double tap gesture code get called, and the app has to determine which one to execute. Swift Splash makes this determination by using a Boolean state variable. If a player single taps, it sets that variable — called `shouldSingleTap` — to `true`. Then it waits for a period of time before executing the rest of its code. If `shouldSingleTap` gets set to `false` while it's waiting, the code doesn't execute. When SwiftSplash detects a double tap gesture, it sets `shouldSingleTap` to `false`, preventing the single-tap code from firing when it executes the double-tap code.

```
.simultaneousGesture(
    TapGesture()
```

```
        .targetedToAnyEntity()
        .onEnded({ value in
            guard appState.phase == .buildingTrack else { return }
            Task {
                shouldSingleTap = true
                try? await Task.sleep(for: .seconds(doubleTapTolerance))
                if shouldSingleTap {
```

See Also

Related samples

- Hello World
  Use windows, volumes, and immersive spaces to teach people about the Earth.
- Happy Beam
  Leverage a Full Space to create a fun game using ARKit.
- Destination Video
  Leverage SwiftUI to build an immersive media experience in a multiplatform app.
- Diorama
  Design scenes for your visionOS app using Reality Composer Pro.

Related articles

- Adding 3D content to your app
  Add depth and dimension to your visionOS app and discover how to incorporate your app's content into a person's surroundings.
- Understanding the modular architecture of RealityKit
  Learn how everything fits together in RealityKit.
- Composing interactive 3D content with RealityKit and Reality Composer Pro
  Build an interactive scene using an animation timeline.
- Implementing systems for entities in a scene
  Apply behaviors and physical effects to the objects and characters in a RealityKit scene with the Entity Component System (ECS).

Creating USD files for Apple devices

Generate 3D assets that render as expected.

Related videos

Meet Reality Composer Pro

Explore materials in Reality Composer Pro

Work with Reality Composer Pro content in Xcode

Build spatial experiences with RealityKit

Enhance your spatial computing app with RealityKit

Build great games for spatial computing