Sample Code

Bringing your SceneKit projects to RealityKit

Adapt a platformer game for RealityKit's powerful ECS and modularity.

Download

iOS 26.0+ | iPadOS 26.0+ | macOS 26.0+ | tvOS 26.0+ | visionOS 26.0+ | Xcode 26.0+

Overview

This sample project demonstrates a cross-platform 3D platformer game implemented in two ways: one using the SceneKit framework, and another built with the RealityKit framework. By seeing both versions side by side, you can directly compare implementation approaches and understand the migration path from SceneKit to RealityKit.

The RealityKit version showcases modern Swift development practices, including:

- Modular architecture through Swift packages
- Entity-Component-System (ECS) design patterns
- Cross-platform compatibility, including specialized features for visionOS
- SwiftUI integration for user interface elements
- Input handling systems for multiple device types

Play ⊙

Whether you're maintaining an existing SceneKit project or planning a new 3D experience, this sample demonstrates how RealityKit enables more maintainable and performant 3D applications across Apple platforms.

> Note
>
> This article and sample code project are associated with WWDC25 session 288: Bring your SceneKit projects to RealityKit.

Understand SceneKit deprecation

SceneKit is soft-deprecated. This means that your existing apps and games continue to work, and you aren't required to take immediate action.

But the implications of any deprecation, however soft, are still significant:

- SceneKit is unlikely to gain new capabilities, integrations, or API paradigms.
- SceneKit's critical security or stability issues might be addressed, but general bug fixes or performance optimizations are unlikely.

For any project with a long-term vision, or for those looking to harness the latest innovations, migrate to another native framework like RealityKit.

Design modular Swift packages

The core architectural strength of RealityKit comes from its modular design, which you can leverage through Swift packages. Each package encapsulates specific functionality, making the codebase more maintainable, testable, and reusable. This approach aligns perfectly with RealityKit's ECS architecture:

- **Entities** are lightweight containers.
- **Components** hold pure data, defining properties or state.
- **Systems** encapsulate all the logic, operating on entities with specific components.

Explore key packages in the sample project

The sample project demonstrates how to organize RealityKit functionality into cohesive, reusable Swift packages:

- **AgentComponent** moves entities according to agency goals and constraints, enabling AI-driven behaviors.
- **CharacterMovement** updates playable entities with transform updates and animation playback.
- **ControllerInput** handles and delivers controller input to RealityKit entities.
- **WASDInput** processes keyboard input for RealityKit entities.
- **WorldCamera** manages the scene's active camera and handles repositioning for portals or immersive spaces.
- **PyroPanda** contains all Reality Composer Pro content for the game.
- **ThumbStickView** provides UI components for touchscreen thumbstick control.

This modular approach enables clean separation of concerns, promotes code reusability, and makes complex systems more understandable and maintainable.

Implement character movement and input handling

The `CharacterMovement` package provides a flexible system for controlling character movement in 3D space, supporting multiple input methods across different platforms. This package simplifies one of the most complex aspects of migrating from SceneKit: implementing responsive character controls.

At its core is the `CharacterMovementComponent`, which stores motion data and input state:

```swift
// Create a basic movement component for a character.
var moveComponent = CharacterMovementComponent(characterProxy: "Max")

// Add the component to your character entity.
character.components.set(moveComponent)
```

The component handles multiple input sources simultaneously, combining them for smooth control:

```swift
// Example: Processing touch input from a virtual thumbstick.
ThumbStickView(updatingValue: $movementThumbstick)
    .onChange(of: movementThumbstick) { _, newValue in
        // Convert 2D input to 3D movement direction.
        let movementVector: SIMD3<Float> = [Float(newValue.x), 0, Float(newValue.y)]
        character.components[CharacterMovementComponent.self]?.controllerDirection =
    }

// Example: Processing keyboard input.
func processKeyboardInput(wasd: SIMD2<Float>) {
    // Convert WASD input to 3D movement direction.
    character.components[CharacterMovementComponent.self]?.wasdDirection = [wasd.x,
}
```

The package also handles character state management, transitioning between animations based on movement:

```swift
// Define animations for different character states.
var animations = [CharacterStateComponent.CharacterState: AnimationResource]()
animations[.idle] = idleAnimation.repeat()
animations[.walking] = walkAnimation.repeat()
animations[.jump] = jumpAnimation.combineWithAudio(named: "jump")

// Create and add the state component to your character.
let stateComponent = CharacterStateComponent(animations: animations)
character.components.set(stateComponent)

// Trigger actions through the movement component.
character.components[CharacterMovementComponent.self]?.jumpPressed = true
```

This package enables the main character, Max, to move around the scene, and animate as he does so.

Play ⊙

Configure camera management

With most platforms, you can add a camera component, such as `PerspectiveCamera Component`, to an entity to change the perspective, but this isn't possible on visionOS because the person wearing an Apple Vision Pro is the camera. To get around this factor, the app instead moves the world and physics simulation root entities around the wearer.

The `WorldCamera` package provides a component-based solution for positioning a camera around an entity of interest in 3D space, particularly useful for third-person games, while accounting for platform differences.

The `WorldCameraComponent` manages spherical coordinate positioning around a target:

```
// Create a simple world camera component to follow a character.
var worldCameraComponent = WorldCameraComponent(
    azimuth: .pi,     // Position behind the character.
    elevation: 0.2,   // Slightly above the character.
    radius: 3         // Position 3 meters away from the character.
)
```

To avoid jarring behaviors, the system can also implement soft camera movements by having the camera follow a target. The `WorldCamera` package provides a `FollowComponent` to perform smooth transitions as the target moves:

```
// Apply a softer movement in the y-axis, avoiding sharp movements when the characte
let followSmoothing: SIMD3<Float> = [3, 1.2, 3]

// Create a camera entity with both components.
let worldCamera = Entity(components:
    worldCameraComponent,
    FollowComponent(targetId: target.id, smoothing: )
)

#if !os(visionOS)
// Add the actual camera component for rendering on non-visionOS platforms.
worldCamera.addChild(Entity(components: PerspectiveCameraComponent()))
#endif
```

This approach ensures that on platforms like iOS and macOS, the camera moves around the world, while on visionOS, the world moves around the wearer for a comfortable immersive experience that prevents motion sickness and maintains proper spatial awareness.

Play ⊙

Important

On visionOS, moving either the world or the camera in a fully immersive space can be disorienting for people. Use this technique primarily for experiences with finite bounds, such as volumes, portals, windows, or immersive spaces that aren't fully immersive. When implementing such camera or world movements, keep transitions smooth and predictable to maintain comfort.

Create enemy AI and agency

The `AgentComponent` package adds autonomous behavior to enemy entities, allowing them to chase or flee from the player character.

This example defines a patrol area for an enemy:

```
let wanderPathPoints: [SIMD3<Float>] = [
    [-1, -0.25, 13], [ 1, -0.25, 13],
    [-1, -0.25, 12], [ 1, -0.25, 12],
    [-1, -0.25, 11], [ 1, -0.25, 11],
    [-1, -0.25, 10], [ 1, -0.25, 10]
]
let stayOnPath = GKPath(points: wanderPathPoints, radius: 0.75, cyclical: true)
let centerGoal = GKGoal(toStayOn: stayOnPath, maxPredictionTime: 1)
let chasingType: AgentComponent.AgentType = .chasing(id: hero.id, distance: 3, speec
```

Next, the app creates an `AgentComponent` with behavior parameters and applies it to the entity:

```
let chasingComponent = AgentComponent(
    agentType: chasingType,
    wanderSpeed: 3, wanderGoal: GKGoal(toWander: 1),
    centerGoal: centerGoal,
    constraints: .position(y: .exact(-0.25))
)
chasing.components.set(chasingComponent)
```

Play ⊙

Convert SceneKit assets to USD

When migrating from SceneKit to RealityKit, asset conversion is a critical step. When building a RealityKit app, Universal Scene Description (USD) is the format of choice for 3D content across all Apple platforms.

Most digital content creation tools offer good support for USD, so the best choice is exporting your assets directly to USD, and importing them into your app or game. If you don't have access to the original asset files and only have assets in SCN file format, you can use `scntool` to convert your 3D models to USD.

You can invoke `scntool` by typing `xcrun scntool` into a terminal window:

To convert an SCN asset to USDZ, you can use:

```
xcrun scntool --convert max.scn --format usdz
```

If you have a separate SCN file, which contains animation data, you can append this too:

```
xcrun scntool --convert max.scn --format usdz --append-animation max_spin.scn
```

Implement Portrait mode for immersive spaces on visionOS

The RealityKit version of the sample demonstrates how to leverage visionOS-specific features to enhance immersive experiences, particularly through the use of immersive spaces with Portrait mode.

Portrait mode provides a grounded experience where users can still see their physical surroundings while engaging with immersive content, perfect for games like platformers.

Pyro Panda uses the new portrait aspect ratio for immersive spaces by first declaring a `ProgressiveImmersionStyle` with the aspect ratio `portrait`:

```
var immersionStyle: ProgressiveImmersionStyle {
    .progressive(
        0.1...0.5,
        initialAmount: 0.15,
        aspectRatio: .portrait
    )
}
```

The app assigns this immersion style to the `ImmersiveWindow`, to apply the portrait styling:

```
ImmersiveSpace(id: appModel.immersiveSpaceID) {
    // ...
}.immersionStyle(
    selection: .constant(self.immersionStyle),
    in: self.immersionStyle
)
```

Play

This implementation preserves the person's awareness of their physical environment when playing the game, while still making the experience immersive.

Handle hardware compatibility

RealityKit is built for modern hardware, with capabilities that may exceed what older devices can handle. The sample shows you how to gracefully handle hardware compatibility:

```
var supportsFullGame: Bool {
    // Check if the device supports Apple GPU Family 2 or later.
    return MTLCreateSystemDefaultDevice()?.supportsFamily(.apple2) ?? false
}
```

With this simple check, the app provides an appropriate experience based on hardware capabilities:

```
if supportsFullGame {
    RealityView { content in
        // Full game experience.
    }
} else {
    // Fallback experience.
    gameNotSupportedUI()
}
```

This approach ensures your RealityKit application performs well across all supported devices while maximizing modern hardware capabilities.

See Also

Sample code links

{} Creating a Spaceship game

Build an immersive game using RealityKit audio, simulation, and rendering features.

{} BOT-anist

Build a multiplatform app that uses windows, volumes, and animations to create a robot botanist's greenhouse.

{} Rendering a windowed game in stereo

Bring an iOS or iPadOS game to visionOS and enhance it.

{} Happy Beam

Leverage a Full Space to create a fun game using ARKit.

{} Swift Splash

Use RealityKit to create an interactive ride in visionOS.

{} Destination Video

Leverage SwiftUI to build an immersive media experience in a multiplatform app.

{} Creating a game with scene understanding

Create AR games and experiences that interact with real-world objects on LiDAR-equipped iOS devices.