Sample Code

Creating an immersive space in visionOS

Enhance your visionOS app by adding an immersive space using RealityKit.

Download

visionOS 2.0+ | Xcode 16.0+

Overview

This sample code project demonstrates how to create an `ImmersiveSpace` in visionOS, showcasing a collection of rocks orbiting around a person, like the scene in the following video:

Play

The `TurnTableSystem` runs the core functionality to manage the rotational dynamics of entities through the `TurnTableComponent`, which defines the speed and rotation axis. The `Entity` class extends with a method to populate the immersive space with 3D rock entities and then apply a random transformation to create a halo effect. The `ImmersiveView` integrates these features and showcases the potential of SwiftUI and RealityKit to build an exciting augmented reality experience.

Create the turntable system

The sample creates a custom system and corresponding component to apply a series of random rotations to the content in the immersive environment, resulting in a collection of objects that orbit around a person.

```swift
import SwiftUI
import RealityKit

struct TurnTableComponent: Component {
    /// The duration of the orbit effect.
    var time: TimeInterval = 0

    /// The speed at which the entity moves over time.
    var speed: TimeInterval

    /// The axis that the object orbits around.
    var axis: SIMD3<Float>

    /// Initialize the turntable component by setting the speed and axis variables.
    init(speed: TimeInterval = 1.0, axis: SIMD3<Float> = [0, 1, 0]) {
        self.speed = speed
        self.axis = axis
    }
}
```

The `TurnTableComponent` stores the time, speed, and axis information that decides how each of the entities rotates over time.

The `TurnTableSystem` checks the current scene for any rendered content that contains a `TurnTableComponent`. If it detects the component, the system applies the value of `deltaTime` to the component's `time` value and sets the orientation value to the component's `axis` variable.

```swift
import SwiftUI
import RealityKit

struct TurnTableSystem: System {
    /// A value to check whether the entity has the required component.
    static let query = EntityQuery(where: .has(TurnTableComponent.self))

    /// Initialize the system with the RealityKit scene.
    init(scene: RealityKit.Scene) { }

    // Update the entities to apply the time value and orientation.
    func update(context: SceneUpdateContext) {
        // Iterate over entities that match the query and are currently rendering.
        for entity in context.entities(matching: Self.query, updatingSystemWhen: .re
            /// The variable to get the `TurnTableComponent` from the entity.
            var comp = entity.components[TurnTableComponent.self]!

            // Set the time variable to increase as time passes.
            comp.time += context.deltaTime

            // Update the component in the entity.
            entity.components[TurnTableComponent.self] = comp

            // Adjust the orientation to update the angle, speed, and axis of rotati
            entity.setOrientation(simd_quatf(angle: Float(0.1 * comp.speed), axis: c
        }
    }
}
```

Extend the entity class

The sample extends the Entity class to load a file as a model component and apply the model for the halo effect.

```
import RealityKit

extension Entity {
    /// The name of the 3D model resource file.
    static var fileName: String = "rock"

    /// The model component loads a given filename.
    static var rockModel: ModelComponent = {
        let rock = try! Entity.loadModel(named: fileName)
        return rock.model!
    }()

    //...
}
```

To create a halo effect, the sample extends the `Entity` class with a custom method called `addHalo()`. The method loops through the total number of rocks that the app creates, and makes an array of entities with random size, speed, and rotation.

```swift
import RealityKit

extension Entity {
    // ...

    /// Assign a series of random rotations, translations, and scales for each rock
    func addHalo() {
        /// The number of models to generate.
        let modelCount: Int = 50

        // Create each model and apply a random transform, scale, and time interval.
        for _ in 0..<modelCount {
            /// The dedicated entity for the model.
            let entity = Entity()

            // Assign the model as a component.
            entity.components.set(Self.rockModel)

            /// A transform that contains a random rotation along the y-axis.
            let rotate0 = Transform(rotation: simd_quatf(angle: .random(in: 0...(2 *

            /// A transform that adjusts along the z-axis.
            let translate = Transform(translation: [0, 0, 1]).matrix

            /// A transform that contains a random rotation along the y-axis.
            let rotate1 = Transform(rotation: simd_quatf(angle: .random(in: 0...(2 *

            /// A transform that contains a random rotation along the x-axis.
            let rotate2 = Transform(rotation: simd_quatf(angle: .random(in: (-.pi /

            // Assign the entity a transform based on the matrix product of each rot
            entity.transform = Transform(matrix: rotate1 * rotate2 * translate * rot

            // Assign the entity a random scale along all axes.
            entity.setScale(SIMD3(repeating: 0.001 * .random(in: 0.5...2)), relative

            // Assign the entity's `TurnTableComponent` a random speed at which to r
            entity.components.set(TurnTableComponent(
                speed: .random(in: 0.001...0.1),
                axis: [
                    .random(in: -1...1),
                    .random(in: -1...1),
                    .random(in: -1...1)]
            ))

            // Add the entity as a subentity.
            self.addChild(entity)
        }
    }
}
```

The method constructs and returns a single entity that contains multiple subentities, each with a model that applies unique rotations and speed through the `TurnTableComponent`.

Create the immersive view

The sample defines a custom view for an immersive space that creates a single entity and applies the `addHalo()` method.

```
import SwiftUI
import RealityKit

struct ImmersiveView: View {
    /// The average human height in meters.
    let avgHeight: Float = 1.70

    /// The rate of movement at which the rocks orbit.
    let speed: TimeInterval = 0.03

    var body: some View {
        // Initiate a `RealityView` to create a ring
        // of rocks to orbit around a person.
        RealityView { content in
            /// The entity to contain the models.
            let rootEntity = Entity()

            // Set the y-axis position to the average human height.
            rootEntity.position.y += avgHeight

            // Create the halo effect with the `addHalo` method.
            rootEntity.addHalo()

            // Set the rotation speed for the rocks.
            rootEntity.components.set(TurnTableComponent(speed: speed))
            // Register the `TurnTableSystem` to handle the rotation logic.
            TurnTableSystem.registerSystem()

            // Add the entity to the view.
            content.add(rootEntity)
        }
    }
}
```

The `rootEntity` includes a `TurnTableComponent` to enable an orbit effect similar to the collective rotation of all the rock entities.

Run the immersive scene

The `Immersion` app structure includes an `ImmersiveSpace` entry to the scene to include in the app's environment.

```
import SwiftUI

@main
struct Immersion: App {
    var body: some Scene {
        WindowGroup {
            MainView()
        }

        // Defines an immersive space as a part of the scene.
        ImmersiveSpace(id: "ImmersiveScene") {
            ImmersiveView()
        }
    }
}
```

The sample's main view uses the `openImmersiveSpace` instance property to call the immersive space that the `Immersion` app structure defines.

```
import SwiftUI

struct MainView: View {
    /// The environment value to get the `OpenImmersiveSpaceAction` instance.
    @Environment(\.openImmersiveSpace) var openImmersiveSpace
    var body: some View {
        // Displays a line of text and
        // opens a new `ImmersiveSpace` environment.
        Text("Immersive Content Example")
            .onAppear {
                Task {
                    await openImmersiveSpace(id: "ImmersiveScene")
                }
            }
    }
}
```

See Also

Related samples

{} Combining 2D and 3D views in an immersive app

Use attachments to place 2D content relative to 3D content in your visionOS app.

{} Building an immersive experience with RealityKit

Use systems and postprocessing effects to create a realistic underwater scene.

{} Construct an immersive environment for visionOS

Build efficient custom worlds for your app.

{} Creating an interactive 3D model in visionOS

Display an interactive car model using gestures in a reality view.

Related articles

Understanding the modular architecture of RealityKit

Learn how everything fits together in RealityKit.

Creating fully immersive experiences in your app

Build fully immersive experiences by combining spaces with content you create using RealityKit or Metal.

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).