Sample Code

Petite Asteroids: Building a volumetric visionOS game

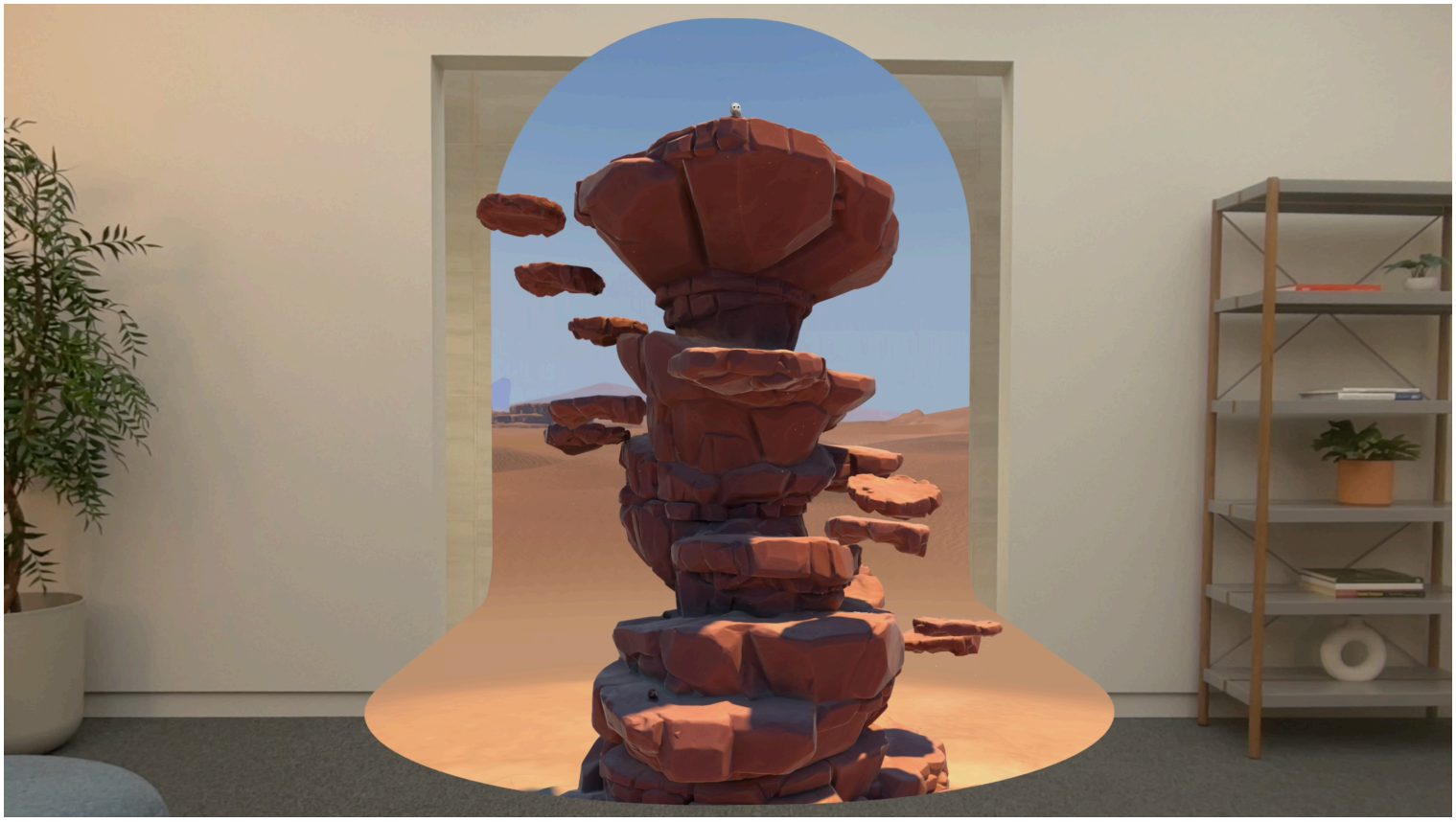
Use the latest RealityKit APIs to create a beautiful video game for visionOS.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

This sample code project uses RealityKit for visionOS to create a video game that tells the story of a lost chondrite as she collects her missing rock friends in a beautifully rendered environment.



The sample shows you how to use native APIs to leverage the full power of Apple Vision Pro in a real-world scenario. Its code and assets provide examples and inspiration so that you can create your own spectacular apps and games for Apple Vision Pro. The game supports two input modes for jumping: single input look-based jumping, and dual input pinch-based jumping.

Climb the butte with single input gestures

After our hero crash-lands on Earth, you begin controlling her movement using spatial gestures. By pinching and dragging, you can guide the character toward her destination.

When the single input mode is active, the player looks and taps a target destination and the character leaps into the air toward it, allowing her to begin the treacherous journey up the rocky landmark.

A `SpatialTapGesture` handles look-based jumping.

```
struct SingleInputJumpGesture: Gesture {
    @Environment(AppModel.self) private var appModel

    var body: some Gesture {
        SpatialTapGesture()
            // Only target this gesture to entities with the custom component.
            .targetedToEntity(where: .has(LevelInputTargetComponent.self))
             // The character jumps when the gesture ends.
            .onEnded() { event in
                // Guard for the character's container entity.
                guard let containerEntity = appModel.character.parent else { return

                // Convert the tap position to scene space.
                var targetPosition = event.convert(event.location3D, from: .local, t

                // Next, convert the scene-space position to one in the character's
                targetPosition = containerEntity.convert(position: targetPosition, t

                // Pass the jump target position to a custom component for this game
                appModel.character.components[CharacterMovementComponent.self]?.targ

                // Reset the jump buffer timer, which helps the game feel more respo
                // ground.
                appModel.character.components[CharacterMovementComponent.self]?.jump
            }
    }
```

A separate `DragGesture` handles rolling the character on the ground.

```
struct SingleInputDragGesture: Gesture {
    @Environment(AppModel.self) private var appModel

    var isDragActive: GestureState<Bool>
    @State private var dragStartPosition: SIMD3<Float> = .zero
    @State private var isDragging = false

    var body: some Gesture {
        DragGesture(minimumDistance: CGFloat(GameSettings.dragMinimumDistance), coor
            .targetedToAnyEntity()
            .updating(isDragActive) { value, state, transaction in
                state = true
            }
            .onChanged() { event in
                // Guard for the nearest physics simulation entity.
                guard let physicsRoot = PhysicsSimulationComponent.nearestSimulation

                // Get the drag position in scene space.
                let dragPosition = event.convert(event.location3D, from: .local, to:

                // Start the drag if the player isn't already dragging.
                if !isDragging {
                    dragStartPosition = dragPosition
                    isDragging = true
                }

                // Update the scene-space, drag-start position.
                dragStartPosition = updateDragStartPosition(
                    dragStartPosition: dragStartPosition,
                    dragPosition: dragPosition,
                    physicsRoot: physicsRoot,
                    useRelativeDragInput: appModel.rollInputMode == .relative
                )

                let sceneDragDelta = dragPosition - dragStartPosition
                // Normalize the scene-space drag translation and pass it to the cha
                let normalizedSceneDragDelta = sceneDragDelta == .zero ? .zero : sim
                let inputDirection = normalizedSceneDragDelta * (min(length(sceneDra
                appModel.character.components[CharacterMovementComponent.self]?.inpu
```

```
                appModel.character.components[CharacterMovementComponent.self]?.drag
            }
            .onEnded() { event in
                isDragging = false
            }
    }
}
```

The `updateDragStartPosition` method updates the drag start position so that it remains coplanar with the current drag position. When the `useRelativeDragInput` parameter is true, the method also updates the drag start position to follow behind the current drag position when the player drags beyond a specific radius, which improves the input experience for some players.

```
func updateDragStartPosition(dragStartPosition: SIMD3<Float>,
                             dragPosition: SIMD3<Float>,
                             physicsRoot: Entity,
                             useRelativeDragInput: Bool) -> SIMD3<Float> {
    // Convert the drag start and current position to the local space of the physics
    let dragPositionInPhysicsSpace = physicsRoot.convert(position: dragPosition, fro
    var dragStartPositionInPhysicsSpace = physicsRoot.convert(position: dragStartPos
    // Project the drag start position to an XZ-plane that's parallel to the current
    dragStartPositionInPhysicsSpace.y = dragPositionInPhysicsSpace.y
    // Get the drag translation in the XZ-plane of the local space of the physics ro
    let dragDelta = (dragPositionInPhysicsSpace - dragStartPositionInPhysicsSpace)

    // When `useRelativeDragInput` is true, the drag start point will follow behind
    let dragDistance = length(dragDelta)
    let dragRadius = GameSettings.dragRadius / GameSettings.scale
    if useRelativeDragInput && dragDistance > dragRadius {
        // Move the drag start position so that it follows behind the current drag p
        // input device all the way back to change direction.
        let normalizedDragDelta = dragDelta / dragDistance
        dragStartPositionInPhysicsSpace = dragPositionInPhysicsSpace - normalizedDra
    }

    // Update the scene-space, drag-start position.
    return physicsRoot.convert(position: dragStartPositionInPhysicsSpace, to: nil)
}
```

> **Note**
>
> When running the sample project in Simulator, you can hold the Shift key on your keyboard while dragging the mouse to improve the feel of the simulated drag gesture. To learn more, see Interacting with your app in the visionOS simulator.

Climb the butte with dual input gestures

In the dual input mode, a custom `SpatialEventGesture` allows the player to move the character and jump at the same time by tracking the state of two spatial events simultaneously.

```
struct DualInputGesture: Gesture {
    enum SpatialEventClassification {
        case pinch
        case drag
        case unresolved
    }

    struct SpatialEventState {
        var classification: SpatialEventClassification = .unresolved
        var chirality: Chirality
        var startPosition: SIMD3<Float>
        var translation: SIMD3<Float> = .zero
        var startTime: TimeInterval
        var duration: TimeInterval = 0
    }

    @Environment(AppModel.self) private var appModel

    var isDragActive: GestureState<Bool>
    @State var activeSpatialEvents: [SpatialEventCollection.Event.ID: SpatialEventSt

    private func handleSpatialEventEnded(spatialEvent: SpatialEventState?) {
        if spatialEvent?.classification == .unresolved {
            appModel.character.components[CharacterMovementComponent.self]?.jumpBuf
        }
    }

    // ...

    var body: some Gesture {
```

```
        SpatialEventGesture()
            .targetedToAnyEntity()
            .updating(isDragActive) { value, state, transaction in
                state = activeSpatialEvents.values.contains(where: { $0.classificati
            }
            .onChanged() { event in
                // Update the active spatial events.
                updateActiveSpatialEvents(event: event)

                // Classify unresolved spatial events.
                classifyUnresolvedSpatialEvents()

                // Respond to the active spatial events.
                respondToActiveSpatialEvents()
            }.onEnded() { event in
                // Handle and remove any events that ended.
                for value in event.gestureValue {
                    if value.phase == .ended {
                        handleSpatialEventEnded(spatialEvent: activeSpatialEvents[va
                    }
                    activeSpatialEvents[value.id] = nil
                }
            }
    }
}
```

The `updateActiveSpatialEvents` method updates the `SpatialEventState` of all active spatial events by tracking their chirality, position, translation, and duration.

```
private func updateActiveSpatialEvents(event: EntityTargetValue<SpatialEventGesture.
    // Guard for the nearest physics simulation entity.
    guard let physicsRoot = PhysicsSimulationComponent.nearestSimulationEntity(for:

    for value in event.gestureValue {
        // Skip spatial events without chirality.
        guard let chirality = value.chirality else {
            continue
        }

        // Get the event position in scene space.
        let spatialEventPosition = event.convert(value.location3D, from: .local, to:
```

```
        // Handle and remove the event if it ended.
        if value.phase == .ended {
            handleSpatialEventEnded(spatialEvent: activeSpatialEvents[value.id])
            activeSpatialEvents[value.id] = nil
        // Update the event state if it's already active.
        } else if var activeSpatialEvent = activeSpatialEvents[value.id] {
            // Update the scene-space, event-start position.
            activeSpatialEvent.startPosition = updateDragStartPosition(
                dragStartPosition: activeSpatialEvent.startPosition,
                dragPosition: spatialEventPosition,
                physicsRoot: physicsRoot,
                useRelativeDragInput: appModel.rollInputMode == .relative
            )
            // Update the scene-space event translation.
            activeSpatialEvent.translation = spatialEventPosition - activeSpatialEve
            activeSpatialEvent.duration = value.timestamp - activeSpatialEvent.start
            activeSpatialEvents[value.id] = activeSpatialEvent
        // Otherwise, create a new state structure to track this event.
        } else {
            // Add the event to the dictionary of active spatial events.
            let spatialEventState = SpatialEventState(chirality: chirality,
                                                      startPosition: spatialEventPos
                                                      startTime: value.timestamp)
            activeSpatialEvents[value.id] = spatialEventState
        }
    }
}
```

The `classifyUnresolvedSpatialEvents` method classifies any `.unresolved` spatial events as either a `.pinch` or a `.drag`.

```
private func classifyUnresolvedSpatialEvents() {
    for (spatialEventId, spatialEvent) in activeSpatialEvents where spatialEvent.cla
        // Classify the event as a pinch if there's already an active drag event.
        if activeSpatialEvents.values.contains(where: { $0.classification == .drag }
            activeSpatialEvents[spatialEventId]?.classification = .pinch
        // Classify the event as a drag if there's already an active pinch event
        // or the length of event's translation is larger than the drag minimum dist
        } else if activeSpatialEvents.values.contains(where: { $0.classification ==
                    length_squared(spatialEvent.translation) > GameSettings.dragMini
            activeSpatialEvents[spatialEventId]?.classification = .drag
        }
    }
```

```
}
```

Finally, the `respondToActiveSpatialEvents` uses the classification of each active spatial event along with their state data to move the character and make the character jump.

```
private func respondToActiveSpatialEvents() {
    for spatialEvent in activeSpatialEvents.values {
        switch spatialEvent.classification {
            case .drag:
                // Move the character in the direction of the spatial event translat
                var inputDirection = spatialEvent.translation / GameSettings.dragRad
                let inputDirectionMagnitude = length(inputDirection)
                if inputDirectionMagnitude > 1 {
                    inputDirection /= inputDirectionMagnitude
                }
                appModel.character
                    .components[CharacterMovementComponent.self]?.inputMoveDirectior
                appModel.character
                    .components[CharacterMovementComponent.self]?.dragDelta = spatia
            case .pinch:
                // Make the character jump if the player pinched this frame.
                if spatialEvent.duration == 0 {
                    appModel.character.components[CharacterMovementComponent.self]?.
                }
            default:
                break
        }
    }
}
```

Rotate the world in a mixed space

In this game, the world itself rotates as the character circles the butte. All physics entities in this sample app are descendants of a `PhysicsSimulationComponent` entity. When you translate, rotate, or scale this entity, the entire physics world transforms with it. The physics simulation component entity serves as the root entity for the physics world, and you can move it like a camera inside custom systems (although the transformations are inverted). For more information, see Handling different-sized objects in physics simulations.

Before animating the physics root, create an extension method to interpolate floating-point values using a damping function. This makes animations feel less abrupt.

```
private func dampingFactor(smoothing: Float, deltaTime: Float) -> Float {
    smoothing == 0 ? 0 : 1 - exp2(-deltaTime / smoothing)
}

public extension Float {
    /// Perform a damped interpolation between the current value and a target value.
    mutating func lerpTo(_ targetFloat: Float, smoothing: Float, deltaTime: Float) {
        self = simd_mix(self, targetFloat, dampingFactor(smoothing: smoothing, delta
    }
}
```

Additionally, to perform the necessary calculations to determine which direction the butte rotates, the sample uses helper functions to derive the signed angle between two directions.

```
public func angleBetween(from fromVector: SIMD3<Float>, to toVector: SIMD3<Float>) -
    acos(simd_clamp(dot(normalize(fromVector), normalize(toVector)), -1, 1))
}

public func signedAngleBetween(from fromVector: SIMD3<Float>, to toVector: SIMD3<Flo
    let sign: Float = dot(cross(fromVector, toVector), axis) > 0 ? 1 : -1
    let angleBetween = angleBetween(from: fromVector, to: toVector)
    return angleBetween * sign
}
```

There is no camera entity in this sample code project. Instead, the butte itself rotates as the character progresses through the level. When the character moves outside a threshold, the sample calculates the angle necessary to rotate the butte so the character is always visible to the player.

```
// Get the direction to the follow target entity.
var toFollowTarget = rotationComponent.followTarget.position(relativeTo: rotationEnt
toFollowTarget.y = 0

// Calculate the angle between the follow target and the forward direction.
var forwardDirection = rotationEntity.convert(direction: .forward, from: nil)
forwardDirection.y = 0
let angleBetweenForward = signedAngleBetween(from: toFollowTarget, to: forwardDirect
let isOutsideThreshold = abs(angleBetweenForward) > rotationComponent.rotationThresh

// Determine whether the target entity is in the camera rotation deadzone.
let distance = length(SIMD3<Float>(rotationComponent.followTarget.position.x, 0, rot
```

```
let isInHorizontalDeadzone = distance <= rotationComponent.deadZoneRadiusAndHeight.r
let height = rotationComponent.followTarget.position.y
let isInDeadzone = (isInHorizontalDeadzone && height >= rotationComponent.deadZoneRa
    || height >= rotationComponent.deadzoneMinHeight

// When outside the threshold, calculate a new rotation for the butte that brings th
if isOutsideThreshold && isInDeadzone == false {
    let angleDifference = if angleBetweenForward > 0 {
        angleBetweenForward - rotationComponent.rotationThreshold
    } else {
        angleBetweenForward + rotationComponent.rotationThreshold
    }
    rotationComponent.targetAngle = rotationComponent.angle + angleDifference
}

// Increase the rotation smoothing when inside the deadzone, and decrease it back to
if isInDeadzone {
    rotationComponent.dyanamicRotationSmoothing.lerpTo(1, smoothing: 1, deltaTime: d
} else {
    rotationComponent.dyanamicRotationSmoothing.lerpTo(rotationComponent.rotationSmo
}

// Rotate the camera rotation angle toward the target rotation angle.
rotationComponent.angle
    .lerpTo(rotationComponent.targetAngle, smoothing: rotationComponent.dyanamicRota
```

Prepare assets for gameplay

Using third-party digital content creation (DCC) tools to create the visual assets for this sample app, you can export those assets as USD files, and then import and arrange them inside Reality Composer Pro. Then you can apply custom components to the entities in a Reality Composer Pro scene, and custom systems can look for those components to process entities for gameplay. For more information, see Adding assets to your Reality Composer Pro scene.

To generate the collision component that uses the shape of the butte, you first use a DCC to generate a model that matches the shape of the butte and platforms, but that contains fewer vertices. In Reality Composer Pro, you apply a custom component to the model entity. The custom system looks for that component by subscribing to the `ComponentEvents.DidAdd` event for a custom type in the initializer for a custom system.

```
// Store subscriptions in a list.
var subscriptions: [AnyCancellable] = .init()
```

```
required init (scene: Scene ) {
    // Register the `onDidAddCompoundCollisionMarker` callback when adding a custom
    // The callback runs on the scene load.
    scene.subscribe(to: ComponentEvents.DidAdd.self, componentType: CompoundCollisio
        self.onDidAddCompoundCollisionMarker(event: $0)
    }.store(in: &subscriptions)
}
```

On the first scene load, RealityKit adds the component to an entity. The custom system searches for model components that descend from that entity. The system then creates a collision component on the entity using all the shapes that the mesh data generates.

To perform a recursive operation on each descendant entity, the sample uses an extension method for `Entity`.

```
/// A recursive search of all descendants with a specific component.
public func forEachDescendant<T: Component>(withComponent componentClass: T.Type, _
    for descendant in children {

        // Run the closure using the subentity and its component as parameters.
        if let component = descendant.components[componentClass] {
            closure(descendant, component)
        }

        // Call this same function for each descendant entity.
        descendant.forEachDescendant(withComponent: componentClass, closure)
    }
}
```

You can then use this extension method to generate the shape data for every descendant model component.

```
var meshes = [(entity: Entity, mesh: MeshResource)]()
collisionRoot.forEachDescendant(withComponent: ModelComponent.self) {
    (entity, modelComponent) in

    // Skip descendant entities that you don't want to become part of the collision
    guard entity.components.has(IgnoreCompoundCollisionMarkerComponent.self) == fals

    meshes.append((entity: entity, mesh: modelComponent.mesh))
```

```
    // Optionally, delete the source model component if you're no longer using it.
    if deleteModel {
        entity.components.remove(ModelComponent.self)
    }
}
```

Next, use `generateStaticMesh(from:)` to create a shape from each discovered mesh resource, and then offset that shape relative to the collision root entity (the original entity with the custom component).

```
for (entity, mesh) in meshes {
    // Generate the shape from the mesh data.
    guard var shape = if isStatic {
        try? await ShapeResource.generateStaticMesh(from: mesh)
    } else {
        try? await ShapeResource.generateConvex(from: mesh)
    } else {
        continue
    }

    // Offset the shape by its translation and orientation relative to the collisior
    shape = shape.offsetBy(rotation: entity.orientation(relativeTo: collisionRoot),
    shapes.append(shape)
}
```

Finally, the sample initializes the collision component with the array of shapes and then adds it to the collision root:

```
let collision = CollisionComponent(shapes: shapes, mode: collisionMode)
collisionRoot.components.set(collision)
```

The sample also loads and configures audio assets in code. In this sample, a custom system accumulates collision sounds into a Swift list, and then passes the sounds into the initializer, `init(_:)`, for an `AudioFileGroupResource`. On startup, the app loads audio files into the scene using the `AudioResourcesComponent`. This component's load function then caches the `AudioResource` using an `AudioLibraryComponent` for retrieval by name later in the app code. The app also adds other sounds, such as music and environmental ambiences, into the `AudioResourcesComponent`, in addition to the collision sounds, for later use.

Structure your project for development

During development, many people with a diverse set of expertise work on the same Xcode project and in the same Reality Composer Pro scenes. It's important to think strategically about your project structure to avoid cumbersome merge conflicts or accidentally undoing someone else's changes.

Within Reality Composer Pro, USD references allow you to isolate your work to individual files. The same asset becomes available for reference multiple times throughout a Reality Composer Pro project.

As an example, in this sample code project, the original materials are in a separate scene. Additionally, the main materials scene instances and reuses custom node graphs in other materials. One example is `DropShadow`, the node graph for rendering drop shadows.

For USD assets, the source models are in their own folder. These assets don't have applied materials, and don't contain any configuration data necessary for gameplay.

> **Important**
>
> Be mindful of the size of your assets during development. Textures and other large assets affect the build times for your app.

In the `GameAssets` folder, create game assets by configuring source assets with materials and any custom component data necessary. Those game assets are then ready for a designer to assemble into levels.

Finally, assemble the game assets in the completed game level scene.

Create effects with the shader graph

Adopting a variety of techniques, including making custom materials with ShaderGraph in Reality Composer Pro, promotes efficient rendering of the towering landmark at the center of the hero's journey. A combination of baked light maps, which you generate in an external DCC, and clever lighting techniques come together to make the player's experience smooth and rewarding.

Unlit materials are very performant because they don't require lighting calculations from the GPU to determine their color. The materials for the butte use textures you create in an external DCC, allowing you to calculate shadows from the sun ahead of time, and preventing real-time lights from casting shadows onto the butte.

To achieve effective grounding shadows beneath the character, perform a ray cast downward from her position and check for collisions with geometry. `calculateParametersForShadow` implements the check and returns shader parameters in a tuple.

```
func calculateParametersForShadow(_ entity: Entity, _ physicsRoot: Entity)
    -> (characterPosition: SIMD3<Float>, shadowYPosition: Float)? {
    // Get the origin relative to the physics root entity.
    let origin = entity.position(relativeTo: physicsRoot)

    // Perform a ray cast against the scene downward from the origin.
    return if let hit = entity.scene?.raycast(
        origin: origin,
        direction: [0, -1, 0],
        query: .nearest,
        // Use a mask to make sure you're only performing a ray cast against entitie
        mask: GameCollisionGroup.shadowReceiver.collisionGroup,
        relativeTo: physicsRoot
    ).first {
        // Return a tuple when the ray cast is successful.
        (origin, hit.position.y)
    } else {
        nil
    }
}
```

On each frame, the CPU calculates the shader parameters and passes them to a GPU compute shader which writes them into a texture. This happens in the update function of a custom system.

```
func update(context: SceneUpdateContext) {
        // Guard for the physics root and the character entity.
        guard let physicsRoot = context.first(withComponent: PhysicsSimulationCompor
                let character = context.first(withComponent: CharacterMovementCompor

        // Get the matrix that transforms from world space to level space.
        let worldToLevelMatrix = physicsRoot.transformMatrix(relativeTo: nil).invers

        // Ray cast downward to determine where the character's shadow lands.
        if let (characterPosition, characterShadowYPosition) = calculateParametersFo

            // Ray cast downward for each rock friend to determine where their shado
            var rockFriendPositions = [(position: SIMD3<Float>, shadowYPosition: Flo
            for rockFriend in context.entities(matching: rockFriendQuery, updatingSy
```

```
                if let (friendPosition, friendShadowYPosition) = calculateParameters
                    rockFriendPositions.append((friendPosition, friendShadowYPositio
                }
            }

            // Dispatch a compute shader to write the shadow positions to the low-le
            // ...

            // Send the shadow parameters to the shader.
            for dropShadowReceiver in context.entities(matching: dropShadowReceiverQ
                setShadowShaderParameters(entity: dropShadowReceiver, worldToLevelMa
            }
        }
    }
```

See the `DropShadowComputeShader.metal` file in the sample project for the full compute shader implementation.

Inside `setShadowShaderParameters`, the sample sets the properties on the custom material by getting a reference to the `ShaderGraphMaterial` on the entity's `ModelComponent`. The sample then applies the modified shader graph material back to the shadow receiver entity directly.

```
func setShadowShaderParameters (entity: Entity, worldToLevelMatrix: simd_float4x4) {
    if let dropShadowReceiverModelComponent = entity.components[DropShadowReceiverMo
        // Iterate through each shadow material on this model and apply the shadow s
        for materialIndex in dropShadowReceiverModelComponent.shadowMaterialIndices
            guard var shaderGraphMaterial = entity.components[ModelComponent.self]?.
                continue
            }

            try? shaderGraphMaterial.setParameter(handle: dropShadowReceiverModelCom
                                                  value: .float4x4(worldToLevelMatri

            entity.components[ModelComponent.self]?.materials[materialIndex] = shade
        }
    }
}
```

Important

Updating parameters on an entity's material can be computationally expensive. Avoid sharing shader graph material handles between entity instances, or setting the model component back on the entity unnecessarily.

Understand how collision audio works

In Petite Asteroids, the audio system has multiple types of collision sounds. These sounds play depending on the `CollisionEvents` of their respective component. These events govern when and how to play the audio accordingly. The information that the system receives from the physics and collision events determines the loudness of the audio playback.

The physics event calculates the velocity of the character or whether the character stops jumping, which changes the nature of the audio playback. The collision event provides information on the `impulse`, which is directly proportional to the loudness of the audio playback. When the character jumps or falls off the butte, she lands on a virtual surface. The app plays a sound whenever the character collides with a virtual surface.

The sample shows how to handle collision events, play a sound upon collision, and track the collision entity throughout events:

```
import RealityKit

class GameMovementSystem: System {
    var subscriptions: [AnyCancellable] = .init()

    required init(scene: RealityKit.Scene) {
    // Subscribe to the CollisionEvents and connect to class methods.
    scene.subscribe(to: CollisionEvents.Began.self, componentType: GameMovementCompo
    scene.subscribe(to: CollisionEvents.Updated.self,
                    componentType: CharacterMovementComponent.self,
                    onCollisionUpdated).store(in: &subscriptions)
    scene.subscribe(to: CollisionEvents.Ended.self, componentType: GameMovementCompo
    }

    @MainActor
    func onCollisionBegan(event: CollisionEvents.Began) {
        let gameEntity = event.entityA
        let collisionEntity = event.entityB

        updateCollisionClassification(entityA: event.entityA, entityB: event.entityB
```

```
        event.entityA.components[GameMovementComponent.self].currentlyTrackedCollisi

        if let collisionClassification = event.entityA.components[GameMovementCompor
            // If collision impulse reaches over a specific threshold, play a sound.
            if event.impulse > GameSettings.collisionImpulseThreshold {
                let audioEvent = AudioEventComponent(resourceName: "CollisionSound")
                gameEntity.components.set(audioEvent)
            }
        }
    }

    @MainActor
    func onCollisionUpdated(event: CollisionEvents.Updated) {
        updateCollisionClassification(entityA: event.entityA, entityB: event.entityB
    }

    @MainActor
    func onCollisionEnded(event: CollisionEvents.Ended) {
        // Stop tracking.
        event.entityA.components[GameMovementComponent.self]?.trackedCollisionEntiti

        if event.entityB == event.entityA.components[GameMovementComponent.self]?.cu
            event.entityA.components[GameMovementComponent.self]?.currentlyTrackedCo
    }



    private func updateCollisionClassification(entityA: Entity, entityB: Entity, cor
        guard var collisionNormal = contacts.first?.normal else { return }

        collisionNormal = normalize(collisionNormal)

        let collisionDot = dot(collisionNormal, [0, 1, 0])
        let classification: CollisionClassification = if collisionDot < -GameSetting
            .top
        } else if collisionDot == GameSettings.floorThreshold {
            .floor(normal: collisionNormal)
        }  else {
           .inTheAir(normal: collisionNormal)
        }

        entityA.components[GameMovementComponent.self].trackedCollisionEntities[enti
    }
```

```
}
```

The collision sounds in Petite Asteroids are usually one-shot collision sounds, which the app plays using `playAudio(_:)`. For other collision sounds, the app groups a set of similar sounds together using an `AudioFileGroupResource` to play nonrepeating random sounds for audio playback.

Design dynamic sounds

In this game, the Audio `Entity` uses an `AmbientAudioComponent` for ambient audio. The system plays two audio files using `AudioPlaybackController` for the environment audio of the game. The character starts at the bottom of the butte with a calmer environment. As she reaches higher parts of the butte, the calmer environment cross-fades with the windier environment. The system blends these two files according to how high the character ascends. If she falls, the windier environment fades gracefully by interpolating values over a number of seconds.

The soundstage design intentionally utilizes stereo music with spread and width (decorrelated content), so any spatial sound effects in the game play closer to the center of the view. This way, the music doesn't distract from the overall game experience, and improves the sense of immersion. To accomplish this effect, the app uses psychoacoustic and filtering techniques:

- At the bottom of the butte, the app uses a stereo recording asset, anchored to the volume.
- As you reposition the volume, the stereo recording follows it, so you can localize the app.
- The top of the butte has a quadraphonic (may be cube) surround layout recording, so as the character ascends the butte, the audio experience becomes increasingly immersive.

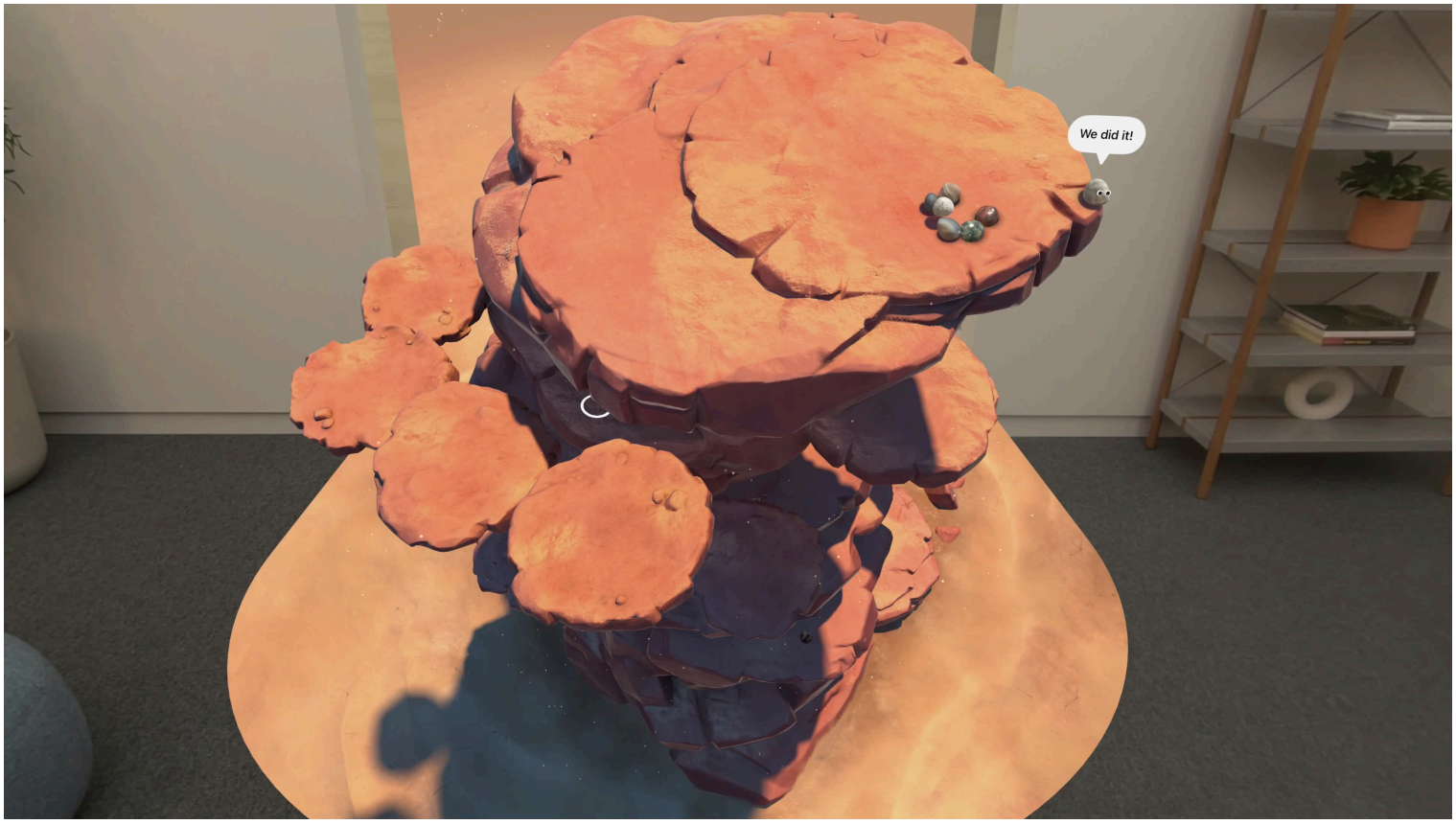
An audio cue subsystem in Petite Asteroids' audio system controls playback of the app's music. The sound effects of the game differ, depending on the scenes of the game. In the Fiery Descent sequence, the app plays back two layers simultaneously:

Mono layer
: Playing a spatial, high-frequency, single-channel point source.

Stereo layer
: Playing lower-frequency sounds with some subtle panning movement for immersion. One of the `fieryDescent` sounds, `fieryDescentSFXSpatial`, plays on `ParticleEmitter` entities for spatial audio that layers with ambient audio `fieryDescentSky` and `fieryDescentSFXAmbient`.

The design of the music scoring separates linear and nonlinear categories. The linear music at the end of the game triggers a cut scene and the app plays a linear music sequence. The nonlinear music scores make the sound nonrepetitive. This design means the app can cut the score into segments that it can loop infinitely and cleanly, while also allowing the audio to start playback at

any randomized time. The `gameplayMusic`, `tutorialMusic`, and `menuMusic` all fall under this category.



Play ⊙

See Also

RealityKit and Reality Composer Pro

- Reality Composer Pro

  Build, create, and design 3D content for your RealityKit apps.

- {} BOT-anist

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

- {} Swift Splash

  Use RealityKit to create an interactive ride in visionOS.

- {} Diorama

  Design scenes for your visionOS app using Reality Composer Pro.

- {} Building an immersive media viewing experience

Add a deeper level of immersion to media playback in your app with RealityKit and Reality Composer Pro.

{} Enabling video reflections in an immersive environment

Create a more immersive experience by adding video reflections in a custom environment.

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

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

Understanding the modular architecture of RealityKit

Learn how everything fits together in RealityKit.

Using transforms to move, scale, and rotate entities

Learn how to use Transforms to move, scale, and rotate entities in RealityKit.

Capturing screenshots and video from Apple Vision Pro for 2D viewing

Create screenshots and record high-quality video of your visionOS app and its surroundings for app previews.

Implementing object tracking in your visionOS app

Create engaging interactions by training models to recognize and track real-world objects in your app.

{} Placing entities using head and device transform

Query and react to changes in the position and rotation of Apple Vision Pro.