Sample Code

Simulating physics with collisions in your visionOS app

Create entities that behave and react like physical objects in a RealityKit view.

Download

visionOS 1.1+ | Xcode 15.3+

Overview

This sample demonstrates how to simulate multiple physical objects that collide with each other by creating a volumetric window with sphere entities that each have a collision component and a physics body component.

Play

The sample also adds a custom component and system that applies an attraction force between the spheres, and gesture support so that a person can move the spheres within the volume.

Add a volumetric window to your app

The app starts by creating a scene that includes a window group with a `volumetric` style so that the spheres behave like physical objects in the environment.

```
import SwiftUI

@main
struct PhysicsBodiesApp: App {
    var body: some Scene {
        WindowGroup {
            MainView()
        }.windowStyle(.volumetric)
    }
}
```

Volumetric windows are viewable from all sides and have a constant size in the environment. If you want to prioritize visibility from a distance rather than from all sides, use the `plain` window style instead. Plain windows are resizable, which can help people see the contents of a window from a distance, but there's one optimal viewing angle.

The app's main view creates the spheres and an invisible containment box that keeps the spheres from drifting out of the volume.

```
var body: some View {
    GeometryReader3D { geometry in
        RealityView { content in
            addSpheres(content)
            content.add(containmentCollisionBox)
        } update: { content in
            let localFrame = geometry.frame(in: .local)
            let sceneFrame = content.convert(localFrame,
                                             from: .local, to: .scene)

            containmentCollisionBox.update(sceneFrame)
        }.gesture(ForceDragGesture())
    }
}
```

SwiftUI calls the view's `update` closure when one of its inputs change, such as `geometry` in this example, which is a `GeometryReader3D` instance. The closure updates the collision box by passing the view's current position and size in scene coordinates. To do this, the closure:

- Retrieves the view's current position and size in SwiftUI coordinates, in points, as a `Rect3D` instance by calling the `frame(in:)` method of the `geometryReader` that contains the `RealityView`
- Converts the 3D rectangle from points to RealityKit scene coordinates, in meters, by calling the `RealityViewContent` instance's `convert(_:from:to:)` method

Coordinate values in SwiftUI are typically in the hundreds of points, while coordinates in RealityKit often have much smaller values, typically fractions of a meter.

The view also adds a `DragGesture` instance so that a person can move a sphere around and make it collide with the other spheres in the volume.

Create a model entity with a physics body

The app creates each sphere as a `ModelEntity` instance with a spherical mesh.

```
let sphereEntity = ModelEntity(
    mesh: MeshResource.generateSphere(radius: sphereRadius),
    materials: [metallicSphereMaterial()]
)
```

Alternatively, you can create an `Entity` instance and add a `ModelComponent` to it, which is the equivalent of a `ModelEntity`.

To customize each sphere's appearance, the app configures a material. The app applies a unique material with a random color to each sphere by creating and configuring a new `PhysicallyBasedMaterial` instance.

```
private func metallicSphereMaterial(
    hue: CGFloat = CGFloat.random(in: (0.0)...(1.0))
) -> PhysicallyBasedMaterial {
    var material = PhysicallyBasedMaterial()

    let color = RealityKit.Material.Color(
        hue: hue,
        saturation: CGFloat.random(in: (0.5)...(1.0)),
        brightness: 0.9,
        alpha: 1.0)

    material.baseColor = PhysicallyBasedMaterial.BaseColor(tint: color)
    material.metallic = 1.0
    material.roughness = 0.5
    material.clearcoat = 1.0
    material.clearcoatRoughness = 0.1

    return material
}
```

The app configures each material so that is has a shiny, metallic look. It sets:

- `baseColor` to a random hue within a range of 80-100%.
- `metallic` to `1.0`, which makes the material reflective.
- `roughness` of `0.5`, a relatively high value, which makes the reflections blurrier.
- `clearcoat` to `1.0`, which gives the material white reflections on top of the metallic ones.
- The `clearcoatRoughness` to `0.1`, a relatively low value, which gives sharp clear coat reflections.

Customizing the attributes of materials, including color, roughness, and reflectiveness, adds a realistic appearance to the items in your scene.

Experiment

Change the visual appearance of the spheres by altering the material, or by applying the plastic, wood, rubber, or glass materials with a `ShaderGraphMaterial` and a `Environment Radiance (RealityKit)` node.

To make the spheres interact with the physics system, the app adds a `CollisionComponent` and a `PhysicsBodyComponent` to each sphere's entity.

```
// Create the physics body from the same shape.
let shape = ShapeResource.generateSphere(radius: sphereRadius)
sphereEntity.components.set(CollisionComponent(shapes: [shape]))

var physics = PhysicsBodyComponent(
    shapes: [shape],
    density: 10_000
)

// Make each sphere float in the air by turning off gravity.
physics.isAffectedByGravity = false

// Add the physics component to the sphere.
sphereEntity.components.set(physics)
```

The collision component includes the entity in the scene's physics simulation. The physics body component defines the entity's physical shape and mass. By default, entities with a physics body component react to gravity, but the app makes the spheres float in the scene by setting the component's `isAffectedByGravity` property to `false`.

Add attraction forces between all the spheres by creating a custom system

The app simulates forces of attraction between the spheres by defining two types:

- `SphereAttractionSystem`
- `SphereAttractionComponent`

The app adds each sphere to the system by setting the custom component to the sphere's component set.

```
sphereEntity.components.set(SphereAttractionComponent())
```

The sample's `SphereAttractionComponent` structure conforms to the `System` protocol by implementing the `init(scene:)` initializer and `update(context:)` method it requires.

```
struct SphereAttractionSystem: System {
    let entityQuery: EntityQuery

    init(scene: RealityKit.Scene) {
        let attractionComponentType = SphereAttractionComponent.self
        entityQuery = EntityQuery(where: .has(attractionComponentType))
    }

    func update(context: SceneUpdateContext) {
        // ...
    }
}
```

The app simulates an attractive force between each sphere and all the other spheres in the scene in the custom system's `update()` method.

```
func update(context: SceneUpdateContext) {
    let sphereEntities = context.entities(
        matching: entityQuery,
        updatingSystemWhen: .rendering
    )

    for case let sphere as ModelEntity in sphereEntities {
        var aggregateForce: SIMD3<Float>

        // Start with a force back to the center.
        let centerForceStrength = Float(0.05)
        let position = sphere.position(relativeTo: nil)
        let distance = length_squared(position)

        // Set the initial force with the inverse-square law.
        aggregateForce = normalize(position) / distanceSquared
```

```
        // Direct the force back to the center by negating the position vector.
        aggregateForce *= -centerForceStrength

        let neighbors = context.entities(matching: entityQuery,
                                         updatingSystemWhen: .rendering)

        for neighbor in neighbors where neighbor != sphere {

            let spherePosition = sphere.position(relativeTo: nil)
            let neighborPosition = neighbor.position(relativeTo: nil)

            let distance = length(neighborPosition - spherePosition)

            // Calculate the force from the sphere to the neighbor.
            let forceFactor = Float(0.1)
            let forceVector = normalize(neighborPosition - spherePosition)
            let neighborForce = forceFactor * forceVector / pow(distance, 2)
            aggregateForce += neighborForce
        }

        // Add the combined force from all the sphere's neighbors.
        sphere.addForce(aggregateForce, relativeTo: nil)
    }
}
```

When the app runs, the spheres move towards each other and clump together within the volume.

Play

For more information about systems, see Systems and Implementing systems for entities in a scene.

Keep the spheres in the volume by creating an invisible container with physics

The app prevents the spheres from moving out of the volume by creating a relatively thin box for each of the volume's six boundaries.

The sample's `ContainmentCollisionBox` class, which inherits from `Entity`, creates the six faces of the containment box in its `update(_:)` method.

```
func update(_ boundingBox: BoundingBox) {
    // ...

    // Define the constants for the faces' geometry for convenience.
    let min = boundingBox.min
    let max = boundingBox.max
```

```
    let center = boundingBox.center

    let lHandFace = SIMD3<Float>(x: min.x, y: center.y, z: center.z)
    let rHandFace = SIMD3<Float>(x: max.x, y: center.y, z: center.z)
    let lowerFace = SIMD3<Float>(x: center.x, y: min.y, z: center.z)
    let upperFace = SIMD3<Float>(x: center.x, y: max.y, z: center.z)
    let nearFace = SIMD3<Float>(x: center.x, y: center.y, z: min.z)
    let afarFace = SIMD3<Float>(x: center.x, y: center.y, z: max.z)

    // Make each box relatively thin.
    let thickness = Float(1E-3)

    // Configure the size for the left and right faces.
    var size = boundingBox.extents
    size.x = thickness

    // Create the left face of the collision containment cube.
    var face = Entity.boxWithCollisionPhysics(lHandFace, size)
    addChild(face)

    // Create the right face of the collision containment cube.
    face = Entity.boxWithCollisionPhysics(rHandFace, size)
    addChild(face)

    // Configure the size for the top and bottom faces.
    size = boundingBox.extents
    size.y = thickness

   // ...
}
```

The method creates each face by calling the `boxWithCollisionPhysics(_:_:boxMass:)` method the sample adds to `Entity` in an extension.

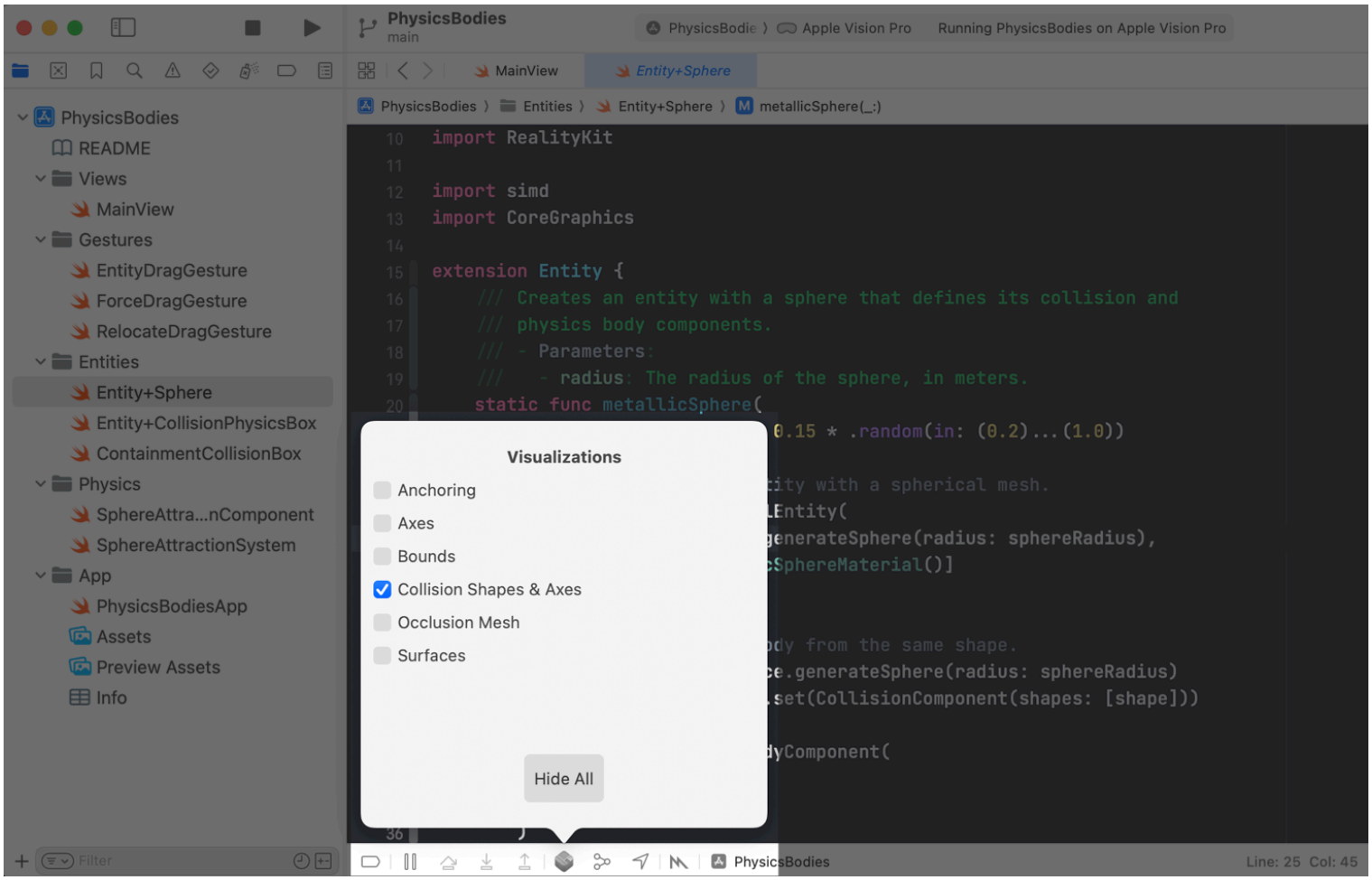
```
import RealityKit

/// The default mass for a new box.
private let defaultMass1Kg = Float(1.0)

extension Entity {
    // ...
    static func boxWithCollisionPhysics(
        _ location: SIMD3<Float>,
```

```
        _ boxSize: SIMD3<Float>,
        boxMass: Float = defaultMass1Kg
    ) -> Entity {
        // Create an entity for the box.
        let boxEntity = Entity()

        // Create the box's shape from the size.
        let boxShape = ShapeResource.generateBox(size: boxSize)

        // Create a collision component with the box's shape.
        let collisionComponent = CollisionComponent(
            shapes: [boxShape],
            isStatic: true)

        // Create a physics body component with the box's shape.
        let physicsBodyComponent = PhysicsBodyComponent(
            shapes: [boxShape],
            mass: boxMass,
            mode: PhysicsBodyMode.static
        )

        // Set the entity's position in the scene.
        boxEntity.position = location

        // Add the collision physics to the box entity.
        boxEntity.components.set(collisionComponent)
        boxEntity.components.set(physicsBodyComponent)
        return boxEntity
    }
}
```

The method configures each face to physically interact with the spheres in the scene by adding an instance of `CollisionComponent` and `PhysicsBodyComponent` to the face's entity.

Tip

For easier debugging, you can view every entity's collision shape. Select the Collision Shapes & Axes checkbox from the Debug Visualizations menu that appears in the debug bar at the bottom of the main pane in Xcode after you build the app.



Add human interaction by including a gesture

The app lets a person move the spheres within the volume by adding a `ForceDragGesture` instance to the main view.

```
import SwiftUI
import RealityKit

struct ForceDragGesture: Gesture {

    var body: some Gesture {
        EntityDragGesture { entity, targetPosition in
            guard let modelEntity = entity as? ModelEntity else { return }

            let spherePosition = entity.position(relativeTo: nil)

            let direction = targetPosition - spherePosition
            var strength = length(direction)
            if strength < 1.0 {
                strength *= strength
            }
```

```
            let forceFactor: Float = 3000
            let force = forceFactor * strength * simd_normalize(direction)
            modelEntity.addForce(force, relativeTo: nil)
        }
    }
}
```

The type applies a force to the sphere where a drag gesture begins, which increases with the distance between the sphere and drag gesture's current position, similar to a spring.

Play ⊙

To enable gestures on the Entities, the sample adds an `InputTargetComponent` as well as a `HoverEffectComponent` to provide feedback when a person looks at a sphere:

```
// Highlight the sphere when a person looks at it.
sphereEntity.components.set(HoverEffectComponent())

// Configure the sphere to receive gesture inputs.
sphereEntity.components.set(InputTargetComponent())
```

The `ForceDragGesture` type depends on the sample's `EntityDragGesture` type, which handles the logic to start and end a drag gesture and updates its current position during the drag.

The sample also defines another gesture type, `RelocateDragGesture` which moves a sphere entity by changing its location, instead of applying a force.

```
import SwiftUI
import RealityKit

struct RelocateDragGesture: Gesture {
    var body: some Gesture {
        EntityDragGesture { entity, targetPosition in
            entity.setPosition(targetPosition, relativeTo: nil)
        }
    }
}
```

This gesture effectively applies the equivalent of an infinite force, which can quickly move the sphere and the spheres it collides with. It directly modifies a sphere's location and bypasses physics.

Experiment

Replace the call to `ForceDragGesture()` in the app's main view with `RelocateDragGesture()` and compare the behavioral differences.

See Also

Collision shapes and groups

{} Configuring Collision in RealityKit

Use collision groups and collision filters to control which objects collide.

`struct CollisionComponent`

A component that gives an entity the ability to collide with other entities that also have collision components.

`enum Mode`

A mode that dictates how much collision data is collected for a given entity.

`class` **ShapeResource**

A representation of a shape.

`enum` **ShapeResourceError**

`struct` **CollisionGroup**

A bitmask used to define the collision group to which an entity belongs.

`struct` **CollisionFilter**

A set of masks that determine whether entities can collide during simulations.

`class` **TriggerVolume**

An invisible 3D shape that detects when objects enter or exit a given region of space.