Sample Code

Transforming RealityKit entities using gestures

Build a RealityKit component to support standard visionOS gestures on any entity.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

Manipulating virtual objects using standard system drag, rotate, and scale gestures is a common task in visionOS apps. This sample project demonstrates how to apply SwiftUI gestures to a RealityKit `Entity` by implementing the gestures with a component and an extension on `RealityView`. The component marks entities as supporting transform gestures and the `RealityView` extension passes events from the SwiftUI gesture actions to the component, which also contains the logic to implement the gestures. You can add transformation gesture support to any entity in your `RealityView` just by adding the gesture component to the entity.

Create the main component

To implement the gesture functionality, the sample app first creates a `struct` that conforms to `Component`. This component marks entities that support transform gestures, and contains the logic to implement those gestures. In order to support Reality Composer Pro, the component also conforms to `Codable`. To include the ability to turn different gestures on and off, the component contains three `Bool` properties, one for each of the transforms the component supports. Reality Composer Pro exposes the transforms as checkboxes, which enable or disable specific gestures for an entity.

```swift
/// A component that handles gesture logic for an entity.
public struct GestureComponent: Component, Codable {

    /// A Boolean value that indicates whether a gesture can drag the entity.
    public var canDrag: Bool = true

    /// ...

    /// A Boolean value that indicates whether a gesture can scale the entity.
    public var canScale: Bool = true

    /// A Boolean value that indicates whether a gesture can rotate the entity.
    public var canRotate: Bool = true

    /// ...
```

The component has two other properties for configuring the drag style, shown below:

```swift
    /// A Boolean value that indicates whether the drag gesture can move the object
    public var pivotOnDrag: Bool = true

    /// A Boolean value that indicates whether a pivot drag keeps the orientation tc
    /// viewer throughout the drag gesture.
    ///
    /// The property only applies when `pivotOnDrag` is `true`.
    public var preserveOrientationOnPivotDrag: Bool = true
```

The `pivotOnDrag` properties configure whether the object moves along the X-axis in a straight line, or pivots around a person, similar to the visionOS keyboard. When `pivotOnDrag` is `true`, the `preserveOrientationOnPivotDrag` determines if the object rotates to face the viewer as they drag it, such as for the visionOS keyboard, or keeps its original orientation throughout the drag gesture.

Pivoting on drag is more flexible because it allows a person to change their own orientation without the entity they're dragging from disappearing out of view. However, in some cases, dragging without the pivot may be a better option, such as when they need to precisely line up an entity with another.

Create a state object

To implement these transform gestures, the app needs to maintain some state. The onChanged(_:) action passes a delta from the start transform, *not* the delta from the previous onChanged(_:) call, so the app needs to keep track of the entity's starting position, rotation, and scale. For example, each time SwiftUI calls the onChanged(_:) action for a drag gesture, the action provides the total distance dragged on each axis since the gesture started. The app also keeps track of whether a gesture is already in progress. Gestures don't have an `.onStarted` action, so the app keeps track of whether the gesture has already started so it knows if it needs to store the starting position, rotation, or scale. Lastly, the sample app keeps a reference to the pivot entity. By parenting the dragged entity to the pivot entity, the system calculates the dragged entity's rotation when the app rotates the pivot entity.

This component only supports dragging a single entity at a time, so there's no need to store state on a per-entity level. As a result, the app uses a singleton object to store the state instead of a component:

```
public class EntityGestureState {

    /// The entity currently being dragged if a gesture is in progress.
    var targetedEntity: Entity?

    // MARK: - Drag

    /// The starting position.
    var dragStartPosition: SIMD3<Float> = .zero

    /// Marks whether the app is currently handling a drag gesture.
    var isDragging = false

    /// When `rotateOnDrag` is`true`, this entity acts as the pivot point for the dr
    var pivotEntity: Entity?

    var initialOrientation: simd_quatf?

    // MARK: - Magnify

    /// The starting scale value.
    var startScale: SIMD3<Float> = .one

    /// Marks whether the app is currently handling a scale gesture.
    var isScaling = false

    // MARK: - Rotation

    /// The starting rotation value.
    var startOrientation = Rotation3D.identity

    /// Marks whether the app is currently handling a rotation gesture.
    var isRotating = false

    // MARK: - Singleton Accessor

    /// Retrieves the shared instance.
    static let shared = EntityGestureState()
}
```

Add transform logic to the main component

`GestureComponent` needs to implement functions the app calls from the `onChanged(_:)` and `onEnded(_:)` actions for each of the three supported types of gestures. The onChange functions first retrieve the gesture entity and its state component, creating a new state component if one doesn't already exist, as shown below:

```
mutating func onChanged(value: EntityTargetValue<DragGesture.Value>) {
    guard canDrag else { return }
    let entity = value.entity

    var state: GestureStateComponent = entity.gestureStateComponent ?? GestureState(

    // ...

}
```

The first time the app calls the function for a particular gesture, the function stores the starting position, orientation, or scale. The drag function looks like this:

```
if state.targetedEntity == nil {
    state.targetedEntity = value.entity
    state.initialOrientation = value.entity.orientation(relativeTo: nil)
}
```

Finally, the function calculates the entity's new position, rotation, or scale by applying the information from the gesture to the starting value stored in the state component. Here's the logic for the rotate gesture:

```swift
let flippedRotation = Rotation3D(angle: rotation.angle,
                                 axis: RotationAxis3D(x: -rotation.axis.x,
                                                      y: rotation.axis.y,
                                                      z: -rotation.axis.z))
let newOrientation = state.startOrientation.rotated(by: flippedRotation)
entity.setOrientation(.init(newOrientation), relativeTo: nil)
```

Create a RealityView extension

To connect the SwiftUI gestures to the component, this sample uses an extension on `RealityView` that contains functions that send the gesture information to the component. For example, here's the gesture property that forwards the drag gesture events to the component:

```swift
/// Builds a drag gesture.
var dragGesture: some Gesture {
    DragGesture()
        .targetedToAnyEntity()
        .useGestureComponent()
}
```

The extension also contains a function that installs all three of the gestures to a `RealityView` at once:

```swift
/// Apply this to a `RealityView` to pass gestures on to the component code.
func installGestures() -> some View {
    simultaneousGesture(dragGesture)
        .simultaneousGesture(magnifyGesture)
        .simultaneousGesture(rotateGesture)
}
```

Install the gestures on the RealityView

To forward the gesture information to the entity components, the app calls the `installGestures()` function on the `RealityView` returned from the initializer.

```swift
RealityView { content in
    // Add the initial RealityKit content.
    if let scene = try? await Entity(named: "Scene", in: realityKitContentBundle) {
        content.add(scene)
    }
} update: { content in

}
.installGestures()
```

Add the gesture component to entities

Once the app installs the gestures on the `RealityView`, people can manipulate any entity containing a `GestureComponent`. This sample uses a Reality Composer Pro scene to add a `GestureComponent` to each of its entities. It also adds an `InputTargetComponent` and a `CollisionComponent` to those entities, because all three components are necessary for the entity to support gestures. Each of the four entities in the sample's Reality Composer scene support a different combination of gestures.

Instead of adding the component in Reality Composer Pro, the app could add the components to entities in code, like this:

```
var component = GestureComponent()
component.canDrag = true
component.canScale = false
component.canRotate = true
myEntity.components.set(component)
```

See Also

Scene content

{} Hello World

Use windows, volumes, and immersive spaces to teach people about the Earth.

{} Enabling video reflections in an immersive environment

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

{} Creating a spatial drawing app with RealityKit

Use low-level mesh and texture APIs to achieve fast updates to a person's brush strokes by integrating RealityKit with ARKit and SwiftUI.

{} Generating interactive geometry with RealityKit

Create an interactive mesh with low-level mesh and low-level texture.

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

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

{} Responding to gestures on an entity

Respond to gestures performed on RealityKit entities using input target and collision components.

Models and meshes

Display virtual objects in your scene with mesh-based models.

Materials, textures, and shaders

Apply textures to the surface of your scene’s 3D objects to give each object a unique appearance.

Anchors

Lock virtual content to the real world.

Lights and cameras

Control the lighting and point of view for a scene.

Content synchronization

Synchronize the contents of entities locally or across the network.

Audio

Create personalized and realistic spatial audio experiences.

Videos

Present videos in your RealityKit experiences.

Images

Present images and spatial scenes in your RealityKit experiences.