Sample Code

Combining 2D and 3D views in an immersive app

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

Download

visionOS 2.0+ | Xcode 16.0+

Overview

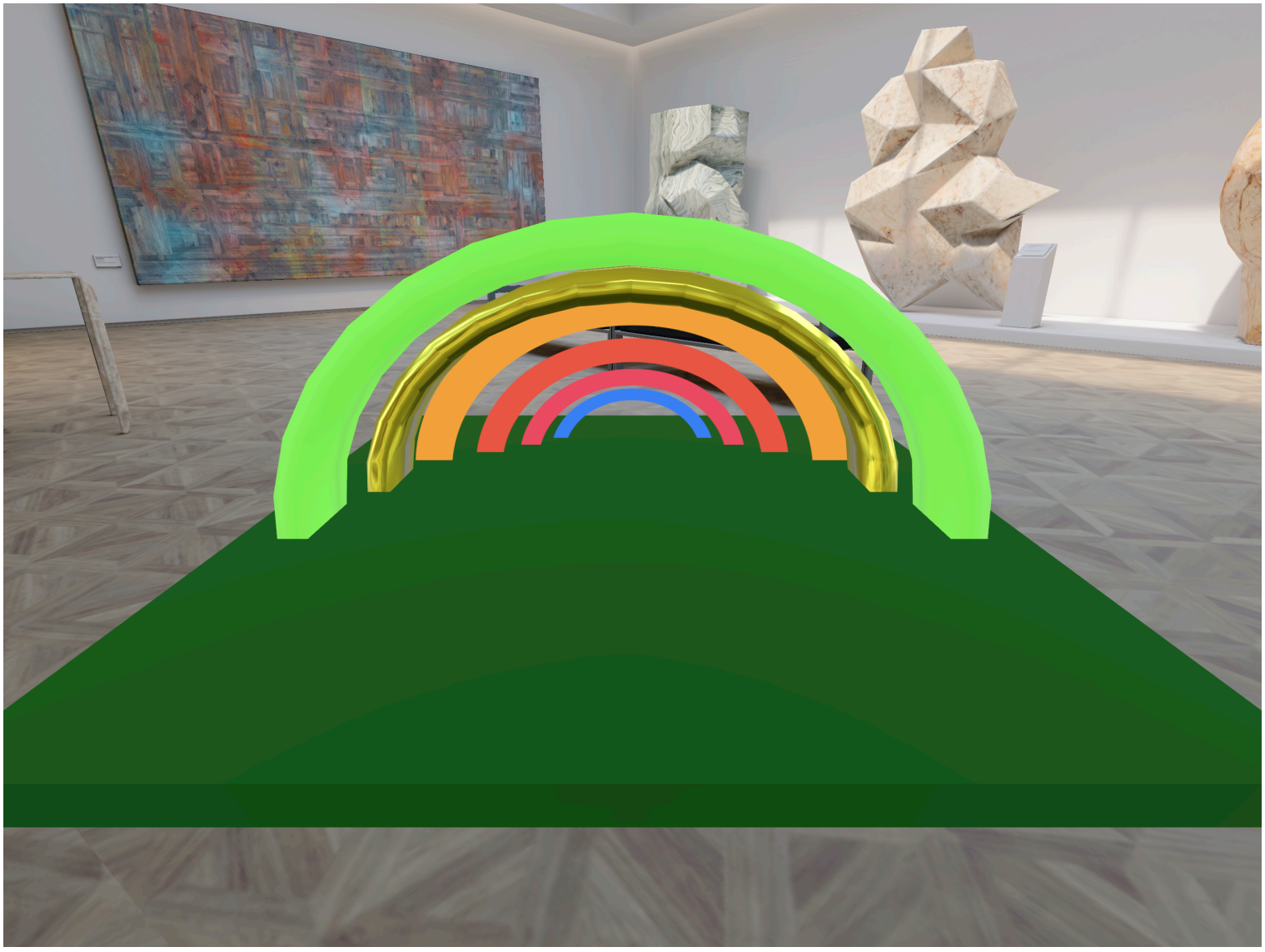
To demonstrate how you can integrate any 2D content into your 3D app, this sample code project uses a variety of frameworks to create both 2D and 3D views, and places them relative to each other in an immersive space. It also illustrates how to position an attachment at the location of a tap gesture.

The rainbow that appears in the sample app contains two USDZ models and four `RealityViewAttachments`.

- The green arch is a USDZ file from Reality Composer Pro with a custom shader graph material.
- The yellow arch is a USDZ file from Reality Composer Pro with a programmatically created simple material.
- The orange arch is a reality view attachment containing a `UIView` in a `UIViewRepresentable` with a custom 2D arc shape.
- The red arch is a reality view attachment containing a `UIView` in a `UIViewRepresentable` with a custom 2D arc shape.
- The pink arch is a reality view attachment containing a SwiftUI `View` with a custom 2D arc shape.
- The blue arch is a reality view attachment containing a SwiftUI `View` with a custom 2D arc shape.

The app loads the 3D assets from Reality Composer Pro as a `ModelEntity` in a `RealityView`, and creates a reality view attachment for each of the 2D arches to attach them to the view.

The cloud attachments at the locations of tap gestures are `RealityViewAttachments` containing `Text` with an SF Symbols image.



Load and configure entities from Reality Composer Pro

This sample creates 3D assets in an asset creator and imports them into Reality Composer Pro as `.usdc` files. See doc://com.apple.documentation/documentation/visionos/designing-realitykit-content-with-reality-composer-pro for more information.

The app then configures the appearance of the `ModelEntity` by setting the material of the `ModelComponent`, which is the `Component` that affects appearance. The following code example demonstrates loading a model and configuring the material:

```
/// Creates an entity from the data model for each Reality Composer Pro asset.
func createEntity(for item: EntityData) async -> Entity {
```

```
    // Load the entity from Reality Composer Pro.
    let realityComposerEntity = try! await Entity(named: item.title, in: realityKitC

    // Find the model component entity and model component.
    guard
        let modelEntity = realityComposerEntity.findEntity(named: item.title),
        var modelComponent = modelEntity.components[ModelComponent.self]
    else {
        return Entity()
    }

    // Set the material if it has a simple material.
    if let material = item.simpleMaterial {
        modelComponent.materials = [material]
    }

    // Set the model component.
    modelEntity.components.set(modelComponent)

    return modelEntity
}
```

Create attachments that contain SwiftUI views

The sample includes the remaining four arches as reality view attachments by creating attachments of various types in the `attachments` closure of a reality view instance. These types include both SwiftUI and UIKit to exemplify how to use any framework in your visionOS app.

```
// Iterate over the attachments array and create the various arches.
ForEach(rainbowModel.archAttachments) { entity in
    // Create an attachment with an ID that the `update` closure references.
    Attachment(id: "\(entity.title.rawValue)ArchAttachmentEntity") {
        createArchAttachment(for: entity.title)
    }
}
```

```
/// Creates the arch view for each attachment based on the color.
@ViewBuilder func createArchAttachment(for arch: ArchAttachmentColor) -> some View {
        switch arch {
        case .blue:
            SwiftUIArcView(color: .blue)
```

```
        case .orange:
            UIViewArcViewRep(color: .orange)
        case .pink:
            SwiftUIArcView(color: .pink)
        case .red:
            CALayerArcViewRep(color: .red)
    }
}
```

Attachments can contain views from other frameworks that adopt the `UIViewRepresentable` protocol.

Add and position entity attachments

The sample loads the attachments as reality view attachments in the `update` closure of the reality view. If there's an existing attachment for an `id`, the sample adds the attachment entity as a subentity of the plane entity to display it in the scene, and then configures the scale and position.

```
// Add and configure attachments.
for viewAttachmentEntity in rainbowModel.archAttachments {

    // Check whether there's an attachment.
    if let attachment = attachments.entity(for: "\(viewAttachmentEntity.title)ArchAt

        attachment.name = viewAttachmentEntity.title.rawValue

        // Add it as a subentity of the plane.
        plane?.addChild(attachment)

        // Set the scale and position.
        attachment.scale = viewAttachmentEntity.scale
        attachment.setPosition(viewAttachmentEntity.position, relativeTo: yellowArch
    }
}
```

This method sets the scales and positions for each attachment by using the yellow arch's bounding box. This ensures each arch is smaller and further back than the previous. The app applies these scale and position values to each entity in the `update` closure as the code example below shows:

```
/// Updates the array containing the scale and position for each attachment entity.
func scaleAndPositionArches(yellowArchSize: BoundingBox) {
    // MARK: - Scaling properties

    // Set the x scale to be the same as the yellow arch.
    // Set the y scale to be double the yellow arch to account for the larger frame
    var archScale = SIMD3(x: yellowArchSize.extents.x, y: yellowArchSize.max.y * 2,

    // MARK: - Positioning properties

    // Set the y position to be the same as the yellow arch.
    let yPosition = yellowArchSize.min.y

    // Set the z position to be 0.1 meters back.
    var zPosition: Float = -0.1
    var position = SIMD3(x: 0, y: yPosition, z: zPosition)

    for (index, attachment) in rainbowModel.archAttachments.enumerated() {

        // Push the arch back by 0.1 meters.
        zPosition -= 0.1
        position.z = zPosition

        // Update the attachments in the view attachment array to include position a
        rainbowModel.archAttachments[index] = ArchAttachment(title: attachment.title

        // Scale the next attachment to be 75% of the size of the previous arch.
        archScale *= 3 / 4
    }
}
```

Position attachments at the tapped location

Follow these steps to add attachments to RealityKit entities and position them at the tapped location. Ensure that your entities have both an `InputTargetComponent` and a `CollisionComponent`.

```
/// Sets the components necessary for hover and tap gestures.
func configureForTapGesture(entity: Entity) async {
    // Set the hover effect component.
    entity.components.set(HoverEffectComponent())
```

```
    // Find the `ModelComponent` to get the mesh and create a static mesh in the sha
    guard let modelComponent = entity.components[ModelComponent.self] else { return
    let entityMesh = modelComponent.mesh
    let shapeResource = try! await ShapeResource.generateStaticMesh(from: entityMesh
    entity.components.set(CollisionComponent(shapes: [shapeResource]))

    // Set the input target component.
    entity.components.set(InputTargetComponent())
}
```

Add a SpatialTapGesture to the RealityView and make sure it uses targetedToAnyEntity(), or specify which entities to target with targetedToEntity(_:). Then use convert(_:from:to:) to convert the location of the tap gesture from the local coordinate space of the entity to the scene's coordinate space.

```
.simultaneousGesture(
    SpatialTapGesture()
        .targetedToAnyEntity()
        .onEnded { value in
            // Convert the tap location to the scene's coordinate space.
            var location3D = value.convert(value.location3D, from: .local, to: .scen
            // Move the z index forward to ensure it doesn't overlap with the entity
            location3D.z += 0.02

            // You don't need to set the position of attachments on entities relativ
            // The system handles this with the location conversion.
            rainbowModel.tapAttachments.append(CloudTapAttachment(position: locatior
        }
)
```

Create the attachment in the attachments closure by iterating over the array of attachments.

```
// Iterate over the tap attachments and provide content for each.
ForEach(rainbowModel.tapAttachments) { cloud in
    Attachment(id: cloud.position) {
        Image(systemName: "cloud.fill")
    }
}
```

Finally, add each attachment in the update closure by iterating over the array of attachments and setting their stored position and root entity.

```
for cloud in rainbowModel.tapAttachments {
    if let cloudEntity = attachments.entity(for: cloud.position) {
        // Scale the attachment larger and add it.
        cloudEntity.scale = [5, 5, 5]
        cloudEntity.name = "\(cloud.position)tapEntity"
        root.addChild(cloudEntity)

        // Set the position of the attachment.
        cloudEntity.setPosition(cloud.position, relativeTo: cloud.parent)
    }
}
```

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.

{} Transforming RealityKit entities using gestures

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

{} 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.