Sample Code

Diorama

Design scenes for your visionOS app using Reality Composer Pro.

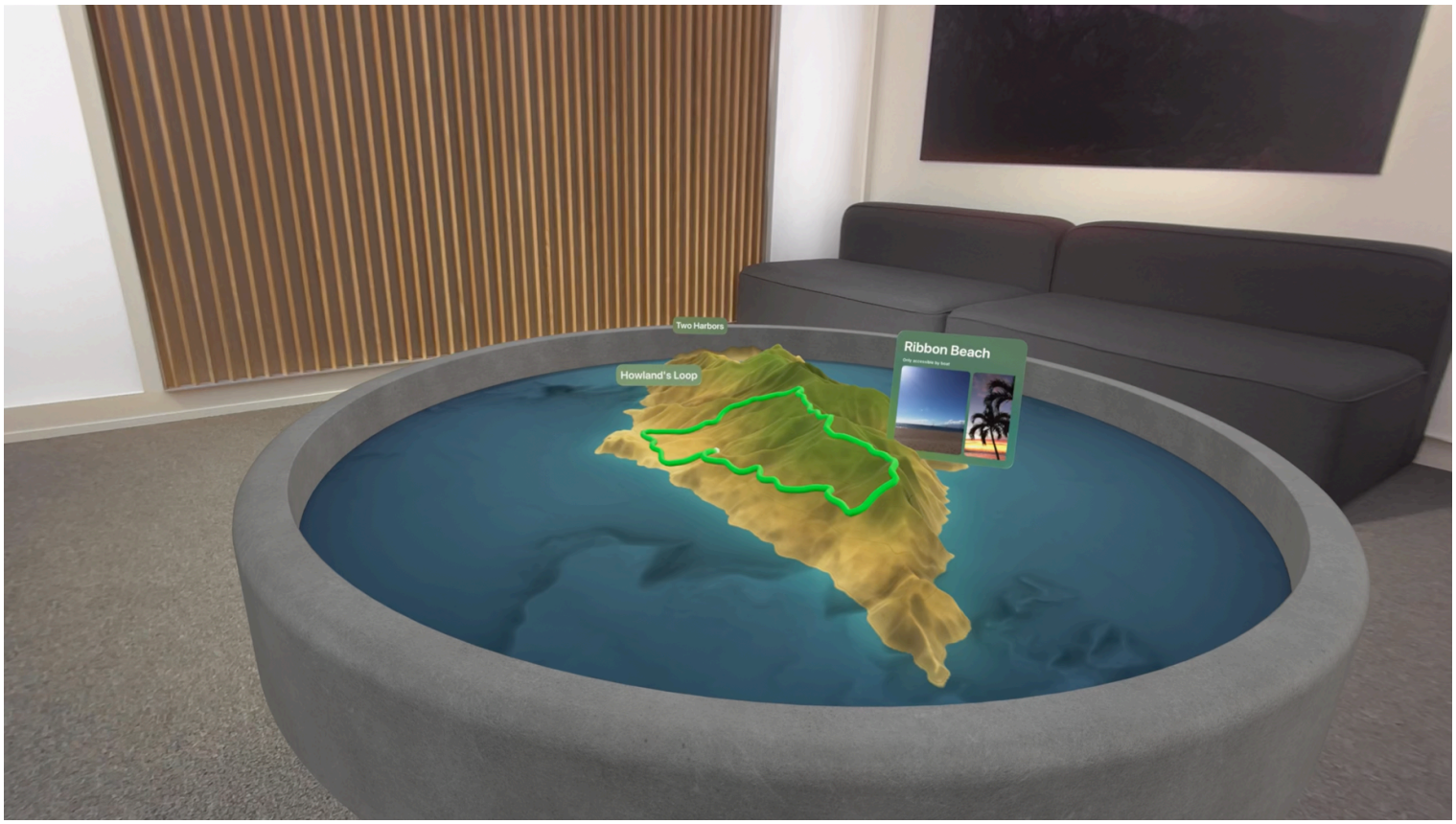
Download

visionOS 2.0+ | Xcode 16.1+

Overview

Use Reality Composer Pro to compose, edit, and preview RealityKit content for your visionOS app. In your Reality Composer Pro project, you can create one or more scenes, each of which contains a hierarchy of virtual objects called entities that your app can efficiently load and display.

In addition to helping you compose entity hierarchies, Reality Composer Pro also gives you the ability to add and configure components — even custom components that you've written — to the entities in your scenes.



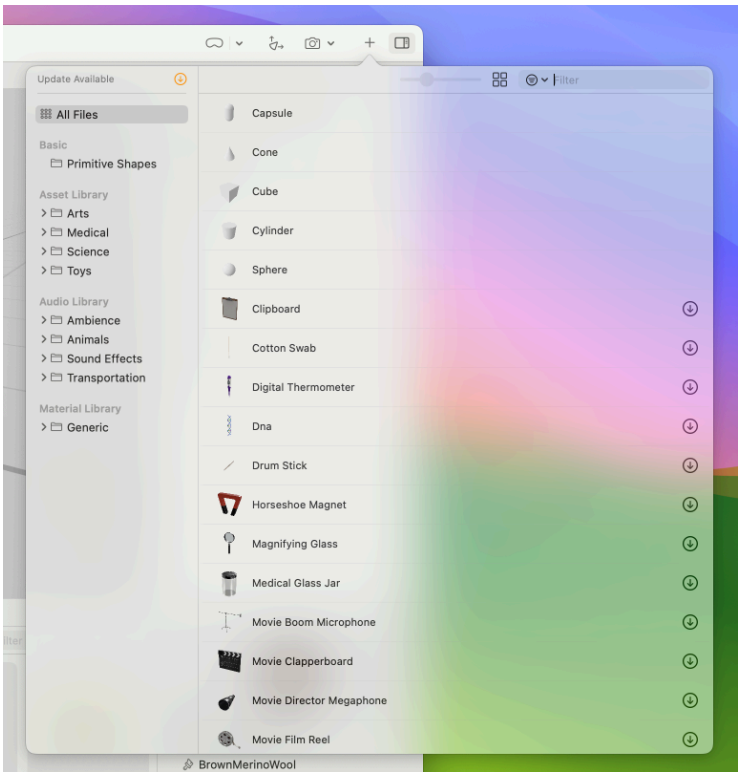
Play ⊙

You can also design the visual appearance of entities using *Shader Graph*, a node-based visual tool for creating RealityKit materials. Shader Graph gives you a tremendous amount of control over the surface details and shape of entities. You can even create animated materials and dynamic materials that change based on the state of your app or user input.

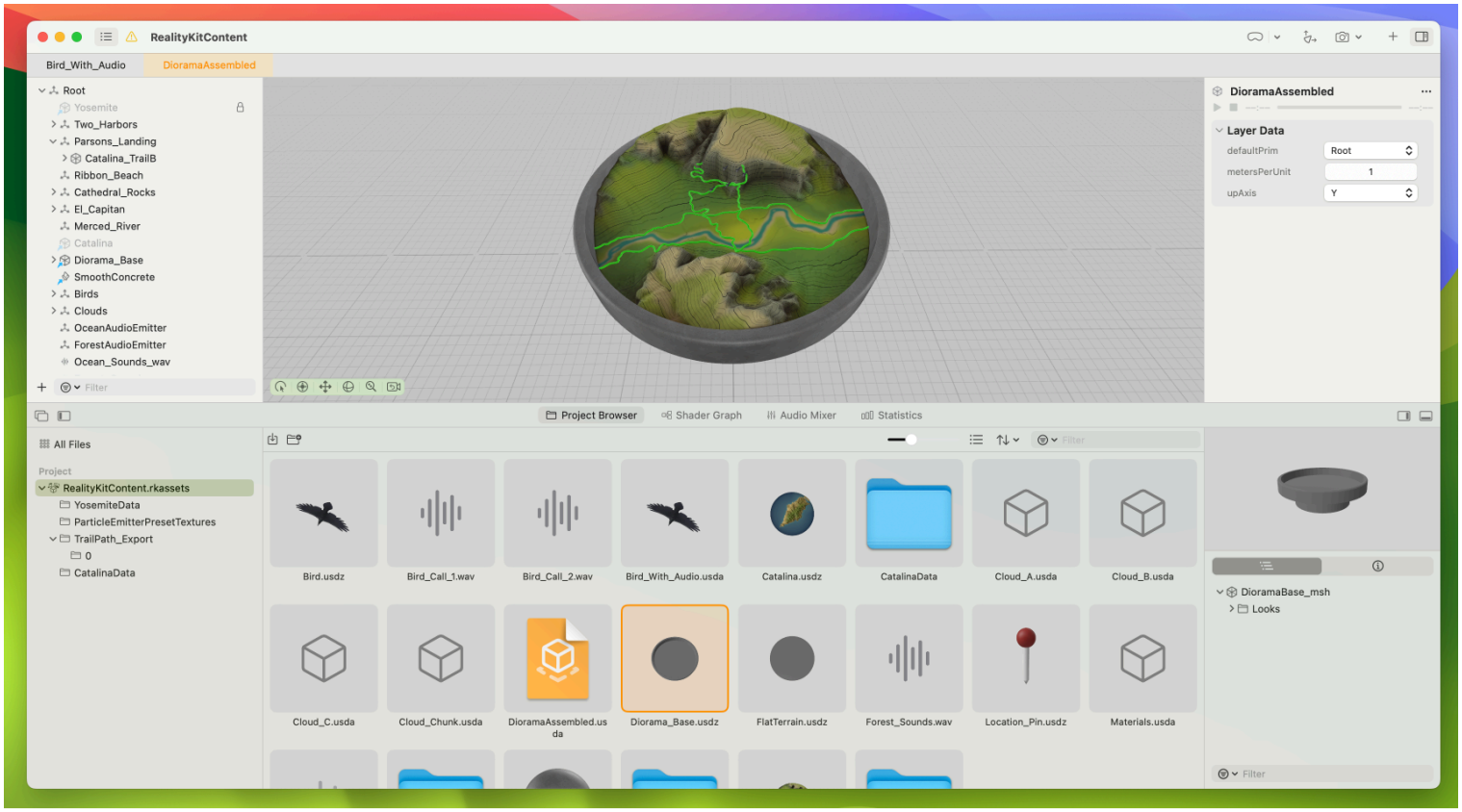
Diorama demonstrates many of RealityKit and Reality Composer Pro's features. It displays an interactive, virtual topographical trail map, much like the real-world dioramas you find at trailheads and ranger stations in national parks. This virtual map has points of interest you can tap to bring up more detailed information. You can also smoothly transition between two trail maps: Yosemite and Catalina Island.

Import assets for building the scene

Your Reality Composer Pro project must contain assets, which you use to compose scenes for your app. Diorama's project has several assets, including 3D models like the diorama table, trail map, some birds and clouds that fly over the map, and a number of sounds and images. Reality Composer Pro provides a library of 3D models you can use. Access the library by clicking the Add (+) button on the right side of the toolbar. Selecting objects from the library imports them into your project.



Diorama uses custom assets instead of the available library assets. To use custom assets in your own Reality Composer Pro scenes, import them into your project in one of three ways: by dragging them to Reality Composer Pro's project browser, using File > Import from the File menu, or copying the assets into the `.rkassets` bundle inside your project's Swift package.

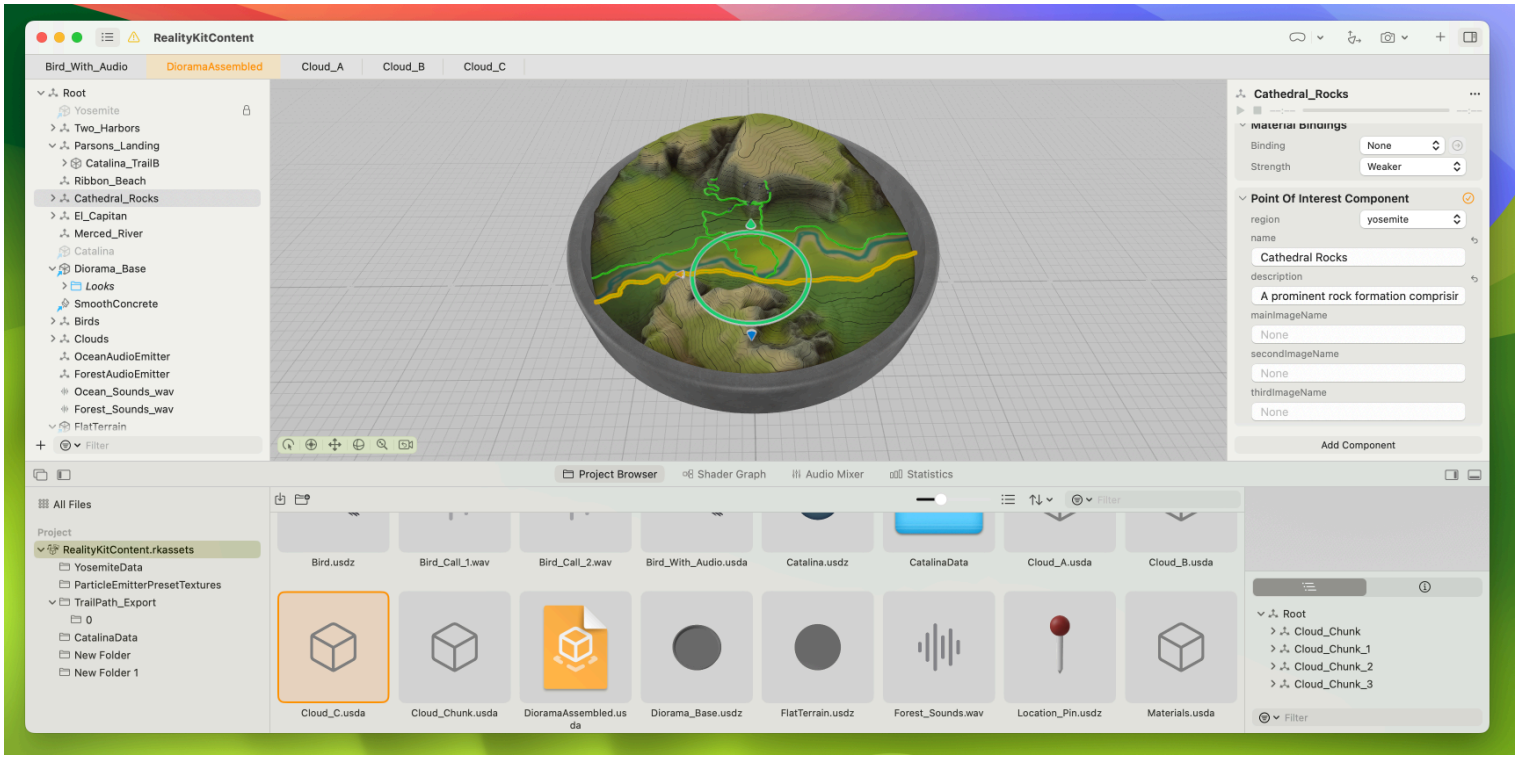


> **Note**
>
> Although you can still load USDZ files and other assets directly in visionOS, RealityKit compiles assets in your Reality Composer Pro project into a binary format that loads considerably faster than loading from individual files.

Create scenes containing the app's entities

A single Reality Composer Pro project can have multiple scenes. A *scene* is an entity hierarchy stored in the project as a `.usda` file that you can load and display in a `RealityView`. You can use Reality Composer's scenes to build an entire RealityKit scene, or to store reusable entity hierarchies that you can use as building block for composing scenes at runtime — the approach Diorama uses. You can add as many different scenes to your project as you need by selecting File > New > Scene, or pressing ⌘N.

At the top of the Reality Composer Pro window, there's a separate tab for every scene that's currently open. To open a scene, double-click the scene's `.usda` file in the project browser. To edit a scene, select its tab, and make changes using the hierarchy viewer, the 3D view, and the inspector.

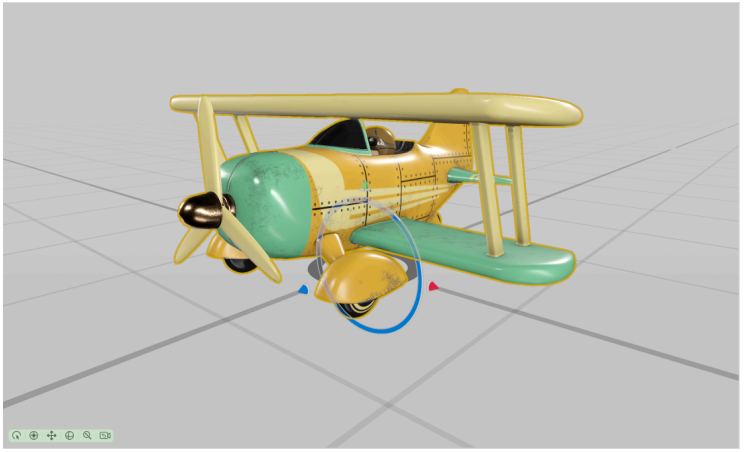


Add assets to your scenes

RealityKit can only include entities in a scene, but it can't use every type of asset that Reality Composer Pro supports as an entity. Reality Composer Pro automatically turns some assets, like

3D models, into an entity when you place them in a scene. It uses other assets indirectly. It uses image files, for example, primarily to define the surface details of model entities.

Diorama uses multiple scenes to group assets together and then, at runtime, combines those scenes into a single immersive experience. For example, the diorama table has its own scene that includes the table, the map surface, and the trail lines. There are separate scenes for the birds that flock over the table, and for the clouds that float above it.



To add entities to a scene, drag assets from the project browser to the scene's hierarchy view or 3D view. If the asset you drag is a type that can be represented as an entity, Reality Composer Pro adds it to your scene. You can select any asset in the scene hierarchy or the 3D view and change its location, rotation, and scale using the inspector on the right side of the window or the manipulator in the 3D view.

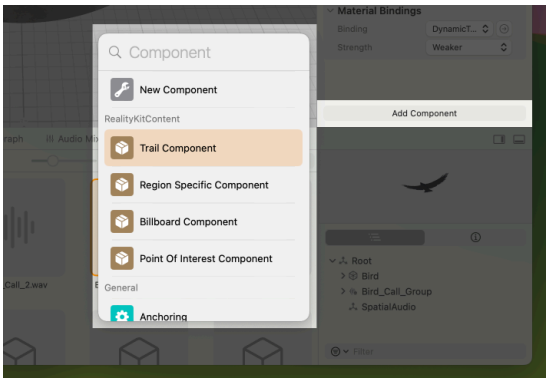
Add components to entities

RealityKit follows a design pattern called Entity Component System (ECS). In an ECS app, you store additional data on an entity using components and can implement entity behavior by writing systems that use the data from those components. You can add and configure components to entities in Reality Composer Pro, including both shipped components like `PhysicsBodyComponent`, and custom components that you write and place in the Sources folder of your Reality Composer Pro Swift package. You can even create new components in Reality Composer Pro and then edit them in Xcode. For more information about ECS, see Understanding the modular architecture of RealityKit.

Diorama uses custom components to identify which transforms are points of interest, to mark the birds so the app can make sure they flock together, and to control the opacity of entities that are specific to just one of the two maps.

To add a component to an entity, select that entity in the hierarchy view or 3D view. At the bottom right of the inspector window, click on the Add Component button. A list of available components appears and the first item in that list is New Component. This item creates a new component class, and optionally a new system class, and adds the component to the selected entity.

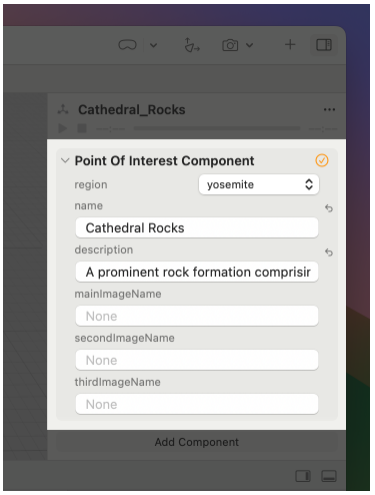
If you look at the list of components, you see the `PointOfInterestComponent` that Diorama uses to indicate which transforms are points of interest. If the selected entity doesn't already contain a `PointOfInterestComponent`, selecting that adds it to the selected entity. Each entity can only have one component of a specific type. You can edit the values of the existing component in the inspector, which changes what shows up when you tap that point of interest in the app.



Use transforms to mark locations

In Reality Composer Pro, a *transform* is an empty entity that marks a point in space. A transform contains a location, rotation, and scale, and its child entities inherit those. But, transforms have no visual representation and do nothing by themselves. Use transforms to mark locations in your scene or organize your entity hierarchy. For example, you might make several entities that need to move together into child entities of the same transform, so you can move them together by moving the parent transform.

Diorama uses transforms with a `PointOfInterestComponent` to indicate points of interest on the map. When the app runs, those transforms mark the location of the floating placards with the name of the location. Tapping on a placard expands it to show more detailed information. To turn transforms into an interactive view, the app looks for a specific component on transforms called a `PointOfInterestComponent`. Because a transform contains no data other than location, orientation, and scale, it uses this component to hold the data the app needs to display on the placards. If you open the `DioramaAssembled` scene in Reality Composer Pro and click on the transform called `Cathedral_Rocks`, you see the `PointOfInterestComponent` in the inspector.



Load a scene at runtime

To load a Reality Composer Pro scene, use `load(named:in:)`, passing the name of the scene you want to load and the project's bundle. Reality Composer Pro Swift packages define a constant that provides ready access to its bundle. The constant is the name of the Reality Composer Pro project with "Bundle" appended to the end. In this case, the project is called `RealityKitContent`, so the constant is called `RealityKitContentBundle`. Here's how Diorama loads the map table in the `RealityView` initializer:

```
let entity = try await Entity.load(named: "DioramaAssembled",
                                   in: RealityKitContent.RealityKitContentBundle)
```

The `load(named:in:)` function is asynchronous when called from an asynchronous context. Because the content closure of the `RealityView` initializer is asynchronous, it automatically uses the `async` version to load the scene. Note that when using it asynchronously, you must call it using the `await` keyword.

Create the floating view

Diorama adds a `PointOfInterestComponent` to a transform to display details about interesting places. Every point of interest's name appears in a view that floats above its location on the map. When you tap the floating view, it expands to show detailed information, which the app pulls from the `PointOfInterestComponent`. The app shows these details by creating a SwiftUI view for each point of interest and querying for all entities that have a `PointOfInterestComponent` using this query declared in `ImmersiveView.swift`:

```
static let markersQuery = EntityQuery(where: .has(PointOfInterestComponent.self))
```

In the `RealityView` initializer, Diorama queries to retrieve the points of interest entities and passes them to a function called `createLearnMoreView(for:)`, which creates the view and saves it for display when it's tapped.

```
subscriptions.append(content.subscribe(to: ComponentEvents.DidAdd.self, componentTyp
    createLearnMoreView(for: event.entity)
}))
```

Create attachments for points of interest

Diorama displays the information added to a `PointOfInterestComponent` in a `LearnMoreView`, which it stores as an attachment. *Attachments* are SwiftUI views that are also RealityKit entities and that you can place into a RealityKit scene at a specific location. Diorama uses attachments to position the view that floats above each point of interest.

The app first checks to see if the entity has a component called `PointOfInterestRuntimeComponent`. If it doesn't, it creates a new one and adds it to the entity. This new component contains a value you only use at runtime that you don't need to edit in Reality Composer Pro.

By putting this value into a separate component and adding it to entities at runtime, Reality Composer Pro never displays it in the inspector. The `PointOfInterestRuntimeComponent`

stores an identifier called an *attachment tag*, which uniquely identifies an attachment so the app can retrieve and display it at the appropriate time.

```
struct PointOfInterestRuntimeComponent: Component {
    let attachmentTag: ObjectIdentifier
}
```

Next, Diorama creates a SwiftUI view called a `LearnMoreView` with the information from the `PointOfInterestComponent`, tags that view, and stores the tag in the `PointOfInterestRuntimeComponent`. Finally, it stores the view in an `AttachmentProvider`, which is a custom class that maintains references to the attachment views so they don't get deallocated when they're not in a scene.

```
let tag: ObjectIdentifier = entity.id

let view = LearnMoreView(name: pointOfInterest.name,
                         description: pointOfInterest.description ?? "",
                         imageNames: pointOfInterest.imageNames,
                         trail: trailEntity,
                         viewModel: viewModel)
    .tag(tag)
entity.components[PointOfInterestRuntimeComponent.self] = PointOfInterestRuntimeComp

attachmentsProvider.attachments[tag] = AnyView(view)
```

Display point of interest attachments

Assigning a view to an attachment provider doesn't actually display that view in the scene. The initializer for `RealityView` has an optional view builder called `attachments` that's used to specify the attachments.

```
ForEach(attachmentsProvider.sortedTagViewPairs, id: \.tag) { pair in
    pair.view
}
```

In the `update` closure of the initializer, which RealityKit calls when the contents of the view change, the app queries for entities with a `PointOfInterestRuntimeComponent`, uses the tag from that component to retrieve the correct attachment for it, and then adds that attachment and places it above its location on the map.

```
viewModel.rootEntity?.scene?.performQuery(Self.runtimeQuery).forEach { entity in

    guard let attachmentEntity = attachments.entity(for: component.attachmentTag) el

    if let pointOfInterestComponent = entity.components[PointOfInterestComponent.sel
        attachmentEntity.components.set(RegionSpecificComponent(region: pointOfInter
        attachmentEntity.components.set(OpacityComponent(opacity: 0))
    }

    viewModel.rootEntity?.addChild(attachmentEntity)
    attachmentEntity.setPosition([0, 0.2, 0], relativeTo: entity)
}
```

Create custom materials with Shader Graph

To switch between the two different topographical maps, Diorama shows a slider that morphs the map between the two locations. To accomplish this, and to draw elevation lines on the map, the `FlatTerrain` entity in the `DioramaAssembled` scene uses a *Shader Graph material*. Shader Graph is a node-based material editor that's built into Reality Composer Pro. Shader Graph gives you the ability to create dynamic materials that you can change at runtime. Prior to Reality Composer Pro, the only way to implement a dynamic material like this was to create a `CustomMaterial` and write Metal shaders to implement the necessary logic.

Diorama's `DynamicTerrainMaterialEnhanced` does two things. It draws contour lines on the map based on height data stored in displacement map images, and it also offsets the vertices of the flat disk based on the same data. By interpolating between two different height maps, the app achieves a smooth transition between the two different sets of height data.

When you build Shader Graph materials, you can give them input parameters called *promoted inputs* that you set from Swift code. This allows you to implement logic that previously required writing a Metal shader. The materials you build in the editor can affect both the look of an entity using the custom surface output node, which equates to writing Metal code in a fragment shader, or the position of vertices using the geometry modifier output, which equates to Metal code running in a vertex shader.

Node graphs can contain *subgraphs*, which are similar to functions. They contain reusable sets of nodes with inputs and outputs. Subgraphs contain the logic to draw the contour lines and the logic to offset the vertices. Double-click a subgraph to edit it. For more information about building materials using Shader Graph, see Explore Materials in Reality Composer Pro.

Update the Shader Graph material at runtime

To change the map, `DynamicTerrainMaterialEnhanced` has a promoted input called `Progress`. If that parameter is set to `1.0`, it displays Catalina Island. If it's set to `0`, it displays Yosemite. Any other number shows a state in transition between the two. When someone manipulates the slider, the app updates that input parameter based on the slider's value.

> **Important**
>
> Shader Graph material parameters are case-sensitive. If the capitalization is wrong, your code won't actually update the material.

The app sets the value of the input parameter in a function called `handleMaterial()` that the slider's `.onChanged` closure calls. That function retrieves the `ShaderGraphMaterial` from the terrain entity and calls `setParameter(name:value:)` on it.

```
private func handleMaterial() {
    guard let terrain = viewModel.rootEntity?.terrain,
            let terrainMaterial = terrainMaterial else { return }
    do {
        var material = terrainMaterial
        try material.setParameter(name: materialParameterName, value: .float(viewMod

        if var component = terrain.modelComponent {
            component.materials = [material]
            terrain.components.set(component)
        }

        try terrain.update(shaderGraphMaterial: terrainMaterial, { m in
            try m.setParameter(name: materialParameterName, value: .float(viewModel.
        })
    } catch {
        print("problem: \(error)")
    }
}
```

See Also

Related samples

{} Hello World

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

{} Destination Video

Leverage SwiftUI to build an immersive media experience in a multiplatform app.

{} Happy Beam

Leverage a Full Space to create a fun game using ARKit.

Related articles

Adding 3D content to your app

Add depth and dimension to your visionOS app and discover how to incorporate your app's content into a person's surroundings.

Understanding the modular architecture of RealityKit

Learn how everything fits together in RealityKit.

{} Composing interactive 3D content with RealityKit and Reality Composer Pro

Build an interactive scene using an animation timeline.

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).

Related videos

Meet Reality Composer Pro

Explore materials in Reality Composer Pro

Work with Reality Composer Pro content in Xcode