Sample Code

BOT-anist

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

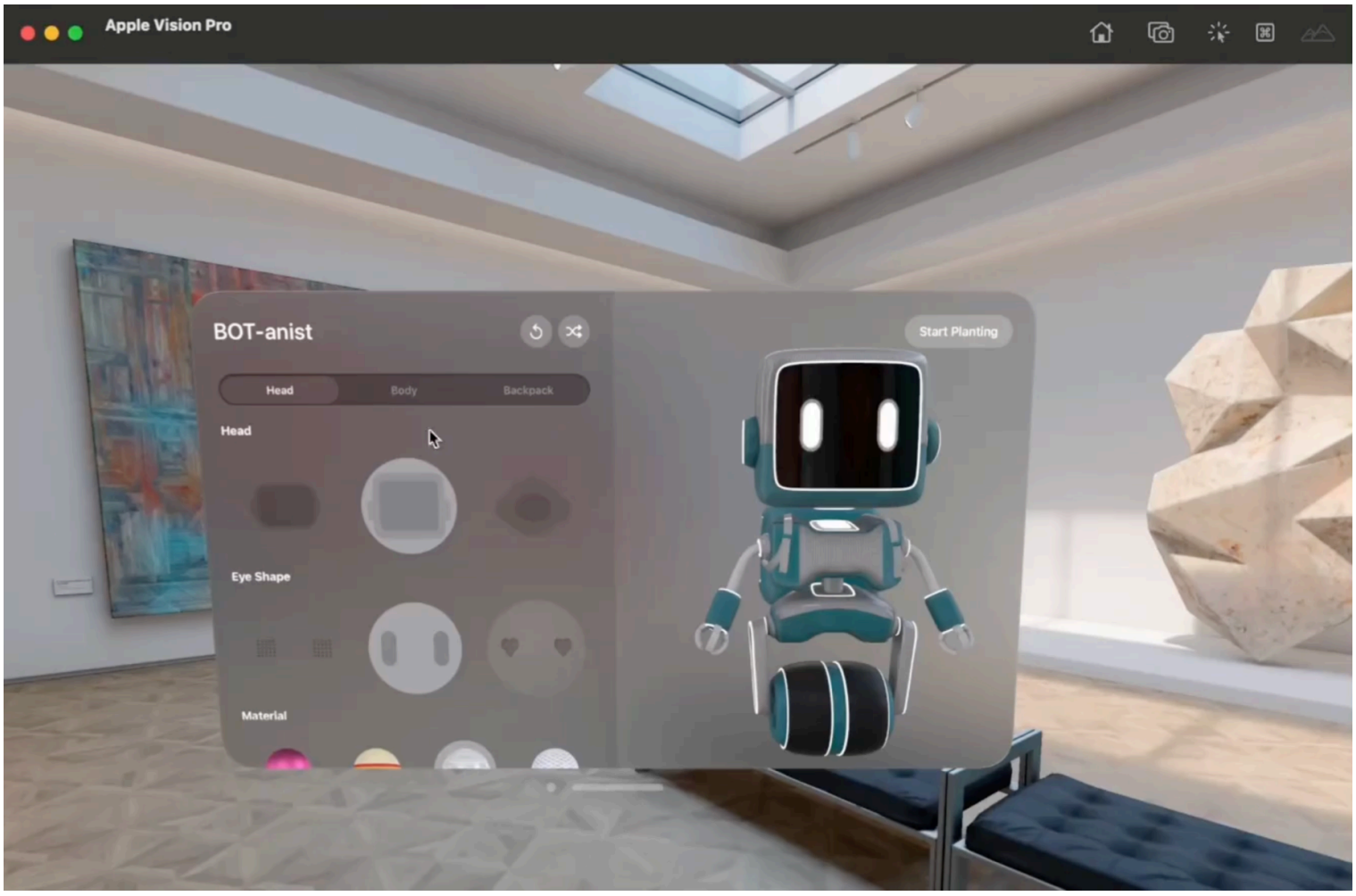
Download

iOS 18.0+ | iPadOS 18.0+ | macOS 15.0+ | visionOS 2.0+ | Xcode 16.0+

Overview

BOT-anist is a game-like experience where you build a custom robot botanist by selecting from a variety of color and shape options, and then guide your robot around a futuristic greenhouse to plant alien flowers. This app demonstrates how to build an app for iOS, iPadOS, macOS, and visionOS using a single shared Xcode target and a shared Reality Composer Pro project.

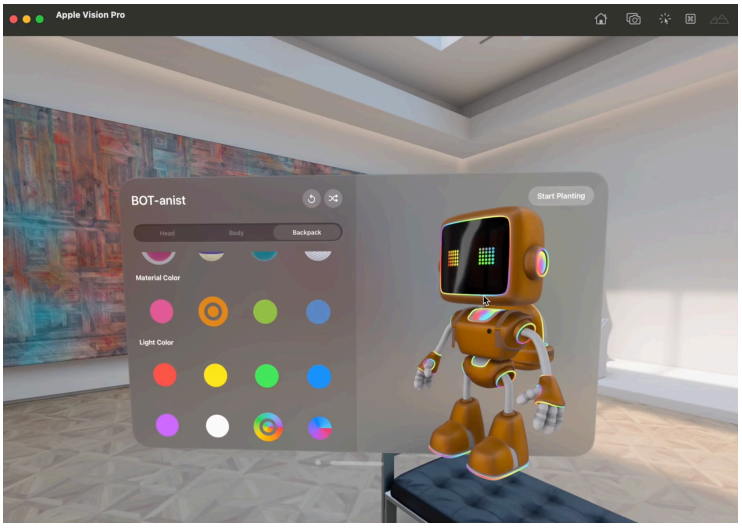
This sample shows off a number of RealityKit and visionOS features, including volume ornaments, dynamic lights and shadows, animation library components, and vertex animation using blend shapes. It also demonstrates how to set a volume's default size and enable user resizing of volumes.



Play

Customize the robot and explore

When the app launches, you see a window that contains your robot and a number of different user interface elements you can use to customize it. You can change the shape of your robot's head, backpack, and body, as well as the material and color scheme for each part. You can also change the color of the lights for each piece.

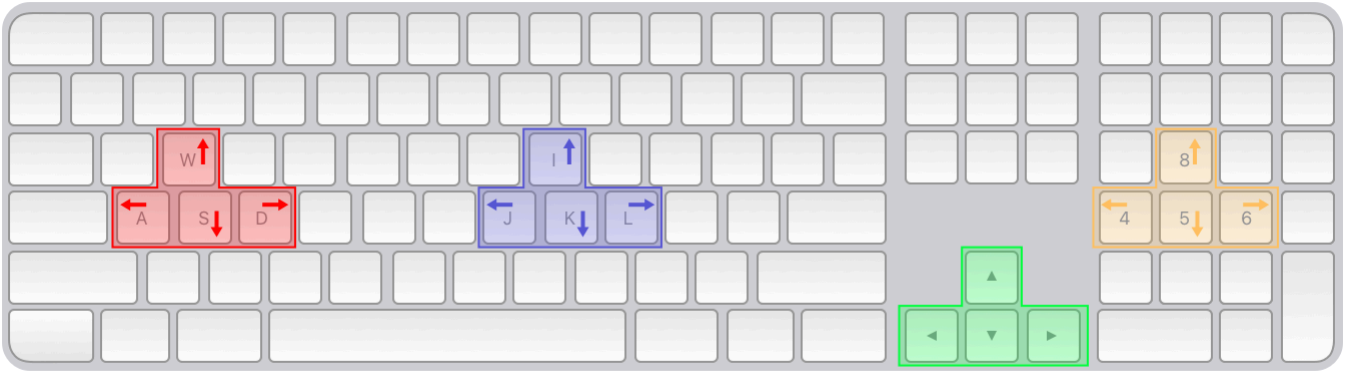


Play

To get a better look at your robot while customizing it, spin it using a drag gesture on iOS and visionOS, or by dragging it with your mouse in macOS.

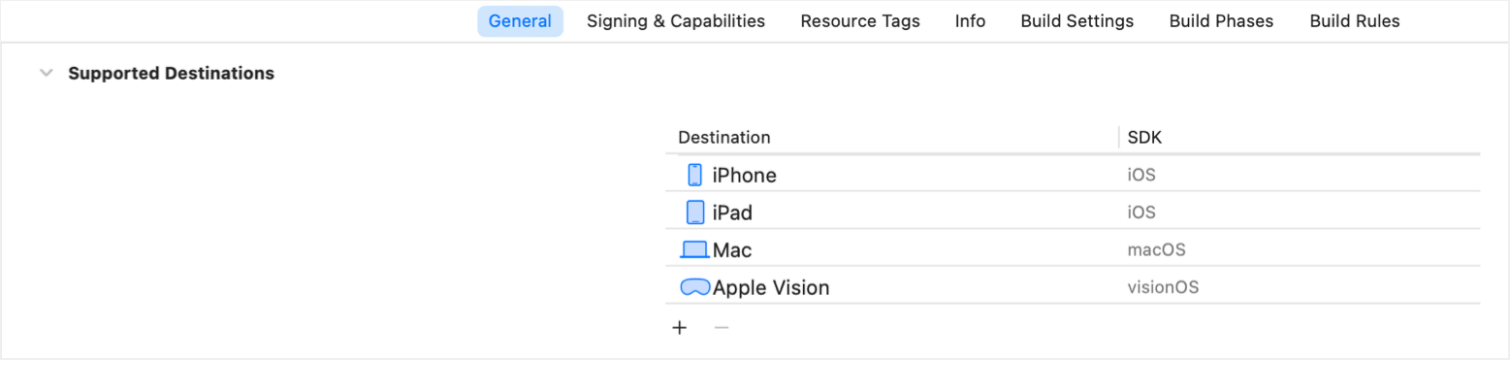
When you're happy with the look of your robot botanist, tap or click the Start Planting button to send the robot to explore its futuristic greenhouse. In visionOS, BOT-anist displays the greenhouse in a resizable 3D volume. In iOS and macOS, it appears in the same window as the customization tools. There are three illuminated planters in different colors on the floor of the greenhouse. Move

your robot around using drag gestures or keyboard controls to plant flowers in each one. When using the keyboard to control the robot, you have the option to use the traditional WASD key combination on QWERTY keyboards, as well as the right-handed equivalent (IJKL). You can also use the arrow keys and, if using an extended keyboard, the numeric keypad (8456). You can find the key bindings in `RealityView+KeyboardControls.swift` if you want to change them to an alternative scheme.



Make the project multiplatform

You can now use RealityKit to create multiplatform apps that run in iOS, iPadOS, macOS, and visionOS using `RealityView`. As long as your Xcode project uses only SwiftUI for its user interface, you can convert it to a multiplatform app by navigating to your app target in Xcode and adding the platforms you want to support. You don't need to add a new target or scheme. When you're developing your app, Xcode compiles the right code for the selected destination. When you build your app for distribution, it builds it for all the platforms you select.



There are, however, platform differences you need to take into account in some apps. For example, visionOS uses a different unit scale for RealityKit scenes than other platforms do. Also, different devices have different screen sizes and aspect ratios. To account for these differences, you may want to set the scale and position of the root entity in the `RealityView` using different values.

```
#if os(visionOS)
self.creationRoot.scale = SIMD3<Float>(repeating: 0.23)
```

```
self.creationRoot.position = SIMD3<Float>(x: -0.02, y: -0.175, z: -0.05)
#else
self.creationRoot.scale = SIMD3<Float>(repeating: 0.027)
self.creationRoot.position = SIMD3<Float>(x: -0, y: -0.022, z: -0.05)
#endif
```

In BOT-anist on visionOS, you use a `GeometryReader3D` to position and resize the robot view to fill 80% of the available space

```
let robotVisualBounds = appState.creationRoot.visualBounds(relativeTo: nil)

appState.creationRoot.position = SIMD3<Float>.zero

// Adjust the model's position on the y-axis to align with the center of the view bo
appState.creationRoot.position.y -= appState.creationRoot.visualBounds(relativeTo: r

// Adjust the robot to be positioned against the window, rather than in the center o
appState.creationRoot.position.z -= viewBounds.max.z / 2
appState.creationRoot.position.z += appState.creationRoot.visualBounds(relativeTo: r

/// The base size of the model when the scale is 1.
let baseExtents = robotVisualBounds.extents / appState.creationRoot.scale

/// The scale required for the model to fit the bounds of 80% of the volumetric wind
let scaleToFitHeight = Float(viewBounds.extents.y * 0.8) / baseExtents.y
let scaleToFitWidth = Float(viewBounds.extents.x * 0.8) / baseExtents.x

// Apply the scale to the model to fill the full size of the window.
appState.creationRoot.scale = SIMD3<Float>(repeating: min(scaleToFitWidth, scaleToFi
```

Set up the window groups

BOT-anist sets up two window groups because it uses both a window and a volume in visionOS, but runs the entire app in a single window on its other supported platforms. The first window group uses the default platform window style, which creates a standard window for the platform it's running on. In visionOS only, the app configures this window group to dismiss the other window group, which holds the 3D volume, when this window appears. That ensures the window and volume are never visible at the same time.

```
WindowGroup(id: "RobotCreation") {
    ContentView()
```

```
        .environment(appState)
    .onAppear {
        #if os(visionOS)
            dismissWindow(id: "RobotExploration")
        // ...
        #endif
    }
}
```

The app class contains a second window group with a `volumetric` style to hold the greenhouse in visionOS. This second window group uses platform conditionals to ensure that it only compiles for visionOS. The default window style behavior in visionOS for 2D windows is `dynamic`, which means the window changes its size as it changes its distance to the player to ensure the window is always a good size for them to interact with.

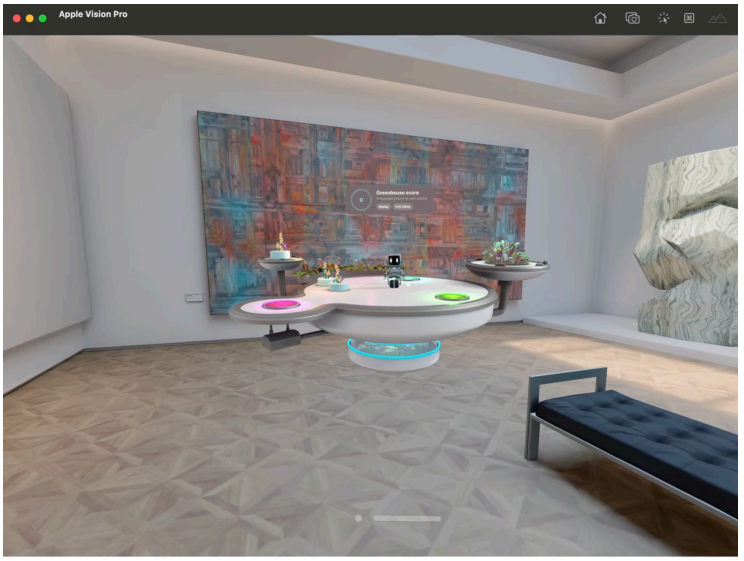
Volumes, on the other hand, default to a fixed window style, which means the farther away from the player the volume is, the smaller it appears. This is often the desired behavior for volumes because you want the virtual contents to blend in with real-world perspective. In this case, however, the player needs to interact with the greenhouse features much like they do with the UI elements in a 2D window. BOT-anist changes the default scaling of the volume to `dynamic` so it stays usable even if the player moves away from it.

```
#if os(visionOS)
WindowGroup(id: "RobotExploration") {
    GeometryReader3D { geometry in
        ExplorationView()
            .volumeBaseplateVisibility(.visible)
            .environment(appState)
            .scaleEffect(geometry.size.width / initialVolumeSize.width)
            .ornament(attachmentAnchor: .scene(.topBack)) {
                OrnamentView()
                    .environment(appState)
            }
            .onAppear {
                dismissWindow(id: "RobotCreation")
            }
            .onChange(of: geometry.size) { _, newSize in
                appState.robot?.speedScale = Float(newSize.width / initialVolumeSize
            }
    }
}
.windowStyle(.volumetric)
.defaultWorldScaling(.dynamic)
```

```
.defaultSize(initialVolumeSize)
#endif
```

Show the volume's baseplate

visionOS volumes are, by default, user resizable. People can resize them by looking at one of the four bottom corners of the volume, and then pinching and dragging the control that appears.



Play ⊙

BOT-anist uses the default behavior for the volume that displays the greenhouse. To make it more obvious to the player that they can resize the volume, and to give them better visual feedback when doing it, BOT-anist makes the volume's baseplate visible. The *baseplate* is a white, rounded rectangle on the bottom plane of the volume that the app enables by calling `volumeBaseplateVisibility(_:)` on the volume's root view.

```
ExplorationView()
.volumeBaseplateVisibility(.visible)
```

BOT-anist also sets the default size of the volume to make sure it starts large enough for the player to interact with.

```
private var initialVolumeSize: Size3D = Size3D(width: 900, height: 500, depth: 900)

// ...

.defaultSize(initialVolumeSize)
```

> **Note**
>
> To create a volume with fixed style, don't specify a default size. Instead, use `frame(minDepth:idealDepth:maxDepth:alignment:)` on the volume's root view and pass the same value for `minDepth`, `idealDepth`, and `maxDepth`.

By default, when a volume changes size, the size of its contents don't scale with it. BOT-anist's contents do resize with the volume, which it accomplishes using the `scaleEffect(_:anchor:)` modifier.

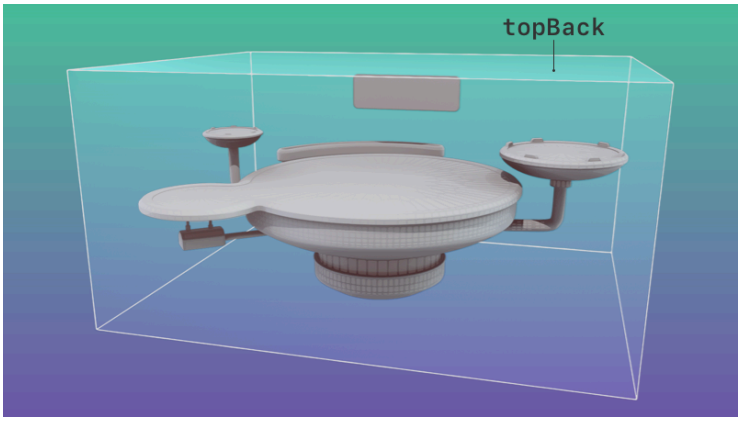
```
.scaleEffect(geometry.size.width / initialVolumeSize.width)
```

When the contents resize relative to the real-world surroundings, it affects the robot's speed of movement, causing it to move too fast when you make the volume smaller and move too slow when you make it larger. To make sure the robot moves at a consistent speed no matter the size of the volume, the window group uses an `onChange(of:initial:_:)` modifier to update the robot's speed based on the volume's size.

```
.onChange(of: geometry.size) { _, newSize in
    appState.robot?.speedScale = Float(newSize.width / initialVolumeSize.width)
}
```

Specify the volume's ornament view

Starting with visionOS 2, you can place ornaments in different locations in the 3D space of a volume. BOT-anist uses an ornament view to display the score, along with buttons for re-starting and for going back to the robot customization screen.



Instead of the default placement, BOT-anist displays the ornament view at the top back. To specify its ornament view, it uses `ornament(visibility:attachmentAnchor:contentAlignment:ornament:)`, and a value of `topBack`, which centers it at the top of the far side of the volume.

```
.ornament(attachmentAnchor: .scene(.topBack)) {
    OrnamentView()
        .environment(appState)
}
```

Create dynamic lights with shadows using Reality Composer Pro

To create dynamic lighting effects with shadows, add lights to your Reality Composer Pro project. To see the lights that BOT-anist uses, open `BOTanistAssets.swift` in Reality Composer Pro and open the scene called `volume.usda`.

After you add lights to your scene, build and run your app to see it with the new lights, including dynamic shadows. If you watch the robot botanist as you move it around the greenhouse, you see that it casts a shadow.

You can use up to eight lights in a RealityKit scene, but lights have a nontrivial performance impact, so use them strategically. Even when using dynamic lights, your entities still receive light from any image-based or environmental lighting your app uses.

Detect viewpoint changes in volumes

In visionOS 2 and later, apps can use `onVolumeViewpointChange(updateStrategy: initial: _:)` to receive updates when the player moves to a different side. When BOT-anist receives an update, it rotates the robot toward the viewer and waves to them at their new location.

Play ⊙

In `ExplorationView.swift`, which is the top-level view in the volume's window group, the app uses `onVolumeViewpointChange(updateStrategy:initial:_:)` to receive updates about which side of the volume is facing the person, and stores the new facing in a property.

```
#if os(visionOS)
// ...
.onVolumeViewpointChange { _, newViewpoint in
    currentViewpoint = newViewpoint
}
#endif
```

The value the app receives is of type `Viewpoint3D`, and it identifies which side of the volume is currently facing the viewer relative to which side was the front face when the app first launched. The code that handles movement input monitors this property. When it detects a viewpoint change and the robot isn't moving, it starts the animations that cause the robot to rotate toward the new front face and wave cheerily.

Animate the robot

BOT-anist contains multiple different body types that players can choose when building their robot, including one that walks, one that rolls, and one that floats. Each of these bodies has a different set of animations, and the app uses a state machine defined in `AnimationState`

`Machine.swift` to keep track of which animation is currently playing, and when and how it transitions to a different animation.

RealityKit can load multiple animations from different USDZ files and store them in an `AnimationLibraryComponent`. As long as two rigged entities have the same joint hierarchy, they can use each other's animations. BOT-anist uses one `AnimationLibraryComponent` per body entity to store the animations for that body type.

At launch, BOT-anist loads each of the different modular parts that players use to build their robot. When it loads the bodies, it creates an `AnimationLibraryComponent` on each loaded entity, then loads and stores one animation per animation state.

```
if part == .body {
    var libComponent = AnimationLibraryComponent()
    let animationDirectory = "Assets/Robot/animations/\(partName)/"
    for animationType in AnimationState.allCases {
        if let rootEntity = try? await Entity(named: "\(animationDirectory)\(partNam
                                              in: BOTanistAssetsBundle) {
            if let animationEntity = await rootEntity.findEntity(named: "rig_grp") {
                if let animationLibraryComponent = await animationEntity.animationLi
                    libComponent.animations[animationType.rawValue] = animationLibra
                }
            }
        }
    }
    await entity.components.set(libComponent)
}
```

When the app transitions to a new animation state, it can find and play the correct animation by retrieving the animation for the current state from the animation library on the body entity that's in the scene.

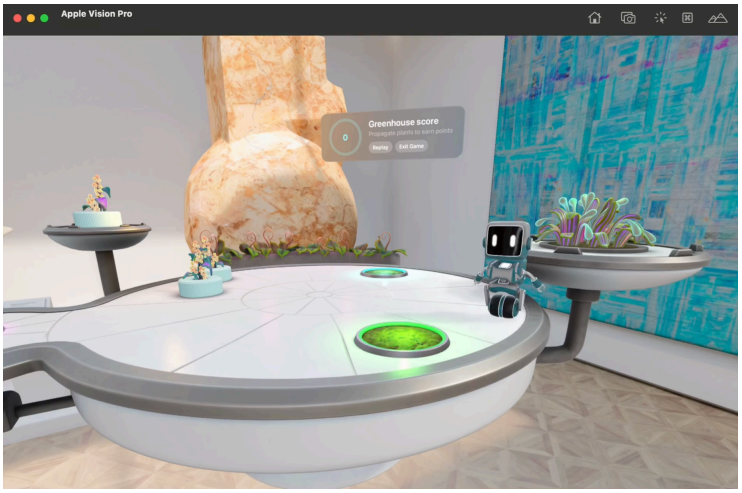
```
guard let anim = body.animationLibraryComponent?.animations[animState.rawValue] else
    fatalError("Didn't find requested animation in library.")
}
```

Animate the plants using blend shapes

While skeletal animations are an incredibly powerful and useful tool, certain types of animations need to move each vertex in the model individually. RealityKit stores vertex-level changes to a model using *blend shapes*, which contain offset data for the model entity's vertices. You can set

each blend shape to a value between `0.0` and `1.0`. Any value other than `0.0` or `1.0` represents a partial state in-between the model's default shape, and the shape contained in that blend shape.

For example, if you have a model of a plant as a sprout, and a shape key representing its fully grown shape, you can set the plant to grow only partially by setting that blend shape to a fractional value. Animating that value over time creates a blend-shape animation, which is how BOT-anist grows the flowers when you plant them.



Play ⊙

To access blend shapes, use `BlendShapeWeightsComponent`. You can create blend shapes and set their values procedurally but, more often, you create blend shapes and blend shape animations using a 3D modeling tool, then store them in the model's USDZ file. RealityKit automatically creates a `BlendShapeWeightsComponent` for any model entity it loads from a USDZ file that contains blend shapes. It also adds any blend shape animations in the USDZ file to the entity's `AnimationLibraryComponent`.

> **Note**
>
> Some software uses different terms when referring to per-vertex offset data. In addition to blend shape, you may also find the same functionality referred to as *morph targets* or *shape keys*. All of these export to USDZ files as blend shapes and work identically.

To animate the plants growing, BOT-anist uses blend shape animations created in a 3D modeling program and stored in the model's USDZ file. It uses the same approach to animate the celebratory dancing the flowers do once the robot has planted them all. Each type of plant has its own blend shapes and blend shape animations to show the plant growing and celebrating.

To play the blend shape animations, the app iterates through entities in the scene that have a `BlendShapeWeightsComponent` and plays the corresponding blend shape weight animation. For example, here's how it generates the grow animation:

```
private func generateGrowAnimationResource(for plantType: PlantComponent.PlantTypeKe
    -> AnimationResource {
    let sceneName = "Assets/plants/animations/\(plantType.rawValue)_grow_anim"
    var ret: AnimationResource? = nil
    do {
```

```
        let rootEntity = try await Entity(named: sceneName, in: BOTanistAssetsBundle
        rootEntity.forEachDescendant(withComponent: BlendShapeWeightsComponent.self)
        if let index = entity.animationLibraryComponent?.animations.startIndex {
            ret = entity.animationLibraryComponent?.animations[index].value
        }
        }
        guard let ret else {
            fatalError("Animation resource unexpectedly nil.")
        }
        return ret
    } catch {
        fatalError("Error: \(error.localizedDescription)")
    }
}
```

When BOT-anist transitions to the greenhouse, it has to make sure that all the growing plants are reset to their initial value. To do that, it manually sets all of the blend shapes to `0.0` except for the one that represents the initial hidden state.

```
guard let modelComponent = entity.modelComponent else {
    fatalError("Entity must be model entity. No ModelComponent found.")
    }
let meshResource = modelComponent.mesh
let blendShapeWeightsMapping = BlendShapeWeightsMapping(meshResource: meshResource)
var blendComponent = BlendShapeWeightsComponent(weightsMapping: blendShapeWeightsMap
blendComponent.weightSet[0].weights = BlendShapeWeights([0, 1, 0, 0, 0, 0, 0])
entity.components.set(blendComponent)
```

Animate the head and backpack

To make the robot customizable, the app combines three separate entities to build it. Each of the three bodies (walking, rolling, floating) is a skeletal mesh with its own unique set of animations. When the app enters the greenhouse mode, it combines the selected head and backpack, which are static meshes, with the animated entity for the selected body.

For the head and backpack to animate correctly, BOT-anist needs to update their position and rotation every frame so they line up with the appropriate joint in the body's skeleton. BOT-anist does this using a system and custom component called `JointPin`

`System` and `JointPinComponent`, which together to keep the robot parts animating in sync. When the player taps or clicks the Start Planting button, the app adds the component to the parent entity that the three body parts share. It identifies that its entity has children that `JointPinSystem` needs to reposition. `JointPinSystem` then uses the data `JointPinComponent` provides to reposition the backpack and head to match the body's animated position on each frame.

When the player selects the Start Planting button, the app combines the three selected entities using the `RobotCharacter` class. That class's initializer retrieves the transforms for the head and backpack joints using the `pins` property on `Entity`. This property provides access to the entity's `GeometricPinsComponent`, which stores a collection of transforms, each of which identifies a different location, orientation, and scale relative to the entity, but without the overhead of a separate child entity for each one. People can create pins, but RealityKit also automatically creates a collection of pins to represent the joints in a rigged model.

After the player taps or clicks the button, the app retrieves the two geometric pins that represent the head and backpack joints in the body's skeleton. Because skeleton joints are arranged in a hierarchy, with each joint inheriting its parent's transform, the app retrieves the entire joint chain from the root joint to the backpack or head joint using a private function called `getJointHierarchy(_:for:)`.

```
let headJointIndices = getJointHierarchy(skeleton, for: "head")
let backpackJointIndices = getJointHierarchy(skeleton, for: "backpack")
```

Next, it calculates an offset for the two pins. The back and head model entities are places so they align with the correct spot on the body model. Because they're not at the origin, in order to rotate them on the origin, the `JointPinSystem` needs to move them before applying the transform, otherwise they have the wrong pivot point. To calculate the offset, it gets the position of each pin and inverts it by multiplying the position by −1.

```
guard var headOffset = headOffset ?? skeleton.pins["head"]?.position,
      var backpackOffset = backpackOffset ?? skeleton.pins["backpack"]?.position els
    fatalError("Didn't find expected joint for head or backpack.")
}
headOffset *= -1
backpackOffset *= -1
```

Finally, it creates the `JointPinComponent` with all the information the system needs to update the head and backpack entities.

```
skeletonClone.jointPinComponent = JointPinComponent(headEntity: self.head,
                                                    headJointIndices: headJointIndic
                                                    headOffset: Transform(translatic
                                                    backpackEntity: self.backpack,
                                                    backpackJointIndices: backpackJc
                                                    backpackOffset: Transform(transl
                                                    bodyEntity: self.body)
```

To move the head and body each frame, `JointPinSystem` uses an entity query to find the parent entity the head, body, and backpack share. It then retrieves the entity representing the body's skeleton and also retrieves all of the skeleton's joint transforms.

```
guard let skeleton = skeleton as? ModelEntity,
      let component = skeleton.jointPinComponent else { fatalError("Skeleton doesn't

let transforms = skeleton.jointTransforms
```

Because BOT-anist has to apply the same logic to two different meshes, only using a different joint and offset, it uses a private function called `pinEntity(indices:skeleton:transforms:offset:staticEntity:shouldRotate)` to apply that logic, which it then calls twice — once for the head and once for the backpack — after retrieving the data it needs from the component.

```
guard let skeleton = skeleton as? ModelEntity,
      let component = skeleton.jointPinComponent else {
        fatalError("Skeleton doesn't have required joint pin component.")
    }

let transforms = skeleton.jointTransforms

pinEntity(indices: component.headJointIndices,
          skeleton: skeleton,
          transforms: transforms,
          offset: component.headOffset,
          staticEntity: component.headEntity,
          shouldRotate: component.bodyEntity.name == "body1")

pinEntity(indices: component.backpackJointIndices,
          skeleton: skeleton,
```

```
            transforms: transforms,
            offset: component.backpackOffset,
            staticEntity: component.backpackEntity)
```

To calculate the correct transform for each joint, the system uses the joint chain it put in the component, and multiplies each joint's transform matrix together starting with the root joint. It uses `reduce(_:_:)` to iterate through the joint chain, multiplying each transform with the next one. It then takes that calculated transform and offsets it to move it back to its original location.

```
var headTransform = component.headJointIndices.reduce(matrix_identity_float4x4) { pa
    transforms[index].matrix * partialResult
}
let previousHeadScale = component.headEntity.scale(relativeTo: skeleton)
component.headEntity.setTransformMatrix(headTransform * component.headOffset, relati
component.headEntity.setScale(previousHeadScale, relativeTo: skeleton)
```

See Also

Related samples

{} Hello World

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

{} Swift Splash

Use RealityKit to create an interactive ride in visionOS.

{} Happy Beam

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

{} Destination Video

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

{} Diorama

Design scenes for your visionOS app using Reality Composer Pro.

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).
- Creating USD files for Apple devices
  Generate 3D assets that render as expected.

Related videos

Dive deep into volumes and immersive spaces

Compose interactive 3D content in Reality Composer Pro

Discover RealityKit APIs for iOS, macOS, and visionOS

What's new in USD and MaterialX

Create custom visual effects with SwiftUI

Optimize your 3D assets for spatial computing