Article

Using transforms to move, scale, and rotate entities

Learn how to use Transforms to move, scale, and rotate entities in RealityKit.

Overview

RealityKit `Entity` objects exist in a tree, and each entity can have any number of subentities. (The entities themselves can standalone, or can be in a single container.) Every entity in the tree stores its own transform component. The transform contains the `translation`, `scale`, and `orientation` relative to its container entity. The *root* of each tree is an entity without a container entity.

Each entity exists in its own coordinate system that defines the origin and orientation of the three ordinal directions (the x, y, and z axes). The coordinate system is relative to its container coordinate system and is defined by its transform.

Arrange entities with transforms

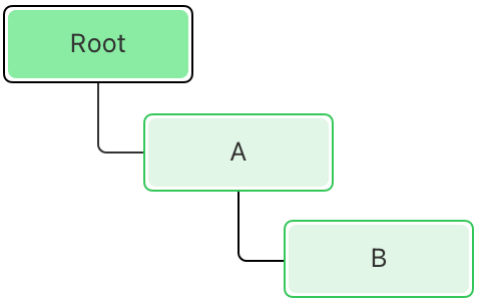
A root entity has no parent entity. Its location in the scene is either controlled by SwiftUI or placed via a `SpatialTrackingSession`. SwiftUI provides a root entity for the volume defined by the `RealityView`. The root entity defines the root coordinate system.

> **Note**
>
> In addition to a spatial tracking session, apps can use an `ARSession` with any number of data providers. The available list can be found in the type `DataProvider`, and a full list of anchor types is found in `AnchoringComponent.Target`.

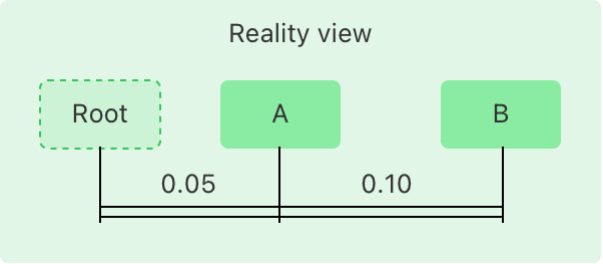
Each entity added to the tree adds a new coordinate system defined by its `transform` and is relative to its container entity. Each of the coordinate systems relate to each other by the hierarchy of entities and their transforms. For example a hierarchy of entities built with this code:

```
RealityView { content in
  let a = ModelEntity(mesh: .generateBox(size: 0.05), materials: [SimpleMaterial(co
  a.name = "A"
  a.transform = Transform(translation: SIMD3<Float>(0.05, 0.0, 0.0))
  content.add(a)

  let b = ModelEntity(mesh: .generateBox(size: 0.05), materials: [SimpleMaterial(co
  b.name = "B"
  b.transform = Transform(translation: SIMD3<Float>(0.1, 0.0, 0.0))

  a.addChild(b)
}
```

This reality view has three entities B, A, and the root entity. These three entities form a tree, with one root entity at the center of the reality view's volume. Entity A is a subentity of the root, and B is a subentity of A.



The reality view provides the root entity, which is located at the center of a volumetric window or near the floor in an immersive space. Use the `add(_:Entity)` method on the `content` supplied by the reality view to add entities as subentites of that root entity. The coordinate system defined by B is `0.1` units along the `x-axis` of the coordinate system defined by A. The coordinate system defined by A is `0.05` units along the `x-axis` defined by the root. With three entities there are three coordinate systems.



In this example there are two cubes. Each cube has eight corners, and each corner is `0.025` units away from the origin. The cubes appear in different locations in the scene because the system applies the `transform` to each corner of the cubes moving them from the local coordinate system (also called *model space*) to the world coordinate system. For example, the top, right, forward corner of the cube is at `{0.025, 0.025, 0.025}` in `model space`. The entity is translated by `{0.05, 0.0, 0.0}` The top-right-forward corner is then at `{0.075, 0.025, 0.025}`.

Build a simple entity to experiment with

To be visible, an entity must have a `MeshDescriptor` and a `Material`. A `MeshDescriptor` contains the description of a mesh. In this case, the mesh contains all of the vertices and how they connect into triangles. A Material specifies the color and appearance of the entity.

The previous example used `generateBox(size:)` to generate the mesh. This convenience obscures what the transform does. The remaining examples use a mesh built from scratch.

All entities have a coordinate space, often called *model space*. This coordinate system determines the location of the *vertices*.

The code below builds an entity with the following properties:

- The entity has one material.
- The mesh has 8 vertices and 12 triangles. Each vertex is one corner of the cube. Each triangle is one half of each side and involves three of the vertices. There are six sides, with two triangles each, for a total of 12 triangles. These vertices are in the model space coordinate system. This function builds the entity.

```
func createCube() -> ModelEntity? {
  let material = SimpleMaterial(color: .blue, isMetallic: false)
  var descriptor = MeshDescriptor(name: "Simple Cube")
  let allVerts: Array<SIMD3<Float>> = [[0.05,  0.05, -0.05], // Right-top-back verte
                                       [-0.05,  0.05, -0.05], // Left-top-back verte
                                       [-0.05,  0.05,  0.05], // Left-top-front vert
                                       [0.05,  0.05,  0.05], // Right-top-front vert
                                       [0.05, -0.05, -0.05], // Right-bottom-back ve
                                       [-0.05, -0.05, -0.05], // Left-bottom-back ve
                                       [-0.05, -0.05,  0.05], // Left-bottom-front v
                                       [0.05, -0.05,  0.05]] // Right-bottom-front v
  let allTris: Array<UInt32> = [
    // Index into the vertex array for the top triangles.
    0, 1, 2,
    0, 2, 3,
```

```
    // Index into the vertex array for the bottom triangles.
    4, 6, 5,
    4, 7, 6,
    // Index into the vertex array for the right triangles.
    0, 3, 7,
    7, 4, 0,
    // Index into the vertex array for the back triangles.
    0, 4, 1,
    4, 5, 1,
    // Index into the vertex array for the left triangles.
    2, 1, 5,
    5, 6, 2,
    // Index into the vertex array for the front triangles.
    3, 2, 6,
    6, 7, 3
  ]
  descriptor.positions = MeshBuffer(allVerts)
  descriptor.primitives = .triangles(allTris)

  if let resource = try? MeshResource.generate(from: [descriptor]) {
     return ModelEntity(mesh: resource, materials: [material])
  }
  return nil
}
```

Play ⊙

> **Note**
>
> For more information about constructing meshes, see `MeshDescriptor`. When you set the `transform` on an entity, the system transforms the mesh vertices to the new coordinate system.

Add the cube entity to a reality view

You add the `cube` entity to the volumetric window via SwiftUI like this:

```
RealityView { content in
  if let cube = createCube() {
    content.add(cube)
  }
}
```

The cube entity appears at the center of the volume, the origin of the volume's root entity.

Move the cube with a transform

To move the cube use the `Transform` component with the `translation` argument:

```
    cube.transform = Transform(translation: SIMD3<Float>(0.1, 0.0, 0.0))
```

Applying this `Transform` to the cube moves all eight vertices in the `x` direction by `0.1`.

The system moves the entity from its 'model space' origin to the location in world space. To achieve that effect, RealityKit performs some linear algebra behind the scenes to 'transform' the points into world space. The left matrix is the `Transform` converted to a matrix. The right vertex is the list of vertices. Here is the full multiplication for the transform and the first vertex.

$$\begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0.05 \\ 0.05 \\ -0.05 \\ 1.0 \end{bmatrix}$$

The left matrix is a direct representation of the `Transform` you made earlier and applied to the `cube`. The right matrix is the first vertex from the cube represented by a vector with `1.0` in the last position. Performing that multiplication (the `dot(_:_:)` product of each row of the matrix with the vertex) yields:

$$\begin{bmatrix} 0.15 \\ 0.05 \\ -0.05 \\ 1.0 \end{bmatrix}$$

The net effect is that the new vertices have `0.1` added to their `x` component. This approach generalizes to all other forms of transformation that you use to manipulate entities in RealityKit:

Play ⦿

Scale the cube with a transform

To scale an entity use the `transform` property. To apply a uniform scale of 2 to the entity change the code, like this:

```
cube?.transform = Transform(scale: SIMD3<Float>(2.0, 2.0, 2.0))
```

That yields a matrix multiplication that looks like this:

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -0.05 \\ 0.05 \\ 0.05 \\ 1.0 \end{bmatrix}$$

After the multiplication yields a transformed vertex like this:

$$\begin{bmatrix} -0.1 \\ 0.1 \\ 0.1 \\ 1.0 \end{bmatrix}$$

After multiplying all of the vertices by the scale matrix, the cube is twice as large in each direction (`0.2` versus `0.1`):

Play ⦿

Combine transforms

These two operations combine into a single operation with the `Transform` type like this:

```
cube?.transform = Transform(scale: SIMD3<Float>(2.0, 2.0, 2.0), translation: SIMD3
```

That transform yields a matrix multiplication for all the vertices, laid out as column vectors. The multiplication looks like this:

The order is important: scale first then translate.

> **Important**
>
> Multiplying matrixes isn't commutative, which means that A * B is not equal to B * A.

Multiplying these two transformation matrices in the order shown above yields this result:

The result scales the model by 2 uniformly and translates the model by `0.1` in the x direction:

Play ⊙

Switching that order yields a different matrix:

This resulting matrix yields a similar uniform scale of 2, but the translation is scaled by 2 as well. The net result of this matrix is to scale the model uniformly by 2 and move it in the positive x direction by `0.2`:

Play ⦿

> **Important**
>
> Matrix multiplication is associative, which means that you can move the parenthesis around. So A * B * C can be done as A * (B * C) or (A * B) * C. This allows the `transform` to be one matrix application instead of two.

Multiply the `scale` matrix by the `translation` matrix to get the combined `transform` matrix. RealityKit then applies the combined matrix to the vertices:

Which yields a matrix like this:

The scaled and translated vertices yield a cube that is twice as large in each direction and moved `0.1` units to the right.

Rotate entities

To rotate the cube 45° (π/4 radians), use the `Transform` type with the `rotation` argument like this:

```
RealityView { content in
  if let cube {
    content.add(cube)
  }
  cube?.transform = Transform(rotation: simd_quatf(angle: .pi/4, axis: SIMD3<Float>(
 }
```

This causes the cube to rotate 45° around its origin along the `x-axis`. The matrix for this rotation looks like this:

Play ⊙

Rotation around any other axis is achieved in the same way. For example, to rotate 45° around the axis through the top-right corner of the cube you could use:

```
cube?.transform = Transform(rotation: simd_quatf(angle: .pi/4, axis: normalize(SIM
```

Important

The `normalize(_:)` function returns a vector pointing in the same direction with a length of 1.0. Make sure to normalize the `axis` argument when creating quaternions.

That code performs a rotation that looks like this:

Play ⊙

Applying this transformation matrix to the full set of vertices yields this new set of transformed vertices:

Notice that the fourth and sixth vertex didn't change. The axis of rotation goes through those two vertices so nothing changes on that axis.

Combine rotation, translation, and scale in one transform

Rotation combined with other transforms might yield unexpected results depending on the order of the application. You can combine all three transformations in the `Transform` initializer like this:

```
cube?.transform = Transform(scale: SIMD3<Float>(2.0, 2.0, 2.0),
                            rotation: simd_quatf(angle: .pi/4, axis: normalize(SIM
                            translation: SIMD3<Float>(0.1, 0.0, 0.0))
```

The order of these transforms is `translation` followed by `rotation` then `scale`.

See Also

RealityKit and Reality Composer Pro

- Reality Composer Pro

  Build, create, and design 3D content for your RealityKit apps.

- {} Petite Asteroids: Building a volumetric visionOS game

  Use the latest RealityKit APIs to create a beautiful video game for visionOS.

- {} BOT-anist

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

- {} Swift Splash

  Use RealityKit to create an interactive ride in visionOS.

- {} Diorama

  Design scenes for your visionOS app using Reality Composer Pro.

- {} Building an immersive media viewing experience

  Add a deeper level of immersion to media playback in your app with RealityKit and Reality Composer Pro.

- {} Enabling video reflections in an immersive environment

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

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

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

- Understanding the modular architecture of RealityKit

  Learn how everything fits together in RealityKit.

- Capturing screenshots and video from Apple Vision Pro for 2D viewing

  Create screenshots and record high-quality video of your visionOS app and its surroundings for app previews.

- Implementing object tracking in your visionOS app

  Create engaging interactions by training models to recognize and track real-world objects in your app.

- Placing entities using head and device transform

  Query and react to changes in the position and rotation of Apple Vision Pro.