Article

Understanding the modular architecture of RealityKit

Learn how everything fits together in RealityKit.

Overview

RealityKit is a 3D framework designed for building apps, games, and other immersive experiences. Although it's built in an object-oriented language and uses object-oriented design principles, the architecture of RealityKit avoids heavy use of composition — where objects are built by adding instance variables that hold references to other objects — in favor of a modular design based on a paradigm called Entity Component System (ECS) that divides application objects into one of three types.

Following the ECS paradigm allows you to re-use the functionality contained in a component in many different entities, even if they have very different inheritance chains. Even if two objects have no common ancestors other than `Entity`, you can add the same components to both of them and give them the same behavior or functionality.

Start with entities

Entities are the core actors of RealityKit. Any object that you can put into a scene, whether visible or not, is an entity and must be a descendent of `Entity`. Entities can be 3D models, shape primitives, lights, or even invisible items like sound emitters or trigger volumes. Add components to entities to let them store additional state relevant to a specific type of functionality. Entities themselves contain relatively few properties: Nearly all entity state is stored on an entity's components.

RealityKit provides a number of entity types you use to represent different kinds of objects. For example, a `ModelEntity` represents a 3D model, such as one imported from a `.usdz` or `.reality` file. These provided entities are essentially just an `Entity` with certain components already added to them. Adding a `ModelComponent` to an instance of `Entity`, for example, results in an entity with identical functionality to a `ModelEntity`.

Add components to entities

Components are modular building blocks that you add to an entity; they identify which entities a system will act on, and maintain the per-entity state that systems rely on. Components can contain logic, but limit component logic to code that validates its property values or sets its initial state. Use systems for any logic that affects the behavior of entities or that potentially changes their state on every frame. To add accessibility information to an entity, for example, add a `AccessibilityComponent` to it and populate its fields with the information the accessibility system needs, such as putting the description that VoiceOver reads into its `label` property.

Keep in mind that an entity can only hold one copy of any particular type of component at a time. So, for example, you can’t add two accessibility components to one entity. If you add an accessibility component to an entity that already has one, the new component replaces the previous one.

Create systems to implement entity behavior

A `System` contains code that RealityKit calls on every frame to implement a specific type of entity behavior or to update a particular type of entity state. Systems use components to store their entity-specific state and query for entities to act on by looking for ones with a specific component or combination of components.

For example, a game might have a damage system that monitors and updates the health of every entity that can be damaged or destroyed. Systems typically work together with one or more components, so that damage system might use a health component to keep track of how much damage each entity has taken and how much each one is able to take before it’s destroyed. It might also interact with other components. For example, an entity might have an armor component that provides protection to the entity, and the damage system would also need to use the state stored in that component.

Every frame, the damage system queries for entities that have the health component and updates values on those entities’ components based on the current state of the app. If an entity has taken too much damage, the system might trigger a specific animation or remove the entity from the scene.

Writing entity logic in a system avoids duplication of work. Using traditional OOP design patterns, where this type of logic would reside on the entity class, can often result in the same calculations being performed multiple times, once for every entity potentially affected. No matter how many entities the calculation potentially impacts the system only has to do the calculation once.

For more information on creating systems, see Implementing systems for entities in a scene

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.

Using transforms to move, scale, and rotate entities

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