

[RealityKit](#) / [Simulations and motion](#) / Simulating particles in your visionOS app

Sample Code

Simulating particles in your visionOS app

Add a range of visual effects to a RealityKit view by attaching a particle emitter component to an entity.

Download

visionOS 1.0+ | Xcode 15.3+

Overview

Define and tailor a particle system by creating and configuring a particle emitter component and then adding it to an entity. A particle emitter component can simulate various effects including fire, smoke, confetti, and so on.

The sample app presents sliders, toggles, buttons, and color pickers that adjust the particle emitter's settings. You change the appearance and behavior of the particles it emits with these controls, including their size, shape, and speed.



Play ▶

The app's controls let you quickly try a variety of particle effects that illustrate the range of possibilities for your app.

You can add a particle emitter to your RealityKit scene in four steps.

1. Create an entity.
2. Create a particle emitter component.
3. Add that component to the entity.
4. Add the entity to a scene.

Create an entity

You can turn any entity into a particle emitter, including one that's already in your app's scene. The app creates a new entity with the default initializer `init()`.

```
let emitterEntity = Entity()
```

Create a particle emitter component

Create a `ParticleEmitterComponent` instance by calling one of its initializers.

```
var emitterComponent = ParticleEmitterComponent()
```

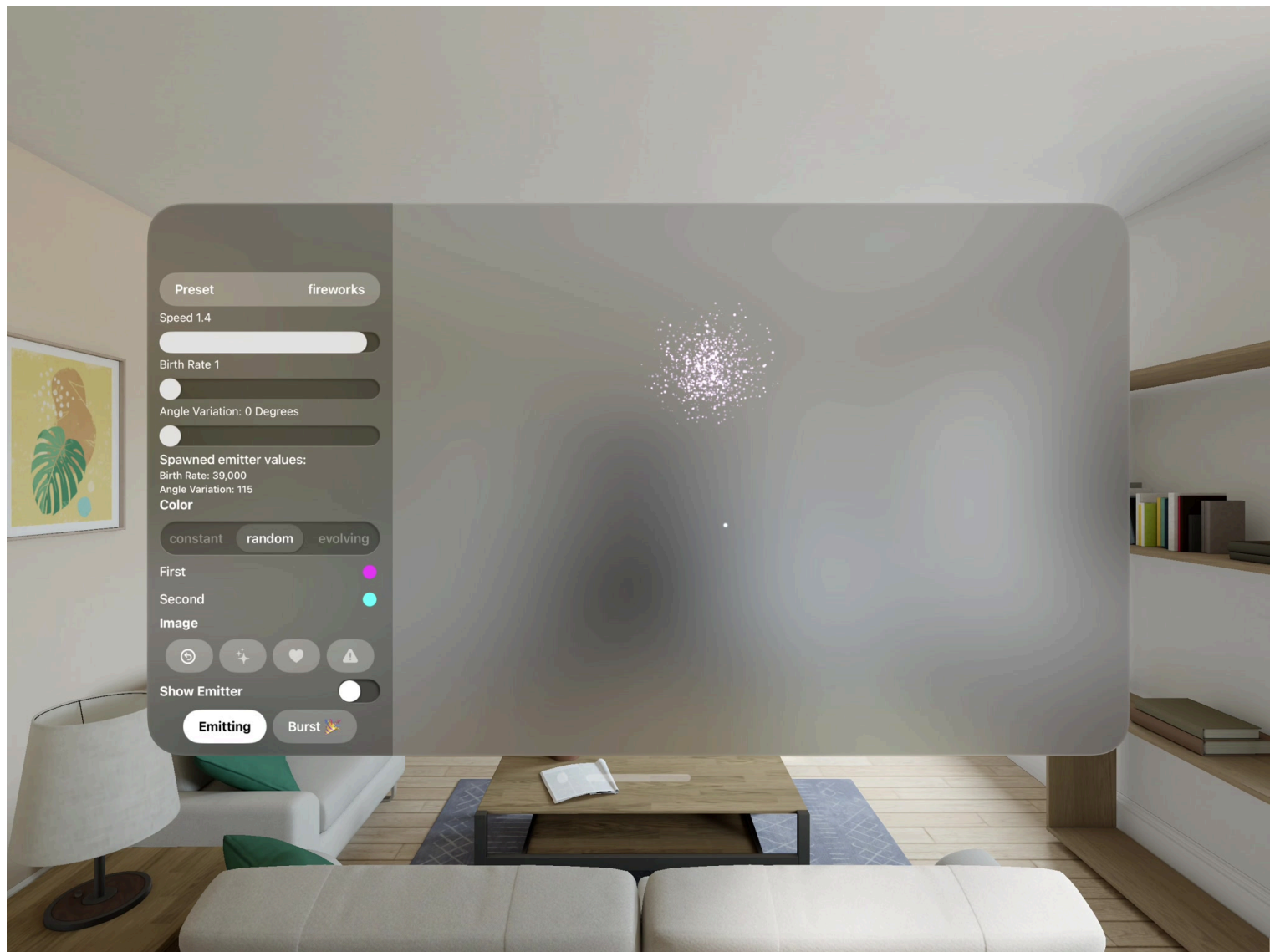
You can also create a particle emitter from a preset by accessing one of the properties of the `ParticleEmitterComponent.Presets` type.

```
typealias Presets = ParticleEmitterComponent.Presets

switch presetSelection {
    case .default:
        emitterComponent = ParticleEmitterComponent()
    case .fireworks:
        emitterComponent = Presets.fireworks
    case .impact:
        emitterComponent = Presets.impact
    case .magic:
        emitterComponent = Presets.magic
    case .rain:
        emitterComponent = Presets.rain
    case .snow:
        emitterComponent = Presets.snow
    case .sparks:
        emitterComponent = Presets.sparks
}
```

Each preset is an initial configuration that sets up a new particle emitter system.

Fireworks Impact Magic Rain Snow Sparks



Play ▶

You can use a preset as a starting point and then alter the behavior and appearance of the emitter and its particles by modifying the component's properties. This approach may be more convenient than configuring every property of a particle emitter component.

Add a particle emitter component to an entity

Add the particle emitter component to an entity by calling the `set(_:)` method of its `components` property.

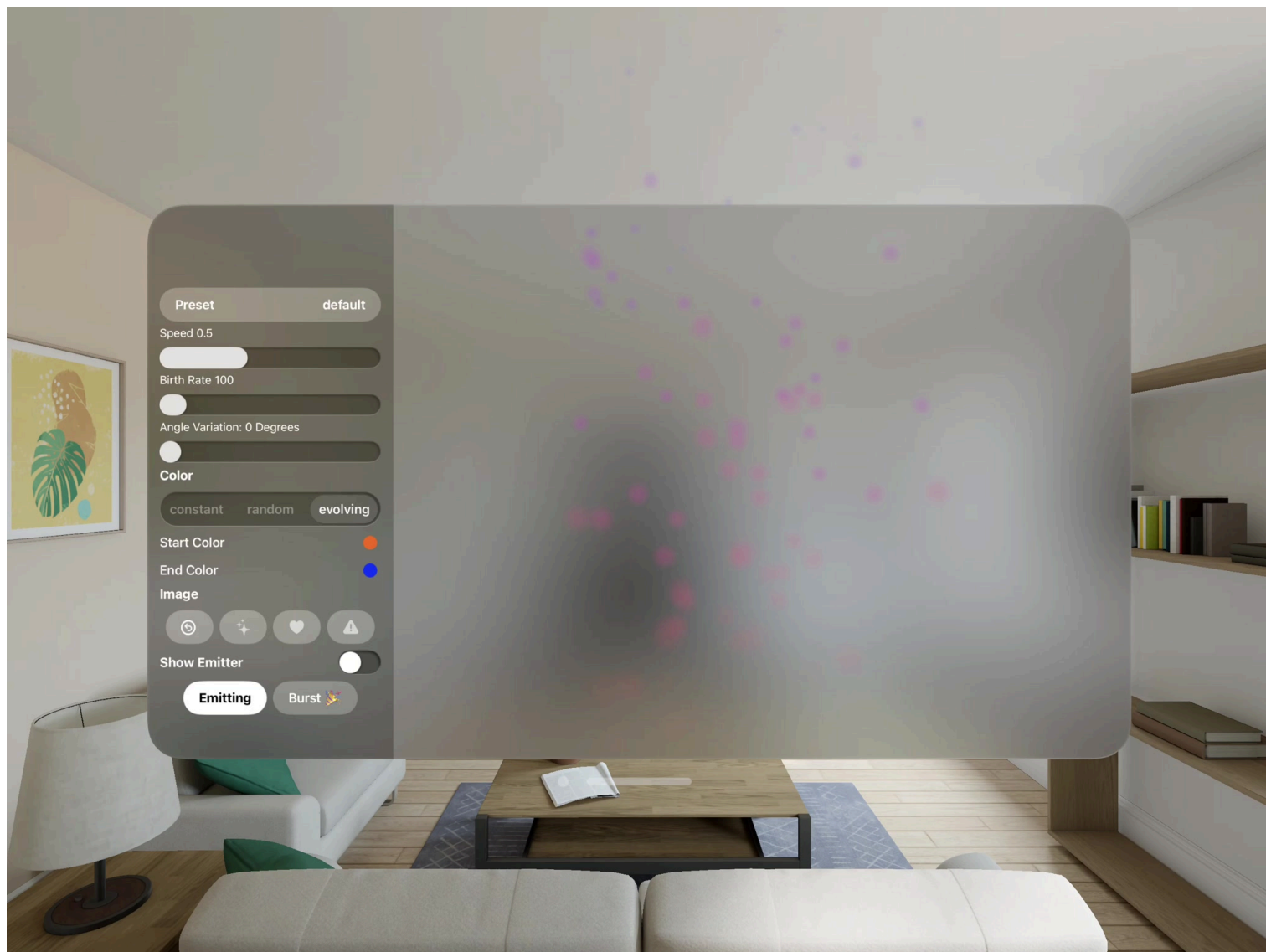
```
// Add the particle component to the entity.  
emitterEntity.components.set(emitterComponent)
```

Add the entity to a scene

Add the entity with the particle emitter component to the scene by calling the `add(_:)` method of the `RealityViewContent` instance from a `RealityView`.

```
// Add the emitter entity to the scene.  
content.add(emitterEntity)
```

The scene can now present the entity as a particle emitter because it has the component that defines the look and behavior of the emitter and the particles it creates.



Play ▶

Configure the behavior of the emitter's particles

You can alter the behavior of a particle system by changing the values of a [ParticleEmitter Component](#) instance's properties. For example, the [speed](#) property adjusts the starting speed of the particles as the emitter creates them, and the [birthRate](#) property modifies how many times per second the emitter creates a new particle.

```
emitterComponent.speed = 0.2  
emitterComponent.mainEmitter.birthRate = 150
```

```
emitterEntity.components.set(emitterComponent)
```

This example modifies the component's speed and birthrate properties to 0.2 and 150, respectively, from their default values 0.5 and 100, which creates more particles that move slower.



Play ▶

Important

If you adjust one or more particle component properties, you need to replace the entity's current particle component instance by setting it again because component instances are from value types.

Configure the color for the emitter's particles

You can customize the color of the particles an emitter component creates by setting the color property to one of the enumeration cases, including:

- ParticleEmitterComponent.ParticleEmitter.ParticleColor.constant(:)
- ParticleEmitterComponent.ParticleEmitter.ParticleColor
.evolving(start:end:)

A particle with a constant color maintains the same color throughout its lifetime, but a particle with an evolving color gradually changes from one color to another over the duration of its lifetime.

The app has two methods that configure particles with a constant color. The `setConstantColor(_ :)` method creates a ParticleEmitterComponent.ParticleEmitter.ParticleColor.ColorValue instance by passing the color a person selects in the app's UI to the ParticleEmitterComponent.ParticleEmitter.ParticleColor.ColorValue
.single(_ :) enumeration case, and then creates a constant color from that value.

```
typealias ParticleEmitter = ParticleEmitterComponent.ParticleEmitter
typealias ParticleColor = ParticleEmitter.ParticleColor
typealias ParticleColorValue = ParticleColor.ColorValue

// ...

func setConstantColor(_ swiftUIColor: SwiftUI.Color) {
    // Create a single color value instance.
    let color1 = ParticleEmitter.Color(swiftUIColor)
    let singleColorValue = ParticleColorValue.single(color1)

    // Create a constant color from the single color value.
    let constantColor = ParticleColor.constant(singleColorValue)

    // Change the particle color for the emitter.
    emitterComponent.mainEmitter.color = constantColor

    // Replace the entity's emitter component with the current configuration.
    emitterEntity.components.set(emitterComponent)
}
```

The app's `setRandomColor(_:_ :)` method also creates a constant color that's random by passing two colors from the app's UI to the ParticleEmitterComponent.ParticleEmitter
.ParticleColor.ColorValue.random(a:b:) enumeration case.

```
func setRandomColor(_ swiftUIColor1: SwiftUI.Color,
                   _ swiftUIColor2: SwiftUI.Color) {
    // Create a random color value instance between two colors.
    let color1 = ParticleEmitter.Color(swiftUIColor1)
```

```

let color2 = ParticleEmitter.Color(swiftUIColor2)
let randomColor = ParticleColorValue.random(a: color1, b: color2)

// Create a constant color from the random color value.
let constantColor = ParticleColor.constant(randomColor)

// Change the particle color for the emitter.
emitterComponent.mainEmitter.color = constantColor

// Replace the entity's emitter component with the current configuration.
emitterEntity.components.set(emitterComponent)
}

```

The enumeration case creates a color value from the two input colors by selecting a random interpolation value between them.

The app's `setEvolvingColor(_:_:)` method configures an emitter's particles so that they shift from one color to another over time by creating an evolving color.

```

func setEvolvingColor(_ swiftUIColor1: SwiftUI.Color,
                     _ swiftUIColor2: SwiftUI.Color) {

    // Create two single color value instances.
    let color1 = ParticleEmitter.Color(swiftUIColor1)
    let color2 = ParticleEmitter.Color(swiftUIColor2)
    let singleColorValue1 = ParticleColorValue.single(color1)
    let singleColorValue2 = ParticleColorValue.single(color2)

    // Create an evolving color that shifts from one color value to another.
    let evolvingColor = ParticleColor.evolving(start: singleColorValue1,
                                              end: singleColorValue2)

    // Change the particle color for the emitter.
    emitterComponent.mainEmitter.color = evolvingColor

    // Replace the entity's emitter component with the current configuration.
    emitterEntity.components.set(emitterComponent)
}

```

The method creates an evolving color by passing two single color value instances to the `ParticleEmitterComponent.ParticleEmitter.ParticleColor.evolving(start: end:)` enumeration case.

Constant single color

Constant random color

Evolving color



Play ▶

Customize the appearance of the emitter's particles with images

The app lets a person change the appearance of the emitter's particles by applying an image to each. It does this by creating a [TextureResource](#) instance from an image and assigning it to the emitter's [image](#) property.

```
emitterComponent.mainEmitter.image = textureResource
```

You can create a texture resource by loading an image from the app's bundle with the [load\(named:in:\)](#) method, or by passing a [CGImage](#) instance to the [generate\(from:with Name:options:\)](#) factory method.

The app generates a texture resource in its `generateTextureFromSystemName(_:)` method by taking these steps:

1. Retrieve a symbol from SF Symbols using the symbol's name.
2. Draw the image as a template onto a white background in a graphics context.
3. Retrieve the image from the context.
4. Pass the final image to the texture resource type's factory method.

```
func generateTextureFromSystemName( _ name: String) -> TextureResource? {
    let imageSize = CGSize(width: 128, height: 128)

    // Create a UIImage from a symbol name.
    guard var symbolImage = UIImage(systemName: name) else {
        return nil
    }

    // Create a new version that always uses the template rendering mode.
    symbolImage = symbolImage.withRenderingMode(.alwaysTemplate)

    // Start the graphics context.
    UIGraphicsBeginImageContextWithOptions(imageSize, false, 0)

    // Set the color's texture to white so that the app can apply a color
    // on top of the image.
    UIColor.white.set()

    // Draw the image with the context.
    let rectangle = CGRect(origin: CGPoint.zero, size: imageSize)
    symbolImage.draw(in: rectangle, blendMode: .normal, alpha: 1.0)

    // Retrieve the image from the context.
    let contextImage = UIGraphicsGetImageFromCurrentImageContext()

    // End the graphics context.
    UIGraphicsEndImageContext()

    // Retrieve the Core Graphics version of the image.
    guard let coreGraphicsImage = contextImage?.cgImage else {
        return nil
    }
}
```

```
// Generate the texture resource from the Core Graphics image.
let creationOptions = TextureResource.CreateOptions(semantic: .raw)
return try? TextureResource.generate(from: coreGraphicsImage,
                                     options: creationOptions)
```

```
}
```

The method makes a texture resource that works with the emitter's current color by starting with all white before drawing the image onto it.

The app presents a slider that changes the angleVariation property. In the example below, the slider's value is $\pi / 6$ radians, which is equivalent to 30° .



Play ▶

See Also

Particle simulation

```
struct ParticleEmitterComponent
```

A component that emits particles.

```
struct ParticleEmitter
```

```
struct Presets
```

Initial configurations that can be set when starting a new simulation.