Sample Code

Altering RealityKit Rendering with Shader Functions

Create rendering effects by writing surface shaders and geometry modifiers.

Download

iOS 15.0+ | iPadOS 15.0+ | Xcode 14.0+

Overview

In iOS 15 and later, and macOS 12 and later, you can write Metal functions to alter the way RealityKit renders entities that uses a `CustomMaterial`. Custom materials require a Metal function called a *surface shader*, which sets the final rendering attributes for each of the entity's fragments. A *fragment* is a pixel that is potentially affected by rendering that entity. Custom materials also support an optional Metal function called a *geometry modifier* that can move vertices or alter other vertex data, such as vertex colors or UV coordinates.

This sample demonstrates using custom materials with Metal functions to create special rendering effects. In this project, several toy robot entities are placed on a detected horizontal plane. Tapping a robot causes it to rapidly expand and become transparent so it appears to pop like a bubble. This project demonstrates creating a custom material from the `PhysicallyBasedMaterial` on a loaded model, using the material's custom texture and vector properties to pass data to shader functions, and writing a surface shader and a geometry modifier to actually implement the rendering effects.

For more information on using custom materials, see Modifying RealityKit rendering using custom materials

Configure the Sample Code Project

Because this sample app uses RealityKit, you can't run it in Simulator — you'll need to run it on a device. To run this sample, you'll need the following:

- A Mac running macOS 12 or later
- Xcode 13 or later
- An iOS device running iOS or iPadOS 15 or later

Create a Surface Shader to Animate Transparency

The surface shader in this project overrides RealityKit's default transparency behavior to implement the dissolve effect that makes the robots disappear. The shader is contained in the `DissolveSurfaceShader.metal` file. The dissolve process is controlled by a value between `0.0` and `1.0` that you pass to the shader function using the first component of the `custom` property's `value` vector. The first thing the surface shader does is retrieve that value.

```
// Get the first float in the custom vector, which contains the animation
// progress for this entity as a value between 0.0 and 1.0.
float animationProgress = params.uniforms().custom_parameter()[0];
```

If the progress value is `1.0` or greater, the dissolve is complete and there's nothing to render, so the shader returns.

```
// If the value is greater than 1.0, the dissolve has completed, so there's
// no reason to draw anything. Discard the fragment to ensure that
// RealityKit draws nothing for this fragment.
if (animationProgress >= 1.0) {
    discard_fragment();
    return;
}
```

A custom material's surface shader is responsible for setting all of the rendering attributes of the current fragment. This project uses a robot model from RealityComposer that uses physically based rendering (PBR). In order to replicate that behavior, the shader has to set all of the properties used by `PhysicallyBasedMaterial` to ensure that RealityKit renders the model correctly. The project calls several Metal functions contained in the `CustomMaterialHelpers.metal` file that emulate the logic used by the RealityKit's `PhysicallyBasedMaterial` shaders.

```
// Replicate PhysicallyBasedMaterial's behavior for each of the physically
// based rendering (PBR) attributes supported by CustomMaterial.
```

```
baseColorPassThrough(params);
normalPassThrough(params);
roughnessPassThrough(params);
metallicPassThrough(params);
specularPassThrough(params);
ambientOcclusionPassThrough(params);
clearcoatPassThrough(params);
emissiveColorPassThrough(params);
```

Next, the surface shader implements a texture-driven dissolve by checking if the retrieved progress value is greater than `0.0`. If it is, the shader samples a UV-mapped value from the `custom` property's `texture`. If the sampled value is less than a threshold calculated from the dissolve progress, the fragment is rendered opaque; otherwise, the fragment is discarded, which allows anything behind the fragment to show through.

```
// Because the project loaded this entity from a USDZ file, get and
// flip the UV coordinates. This is equivalent to:
//
//     float2 uv = params.geometry().uv0();
//     uv.y = 1.0 - uv.y;
auto uv = getFlippedUVs(params);

// Sample the opacity texture value. The sampled value controls how
// different parts of the entity dissolve. The lighter the color of the
// texture the later in the dissolve it becomes invisible. Changing the
// material's custom texture will yield different dissolve effects.
float textureColor = params.textures().custom().sample(textureSampler, uv).r;

// Implement the dissolve so that all pixels are either opaque or
// dissolved (fully transparent). Render any value above the threshold
// as transparent, and any value below the threshold as opaque.
float threshold = 1.0 - animationProgress;
if (textureColor < threshold) {
    params.surface().set_opacity(1.0);
} else {
    // Setting the opacity to 0.0 using PBR (.lit or clearcoat) results
    // in a transparent glass-like object. This means that RealityKit
    // might render some value for this fragment due to specular
    // highlights or clearcoat. To render nothing for this fragment,
    // completely discard transparent fragments to avoid the possibility
    // of RealityKit rendering a value for this fragment.
    discard_fragment();
    // Once the fragment is discarded, there's no reason to continue.
    return;
}
```

Write a Geometry Modifier that Expands an Entity

This project uses a geometry modifier to scale the robot entity along its vertex normals. By scaling along normals instead of scaling uniformly, the entity expands in a way that resembles an expanding bubble. The geometry modifier determines how much to scale the entity based on the same progress value retrieved in the surface shader.

```
[[visible]]
void ExpandGeometryModifier(realitykit::geometry_parameters params)
{
    // Retrieve the progress value from the material.
    auto uniforms = params.uniforms();
    float progress = uniforms.custom_parameter()[0];

    // If the progress value is 0.0 or less, the entity isn't animating, so
    // there's no work to do.
    if (progress <= 0.0) {
        return;
    }

    // Get the current vertex's normal vector.
    auto vertexNormal = params.geometry().normal();

    // Offset the vertex along the normal. The distance is based on the progress
    // value.
    params.geometry().set_model_position_offset(vertexNormal * progress * 3.0);

}
```

Create a Custom Material

With the shader functions written, the next step is to create a custom material in Swift. To load the surface shader and geometry modifier, get a reference to the Metal library that contains the project’s shader functions.

```
/// Creates references to the Metal device and Metal library, which are needed to lo
private func initializeMetal() {
    guard let maybeDevice = MTLCreateSystemDefaultDevice() else {
        fatalError("Error creating default metal device.")
    }
    device = maybeDevice
    guard let maybeLibrary = maybeDevice.makeDefaultLibrary() else {
        fatalError("Error creating default metal library")
    }
    library = maybeLibrary
}
```

Next, use that Metal library to get references to both of the shader functions.

```
let surfaceShader = CustomMaterial.SurfaceShader(
    named: "DissolveSurfaceShader",
    in: library
)

let geometryModifier = CustomMaterial.GeometryModifier(
    named: "ExpandGeometryModifier",
    in: library
)
```

Because there are multiple entities using the same effect, the next step is to retrieve all of the robot entities from the scene. Reality Composer and USDZ both support multiple *levels of detail* (LOD) for models, which means that the robot entity potentially has multiple child entities that contain different versions of the model with different amounts of detail. RealityKit automatically swaps to lower-detailed versions of the entity when the entity is further away from the camera.

To make sure RealityKit uses this custom material regardless of which LOD model is being displayed, iterate over the entity's children looking for a `ModelComponent`, which indicates a renderable non-primitive entity. When you find a model component, create a new custom material based on its existing `PhysicallyBasedMaterial`. RealityKit automatically creates a `PhysicallyBasedMaterial` instance for each material in the USDZ or .reality file. After creating the material, load the custom image texture and add it to the material, then set the first value of the custom vector to `0.0` to indicate that the entity isn't yet animating.

```
do {
    try robotTemplate.modifyMaterials {
        // Create a custom material based on the material ($0) that
        // RealityKit created automatically when loading the Reality
        // Composer file, and assign it.
        var customMaterial = try CustomMaterial(from: $0,
                                                surfaceShader: surfaceShader,
                                                geometryModifier: geometryModifier)

        // Use the first value of the custom vector to pass the
        // progress value to the shader functions.
        customMaterial.custom.value[0] = 0.0

        // Load the texture to pass to the shader functions, using
        // the custom texture slot.
        if let textureResource = try? TextureResource.load(named: "texture.jpg") {
            let texture = CustomMaterial.Texture(textureResource)
            customMaterial.custom.texture = .init(texture)
        }

        return customMaterial
    }
} catch {
    fatalError("Error creating custom material.")
}
```

Animate the Dissolve

When the user taps on one of the robots in the scene, the project animates the custom vector value used by the two shader functions over a short period of time. RealityKit automatically sends the updated values to the shader functions every frame, which causes the robots scale and dissolve. Once the animation is finished, it plays a “pop” sound.

```
private func incrementPopProgress(entity: Entity) async {
    let popDuration = 0.18
    let start = Date.now.timeIntervalSince1970
    var done = false

    while !done {
        let progress = (Date.now.timeIntervalSince1970 - start) / popDuration
        if progress > 1.0 {
            done = true
        }
        await Task { @MainActor in
            entity.setCustomVector(vector: SIMD4<Float>(x: Float(progress), y: 0.0,
        }.value
    }

    await Task { @MainActor in
        // The entity is invisible at this point, but it still responds to
        // taps unless it's disabled.
        entity.isEnabled = false

        // Play a fun sound as the robot pops.
        ApplicationActions.shared.playPop()
    }.value
}
```