Sample Code

Postprocessing a Scene With Custom Symbols

Create visual effects in a scene by defining a rendering technique with custom symbols.

Download

iOS 12.0+ | iPadOS 12.0+ | Xcode 16.0+

Overview

Games and 3D apps are often distinguishable because of unique visual effects. Many games offer settings where users can select filters which change the visual aesthetic of a game. For example, a user might select a grayscale filter to achieve a grittier feeling, or a pixelation filter make the game retro.

This sample app uses an `SCNTechnique` with custom symbols to create a user-adjustable color filter for the scene. The sample app binds values for each custom symbol to the Metal shader used by the `SCNTechnique`. The Metal shader accesses the bound values to produce the color filter effect in the scene. See the “Defining a Technique” section of the `SCNTechnique` documentation for detailed information about the steps required to define a technique.

Configure the Sample Code Project

The scene used in this project is configured entirely within the `max.scn` file.

Define the Postprocess Rendering Technique

The sample uses custom symbols as input to the fragment function in `MyShaders.metal`. The technique defines the custom symbols used by the fragment function:

1. The `symbols` dictionary defines the custom symbols used by the `inputs` dictionary of each pass. Each custom symbol specifies a `type` using the GLSL type that logically maps to the corresponding Metal type.

```
    "symbols": [
        "color_weights_symbol": [
            "type": "vec3"
        ],
        "time_symbol": [
            "type": "float"
        ]
    ],
```

In this sample, the `color_weights_symbol` has a type of `vec3`, even though it's a `float3` in the Metal shaders.

2. The `inputs` dictionary of each pass associates each custom symbol with the corresponding field in Metal.

```
    "inputs": [
        "color": "COLOR",
        "color_weights": "color_weights_symbol",
        "time": "time_symbol"
    ],
```

`"color_weights": "color_weights_symbol"`

In the key-value pair above, the key, on the left, is the name of the symbol as defined in the shader, and the value, on the right, is the name of the symbol as defined in the `symbols` dictionary of the technique.

Bind a Value to Each Custom Symbol

To bind values to the custom symbols, the sample project uses the `setValue:forKey:` method of `SCNTechnique`. Once bound, the values are available in the Metal shading program.

Value bindings use the logical type that corresponds with the Metal type. For example, custom symbols defined in the technique with a type of `vec3`, with a type of `float3` in the shader, use a

SCNVector3 value when bound. Reference the table on the SCNShadable page of the documentation to identify corresponding types between GLSL, Metal, and Swift.

Access Your Custom Symbols in Metal

This sample declares the Symbols struct as a constant input and assigns it to buffer index 0 of the fragment shader in MyShaders.metal. The name of the struct, and the buffer index it is assigned to is flexible. SceneKit binds the values solely based on the names of the fields in the struct. The Symbols struct defined in MyShaders.metal contains a field for each custom symbol.

```
struct Symbols {
    float3 color_weights;
    float time;
};
```

```
fragment half4 myFragmentShader(VertexOut in [[stage_in]],
                                constant Symbols &symbols [[buffer(0)]],
                                texture2d<half, access::sample> color [[texture(0)]]
```

See Also

Renderer Customization

`protocol` SCNShadable

Methods for customizing SceneKit's rendering of geometry and materials using Metal or OpenGL shader programs.

`class` SCNProgram

A complete Metal or OpenGL shader program that replaces SceneKit's rendering of a geometry or material.

`protocol` SCNBufferStream

An object that manages a Metal buffer used by a custom shader program.

`class` SCNTechnique

A specification for augmenting or postprocessing SceneKit's rendering of a scene using additional drawing passes with custom Metal or OpenGL shaders.

`protocol` SCNTechniqueSupport

The common interface for SceneKit objects that support multipass rendering using SCNTechnique objects.

`protocol` SCNNodeRendererDelegate

Methods you can implement to use your own custom Metal or OpenGL drawing code to render content for a node.