
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

16: Integrating Metal Shaders with SceneKit

Metal does not work on the iOS simulator yet, so you must have a Metal-enabled device to test on if you'd like to follow along.

16: Integrating Metal Shaders with SceneKit

Metal is a low level framework that allows you to control your code down to the bit level. However, many common operations don't require you to get down to that level because they are handled by Core Image and SceneKit. This session will show you what operations you get with SceneKit and how you can go deeper with Metal when you need to without losing the convenience of SceneKit.

46 Integrating Metal Shaders with SceneKit: Demo 1

By Janie Clayton

In this demo, you will learn how to set up a SceneKit project and use code snippets to inject shader code into the rendering pipeline.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo1\starter**.

1) Create SceneKit Properties

In `GameViewController.swift`, add the following to the top of the class:

```
var scnView: SCNView!
var scnScene: SCNScene!
var cameraNode: SCNNode!
```

2) Setup View

Next, fill in the `setupView()` function:

```
func setupView() {
  scnView = self.view as! SCNView
  scnView.showsStatistics = true
  scnView.allowsCameraControl = true
  scnView.autoenablesDefaultLighting = true
}
```

3) Setup Scene

Next, fill in the `setupScene()` function:

```
func setupScene() {
  scnScene = SCNScene()
  scnView.scene = scnScene
  scnScene.background.contents = UIColor.purple
}
```

4) Setup Camera

Next, fill in the `setupCamera()` function:

```
func setupCamera() {
  cameraNode = SCNNode()
  cameraNode.camera = SCNCamera()
  cameraNode.position = SCNVector3(x: 0, y:0, z: 10)
  scnScene.rootNode.addChildNode(cameraNode)
}
```

5) Setup ViewDidLoad

Next, add the following method calls to `viewDidLoad()`:

```
override func viewDidLoad() {
  super.viewDidLoad()
  setupView()
  setupScene()
  setupCamera()
}
```

6) Setup Box

Next, create the following method `addBox()`:

```
func addBox() {
  let box = SCNBox(width: 2.0, height: 2.0, length: 2.0, chamferRadius:
0.0)
  let geometryNode = SCNNode(geometry: box)
  scnScene.rootNode.addChildNode(geometryNode)
}
```

7) Create and Add Box

Next, add the method call to `viewDidLoad()`:

```
addBox()
```

8) Insert Solid Color Snippet

After the `box` object is created in `setupBox()` add the following snippet:

```
box.shaderModifiers = [.fragment:
  """
  _output.color.rgb = float3(1.0, 0.0, 0.0);
  """
]
```

9) That's it!

Congrats, at this time you should have a good understanding of inserting Metal snippets! It's time to move onto the Metal Shading Language.

Integrating Metal Shaders with SceneKit: Demo 2

By Janie Clayton

In this demo, you will explore a more complex example of how to inject Metal shaders into SceneKit projects.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo2\starter**.

1) Texel Coordiantes

Open **GameViewController.swift**. In `addBox()`, after you create the `SCNBox`, add:

```
box.shaderModifiers = [.surface:
  """
  float2 st = _surface.diffuseTexcoord.xy;
  """
]
```

In 3D space, the coordinates are labeled x, y, z, and w. To differentiate spatial coordinates from texture coordinates, the texels have correlating coordinates of s, t, u, and v. So this line of code creates texture coordinates that correlate to the x and y coordinates on the cube.

2) Invert and Scale

Next, add the following to the end of the string literal you just created:

```
st.y = 1.0 - st.y;
float2 ts = float2(st.x, 0.82 - st.y);
```

The first line of code is inverting the value of the y-coordinate by subtracting it from one. The second line is creating a new two-value vector where the first value is the x-coordinate of the first vector. The second value is scaling down the y-coordinate so that the shading image is small enough to fit on the cube with a margin.

3) Calculate the Triangle

Next, add:

```
float2 st2 = (st * 2.0 - 1.0) * 2.0;
float triSDF1 = max(abs(st2.x) * 0.866025 + st2.y * 0.5, -st2.y * 0.5);
float triSDFFill1 = 1.0 - step(0.7, triSDF1);
```

This is where the magic of the shader is being done. These properties are calculating how the triangle will be constructed one the cube.

4) Replicate Work on the ts Variable

Now, add:

```
float2 ts2 = (ts * 2.0 - 1.0) * 2.0;
float triSDF2 = max(abs(ts2.x) * 0.866025 + ts2.y * 0.5, -ts2.y * 0.5);
float triSDFFill2 = 1.0 - step(0.36, triSDF2);
```

This is the same work we did on the `st` vector. `SCNProgram` code injection does not currently support functions within text blocks, so we're unable to reuse these equations in a function by doing code injection in a string.

5) Fill Color

Next, add:

```
float fillColor = triSDFFill1;
fillColor -= triSDFFill2;
```

This is where the color of the pixel is determined. If `triSDFFill2` is 0, then it doesn't impact the value of `triSDFFill1` and the color will be white. If, however, it's 1, then it will cancel out the value of `triSDFFill1` and the color will be black. Therefore, anything that exists within the triangle will be shaded black while anything outside the triangle will be shaded white.

5) Shade the box

Finally, add:

```
_surface.diffuse = mix(float4(1.0), float4(0.0), fillColor);
```

This is the last line of the shader. It sets the diffuse color of the box.

6) That's it!

Congrats, at this time you should have a good understanding of the Metal Shading Language and `SCNProgram`! Take a break, and then it's time to move onto a full `SCNProgram` implementation and geometry warping.

48 Integrating Metal Shaders with SceneKit: Demo 3

By Janie Clayton

In this demo, you will learn how to replace SceneKit's rendering pipeline with custom Metal shaders.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo3\starter**.

1) Add Metal Data Structures

In the `Shaders.metal` file, replace the `MyVertexInput` structure with this:

```
typedef struct {
    float4 position [[ attribute(SCNVertexSemanticPosition) ]];
    float3 normal   [[ attribute(SCNVertexSemanticNormal) ]];
} MyVertexInput;
```

Next, add this data structure above the function declarations:

```
struct BrickVertex
{
    float4 position [[position]];
    float lightIntensity;
    float2 inputPosition;
};
```

The `BrickVertex` data structure is the return type you need for the vertex function. These properties are what you are going to pass to the fragment shader.

Leave the rest of the data structures alone.

2) Brick Vertex Constants

Next, add the following constants at the bottom of your file, below the red vertex and fragment shaders:

```
constant float SpecularContribution = 0.3;
constant float DiffuseContribution  = 1.0 - SpecularContribution;
constant float3 LightPosition = float3(0.0, -0.7071, -0.7071);
```

These values are lighting values necessary to calculate the per vertex lighting. These values can be passed in and determined by your main application as opposed to hard coded. If you want a challenge you can look into setting these up as inputs rather than constants.

3) Declaring the Brick Vertex Function

Beneath your constants, declare a new vertex function:

```
vertex BrickVertex brickVertex(MyVertexInput in [[ stage_in ]],
                               constant SCNSceneBuffer& scn_frame
[[buffer(0)]],
                               constant MyNodeBuffer& scn_node
[[buffer(1)]])
{
    BrickVertex vert;
```

`SCNProgram` requires you to pass in the `SCNSceneBuffer`. The `MyVertexInput` and `MyNodeBuffer` data structures includes any SceneKit parameters you need for your shader, so both of these have an expansive but limited number of options that are documented in the docs.

4) Brick Vertex Properties

Inside the curly brace after the `vert` declaration, add these propeties:

```
    float3 transformedPosition = (scn_node.modelViewProjectionTransform *
in.position).xyz;
    float3 transformedNormal      =
normalize((scn_node.modelViewProjectionTransform * float4(in.normal,
0.0)).xyz);
    float3 lightVec   = normalize(LightPosition - transformedPosition);
    float3 reflectVec = reflect(-lightVec, transformedNormal);
    float3 viewVec    = normalize(-transformedPosition);
    float diffuse     = max(dot(lightVec, transformedNormal), 0.0);
    float specular        = 0.0;
```

These properties represent the calculated properties you will need to calculate your

per-vertex lighting. These are calculated based on parameters you passed in through the various SceneKit buffers.

5) Finish Brick Vertex Logic

Complete your vertex function with the following code:

```
    if (diffuse > 0.0) {
        specular = dot(reflectVec, viewVec);
        specular = pow(specular, 16.0);
    }

    float lightIntensity = DiffuseContribution * diffuse +
SpecularContribution * specular;
    float2 inputPosition = in.position.xy;

    vert.position = scn_node.modelViewProjectionTransform * in.position;
    vert.lightIntensity = lightIntensity;
    vert.inputPosition = inputPosition;

    return vert;
}
```

The if statement is checking to ensure that there is enough light to make it worth calculating the value. If you are looking behind the cube there is no point in calculating the lighting.

The bottom block of three lines of code are setting the three vertex properties you specified in the brick vertex structure. These will be passed to the fragment shader

6) Create the Fragment Constants

Before you set up the fragment shader, you need a few fragment-specific constants. Add these below the brick vertex shader:

```
constant float3 BrickColor = float3(0.52, 0.12, 0.15);
constant float3 MortarColor = float3(0.91, 0.90, 0.90);
constant float2 BrickSize = float2(0.3, 0.15);
constant float2 BrickPct = float2(0.9, 0.85);
```

These constants represent the colors and sizes of the bricks and the mortar. Again, these are properties you could make customizable and pass in through a buffer if you're looking for a challenge.

7) Create the Brick Fragment Function

Beneath the fragment constants, add the following code:

```
fragment half4 brickFragment(BrickVertex in [[stage_in]])
{
    half4 returnColor;
    float3 color;
    float2 position, useBrick;
```

The fragment shader only returns a single value, which represents the color. It also takes in a single parameter, which is the data structure that you returned at the end of the vertex shader.

Again, you create a few properties here to hold values necessary to calculate the color. You want to keep the fragment shader as light-weight as possible because it is called for every pixel.

8) Brick Position Calculations

Next, add the following calculations:

```
    position = in.inputPosition / BrickSize;

    if (fract(position.y * 0.5) > 0.5) {
        position.x += 0.5;
    }

    position = fract(position);
```

These calculations are determining if the current position is located on a brick or on the mortar. The if statement is creating an offset every other row so that the brick pattern is a little more interesting.

9) Finish Brick Fragment Shader

Finish your brick fragment shader as follows:

```
    useBrick = step(position, BrickPct);

    color = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
    color *= in.lightIntensity * 1.5;
    returnColor = half4(half3(color), 1.0);

    return returnColor;
}
```

The first line here is determining if the fragment is on a brick or on the mortar. `step` functions are binary and return either a 1 or a 0. This is used in the next line within the `mix` function. If the fragment is between the bricks, the mix will be weighted 100% to the mortar color. Otherwise, the mix is 100% weighted to the brick color.

The calculated light intensity from the vertex shader is blended into the default color. Finally these values are assigned to the proper data type and returned from the fragment shader

Lastly, you will need to swap out the red shaders for the brick shaders in the `addBox()` function in `GameViewController.swift`:

```
//        program.vertexFunctionName = "redVertex"
//        program.fragmentFunctionName = "redFragment"
        program.vertexFunctionName = "brickVertex"
        program.fragmentFunctionName = "brickFragment"
```

10) That's it!

Congrats, at this time you should have a good understanding of integrating Metal shaders into SceneKit!