啥是馒头(Metal)

纹理(一)

UV 坐标

UV 坐标是用于描述纹理贴纸的坐标系。所有的图像文件都是一个二维的平面，水平方向为 U 轴，垂直方向为 V 轴。通过这个二维的 UV 坐标系，可以定位图像上的任意一个像素。纹理的 UV 坐标可以通过映射将模型表面的点和平面图像上的像素对应起来，这样就可以在模型表面上定位纹理贴图了。

在我们用建模工具导出一个带有纹理的模型到 .obj 文件的时候，会将纹理 UV 坐标的数据同时导出在 obj 文件中。我们可以打开 Resource 文件夹下的 monkey.obj 文件一探究竟。在用记事本打开 obj 文件后，我们可以看到紧跟着顶点数据之后，会有 vt 开头的纹理坐标数据，它代表的就是当前顶点映射的纹理贴图中的纹理坐标数据。

如果你想自己动手搭一个简单的 3d 模型，除了用 PhotoShop，我推荐使用免费的 Blender 工具。当然还有其他收费的建模工具，比如 Substance Designer 和 Substance Painter、3DCoat 等等。

如何给模型贴纹理

接下来我们开始实践给模型贴纹理，首先在工程的 Resource 文件夹下，我们有完整的模型文件以及贴图文件，先将它们导入到项目中来。接下来打开本章 Texture 目录下的 Start 文件夹，打开工程文件。没错，工程文件就是上一章 Transform 中的 final 工程，我们接着给这个猴子贴图，让它变成一只更好(chou)看(lou)的猴子。

首先打开 Shaders.metal 文件，看到 fragment_main 的片元处理函数。这里以前的代码是返回 float4(0, 1, 0, 1) 的颜色值，也就是为什么我们看到的是一只绿猴子的原因。接下来我们将改造这个函数，实现给猴子贴纹理的功能。

那么如何在片元处理函数中读取纹理贴图呢，我们将过程主要分为以下几步： 1. 将纹理的 UV 坐标添加到 model 的顶点描述器（vertex descriptor）中。 2. 在 shader 中的 VertexIn 结构中添加一个属性来匹配该顶点的 UV 坐标。 3. 写一个函数来负责加载纹理图片。 4. 在绘制 model 之前将加载进来的纹理贴图传给 fragment 函数。 5. 在 fragment 函数中实现从纹理贴图中对像素进行采样

接下来我们会根据这几个主要步骤添加相关的代码实现

1.将纹理的 UV 坐标添加到 model 的顶点描述器（vertex descriptor）中

首先在 Common.h 文件中新增 Attributes 的枚举类型，用于 表示index 值。

```
typedef enum {
  Position = 0,
  Normal = 1,
  UV = 2
} Attributes;
```

然后在 Model.swift 文件中，给 defaultDescriptor 添加一个新属性。

```
 vertexDescriptor.attributes[Int(UV.rawValue)] = MDLVertexAttribute(name: MDLVerte
xAttributeTextureCoordinate, format: .float2, offset: 12, bufferIndex: 0)

         vertexDescriptor.layouts[0] = MDLVertexBufferLayout(stride: 20)
```

这里由于 position 的顶点位置数据已经占有 12 字节，float 4个字节乘以 x，y，z 三个数据。所以 textureCoordinate 的数据将从 offset 为 12 的地方开始读取。最后由于添加了 textureCoordinate 的数据，所以我们需要将 layout 的跨度加上 8 byte (textureCoordinate 数据为 float 4字节 * u、v两个数据)。

2.更新 Shader 中的 VertexIn 结构

在 shader 文件中，顶点处理函数的参数 vertexIn 通过 stage_in 属性与顶点描述器中的 layout 布局绑定。由于上面我们已经给顶点描述器添加了 textureCoordinate 属性，所以现在我们可以通过给 vertexIn 添加一个 UV 属性来获取顶点描述器中传递过来的 texturecoordinate 数据。

```
struct VertexIn {
    float4 position [[ attribute(Position) ]];
    float2 uv [[ attribute(UV) ]];
};
```

当然，与之对应的，在传递给片元处理器的数据结构中也要加上 uv 属性，所以我们要在 VertexOut 中添加上 uv 属性。

```
struct VertexOut {
    float4 position [[ position ]];
    float3 worldPosition;
    float2 uv;
};
```

然后在顶点处理函数 vertex_main() 中，在函数返回前添加 vertexout 从 vertexin 中赋值 uv 数据的代码。

```
out.uv = vertexIn.uv;
```

3.加载纹理图片

首先新建一个 swift 文件叫 Texturable.swift 用于负责加载纹理图片。然后添加一个 Texturable 的协议。

```
import MetalKit

protocol Texturable {}

extension Texturable {
}
```

然后在 Texturable 的扩展中，添加 loadTexture 函数。

```
    static func loadTexture(imageName: String) throws -> MTLTexture? {
        //5
        let textureLoader = MTKTextureLoader(device: Renderer.device)

        //6
        let textureLoaderOptions : [MTKTextureLoader.Option : Any] =
            [.origin:
                MTKTextureLoader.Origin.bottomLeft]

        //7
        let fileExtension = URL(fileURLWithPath: imageName).pathExtension.isEmpty
? "png" : nil

        //8
        guard let url = Bundle.main.url(forResource: imageName, withExtension: fil
eExtension) else {
            print("Failed to load")
            return nil
        }

        let texture = try textureLoader.newTexture(URL: url, options: textureLoade
rOptions)
        print("loaded texture")
        return texture;
    }
```

然后在 SubMesh.swift 文件中，修改 SubMesh 使其遵循 Texturable 的协议,使 SubMesh 可以加载纹理图片。

```
extension Submesh : Texturable {

}
```

Model I/O 可以方便地将整个模型以及所有的材质都加载进来。点击查看 mtl 文件，可以在最下方看到 map*Kd monkey.png* 。*map*Kd 表示为漫反射指定颜色纹理文件(.mpc)或程序纹理文件(.cxc)，或是一个位图

文件。所以 Model I/O 在加载 mtl 文件时可以同时拿到 monkey.png 的纹理贴图文件。

由于每个 Submesh 可能对应不同的纹理，所以在 Submesh 中，我们需要创建一个 Textures 的结构体，并且持有一个 Textures 的属性。

```
struct Textures {
  let baseColor: MTLTexture?
}
let textures: Textures
```

Textures 结构体的属性是根据 MDLMaterialSemantic 的属性来定义的。MDLMaterialSemantic 表示对 Material 的语义描述，可以通过 semantic 获取到 material 对应的属性值。

接下来在 Submesh 文件中添加 Textures 的初始化函数

```
private extension Submesh.Textures {
  init(material: MDLMaterial?) {
    func property(with semantic: MDLMaterialSemantic) -> MTLTexture? {
      guard let property = material?.property(with: semantic),
        property.type == .string,
        let filename = property.stringValue,
        let texture = try? Submesh.loadTexture(imageName: filename)
else {
return nil
}
      return texture
    }
    baseColor = property(with: MDLMaterialSemantic.baseColor)
  }
}
```

上述初始化函数从 material 中加载了 base color texture 贴图，这里的 base Color 表示的是漫反射光的基础颜色。后面的章节中，会涉及到加载镜像反射的纹理，加载的过程都是类似的。

到这里，我们可以 run 一下看控制台输出，如果有输出 loaded texture 的话说明纹理已经可以成功加载啦。

4.在绘制 model 之前将加载进来的纹理贴图传给 fragment 函数

在之后的章节中，我们会用到不同的纹理类型，不同的纹理类型会用不同的索引来传递给片元处理函数。所以我们现在先在 Common.h 中创建一个新的枚举类型 Textures 用于区分不同的纹理缓冲区索引值。

```
typedef enum {
  BaseColorTexture = 0
} Textures;
```

然后在 Renderer.swift 中的 draw 函数，找到//9 的注释处，在处理每个 submesh 的代码中，修改代码如

下：

```
      for modelSubmesh in model.submeshes {
                renderEncoder.setFragmentTexture(modelSubmesh.textures.baseColor,
index: Int(BaseColorTexture.rawValue))
                let submesh = modelSubmesh.submesh
                renderEncoder.drawIndexedPrimitives(type: .triangle,
                                                    indexCount: submesh.indexCount
,
                                                    indexType: submesh.indexType,
                                                    indexBuffer: submesh.indexBuff
er.buffer,
                                                    indexBufferOffset: submesh.ind
exBuffer.offset)
            }
```

这样我们就通过 renderEncoder 将纹理传递给了 fragment 函数，并且保存在 texture buffer 0中。

所有的 Buffers，Textures 和 Sampler states 都是被一张参数表所持有的。我们是通过索引值来获取相对应的 buffer、texture 或者 sampler state。你可以将这个索引值理解为一个句柄，我们通过句柄去获取我们需要的东西。在 iOS 中，最多可以存在 31 个 buffer 缓冲区，31个纹理缓冲区以及 16 个采样器状态值。在 MacOS 中，纹理缓冲区的最大数量可以有 128 个。

5. 在 fragment 函数中实现从纹理贴图中对像素进行采样

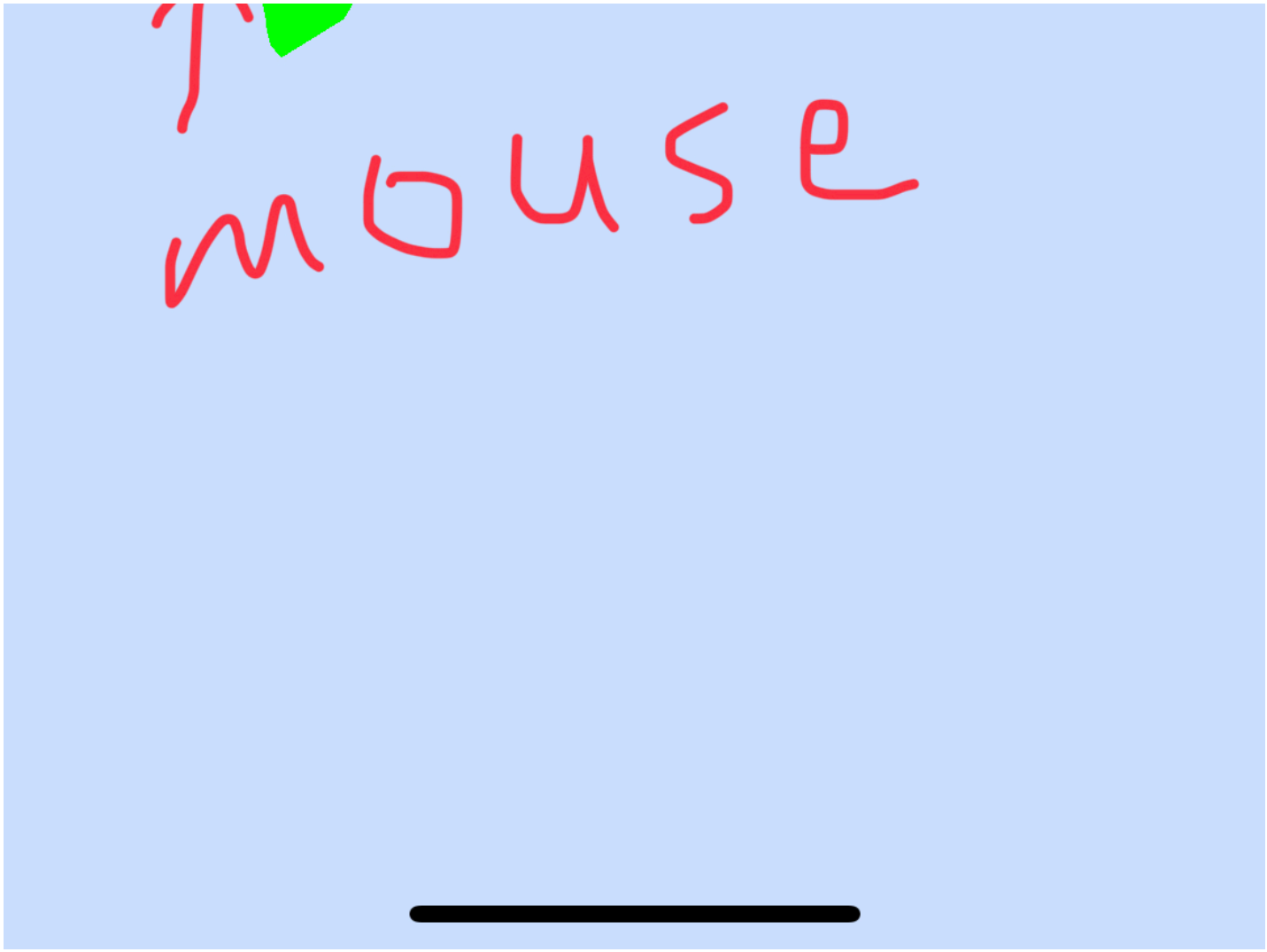
接下来我们需要修改片元处理函数，去接收并且读取纹理数据。 首先我们在 Shaders.metal 文件的 fragment_main,也就是片元处理函数中添加一个新的参数代表 BaseColor 的纹理。

```
texture2d<float> baseColorTexture [[ texture(BaseColorTexture) ]],
```

当我们在对一个纹理进行读取或者采样时，我们不一定刚好对某一个像素能够采样到贴切的值。在纹理中，我们对纹理进行采样的基本单元叫做纹素(Texels)。我们可以通过采样器(Sampler)来决定如何去采样每个纹素。现在我们可以先在片元处理函数中创建一个最简单的默认采样器,修改 fragment_main 函数内容如下：

```
fragment float4 fragment_main(VertexOut in [[stage_in]],
                              texture2d<float> baseColorTexture [[ texture(BaseCol
orTexture)]]) {
    constexpr sampler textureSampler;
    float3 baseColor = baseColorTexture.sample(textureSampler, in.uv).rgb;
    return float4(baseColor ,1);
}
```

最后为了便于观察结果，我们将 metalView 的 clearColor 改为 RGB(202,225,255)，然后 run 一下，可以看到结果：



最后

本章节的主要工作是将纹理贴纸渲染出来，接下来的章节中会继续介绍 Samplers 采样、Mipmaps 以及其他相关内容。