BEGINNING METAL

HANDS-ON CHALLENGES

Beginning Metal

Caroline Begbie

Notice of Rights

Notice of Liability

Trademarks

Challenge #4: Scene Graph

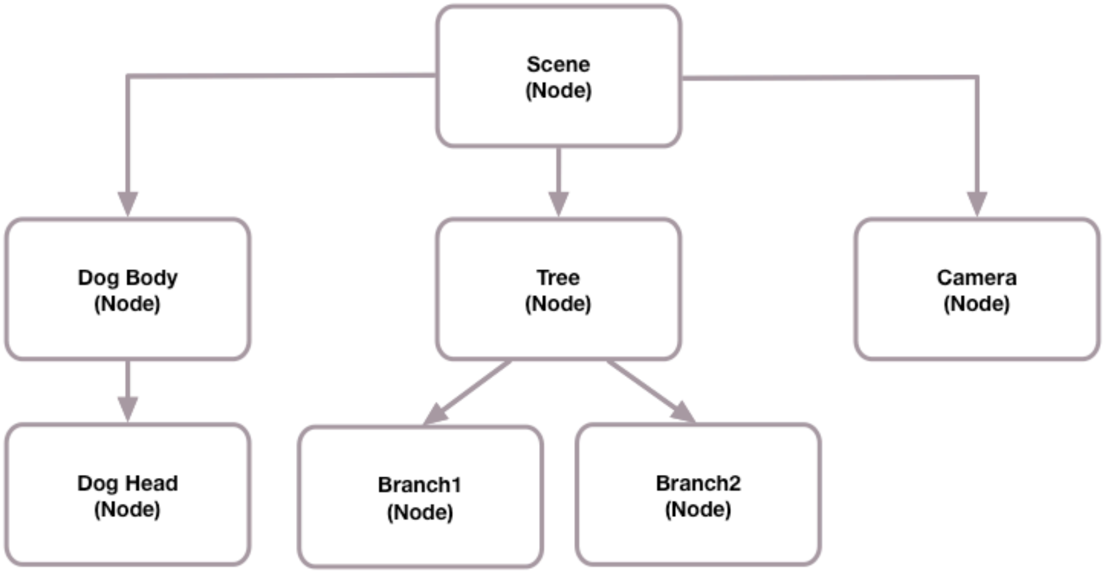
By Caroline Begbie

Your challenge is to restructure our current app to have a scene graph. There are two main reasons for doing this.

The first is to abstract away the construction of the scene and game play from the Metal rendering code. When we come to create the game in the last two videos, we won't even be looking at the Metal code, so we can concentrate on how the game works and not how Metal works.

The second is so that we can create scenes easily. We'll be able to add models just by adding them to a scene. We'll also be able to transition easily between scenes.

A scene graph is a hierarchy of nodes. For example, if you had a scene with a dog, a tree and a camera, these would all be nodes. The dog might have a body and a nodding head. The body node might be a child of the scene node, and the head node a child of the body node.



The node will be a Swift class. It will have an array property that holds its child nodes and a method to add children.

The node will also have a recursive render method, so that all we have to do is tell the scene node to render, and all its child nodes will recursively render.

Later on we'll add position data. When we move the position of the scene, then all the child nodes will move by that same position. Going back to the dog example, if we move the dog's body node, the head, being a child node, will move with it, but we could also move the head child node without affecting the dog's body. The body node is termed the **parent** of the head node.

Create a new Swift file and call it **Node.swift**.

Replace the contents of this file with:

```
import MetalKit

class Node {
}
```

Create two properties in `Node` to contain the node's name and the node's children.

```
var name = "Untitled"
var children: [Node] = []
```

Add a new method to add a child node to the node:

```
func add(childNode: Node) {
  children.append(childNode)
}
```

Add the recursive render method:

```
func render(commandEncoder: MTLRenderCommandEncoder,
            deltaTime: Float) {
  for child in children  {
    child.render(commandEncoder: commandEncoder,
                 deltaTime: deltaTime)
  }
}
```

We'll send as parameters the command encoder and the time since the previous frame.

Now create a new Swift file to hold the details of the quad. We'll call this **Plane.swift**. It's a useful class to have - planes can be used for many things - backgrounds and ground for example.

Replace the contents of **Plane.swift** with:

```
import MetalKit

class Plane: Node {
}
```

Plane is a subclass of `Node`, so it can be added to a scene.

This class will hold all the code from `Renderer` that applies solely to the plane.

Move these properties from `Renderer` to `Plane`:

```
var vertexBuffer: MTLBuffer?
var indexBuffer: MTLBuffer?

var vertices: [Float] = [
    -1,  1, 0,
    -1, -1, 0,
     1, -1, 0,
     1, 1, 0]

var indices: [UInt16] = [
    0, 1, 2,
    0, 2, 3
]

var time: Float = 0

struct Constants {
  var animateBy: Float = 0
}

var constants = Constants()
```

Move the `buildModel()` method from `Renderer` to `Plane` and rename it to `buildBuffers(device:)`:

```
private func buildBuffers(device: MTLDevice) {
  vertexBuffer = device.makeBuffer(bytes: vertices,
                     length: vertices.count * MemoryLayout<Float>.size,
                     options: [])
  indexBuffer = device.makeBuffer(bytes: indices,
                     length: indices.count * MemoryLayout<UInt16>.size,
                     options: [])
}
```

and create an initializer in `Plane`:

```
init(device: MTLDevice) {
  super.init()
  buildBuffers(device: device)
}
```

Remove `buildModel()` from `Renderer`, and remove the code that calls it in `init(device:)`.

Don't worry about compilation errors in `Renderer` yet - we'll fix all that up shortly.

In **Plane.swift**, override `Node`'s `render(commandEncoder:deltaTime:)` using some of the code from `Renderer`:

```
override func render(commandEncoder: MTLRenderCommandEncoder,
                     deltaTime: Float) {
  super.render(commandEncoder: commandEncoder,
               deltaTime: deltaTime)

  guard let indexBuffer = indexBuffer else { return }

  time += deltaTime
  let animateBy = abs(sin(time)/2 + 0.5)
  constants.animateBy = animateBy

  commandEncoder.setVertexBuffer(vertexBuffer, offset: 0, at: 0)
  commandEncoder.setVertexBytes(&constants,
                                length: MemoryLayout<Constants>.stride,
                                at: 1)
  commandEncoder.drawIndexedPrimitives(type: .triangle,
                                       indexCount: indices.count,
                                       indexType: .uint16,
                                       indexBuffer: indexBuffer,
                                       indexBufferOffset: 0)
}
```

This code should be familiar to you from `Renderer`.

Create a new Swift file for the `Scene` class called **Scene.swift**.

Replace the contents of the file with:

```
import MetalKit

class Scene: Node {
}
```

`Scene` is a subclass of `Node`. When we want to render the scene, the scene will render all its children recursively.

The scene will need to store the device and the size of the scene. When we instantiate the scene, we'll set the size to be the view's bounds.

Add these two properties to **Scene.swift**:

```
var device: MTLDevice
var size: CGSize
```

And create the initializer:

```
init(device: MTLDevice, size: CGSize) {
  self.device = device
  self.size = size
  super.init()
}
```

In our game we may have several scenes. We might have different game levels and we might have scenes that show when the player wins or loses. Each of these scenes will be a subclass of `Scene`. `Scene` will only hold things that are common to all scenes. Later we'll add lighting and a camera to `Scene`.

For the moment we'll just have one game scene. Create a new file called **GameScene.swift** and replace the contents with this:

```
import MetalKit

class GameScene: Scene {

}
```

This is where we'll add the quad to the scene. In `GameScene`, create a property for the quad:

```
var quad: Plane
```

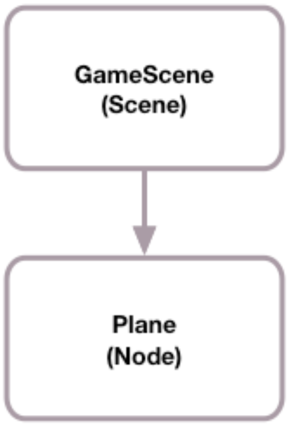
And create the initializer:

```
override init(device: MTLDevice, size: CGSize) {
  quad = Plane(device: device)
  super.init(device: device, size: size)
}
```

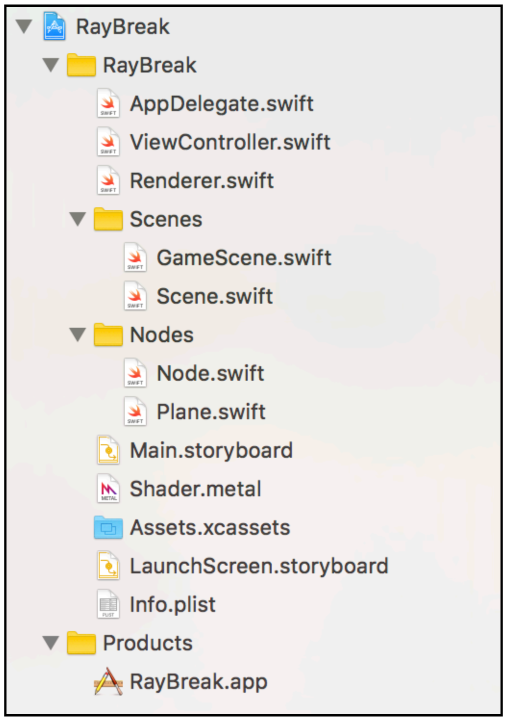
At the end of the initializer, add the quad to the scene as a child node:

```
add(childNode: quad)
```

Now we have a `GameScene`, which is a subclass of `Scene`. We have a `Plane` which we've added as a child to `GameScene`. `Scene` and `Plane` are subclasses of `Node`, so we now have the start of a scene graph.



Tidy up the project by placing **GameScene.swift** and **Scene.swift** in a group called **Scenes**, and place **Node.swift** and **Plane.swift** in a group called **Nodes**



We just need to fix up `Renderer`. Add a new `scene` property to `Renderer`:

```
var scene: Scene?
```

Change `draw(in:)` to:

```
func draw(in view: MTKView) {
  guard let drawable = view.currentDrawable,
    let descriptor = view.currentRenderPassDescriptor,
    let pipelineState = pipelineState
    else { return }

  let commandBuffer = commandQueue.makeCommandBuffer()
  let commandEncoder =
        commandBuffer.makeRenderCommandEncoder(descriptor: descriptor)

  commandEncoder.setRenderPipelineState(pipelineState)

  let deltaTime = 1/Float(view.preferredFramesPerSecond)

  scene?.render(commandEncoder: commandEncoder, deltaTime: deltaTime)

  commandEncoder.endEncoding()
  commandBuffer.present(drawable)
  commandBuffer.commit()
}
```

Here we've removed the code that is now in `Plane`. We are rendering the `scene` using the `commandEncoder` and the time since the last frame (this may not be completely accurate if frames are taking longer than the `preferredFramesPerSecond` to render).

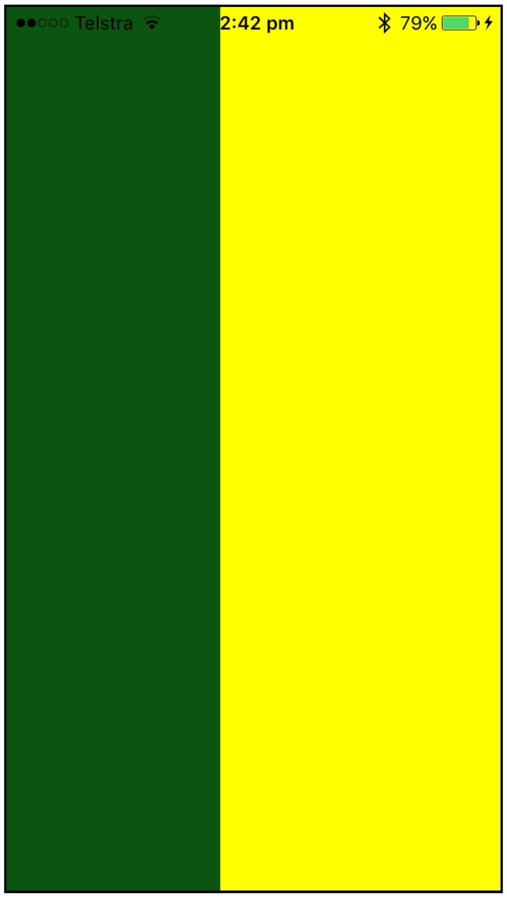
Your project should now compile properly.

One final thing to do - set `Renderer`'s scene.

In **ViewController.swift**, in `viewDidLoad()`, add this line after instantiating `renderer`:

```
renderer?.scene = GameScene(device: device, size: view.bounds.size)
```

Build and run, and the yellow quad should be animating in exactly the same way as before.



The code is much improved now because we have abstracted away the Metal code and created a scene graph. You could easily add a second quad to the scene, although you wouldn't see it because it would lie on top of the first quad. Later we'll add multiple models to this scene.

There is one flaw with our code which we'll fix up as we go along. The render method in `Plane` has animation code in it. As we might have several planes in our app, this animation code should really be in `GameScene`. Later on we'll create an update method in the scene that's updated every frame.