Adopting Metal, Part 1

Session 602

Warren Moore GPU Software Engineer
Matt Collins GPU Software Engineer

Metal

Apple's low-overhead API for GPUs

Unified graphics and compute

Built for efficient multithreading

Designed for Apple platforms

Metal

Supporting Technologies

MetalKit

Metal Performance Shaders

Xcode and Instruments

- Metal Compiler
- Frame Debugger
- Metal System Trace

iOS

X

iOS

tvOS


MacBook Air

Preview

UIKit

Maps

Safari

Core Image

Core Graphics

OpenGL ES

SpriteKit

Core Animation

Model I/O

SceneKit

EA GAMELOFT Disney naturalmotion GAMES

EPIC GAMES UBISOFT DIGITAL LEGENDS ENTERTAINMENT® HEAVY IRON STUDIOS REBELLION

ROVIO unity SUPER EVIL MEGACORP BLIZZARD® ENTERTAINMENT FIRI GAMES

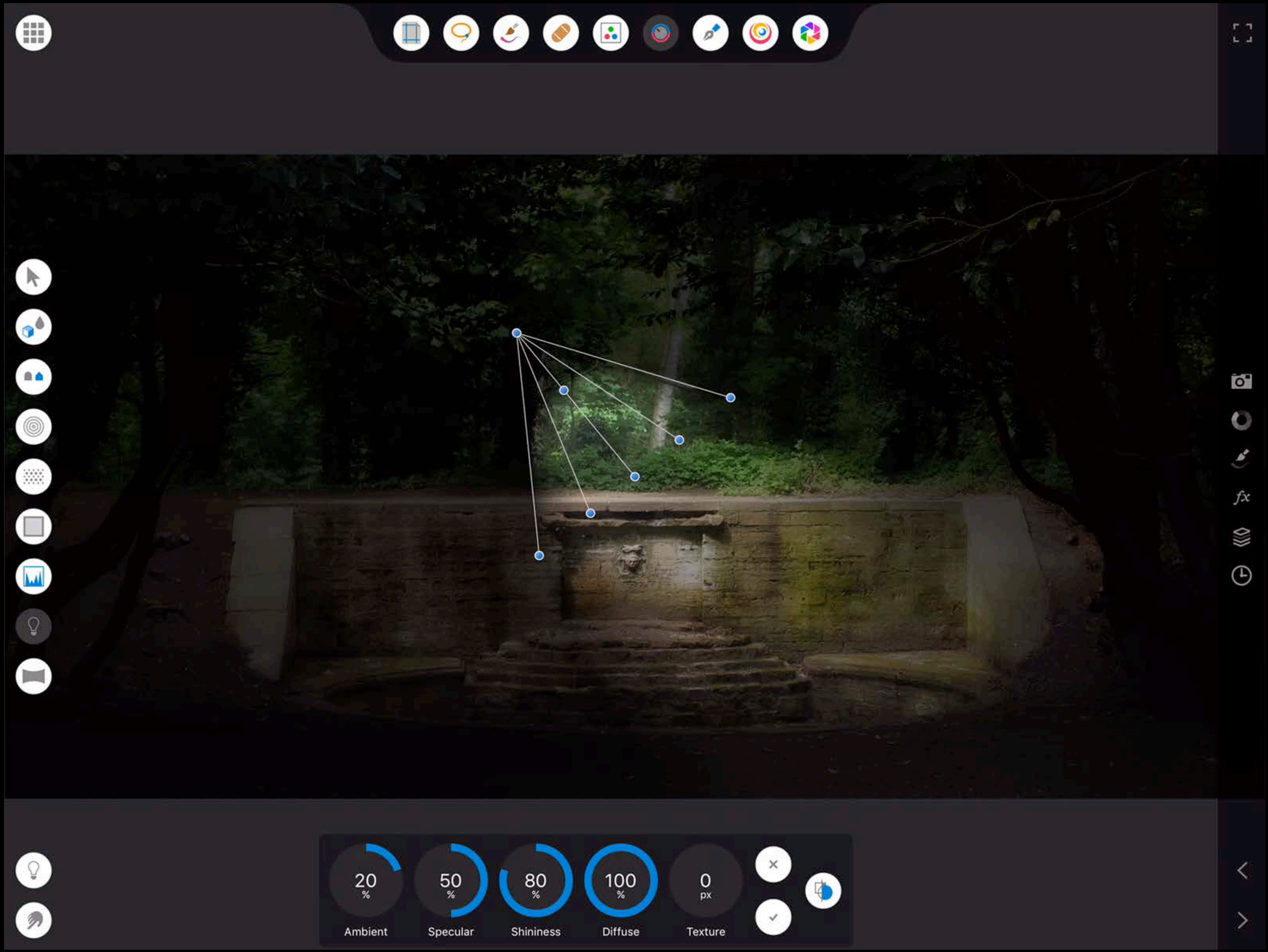
DOUBLE FINE Adobe CALVINO • NOIR zynga

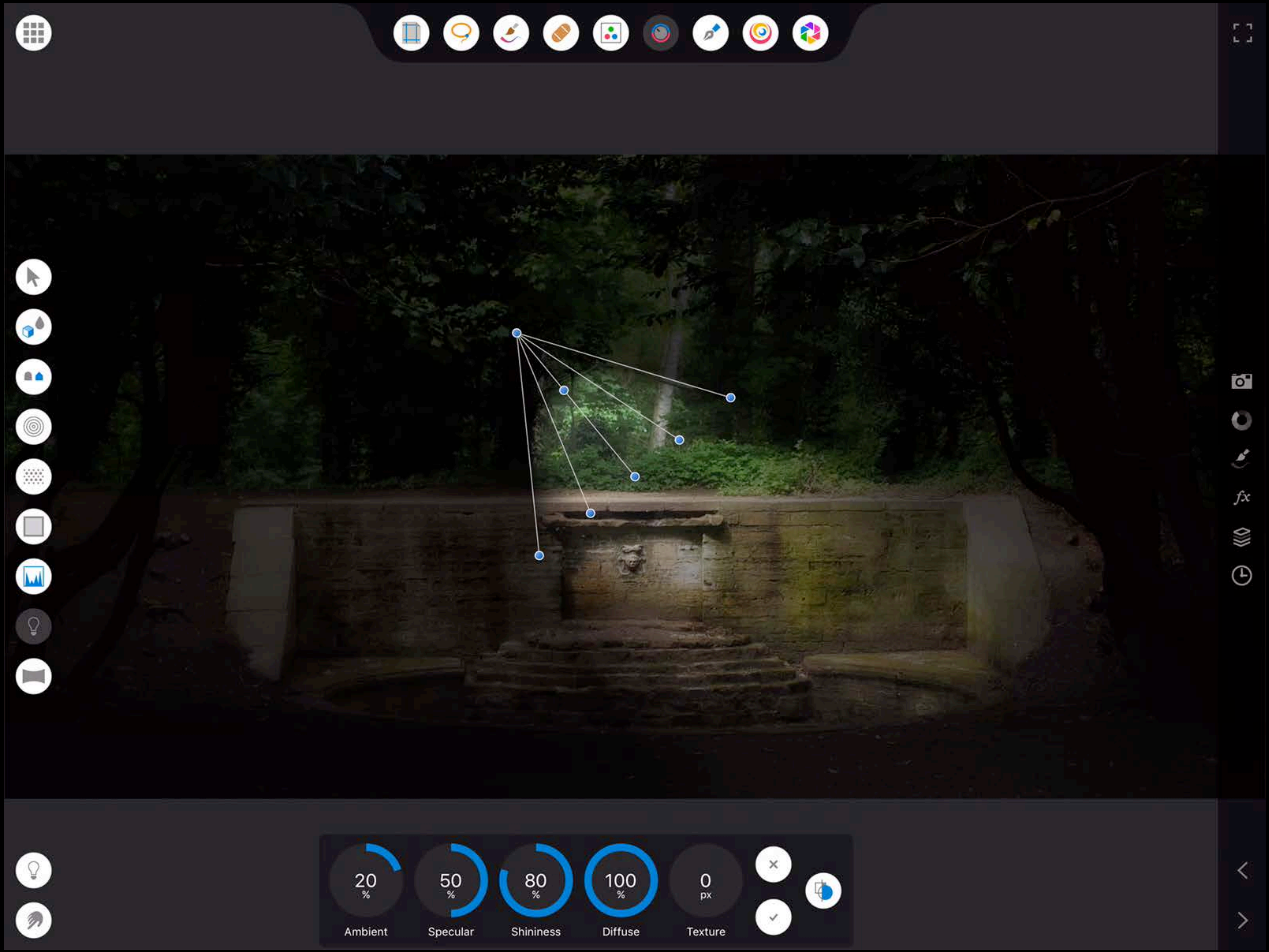
ATYPICAL GAMES pixelbite SQUARE ENIX VECTOR UNIT

PIXELMATOR TEAM PROLETARIAT TECHLAND® GOOD GAMES FANCYGUO

Furious Wings

Affinity


20 %
Ambient
50 %
Specular
80 %
Shininess
100 %
Diffuse
0 px
Texture



20 %
Ambient
50 %
Specular
80 %
Shininess
100 %
Diffuse
0 px
Texture

Metal at WWDC This Year

A look at the sessions

Adopting Metal

Part One

- Fundamental Concepts
- Basic Drawing
- Lighting and Texturing

Part Two

- Dynamic Data Management
- CPU-GPU Synchronization
- Multithreaded Encoding

Metal at WWDC This Year

A look at the sessions

What's New in Metal

Part One

- Tessellation
- Resource Heaps and Memoryless Render Targets
- Improved Tools

Part Two

- Function Specialization and Function Resource Read-Writes
- Wide Color and Texture Assets
- Additions to Metal Performance Shaders

Metal at WWDC This Year

A look at the sessions

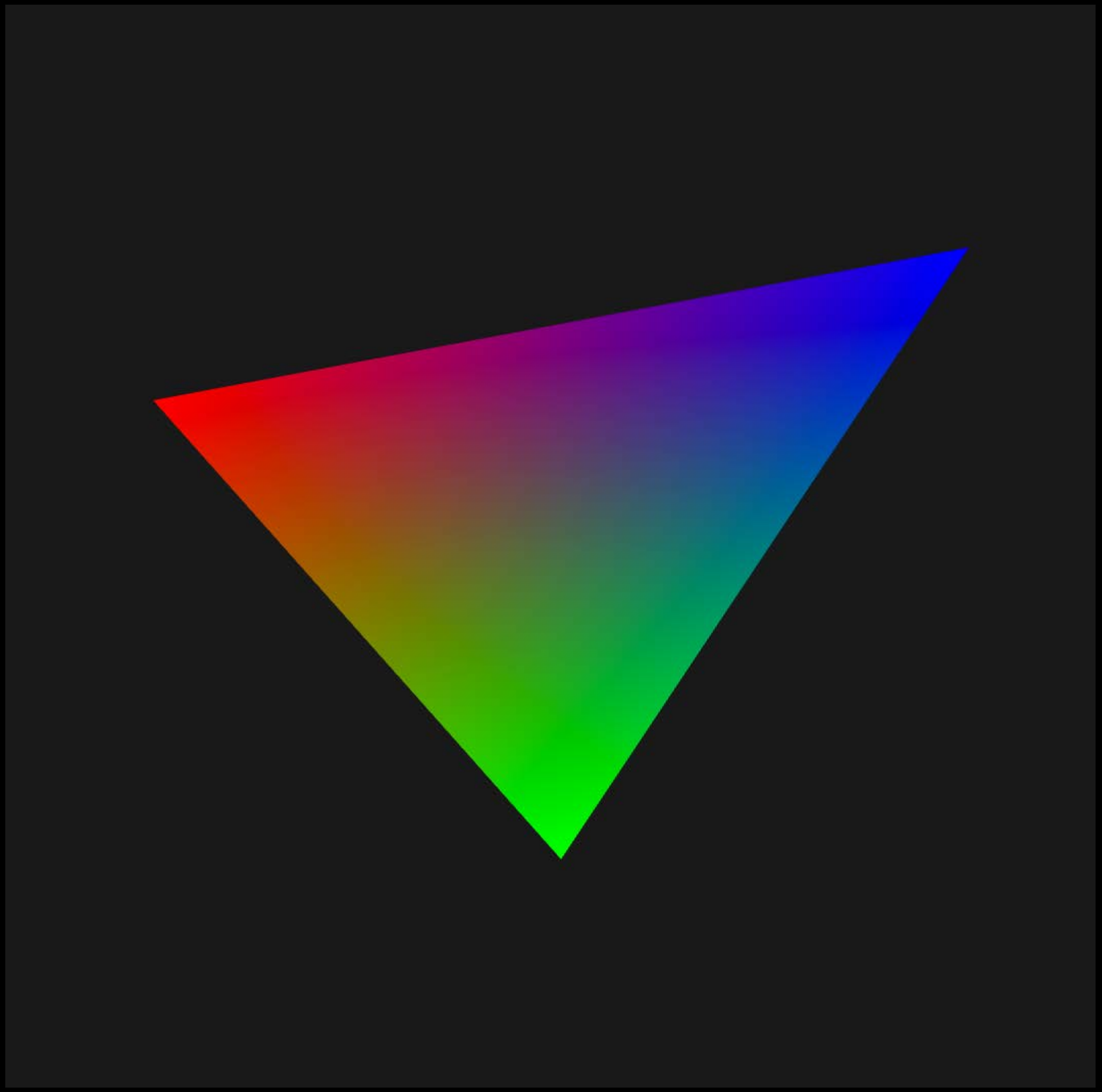
Advanced Shader Optimization

- Shader Performance Fundamentals
- Tuning Shader Code

The Sample Project

This session

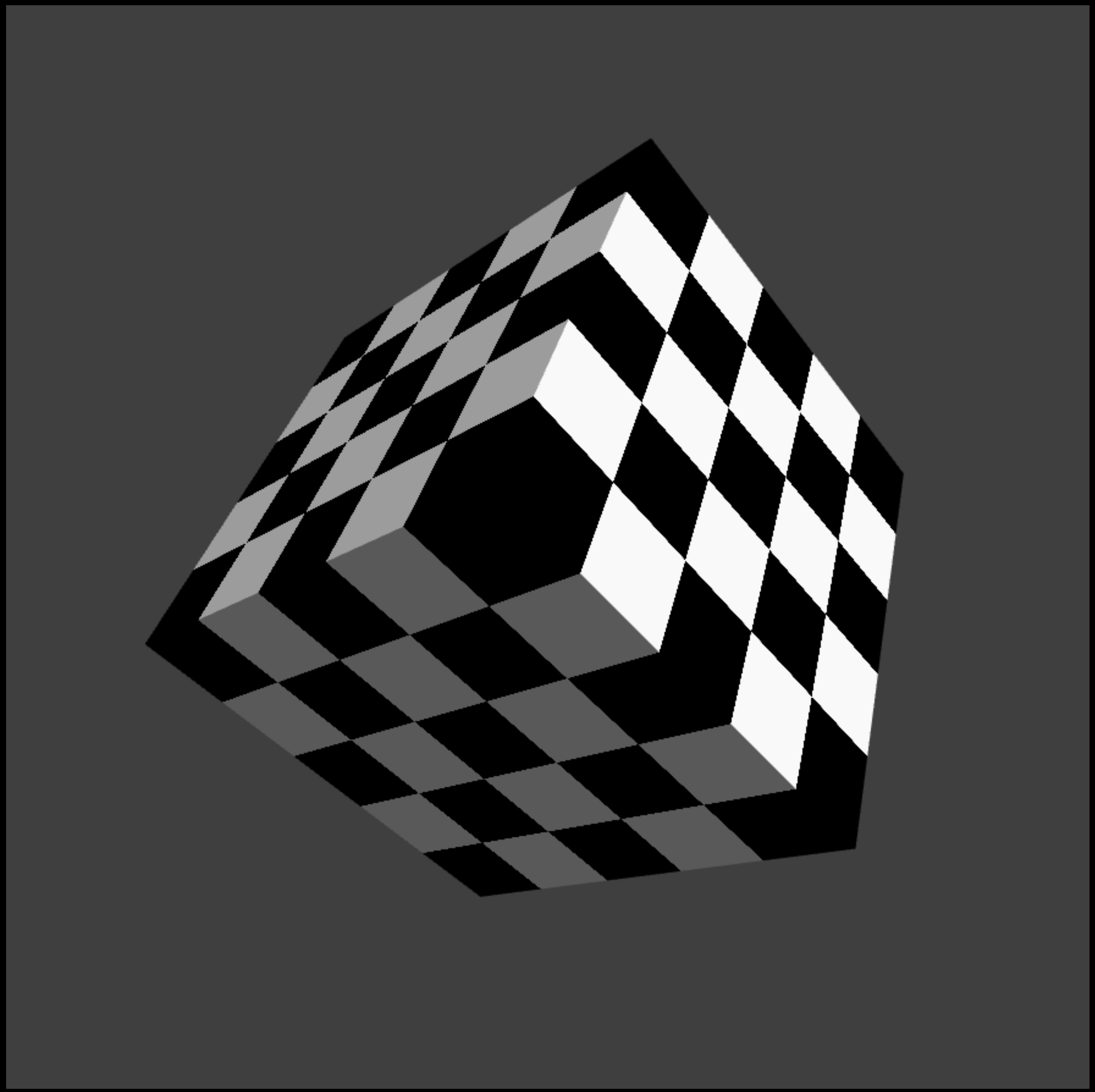
- Geometry



The Sample Project

This session

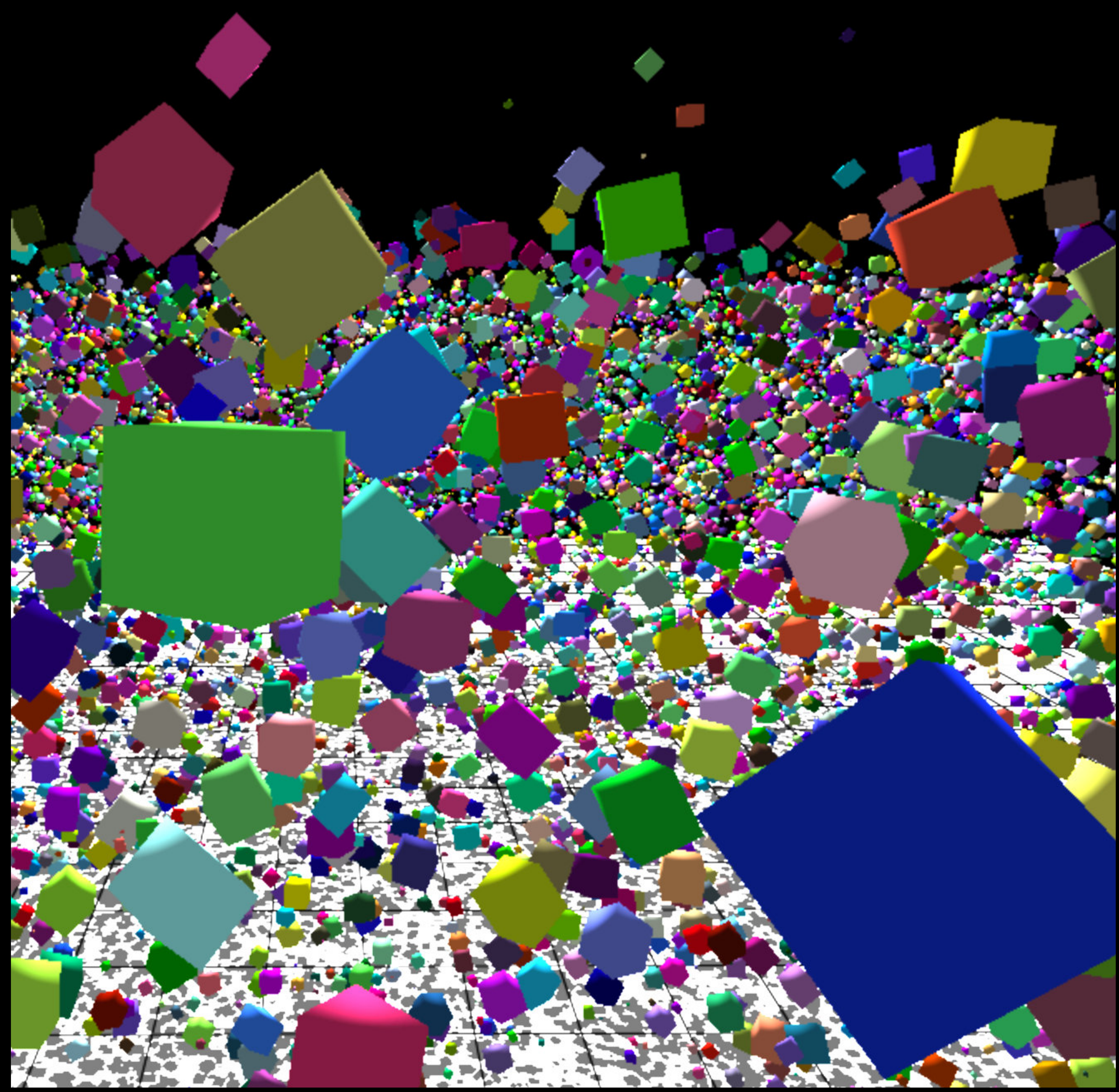
- Geometry
- Animation
- Texturing



The Sample Project

Adopting Metal Part 2

- Updating object data
- Multithreaded draw calls



Assumptions

You're familiar with the fundamentals of graphics programming

You have experience with a graphics API that has shaders

You're interested in using Metal to make your games and apps even more awesome

Agenda

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Conceptual Overview

Use an API that matches the hardware and driver

Favor explicit over implicit

Do expensive work less often

An API that Matches the Hardware

Thoroughly modern

- Integrates with and exposes latest hardware features
- Thin API with no historical cruft

No fancy tricks required for low-overhead operation

Uniform across platforms (OS X, iOS, tvOS)

Favor Explicit over Implicit

…when implicit has a high cost

Command submission model maps closely to actual hardware operation

Explicit control over memory and synchronization

With great responsibility comes great performance

Do Expensive Work Less Often

When	How Often
Build time	—
Load time	Infrequently
Draw time	Many times per second

Do Expensive Work Less Often

When	How Often	OpenGL
Build time	—	—
Load time	Infrequently	—
Draw time	Many times per second	State validation Shader compilation Encode work for GPU

Do Expensive Work Less Often

When	How Often	OpenGL	Metal
Build time	—	—	Shader compilation
Load time	Infrequently	—	State validation
Draw time	Many times per second	State validation Shader compilation Encode work for GPU	Encode work for GPU

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

MTLDevice

Abstract representation of a GPU

The “root object” of the Metal object graph

Used to create resources, pipeline state objects, and command queues

```swift
// MTL Device
// API


let device = MTLCreateSystemDefaultDevice()
```

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

Multithreaded Encoding

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

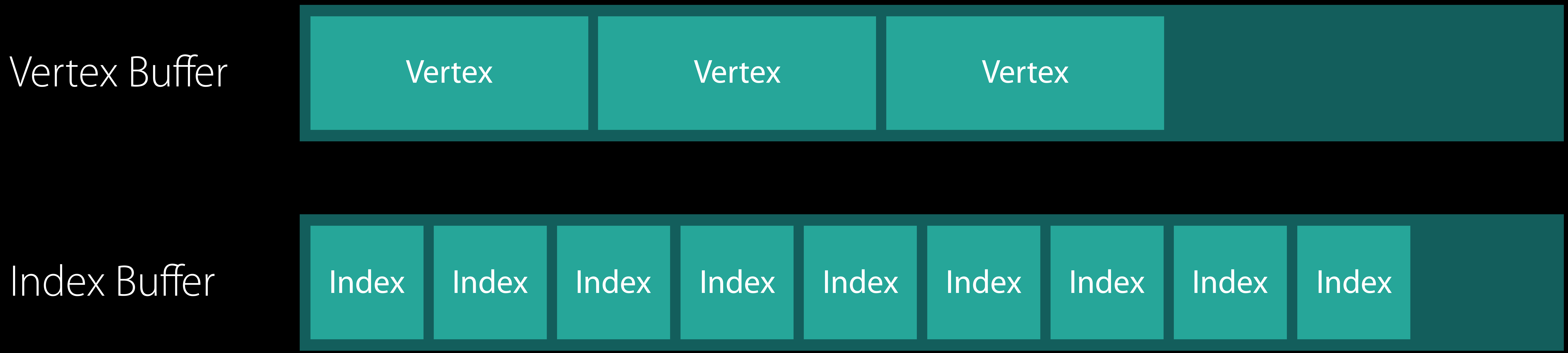
Buffers

Allocations of memory that can store data in a format of your choosing

Vertex data, index data, constants

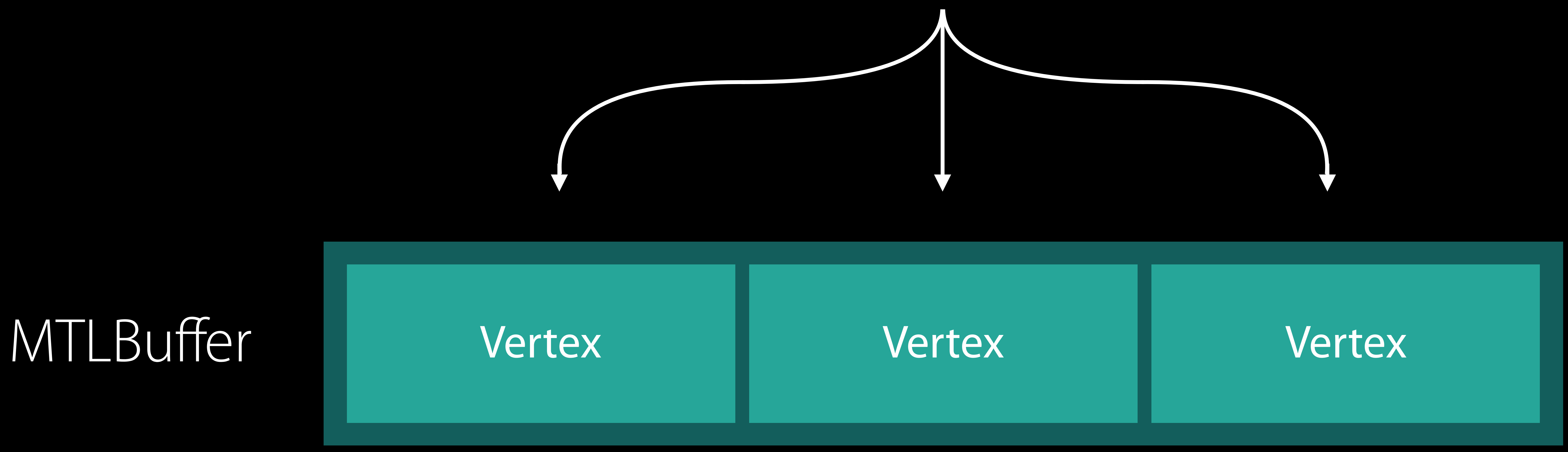
Can be read in vertex and fragment functions

Buffer Layout



Buffers as Arrays of Structures

```
struct Vertex {
    var position: float4
    var color: float4
}
```



MTLBuffer

API

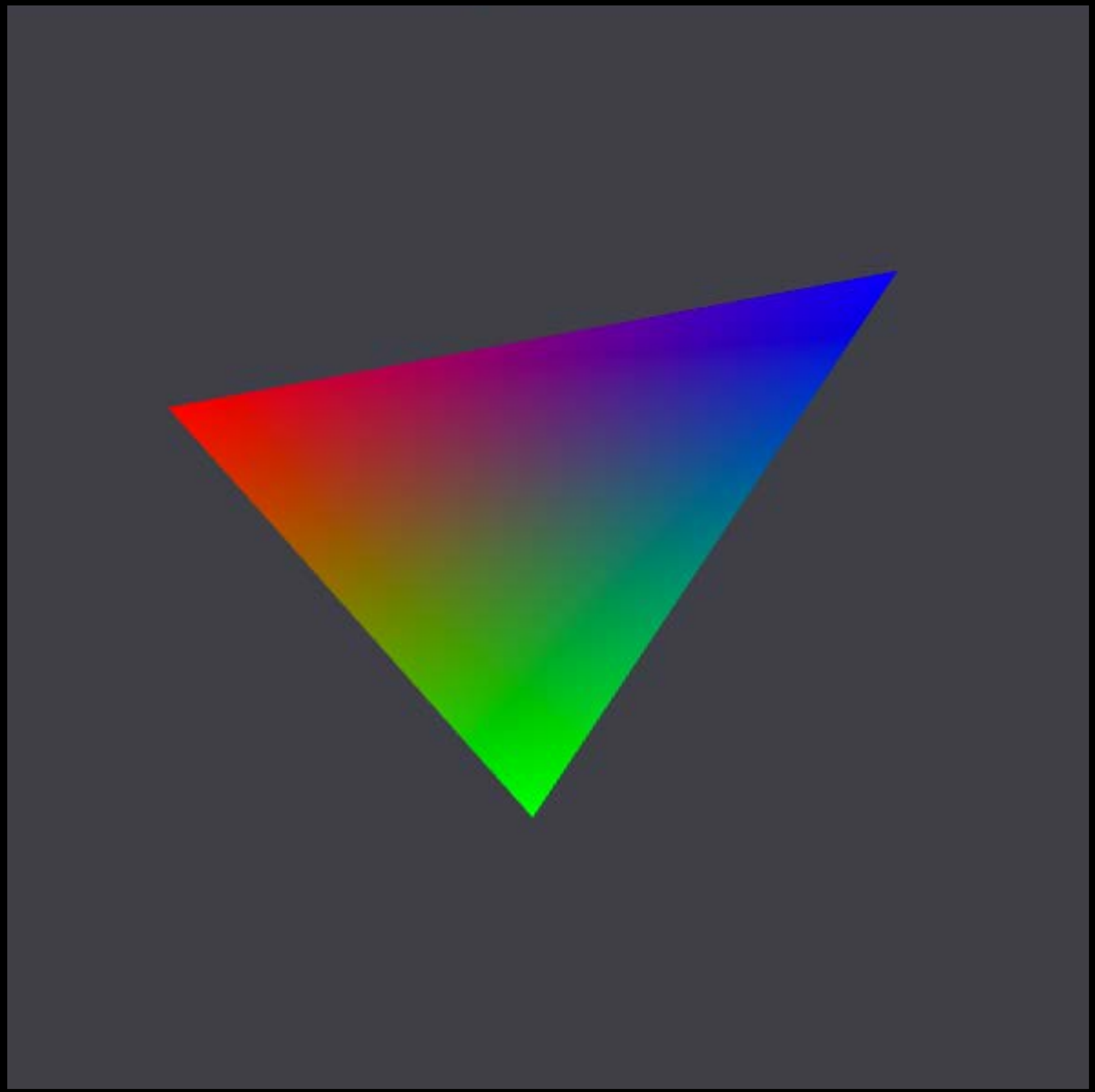
Creating a buffer of particular size

```
let buffer = device.newBuffer(withLength: strideof(myData), options:[])
```

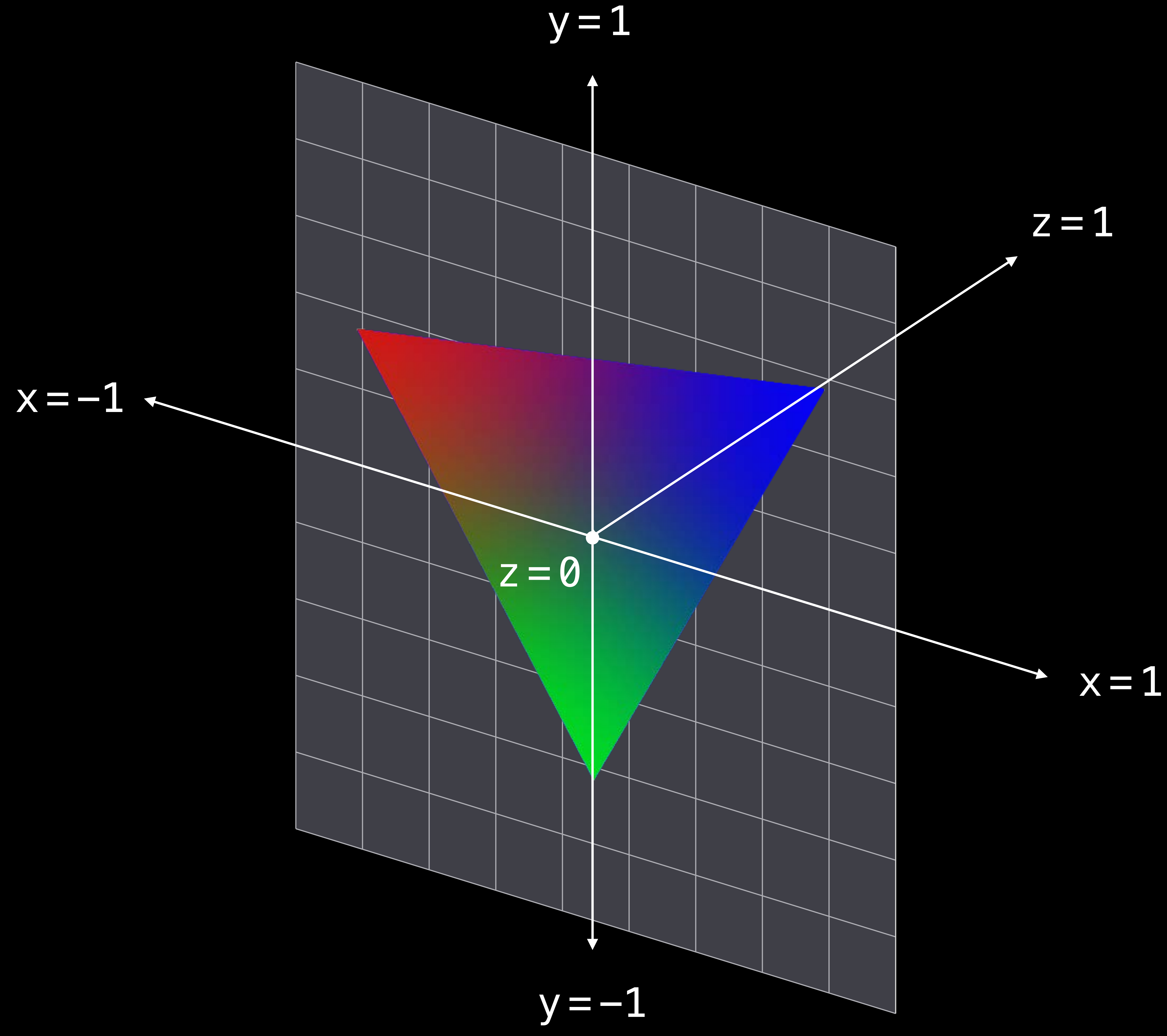
Creating a buffer containing a copy of existing memory

```
let buffer = device.newBuffer(withBytes: &myData, length: strideof(myData), options: [])
```

Defining Geometry for the Demo



Clip Space Coordinates



```swift
var vertices = [Vertex]()
vertices.append(Vertex(position: float4(-0.5, 0.25, 0, 1), color: float4(1, 0, 0, 1)))
vertices.append(Vertex(position: float4(   0, -0.5, 0, 1), color: float4(0, 1, 0, 1)))
vertices.append(Vertex(position: float4( 0.5,  0.5, 0, 1), color: float4(0, 0, 1, 1)))

var indices = [UInt16]()
indices.append(0)
indices.append(1)
indices.append(2)

let vertexBuffer = device.newBuffer(withBytes: vertices,
                                    length: strideof(Vertex) * vertices.count,
                                    options: [])

let indexBuffer = device.newBuffer(withBytes: indices,
                                   length: strideof(UInt16) * indices.count,
                                   options: [])
```

```swift
var vertices = [Vertex]()
vertices.append(Vertex(position: float4(-0.5, 0.25, 0, 1), color: float4(1, 0, 0, 1)))
vertices.append(Vertex(position: float4(   0, -0.5, 0, 1), color: float4(0, 1, 0, 1)))
vertices.append(Vertex(position: float4( 0.5,  0.5, 0, 1), color: float4(0, 0, 1, 1)))

var indices = [UInt16]()
indices.append(0)
indices.append(1)
indices.append(2)

let vertexBuffer = device.newBuffer(withBytes: vertices,
                                    length: strideof(Vertex) * vertices.count,
                                    options: [])

let indexBuffer = device.newBuffer(withBytes: indices,
                                   length: strideof(UInt16) * indices.count,
                                   options: [])
```

```swift
var vertices = [Vertex]()
vertices.append(Vertex(position: float4(-0.5, 0.25, 0, 1), color: float4(1, 0, 0, 1)))
vertices.append(Vertex(position: float4(   0, -0.5, 0, 1), color: float4(0, 1, 0, 1)))
vertices.append(Vertex(position: float4( 0.5,  0.5, 0, 1), color: float4(0, 0, 1, 1)))

var indices = [UInt16]()
indices.append(0)
indices.append(1)
indices.append(2)

let vertexBuffer = device.newBuffer(withBytes: vertices,
                                    length: strideof(Vertex) * vertices.count,
                                    options: [])

let indexBuffer = device.newBuffer(withBytes: indices,
                                   length: strideof(UInt16) * indices.count,
                                   options: [])
```

```swift
var vertices = [Vertex]()
vertices.append(Vertex(position: float4(-0.5, 0.25, 0, 1), color: float4(1, 0, 0, 1)))
vertices.append(Vertex(position: float4(   0, -0.5, 0, 1), color: float4(0, 1, 0, 1)))
vertices.append(Vertex(position: float4( 0.5,  0.5, 0, 1), color: float4(0, 0, 1, 1)))

var indices = [UInt16]()
indices.append(0)
indices.append(1)
indices.append(2)
```

```swift
let vertexBuffer = device.newBuffer(withBytes: vertices,
                                    length: strideof(Vertex) * vertices.count,
                                    options: [])

let indexBuffer = device.newBuffer(withBytes: indices,
                                   length: strideof(UInt16) * indices.count,
                                   options: [])
```

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

Multithreaded Encoding

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

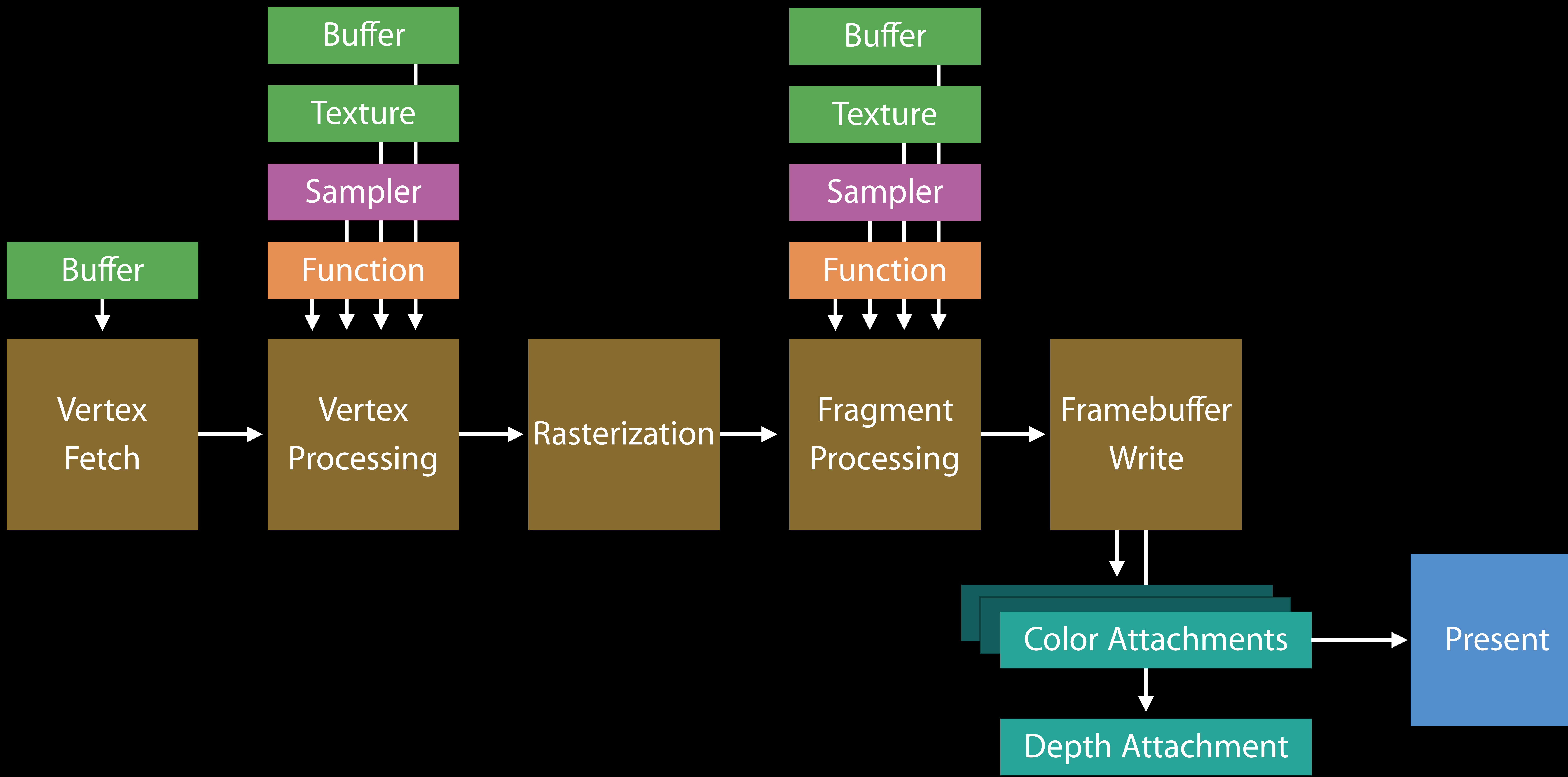
Multithreaded Encoding

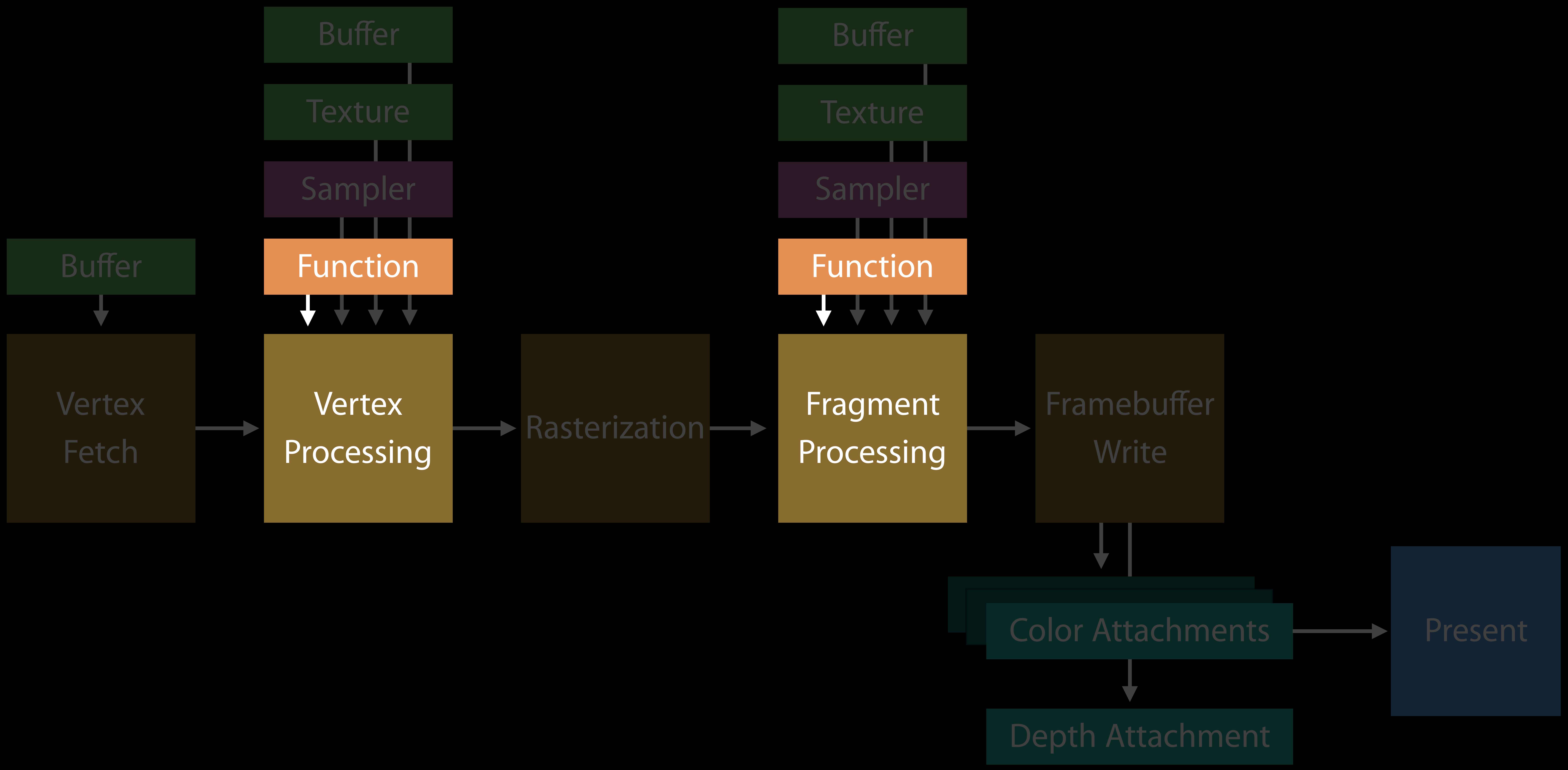
Metal Shading Language

Extended subset of C++14

Unified language for graphics and compute

Lets you write programs for the GPU

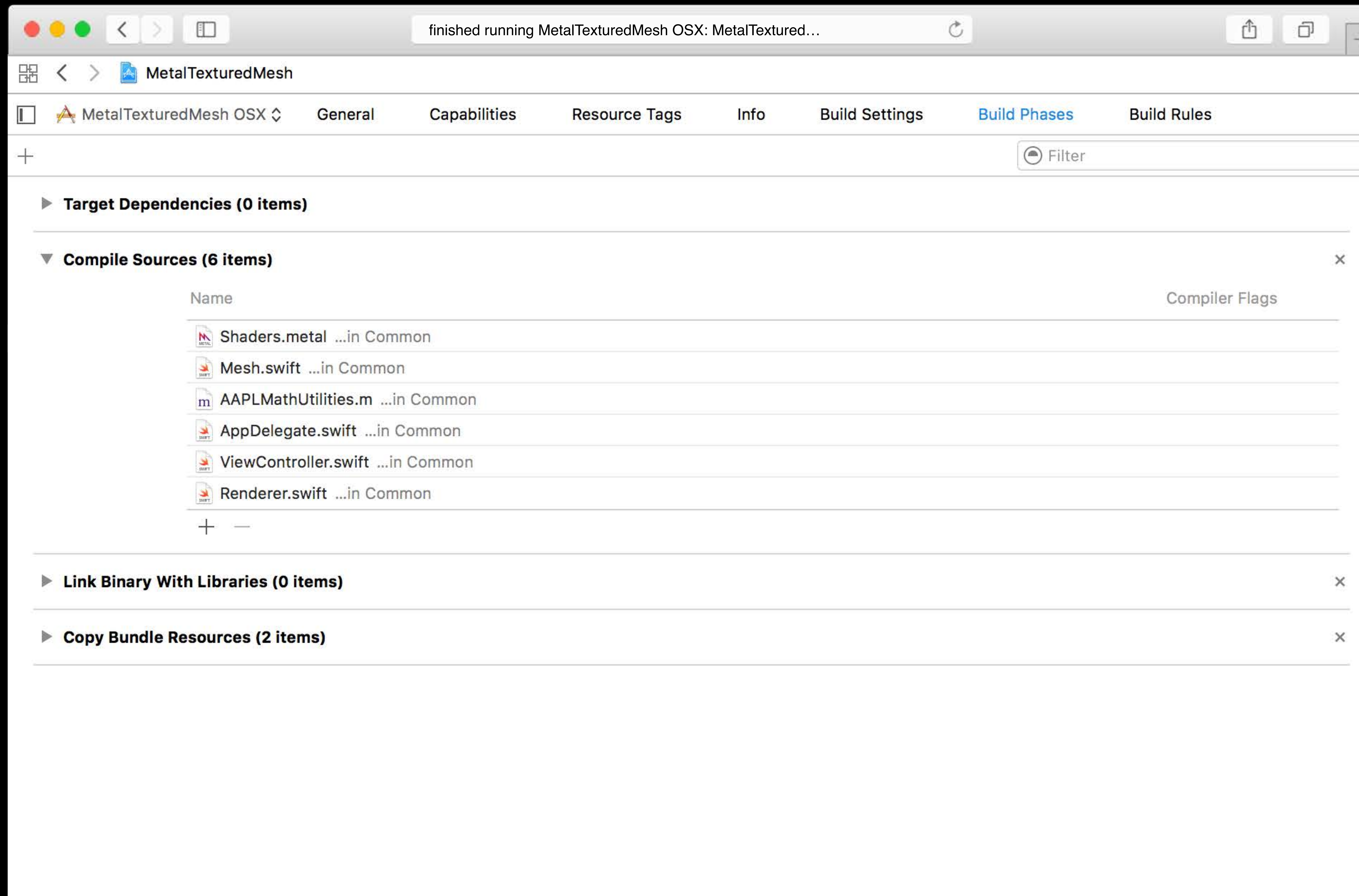

Buffer
Texture
Sampler
Function
Buffer
Texture
Sampler
Function
Buffer
Vertex Fetch
Vertex Processing
Rasterization
Fragment Processing
Framebuffer Write
Color Attachments
Depth Attachment
Present

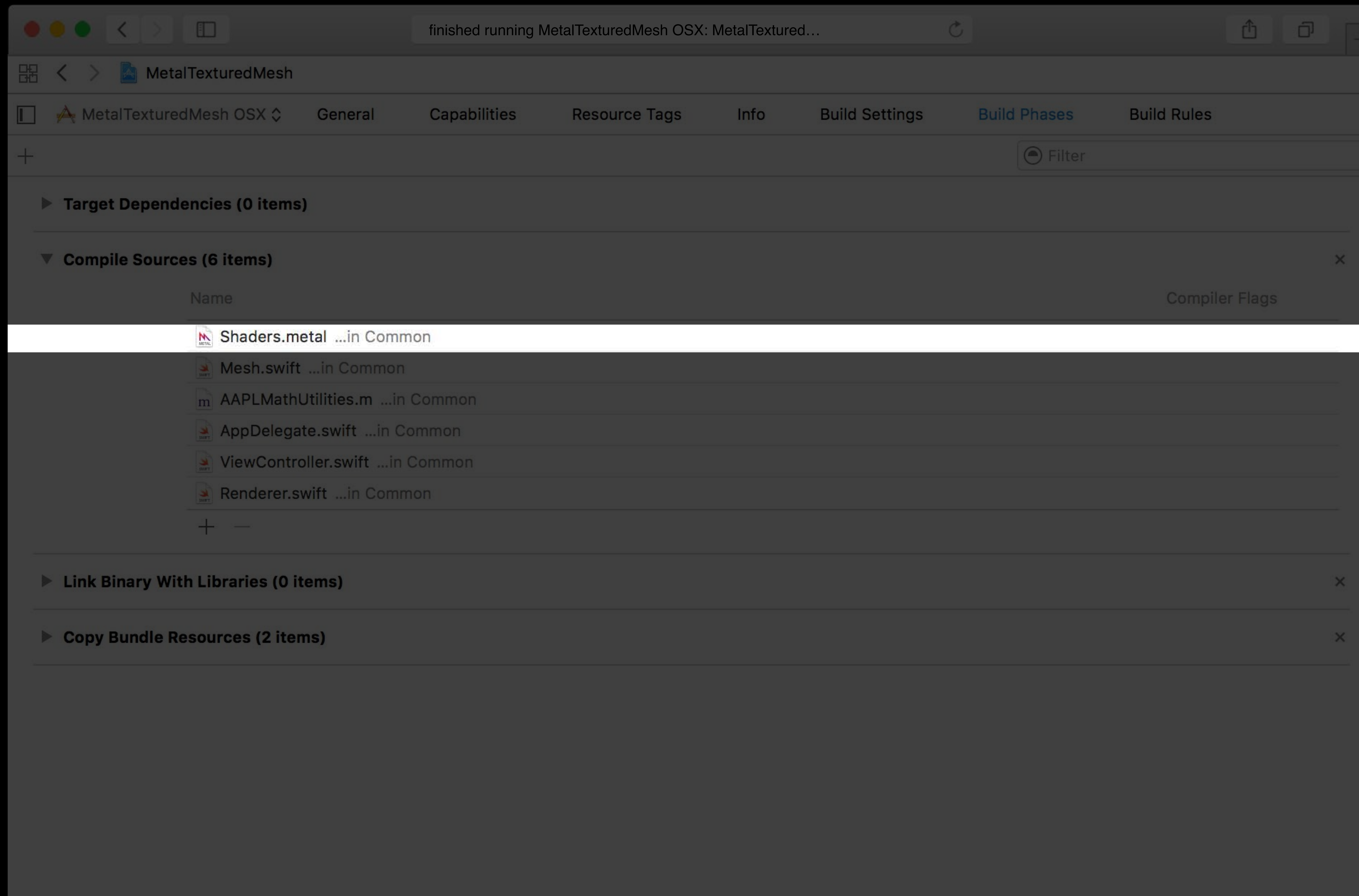

Buffer
Texture
Sampler
Function
Buffer
Texture
Sampler
Function
Buffer
Vertex Fetch
Vertex Processing
Rasterization
Fragment Processing
Framebuffer Write
Color Attachments
Depth Attachment
Present

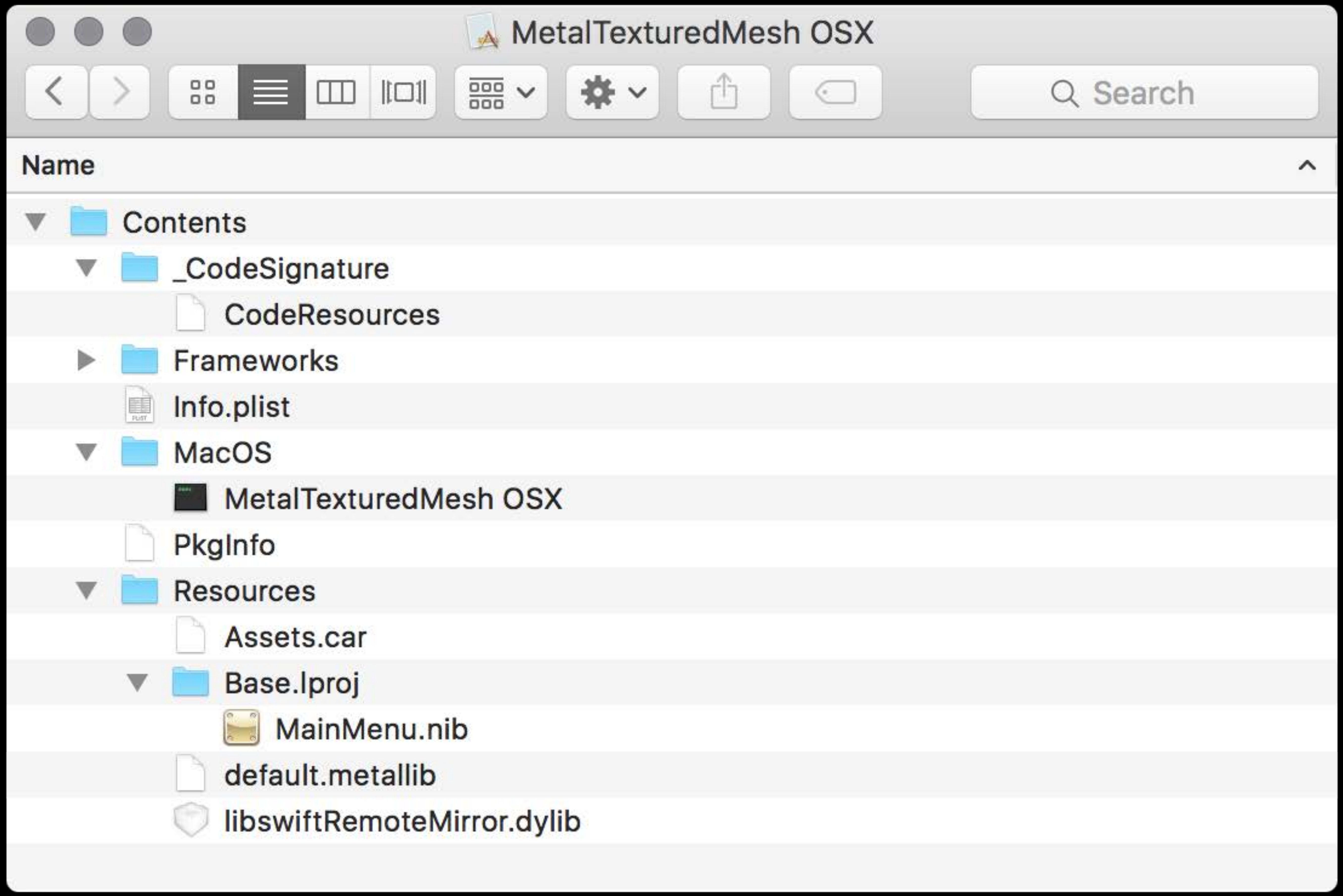
```
vertex VertexOut vertex_transform(...)
{
   VertexOut out;
   ...
   return out;
}
fragment half4 fragment_lighting(VertexOut fragmentIn [[stage_in]])
{
  ...
   return color;
}
```

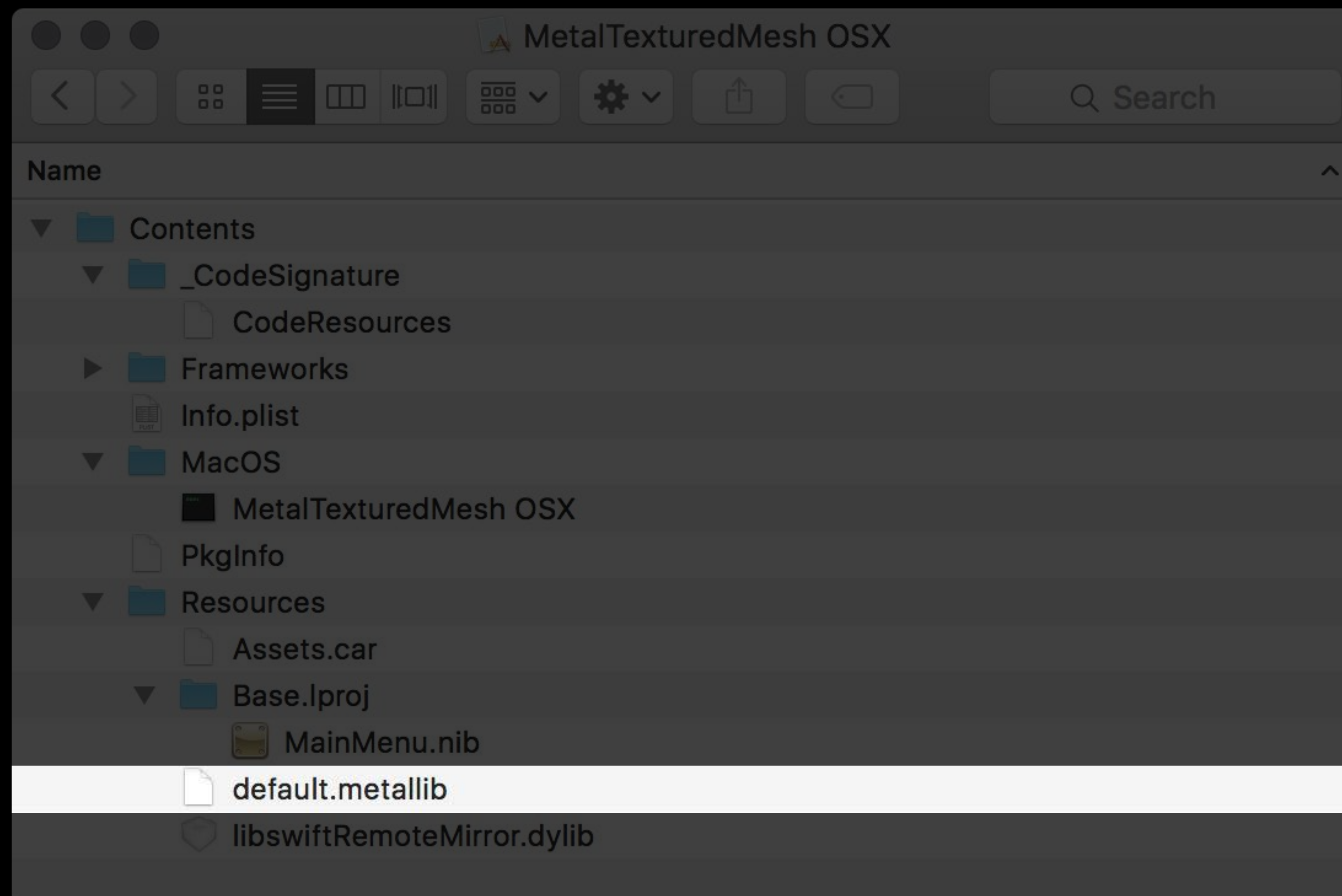
```
vertex VertexOut vertex_transform(...)
{
   VertexOut out;
   ...
   return out;
}
fragment half4 fragment_lighting(VertexOut fragmentIn [[stage_in]])
{
  ...
   return color;
}
```

```
vertex VertexOut vertex_transform(...)
{
   VertexOut out;
   ...
   return out;
}
fragment half4 fragment_lighting(VertexOut fragmentIn [[stage_in]])
{
  ...
   return color;
}
```


finished running MetalTexturedMesh OSX: MetalTextured…
MetalTexturedMesh
MetalTexturedMesh OSX
General
Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
Filter
Target Dependencies (0 items)
Compile Sources (6 items)
Name
Compiler Flags
Shaders.metal ...in Common
Mesh.swift ...in Common
AAPLMathUtilities.m ...in Common
AppDelegate.swift ...in Common
ViewController.swift ...in Common
Renderer.swift ...in Common
Link Binary With Libraries (0 items)
Copy Bundle Resources (2 items)


finished running MetalTexturedMesh OSX: MetalTextured…
MetalTexturedMesh
MetalTexturedMesh OSX
General
Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
Filter
Target Dependencies (0 items)
Compile Sources (6 items)
Name
Compiler Flags
Shaders.metal ...in Common
Mesh.swift ...in Common
AAPLMathUtilities.m ...in Common
AppDelegate.swift ...in Common
ViewController.swift ...in Common
Renderer.swift ...in Common
Link Binary With Libraries (0 items)
Copy Bundle Resources (2 items)


MetalTexturedMesh OSX
Search
Name
Contents
_CodeSignature
CodeResources
Frameworks
Info.plist
MacOS
MetalTexturedMesh OSX
PkgInfo
Resources
Assets.car
Base.lproj
MainMenu.nib
default.metallib
libswiftRemoteMirror.dylib


MetalTexturedMesh OSX
Search
Name
Contents
_CodeSignature
CodeResources
Frameworks
Info.plist
MacOS
MetalTexturedMesh OSX
PkgInfo
Resources
Assets.car
Base.lproj
MainMenu.nib
default.metallib
libswiftRemoteMirror.dylib

Build-Time Compilation

Shading language files `(.metal)`

- Compiled by Xcode using Metal toolchain
- Produce `.metallibs` (collection of compiled functions)
- `default.metallib` copied automatically into app bundle

MTLLibrary

A collection of compiled function objects

Multiple ways to create

- Loaded from `default.metallib`
- Loaded from `.metallib` built with command-line toolchain
- Built from source at run time

```swift
// MTLLibrary
// API


let library = device.newDefaultLibrary()
```

MTLFunction

Metal object representing a single function

Associated with a particular pipeline stage

- Vertex `(vertex)`
- Fragment `(fragment)`
- Compute `(kernel)`

```
vertex VertexOut vertex_transform(...)
{
   VertexOut out;
   ...
   return out;
}
fragment half4 fragment_lighting(VertexOut fragmentIn [[stage_in]])
{
  ...
   return color;
}
```

```
vertex VertexOut vertex_transform(...)
{
   VertexOut out;
   ...
   return out;
}
fragment half4 fragment_lighting(VertexOut fragmentIn [[stage_in]])
{
  ...
   return color;
}
```

```
vertex VertexOut vertex_transform(...)
{
   VertexOut out;
   ...
   return out;
}
fragment half4 fragment_lighting(VertexOut fragmentIn [[stage_in]])
{
  ...
   return color;
}
```

```swift
// MTLFunction
// API


let vertexFunction = library.newFunction(withName: "vertex_transform")
let fragmentFunction = library.newFunction(withName: "fragment_lighting")
```

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

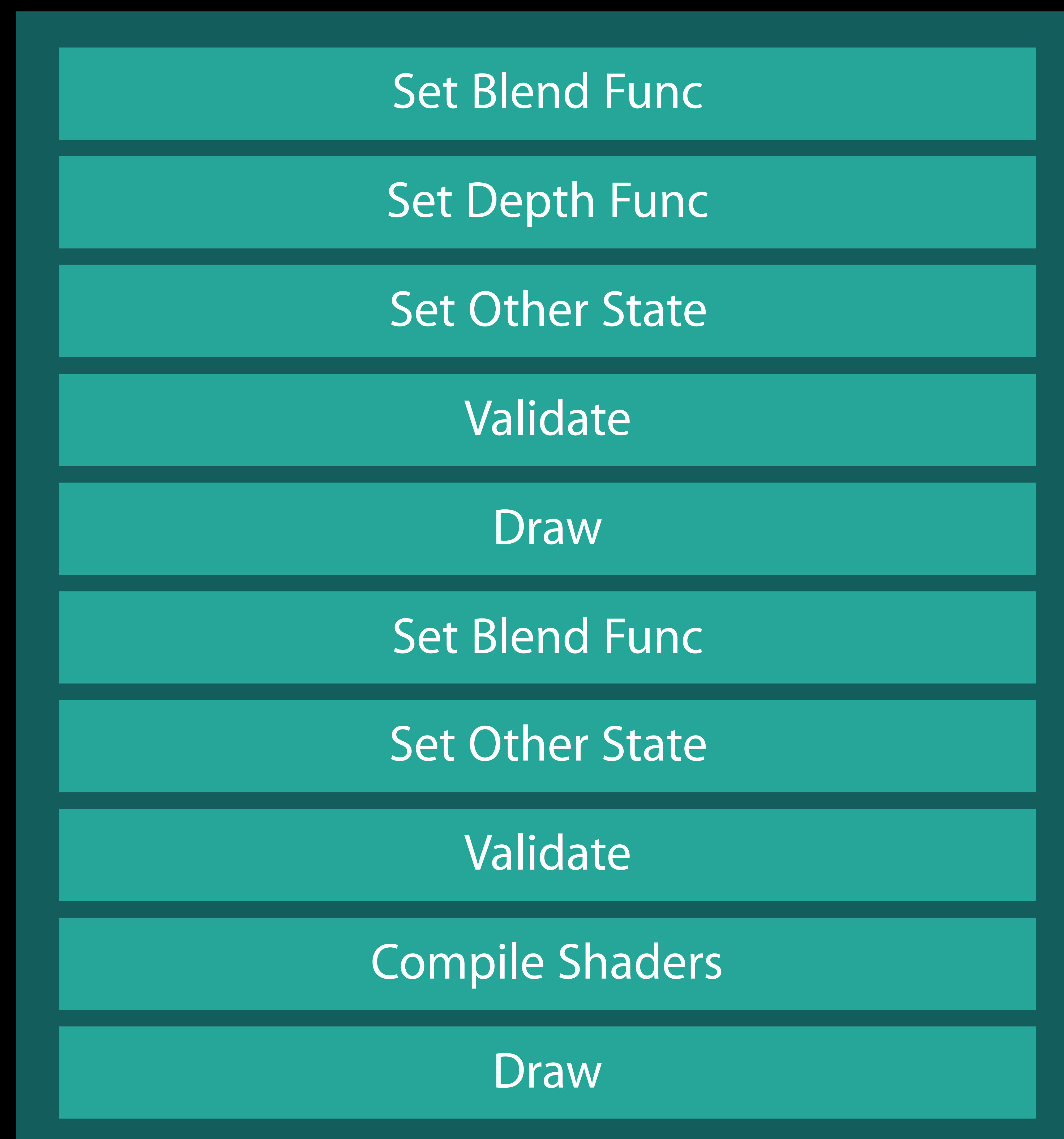
The Virtue of Precompiled State

OpenGL

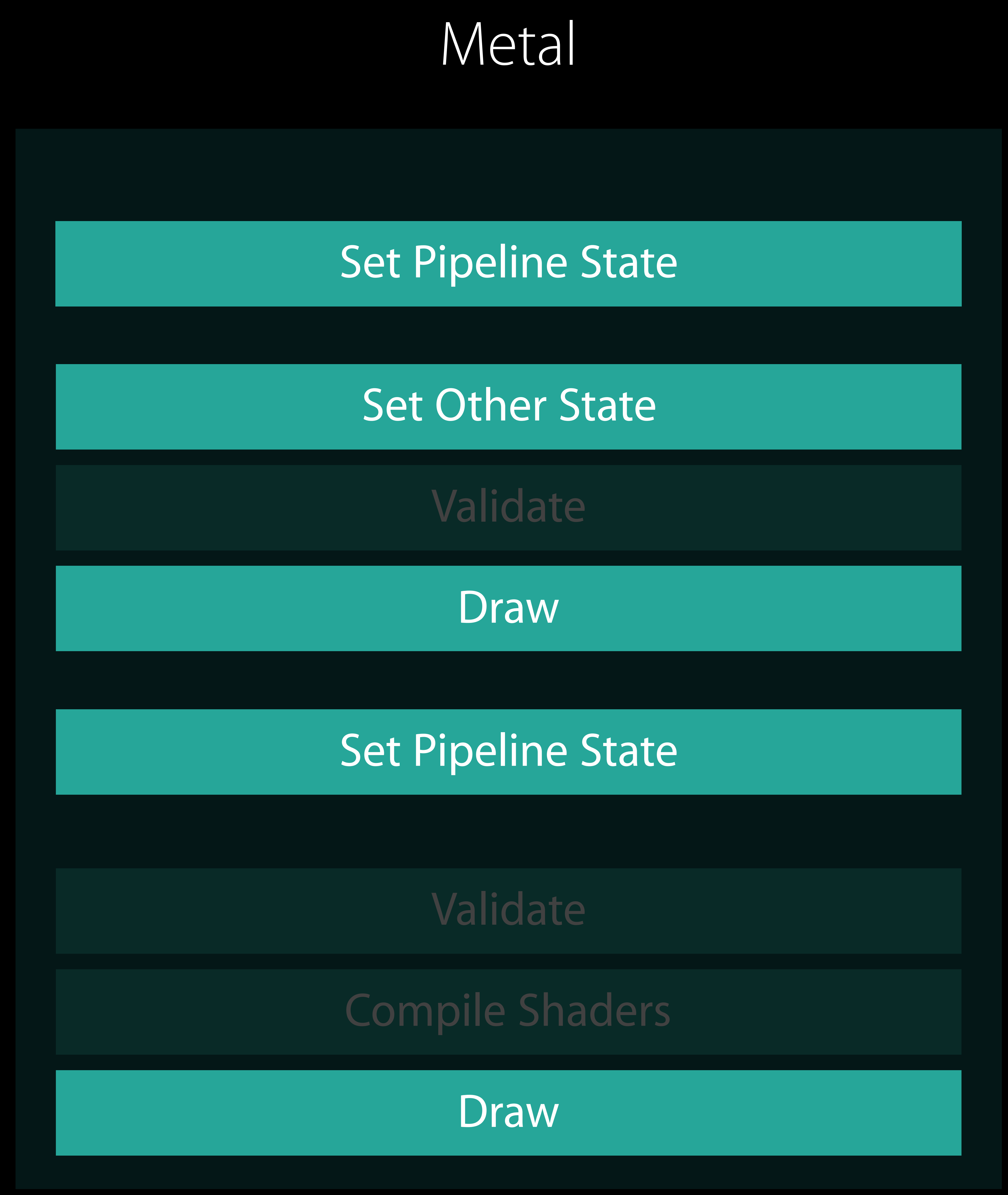
- Set Blend Func
- Set Depth Func
- Set Other State
- Validate
- Draw
- Set Blend Func
- Set Other State
- Validate
- Compile Shaders
- Draw

The Virtue of Precompiled State

OpenGL



The Virtue of Precompiled State



Precompiled State vs. "Any Time" State

Set on Pipeline State	Set While Drawing
Vertex and Fragment Function	Front Face Winding
Alpha Blending	Cull Mode
MSAA Sample Count	Fill Mode
Pixel Formats for Render Targets	Scissor
Vertex Descriptor	Viewport

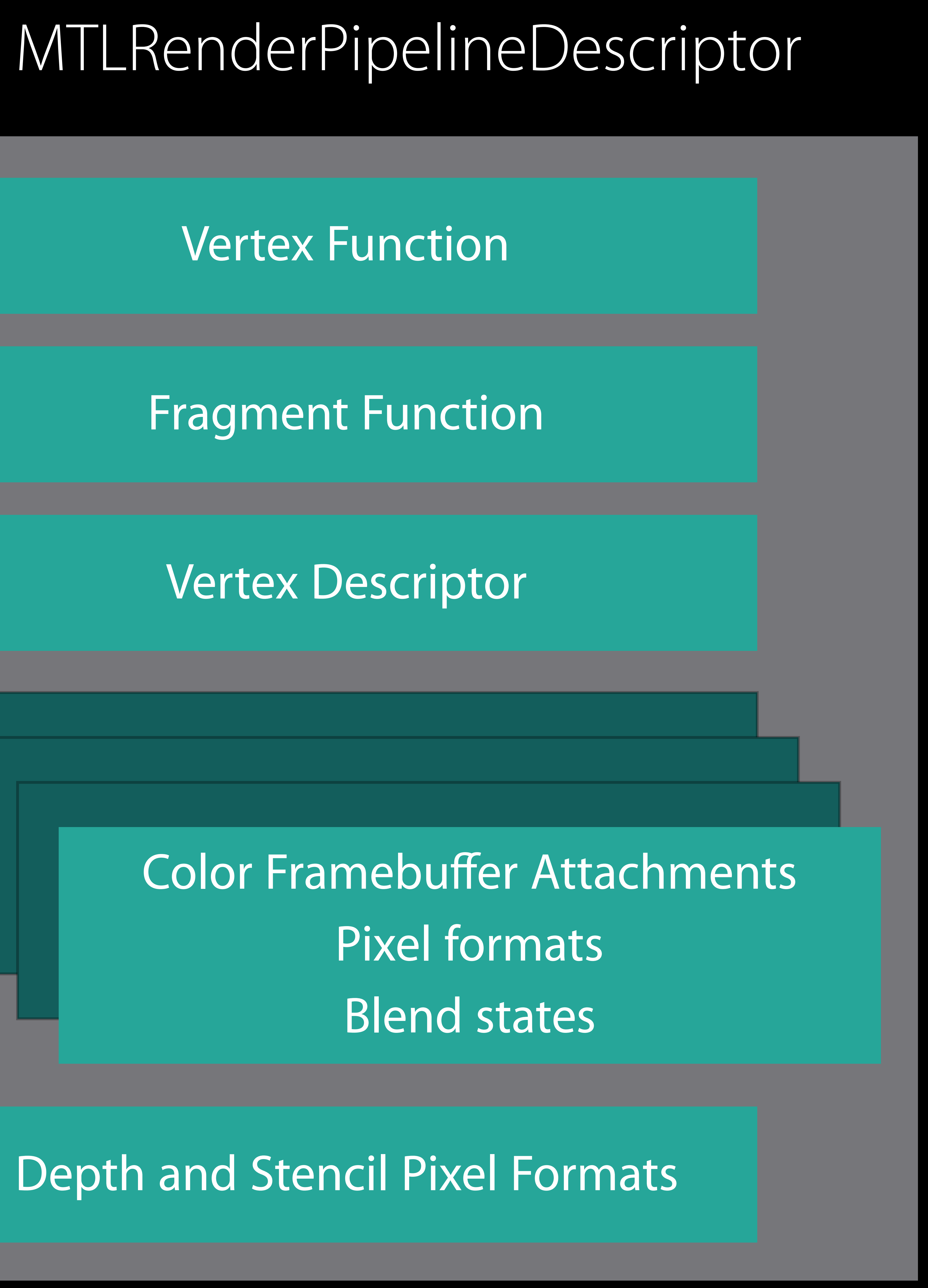
MTLRenderPipelineState

Represents a “configuration” of the GPU pipeline

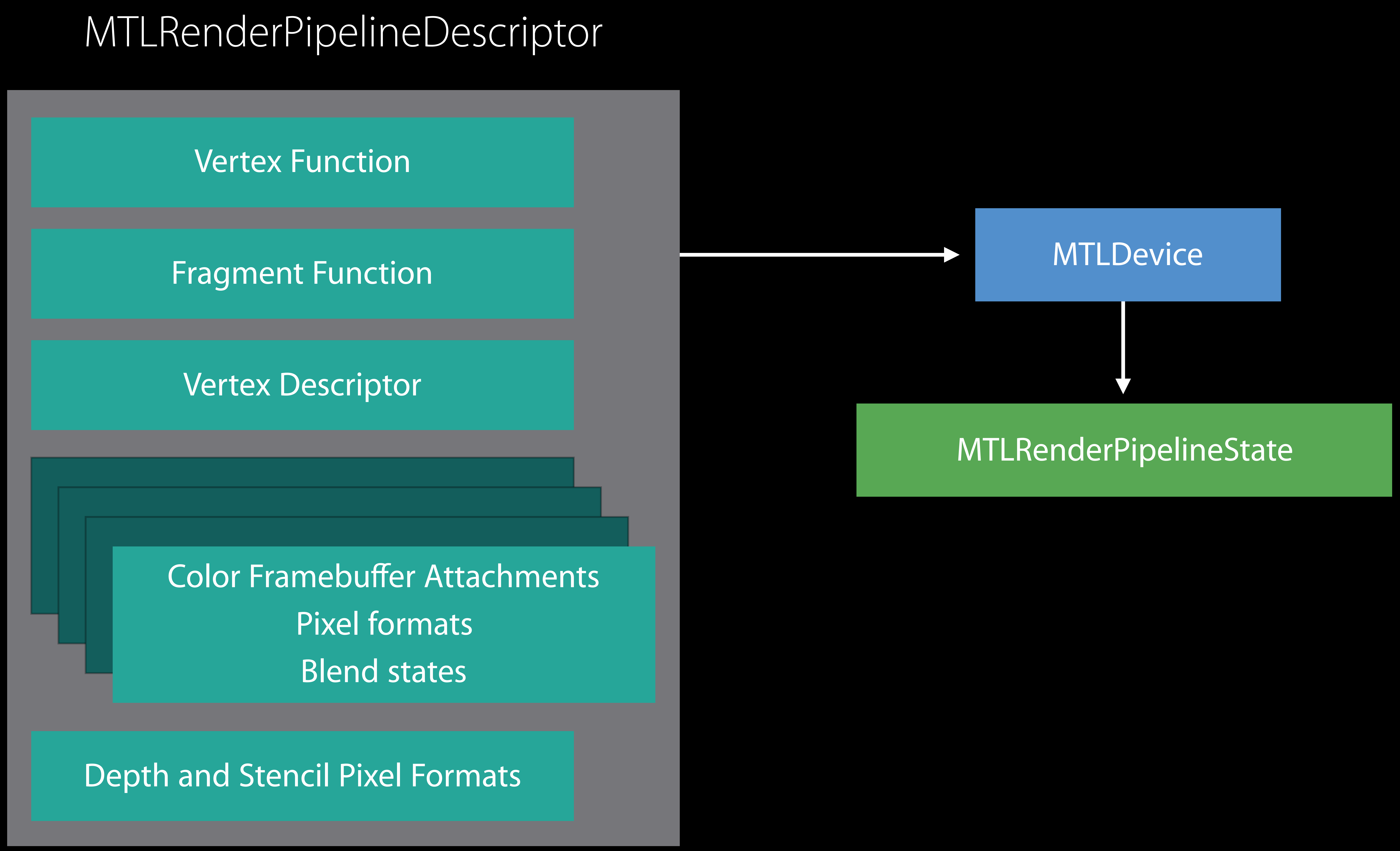
Contains validated set of state used during rendering

Usually created at application load time

Render Pipeline Descriptor



Render Pipeline Descriptor



```swift
// MTLRenderPipelineState
// API


let pipelineDescriptor = MTLRenderPipelineDescriptor()
pipelineDescriptor.vertexFunction = vertexFunction
pipelineDescriptor.fragmentFunction = fragmentFunction
pipelineDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm

let pipelineState = try device.newRenderPipelineState(with: pipelineDescriptor)
```

```swift
// MTLRenderPipelineState
// API

let pipelineDescriptor = MTLRenderPipelineDescriptor()
pipelineDescriptor.vertexFunction = vertexFunction
pipelineDescriptor.fragmentFunction = fragmentFunction
pipelineDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm

let pipelineState = try device.newRenderPipelineState(with: pipelineDescriptor)
```

```swift
// MTLRenderPipelineState
// API



let pipelineDescriptor = MTLRenderPipelineDescriptor()
pipelineDescriptor.vertexFunction = vertexFunction
pipelineDescriptor.fragmentFunction = fragmentFunction
pipelineDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm

let pipelineState = try device.newRenderPipelineState(with: pipelineDescriptor)
```

Pipeline States are Persistent Objects

Create them during load time

Keep them around, as you do your device and resources

Switch among them when drawing

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Issuing GPU Commands

Interfacing with UIKit and AppKit

Metal Command Submission Model

Render Passes

Draw Calls

Getting on the Screen

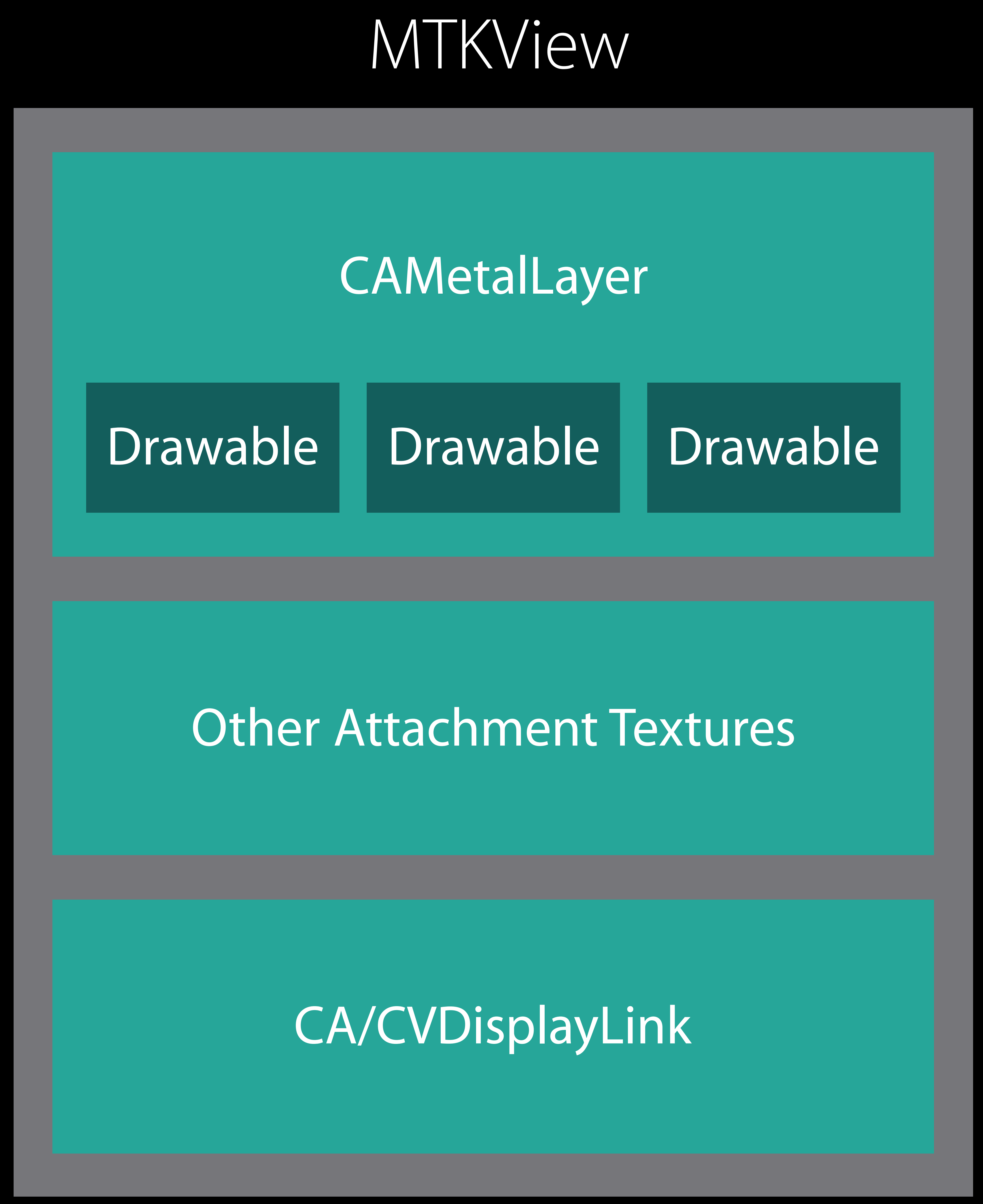
MTKView

Cross-platform view class

- NSView on OS X
- UIView on iOS & tvOS

Reduces boilerplate code

- Creates and manages a CALayer
- Issues periodic drawing callbacks
- Manages render targets

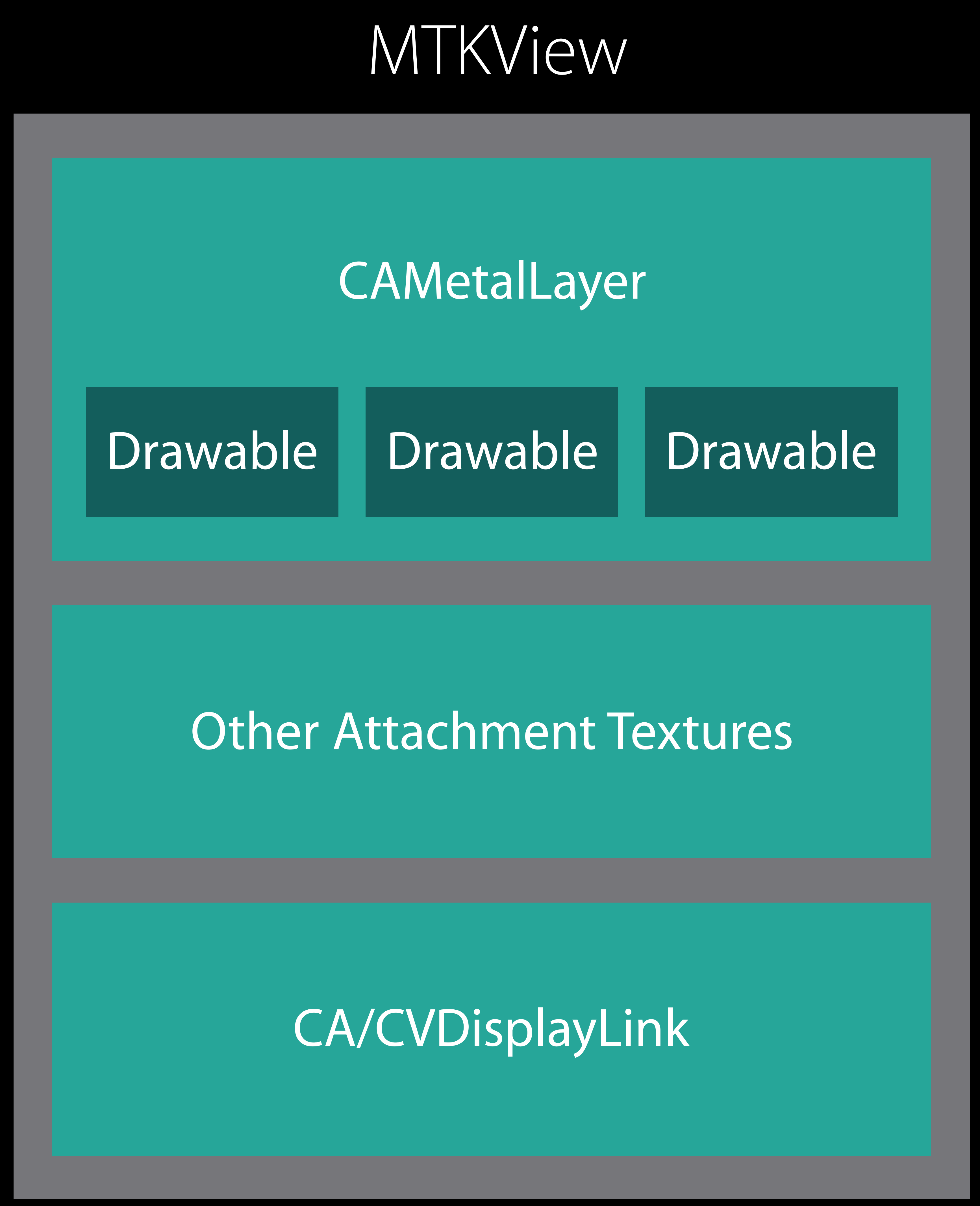


Drawables

Wrap a texture to be displayed on screen

Managed by CAMetalLayer

- Kept in an internal queue
- Reused across frames

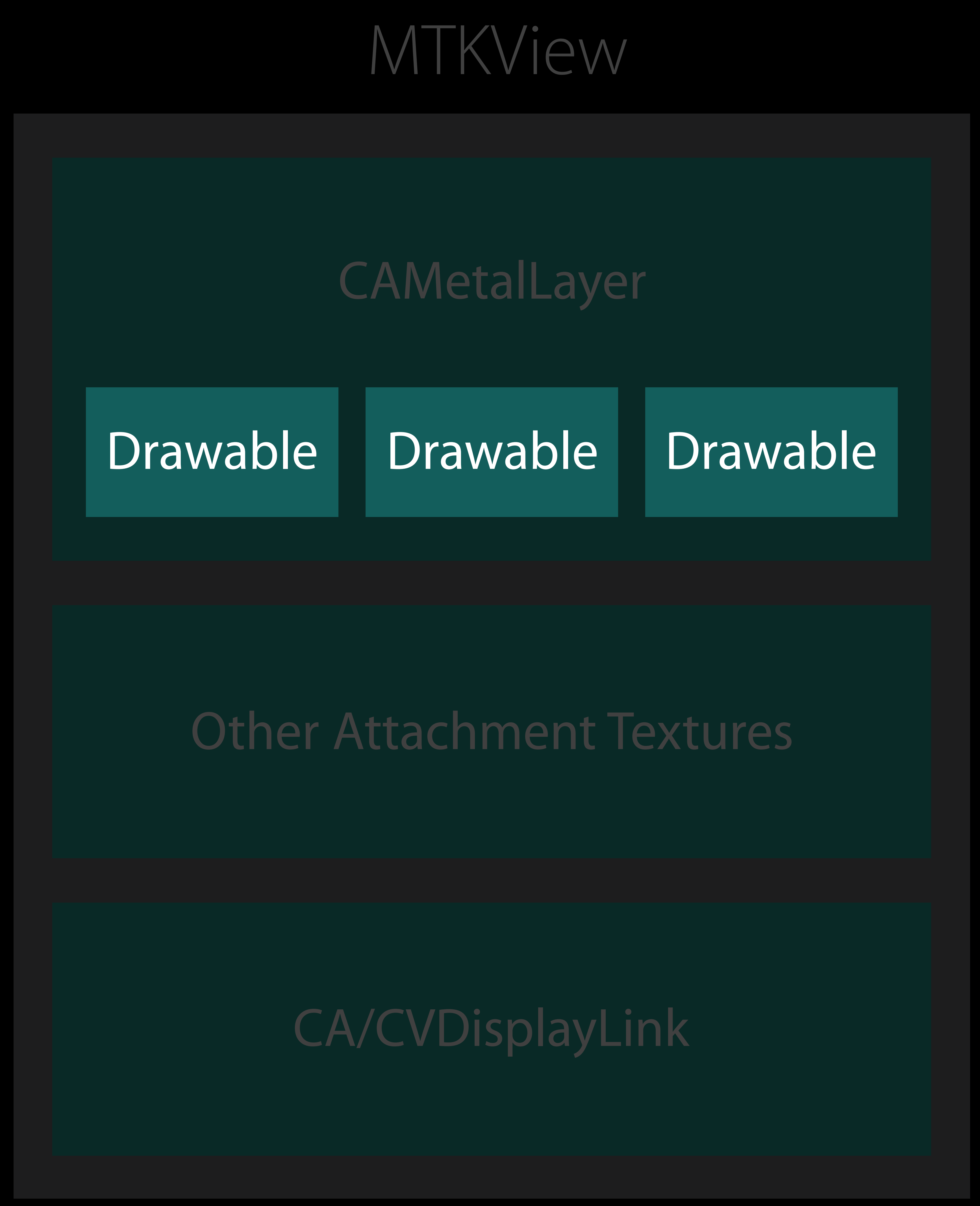


Drawables

Wrap a texture to be displayed on screen

Managed by CAMetalLayer

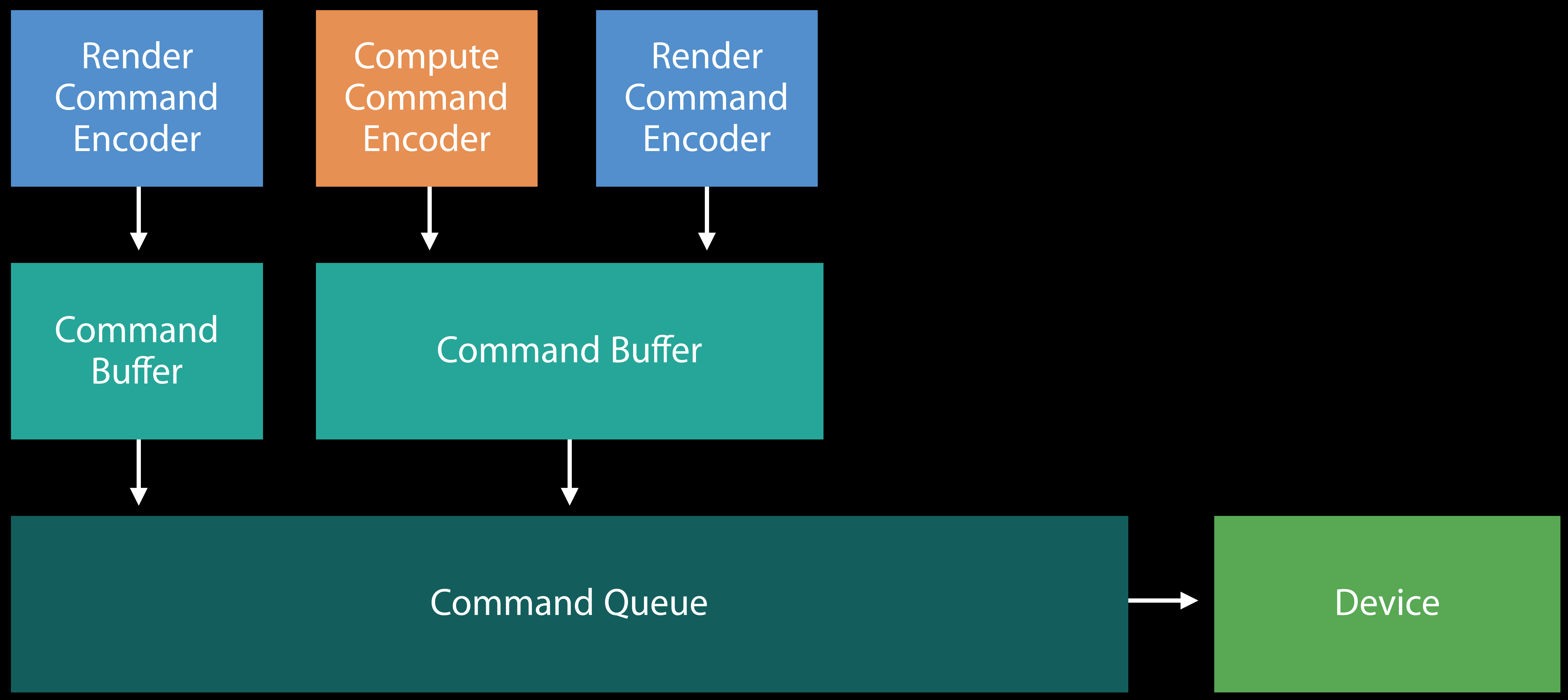
- Kept in an internal queue
- Reused across frames



```swift
// MTKView
// Configuration


// Clear to solid white
view.clearColor = MTLClearColorMake(1, 1, 1, 1)
// Use a BGRA 8-bit normalized texture for the drawable
view.colorPixelFormat = .bgra8Unorm
// Use a 32-bit depth buffer
view.depthStencilPixelFormat = .depth32Float
// Make the renderer object responsible for drawing on our behalf
view.delegate = renderer
```

```swift
// Implementing MTKViewDelegate



func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
    // Respond to resize
}
func draw(in metalView: MTKView)
{
    // Our command buffer is a container for the work we want to perform with the GPU.
    let commandBuffer = commandQueue.commandBuffer()

    // Encode render passes
    ...

    // Now that we're done issuing commands, we commit our buffer so the GPU can get to work.
    commandBuffer.commit()
}
```

Command Submission Model



Command Submission Model

Explicit command buffer construction and submission

- Each buffer is a parcel of work for the GPU
- Command buffer submission is under your control

Command encoders

- Translate from API calls to work for the GPU
- No deferred state validation

Command Submission Model

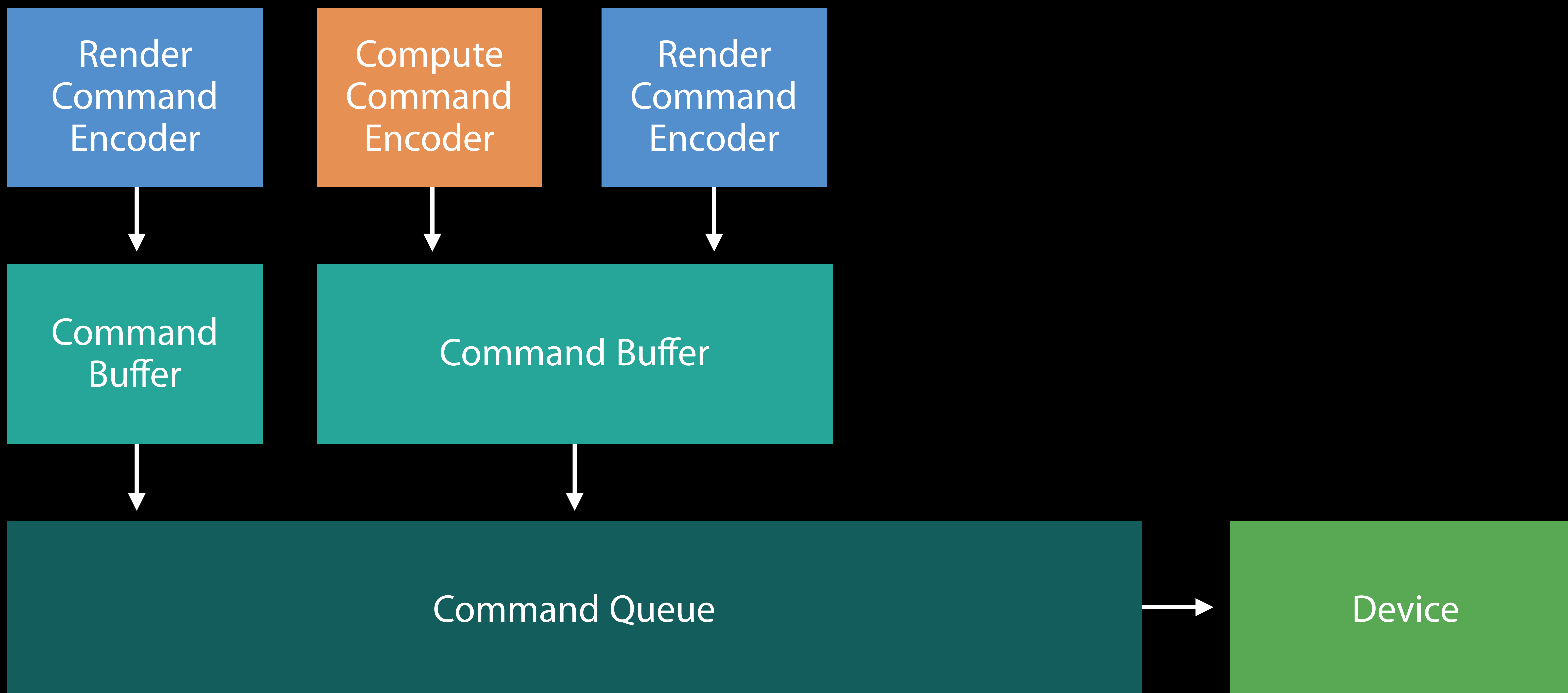
Multithreaded command encoding

- Multiple command buffers can be encoded in parallel
- App decides execution order

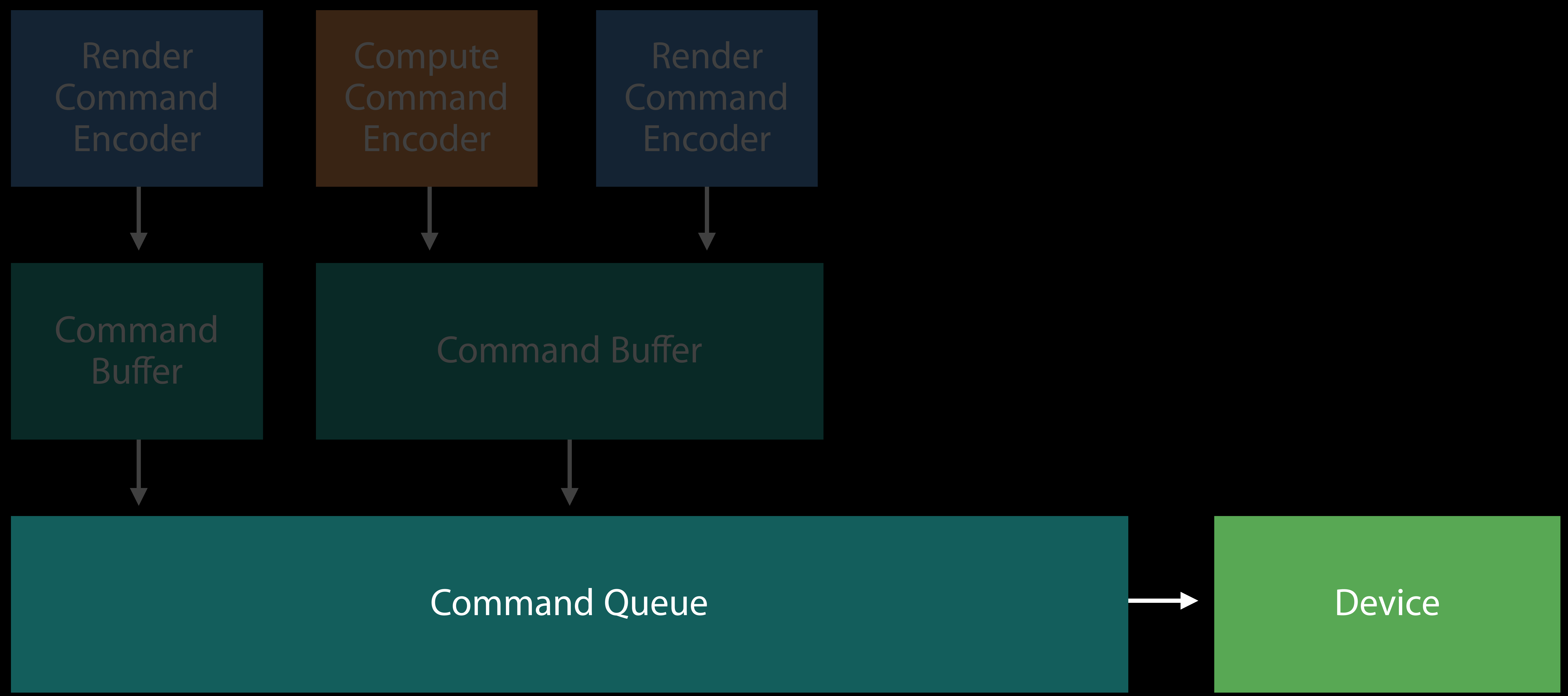
Scales to allow tens of thousands of draw calls per frame

More on this in Part II

Command Queues



Command Queues



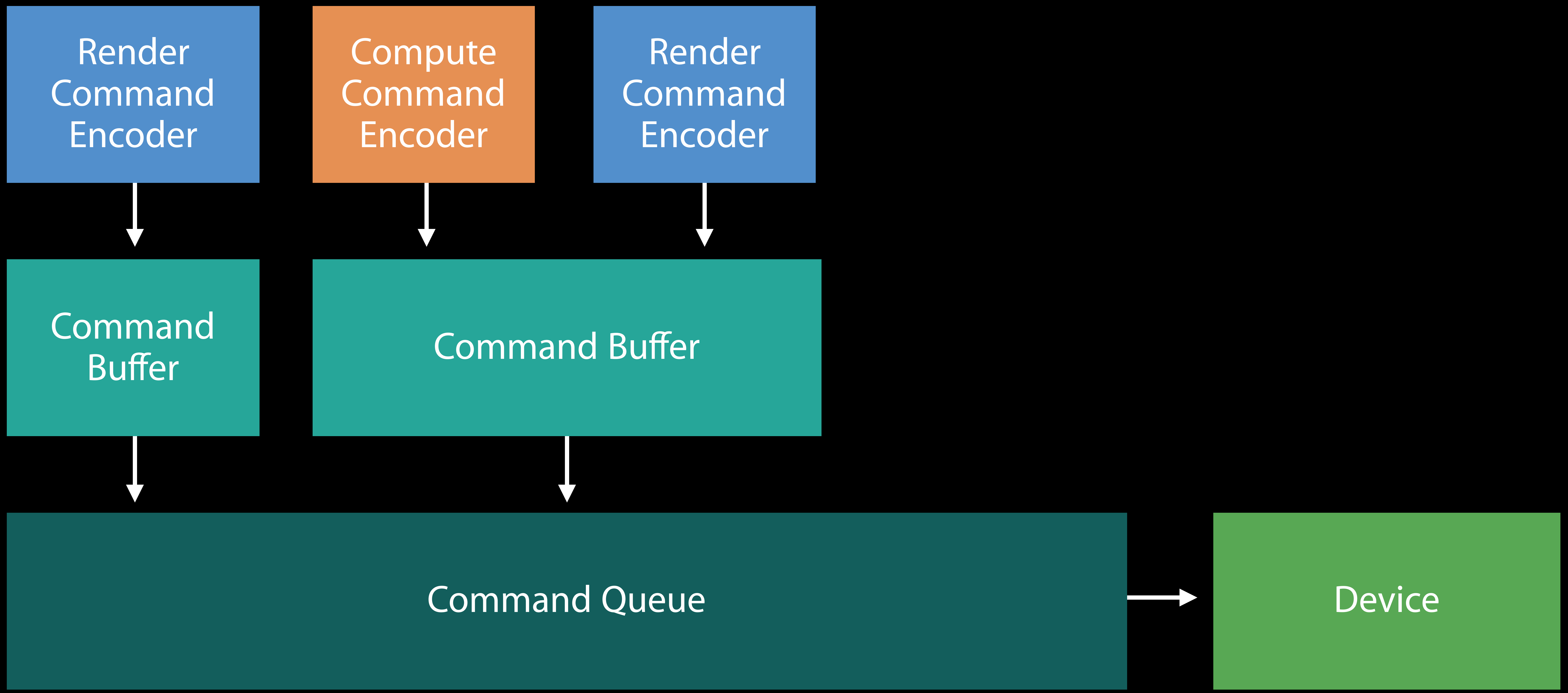
MTLCommandQueue

Manage the work that has been queued up for the device

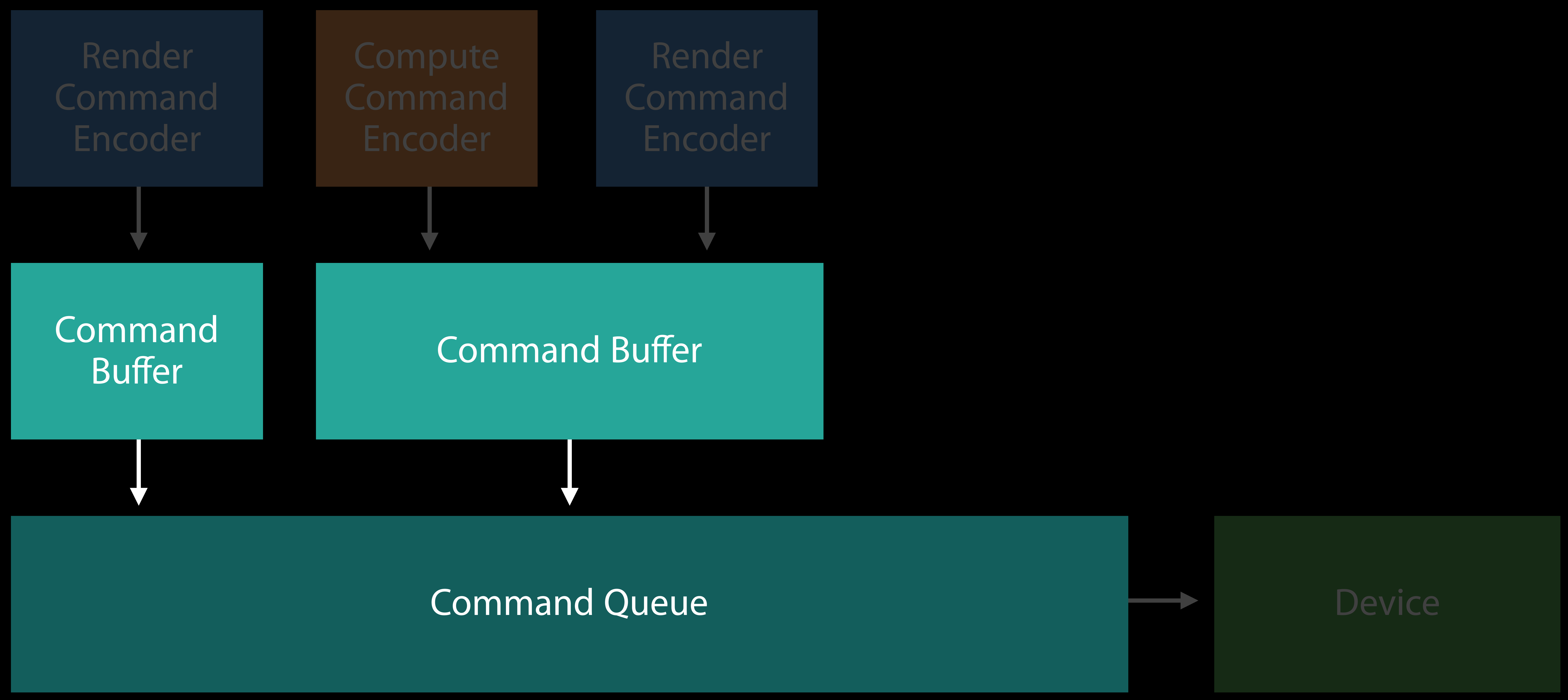
- Should be created up-front and live as long as your device
- You'll often need only one
- Thread-safe

```swift
// MTLCommandQueue
// API



// Create the command queue we will be using to submit work to the GPU.
let commandQueue = device.newCommandQueue()
```

Command Buffers



Command Buffers



MTLCommandBuffer

Contains sets of commands to be executed by the GPU

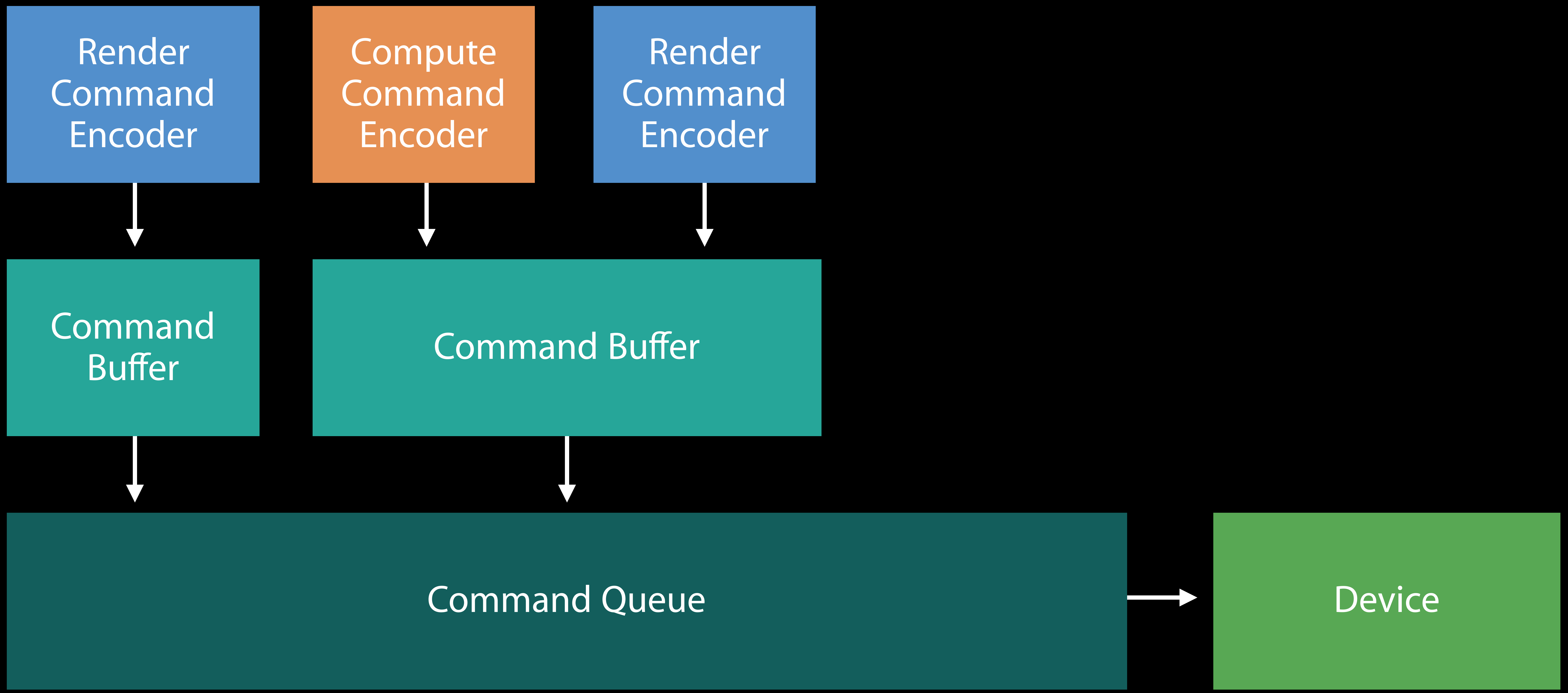
Enqueued on a command queue for scheduling

Transient objects

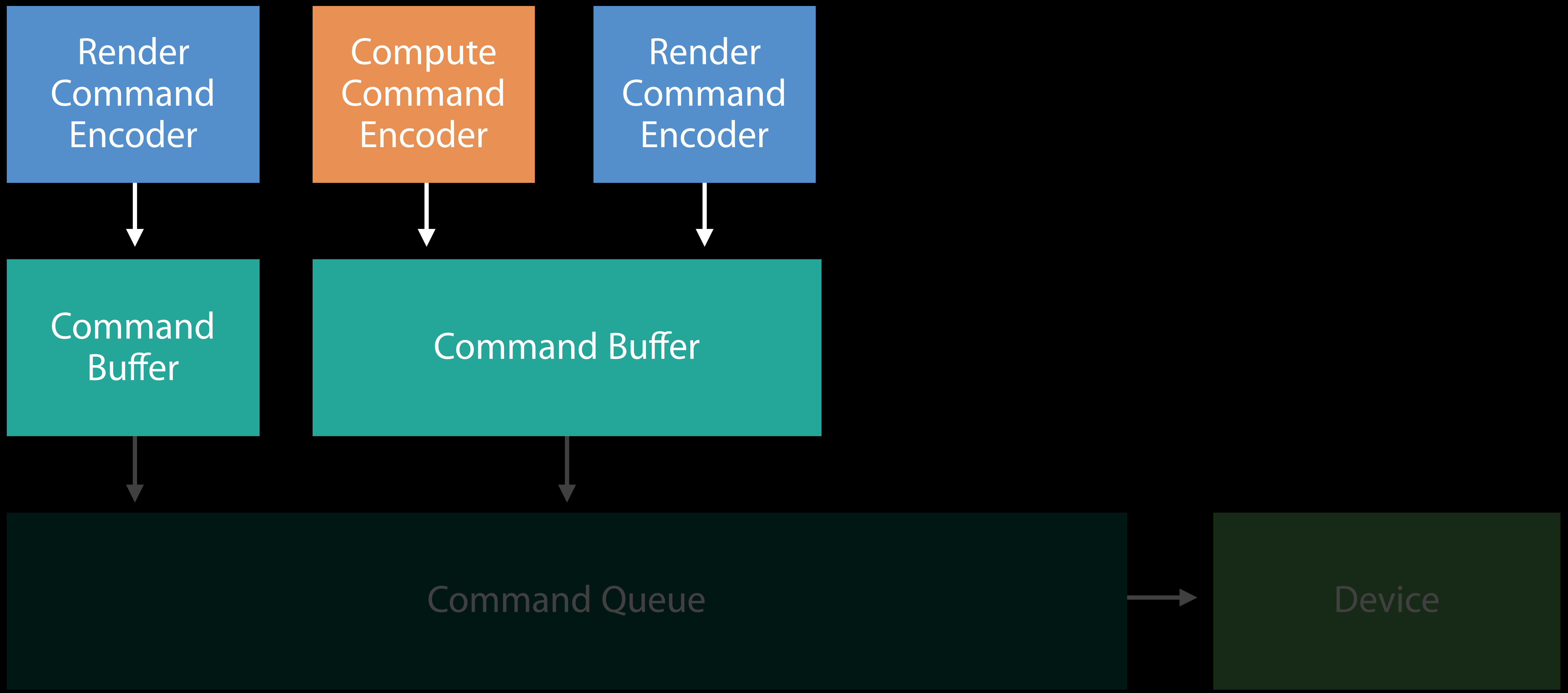
- Create one or more per frame
- Not reusable; fire-and-forget

```swift
// MTLCommandBuffer
// API



// Our command buffer is a container for the work we want to perform with the GPU.
let commandBuffer = commandQueue.commandBuffer()
```

Command Encoders



Command Encoders



Command Encoders

Translate API calls into work for the GPU

Different encoders for different types of work

- Render
- Compute
- Blit

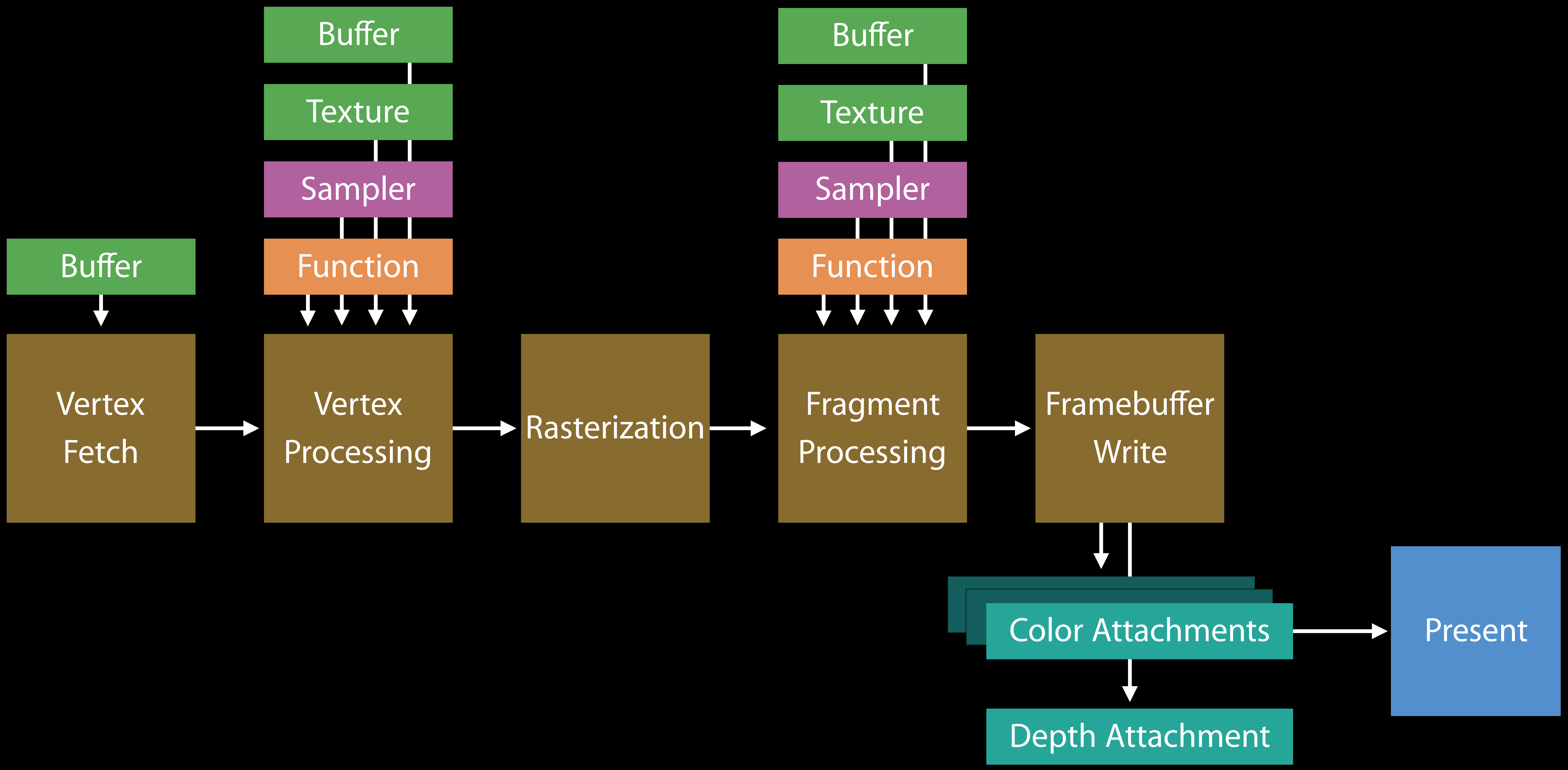
MTLRenderCommandEncoder

Encodes the work of a single pass into a command buffer

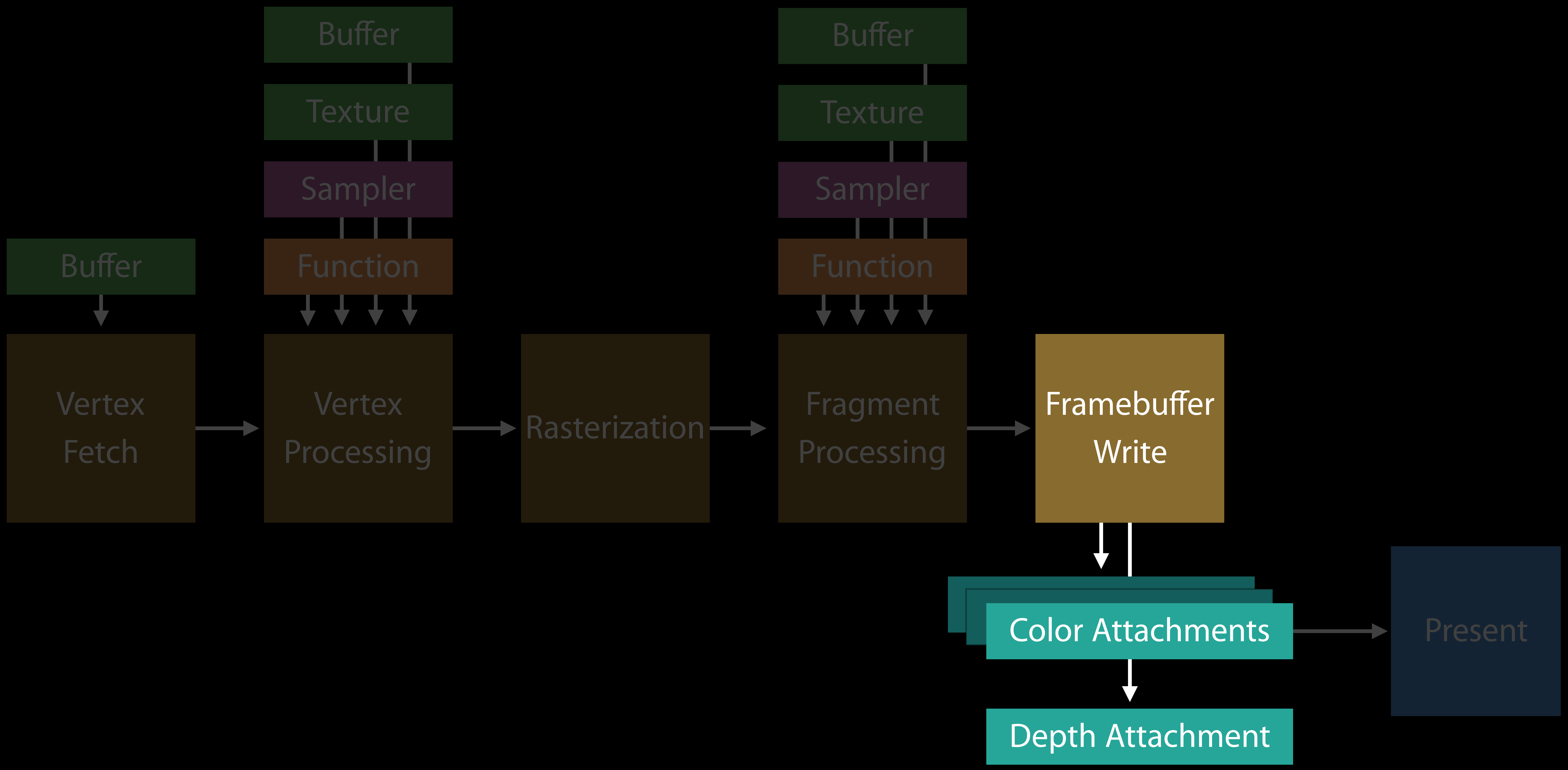
- State changes
- Draw calls

Has a set of render target attachments

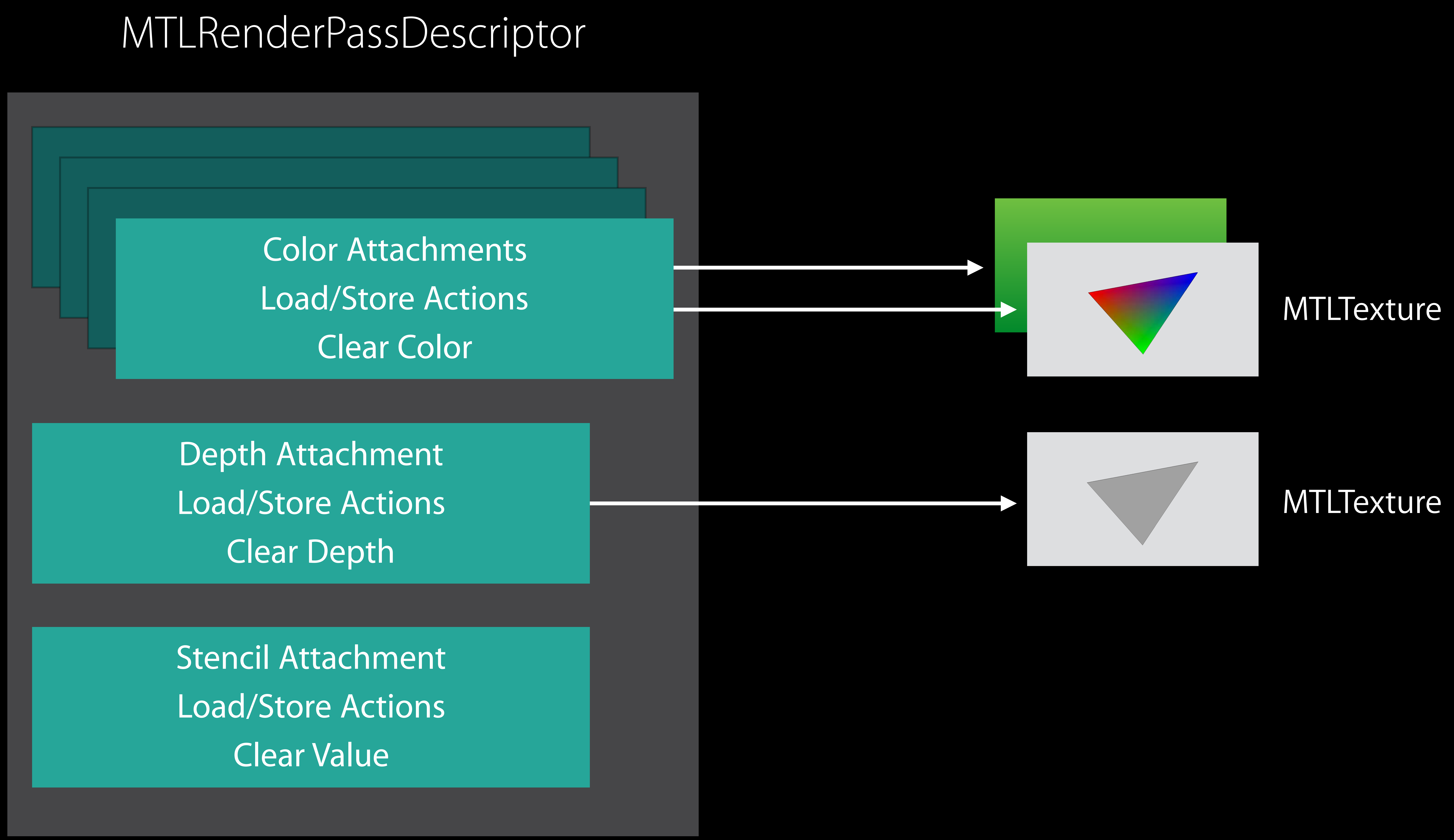
A Single-Pass Frame



A Single-Pass Frame



Render Pass Descriptor



MTLRenderPassDescriptor

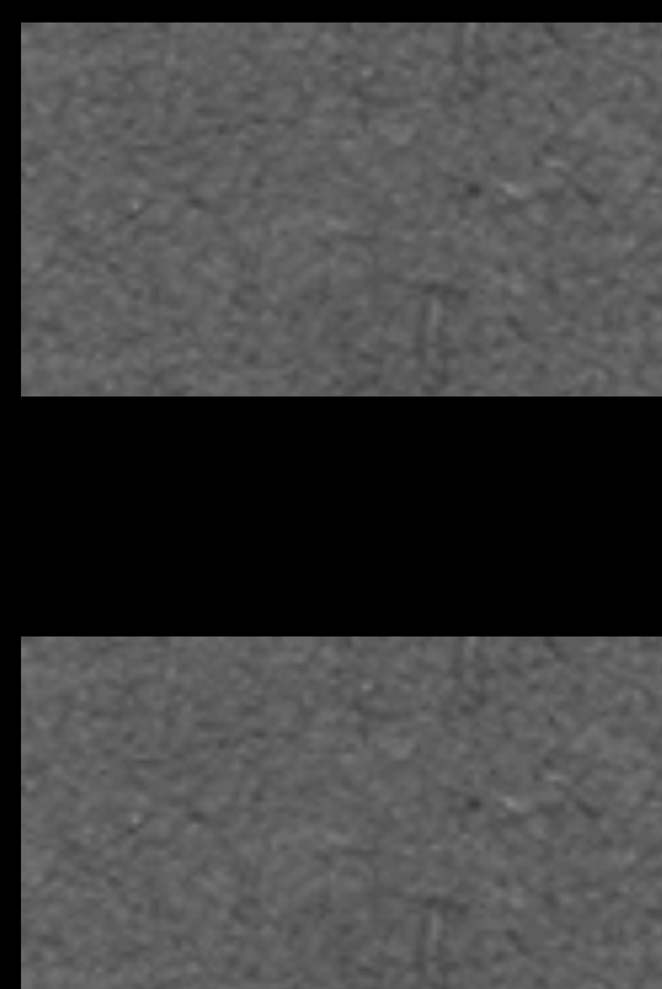
Contains a collection of render pass attachments

- Color, depth, and stencil
- Each refers to a texture to render into
- Also specifies "load and store" actions

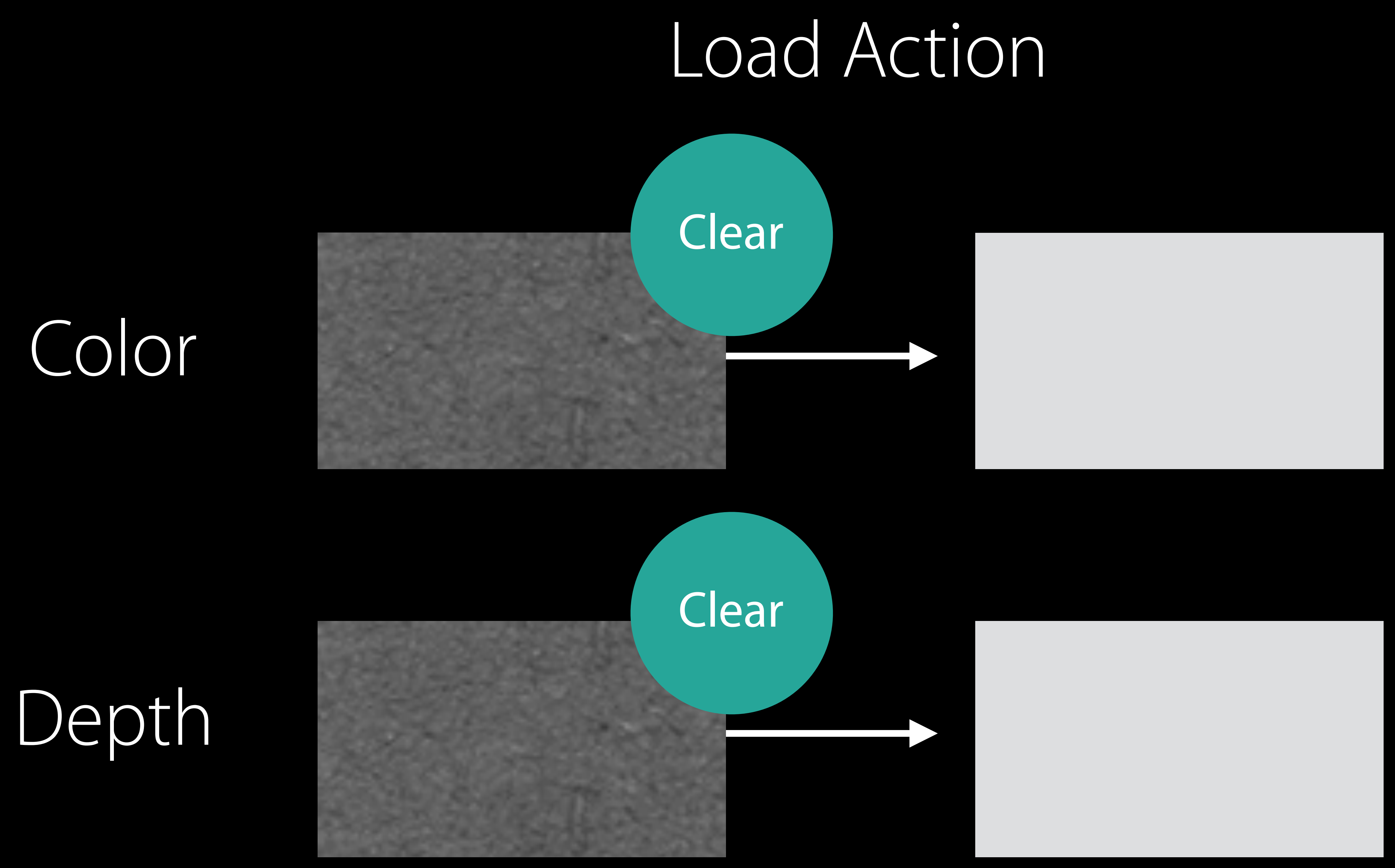
Load and Store Actions

Color

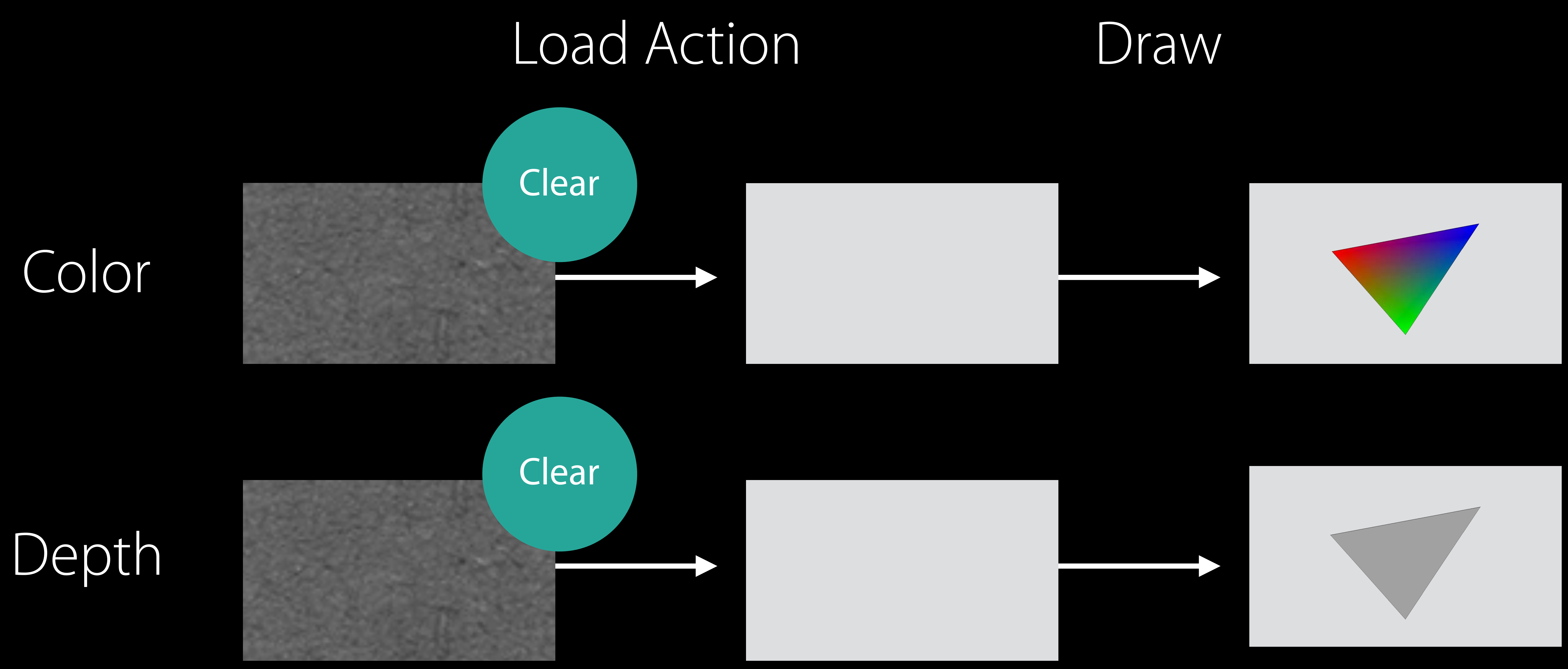
Depth



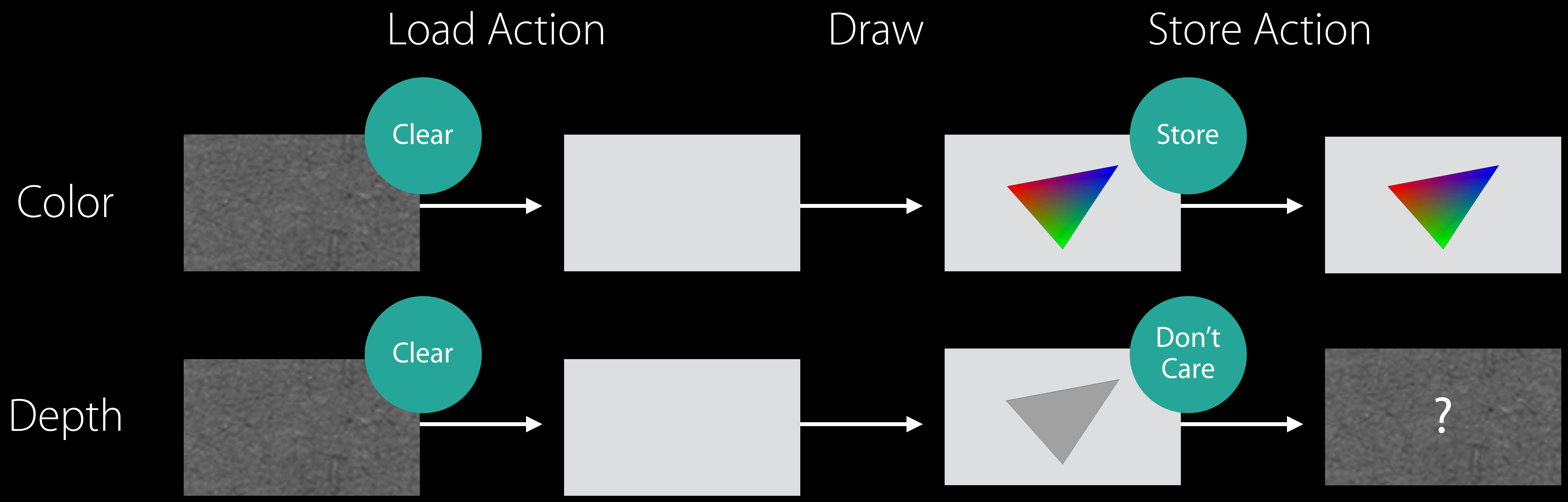
Load and Store Actions



Load and Store Actions



Load and Store Actions



Load and Store Actions

Determine how texture contents should be handled at start and end of pass

Load Actions

- Clear = Clear to specified clear color or clear value
- Load = Load pixel contents with result of previous pass
- Don’t Care

Store Actions

- Store = Write result of rendering into texture
- Don’t Care = Discard result of rendering

```swift
// MTLRenderCommandEncoder
// API


// Ask the view for a configured render pass descriptor (may block!)
let renderPassDescriptor = view.currentRenderPassDescriptor

// Create a render encoder to clear the screen and draw our objects
let renderEncoder = commandBuffer.renderCommandEncoder(with: renderPassDescriptor)
```

Argument Tables

Map from Metal resources to shader parameters

One table per resource or state object type

- Buffer
- Texture
- Sampler

Maximum entry counts vary by device

- Query them

Buffer Argument Table

buffer(0)	Buffer A
buffer(1)	Buffer B
buffer(2)	null
…	…
buffer(n-1)	…

Texture Argument Table

texture(0)	Texture A
texture(1)	Texture C
texture(2)	null
…	Texture B
texture(n-1)	…

```swift
// Binding Resources

renderEncoder.setVertexBuffer(vertexBuffer, offset:0, at:0)
```

```
// Binding Resources

renderEncoder.setVertexBuffer(vertexBuffer, offset:0, at:0)
```

Argument table index

```
vertex VertexOut vertex_passthrough(device VertexIn *vertices [[buffer(0)]],
                                    uint vertexId [[vertex_id]])
{
   ...
}
```

Argument table index

```swift
// Setting the Pipeline State
// API


// Set the pipeline state so the GPU knows which vertex and fragment function to invoke.
renderEncoder.setRenderPipelineState(renderPipelineState)
```

```
vertex VertexOut vertex_passthrough(device VertexIn *vertices [[buffer(0)]],
                                   uint vertexId [[vertex_id]])
{
    VertexOut out;
    out.position = vertices[vertexId].position;
    out.color = vertices[vertexId].color;
    return out;
}
```

```
fragment half4 fragment_passthrough(VertexOut fragmentIn [[stage_in]])
{
    return half4(fragmentIn.color);
}
```

```swift
// Setting Additional State
// API


// Since we specified the vertices of our triangles in counter-clockwise
// order, we need to switch from the default of clockwise winding.
renderEncoder.setFrontFacing(.counterClockwise)
```

Draw Calls

Metal has numerous functions for drawing geometry

- Indexed
- Instanced
- Indirect

We'll just look at basic indexed drawing

```swift
// Issuing Draw Calls



// Issue the draw call to draw the indexed geometry of the mesh
renderEncoder.drawIndexedPrimitives(.triangle,
                                    indexCount: 3,
                                    indexType: .uInt16,
                                    indexBuffer: indexBuffer,
                                    indexBufferOffset: 0)
```

```swift
// Concluding a Pass


// We are finished with this render command encoder, so end it.
renderEncoder.endEncoding()
```

Render Command Encoder

Recap

Create or request render pass descriptor

Create a render command encoder

Set a render pipeline state

Set any other necessary state

Issue draw calls

End encoding

```swift
// Create a render encoder to clear the screen and draw our objects
let renderEncoder = commandBuffer.renderCommandEncoder(with: renderPassDescriptor)

// Since we specified the vertices of our triangles in counter-clockwise
// order, we need to switch from the default of clockwise winding.
renderEncoder.setFrontFacing(.counterClockwise)

// Set the pipeline state so the GPU knows which vertex and fragment function to invoke.
renderEncoder.setRenderPipelineState(renderPipelineState)

// Bind the buffer containing the array of vertex structures so we can
// read it in our vertex shader.
renderEncoder.setVertexBuffer(vertexBuffer, offset:0, at:0)

// Issue the draw call to draw the indexed geometry of the mesh
renderEncoder.drawIndexedPrimitives(.triangle,
                                    indexCount: 3,
                                    indexType: .uInt16,
                                    indexBuffer: indexBuffer,
                                    indexBufferOffset: 0)
```

Getting onto the Screen

The first color attachment of the render pass is usually a drawable's texture

A request to present to the screen can be added to a command buffer

Drawable will be displayed once all preceding passes are complete

```
commandBuffer.present(drawable)
```

Finishing Up the Frame

Committing tells the driver the command buffer is ready to execute

```
// Now that we're done issuing commands, we commit our buffer so the GPU can get to work.
commandBuffer.commit()
```

Command Submission

Recap

Create a command queue at startup

Each frame, create a command buffer

Encode one or more passes with render command encoders

Present drawable to screen

Commit command buffer

Demo

Drawing 2D Content

Agenda

- Conceptual Overview
- Creating a Metal Device
- Loading Data
- Metal Shading Language
- Building Pipeline States
- Issuing GPU Commands
- Animation and Texturing
- Managing Dynamic Data
- CPU/GPU Synchronization
- Multithreaded Encoding

Agenda

Conceptual Overview

Creating a Metal Device

Loading Data

Metal Shading Language

Building Pipeline States

Issuing GPU Commands

Animation and Texturing

Managing Dynamic Data

CPU/GPU Synchronization

Multithreaded Encoding

Animation and Texturing in 3D

Moving into 3D

Animating with a constant buffer

Texturing and sampling

Moving into 3D

Specify vertices in model-local space rather than clip space

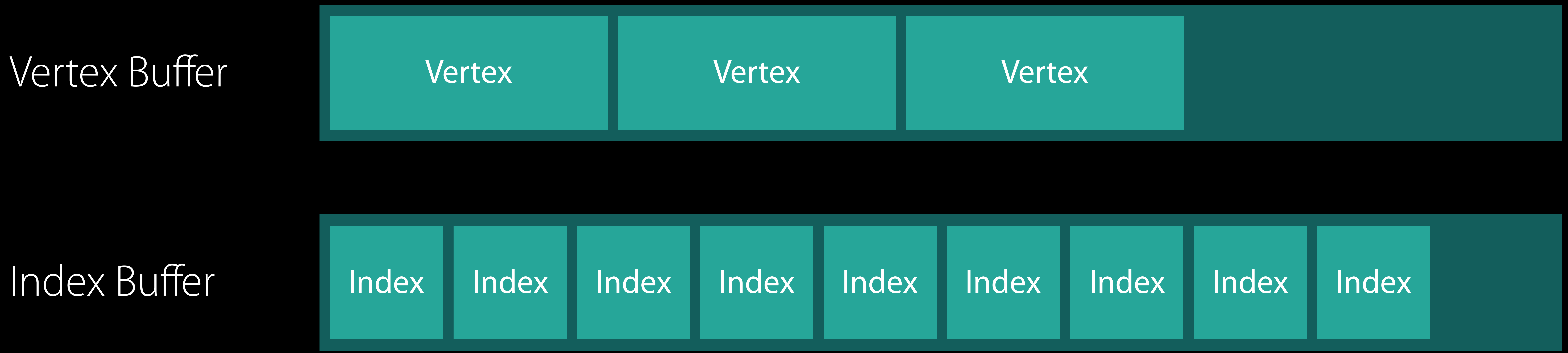
Multiply by a suitable model-view-projection matrix

Add properties for vertex normal and texture coordinates

```swift
// Vertex Format

struct Vertex {
    var position: float3
    var normal: float3
    var texCoords: float2
}
```

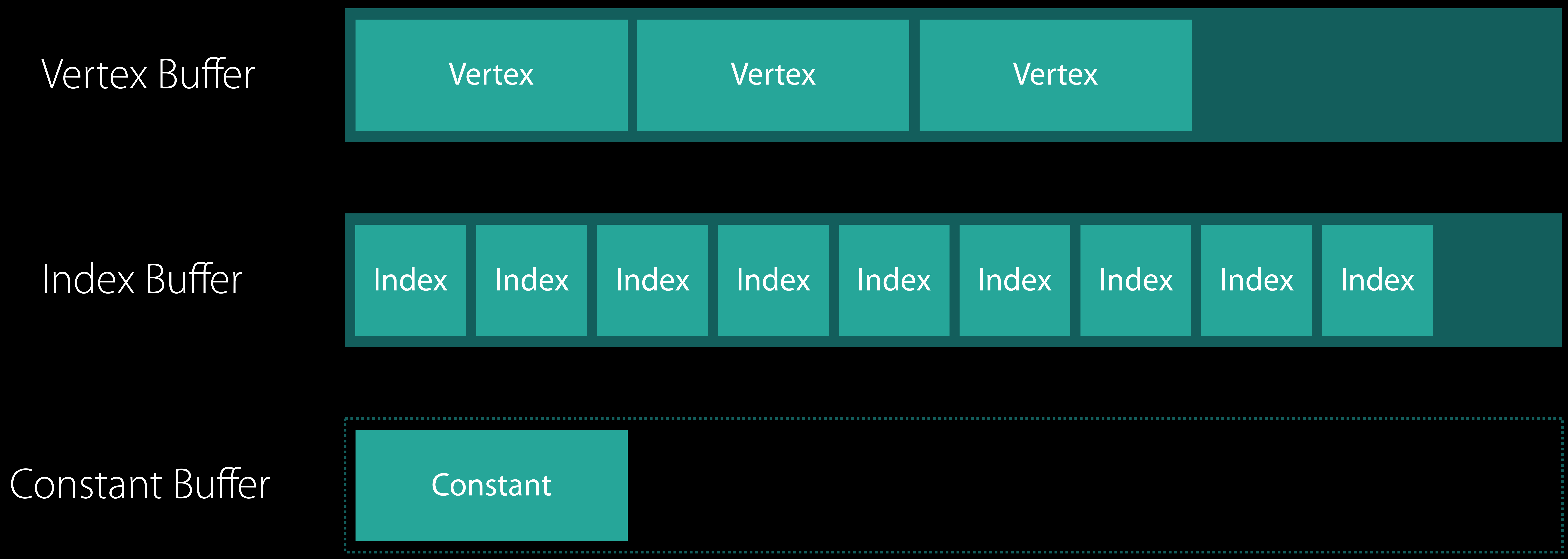
Buffer Layout

+ Constant



Buffer Layout

+ Constant



Binding Small Constant Buffers

For small (< 4kB) pieces of data

Metal implicitly creates and manages buffers

No synchronization concerns

```
renderEncoder.setVertexBytes(&constants, length: strideof(ShaderConstants), at: 1)
```

```swift
// Structure for Gathering Per-Frame Constants
struct Constants {
    var modelViewProjectionMatrix = matrix_identity_float4x4
    var normalMatrix = matrix_identity_float3x3
}

// Construct model-to-world, view, and projection matrices
...

// The combined MVP matrix moves our vertices from model space into clip space
let modelViewMatrix = matrix_multiply(viewMatrix, modelToWorldMatrix);
constants.modelViewProjectionMatrix = matrix_multiply(projectionMatrix, modelViewMatrix)
constants.normalMatrix = matrix_inverse_transpose(matrix_upper_left_3x3(modelViewMatrix))

// Bind the uniform buffer so we can read our model-view-projection matrix in the shader.
renderEncoder.setVertexBytes(&constants, length: strideof(ShaderConstants), at: 1)
```

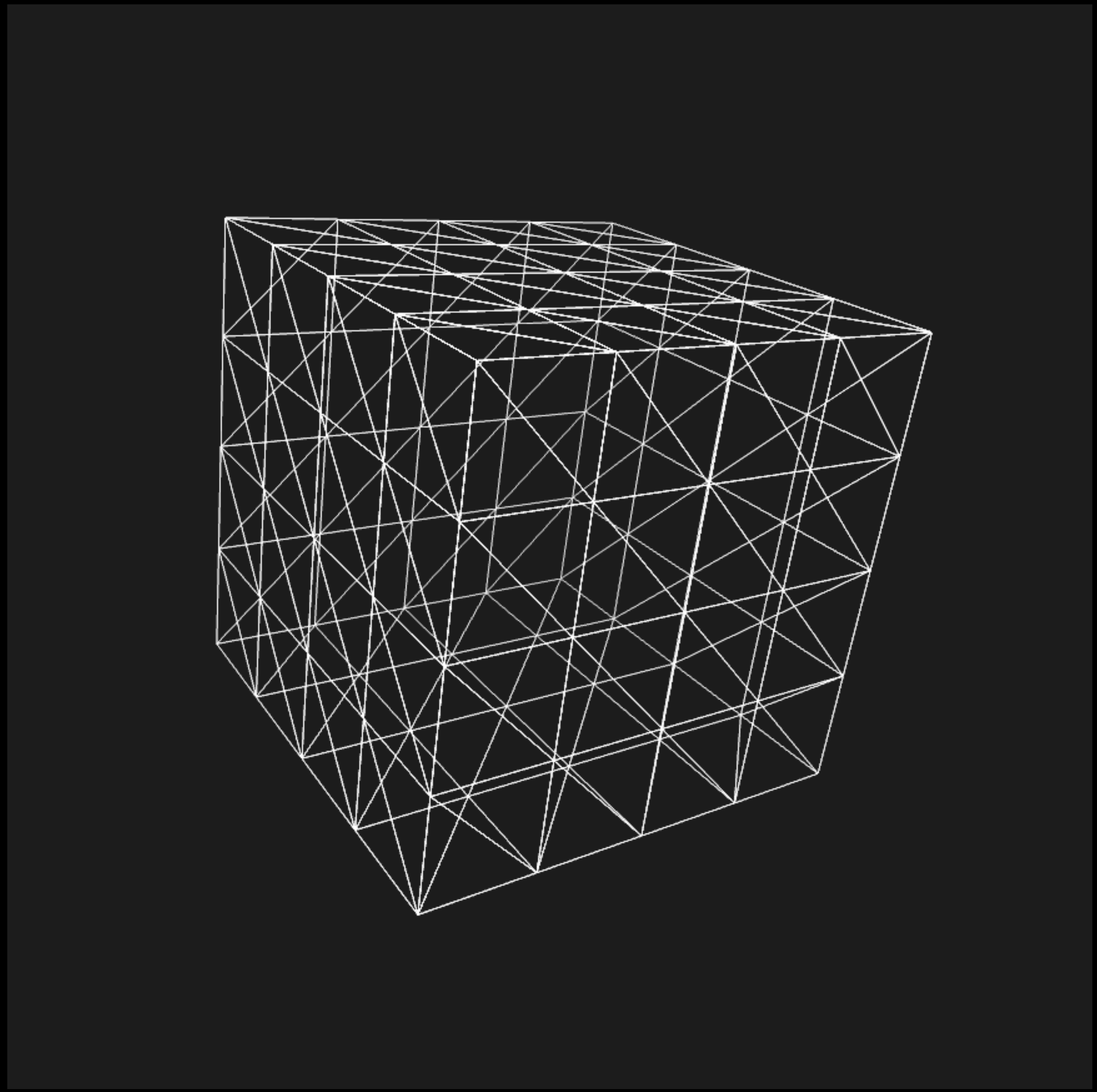
```swift
// Structure for Gathering Per-Frame Constants
struct Constants {
    var modelViewProjectionMatrix = matrix_identity_float4x4
    var normalMatrix = matrix_identity_float3x3
}

// Construct model-to-world, view, and projection matrices
...

// The combined MVP matrix moves our vertices from model space into clip space
let modelViewMatrix = matrix_multiply(viewMatrix, modelToWorldMatrix);
constants.modelViewProjectionMatrix = matrix_multiply(projectionMatrix, modelViewMatrix)
constants.normalMatrix = matrix_inverse_transpose(matrix_upper_left_3x3(modelViewMatrix))

// Bind the uniform buffer so we can read our model-view-projection matrix in the shader.
renderEncoder.setVertexBytes(&constants, length: strideof(ShaderConstants), at: 1)
```

Creating Buffers with Model I/O

Model I/O can generate common shapes

- Box
- Ellipsoid
- Cylinder
- Plane

Get MTLBuffer objects via MetalKit



```swift
let allocator = MTKMeshBufferAllocator(device: device)

let mdlMesh = MDLMesh(boxWithExtent: vector_float3(1, 1, 1),
                      segments: vector_uint3(10, 10, 10),
                      inwardNormals: false,
                      geometryType: .triangles,
                      allocator: allocator)
```

```
let allocator = MTKMeshBufferAllocator(device: device)

let mdlMesh = MDLMesh(boxWithExtent: vector_float3(1, 1, 1),
                      segments: vector_uint3(10, 10, 10),
                      inwardNormals: false,
                      geometryType: .triangles,
                      allocator: allocator)
```

```swift
let allocator = MTKMeshBufferAllocator(device: device)

let mdlMesh = MDLMesh(boxWithExtent: vector_float3(1, 1, 1),
                      segments: vector_uint3(10, 10, 10),
                      inwardNormals: false,
                      geometryType: .triangles,
                      allocator: allocator)
```

```swift
// Attempt to convert the Model I/O mesh to a MetalKit mesh
let mesh: MTKMesh = try MTKMesh(mesh: mdlMesh, device: device)

// Extract the vertex buffer for the whole mesh
let vertexBuffer: MTLBuffer = mesh.vertexBuffers[0].buffer
// Get a reference to the first submesh of the mesh
let submesh = mesh.submeshes[0]
// Extract the index buffer of the submesh
let indexBuffer: MTLBuffer = submesh.indexBuffer.buffer

// Get the primitive type of the mesh (triangle, triangle strip, etc.)
let primitiveType: MTLPrimitiveType = submesh.primitiveType
// Get the number of indices for this submesh
let indexCount: Int = submesh.indexCount
// Get the type of the indices (16-bit or 32-bit uints)
let indexType: MTLIndexType = submesh.indexType
```

```swift
// Attempt to convert the Model I/O mesh to a MetalKit mesh
let mesh: MTKMesh = try MTKMesh(mesh: mdlMesh, device: device)

// Extract the vertex buffer for the whole mesh
let vertexBuffer: MTLBuffer = mesh.vertexBuffers[0].buffer
// Get a reference to the first submesh of the mesh
let submesh = mesh.submeshes[0]
// Extract the index buffer of the submesh
let indexBuffer: MTLBuffer = submesh.indexBuffer.buffer

// Get the primitive type of the mesh (triangle, triangle strip, etc.)
let primitiveType: MTLPrimitiveType = submesh.primitiveType
// Get the number of indices for this submesh
let indexCount: Int = submesh.indexCount
// Get the type of the indices (16-bit or 32-bit uints)
let indexType: MTLIndexType = submesh.indexType
```

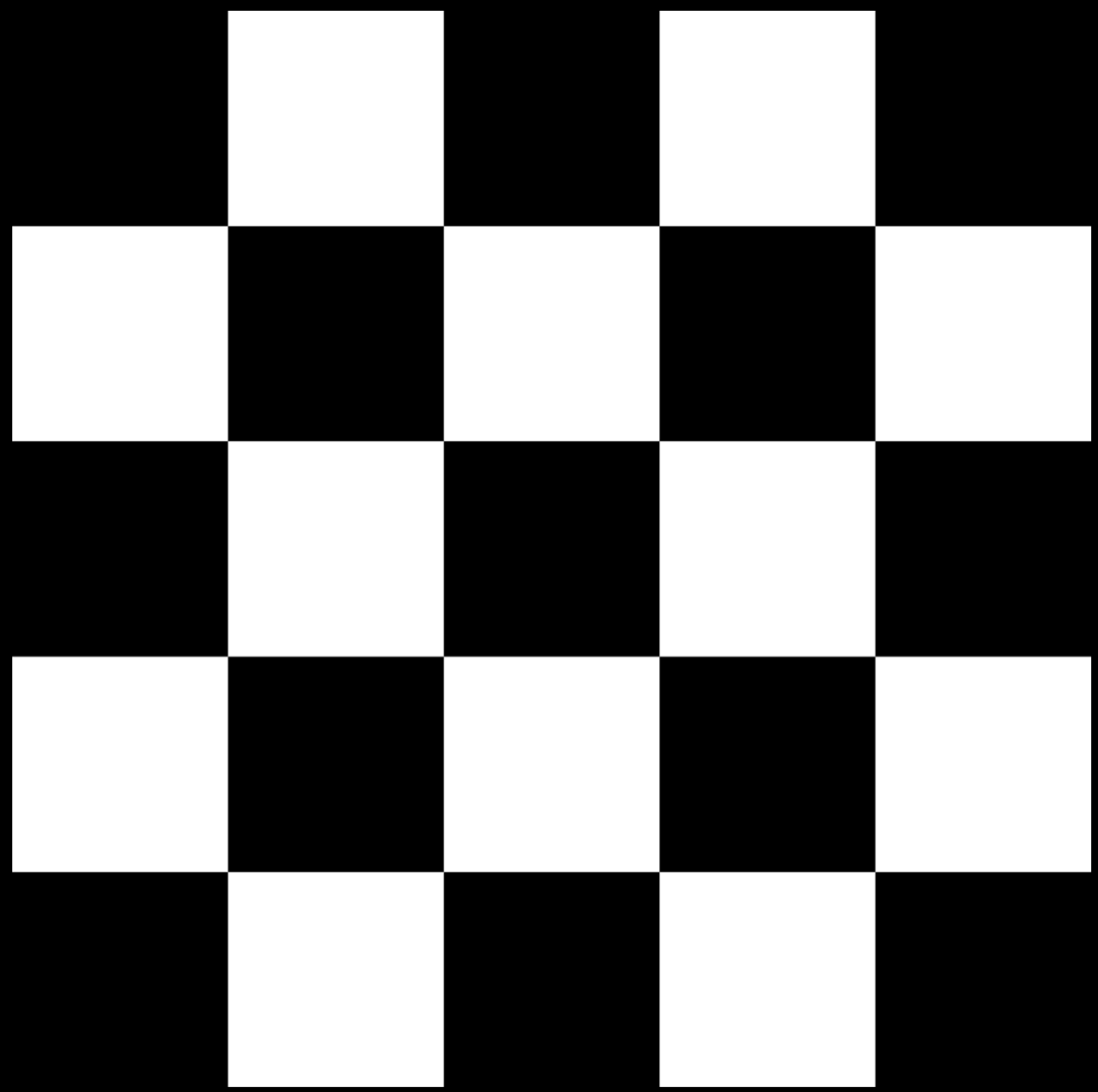
```swift
// Attempt to convert the Model I/O mesh to a MetalKit mesh
let mesh: MTKMesh = try MTKMesh(mesh: mdlMesh, device: device)

// Extract the vertex buffer for the whole mesh
let vertexBuffer: MTLBuffer = mesh.vertexBuffers[0].buffer
// Get a reference to the first submesh of the mesh
let submesh = mesh.submeshes[0]
// Extract the index buffer of the submesh
let indexBuffer: MTLBuffer = submesh.indexBuffer.buffer

// Get the primitive type of the mesh (triangle, triangle strip, etc.)
let primitiveType: MTLPrimitiveType = submesh.primitiveType
// Get the number of indices for this submesh
let indexCount: Int = submesh.indexCount
// Get the type of the indices (16-bit or 32-bit uints)
let indexType: MTLIndexType = submesh.indexType
```

```swift
// Attempt to convert the Model I/O mesh to a MetalKit mesh
let mesh: MTKMesh = try MTKMesh(mesh: mdlMesh, device: device)

// Extract the vertex buffer for the whole mesh
let vertexBuffer: MTLBuffer = mesh.vertexBuffers[0].buffer
// Get a reference to the first submesh of the mesh
let submesh = mesh.submeshes[0]
// Extract the index buffer of the submesh
let indexBuffer: MTLBuffer = submesh.indexBuffer.buffer

// Get the primitive type of the mesh (triangle, triangle strip, etc.)
let primitiveType: MTLPrimitiveType = submesh.primitiveType
// Get the number of indices for this submesh
let indexCount: Int = submesh.indexCount
// Get the type of the indices (16-bit or 32-bit uints)
let indexType: MTLIndexType = submesh.indexType
```

Textures

Blocks of memory in a pre-selected pixel format

Store image data



Texture Descriptors

Parameter objects that gather texture properties

- Texture type (2D, array, cube, etc.)
- Size
- Pixel format
- Mipmap level count

Used by device to create `MTLTextures`

```swift
let descriptor = MTLTextureDescriptor.texture2DDescriptor(with: .bgra8Unorm,
                                                          width: 256,
                                                          height: 256,
                                                          mipmapped: true)

let texture = device.newTexture(with: descriptor)

// Load with texture data and generate mipmaps
...
```

```swift
let descriptor = MTLTextureDescriptor.texture2DDescriptor(with: .bgra8Unorm,
                                                           width: 256,
                                                           height: 256,
                                                           mipmapped: true)

let texture = device.newTexture(with: descriptor)

// Load with texture data and generate mipmaps
...
```

MTKTextureLoader

Easier texture creation

Utility class provided by MetalKit

Load images from:

- Asset catalogs
- File URLs
- Pre-existing `CGImages`

Generates and populates `MTLTextures` of appropriate size and format

```swift
// Create a texture loader with a MTLDevice
let textureLoader = MTKTextureLoader(device: device)

// Fetch the asset from the asset catalog
let asset = NSDataAsset.init(name: "asset-name")

// Load and use the texture if we successfully retrieved the asset
if let data = asset?.data {
    let texture = try textureLoader.newTexture(with: data, options: [:])
    // ...
}
```

```swift
// Create a texture loader with a MTLDevice
let textureLoader = MTKTextureLoader(device: device)

// Fetch the asset from the asset catalog
let asset = NSDataAsset.init(name: "asset-name")

// Load and use the texture if we successfully retrieved the asset
if let data = asset?.data {
    let texture = try textureLoader.newTexture(with: data, options: [:])
    // ...
}
```

```swift
// Create a texture loader with a MTLDevice
let textureLoader = MTKTextureLoader(device: device)

// Fetch the asset from the asset catalog
let asset = NSDataAsset.init(name: "asset-name")

// Load and use the texture if we successfully retrieved the asset
if let data = asset?.data {
    let texture = try textureLoader.newTexture(with: data, options: [:])
    // ...
}
```

```swift
// Create a texture loader with a MTLDevice
let textureLoader = MTKTextureLoader(device: device)

// Fetch the asset from the asset catalog
let asset = NSDataAsset.init(name: "asset-name")

// Load and use the texture if we successfully retrieved the asset
if let data = asset?.data {
    let texture = try textureLoader.newTexture(with: data, options: [:])
    // ...
}
```

Samplers

Contain state related to texture sampling

- Filtering modes
  - Nearest
  - Linear
- Address modes
  - Wrap
  - Clamp to edge
  - Clamp to zero
- LOD

```swift
// Creating a Sampler Object

let samplerDescriptor = MTLSamplerDescriptor()
samplerDescriptor.sAddressMode = .repeat
samplerDescriptor.tAddressMode = .repeat
samplerDescriptor.minFilter = .nearest
samplerDescriptor.magFilter = .linear
let sampler = device.newSamplerState(with: samplerDescriptor)
```

```swift
// Creating a Sampler Object


let samplerDescriptor = MTLSamplerDescriptor()
samplerDescriptor.sAddressMode = .repeat
samplerDescriptor.tAddressMode = .repeat
samplerDescriptor.minFilter = .nearest
samplerDescriptor.magFilter = .linear
let sampler = device.newSamplerState(with: samplerDescriptor)
```

```swift
// Creating a Sampler Object

let samplerDescriptor = MTLSamplerDescriptor()
samplerDescriptor.sAddressMode = .repeat
samplerDescriptor.tAddressMode = .repeat
samplerDescriptor.minFilter = .nearest
samplerDescriptor.magFilter = .linear
let sampler = device.newSamplerState(with: samplerDescriptor)
```

```swift
// Creating a Sampler Object

let samplerDescriptor = MTLSamplerDescriptor()
samplerDescriptor.sAddressMode = .repeat
samplerDescriptor.tAddressMode = .repeat
samplerDescriptor.minFilter = .nearest
samplerDescriptor.magFilter = .linear
let sampler = device.newSamplerState(with: samplerDescriptor)
```

```swift
// Binding Textures and Samplers



// Bind our texture so we can sample from it in the fragment shader
renderEncoder.setFragmentTexture(texture, at: 0)

// Bind our sampler state so we can use it to sample the texture in the fragment shader
renderEncoder.setFragmentSamplerState(sampler, at: 0)
```

The Vertex Function

Multiplies by the model-view-projection matrix from the constant buffer

Transforms vertex positions from model-local space to clip space

Transforms vertex normals from model-local space to eye space for lighting

```
vertex VertexOut vertex_transform(device VertexIn *vertices [[buffer(0)]],
                                  constant Constants &uniforms [[buffer(1)]],
                                  uint vertexId [[vertex_id]])
{
   VertexOut out;
   // Multiplying the position by the model-view-projection matrix moves us into clip space
   out.position = uniforms.modelViewProjectionMatrix * vertices[vertexId].position;
   // Transform the vertex normal into eye space so we can use it for lighting
   out.normal = uniforms.normalMatrix * vertices[vertexId].normal;
   // Just copy the tex coords so they can be interpolated by the rasterizer
   out.texCoords = vertices[vertexId].texCoords;
   return out;
}
```

```
vertex VertexOut vertex_transform(device VertexIn *vertices [[buffer(0)]],
                                  constant Constants &uniforms [[buffer(1)]],
                                  uint vertexId [[vertex_id]])
{
   VertexOut out;
   // Multiplying the position by the model-view-projection matrix moves us into clip space
   out.position = uniforms.modelViewProjectionMatrix * vertices[vertexId].position;
   // Transform the vertex normal into eye space so we can use it for lighting
   out.normal = uniforms.normalMatrix * vertices[vertexId].normal;
   // Just copy the tex coords so they can be interpolated by the rasterizer
   out.texCoords = vertices[vertexId].texCoords;
   return out;
}
```

```
vertex VertexOut vertex_transform(device VertexIn *vertices [[buffer(0)]],
                                  constant Constants &uniforms [[buffer(1)]],
                                  uint vertexId [[vertex_id]])
{
    VertexOut out;
    // Multiplying the position by the model-view-projection matrix moves us into clip space
    out.position = uniforms.modelViewProjectionMatrix * vertices[vertexId].position;
    // Transform the vertex normal into eye space so we can use it for lighting
    out.normal = uniforms.normalMatrix * vertices[vertexId].normal;
    // Just copy the tex coords so they can be interpolated by the rasterizer
    out.texCoords = vertices[vertexId].texCoords;
    return out;
}
```

```
vertex VertexOut vertex_transform(device VertexIn *vertices [[buffer(0)]],
                                  constant Constants &uniforms [[buffer(1)]],
                                  uint vertexId [[vertex_id]])
{
   VertexOut out;
   // Multiplying the position by the model-view-projection matrix moves us into clip space
   out.position = uniforms.modelViewProjectionMatrix * vertices[vertexId].position;
   // Transform the vertex normal into eye space so we can use it for lighting
   out.normal = uniforms.normalMatrix * vertices[vertexId].normal;
   // Just copy the tex coords so they can be interpolated by the rasterizer
   out.texCoords = vertices[vertexId].texCoords;
   return out;
}
```

```
vertex VertexOut vertex_transform(device VertexIn *vertices [[buffer(0)]],
                                  constant Constants &uniforms [[buffer(1)]],
                                  uint vertexId [[vertex_id]])
{
    VertexOut out;
    // Multiplying the position by the model-view-projection matrix moves us into clip space
    out.position = uniforms.modelViewProjectionMatrix * vertices[vertexId].position;
    // Transform the vertex normal into eye space so we can use it for lighting
    out.normal = uniforms.normalMatrix * vertices[vertexId].normal;
    // Just copy the tex coords so they can be interpolated by the rasterizer
    out.texCoords = vertices[vertexId].texCoords;
    return out;
}
```

The Fragment Function

Computes ambient and diffuse lighting

Samples from texture to apply texture to surface

```
fragment half4 fragment_lit_textured(VertexOut fragmentIn [[stage_in]],
                                     texture2d<float, access::sample> tex2d [[texture(0)]],
                                     sampler sampler2d [[sampler(0)]])
{
    // Sample the texture to get the surface color at this point
    half3 surfaceColor = half3(tex2d.sample(sampler2d, fragmentIn.texCoords).rgb);
    // Re-normalize the interpolated surface normal
    half3 normal = normalize(half3(fragmentIn.normal));
    // Compute the ambient color contribution
    half3 color = ambientLightIntensity * surfaceColor;
    // Calculate the diffuse factor as the dot product of the normal and light direction
    float diffuseFactor = saturate(dot(normal, -lightDirection));
    // Add in the diffuse contribution from the light
    color += diffuseFactor * diffuseLightIntensity * surfaceColor;
    return half4(color, 1);
}
```

```
fragment half4 fragment_lit_textured(VertexOut fragmentIn [[stage_in]],
                                     texture2d<float, access::sample> tex2d [[texture(0)]],
                                     sampler sampler2d [[sampler(0)]])
{
    // Sample the texture to get the surface color at this point
    half3 surfaceColor = half3(tex2d.sample(sampler2d, fragmentIn.texCoords).rgb);
    // Re-normalize the interpolated surface normal
    half3 normal = normalize(half3(fragmentIn.normal));
    // Compute the ambient color contribution
    half3 color = ambientLightIntensity * surfaceColor;
    // Calculate the diffuse factor as the dot product of the normal and light direction
    float diffuseFactor = saturate(dot(normal, -lightDirection));
    // Add in the diffuse contribution from the light
    color += diffuseFactor * diffuseLightIntensity * surfaceColor;
    return half4(color, 1);
}
```

```
fragment half4 fragment_lit_textured(VertexOut fragmentIn [[stage_in]],
                                     texture2d<float, access::sample> tex2d [[texture(0)]],
                                     sampler sampler2d [[sampler(0)]])
{
    // Sample the texture to get the surface color at this point
    half3 surfaceColor = half3(tex2d.sample(sampler2d, fragmentIn.texCoords).rgb);
    // Re-normalize the interpolated surface normal
    half3 normal = normalize(half3(fragmentIn.normal));
    // Compute the ambient color contribution
    half3 color = ambientLightIntensity * surfaceColor;
    // Calculate the diffuse factor as the dot product of the normal and light direction
    float diffuseFactor = saturate(dot(normal, -lightDirection));
    // Add in the diffuse contribution from the light
    color += diffuseFactor * diffuseLightIntensity * surfaceColor;
    return half4(color, 1);
}
```

```
fragment half4 fragment_lit_textured(VertexOut fragmentIn [[stage_in]],
                                     texture2d<float, access::sample> tex2d [[texture(0)]],
                                     sampler sampler2d [[sampler(0)]])
{
    // Sample the texture to get the surface color at this point
    half3 surfaceColor = half3(tex2d.sample(sampler2d, fragmentIn.texCoords).rgb);
    // Re-normalize the interpolated surface normal
    half3 normal = normalize(half3(fragmentIn.normal));
    // Compute the ambient color contribution
    half3 color = ambientLightIntensity * surfaceColor;
    // Calculate the diffuse factor as the dot product of the normal and light direction
    float diffuseFactor = saturate(dot(normal, -lightDirection));
    // Add in the diffuse contribution from the light
    color += diffuseFactor * diffuseLightIntensity * surfaceColor;
    return half4(color, 1);
}
```

```
fragment half4 fragment_lit_textured(VertexOut fragmentIn [[stage_in]],
                                     texture2d<float, access::sample> tex2d [[texture(0)]],
                                     sampler sampler2d [[sampler(0)]])
{
    // Sample the texture to get the surface color at this point
    half3 surfaceColor = half3(tex2d.sample(sampler2d, fragmentIn.texCoords).rgb);
    // Re-normalize the interpolated surface normal
    half3 normal = normalize(half3(fragmentIn.normal));
    // Compute the ambient color contribution
    half3 color = ambientLightIntensity * surfaceColor;
    // Calculate the diffuse factor as the dot product of the normal and light direction
    float diffuseFactor = saturate(dot(normal, -lightDirection));
    // Add in the diffuse contribution from the light
    color += diffuseFactor * diffuseLightIntensity * surfaceColor;
    return half4(color, 1);
}
```

Demo

Drawing 3D textured content

Summary

Metal is a powerful, low-overhead GPU programming technology

Doing expensive work up front saves time where it matters most

Explicit memory management and command submission let you work smarter

More Information

https://developer.apple.com/wwdc16/602

Related Sessions

Adopting Metal, Part 2	Nob Hill	Tuesday 3:00PM
What's New in Metal, Part 1	Pacific Heights	Wednesday 11:00AM
What's New in Metal, Part 2	Pacific Heights	Wednesday 1:40PM
Advanced Metal Shader Optimization	Pacific Heights	Wednesday 3:00PM

Labs

Metal Lab	Graphics, Games, and Media Lab A	Tuesday 4:00PM
Metal Lab	Graphics, Games, and Media Lab A	Thursday 12:00PM

WWDC16