What’s New in Metal

Part 1

Session 604

Aaftab Munshi GPU Software Engineer
James Ding GPU Software Engineer
Jose Enrique D’Arnaude del Castillo GPU Software Engineer
Alp Yucebilgin GPU Software Engineer
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
Agenda

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

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

Function Specialization and Function Resource Read-Writes

Wide Color and Texture Assets

Additions to Metal Performance Shaders
Agenda

Tessellation

Resource Heaps and Memoryless Render Targets

Improved Tools

Function Specialization and Function Resource Read-Writes

Wide Color and Texture Assets

Additions to Metal Performance Shaders
Tessellation

Aaftab Munshi  GPU Software Engineer
Motivation
Motivation

Large increase in memory bandwidth required to render high resolution geometry
Motivation

Large increase in memory bandwidth required to render high resolution geometry
Describe input geometry as a coarser or lower resolution model
Motivation

Large increase in memory bandwidth required to render high resolution geometry

Describe input geometry as a coarser or lower resolution model

Generate high resolution model from the coarser model

• High resolution model not stored in graphics memory

• Method to generate high resolution model is programmable

Tessellation — A technique to amplify and refine the geometric details of an object
Tessellation in Action
GFXBench 4.0
Tessellation in Action
GFXBench 4.0
Tessellation in Action

GFXBench 4.0
Tessellation in Action

GFXBench 4.0
Tessellation in Action

GFXBench 4.0
Tessellation in Action
GFXBench 4.0
Tessellation in Metal
Metal Tessellation
A simpler and efficient approach
Metal Tessellation

A simpler and efficient approach

Modern clean-sheet approach
Easy to use
Efficient and performant
Tessellation in Metal

Availability

macOS Sierra

All Configurations
Tessellation in Metal
Availability

macOS Sierra
All Configurations

iOS
A9 Processor
Tessellation in Metal

Topics we’ll cover
Tessellation in Metal

Topics we’ll cover

Metal Graphics Pipeline
Rendering Geometry
Adopting Metal Tessellation
Metal Graphics Pipeline
With Tessellation
Tessellation Input Primitives

Patches
Tessellation Input Primitives

Patches

A patch is a parametric surface made from spline curves
Described by a set of control vertices
Tessellation Input Primitives

Patches

A patch is a parametric surface made from spline curves.
Described by a set of control vertices.
Tessellation controls how to render the patch as triangles.
Tessellation Input Primitives

Patches

A **patch** is a parametric surface made from spline curves.

Described by a set of control vertices.

Tessellation controls how to render the patch as triangles.
A patch is a parametric surface made from spline curves. Described by a set of control vertices. Tessellation controls how to render the patch as triangles.
Metal Tessellation
First stage — Tessellation kernel

Graphics Memory
Metal Tessellation
First stage — Tessellation kernel

A programmable stage
Metal Tessellation
First stage — Tessellation kernel

A programmable stage
Computations to generate
• Tessellation factors for a patch — determines how much to subdivide the patch
• Additional patch data, if required
Metal Tessellation
Second stage — Tessellator

Graphics Memory

Patch Data
Tessellation Factors

Tessellation Kernel
Metal Tessellation
Second stage — Tessellator

A fixed-function stage but configurable
Metal Tesselllation
Second stage — Tessellator

A fixed-function stage but configurable
- Subdivides the patch into triangles using the tessellation factors
- Triangles describe the high resolution geometry to draw
Metal Tessellation
Second stage — Tessellator

A fixed-function stage but configurable

- Subdivides the patch into triangles using the tessellation factors
- Triangles describe the high resolution geometry to draw
- Triangle list not stored in graphics memory
- Generates parametric \((u, v)\) values for vertices on subdivided patch
Metal Tessellation

Third stage — Post Tessellation Vertex Shader

- Tessellation Kernel
- Tessellator
- Graphics Memory
- Patch Data
- Tessellation Factors
Metal Tessellation

Third stage — Post Tessellation Vertex Shader

A programmable stage
Metal Tessellation

Third stage — Post Tessellation Vertex Shader

A programmable stage

- Executes for vertices generated by tessellator
- Evaluates position and other attributes on high resolution surface
- Similar role as the domain shader in DirectX
Metal Tessellation
Complete pipeline
Metal Tessellation
Complete pipeline

- Tessellation factors and patch data can also be generated by vertex or fragment shader
  - Compute kernel preferred as it can asynchronously execute with draw commands on GPU
Metal Tessellation
Complete pipeline

Tessellation factors and patch data can also be generated by vertex or fragment shader
• Compute kernel preferred as it can asynchronously execute with draw commands on GPU
Tessellation kernel does not need to execute every frame
• Apply a scale value to the tessellation factors
Metal Graphics Pipeline
Without Tessellation
Metal Graphics Pipeline

Without Tessellation

Graphics Memory

Vertex Data Buffers, Textures

Vertex Shader → Rasterizer → Fragment Shader
Metal Graphics Pipeline
With Tessellation
Metal Graphics Pipeline
With Tessellation

- Tessellator
- Post-Tessellation Vertex Shader
- Rasterizer
- Fragment Shader

Flow:
- Tessellation Factors
- Patch Data Buffers, Textures
- Graphics Memory
Rendering Geometry with Tessellation
Rendering Geometry with Tessellation
Rendering Geometry with Tessellation

Writing a Post-Tessellation Vertex Shader
Specifying Patch Data Inputs
Configuring the Tessellator
Drawing Patches
Writing a Post-Tessellation Vertex Shader

Meet the new shader, same as the old shader

```cpp
[[ patch(quad, 16) ]]

vertex VertexOutput

myPostTessellationVertexShader(uint patchID [[ patch_id ]],
    float2 patchUV [[ position_in_patch ]],
    MyPatchData patchData [[ stage_in ]],
    ...)
```
A vertex shader that executes for vertices of a tessellated surface

```glsl
[[ patch(quad, 16) ]]  
vertex VertexOutput  
myPostTessellationVertexShader(uint patchID [[ patch_id ]],  
    float2 patchUV [[ position_in_patch ]],  
    MyPatchData patchData [[ stage_in ]],  
    ...)  
```
Writing a Post-Tessellation Vertex Shader

Meet the new shader, same as the old shader

A vertex shader that executes for vertices of a tessellated surface

```glsl
[[ patch(quad, 16) ]] 
vertex VertexOutput

myPostTessellationVertexShader(uint patchID [[ patch_id ]],
                                  float2 patchUV [[ position_in_patch ]],
                                  MyPatchData patchData [[ stage_in ]],
                                  ...
)```
Writing a Post-Tessellation Vertex Shader

Meet the new shader, same as the old shader

A vertex shader that executes for vertices of a tessellated surface

Shader Inputs

- Patch Data vs. Vertex Data

```cpp
[[ patch(quad, 16) ]]

vertex VertexOutput

myPostTessellationVertexShader(
  uint patchID [[ patch_id ]],
  float2 patchUV [[ position_in_patch ]],
  MyPatchData patchData [[ stage_in ]],
  ...
)
```
Writing a Post-Tessellation Vertex Shader
Meet the new shader, same as the old shader

A vertex shader that executes for vertices of a tessellated surface

Shader Inputs
• Patch Data vs. Vertex Data

```cpp
[[ patch(quad, 16) ]]
vertex VertexOutput
myPostTessellationVertexShader(uint patchID [[ patch_id ]],
  float2 patchUV [[ position_in_patch ]],
  MyPatchData patchData [[ stage_in ]],
  ...
)
Writing a Post-Tessellation Vertex Shader

Meet the new shader, same as the old shader

A vertex shader that executes for vertices of a tessellated surface

Shader Inputs

- Patch Data vs. Vertex Data

```shadertex
[[ patch(quad, 16) ]]
vertex VertexOutput

myPostTessellationVertexShader(\n  uint patchID [[ patch_id ]],\n  float2 patchUV [[ position_in_patch ]],\n  MyPatchData patchData [[ stage_in ]],\n  ...
)```

[132x359]
[132x303]vertex VertexOutput
[132x247]myPostTessellationVertexShader(uint patchID [[ patch_id ]],
[132x191]float2 patchUV [[ position_in_patch ]],
[132x135]MyPatchData patchData [[ stage_in ]],
[132x79]…)
[125x807]A vertex shader that executes for vertices of a tessellated surface
[125x727]Shader Inputs
[124x962]Meet the new shader, same as the old shader
[120x1061]Writing a Post-Tessellation Vertex Shader
```shadertex
[[ patch(quad, 16) ]]
vertex VertexOutput

myPostTessellationVertexShader(\n  uint patchID [[ patch_id ]],\n  float2 patchUV [[ position_in_patch ]],\n  MyPatchData patchData [[ stage_in ]],\n  ...
)```
Writing a Post-Tessellation Vertex Shader

Meet the new shader, same as the old shader

A vertex shader that executes for vertices of a tessellated surface

Shader Inputs
- Patch Data vs. Vertex Data

Shader Outputs
- Similar to outputs from a vertex shader

```cpp
[[ patch(quad, 16) ]]

vertex VertexOutput

myPostTessellationVertexShader(uint patchID [[ patch_id ]],
    float2 patchUV [[ position_in_patch ]],
    MyPatchData patchData [[ stage_in ]],
    ...)
```
Patch Data Inputs
Two types of input data
Patch Data Inputs

Two types of input data

Per-patch and control-point data

• Multiple control-points per patch
  - Example: 16 control-points in a Bezier patch
Patch Data Inputs
Two types of input data

Per-patch and control-point data

- Multiple control-points per patch
  - Example: 16 control-points in a Bezier patch

Use `MTLVertexDescriptor` to specify the patch data layout in memory
Patch Data Inputs

Two types of input data

Per-patch and control-point data

- Multiple control-points per patch
  - Example: 16 control-points in a Bezier patch

Use **MTLVertexDescriptor** to specify the patch data layout in memory

Declared as a struct with **[[ stage_in ]]** qualifier in post-tessellation vertex shader

- Attribute index used to identify and match patch input data with **MTLVertexDescriptor**
- Control-point data specified as a templated type
/ Control-Point Data

```cpp
struct ControlPointData {
    float3 position [[ attribute(0) ]];
    uint   a        [[ attribute(1) ]];
};
```

// Per-Patch Data

```cpp
struct PerPatchData {
    float b [[ attribute(2) ]];
    uint  c [[ attribute(3) ]];
};
```

// struct that stores the per-patch and control-point data

```cpp
struct MyPatchData {
    PerPatchData patchData;
    patch_control_point<ControlPointData> controlPointData;
};
```

// post-tessellation vertex shader

```cpp
[[ patch(quad, 16) ]] vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], …)
```
// Control-Point Data
struct ControlPointData {
    float3 position [[ attribute(0) ]];
    uint a        [[ attribute(1) ]];
};

// Per-Patch Data
struct PerPatchData {
    float b [[ attribute(2) ]];
    uint  c [[ attribute(3) ]];
};

// struct that stores the per-patch and control-point data
struct MyPatchData {
    PerPatchData patchData;
    patch_control_point<ControlPointData> controlPointData;
};

// post-tessellation vertex shader
[[ patch(quad, 16) ]]
vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], …)
// Control-Point Data
struct ControlPointData {
    float3 position [[ attribute(0) ]];
    uint a [[ attribute(1) ]];
};

// Per-Patch Data
struct PerPatchData {
    float b [[ attribute(2) ]];
    uint c [[ attribute(3) ]];
};

// struct that stores the per-patch and control-point data
struct MyPatchData {
    PerPatchData patchData;
    patch_control_point<ControlPointData> controlPointData;
};

// post-tessellation vertex shader
[[ patch(quad, 16) ]] vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], ...)
// Control-Point Data
struct ControlPointData {
  float3 position [[ attribute(0) ]];
  uint a [[ attribute(1) ]];
};

// Per-Patch Data
struct PerPatchData {
  float b [[ attribute(2) ]];
  uint c [[ attribute(3) ]];
};

// struct that stores the per-patch and control-point data
struct MyPatchData {
  PerPatchData patchData;
  patch_control_point<ControlPointData> controlPointData;
};

// post-tessellation vertex shader
[[ patch(quad, 16) ]]
vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], …)
// Control-Point Data

```cpp
struct ControlPointData {
    float3 position [[ attribute(0) ]];
    uint   a        [[ attribute(1) ]];
};
```

// Per-Patch Data

```cpp
struct PerPatchData {
    float b [[ attribute(2) ]];
    uint  c [[ attribute(3) ]];
};
```

// struct that stores the per-patch and control-point data

```cpp
struct MyPatchData {
    PerPatchData patchData;
    patch_control_point<ControlPointData> controlPointData;
};
```

// post-tessellation vertex shader

```cpp
[[ patch(quad, 16) ]]
vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], …)
```
// Control-Point Data
struct ControlPointData {
    float3 position [[ attribute(0) ]];
    uint a [[ attribute(1) ]];
};

// Per-Patch Data
struct PerPatchData {
    float b [[ attribute(2) ]];
    uint c [[ attribute(3) ]];
};

// struct that stores the per-patch and control-point data
struct MyPatchData {
    PerPatchData patchData;
    patch_control_point<ControlPointData> controlPointData;
};

// post-tessellation vertex shader
[[ patch(quad, 16) ]]
vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], ...)
// Control-Point Data
struct ControlPointData {
    float3 position [[ attribute(0) ]];
    uint   a        [[ attribute(1) ]];
};

// Per-Patch Data
struct PerPatchData {
    float  b [[ attribute(2) ]];
    uint   c [[ attribute(3) ]];
};

// struct that stores the per-patch and control-point data
struct MyPatchData {
    PerPatchData patchData;
    patch_control_point<ControlPointData> controlPointData;
};

// post-tessellation vertex shader
[[ patch(quad, 16) ]]
vertex VertexOutput
myPostTessellationVertexShader(MyPatchData patchData [[ stage_in ]], ...
Configuring the Tessellator
Configuring the Tessellator

Tessellation Properties

• Specified in MTLRenderPipelineDescriptor
Configuring the Tessellator

Tessellation Properties

• Specified in MTLRenderPipelineDescriptor

Tessellation Factors

• Edge and Inside factors

• 16-bit floating-point values

// specify the tessellation factor buffer in the MTLRenderCommandEncoder
renderEncoder.setTessellationFactorBuffer(buffer, offset: offset, instanceStride: stride)
Drawing Patches

New APIs in MTLRenderCommandEncoder
Drawing Patches

New APIs in MTLRenderCommandEncoder

Non-indexed and indexed draw patch APIs

- Specify patch start, number of patches to draw, control-point index buffer …

```swift
renderEncoder.drawPatches(numberOfControlPoints,
    patchStart: patchStart, numberofPatches: n, …)
renderEncoder.drawIndexedPatches(numberOfControlPoints,
    patchStart: patchStart, numberofPatches: n, …,
    controlPointIndexBuffer: buffer,
    controlPointIndexBufferOffset: offset, …)
```
Drawing Patches

New APIs in MTLRenderCommandEncoder

Non-indexed and indexed draw patch APIs

• Specify patch start, number of patches to draw, control-point index buffer …

```swift
renderEncoder.drawPatches(numberOfControlPoints,
    patchStart: patchStart, numberOfPatches: n, …)
renderEncoder.drawIndexedPatches(numberOfControlPoints,
    patchStart: patchStart, numberOfPatches: n, …,
    controlPointIndexBuffer: buffer,
    controlPointIndexBufferOffset: offset, …)
```

DrawIndirect variants

• Allows previous draw or dispatch commands to generate draw parameters

• Draw parameters such as patch start, patch count specified in a buffer filled out on GPU
Adopting Metal Tessellation
“Unity Technologies is proud to be working with Apple to make our Metal renderer the best of its kind, enabling Unity developers to harness the power of Metal. Later this year, we will be shipping support for Metal Tessellation, Metal Compute, and the ability to write native Metal shaders in Unity.”
“Metal provides great performance and efficiency improvements alongside a wonderfully clean programming model. We're very enthusiastic about the ongoing revolution in high-end graphics on iOS and Mac enabled by Metal.”

Epic Games
Demo
Tessellation in Action
Adopting Metal Tessellation
Digital content creation tools
“OpenSubdiv is a key technology at Pixar that helps our artists create expressive performances and beautiful worlds. Pixar is thrilled to see the high performance and full fidelity of OpenSubdiv realized with Metal Tessellation on macOS and iOS devices and with Apple’s contribution of a native Metal implementation to the OpenSubdiv open source project.”

David Yu
Senior Software Engineer of Pixar’s GPU Team
Adopting Metal Tessellation
From DirectX to Metal

Graphics Memory
Adopting Metal Tessellation
From DirectX to Metal
Adopting Metal Tessellation
From DirectX to Metal

One-to-one mapping between domain and post-tessellation vertex shader
Tessellator remains the same
Adopting Metal Tessellation
From DirectX to Metal

One-to-one mapping between domain and post-tessellation vertex shader
Tessellator remains the same
Adopting Metal Tessellation
From DirectX to Metal

One-to-one mapping between domain and post-tessellation vertex shader
Tessellator remains the same
Vertex and hull shader to be translated to a Metal kernel
Moving Existing Tessellation Shaders to Metal

Vertex + Hull Shader = Metal Tessellation Kernel
Moving Existing Tessellation Shaders to Metal

Vertex + Hull Shader = Metal Tessellation Kernel

Vertex descriptor used to describe vertex data layout

- `[[ stage_in ]]` in a Metal kernel
- Input data layout described in a `MTLStageInputOutputDescriptor`
Moving Existing Tessellation Shaders to Metal

Vertex + Hull Shader = Metal Tessellation Kernel

Vertex descriptor used to describe vertex data layout

- `[[ stage_in ]]` in a Metal kernel
- Input data layout described in a `MTLStageInputOutputDescriptor`

Vertex and hull shader observations

- Vertex shader executes for each control-point
- Hull shader described by control-point and per-patch functions
Moving Existing Tessellation Shaders to Metal

Vertex + Hull Shader = Metal Tessellation Kernel

Vertex descriptor used to describe vertex data layout

- `[[ stage_in ]]` in a Metal kernel
- Input data layout described in a `MTLStageInputOutputDescriptor`

Vertex and hull shader observations

- Vertex shader executes for each control-point
- Hull shader described by control-point and per-patch functions

Translate vertex and hull shaders to Metal functions
Metal Tessellation

Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation
Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation

Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation
Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation

Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation
Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation
Vertex + Hull Shader = Metal Tessellation Kernel

- Vertex Function
- Control-Point Hull Function
- Per-Patch Hull Function
- Threadgroup Memory
- Control-Point
- Graphics Memory
Metal Tessellation

Vertex + Hull Shader = Metal Tessellation Kernel
Metal Tessellation

Vertex + Hull Shader = Metal Tessellation Kernel
Tessellation
Summary
Tessellation

Summary

Simple to use and performant
Easy to adapt your existing tessellation code to Metal
Available on iOS and macOS
Tessellation

Summary

Simple to use and performant

Easy to adapt your existing tessellation code to Metal

Available on iOS and macOS

Call to Action

• Use tessellation to improve the visual content rendered by your application
Resource Heaps and Memoryless Render Targets

James Ding GPU Software Engineer
Resource Heaps
Lower overhead resource management

iOS
tvOS
Resource Heaps
Lower overhead resource management

Resource Sub-Allocation

iOS
tvOS
Resource Heaps
Lower overhead resource management

Resource Sub-Allocation
Resource Aliasing

iOS
tvOS
Resource Heaps

Lower overhead resource management

Resource Sub-Allocation
Resource Aliasing
Explicit Command Synchronization

iOS
tvOS
Resource Sub-Allocation
Resource Sub-Allocation

Faster resource creation and binding
Resource Sub-Allocation

Faster resource creation and binding

Resource creation with MTLDevice

- Heavy CPU operation
Resource Sub-Allocation

Faster resource creation and binding

Resource creation with MTLDevice

• Heavy CPU operation
Resource Sub-Allocation

Faster resource creation and binding

Resource creation with MTLDevice

• Heavy CPU operation

Resource binding

• Tracking costs add up for complex scenes
Resource Sub-Allocation
Faster resource creation and binding

Resource creation with MTLDevice
• Heavy CPU operation

Resource binding
• Tracking costs add up for complex scenes
Resource Sub-Allocation

Faster resource creation and binding

Resource creation with MTLDevice

• Heavy CPU operation

Resource binding

• Tracking costs add up for complex scenes

Resource Heaps address these performance issues

• Perform expensive memory operations ahead of time
• Only heaps are tracked, not their resources
Resource Creation
Comparing MTLDevice and MTLHeap
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

- Memory

Resource
Resource Creation

Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Memory
Resource
Resource Creation

Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory

Resource
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory

Resource
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice
Creating Resources with MTLHeap
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory

Resource
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory

Resource
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory

Resource
Resource Creation
Comparing MTLDevice and MTLHeap

Creating Resources with MTLDevice

Creating Resources with MTLHeap

Memory

Resource
// Creating Resources with MTLHeap

// Create heap descriptor, including the heap size.
let heapDesc = MTLHeapDescriptor()
heapDesc.size = heapSize

// Create the heap object ahead of time.
let heap = device.newHeap(with:heapDesc)

// Sub-allocate resources from heap – very fast!
let buffer = heap.newBuffer(withLength:bufferSize)
let texture = heap.newTexture(with:textureDescriptor)
// Creating Resources with MTLHeap

// Create heap descriptor, including the heap size.
let heapDesc = MTLHeapDescriptor()
heapDesc.size = heapSize

// Create the heap object ahead of time.
let heap = device.newHeap(with:heapDesc)

// Sub-allocate resources from heap - very fast!
let buffer = heap.newBuffer(withLength:bufferSize)
let texture = heap.newTexture(with:textureDescriptor)
// Creating Resources with MTLHeap

// Create heap descriptor, including the heap size.
let heapDesc = MTLHeapDescriptor()
heapDesc.size = heapSize

// Create the heap object ahead of time.
let heap = device.newHeap(with:heapDesc)

// Sub-allocate resources from heap - very fast!
let buffer = heap.newBuffer(withLength:bufferSize)
let texture = heap.newTexture(with:textureDescriptor)
// Creating Resources with MTLHeap

// Create heap descriptor, including the heap size.
let heapDesc = MTLHeapDescriptor()
heapDesc.size = heapSize

// Create the heap object ahead of time.
let heap = device.newHeap(with: heapDesc)

// Sub-allocate resources from heap - very fast!
let buffer = heap.newBuffer(withLength: bufferSize)
let texture = heap.newTexture(with: textureDescriptor)
Resource Sub-Allocation

Best practices
Resource Sub-Allocation

Best practices

Use Resource Heaps to create resources on a performance-critical path
Resource Sub-Allocation

Best practices

Use Resource Heaps to create resources on a performance-critical path
Resource Sub-Allocation

Best practices

Use Resource Heaps to create resources on a performance-critical path
Create multiple, size-bucketed heaps to reduce fragmentation
Resource Sub-Allocation

Best practices

Use Resource Heaps to create resources on a performance-critical path
Create multiple, size-bucketed heaps to reduce fragmentation
Resource Sub-Allocation

Best practices

Use Resource Heaps to create resources on a performance-critical path
Create multiple, size-bucketed heaps to reduce fragmentation
Use resource size and alignment queries to determine appropriate heap sizes
Resource Aliasing
Resource Aliasing

Multiple resources sharing memory

Resource Heaps allow multiple resources to occupy the same memory location.
Aliasing resources together reduces total memory usage.
Resource Aliasing

Multiple resources sharing memory

Resource Heaps allow multiple resources to occupy the same memory location.
Aliasing resources together reduces total memory usage.
Resource Aliasing

Multiple resources sharing memory

Resource Heaps allow multiple resources to occupy the same memory location. Aliasing resources together reduces total memory usage.
Shadow maps are produced and consumed in consecutive passes.
Shadow maps are produced and consumed in consecutive passes.

Example

Time

Shadow Map Passes  Main Pass  Post Processing Passes
Example

Shadow maps are produced and consumed in consecutive passes
Shadow maps are produced and consumed in consecutive passes.
Example

Shadow maps are produced and consumed in consecutive passes.
Post-processing uses temporary textures internally.
Shadow maps are produced and consumed in consecutive passes.

Post-processing uses temporary textures internally.
Example

Shadow maps are produced and consumed in consecutive passes
Post-processing uses temporary textures internally
Contents never used at the same time
// Create Aliasing Textures

let heap = device.newHeap(with: heapDesc)
let shadowMapA = heap.newTexture(with: shadowDesc)
let shadowMapB = heap.newTexture(with: shadowDesc)
let shadowMapC = heap.newTexture(with: shadowDesc)

// Allow heap to reassign shadow map memory to new resources
shadowMapA.makeAliasable()
shadowMapB.makeAliasable()
shadowMapC.makeAliasable()

let postProcessingD = heap.newTexture(with: postProcessingDesc)
let postProcessingE = heap.newTexture(with: postProcessingDesc)
// Create Aliasing Textures

let heap = device.newHeap(with:heapDesc)
let shadowMapA = heap.newTexture(with:shadowDesc)
let shadowMapB = heap.newTexture(with:shadowDesc)
let shadowMapC = heap.newTexture(with:shadowDesc)

// Allow heap to reassign shadow map memory to new resources
shadowMapA.makeAliasable()
shadowMapB.makeAliasable()
shadowMapC.makeAliasable()

let postProcessingD = heap.newTexture(with:postProcessingDesc)
let postProcessingE = heap.newTexture(with:postProcessingDesc)
// Create Aliasing Textures

let heap = device.newHeap(with:heapDesc)

let shadowMapA = heap.newTexture(with:shadowDesc)
let shadowMapB = heap.newTexture(with:shadowDesc)
let shadowMapC = heap.newTexture(with:shadowDesc)

// Allow heap to reassign shadow map memory to new resources
shadowMapA.makeAliasable()
shadowMapB.makeAliasable()
shadowMapC.makeAliasable()

let postProcessingD = heap.newTexture(with:postProcessingDesc)
let postProcessingE = heap.newTexture(with:postProcessingDesc)
// Create Aliasing Textures

let heap = device.newHeap(with: heapDesc)
let shadowMapA = heap.newTexture(with: shadowDesc)
let shadowMapB = heap.newTexture(with: shadowDesc)
let shadowMapC = heap.newTexture(with: shadowDesc)

// Allow heap to reassign shadow map memory to new resources
shadowMapA.makeAliasable()
shadowMapB.makeAliasable()
shadowMapC.makeAliasable()

let postProcessingD = heap.newTexture(with: postProcessingDesc)
let postProcessingE = heap.newTexture(with: postProcessingDesc)
// Create Aliasing Textures

let heap = device.newHeap(with:heapDesc)
let shadowMapA = heap.newTexture(with:shadowDesc)
let shadowMapB = heap.newTexture(with:shadowDesc)
let shadowMapC = heap.newTexture(with:shadowDesc)

// Allow heap to reassign shadow map memory to new resources
shadowMapA.makeAliasable()
shadowMapB.makeAliasable()
shadowMapC.makeAliasable()

let postProcessingD = heap.newTexture(with:postProcessingDesc)
let postProcessingE = heap.newTexture(with:postProcessingDesc)
// Create Aliasing Textures

let heap = device.newHeap(with: heapDesc)
let shadowMapA = heap.newTexture(with: shadowDesc)
let shadowMapB = heap.newTexture(with: shadowDesc)
let shadowMapC = heap.newTexture(with: shadowDesc)

// Allow heap to reassigned shadow map memory to new resources
shadowMapA.makeAliasable()
shadowMapB.makeAliasable()
shadowMapC.makeAliasable()

let postProcessingD = heap.newTexture(with: postProcessingDesc)
let postProcessingE = heap.newTexture(with: postProcessingDesc)
Resource Aliasing

Best practices
Resource Aliasing

Best practices

Calling sequence is important

• Create resources in the order they will be used in a frame
• Interleave with makeAliasable after resource contents are consumed
Resource Aliasing

Best practices

Calling sequence is important

• Create resources in the order they will be used in a frame
• Interleave with makeAliasable after resource contents are consumed

Keep dynamic resources in their own heap
Explicit Command Synchronization
Explicit Command Synchronization
Synchronizing heap resource updates
Explicit Command Synchronization

Synchronizing heap resource updates

Metal does not track command updates to heap resources

- Modification of resource contents
- Reinterpretation of aliasing resources
Explicit Command Synchronization
Synchronizing heap resource updates

Metal does not track command updates to heap resources

- Modification of resource contents
- Reinterpretation of aliasing resources
Explicit Command Synchronization

Synchronizing heap resource updates

Metal does not track command updates to heap resources

• Modification of resource contents
• Reinterpretation of aliasing resources

Tell Metal when heap resources are read or updated

• Necessary for correct results
Explicit Command Synchronization
Synchronizing heap resource updates

Metal does not track command updates to heap resources

• Modification of resource contents
• Reinterpretation of aliasing resources

Tell Metal when heap resources are read or updated

• Necessary for correct results
Explicit Command Synchronization

Synchronizing heap resource updates

Metal does not track command updates to heap resources

• Modification of resource contents
• Reinterpretation of aliasing resources

Tell Metal when heap resources are read or updated

• Necessary for correct results

Use GPU fences to explicitly communicate resource dependencies
A GPU fence is a timestamp
GPU Fences
A GPU fence is a timestamp

Commands interact with fences with two operations

- Pass A
- Pass B
GPU Fences

A GPU fence is a timestamp

Commands interact with fences with two operations

- Update the fence with new timestamp
GPU Fences

A GPU fence is a timestamp

Commands interact with fences with two operations

• Update the fence with new timestamp
• Wait for GPU to reach the current timestamp

Time

Pass A

Pass B
Parallel Command Execution

Time

- Shadow Vertex
- Shadow Fragment
- Main Vertex
- Main Fragment
- Post Process Compute
Parallel Command Execution

Metal commands are submitted in serial order
Metal commands are submitted in serial order
GPUs execute encoded commands in parallel
Metal commands are submitted in serial order.

GPUs execute encoded commands in parallel:

- Even across frames

**Parallel Command Execution**

![Diagram showing parallel command execution]

- Shadow Vertex
- Main Vertex
- Shadow Fragment
- Main Fragment
- Shadow Vertex
- Main Vertex
- Shadow Fragment
- Main Fragment
- Post Process Compute
Synchronizing Heap Updates
Incorrect results without synchronization

Two commands write to aliasing heap memory simultaneously
Synchronizing Heap Updates

Serialize commands with a fence
Synchronizing Heap Updates

Serialize commands with a fence

Use a fence to serialize access to the aliasing heap resources
Synchronizing Heap Updates

Serialize commands with a fence

Use a fence to serialize access to the aliasing heap resources

- Post Process command updates the fence
Serialize commands with a fence

Synchronizing Heap Updates

Use a fence to serialize access to the aliasing heap resources

• Post Process command updates the fence
• Shadow command waits on the fence
Synchronizing Heap Updates
Serialize commands with a fence

Use a fence to serialize access to the aliasing heap resources
• Post Process command updates the fence
• Shadow command waits on the fence
// Using Fences To Synchronize Heap Updates Across Commands

let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()
dispatchPostProcessingFilters(postProcessingEnc)
// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with:passDesc)
// Wait until compute encoder from frame A is complete
shadowEnc.wait(for:shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
shadowEnc.endEncoding()
```swift
let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()
dispatchPostProcessingFilters(postProcessingEnc)
// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with: passDesc)
// Wait until compute encoder from frame A is complete
shadowEnc.wait(for: shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
shadowEnc.endEncoding()
```
// Using Fences To Synchronize Heap Updates Across Commands

let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()
dispatchPostProcessingFilters(postProcessingEnc)

// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with: passDesc)

// Wait until compute encoder from frame A is complete
shadowEnc.wait(for: shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
shadowEnc.endEncoding()
// Using Fences To Synchronize Heap Updates Across Commands

let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()

dispatchPostProcessingFilters(postProcessingEnc)
// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with: passDesc)
// Wait until compute encoder from frame A is complete
shadowEnc.wait(for: shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
shadowEnc.endEncoding()
// Using Fences To Synchronize Heap Updates Across Commands

let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()
dispatchPostProcessingFilters(postProcessingEnc)
// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with: passDesc)
// Wait until compute encoder from frame A is complete
shadowEnc.wait(for: shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
specialEnc.endEncoding()
// Using Fences To Synchronize Heap Updates Across Commands

let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()
dispatchPostProcessingFilters(postProcessingEnc)
// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with: passDesc)
// Wait until compute encoder from frame A is complete
shadowEnc.wait(for: shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
shadowEnc.endEncoding();
// Using Fences To Synchronize Heap Updates Across Commands

let shadowPostProcessingFence = device.newFence()

// Create post-processing encoder for frame A
let postProcessingEnc = commandBufA.computeCommandEncoder()
dispatchPostProcessingFilters(postProcessingEnc)
// Tell GPU to update fence when post-processing is complete
computeEnc.update(shadowPostProcessingFence)
computeEnc.endEncoding()

// Create shadow encoder for frame B
let shadowEnc = commandBufB.renderCommandEncoder(with:passDesc)
// Wait until compute encoder from frame A is complete
shadowEnc.wait(for:shadowPostProcessingFence, beforeStages:.fragment)
renderShadows(shadowEnc);
shadowEnc.endEncoding()
Explicit Command Synchronization

Best practices

Express dependencies at appropriate granularity

• Do not track static textures
• Track groups of resources with a single fence
Resource Heaps

Summary

Create resources faster with suballocation

Use memory more efficiently with resource aliasing

Synchronize heap updates across commands with GPU Fences
Memoryless Render Targets

Same Rendering, Less Memory
Memoryless Render Targets
Textures without storage
Memoryless Render Targets
Textures without storage

Memoryless render targets are not backed by system memory
Memoryless Render Targets

Textures without storage

Memoryless render targets are not backed by system memory
Memoryless Render Targets
Textures without storage

Memoryless render targets are not backed by system memory
Use for render pass attachments that are not stored
Memoryless Render Targets
Textures without storage

Memoryless render targets are not backed by system memory
Use for render pass attachments that are not stored
Memoryless Render Targets

Textures without storage

Memoryless render targets are not backed by system memory
Use for render pass attachments that are not stored
Create texture with new storage mode

- MTLStorageModeMemoryless
Memoryless Render Targets
Textures without storage

Memoryless render targets are not backed by system memory
Use for render pass attachments that are not stored
Create texture with new storage mode
• MTLStorageModeMemoryless
iOS and tvOS only
Tile-Based Rendering

Color Attachment Texture

Depth Attachment Texture
Tile-Based Rendering

A7 and later GPUs render to fast GPU tile storage, one tile at a time.
Tile-Based Rendering

A7 and later GPUs render to fast GPU tile storage, one tile at a time.
Store actions control whether to copy results to system memory.

GPU Tile Storage

Color Attachment Texture

Depth Attachment Texture
Use Case
Depth attachments

GPU Tile Storage

Color Attachment Texture

Depth Attachment Texture
Use Case

Depth attachments

Depth attachments are required for depth testing.

GPU Tile Storage

Color Attachment Texture

Depth Attachment Texture
Use Case

Depth attachments

Depth attachments are required for depth testing.

If the depth attachment is not stored, make it memoryless.

GPU Tile Storage

Color Attachment Texture

Depth Attachment Texture
Use Case

Multisample color attachments

GPU Tile Storage

MSAA Color Attachment Texture

Resolve Color Attachment Texture
Use Case

Multisample color attachments

MSAA attachments are required for multisample rendering

GPU Tile Storage

MSAA Color Attachment Texture

Resolve Color Attachment Texture
Use Case

Multisample color attachments

MSAA attachments are required for multisample rendering

If the color attachment is resolved, make the MSAA color attachment memoryless

MSAA Color Attachment Texture

GPU Tile Storage

Resolve Color Attachment Texture
## Use Case

### Memory savings

<table>
<thead>
<tr>
<th>Texture Type</th>
<th>Potential Savings (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth Texture on iPhone 6s Plus 1920x1080</td>
<td>7.9</td>
</tr>
<tr>
<td>Depth Texture on iPad Pro (12.9-inch) 2732x2048</td>
<td>21.3</td>
</tr>
<tr>
<td>4xMSAA Color Texture on iPhone 6s Plus 1920x1080</td>
<td>31.6</td>
</tr>
<tr>
<td>4xMSAA Color Texture on iPad Pro (12.9-inch) 2732x2048</td>
<td>85.4</td>
</tr>
</tbody>
</table>
Improved Tools

Jose Enrique D’Arnaude del Castillo GPU Software Engineer
Alp Yucebilgin GPU Software Engineer
What’s New in Metal Tools

Metal System Trace
What’s New in Metal Tools

Metal System Trace
GPU Overrides
What’s New in Metal Tools

Metal System Trace
GPU Overrides
GPU Frame Debugger
Metal System Trace
Metal System Trace
Metal System Trace
Metal System Trace

iOS
Metal System Trace

iOS  tvOS
Metal System Trace

iOS  tvOS  macOS
Richer Metal Tracing
Richer Metal Tracing

Resource events
Richer Metal Tracing

Resource events
Richer Metal Tracing

Resource events
Richer Metal Tracing

Resource events

Debug groups
Richer Metal Tracing

Resource events
Debug groups
Richer Metal Tracing

Resource events
Debug groups
Multi GPU (macOS)
Richer Metal Tracing

Resource events
Debug groups
Multi GPU (macOS)
Richer Metal Tracing

- Resource events
- Debug groups
- Multi GPU (macOS)
- Scaler (iOS)
Richer Metal Tracing

Resource events
Debug groups
Multi GPU (macOS)
Scaler (iOS)
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Integration with More Instruments
Navigation
Navigation
Navigation
Navigation
Navigation
Performance Observations

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Narrative</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:14:12:430</td>
<td>GPU spent 4.76 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:12:640</td>
<td>&quot;Render Encoder 0&quot; took 6.83 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:12:680</td>
<td>GPU spent 3.31 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:12:830</td>
<td>GPU spent 4.85 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:12:950</td>
<td>&quot;Render Encoder 0&quot; took 7.73 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:13:005</td>
<td>GPU spent 2.93 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:13:095</td>
<td>GPU spent 4.90 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:13:135</td>
<td>&quot;Render Encoder 0&quot; took 7.72 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:13:190</td>
<td>GPU spent 2.87 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:13:245</td>
<td>&quot;Render Encoder 0&quot; took 7.96 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:13:290</td>
<td>GPU spent 4.77 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:13:440</td>
<td>&quot;Render Encoder 0&quot; took 7.99 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:13:510</td>
<td>GPU spent 3.04 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:13:610</td>
<td>GPU spent 4.90 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
</tbody>
</table>
Performance Observations

- GPU spent 4.75 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 6.83 ms to complete (Upper bound 724.08 µs)
- GPU spent 3.11 ms executing Vertex for "Render Encoder 0" (Upper bound 167.12 µs)
- GPU spent 4.85 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 7.73 ms to complete (Upper bound 724.08 µs)
- GPU spent 2.93 ms executing Vertex for "Render Encoder 0" (Upper bound 167.12 µs)
- GPU spent 4.80 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 7.72 ms to complete (Upper bound 724.08 µs)
- GPU spent 2.87 ms executing Vertex for "Render Encoder 0" (Upper bound 167.12 µs)
- "Render Encoder 0" took 7.96 ms to complete (Upper bound 724.08 µs)
- GPU spent 4.77 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- GPU spent 3.04 ms executing Vertex for "Render Encoder 0" (Upper bound 167.12 µs)
- GPU spent 4.80 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
Performance Observations

Timestamp | Narrative
---|---
00:14:12420 | GPU spent 4.75 ms executing Fragment for “Render Encoder 0” (Upper bound 2.20 ms)
00:14:127660 | “Render Encoder 0” took 6.83 ms to complete (Upper bound 72.08 µs)
00:14:128400 | GPU spent 3.11 ms executing Vertex for “Render Encoder 0” (Upper bound 167.12 µs)
00:14:132160 | GPU spent 4.85 ms executing Fragment for “Render Encoder 0” (Upper bound 2.20 ms)
00:14:153951 | “Render Encoder 0” took 7.73 ms to complete (Upper bound 72.08 µs)
00:14:156245 | GPU spent 2.93 ms executing Vertex for “Render Encoder 0” (Upper bound 167.12 µs)
00:14:166230 | GPU spent 4.80 ms executing Fragment for “Render Encoder 0” (Upper bound 2.20 ms)
00:14:168143 | “Render Encoder 0” took 7.72 ms to complete (Upper bound 72.08 µs)
00:14:175521 | GPU spent 2.87 ms executing Vertex for “Render Encoder 0” (Upper bound 167.12 µs)
00:14:180175 | “Render Encoder 0” took 7.96 ms to complete (Upper bound 72.08 µs)
00:14:181443 | GPU spent 4.77 ms executing Fragment for “Render Encoder 0” (Upper bound 2.20 ms)
00:14:183535 | GPU spent 3.04 ms executing Vertex for “Render Encoder 0” (Upper bound 167.12 µs)
00:14:193080 | GPU spent 4.80 ms executing Fragment for “Render Encoder 0” (Upper bound 2.20 ms)
Performance Observations

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Narrative</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:14:203.096</td>
<td>Surface was displayed for 18.54 ms</td>
</tr>
<tr>
<td>00:14:204.910</td>
<td>GPU spent 2.92 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 μs)</td>
</tr>
<tr>
<td>00:14:208.353</td>
<td>GPU spent 4.76 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:208.461</td>
<td>&quot;Render Encoder 0&quot; took 5.58 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:14:210.715</td>
<td>&quot;Render Encoder 0&quot; took 9.88 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:14:219.969</td>
<td>GPU spent 2.91 ms executing Vertex for &quot;n/a&quot; (Upper bound 79.92 μs)</td>
</tr>
<tr>
<td>00:14:223.787</td>
<td>GPU spent 4.79 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:231.432</td>
<td>&quot;Render Encoder 0&quot; took 4.11 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:14:238.832</td>
<td>&quot;Render Encoder 0&quot; took 7.07 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:14:243.844</td>
<td>GPU spent 2.97 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 μs)</td>
</tr>
<tr>
<td>00:14:247.115</td>
<td>&quot;Render Encoder 0&quot; took 6.50 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:14:247.955</td>
<td>GPU spent 4.78 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:254.718</td>
<td>GPU spent 3.68 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 μs)</td>
</tr>
</tbody>
</table>
Performance Observations
Performance Observations

- Fragment Shader took 13.31 ms to compile. Consider compiling your shaders at build time.
- GPU spent 3.31 ms executing Vertex for "nice" (Upper bound 79.92 µs)
- GPU spent 6.91 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 6.90 ms to complete (Upper bound 724.08 µs)
- Gpu spent 2.99 ms executing Vertex for "Render Encoder 0" (Upper bound 167.12 µs)
- "Render Encoder 0" took 7.80 ms to complete (Upper bound 724.08 µs)
- GPU spent 4.88 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- GPU spent 3.07 ms executing Vertex for "Render Encoder 0" (Upper bound 167.12 µs)
- GPU spent 4.88 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 8.19 ms to complete (Upper bound 724.08 µs)
- "Render Encoder 0" took 7.28 ms to complete (Upper bound 724.08 µs)
- GPU spent 4.92 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
Performance Observations

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:10:00.057</td>
<td>Fragment Shader took 11.31 ms to compile. Consider compiling your shaders at build time</td>
</tr>
<tr>
<td>00:10:00.065</td>
<td>GPU spent 3.31 ms executing Vertex for &quot;n&quot; (Upper bound 79.92 μs)</td>
</tr>
<tr>
<td>00:10:00.072</td>
<td>GPU spent 8.93 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.70 ms)</td>
</tr>
<tr>
<td>00:10:00.081</td>
<td>&quot;Render Encoder 0&quot; took 6.00 ms to complete (Upper bound 72.48 μs)</td>
</tr>
<tr>
<td>00:10:00.090</td>
<td>GPU spent 2.99 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 μs)</td>
</tr>
<tr>
<td>00:10:00.099</td>
<td>&quot;Render Encoder 0&quot; took 7.80 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:10:00.108</td>
<td>GPU spent 4.88 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.30 ms)</td>
</tr>
<tr>
<td>00:10:00.117</td>
<td>GPU spent 3.07 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 μs)</td>
</tr>
<tr>
<td>00:10:00.126</td>
<td>&quot;Render Encoder 0&quot; took 6.00 ms to complete (Upper bound 72.48 μs)</td>
</tr>
<tr>
<td>00:10:00.135</td>
<td>GPU spent 4.88 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:10:00.144</td>
<td>&quot;Render Encoder 0&quot; took 8.19 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:10:00.153</td>
<td>GPU spent 3.94 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 μs)</td>
</tr>
<tr>
<td>00:10:00.162</td>
<td>&quot;Render Encoder 0&quot; took 7.25 ms to complete (Upper bound 724.08 μs)</td>
</tr>
<tr>
<td>00:10:00.171</td>
<td>GPU spent 4.92 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
</tbody>
</table>
Performance Observations
Performance Observations

- GPU spent 2.89 ms executing Vertex for "hi" (Upper bound 79.92 ms)
- GPU spent 5.10 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 6.01 ms to complete (Upper bound 724.08 ms)
- GPU spent 2.91 ms executing Vertex for "hi" (Upper bound 79.92 ms)
- GPU spent 4.71 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 4.93 ms to complete (Upper bound 724.08 ms)
- GPU spent 2.91 ms executing Vertex for "hi" (Upper bound 79.92 ms)
- GPU spent 4.91 ms executing Fragment for "Render Encoder 0" (Upper bound 2.20 ms)
- "Render Encoder 0" took 5.73 ms to complete (Upper bound 724.08 ms)
- GPU spent 2.89 ms executing Vertex for "hi" (Upper bound 79.92 ms)
Performance Observations

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Narrative</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:14:12:430</td>
<td>GPU spent 4.75 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:12:660</td>
<td>&quot;Render Encoder 0&quot; took 6.83 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:12:480</td>
<td>GPU spent 3.11 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:14:160</td>
<td>GPU spent 4.85 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:14:551</td>
<td>&quot;Render Encoder 0&quot; took 7.73 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:16:886</td>
<td>GPU spent 4.93 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:16:804</td>
<td>GPU spent 4.80 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:18:131</td>
<td>&quot;Render Encoder 0&quot; took 7.72 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:17:821</td>
<td>GPU spent 2.87 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:18:735</td>
<td>&quot;Render Encoder 0&quot; took 7.96 ms to complete (Upper bound 724.08 µs)</td>
</tr>
<tr>
<td>00:14:18:443</td>
<td>GPU spent 4.77 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
<tr>
<td>00:14:18:834</td>
<td>GPU spent 3.04 ms executing Vertex for &quot;Render Encoder 0&quot; (Upper bound 167.12 µs)</td>
</tr>
<tr>
<td>00:14:18:560</td>
<td>GPU spent 4.80 ms executing Fragment for &quot;Render Encoder 0&quot; (Upper bound 2.20 ms)</td>
</tr>
</tbody>
</table>
Demo

GPU Frame Debugger
What’s New in Metal Tools

Metal System Trace
GPU Overrides
GPU Frame Debugger
What’s New in Metal Tools

Metal System Trace
GPU Overrides
GPU Frame Debugger
• Extended Validation
What’s New in Metal Tools

Metal System Trace
GPU Overrides
GPU Frame Debugger
• Extended Validation
• Metal Library Projects
Summary

Tessellation

Resource Heaps and Memoryless Render Targets

Improved Tools
Summary

Tessellation

Resource Heaps and Memoryless Render Targets

Improved Tools
Summary

- **Tessellation**
- Resource Heaps and Memoryless Render Targets
- **Improved Tools**
  - Function Specialization and Function Resource Read-Writes
  - Wide Color and Texture Assets
  - Additions to Metal Performance Shaders
More Information

## Related Sessions

<table>
<thead>
<tr>
<th>Session</th>
<th>Location</th>
<th>Date, Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adopting Metal, Part 1</td>
<td>Nob Hill</td>
<td>Tuesday 1:40PM</td>
</tr>
<tr>
<td>Adopting Metal, Part 2</td>
<td>Nob Hill</td>
<td>Tuesday 3:00PM</td>
</tr>
<tr>
<td>What’s New in Metal, Part 2</td>
<td>Pacific Heights</td>
<td>Wednesday 1:40PM</td>
</tr>
<tr>
<td>Advanced Metal Shader Optimization</td>
<td>Pacific Heights</td>
<td>Wednesday 3:00PM</td>
</tr>
<tr>
<td>Metal Lab</td>
<td>Graphics, Games, and Media Lab B</td>
<td>Thursday 12:00PM</td>
</tr>
</tbody>
</table>