Working with Metal—Overview

Session 603
Jeremy Sandmel
GPU Software
Metal

Dramatically reduced overhead
Unified graphics and compute
Precompiled shaders
Efficient multithreading
Designed for A7
Agenda

Background
API concepts
Shading language
Developer tools
Agenda

Background
API concepts
Shading language
Developer tools
10x more draw calls
About Draw Calls…
About Draw Calls...

Each draw call requires its own state vector
- Shaders, states, textures, render targets, etc.
About Draw Calls...

Each draw call requires its own state vector
• Shaders, states, textures, render targets, etc.

Changing state vectors can be expensive
• Translation to hardware commands
About Draw Calls…

Each draw call requires its own state vector
• Shaders, states, textures, render targets, etc.

Changing state vectors can be expensive for the CPU
• Translation to hardware commands
About Draw Calls...

Each draw call requires its own state vector
• Shaders, states, textures, render targets, etc.

Changing state vectors can be expensive for the CPU
• Translation to hardware commands

Your Application

Set Shaders
Set Textures
Set Vertex Buffers
Draw #1
Set Shaders
Set Blend
Set Depth Test
Draw #2
About Draw Calls...

Each draw call requires its own state vector
• Shaders, states, textures, render targets, etc.

Changing state vectors can be expensive for the CPU
• Translation to hardware commands
About Draw Calls…

Each draw call requires its own state vector
• Shaders, states, textures, render targets, etc.

Changing state vectors can be expensive for the CPU
• Translation to hardware commands
About Draw Calls…

Each draw call requires its own state vector
- Shaders, states, textures, render targets, etc.

Changing state vectors can be expensive for the CPU
- Translation to hardware commands

More draw calls per frame gives you
- More unique objects
- More visual variety
- More freedom for game artists and designers
Before Metal
Before Metal

Long history of GPU programming APIs

• Standards—OpenGL, OpenCL
• Domains—High level, low level, 2D, 3D
• Architectures—Platforms, devices, GPUs
Before Metal

Long history of GPU programming APIs
• Standards—OpenGL, OpenCL
• Domains—High level, low level, 2D, 3D
• Architectures—Platforms, devices, GPUs

Something was missing…
Deep Integration
Deep Integration
Deep Integration
Deep Integration

What if we took the same approach for GPU programming?
Deep Integration

What if we took the same approach for GPU programming?
Deep Integration

What if we took the same approach for GPU programming?
Metal Design
Metal Design

Thinnest possible API
Metal Design

Thinnest possible API

Modern GPU features
Metal Design

Thinnest possible API

Modern GPU features

Do expensive tasks less often
Metal Design

Thinnest possible API

Modern GPU features

Do expensive tasks less often

Predictable performance
Metal Design

Thinnest possible API

Modern GPU features

Do expensive tasks less often

Predictable performance

Explicit command submission
Metal Design

Thinnest possible API

Modern GPU features

Do expensive tasks less often

Predictable performance

Explicit command submission

Optimized for CPU behavior
Your App

- Scene Graphs
- SceneKit
  - SpriteKit
- 2D Graphics and Imaging
  - Core Animation
  - Core Image
  - Core Graphics

GPU
Your App

Scene Graphs

2D Graphics and Imaging

Standards-Based 3D Graphics

SceneKit
SpriteKit

Core Animation
Core Image
Core Graphics

OpenGL ES

GPU
So how did we do this?
Frame Times

Many games target frame rate of 30 FPS (33.3 milliseconds/frame)
Frame Times

Many games target frame rate of 30 FPS (33.3 milliseconds/frame)

<table>
<thead>
<tr>
<th>0 ms</th>
<th>33.3 ms</th>
<th>66.7 ms</th>
<th>100 ms</th>
</tr>
</thead>
</table>


Many games target frame rate of 30 FPS (33.3 milliseconds/frame)
Frame Times

Many games target frame rate of 30 FPS (33.3 milliseconds/frame)
One “Balanced” Frame

CPU work
frame $N$

GPU work
frame $N-1$
CPU Can Take More Time Than GPU

CPU work
frame N

GPU work
frame N-1
CPU Can Take More Time Than GPU

- CPU work frame N
- GPU work frame N-1
- GPU idle time

GPU utilization = 67%
More GPU Work Requires More CPU Work
CPU Time Includes Application and GPU API

- **Application**
  - Frame N

- **GPU API**
  - Frame N

- **GPU work**
  - Frame N-1

- **GPU idle time**
Targeting CPU Time Spent in GPU API

- **CPU**
  - Application \(frame \, N\)
  - GPU API \(frame \, N\)

- **GPU**
  - GPU work \(frame \, N-1\)
  - GPU idle time

Timeline:
- **0 ms** to **33.3 ms**: CPU Activity
- **33.3 ms** to **50 ms**: GPU Activity
Metal Dramatically Reduces GPU API Time

- **CPU**
  - Application: $frame\ N$
  - CPU idle time: 20 ms

- **GPU**
  - GPU work: $frame\ N-1$
  - GPU API: 33.3 ms
Metal Dramatically Reduces GPU API Time

CPU

0 ms

Application frame N

20 ms

CPU idle time

GPU

33.3 ms

GPU work frame N-1
Use CPU Time to Improve Your Game

- **CPU**
  - Application
    - frame N
  - More Physics
    - More AI
  - CPU idle time

- **GPU**
  - GPU work
    - frame N-1
Use CPU Time to Draw More Objects

CPU

Application
frame N

More Physics
More AI

GPU

GPU work
frame N-1

Up to 10x more draw calls
Use CPU Time to Draw More Objects

- 0 ms
  - Application frame N
  - More Physics
  - More AI

- 33.3 ms
  - GPU work frame N-1

Up to 10x more draw calls
Why Is GPU Programming Expensive?
Why Is GPU Programming Expensive?

State validation
• Confirming API usage is valid
• Encoding API state to hardware state
Why Is GPU Programming Expensive?

State validation
- Confirming API usage is valid
- Encoding API state to hardware state

Shader compilation
- Run-time generation of shader machine code
- Interactions between state and shaders
Why Is GPU Programming Expensive?

State validation
- Confirming API usage is valid
- Encoding API state to hardware state

Shader compilation
- Run-time generation of shader machine code
- Interactions between state and shaders

Sending work to GPU
- Managing resource residency
- Batching commands
Do Expensive Tasks Less Often

<table>
<thead>
<tr>
<th>When</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application build</td>
<td></td>
</tr>
<tr>
<td>Content loading</td>
<td></td>
</tr>
<tr>
<td>Draw time</td>
<td></td>
</tr>
</tbody>
</table>
Do Expensive Tasks Less Often

<table>
<thead>
<tr>
<th>When</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application build</td>
<td>“Never”</td>
</tr>
<tr>
<td>Content loading</td>
<td></td>
</tr>
<tr>
<td>Draw time</td>
<td></td>
</tr>
</tbody>
</table>
## Do Expensive Tasks Less Often

<table>
<thead>
<tr>
<th>When</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application build</td>
<td>“Never”</td>
</tr>
<tr>
<td>Content loading</td>
<td>Rare</td>
</tr>
<tr>
<td>Draw time</td>
<td></td>
</tr>
</tbody>
</table>
# Do Expensive Tasks Less Often

<table>
<thead>
<tr>
<th>When</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application build</td>
<td>“Never”</td>
</tr>
<tr>
<td>Content loading</td>
<td>Rare</td>
</tr>
<tr>
<td>Draw time</td>
<td>1000s of times per frame</td>
</tr>
</tbody>
</table>
## Do Expensive Tasks Less Often

<table>
<thead>
<tr>
<th>When</th>
<th>Frequency</th>
<th>Before Metal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application build</td>
<td>“Never”</td>
<td></td>
</tr>
<tr>
<td>Content loading</td>
<td>Rare</td>
<td></td>
</tr>
<tr>
<td>Draw time</td>
<td>1000s of times per frame</td>
<td>Shader compilation, State validation, Start work on GPU</td>
</tr>
</tbody>
</table>
## Do Expensive Tasks Less Often

<table>
<thead>
<tr>
<th>When</th>
<th>Frequency</th>
<th>Before Metal</th>
<th>After Metal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application build</td>
<td>“Never”</td>
<td>Shader compilation</td>
<td>Shader compilation</td>
</tr>
<tr>
<td>Content loading</td>
<td>Rare</td>
<td>State validation</td>
<td>State validation</td>
</tr>
</tbody>
</table>
| Draw time        | 1000s of times per frame | Shader compilation  
Start work on GPU | Start work on GPU                     |
Agenda

Background
API concepts
Shading language
Developer tools
## Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
</tbody>
</table>
## Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
<tr>
<td>Command Queue</td>
<td>Serial sequence of command buffers</td>
</tr>
</tbody>
</table>
## Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
<tr>
<td>Command Queue</td>
<td>Serial sequence of command buffers</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Contains GPU hardware commands</td>
</tr>
</tbody>
</table>
# Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
<tr>
<td>Command Queue</td>
<td>Serial sequence of command buffers</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Contains GPU hardware commands</td>
</tr>
<tr>
<td>Command Encoder</td>
<td>Translates API commands to GPU hardware commands</td>
</tr>
</tbody>
</table>
Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
<tr>
<td>Command Queue</td>
<td>Serial sequence of command buffers</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Contains GPU hardware commands</td>
</tr>
<tr>
<td>Command Encoder</td>
<td>Translates API commands to GPU hardware commands</td>
</tr>
<tr>
<td>State</td>
<td>Framebuffer configuration, blend, depth, samplers, etc.</td>
</tr>
</tbody>
</table>
# Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
<tr>
<td>Command Queue</td>
<td>Serial sequence of command buffers</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Contains GPU hardware commands</td>
</tr>
<tr>
<td>Command Encoder</td>
<td>Translates API commands to GPU hardware commands</td>
</tr>
<tr>
<td>State</td>
<td>Framebuffer configuration, blend, depth, samplers, etc.</td>
</tr>
<tr>
<td>Code</td>
<td>Shaders</td>
</tr>
</tbody>
</table>
# Metal Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>The GPU</td>
</tr>
<tr>
<td>Command Queue</td>
<td>Serial sequence of command buffers</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Contains GPU hardware commands</td>
</tr>
<tr>
<td>Command Encoder</td>
<td>Translates API commands to GPU hardware commands</td>
</tr>
<tr>
<td>State</td>
<td>Framebuffer configuration, blend, depth, samplers, etc.</td>
</tr>
<tr>
<td>Code</td>
<td>Shaders</td>
</tr>
<tr>
<td>Resources</td>
<td>Textures and Data Buffers (vertices, constants, etc.)</td>
</tr>
</tbody>
</table>
Changeable Source Buffers

Changeable Source Textures

Device

Command Queue

Command Buffer

Render Command Encoder

Data Buffer Resource

Texture Resource

Render Pipeline State

Depth Stencil State

Texture Resource
Command Submission Model

Command encoders convert API commands into hardware commands
Command Submission Model

Command encoders convert API commands into hardware commands.

Hardware commands stored in command buffers.
Command Submission Model

Command encoders convert API commands into hardware commands

Hardware commands stored in command buffers

Three types of command encoders
• Render, Compute, Blit
• Can interleave different types into single command buffer
• Avoids implicit expensive state save and restore operations
Command Submission Model
Explicit command buffer construction and submission

- App creates many lightweight command buffers
- App controls command buffer submission
- Metal signals app when command buffers finish execution
Command Submission Model

Explicit command buffer construction and submission
• App creates many lightweight command buffers
• App controls command buffer submission
• Metal signals app when command buffers finish execution

Command encoders generate commands immediately
• No deferred state validation
• Direct call to driver
Command Submission Model

Multithreaded command encoding
- Multiple command buffers can be encoded in parallel
- App decides execution order
- Very efficient implementation to ensure scalable performance
Resource Update Model
Resource Update Model

Designed for A7’s unified memory system

- CPU and GPU share same storage
- No implicit copies
- Automatic CPU/GPU coherency model
  - CPU and GPU observe writes at command buffer execution boundaries
  - No explicit CPU cache management required
Resource Update Model

Designed for A7’s unified memory system

- CPU and GPU share same storage
- No implicit copies
- Automatic CPU/GPU coherency model
  - CPU and GPU observe writes at command buffer execution boundaries
  - No explicit CPU cache management required

Significantly higher performance

- More synchronization responsibilities for you
Resource Update Model
Resource Update Model

Two types of resources

- Textures (formatted images)
  - Render targets, texture sources
- Data buffers (unformatted memory)
  - “a bag of bytes”
  - Vertex data, shader constants, output memory, etc.
Two types of resources

• Textures (formatted images)
  - Render targets, texture sources

• Data buffers (unformatted memory)
  - “a bag of bytes”
  - Vertex data, shader constants, output memory, etc.

Resource structure (size, levels, format) can’t be changed
• Avoids costly resource validation
• Resource contents can be changed
Resource Update Model
Resource Update Model

Updating data buffers

- Direct access to storage by CPU
- No “lock” API needed
Resource Update Model

Updating data buffers
• Direct access to storage by CPU
• No “lock” API needed

Updating textures
• Implementation private storage
• Metal provides blazing fast texture update routines
Resource Update Model

Updating data buffers
• Direct access to storage by CPU
• No “lock” API needed

Updating textures
• Implementation private storage
• Metal provides blazing fast texture update routines

GPU-accelerated and pipelined updates
• Via Blit command encoder
Resource Update Model
Resource Update Model

Can share texture storage with other textures

- Interpret as different pixel formats with same pixel size
  - eg., sRGB vs. RGB, or single 32-bit component vs. RGBA8888
Resource Update Model

Can share texture storage with other textures

• Interpret as different pixel formats with same pixel size
  - eg., sRGB vs. RGB, or single 32-bit component vs. RGBA8888

Can share texture storage with other buffers

• Assumes “row-linear” pixel data
Command Encoder Types
Command Encoder Types

Render command encoder
- Graphics rendering
Command Encoder Types

Render command encoder
- Graphics rendering

Compute command encoder
- Data parallel computations
Command Encoder Types

Render command encoder
- Graphics rendering

Compute command encoder
- Data parallel computations

Blit command encoder
- GPU-accelerated resource copy operations
Render Command Encoder
Render Command Encoder

Generates hardware commands for single rendering “pass”

- All rendering to single framebuffer object
- Specifies states for vertex and fragment stages of 3D pipeline
- Interleaves resources, state changes, and draw calls
Render Command Encoder

Generates hardware commands for single rendering “pass”
- All rendering to single framebuffer object
- Specifies states for vertex and fragment stages of 3D pipeline
- Interleaves resources, state changes, and draw calls

No “draw time” compilation
- App controls when all significant compilation and state validation occurs
# Render State Objects

<table>
<thead>
<tr>
<th>State Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepthStencil</td>
<td>DepthStencil comparison functions and write masks</td>
</tr>
<tr>
<td>Sampler</td>
<td>Filter states, addressing modes, LOD state</td>
</tr>
<tr>
<td>Render Pipeline</td>
<td>“Everything else”&lt;br&gt;Vertex and pixel shader functions&lt;br&gt;Vertex data layout&lt;br&gt;Multisample state&lt;br&gt;Blend state&lt;br&gt;Color write masks</td>
</tr>
</tbody>
</table>
States affecting compilation can’t be changed after object creation
Inexpensive states can be changed

<table>
<thead>
<tr>
<th>Changeable</th>
<th>Render States</th>
</tr>
</thead>
<tbody>
<tr>
<td>![X]</td>
<td>Vertex and fragment shaders</td>
</tr>
<tr>
<td>![X]</td>
<td># of render targets, pixel format, color write masks</td>
</tr>
<tr>
<td>![X]</td>
<td>Multisample state</td>
</tr>
<tr>
<td>![X]</td>
<td>Blend state</td>
</tr>
<tr>
<td>![X]</td>
<td>DepthStencil state and write masks</td>
</tr>
<tr>
<td>![✓]</td>
<td>Specification of buffers, textures, samplers</td>
</tr>
<tr>
<td>![✓]</td>
<td>Cull mode and facing orientation</td>
</tr>
<tr>
<td>![✓]</td>
<td>Depth clipping and depth bias</td>
</tr>
<tr>
<td>![✓]</td>
<td>Polygon mode</td>
</tr>
<tr>
<td>![✓]</td>
<td>Viewport and scissor</td>
</tr>
<tr>
<td>![✓]</td>
<td>Occlusion queries</td>
</tr>
</tbody>
</table>
Framebuffer Loads and Stores

Framebuffer configuration designed for optimal behavior on A7 GPU

- Tile-based deferred-mode renderer
Framebuffer Loads and Stores

Framebuffer configuration designed for optimal behavior on A7 GPU
• Tile-based deferred-mode renderer

Explicit control of framebuffer tile cache “Load and Store” operations
• Load at start of render pass
• Store at end of render pass
Framebuffer Loads and Stores

Framebuffer configuration designed for optimal behavior on A7 GPU
• Tile-based deferred-mode renderer

Explicit control of framebuffer tile cache “Load and Store” operations
• Load at start of render pass
• Store at end of render pass

App choses per render target
• Load—Don’t care, load, clear
• Store—Don’t care, store, multisample resolve
Before Metal

Framebuffer loads and stores

One Frame

Color Framebuffer → Render Pass #1 → Color Framebuffer → Render Pass #2 → Color Framebuffer
Before Metal

Framebuffer loads and stores

One Frame

Color Framebuffer → Read → Render Pass #1 → Write → Color Framebuffer → Read → Render Pass #2 → Write → Color Framebuffer
Before Metal
Framebuffer loads and stores

One Frame

Color Framebuffer

Depth Framebuffer

Render Pass #1

Color Framebuffer

Depth Framebuffer

Render Pass #2

Color Framebuffer

Depth Framebuffer
Before Metal

Framebuffer loads and stores

One Frame

Color Framebuffer

Depth Framebuffer

Read

Write

Read

Write

Read

Write

Color Framebuffer

Depth Framebuffer

Framebuffer Memory Bandwidth

Color: 2 reads + 2 writes

Depth: 2 reads + 2 writes
Metal

Framebuffer loads and stores

One Frame

Color Framebuffer

Depth Framebuffer

Render Pass #1

Color Framebuffer

Depth Framebuffer

Render Pass #2

Color Framebuffer

Depth Framebuffer
Metal

Framebuffer loads and stores

One Frame

Color Framebuffer → Don't Care → Store → Color Framebuffer → Write → Render Pass #1

Depth Framebuffer → Read → Render Pass #1

Color Framebuffer → Read → Render Pass #2

Depth Framebuffer → Write → Render Pass #2

Color Framebuffer → Write → Depth Framebuffer → Read → Color Framebuffer
Metal

Framebuffer loads and stores

One Frame

Color Framebuffer

Depth Framebuffer

Don't Care

Store

Write

Read

Load

Store

Write

Color Framebuffer

Depth Framebuffer

Render Pass #1

Render Pass #2

Write

Read

Write

Read

Color Framebuffer

Depth Framebuffer

Depth Framebuffer
Metal

Framebuffer loads and stores

One Frame

- **Color Framebuffer**
  - **Depth Framebuffer**
  - **Render Pass #1**
    - **Don't Care**
    - **Clear**
    - **Don't Care**
  - **Store**
    - **Write**
    - **Color Framebuffer**
    - **Load**
      - **Read**
      - **Depth Framebuffer**
      - **Write**
      - **Render Pass #2**
      - **Store**
        - **Write**
        - **Color Framebuffer**
        - **Write**
        - **Depth Framebuffer**
Metal

Framebuffer loads and stores

One Frame

- **Color Framebuffer**
- **Depth Framebuffer**

**Render Pass #1**
- **Don't Care**
- **Store**
- **Write**
- **Clear**

**Color Framebuffer**

**Render Pass #2**
- **Load**
- **Read**
- **Store**
- **Write**

**Color Framebuffer**

**Depth Framebuffer**

- **Don't Care**
- **Clear**

**Depth Framebuffer**
Dramatic Reduction in Memory Traffic

One Frame

Framebuffer Memory Bandwidth

Color: 1 reads + 2 writes
Depth: 0 reads + 0 writes
Compute Command Encoder

Familiar compute run-time and memory model
• Textures and data buffers
• Local and global memory
• Local atomics
• Barriers
• Memory loads and stores
• User-specified workgroup dimensions
Compute Command Encoder
Compute Command Encoder

Fully integrated with graphics

- Unified API, shading language and developer tools
- Efficiently interleaves Compute commands with Render and Blit commands
Compute Command Encoder

Fully integrated with graphics

• Unified API, shading language and developer tools
• Efficiently interleaves Compute commands with Render and Blit commands

No “execution-time” compilation

• App controls when all significant compilation and validation occurs
## Compute Command Encoder

<table>
<thead>
<tr>
<th>State Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute State</td>
<td>Compute functions, workgroup config</td>
</tr>
<tr>
<td>Sampler</td>
<td>Filter states, addressing modes, LOD state</td>
</tr>
</tbody>
</table>
Blit Command Encoder

Enables asynchronous copies

• In parallel with graphics and compute operations
Blit Command Encoder

Enables asynchronous copies
- In parallel with graphics and compute operations

Texture uploads
- Copy to/from other texture or data buffer
- Accelerated mipmap generation
Blit Command Encoder

Enables asynchronous copies
- In parallel with graphics and compute operations

Texture uploads
- Copy to/from other texture or data buffer
- Accelerated mipmap generation

Data Buffer updates
- Copy to/from another data buffer or texture
- Fill with constant values
Agenda

Background
API concepts
Shading language
Developer tools
Shading Language
Shading Language

Unified shading language for graphics and compute processing
Shading Language

Unified shading language for graphics and compute processing

Based on C++11
- Static subset
- Built from LLVM and clang
Shading Language

Unified shading language for graphics and compute processing

Based on C++11
- Static subset
- Built from LLVM and clang

Additions
- GPU hardware features (texture sampling, rasterization, compute operations, etc.)
- Function overloading and templates
Shading Language
Shading Language

Data types for graphics and compute features
- Scalar, vector and matrix types
- Samplers and textures
Shading Language

Data types for graphics and compute features
- Scalar, vector and matrix types
- Samplers and textures

“Attributes”
- Function arguments
- Sampling and interpolation qualifiers
- Per-instance inputs, outputs, and built-in graphics variables
- Programmable blending
Shading Language
Shading Language

Multiple shaders per source file
Shading Language

Multiple shaders per source file

Metal shaders built by Xcode compiler into Metal library files
  • Library contains archive of Metal shaders
  • With run-time APIs
    - Load a Metal library
    - Finalize compilation to GPU machine code
Shading Language

Multiple shaders per source file

Metal shaders built by Xcode compiler into Metal library files
• Library contains archive of Metal shaders
• With run-time APIs
  - Load a Metal library
  - Finalize compilation to GPU machine code

Metal includes standard library for graphics and compute functions
Argument Tables

Textures, buffers, and samplers passed as arguments to functions
- Vertex, fragment, compute shaders
Argument Tables

Textures, buffers, and samplers passed as arguments to functions
• Vertex, fragment, compute shaders

Each command encoder includes set of “argument tables”
• One table per type (texture, buffer, sampler)
Argument Tables

Textures, buffers, and samplers passed as arguments to functions
- Vertex, fragment, compute shaders

Each command encoder includes set of “argument tables”
- One table per type (texture, buffer, sampler)

Metal API and shading language use table index to reference arguments
**Argument Tables**

<table>
<thead>
<tr>
<th>Texture Index</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Texture A</td>
</tr>
<tr>
<td>1</td>
<td>Texture B</td>
</tr>
<tr>
<td>..</td>
<td>Texture ..</td>
</tr>
<tr>
<td>n</td>
<td>Texture Z</td>
</tr>
</tbody>
</table>
Argument Tables

<table>
<thead>
<tr>
<th>Texture Index</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Texture A</td>
</tr>
<tr>
<td>1</td>
<td>Texture B</td>
</tr>
<tr>
<td>..</td>
<td>Texture ..</td>
</tr>
<tr>
<td>n</td>
<td>Texture Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buffer Index</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Buffer A</td>
</tr>
<tr>
<td>1</td>
<td>Buffer B</td>
</tr>
<tr>
<td>..</td>
<td>Buffer ..</td>
</tr>
<tr>
<td>n</td>
<td>Buffer Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sampler Index</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sampler A</td>
</tr>
<tr>
<td>1</td>
<td>Sampler B</td>
</tr>
<tr>
<td>..</td>
<td>Sampler ..</td>
</tr>
<tr>
<td>n</td>
<td>Sampler Z</td>
</tr>
</tbody>
</table>
Argument Tables

vertex VertexOutput
smoothTriangleVertex(constant float4 *pos_data [[ buffer(0) ]],
   constant float2 *uv_data [[ buffer(1) ]],
   uint vid [[ vertex_id ]])
{
    VertexOutput out;
    out.pos = pos_data[vid];
    out.uv = uv_data[vid];
    return out;
}

fragment float4
smoothTriangleFragment(VertexOutput in [[ stage_in ]],
   texture2d<float> tex [[ texture(1) ]])
{
    return tex.sample(s, in.uv);
}

Shader Code
### Argument Tables

**Vertex Shader**

```cpp
vertex VertexOutput
smoothTriangleVertex(constant float4 *pos_data [[ buffer(0) ]],
                     constant float2 *uv_data  [[ buffer(1) ]],
                     uint vid  [[ vertex_id ]])
{
    VertexOutput out;
    out.pos = pos_data[vid];
    out.uv = uv_data[vid];
    return out;
}
```

**Fragment Shader**

```cpp
fragment float4
smoothTriangleFragment(VertexOutput in  [[ stage_in ]],
                        texture2d<float> tex  [[ texture(1) ]])
{
    return tex.sample(s, in.uv);
}
```

---

### Command Encoder

**Texture Index**

<table>
<thead>
<tr>
<th>ID</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Texture A</td>
</tr>
<tr>
<td>1</td>
<td>Texture B</td>
</tr>
<tr>
<td>..</td>
<td>Texture ..</td>
</tr>
<tr>
<td>n</td>
<td>Texture Z</td>
</tr>
</tbody>
</table>

**Buffer Index**

<table>
<thead>
<tr>
<th>ID</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Buffer A</td>
</tr>
<tr>
<td>1</td>
<td>Buffer B</td>
</tr>
<tr>
<td>..</td>
<td>Buffer ..</td>
</tr>
<tr>
<td>n</td>
<td>Buffer Z</td>
</tr>
</tbody>
</table>

**Sampler Index**

<table>
<thead>
<tr>
<th>ID</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sampler A</td>
</tr>
<tr>
<td>1</td>
<td>Sampler B</td>
</tr>
<tr>
<td>..</td>
<td>Sampler ..</td>
</tr>
<tr>
<td>n</td>
<td>Sampler Z</td>
</tr>
</tbody>
</table>
vertex VertexOutput
smoothTriangleVertex(constant float4 *pos_data [[ buffer(0) ]],
        constant float2 *uv_data  [[ buffer(1) ]],
        uint vid [[ vertex_id ]])
{
    VertexOutput out;
    out.pos = pos_data[vid];
    out.uv = uv_data[vid];
    return out;
}

fragment float4
smoothTriangleFragment(VertexOutput in [[ stage_in ]],
        texture2d<float> tex [[ texture(1) ]])
{
    return tex.sample(s, in.uv);
}
vertex VertexOutput
smoothTriangleVertex(constant float4 *pos_data [[ buffer(0) ]],
        constant float2 *uv_data [[ buffer(1) ]],
        uint vid [[ vertex_id ]])
{
    VertexOutput out;
    out.pos = pos_data[vid];
    out.uv = uv_data[vid];
    return out;
}

fragment float4
smoothTriangleFragment(VertexOutput in [[ stage_in ]],
        texture2d<float> tex [[ texture(1) ]])
{
    return tex.sample(s, in.uv);
}
Agenda

Background
API concepts
Shading language
Developer tools
Metal Shader Compiler Process
Metal Shader Compiler Process

Metal shader sources compiled to libraries at application build time
• No need to ship source code with application
• Shading language errors reported at build time
Metal Shader Compiler Process

Metal shader sources compiled to libraries at application build time
• No need to ship source code with application
• Shading language errors reported at build time

Metal libraries compiled to device code at state object creation time
• No draw time compilation
• Device code cached after compilation
Metal Shader Compiler Process

Metal shader sources compiled to libraries at application build time
- No need to ship source code with application
- Shading language errors reported at build time

Metal libraries compiled to device code at state object creation time
- No draw time compilation
- Device code cached after compilation

There is also a run-time shader compiler
- No draw time compilation
- For best performance, use offline compiler
Metal Shader Compiler Process

In Xcode

Metal Shader Compiler

Your Application
Metal Shader Compiler Process

In Xcode

Metal Shader

Metal Shader Compiler

Your Application
Metal Shader Compiler Process

In Xcode

Metal Shader

Metal Shader Compiler

Metal Library

Your Application
Metal Shader Compiler Process

In Xcode

Your Application

Metal Library
Metal Shader Compiler Process

In Xcode

Metal Shader

Metal Shader Compiler

At application run-time

Your Application

Metal Library
Metal Shader Compiler Process

In Xcode

Metal Shader Compiler

Metal Shader

At application run-time

Your Application

Metal Library

State
Vertex Shader
Fragment Shader

Pipeline Object Creation
Metal Shader Compiler Process

In Xcode:
- Metal Shader Compiler
- Metal Shader

At application run-time:
- Your Application
  - Metal Library
  - Pipeline Object Creation
    - State
    - Vertex Shader
    - Fragment Shader
Metal Shader Compiler Process

In Xcode
- Metal Shader Compiler
- Metal Shader

At application run-time
- Your Application
  - Metal Library
  - State
  - Vertex Shader
  - Fragment Shader
  - Pipeline Object Creation
  - Shader Cache
Metal Shader Compiler Process

In Xcode

Metal Shader Compiler

Metal Shader

At application run-time

Your Application

Metal Library

State

Vertex Shader

Fragment Shader

Pipeline Object Creation

Shader Cache

Metal Device Compiler
Metal Shader Compiler Process

In Xcode

At application run-time

Your Application

Metal Library

Pipeline Object Creation

State
Vertex Shader
Fragment Shader

Shader Cache
Metal Device Compiler

GPU
Metal Tools Fully Integrated into Xcode

Visual frame debugger
API trace and navigation
Shader edit-and-continue
Rich source code editing (including shaders)
Graphics and compute state inspection
Shader compiler
Debug mode for Metal framework
Framebuffer View
fragment FragOutput gBufferFrag(VertexOutput in) {
    constant float4 clearColor_gbuffer0 [buffer(0)],
        texture2d<float> bump_texture [texture(0)], sampler bump_sampler [sampler(0)],
        texture2d<float> diffuse_texture [texture(1)], sampler diffuse_sampler [sampler(1)],
        depth2d<float> shadow_texture [texture(2)]
    {
        float3 tangent_normal = bump_texture.sample(bump_sampler, 0.5% in_v_texcoord.xy).xyz * 2.0 - 1.0;
        float4 diffuse_color = diffuse_texture.sample(diffuse_sampler in_v_texcoord.xy);
        float3 world_normal = in_v_normal * tangent_normal.z + in_v_tangent * tangent_normal.x - in_v_bitangent * tangent_normal.y;
        float scale = rsqrt(dot(world_normal, world_normal)) * 14.6 % 0.5;

        const sampler shadow_sampler(coord::normalized, filter::linear, address::clamp_to_edge, compare_func::less);
        float shadowCoeff = shadow_texture.sample_compare(shadow_sampler, in_v_shadowcoord.xy, in_v_shadowcoord.z);

        FragOutput output;
        world_normal = world_normal * scale + 0.5;
        output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
    }
}
Performance Report

fragment FragOutput gBufferFrag(VertexOutput in) {
    constant float4 &clear_color_gbuffer0 = (buffer(0)),
    texture2d<float> bump_texture
        [texture(0)], sampler bump_sampler
        [sampler(0)],
    texture2d<float> diffuse_texture
        [texture(1)], sampler diffuse_sampler
        [sampler(1)],
    depth2d<float> shadow_texture
        [texture(2)]

    float3 tangent_normal = bump_texture.sample(bump_sampler, 0.5%
in_v_texcoord.xy) * 2.0 - 1.0;
    float4 diffuse_color = diffuse_texture.sample(diffuse_sampler
        in_v_texcoord.xy);
    float3 world_normal = in_v_normal * tangent_normal.z + in.v_4 in_v_bitangent *
        tangent_normal.y;
    float scale = rsqrt(dot(world_normal, world_normal)) * 14.6
        0.5;
    const sampler shadow_sampler(coord::normalized, filter::linear,
        address:: clamp_to_edge, compare_func::less);
    float shadowCoeff = shadow_texture
        sample_compare(shadow_sampler, in_v_shadowcoord.xy, in.v_shadowcoord.z);
    FragOutput output;
    world_normal = world_normal * scale + 0.5;
    output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
}
float4 fragment_output gbuffer3 [[ color(3) ]]; // light

fragment_fragOutput gBufferFrag(VertexOutput in [[stage_in]], constant float4 &clear_color_gbuffer3 [[buffer(0)]],
texture2d<float> bump_texture [[texture(0)]], sampler bump_sampler [[sampler(0)]]],
texture2d<float> diffuse_texture [[texture(1)]], sampler diffuse_sampler [[sampler(1)]]],
depth2d<float> shadow_texture [[texture(2)]]
{
    float3 tangent_normal = bump_texture.sample(bump_sampler, in.v_texcoord.xy).xyz * 2.0 - 1.0;
    float4 diffuse_color = diffuse_texture.sample(diffuse_sampler, in.v_texcoord.xy);
    float3 world_normal = in.v_normal * tangent_normal.z + in.v_tangent * tangent_normal.x - in.v_bitangent * tangent_normal.y;
    float scale = rsqrt(dot(world_normal, world_normal)) * 0.5;
    const sampler shadow_sampler = shadow_texture.sample_compare(shadow_sampler, in.
v_shadowcoord.xy, in.v_shadowcoord.z);
    float shadowCoeff = shadow_texture.sample_compare(shadow_sampler, in.
v_shadowcoord.xy, in.v_shadowcoord.z);

    FragOutput output;
    world_normal = world_normal * scale + 0.5;
    output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
    output.fragment_output_gbuffer0.a = shadowCoeff;
    output.fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
    output.fragment_output_gbuffer2 = float4(in.v_lineardepth, in.v_lineardepth, in.v_lineardepth, in.v_lineardepth);
    output.fragment_output_gbuffer3 = clear_color_gbuffer3;

    return output;
}
float4 fragment_output_gBuffer3 [[ color(3) ]]; // light

fragment_FragOutput gBufferFrag(VertexOutput in [[stage_in]], constant float4 &clear_color_gBuffer3 [[buffer(0)]],
texture2d<float> bump_texture [[texture(0)]], sampler bump_sampler [[sampler(0)]],
texture2d<float> diffuse_texture [[texture(1)]], sampler diffuse_sampler [[sampler(1)]],
depth2d<float> shadow_texture [[texture(2)]]
{
    float3 tangent_normal = bump_texture.sample(bump_sampler, in.v_texcoord.xy).xyz * 2.0 - 1.0;
    float4 diffuse_color = diffuse_texture.sample(diffuse_sampler, in.v_texcoord.xy);
    float3 world_normal = in.v_normal * tangent_normal.z + in.v_tangent * tangent_normal.x - in.v_bitangent * tangent_normal.y;
    float scale = rsqrt(dot(world_normal, world_normal)) * 0.5;
    const sampler shadow_sampler = texture2DNormalized(filter:linear, address:clamp_to_edge, compare_func:less);
    float shadowCoeff = shadow_texture.sample_compare(shadow_sampler, in.v_shadowcoord.xy, in.v_shadowcoord.z);

    FragOutput output;
    world_normal = world_normal * scale + 0.5;
    output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
    output.fragment_output_gbuffer0.a = shadowCoeff;
    output.fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
    output.fragment_output_gbuffer2 = float4(in.v_lineardepth, in.v_lineardepth, in.v_lineardepth, in.v_lineardepth);
    output.fragment_output_gbuffer3 = clear_color_gBuffer3;

    return output;
}
Shader Profiler

```swift
float4 fragment_output_gbuffer3 [color(3)]; // light

fragment_fragOutput_gBufferFrag(VertextOutput in [[stage_in]], constant float4 &clear_color_gbuffer3 [[buffer(0)]],
texture2d<float> bump_texture [[texture(0)]], sampler bump_sampler [[sampler(0)]],
texture2d<float> diffuse_texture [[texture(1)]], sampler diffuse_sampler [[sampler(1)]]),
depth2d<float> shadow_texture [[texture(2)]]
{
    float3 tangent_normal = bump_texture.sample(bump_sampler, in.v_texcoord.xy).xyz * 2.0 - 1.0; 0.5%
    float4 diffuse_color = diffuse_texture.sample(diffuse_sampler, in.v_texcoord.xy); 4.1%
    float3 world_normal = in.v_normal * tangent_normal.z + in.v_tangent * tangent_normal.x - in.v_bitangent * tangent_normal.y;
    float scale = rsqrt(dot(world_normal, world_normal)) * 0.5; 14.8%
    const sampler shadow_sampler(coord: normalized, filter: linear, address: clamp_to_edge, compare_func: less);
    float shadowCoeff = shadow_texture.sample(shadow_sampler, in.
v_shadowcoord.xy, in.v_shadowcoord.z); 43.8%

    FragOutput output;
    world_normal = world_normal * scale + 0.5;
    output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
    output.fragment_output_gbuffer0.a = shadowCoeff;
    output.fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
    output.fragment_output_gbuffer2 = float4(in.v_linardepth, in.v_linardepth, in.v_linardepth, in.v_linardepth);
    output.fragment_output_gbuffer3 = clear_color_gbuffer3;

    return output;
}
```
float4 fragment_output_gbuffer3 [[ color(3) ]]; // light

Fragment FragOutput gBufferFrag(VerteXOutput in [[stage_in]], constant float4 &clear_color_gbuffer3 [[buffer(0)]],
texture2d<float> bump_texture [[texture(0)]], sampler bump_sampler [[sampler(0)]]),
texture2d<float> diffuse_texture [[texture(1)]], sampler diffuse_sampler [[sampler(1)]]);

depth2d<float> shadow_texture [[texture(2)]]
{
  float3 tangent_normal = bump_texture.sample(bump_sampler, in.v_texcoord.xy).xyz * 2.0 - 1.0;
  float4 diffuse_color = diffuse_texture.sample(diffuse_sampler, in.v_texcoord.xy);
  float3 world_normal = in.v_normal * tangent_normal.z + in_v_tangent * tangent_normal.x - in_v_bitangent * tangent_normal.y;
  float scale = rsqrt(dot(world_normal, world_normal)) * 0.5;
  const sampler shadow_sampler(coord::normalized, filter::linear, address::clamp_to_edge, compare_func::less);
  float shadowCoeff = shadow_texture.sample_compare(shadowSampler, in.v_shadowcoord.xy, in.v_shadowcoord.z);

  FragOutput output;
  world_normal = world_normal * scale + 0.5;

  output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
  output.fragment_output_gbuffer0.a = shadowCoeff;
  output.fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
  output.fragment_output_gbuffer2 = float4(in.v_lineardepth, in.v_lineardepth, in.v_lineardepth, in.v_lineardepth);
  output.fragment_output_gbuffer3 = clear_color_gbuffer3;

  return output;
}
float4 fragment_output_gbuffer3 [[color(3)]]; // light

fragment_fragOutput gBufferFrag(VertextOutput in [[stage_in]], constant float4 &clear_color_gbuffer3 [[buffer(0)]],
texture2d<float> bump_texture [[texture(0)]], sampler bump_sampler [[sampler(0)]],
texture2d<float> diffuse_texture [[texture(1)]], sampler diffuse_sampler [[sampler(1)]],
depth2d<float> shadow_texture [[texture(2)]]
{
  float3 tangent_normal = bump_texture.sampler(bump_sampler, in.v_texcoord.xy).xyz * 2.0 - 1.0;
  float4 diffuse_color = diffuse_texture.sampler(diffuse_sampler, in.v_texcoord.xy);
  float3 world_normal = in.v_normal * tangent_normal.z + in.v_tangent * tangent_normal.x - in.v_bitangent * tangent_normal.y;

  float scale = rSqrt(dot(world_normal, world_normal)) * 0.5;

  const sampler shadow_sampler = (coord::normalized, filter::linear, address::clamp_to_edge, compare_func::less);

  float shadowCoeff = shadow_texture.sample_compare(shadow_sampler, in.v_shadowcoord.xy, in.v_shadowcoord.z);

  float4 fragment_output_gbuffer0.rgb = diffuse_color.rgb;
  fragment_output_gbuffer0.a = shadowCoeff;
  fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
  fragment_output_gbuffer2 = float4(in.v_lineardepth, in.v_lineardepth, in.v_lineardepth, in.v_lineardepth);
  fragment_output_gbuffer3 = clear_color_gbuffer3;

  return fragment_output;
}
float4 fragment_output_gbuffer3 [[ color(3) ]]; // light

fragment_FragOutput gBufferFrag(VerteXOutput in [[stage_in]], constant float4 &clear_color_gbuffer3 [[buffer(0)]],
texture2d<float> bump_texture [[texture(0)]], sampler bump_sampler [[sampler(0)]],
texture2d<float> diffuse_texture [[texture(1)]], sampler diffuse_sampler [[sampler(1)]],
depth2d<float> shadow_texture [[texture(2)]]
{
    float3 tangent_normal = bump_texture.sample(bump_sampler, in.v_texcoord.xy).xyz * 2.0 - 1.0;
    float4 diffuse_color = diffuse_texture.sample(diffuse_sampler, in.v_texcoord.xy);
    float3 world_normal = in.v_normal * tangent_normal.z + in.v_tangent * tangent_normal.x - in.v_bitangent * tangent_normal.y;
    float scale = rsqrt(dot(world_normal, world_normal)) * 0.5;
    float3 shadowCoeff = shadow_texture.sample_compare(shadow_sampler, in.v_shadowcoord.xy, in.v_shadowcoord.z);
    FragOutput output;
    world_normal = world_normal * scale + 0.5;
    output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
    output.fragment_output_gbuffer0.a = shadowCoeff;
    output.fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
    output.fragment_output_gbuffer2 = float4(in.v_lineardepth, in.v_lineardepth, in.v_lineardepth, in.v_lineardepth);
    output.fragment_output_gbuffer3 = clear_color_gbuffer3;

    return output;
}
struct FragOutput {
    float4 fragment_output_gbuffer0 [[ color(0) ]]; // albedo/final
    float4 fragment_output_gbuffer1 [[ color(1) ]]; // normal
    float4 fragment_output_gbuffer2 [[ color(2) ]]; // depth
    float4 fragment_output_gbuffer3 [[ color(3) ]]; // light
};

fragment FragOutput gBufferFrag(VertexOutput in [[ stage_in ]], constant float4 &clear_color_gbuffer3 [[ buffer(0) ]],
texture2d<float> bump_texture [[ texture(0) ]], sampler bump_sampler [[ sampler(0) ]],
texture2d<float> diffuse_texture [[ texture(1) ]], sampler diffuse_sampler [[ sampler(1) ]],
depth2d<float> shadow_texture [[ texture(2) ]]) {

    float3 tangent_normal = bump_texture.sample(bump_sampler, in.v_texcoord.xyz) * 2.0 - 1.0;
    float4 diffuse_color = diffuse_texture.sample(diffuse_sampler, in.v_texcoord.xy);
    float4 world_normal = in.v_normal * tangent_normal.z + in.v_tangent * tangent_normal.x - in.v_bitangent *
    tangent_normal.y;
    float scale = rsqrt(dot(world_normal, world_normal)) * 0.5;
    const sampler shadow_sampler(coord::normalized, filter::linear, address::clamp_to_edge, compare_func::less);

    const char kSampleXCount = 4.1;
    const char kSampleYCount = 4;
    float sum = 0;
    for (char dy = -kSampleYCount/2; dy < kSampleYCount/2; ++dy) {
        for (char dx = -kSampleXCount/2; dx < kSampleXCount/2; ++dx) {
            sum += shadow_texture.sample_compare(shadow_sampler, in.v_shadowcoord.xy, in.v_shadowcoord.z, char2(dx, dy));
        }
    }
    float shadowCoeff = sum / ((float)kSampleXCount * kSampleYCount);

    FragOutput output;
    world_normal = world_normal * scale + 0.5;
    output.fragment_output_gbuffer0.rgb = diffuse_color.rgb;
    output.fragment_output_gbuffer0.a = shadowCoeff;
    output.fragment_output_gbuffer1 = float4(world_normal.x, world_normal.y, world_normal.z, 1.0);
    output.fragment_output_gbuffer2 = float4(in.v_lineardepth, in.v_lineardepth, in.v_lineardepth, in.v_lineardepth);
    output.fragment_output_gbuffer3 = clear_color_gbuffer3;
}
Demo

“The Collectables” on Metal

Sean Tracey
Crytek
Summary
Summary

Low overhead, high performance GPU programming API
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Designed for A7 and iOS
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Designed for A7 and iOS
Streamlined for modern GPU features
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Designed for A7 and iOS
Streamlined for modern GPU features
Fine-grained control
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Designed for A7 and iOS
Streamlined for modern GPU features
Fine-grained control
Precompiled shaders
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Designed for A7 and iOS
Streamlined for modern GPU features
Fine-grained control
Precompiled shaders
Fantastic developer tools
Summary

Low overhead, high performance GPU programming API
Up to 10x more draw calls
Designed for A7 and iOS
Streamlined for modern GPU features
Fine-grained control
Precompiled shaders
Fantastic developer tools
Enables entirely new class of games
More Information

Filip Iliescu
Graphics and Games Technologies Evangelist
filiescu@apple.com

Allan Schaffer
Graphics and Games Technologies Evangelist
aschaffer@apple.com

Documentation
http://developer.apple.com

Apple Developer Forums
http://devforums.apple.com
## Related Sessions

<table>
<thead>
<tr>
<th>Event</th>
<th>Location</th>
<th>Date/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with Metal—Fundamentals</td>
<td>Pacific Heights</td>
<td>Wednesday 10:15AM</td>
</tr>
<tr>
<td>Working with Metal—Advanced</td>
<td>Pacific Heights</td>
<td>Wednesday 11:30AM</td>
</tr>
</tbody>
</table>
Labs

- Metal Lab Graphics and Games Lab A Wednesday 2:00PM
- Metal Lab Graphics and Games Lab B Thursday 10:15AM