Metal for Game Developers

Session 607

Brian Ross, GPU Software (macOS)
Michael Imbrogno, GPU Software (iOS)
Agenda
Agenda

Harnessing Parallelism
Agenda

Harnessing Parallelism

Taking Explicit Control
Agenda

Harnessing Parallelism

Taking Explicit Control

Building GPU-Driven Pipelines
Agenda

Harnessing Parallelism
Taking Explicit Control
Building GPU-Driven Pipelines
Optimizing for the A11 Bionic GPU
Agenda

Harnessing Parallelism
Taking Explicit Control
Building GPU-Driven Pipelines
Optimizing for the A11 Bionic GPU
Bringing Fortnite to Metal
Harnessing Parallelism
Harnessing Parallelism
Harnessing Parallelism

Scalable multi-threaded encoding is key
Harnessing Parallelism

Scalable multi-threaded encoding is key

Metal makes multi-threaded CPU command generation easy and fast
Harnessing Parallelism

Scalable multi-threaded encoding is key

Metal makes multi-threaded CPU command generation easy and fast

Metal automatically parallelizes GPU tasks
Single-Thread Rendering
Single-Thread Rendering

- Command buffer
- Render
- Blit (Copy)
- G-buffer
- Deferred shading
- FX
- Post process

Thread 0

- Resource updates
- Physics
- Particles
- G-buffer
- Shadow
- Shadow
- Deferred shading
- FX
- Post process

- Compute
- Resource updates
Single-Thread Rendering

- Thread 0
  - Resource updates
  - Physics
  - Particles
  - G-buffer
  - Shadow
  - Shadow
  - Deferred shading
  - FX
  - Post process

- Increased latency

Colors:
- Gold: Compute
- Blue: Render
- Green: Blit (Copy)
Multi-Threaded Rendering

MTLCommandBuffer

Thread 0
- Resource updates
- Physics
- Particles
- G-buffer
- Shadow
- Shadow
- Deferred shading
- FX
- Post process

Thread 1

Thread 2

Thread 3
Multi-Threaded Rendering

MTLCommandBuffer

Thread 0
- Resource updates
- Physics
- Particles
- G-buffer
- Shadow
- Shadow
- Deferred shading
- FX
- Post process

Thread 1

Thread 2

Thread 3
Multi-Threaded Rendering

MTLCommandBuffer

Thread 0
- Resource updates
- Physics
- Particles

Thread 1
- G-buffer

Thread 2
- Shadow
- Shadow
- Deferred shading

Thread 3
- FX
- Post process
Multi-Threaded Rendering
MTLCommandBuffer

Thread 0
- Resource updates
- Physics
- Particles

Thread 1
- G-buffer

Thread 2
- Shadow
- Shadow
- Deferred shading

Thread 3
- FX
- Post process
// Create multiple command buffers
let commandBuffer1 = commandQueue.makeCommandBuffer()!
let commandBuffer2 = commandQueue.makeCommandBuffer()!

// Enqueue to define desired GPU execution order
commandBuffer1.enqueue()
commandBuffer2.enqueue()

// Dispatch encoding on separate threads
queue.async(group: group) {
    encodeGBufferPass( commandBuffer2, ... )
    commandBuffer2.commit()
}
queue.async(group: group) {
    encodeDeferredShadingPass( commandBuffer1, ... )
    commandBuffer1.commit()
}
// Create multiple command buffers
let commandBuffer1 = commandQueue.makeCommandBuffer()!
let commandBuffer2 = commandQueue.makeCommandBuffer()!

// Enqueue to define desired GPU execution order
commandBuffer1.enqueue()
commandBuffer2.enqueue()

// Dispatch encoding on separate threads
queue.async(group: group) {
    encodeGBufferPass( commandBuffer2, ... )
    commandBuffer2.commit()
}
queue.async(group: group) {
    encodeDeferredShadingPass( commandBuffer1, ... )
    commandBuffer1.commit()
}
// Create multiple command buffers
let commandBuffer1 = commandQueue.makeCommandBuffer()!
let commandBuffer2 = commandQueue.makeCommandBuffer()!

// Enqueue to define desired GPU execution order
commandBuffer1.enqueue()
commandBuffer2.enqueue()

// Dispatch encoding on separate threads
queue.async(group: group) {
    encodeGBufferPass( commandBuffer2, ... )
    commandBuffer2.commit()
}
queue.async(group: group) {
    encodeDeferredShadingPass( commandBuffer1, ... )
    commandBuffer1.commit()
}
// Create multiple command buffers
let commandBuffer1 = commandQueue.makeCommandBuffer()!
let commandBuffer2 = commandQueue.makeCommandBuffer()!

// Enqueue to define desired GPU execution order
commandBuffer1.enqueue()
commandBuffer2.enqueue()

// Dispatch encoding on separate threads
queue.async(group: group) {
    encodeGBufferPass( commandBuffer2, ... )
    commandBuffer2.commit()
}
queue.async(group: group) {
    encodeDeferredShadingPass( commandBuffer1, ... )
    commandBuffer1.commit()
}
Multi-Threaded Rendering

MTLCommandBuffer

Thread 0: Resource updates, Physics, Particles
Thread 1: G-buffer
Thread 2: Shadow, Shadow, Deferred shading
Thread 3: FX, Post process
Multi-Threaded Rendering
Without MTLParallelRenderCommandEncoder

Thread 0
- Resource updates
- Physics
- Particles

Thread 1
- G-buffer

Thread 2
- Shadow
- Shadow
- Deferred shading

Thread 3
- FX
- Post process
Multi-Threaded Rendering
With MTLParallelRenderCommandEncoder

Thread 0: Resource updates, Physics, Particles
Thread 1: G-buffer
Thread 2: G-buffer
Thread 3: Shadow, Shadow, Deferred shading
Thread 4: FX, Post process
// Create parallel encoder and subordinate render command encoder objects
let parallelRenderEncoder = commandBuffer.makeParallelRenderCommandEncoder(renderPassDesc)!
let renderEncoder1 = parallelRenderEncoder.makeRenderCommandEncoder()!
let renderEncoder2 = parallelRenderEncoder.makeRenderCommandEncoder()!

// Encode different portions of G-Buffer pass (in any order) on separate threads
queue.async(group: group) {
    encodeGBufferTerrain(renderEncoder2)
}
queue.async(group: group) {
    encodeGBufferObjects(renderEncoder1)
}

// Notify when encoding complete and end the parallel encoder
group.notify(queue: queue) {
    parallelRenderEncoder.endEncoding()
}
// Create parallel encoder and subordinate render command encoder objects
let parallelRenderEncoder = commandBuffer.makeParallelRenderCommandEncoder(renderPassDesc)!
let renderEncoder1 = parallelRenderEncoder.makeRenderCommandEncoder()!
let renderEncoder2 = parallelRenderEncoder.makeRenderCommandEncoder()!

// Encode different portions of G-Buffer pass (in any order) on separate threads
queue.async(group: group) {
    encodeGBufferTerrain(renderEncoder2)
}
queue.async(group: group) {
    encodeGBufferObjects(renderEncoder1)
}

// Notify when encoding complete and end the parallel encoder
group.notify(queue: queue) {
    parallelRenderEncoder.endEncoding()
}
// Create parallel encoder and subordinate render command encoder objects
let parallelRenderEncoder = commandBuffer.makeParallelRenderCommandEncoder(renderPassDesc)!
let renderEncoder1 = parallelRenderEncoder.makeRenderCommandEncoder()!
let renderEncoder2 = parallelRenderEncoder.makeRenderCommandEncoder()!

// Encode different portions of G-Buffer pass (in any order) on separate threads
queue.async(group: group) {
    encodeGBufferTerrain(renderEncoder2)
}
queue.async(group: group) {
    encodeGBufferObjects(renderEncoder1)
}

// Notify when encoding complete and end the parallel encoder
group.notify(queue: queue) {
    parallelRenderEncoder.endEncoding()
}
// Create parallel encoder and subordinate render command encoder objects
let parallelRenderEncoder = commandBuffer.makeParallelRenderCommandEncoder(renderPassDesc)!
let renderEncoder1 = parallelRenderEncoder.makeRenderCommandEncoder()!
let renderEncoder2 = parallelRenderEncoder.makeRenderCommandEncoder()!

// Encode different portions of G-Buffer pass (in any order) on separate threads
queue.async(group: group) {
    encodeGBufferTerrain(renderEncoder2)
}
queue.async(group: group) {
    encodeGBufferObjects(renderEncoder1)
}

// Notify when encoding complete and end the parallel encoder
group.notify(queue: queue) {
    parallelRenderEncoder.endEncoding()
}
GPU Parallelism
Asynchronous compute and render

![GPU Parallelism Diagram](image-url)
GPU Parallelism
Asynchronous compute and render

<table>
<thead>
<tr>
<th>GPU</th>
<th>Resource updates</th>
<th>Physics</th>
<th>Particles</th>
<th>G-buffer</th>
<th>Shadow</th>
<th>Shadow</th>
<th>Deferred shading</th>
<th>FX</th>
<th>Post process</th>
</tr>
</thead>
</table>

Write: Particle

Read: Particle
GPU Parallelism

Asynchronous compute and render

- Resource updates
- Physics
- Particles
- G-buffer
- Shadow
- Shadow
- Deferred shading
- FX
- Post process

Read: GBuffer
Write: GBuffer
Write: Particle
Read: Particle
GPU Parallelism
Asynchronous compute and render

- Resource updates
- Physics
- Particles
- G-buffer
- Shadow
- Shadow
- Deferred shading
- FX
- Post process

Write: Particle
Write: GBuffer
Read: GBuffer
Read: Particle
GPU Parallelism
Asynchronous compute and render

- Resource updates
- Physics
- G-buffer
- Shadow
- Shadow
- Deferred shading
- Post process
- Particles
- Write: GBuffer
- Read: GBuffer
- FX
- Write: Particle
- Read: Particle
GPU Parallelism
Asynchronous compute and render

- Resource updates
- Physics
  - G-buffer
  - Shadow
  - Shadow
  - Deferred shading
  - Post process
- Particles
  - Write: GBuffer
  - Read: GBuffer
  - FX
  - Write: Particle
  - Read: Particle
GPU Parallelism
Asynchronous compute and render

- Resource updates
- Physics
- G-buffer
- Shadow
- Shadow
- Deferred shading
- Post process
- Particles
- FX
Taking Explicit Control
Taking Explicit Control
Taking Explicit Control

Metal offers more direct approaches for even less overhead
Taking Explicit Control

Metal offers more direct approaches for even less overhead

Disable Metal’s automatic reference counting
Taking Explicit Control

Metal offers more direct approaches for even less overhead

Disable Metal’s automatic reference counting

Allocate resources cheaply using heaps
Taking Explicit Control

Metal offers more direct approaches for even less overhead

Disable Metal’s automatic reference counting

Allocate resources cheaply using heaps

Control GPU parallelism with fences and events
Resource Heaps
MTLHeap

Control time of memory allocation

Fast reallocation and aliasing of resources

Cheaper resource binding

Simple API
Resource Heaps
MTLHeap

Memory Allocation for A
Texture A

Memory Allocation for B
Texture B

Memory Allocation for C
Texture C
Resource Heaps
MTLHeap

Memory Allocation for A
Texture A

Memory Allocation for B
Texture B

Memory Allocation for C
Texture C

Metal Resource Heap
Texture A
Texture B
Texture C
Resource Heaps
MTLHeap

- Memory Allocation for A
  Texture A

- Memory Allocation for B
  Texture B

- Memory Allocation for C
  Texture C

Metal Resource Heap
- Texture A
- Texture B
- Texture C
Resource Heaps

MTLHeap

Memory Allocation for A
Texture A

Memory Allocation for B
Texture B

Metal Resource Heap
Texture A
Texture B
Texture C
Resource Heaps

MTLHeap

- Memory Allocation for A: Texture A
- Memory Allocation for B: Texture B
- Memory Allocation for D: Texture D

Metal Resource Heap:
- Texture A
- Texture B
Resource Heaps

MTLHeap

- Memory Allocation for A: Texture A
- Memory Allocation for B: Texture B
- Memory Allocation for D: Texture D

Metal Resource Heap

- Texture A
- Texture B
- Texture D
Resource Heaps

MTLHeap

Memory Allocation for A
Texture A

Memory Allocation for B
Texture B

Memory Allocation for D
Texture D

Metal Resource Heap
Texture D
Resource Heaps
Without dependency tracking

![Diagram showing resource heaps with stages such as Resource updates, Physics, Particles, G-buffer, Shadow, Shadow, Deferred shading, FX, Post process, with write and read points for Particle.](https://via.placeholder.com/150)
MTLFence and MTLEvent
Explicit execution order
MTLFence and MTLEvent
Explicit execution order

- Resource updates
- Physics
- Particles
- G-buffer
- Shadow
- Shadow
- Deferred shading
- FX
- Post process

Update Fence: GBuffer
Wait for Fence: GBuffer
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
// G-Buffer pass creates a temporary target to render output
let temporaryRenderTarget = heap.makeTexture(gBufferTextureDescriptor)

// Render into our temporary render target and return encoder for later use
let renderEncoder = renderSceneIntoRenderTarget(temporaryRenderTarget)!

// Update the G-Buffer fence now that we encoded our scene
renderEncoder.updateFence(gBufferFence after: .fragment)

// Deferred Shading pass uses render target and return encoder for later use
let computeEncoder = computeDeferredShading(temporaryRenderTarget)

// Deferred shading pass needs to wait for fence
computeEncoder.waitForFence(gBufferFence)

// After this point we can re-use the render target in a post-process pass
temporaryRenderTarget.makeAliasable()
let depthOfFieldTarget = heap.makeTexture(depthOfFieldTextureDescriptor)
computeDepthOfField(depthOfFieldTarget)
Building GPU-Driven Pipelines
Building GPU-Driven Pipelines
Building GPU-Driven Pipelines

Games are moving more and more logic onto GPU
• Efficient processing of large datasets
• Growing scene graph complexity
Building GPU-Driven Pipelines

Games are moving more and more logic onto GPU

• Efficient processing of large datasets
• Growing scene graph complexity

Metal 2 enables you to move entire render loop to the GPU

• Argument Buffers—offload parameter management
• Indirect Command Buffers—offload rendering loop
**Argument Buffers**

**Shader example**

```cpp
struct Material
{
    float            roughness;
    float            intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler          textureSampler;
};

kernel void my_kernel(constant Material &material [[buffer(0)]], ...)
{
    ...
}
```
struct Material
{
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
};

kernel void my_kernel(constant Material &material [[buffer(0)]], ...)
{
    ...
}

Argument Buffers
Shader example
struct Material
{
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
};

kernel void my_kernel(device Material &material [[buffer(0)]], ...)
{
    material.surfaceTexture = getDetailedTexture(...);
}
struct Material
{
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
};

kernel void my_kernel(device Material &material [[buffer(0)]], ...)
{
    material.surfaceTexture = getDetailedTexture(...);
}
Argument Buffers
Arrays of materials

```cpp
struct Material {
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
};

fragment float4 my_shader(device Material *material [[buffer(0)]], ...) {
    return calculateMaterial( material[instanceID] );
}
```
Argument Buffers

Arrays of materials

```c
struct Material
{
    float            roughness;
    float            intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler          textureSampler;
};

fragment float4 my_shader(device Material *material [[buffer(0)]], ...)
{
    return calculateMaterial( material[instanceID] );
}
```
Argument Buffers
Arrays of materials

```
struct Material {
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
};

fragment float4 my_shader(device Material *material [[buffer(0)]], ...) {
    return calculateMaterial( material[instanceID] );
}
```
Argument Buffers
New arguments

```cpp
struct Material {
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
    render_pipeline_state pipelineState;
    command_buffer commandBuffer;
};
```
Argument Buffers

New arguments

```c
struct Material {
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    sampler textureSampler;
    render_pipeline_state pipelineState;
    command_buffer commandBuffer;
};
```
Indirect Command Buffers (ICB)
Indirect Command Buffers (ICB)

Allows GPU to build draw calls
• Massively parallel generation of commands
Indirect Command Buffers (ICB)

- Allows GPU to build draw calls
  - Massively parallel generation of commands

- Reuse ICB in multiple frames
  - Modify contents
Indirect Command Buffers (ICB)

Allows GPU to build draw calls
  • Massively parallel generation of commands

Reuse ICB in multiple frames
  • Modify contents

Remove expensive CPU and GPU synchronization
Game Rendering Loop

Traditional running on CPU

Select visible objects, determine level-of-detail
Game Rendering Loop
Traditional running on CPU

Select visible objects, determine level-of-detail
Game Rendering Loop

Traditional running on CPU

- Select visible objects, determine level-of-detail
- Set frame properties, Encode draw commands

System Memory

Command Buffer

Parameters

CPU

GPU

Render pass
Game Rendering Loop
Mixed CPU and GPU work

- **CPU**
  - Set frame properties, Encode draw commands

- **GPU**
  - Select visible objects, determine level-of-detail
  - Render pass

- **System Memory**
  - Command Buffer
  - Parameters
Game Rendering Loop
Mixed CPU and GPU

CPU

Set frame properties, Encode draw commands

GPU

Select visible objects, determine level-of-detail

STALL

System Memory

Command Buffer
Parameters

Render pass
Game Rendering Loop

GPU Driven rendering loop

CPU

Select visible objects, determine level-of-detail

Set frame properties, Encode draw commands

Render pass

System Memory

Indirect Command Buffer

Parameters
Game Rendering Loop

CPU

Select visible objects, determine level-of-detail

Set frame properties, Encode draw commands

Render pass

GPU

System Memory

Indirect Command Buffer

Parameters
GPU Rendering Loop

Visible objects

Object 0
Object 1
...
Object N
GPU Rendering Loop

Visible objects

Object 0
Object 1
...
Object N

Object 1 LOD 0 LOD 1 LOD 2
Position Position Position
Material Args Material Args Material Args
VertexBuffer VertexBuffer VertexBuffer
IndexBuffer IndexBuffer IndexBuffer
GPU Rendering Loop

Visible objects

Object 0
Object 1
...
Object N

Object 1  Lod 0  Lod 1  Lod 2

Position  Position  Position
Material Args  Material Args  Material Args
Vertex Buffer  Vertex Buffer  Vertex Buffer
Index Buffer  Index Buffer  Index Buffer
GPU Rendering Loop

Visible objects

Object 0
Object 1
...
Object N

Kernel

Thread 0
Thread 1
...
Thread N

Object 1

Lod 0
Position
Material Args
Vertex Buffer
Index Buffer

Lod 1
Position
Material Args
Vertex Buffer
Index Buffer

Lod 2
Position
Material Args
Vertex Buffer
Index Buffer
GPU Rendering Loop

Visible objects
- Object 0
- Object 1
- ...
- Object N

Kernel
- Thread 0
- Thread 1
- ...
- Thread N

Indirect Command Buffer
- cmd0
- cmd1
- ...
- cmdN

Object 1
- Lod 0
  - Position
  - Material Args
  - Vertex Buffer
  - Index Buffer
- Lod 1
  - Position
  - Material Args
  - Vertex Buffer
  - Index Buffer
- Lod 2
  - Position
  - Material Args
  - Vertex Buffer
  - Index Buffer
GPU Rendering Loop

Visible objects
- Object 0
- Object 1
- ...
- Object N

Kernel
- Thread 0
- Thread 1
- ...
- Thread N

Indirect Command Buffer
- cmd0
- cmd1
- ...
- cmdN

Object 1
- LOD 0 (Position, Material Args, Vertex Buffer, Index Buffer)
- LOD 1 (Position, Material Args, Vertex Buffer, Index Buffer)
- LOD 2 (Position, Material Args, Vertex Buffer, Index Buffer)
GPU Rendering Loop

Visible objects:
- Object 0
- Object 1
- ... Object N

Kernel:
- Thread 0
- Thread 1
- ... Thread N

Indirect Command Buffer:
- cmd0: setPipeline, setBuffer, setBuffer, setBuffer, setBuffer, draw
- cmd1: setPipeline, setBuffer, draw
- ... cmdN: setPipeline, setBuffer, setBuffer, draw patches

Visible objects:
- Object 1
- Lod 0
- Position
- Material Args
- Vertex Buffer
- Index Buffer
- Lod 1
- Position
- Material Args
- Vertex Buffer
- Index Buffer
- Lod 2
- Position
- Material Args
- Vertex Buffer
- Index Buffer
Encoding Drawcall Example

```c
kernel void encodeDrawcall(device Objects *obj [[buffer(0)]],
                           device command_buffer icb [[buffer(1)]],
                           uint index [[thread_index_in_grid]])
{
    // Select the render command index based on thread ID
    render_command cmd( icb, index );

    // Encode PSO and buffer arguments for each object
    cmd.set_pipeline_state( obj[index].shader );
    cmd.set_vertex_buffer( obj[index].geometry, 0);
    cmd.set_fragment_buffer( obj[index].material, 0);

    // Issue the drawcall
    cmd.draw(triangle, obj[index].vertexStart, obj[index].vertexCount,
             obj[index].instanceCount, obj[index].baseInstance);
}
```
Encoding Drawcall Example

```c
kernel void encodeDrawcall(device Objects *obj [[buffer(0)]],
    device command_buffer icb [[buffer(1)]],
    uint index [[thread_index_in_grid]])
{
    // Select the render command index based on thread ID
    render_command cmd( icb, index );

    // Encode PSO and buffer arguments for each object
    cmd.set_pipeline_state( obj[index].shader );
    cmd.set_vertex_buffer( obj[index].geometry, 0);
    cmd.set_fragment_buffer( obj[index].material, 0);

    // Issue the drawcall
    cmd.draw(triangle, obj[index].vertexStart, obj[index].vertexCount,
             obj[index].instanceCount, obj[index].baseInstance);
}
```
Encoding Drawcall Example

```c
kernel void encodeDrawcall(device Objects *obj [[buffer(0)]],
    device command_buffer icb [[buffer(1)]],
    uint index [[thread_index_in_grid]])
{
    // Select the render command index based on thread ID
    render_command cmd( icb, index );

    // Encode PSO and buffer arguments for each object
    cmd.set_pipeline_state( obj[index].shader );
    cmd.set_vertex_buffer( obj[index].geometry, 0);
    cmd.set_fragment_buffer( obj[index].material, 0);

    // Issue the drawcall
    cmd.draw(triangle, obj[index].vertexStart, obj[index].vertexCount,
             obj[index].instanceCount, obj[index].baseInstance);
}
```
Encoding Drawcall Example

```c
kernel void encodeDrawcall(device Objects *obj [[buffer(0)]],
    device command_buffer icb [[buffer(1)]],
    uint index [[thread_index_in_grid]])
{
    // Select the render command index based on thread ID
    render_command cmd( icb, index );

    // Encode PSO and buffer arguments for each object
    cmd.set_pipeline_state( obj[index].shader );
    cmd.set_vertex_buffer( obj[index].geometry, 0);
    cmd.set_fragment_buffer( obj[index].material, 0);

    // Issue the drawcall
    cmd.draw(triangle, obj[index].vertexStart, obj[index].vertexCount,
             obj[index].instanceCount, obj[index].baseInstance);
}
```
Executing ICB Example

// Create an Indirect Command Buffer
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw, .patches]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount: 1000, options…)!

// Dispatch compute kernel to encode Indirect Command Buffer
computeEncoder.dispatchThreadGroups(…)

// Optimize the Indirect Command Buffer across specified range
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(10..<900))

// Execute with indirect range buffer
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer: rangeBuffer, …)
Executing ICB Example

// Create an Indirect Command Buffer
let desc = MLIIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw, .patches]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount: 1000, options...)!

// Dispatch compute kernel to encode Indirect Command Buffer
computeEncoder.dispatchThreadGroups(...)

// Optimize the Indirect Command Buffer across specified range
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(10..<900))

// Execute with indirect range buffer
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer: rangeBuffer, ...)
Executing ICB Example

// Create an Indirect Command Buffer
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw, .patches]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:1000, options…)!

// Dispatch compute kernel to encode Indirect Command Buffer
computeEncoder.dispatchThreadGroups(…)

// Optimize the Indirect Command Buffer across specified range
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(10..<900))

// Execute with indirect range buffer
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer:rangeBuffer, …)
Executing ICB Example

```swift
// Create an Indirect Command Buffer
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw, .patches]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount: 1000, options...)

// Dispatch compute kernel to encode Indirect Command Buffer
computeEncoder.dispatchThreadGroups(...)

// Optimize the Indirect Command Buffer across specified range
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(10..<900))

// Execute with indirect range buffer
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer: rangeBuffer, ...)
```
Executing ICB Example

// Create an Indirect Command Buffer
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw, .patches]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount: 1000, options...)

// Dispatch compute kernel to encode Indirect Command Buffer
computeEncoder.dispatchThreadGroups(...)

// Optimize the Indirect Command Buffer across specified range
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(10..<900))

// Execute with indirect range buffer
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer: rangeBuffer, ...)
Optimizing for the A11 Bionic GPU
Tile-Based Deferred Rendering (TBDR)

High performance, low power
Tile-Based Deferred Rendering (TBDR)

High performance, low power

Tightly integrated with Metal
Tile-Based Deferred Rendering (TBDR)

High performance, low power

Tightly integrated with Metal

A11 Bionic takes TBDR to the next level
Metal 2 Features on A11
Metal 2 Features on A11

| Programmable Blending | Combine render passes using attachments |
# Metal 2 Features on A11

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmable Blending</td>
<td>Combine render passes using attachments</td>
</tr>
<tr>
<td>Imageblocks</td>
<td>Control pixel layouts from the shading language</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Programmable Blending</td>
<td>Combine render passes using attachments</td>
</tr>
<tr>
<td>Imageblocks</td>
<td>Control pixel layouts from the shading language</td>
</tr>
<tr>
<td>Tile Shading</td>
<td>Interleave graphics and compute in the render pass</td>
</tr>
</tbody>
</table>
## Metal 2 Features on A11

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmable Blending</td>
<td>Combine render passes using attachments</td>
</tr>
<tr>
<td>Imageblocks</td>
<td>Control pixel layouts from the shading language</td>
</tr>
<tr>
<td>Tile Shading</td>
<td>Interleave graphics and compute in the render pass</td>
</tr>
<tr>
<td>Persistent Threadgroup Memory</td>
<td>Share data across draws and dispatches</td>
</tr>
</tbody>
</table>
## Metal 2 Features on A11

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmable Blending</td>
<td>Combine render passes using attachments</td>
</tr>
<tr>
<td>Imageblocks</td>
<td>Control pixel layouts from the shading language</td>
</tr>
<tr>
<td>Tile Shading</td>
<td>Interleave graphics and compute in the render pass</td>
</tr>
<tr>
<td>Persistent Threadgroup Memory</td>
<td>Share data across draws and dispatches</td>
</tr>
<tr>
<td>Multi-Sample Color Coverage Control</td>
<td>Efficient custom resolves within the render pass</td>
</tr>
</tbody>
</table>
Programmable Blending

Metal provides read-write access to pixels in tile memory

• Implement custom blend operations
Programmable Blending

Metal provides read-write access to pixels in tile memory
  • Implement custom blend operations

Combine passes that read-write the same pixels
  • Eliminates system memory bandwidth between passes
  • Optimizes deferred shading
Deferred Shading
Without Programmable Blending

Pass 1: G-buffer Fill
Pass 2: Lighting

Multiple Render Passes
System Memory
Normal Albedo Roughness
Final Image
Deferred Shading
Without Programmable Blending

- Multiple Render Passes
  - Pass 1: G-buffer Fill
  - Pass 2: Lighting

- System Memory
  - Normal Albedo Roughness
  - Final Image
Deferred Shading
With Programmable Blending

Single Render Pass
- G-buffer Fill
- Lighting

Tile Memory

System Memory
Deferred Shading
With Programmable Blending and Memoryless Render Targets

Single Render Pass

G-buffer Fill

Lighting

Tile Memory

System Memory
typedef struct {
    float4 albedo [[color(0)]];  
    float4 normal [[color(1)]];  
    float depth [[color(2)]];    
    float4 accum [[color(3)]];  
} GBuffer;

fragment GBuffer ApplyLight(VertexOutput in [[stage_in]], GBuffer gBuffers, …) {
    float3 p = ReconstructViewPosition(gBuffers.depth);
    half3 n = gBuffers.normal.rgb;
    half4 lighting = CalculateLighting(p, n, gBuffers.albedo, …);
    GBuffer output = gBuffers;
    output.accum += lighting;
    return output;
}
typedef struct {
    float4 albedo [[color(0)]];
    float4 normal [[color(1)]];
    float depth [[color(2)]];
    float4 accum [[color(3)]];
} GBuffer;

fragment GBuffer ApplyLight(VertexOutput in [[stage_in]], GBuffer gBuffers, …)
{
    float3 p = ReconstructViewPosition(gBuffers.depth);
    half3  n = gBuffers.normal.rgb;
    half4 lighting = CalculateLighting(p, n, gBuffers.albedo, …);
    GBuffer output = gBuffers;
    output.accum += lighting;
    return output;
}
typedef struct {
    float4 albedo [[color(0)]];  
    float4 normal [[color(1)]];  
    float depth  [[color(2)]];  
    float4 accum  [[color(3)]];  
} GBuffer;

fragment GBuffer ApplyLight(VertexOutput in [[stage_in]], GBuffer gBuffers, …) {  
    float3 p = ReconstructViewPosition(gBuffers.depth);  
    half3  n = gBuffers.normal.rgb;  
    half4 lighting = CalculateLighting(p, n, gBuffers.albedo, …);  
    GBuffer output = gBuffers;  
    output.accum += lighting;  
    return output;  
}
typedef struct {
    float4 albedo [[color(0)]];
    float4 normal [[color(1)]];
    float depth [[color(2)]];
    float4 accum [[color(3)]];
} GBuffer;

fragment GBuffer ApplyLight(VertexOutput in [[stage_in]], GBuffer gBuffers, ...)
{
    float3 p = ReconstructViewPosition(gBuffers.depth);
    half3 n = gBuffers.normal.rgb;
    half4 lighting = CalculateLighting(p, n, gBuffers.albedo, ...);
    GBuffer output = gBuffers;
    output.accum += lighting;
    return output;
}
Imageblocks

Flexible per-pixel storage in tile memory
• Layout declared in shading language
Imageblocks

Flexible per-pixel storage in tile memory
• Layout declared in shading language

```cpp
struct GBufferPixel {
  rgba8unorm<half3> albedo;
  rg11b10f<half3> normal;
  float depth;
};

struct MultiLayerAlphaBlendPixel {
  rgba8unorm<half4> color[4];
  half depths[4];
};
```
Imageblocks

Flexible per-pixel storage in tile memory
• Layout declared in shading language
Imageblocks

Flexible per-pixel storage in tile memory
• Layout declared in shading language

Change pixel layout within a pass
• Merge render passes with different layouts

```c
struct GBufferPixel {
    rgba8unorm<half3> albedo;
    rg11b10f<half3>   normal;
    float             depth;
};

struct MultiLayerAlphaBlendPixel {
    rgba8unorm<half4> color[4];
    half              depths[4];
};
```
Deferred Shading
With Imageblocks

Single Render Pass
G-buffer Fill → Lighting

Tile Memory

System Memory
Deferred Shading and Multi-Layer Alpha Blending
With changing Imageblocks

Single Render Pass
- G-buffer Fill
- Lighting
- Multi-Layer Alpha Blending

Tile Memory

System Memory
Deferred Shading and Multi-Layer Alpha Blending
With changing Imageblocks

Single Render Pass
G-buffer Fill

Lighting

Multi-Layer Alpha Blending

Tile Memory

System Memory
Tile Shading

New programmable stage in the render pass
• Dispatch a configurable threadgroup per tile
Tile Shading

New programmable stage in the render pass
• Dispatch a configurable threadgroup per tile

Interleave render and compute processing
• Read rendering results directly from tile memory
Tiled Forward Shading
Mixes render and compute
Tiled Forward Shading
Mixes render and compute

Render and Compute Passes

Depth Pass

Depth Bounds + Light Culling

Forward Pass

System Memory

Depth

Depth bounds

Culled light list

Final Image
Tiled Forward Shading
Combine passes with Tile Shading

Single Render Pass

Tile Memory

System Memory

Depth
Depth Bounds + Light Culling
Forward

Depth bounds
Culled light list
Tiled Forward Shading
Combine passes with Tile Shading

Single Render Pass

Depth

Depth Bounds + Light Culling

Forward

Tile Memory

Depth bounds

Culled light list

System Memory
Persistent Threadgroup Memory

Threadgroup memory available to render passes
• Available to fragment and tile shaders
• Buffer contents persist for the lifetime of tile
Persistent Threadgroup Memory

Threadgroup memory available to render passes
• Available to fragment and tile shaders
• Buffer contents persist for the lifetime of tile

Share data across pixels
• Communicate tile-scoped data between draws and tile dispatches
Tiled Forward Shading

Using Tile Shading

- Single Render Pass
  - Depth
  - Depth Bounds + Light Culling
  - Forward

- Tile Memory

- System Memory
  - Depth bounds
  - Culled light list
Tiled Forward Shading
Using Tile Shading and Persistent Threadgroup Memory

Single Render Pass
- Depth
- Depth Bounds + Light Culling
- Forward

Tile Memory
- Depth bounds
- Culled light list

System Memory
kernel void CullLights(imageblock<FragData> all_frag_data,
    device Light *all_lights [[buffer(0)]],
    threadgroup float2 depth_bounds [[threadgroup(0)]],
    threadgroup uint32_t &active_light_mask [[threadgroup(1)]], …) {
    active_light_mask = 0;
    for (int i = 0; i < MAX_LIGHTS; ++i) {
        if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, …)) {
            active_light_mask = (1u << i);
        }
    }
}

fragment float4 ForwardShade(VertexInputs stage_in [[stage_in]],
    FragData frag_data [[imageblock_data]],
    device Light *all_lights [[buffer(0)]],
    threadgroup uint32_t &active_light_mask [[threadgroup(1)]]) {
    …
}
kernel void CullLights(imageblock<FragData> all_frag_data,
                      device Light *all_lights [[buffer(0)]],
                      threadgroup float2 depth_bounds [[threadgroup(0)]],
                      threadgroup uint32_t &active_light_mask [[threadgroup(1)]], ...) {

    active_light_mask = 0;
    for (int i = 0; i < MAX_LIGHTS; ++i) {
        if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, ...) {
            active_light_mask = (1u << i);
        }
    }
}

fragment float4 ForwardShade(VertexInputs stage_in [[stage_in]],
                              FragData frag_data [[imageblock_data]],
                              device Light *all_lights [[buffer(0)]],
                              threadgroup uint32_t &active_light_mask [[threadgroup(1)]]) {

    ...
}
kernel void CullLights(imageblock<FragData> all_frag_data,  
   device Light *all_lights [[buffer(0)]],  
   threadgroup float2 depth_bounds [[threadgroup(0)]],  
   threadgroup uint32_t &active_light_mask [[threadgroup(1)]], ...) {  
   active_light_mask = 0;  
   for (int i = 0; i < MAX_LIGHTS; ++i) {  
      if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, ...) {  
         active_light_mask = (1u << i);  
      }  
   }  
}  

fragment float4 ForwardShade(VertexInputs stage_in [[stage_in]],  
   FragData frag_data [[imageblock_data]],  
   device Light *all_lights [[buffer(0)]],  
   threadgroup uint32_t &active_light_mask [[threadgroup(1)]]) {  
   ...  
}
kernel void CullLights(imageblock<FragData> all_frag_data,
   device Light *all_lights [[buffer(0)]],
   threadgroup float2 depth_bounds [[threadgroup(0)]],
   threadgroup uint32_t &active_light_mask [[threadgroup(1)]], …) {

   active_light_mask = 0;
   for (int i = 0; i < MAX_LIGHTS; ++i) {
      if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, …)) {
         active_light_mask = (1u << i);
      }
   }
}

fragment float4 ForwardShade(VertexInputs stage_in [[stage_in]],
   FragData frag_data [[imageblock_data]],
   device Light *all_lights [[buffer(0)]],
   threadgroup uint32_t &active_light_mask [[threadgroup(1)]]) {

   ...
}
Deferred Shading and Multi-Layer Alpha Blending

With changing Imageblocks requires Tile Shading

- G-buffer Fill
- Lighting
- Multi-Layer Alpha Blending
Deferred Shading and Multi-Layer Alpha Blending
With changing Imageblocks requires Tile Shading
Multi-Sample Color Coverage Control
Multi-Sample Color Coverage Control

MSAA is efficient on A-series GPUs
• Samples stored in tile memory for fast blending and resolves
Multi-Sample Color Coverage Control

MSAA is efficient on A-series GPUs
• Samples stored in tile memory for fast blending and resolves

MSAA is more efficient on A11
• Tracks unique colors in a pixel
Multi-Sample Color Coverage Control

MSAA is efficient on A-series GPUs
- Samples stored in tile memory for fast blending and resolves

MSAA is more efficient on A11
- Tracks unique colors in a pixel

Tile shading gives control over color coverage
- Resolve in-place, in fast tile memory
Multi-Sample Rendering with Transparent Particles
Without Color Coverage Control

- Pass 1: MSAA Render and Resolve
- Pass 2: Non-MSAA Particle Render

System Memory

Color and Depth

Final Image
Multi-Sample Rendering with Transparent Particles
With Tile Shading and Color Coverage Control

Single Render Pass
- MSAA Render
- In-Place Resolve
- Particle Render

Tile Memory
- MSAA Imageblock
- Non-MSAA Imageblock

System Memory
- Final Image
// Tile shader that resolves in-place using color coverage control

struct FragData {
    half4 color;
};

kernel void InPlaceResolve(imageblock<FragData> img_blk_colors,
                            ushort2 tid [[thread_position_in_threadgroup]]) {
    half4 resolved_color = half4(0);
    for (int c = 0; c < img_blk_colors.get_num_colors(tid); ++c) {
        half4 color = img_blk_colors.data(tid, c, image_block_data_rate::color)->color;
        resolved_color += color * popcount(img_blk_color.get_color_coverage_mask(tid, c));
    }
    resolved_color /= img_blk_colors.get_num_samples();
    ushort output_sample_mask = 0xF;
    img_blk_colors.write(FragData{resolved_color}, tid, output_sample_mask);
}
// Tile shader that resolves in-place using color coverage control

struct FragData {
    half4 color;
};

kernel void InPlaceResolve(imageblock<FragData> img_blk_colors,
                            ushort2 tid [[thread_position_in_threadgroup]]) {
    half4 resolved_color = half4(0);
    for (int c = 0; c < img_blk_colors.get_num_colors(tid); ++c) {
        half4 color = img_blk_colors.data(tid, c, image_block_data_rate::color)->color;
        resolved_color += color * popcount(img_blk_color.get_color_coverage_mask(tid, c));
    }
    resolved_color /= img_blk_colors.get_num_samples();
    ushort output_sample_mask = 0xF;
    img_blk_colors.write(FragData{resolved_color}, tid, output_sample_mask);
}
/ Tile shader that resolves in-place using color coverage control

struct FragData {
    half4 color;
};

kernel void InPlaceResolve(imageblock<FragData> img_blk_colors,
          ushort2 tid [[thread_position_in_threadgroup]]) {
    half4 resolved_color = half4(0);
    for (int c = 0; c < img_blk_colors.get_num_colors(tid); ++c) {
        half4 color = img_blk_colors.data(tid, c, image_block_data_rate::color)->color;
        resolved_color += color * popcount(img_blk_color.get_color_coverage_mask(tid, c));
    }
    resolved_color /= img_blk_colors.get_num_samples();
    ushort output_sample_mask = 0xF;
    img_blk_colors.write(FragData{resolved_color}, tid, output_sample_mask);
}
/ Tile shader that resolves in-place using color coverage control

```c
struct FragData {
    half4 color;
};
kernel void InPlaceResolve(imageblock<FragData> img_blk_colors,
    ushort2 tid [[thread_position_in_threadgroup]]) {
    half4 resolved_color = half4(0);
    for (int c = 0; c < img_blk_colors.get_num_colors(tid); ++c) {
        half4 color = img_blk_colors.data(tid, c, image_block_data_rate::color)->color;
        resolved_color += color * popcount(img_blk_color.get_color_coverage_mask(tid, c));
    }
    resolved_color /= img_blk_colors.get_num_samples();
    ushort output_sample_mask = 0xF;
    img_blk_colors.write(FragData{resolved_color}, tid, output_sample_mask);
}
// Tile shader that resolves in-place using color coverage control

struct FragData {
    half4 color;
};

kernel void InPlaceResolve(imageblock<FragData> img_blk_colors,
                           ushort2 tid [[thread_position_in_threadgroup]]) {
    half4 resolved_color = half4(0);
    for (int c = 0; c < img_blk_colors.get_num_colors(tid); ++c) {
        half4 color = img_blk_colors.data(tid, c, image_block_data_rate::color)->color;
        resolved_color += color * popcount(img_blk_color.get_color_coverage_mask(tid, c));
    }
    resolved_color /= img_blk_colors.get_num_samples();
    ushort output_sample_mask = 0xF;
    img_blk_colors.write(FragData{resolved_color}, tid, output_sample_mask);
}
Fortnite: Battle Royale

Shipping an Unreal Engine 4 console game on iOS with Metal

Nick Penwarden, Epic Games
Technical Challenges

One map, larger than $6\text{km}^2$

Time-of-day, destruction, player-built structures

100 players

50,000+ replicating actors

Crossplay with console and desktop players
One Game, All Platforms

Crossplay means we are limited in our ability to scale down the game.

If it affects gameplay, we can’t change it.

If a player can hide behind an object, we must render it.
Metal

Draw call performance allows us to render complex scenes with thousands of dynamic objects.

Access to hardware features, e.g. programmable blending, allows for big wins on the GPU.

Feature set lets us support artist authored materials, physically based rendering, dynamic lighting and shadows, GPU particle simulation.
Rendering Features Used on iOS

Movable directional light with cascaded shadow maps
Movable skylight
Physically based materials
HDR and Tonemapping
GPU particle simulation
Artist authored materials with vertex animation
Scalability

Scalability across platforms

• Minimum LOD for meshes
• Character LODs, animation update rates, etc.

Defined three performance buckets, assigned devices based on performance

• Low (iPhone 6s, iPad Air 2, iPad mini 4)
• Mid (iPhone 7, iPad Pro)
• High (iPhone 8, iPhone X, iPad Pro 2nd Gen)
Resolution

Backbuffer resolution is what the UI renders at, determined by contentScaleFactor.

3D resolution is separate, upscaled before rendering UI (adds some cost).

Preferred scaling backbuffer, only scaled 3D separately on iPhone 6s, iPad Air 2, and iPad (5th gen).

Tuned per-device.
Dynamic shadows have a large impact on both GPU and CPU perf

Low
No Shadows

Mid
1 cascade, 1024x1024

High
2 cascades, 1024x1024, 25% further out
Grass and foliage scales; both density and cull distance

- **Low**: No foliage
- **Mid**: 30% density vs. console
- **High**: 100% density vs. console
Memory

Memory doesn't always correlate with performance

• iPhone 8 has less physical memory than iPhone 7+ but is faster

Low memory devices

• No foliage, no shadows, 16k max GPU particles, reduced pool for cosmetics, reduced texture memory pool

High memory devices

• Foliage, shadows, 64k max GPU particles, increased pool for cosmetics, increased texture memory pool
Memory Optimizations

Streaming

• Split POIs into streaming levels

Textures

• Used ASTC optimizing for size
• Removed/reduced non-streaming textures
• Adjusted streaming pool
• Content optimization

Meshes

• Per-platform min LOD, cooked out unused LODs
• Cooked out meshes culled by low detail mode
• Don’t load grass/foliage on low memory devices
• Content optimization

Materials

• Cook out medium and high quality variants
• Disable feature permutations we don’t use, e.g. dynamic point lights
• Reduce unique materials in content
• Removed editor-only properties

Audio

• Per-platform variation culling
• Per-platform downsampling
• Per-platform compression quality
• Quality node
• Content optimization
Framerate Targets

30fps at the highest visual fidelity possible

However...

Maxing out the CPU and GPU generates heat which causes the device to downclock

We also want to conserve battery life

So we target 60fps for the environment but vsync at 30fps
Performance Tracking

Capture environment performance at a selection of POIs over time

Daily 100 player playtests to capture dynamic performance

- Key performance stats tracked
- Instrumented profiles gathered from one device
- Replays saved off for later investigation
Metal
Threading

On most iOS devices, we have two cores to work with

Game thread: Input, networking, simulation

Rendering thread: culling and drawing the scene, all Metal work

One task thread for async tasks, mostly streaming

From an iPhone 6s
Threading

On iPhone 8 and X we have two "fast" and four "efficient" cores

We add three more task threads for

• Parallel animation evaluation
• Particle simulation
• Physics tasks
• Async scene queries
• Texture streaming
• Scene culling for rendering
• And a dedicated audio thread

From an iPhone X
Draw Calls

Draw calls were our main performance bottleneck

Metal’s performance really helped here, 3–4x faster than OpenGL, allowed us to ship without more aggressive work to reduce draw calls

Pulled in cull distance on decorative objects

Added HLODs to combine draw calls
## Draw Calls

<table>
<thead>
<tr>
<th></th>
<th>CityOverlook</th>
<th></th>
<th>CityStreets</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Draw Calls</td>
<td>CPU Time (ms)</td>
<td>Draw Calls</td>
<td>CPU Time (ms)</td>
</tr>
<tr>
<td>iPhone 8+</td>
<td>1281</td>
<td>4.7</td>
<td>605</td>
<td>2.73</td>
</tr>
<tr>
<td>iPhone 7+</td>
<td>1246</td>
<td>7.11</td>
<td>544</td>
<td>4.22</td>
</tr>
</tbody>
</table>
Hierarchical LOD

Combine draw calls for a group of meshes

Allow us to render the entire map while skydiving

Even on the ground, POIs are visible from up to 2km away

Added mid-range HLODs to further reduce draw calls in complex scenes like Tilted Towers
Pipeline State Objects

Minimize how many are created at runtime to prevent hitching

Follow best practices

• Compile functions offline
• Build library offline
• Group functions into a single library you can ship with your game

Ideally create all of the PSOs you need at load time
Pipeline State Objects

What if the set of PSOs you *might* need is large?

- Artist authored shaders \times Lighting scenarios \times RT formats \times MSAA \times Stencil state (e.g. LOD dithering) \times Input layout \times Scalability level \times and more

- Minimize permutations where you can!
  - Sometimes a dynamic branch is fine
Pipeline State Objects

Identify the most common subset you are likely to need and create those at load.

We use automation to run the game and gather PSOs used by device across the map, load cosmetics, fire weapons, etc.

We also store PSOs created during daily playtests.

Not perfect, but...

• Number of PSOs created during gameplay is in the single digits on average.
• We only create a small subset of the permutation matrix on load.
Resource Allocation

Creating resources on the fly can hitch due to streaming or creation of dynamic objects.

Treat resource allocation like memory allocation and use the same strategies to minimize "malloc" and "free".

We use a binned allocation strategy for smaller allocations.
Programmable Blending

Optimization to reduce the number of resolves and restores for features that need to read the depth buffer, e.g. decals and soft particle blending.

During forward pass, write linear depth to alpha channel (we render to FP16 render target).

During decal and translucent passes, if needed, read \[[ color(0) ]\] .a as linear depth.
Programmable Blending

MSAA resolve happens before postprocessing and tonemapping

Bilinear filtering of HDR values can lead to very aliased edges, e.g. dark ceiling in shadow against a bright sky
Programmable Blending

Solution

- Pre-tonemap before resolve
- Perform the normal MSAA resolve
- The first postprocessing pass reverses the "pre-tonemap"

Use programmable blending for the pre-tonemap pass to avoid resolving the MSAA color buffer to memory!
Future Metal Work
Parallel Rendering

On macOS we generate command buffers in parallel

• Not using parallel command encoders right now, separate command buffers easier to integrate in existing parallel rendering architecture

• Parallel command encoders would be needed on iOS

Experiment with parallel rendering on iOS

• For example, rendering thread on fast core versus four encoding threads on efficient cores
Metal Heaps

Use MTLHeap instead of buffer sub-allocation

Would also reduce hitches due to texture streaming

Requires precise fences, particularly which stages read which resources
• Full barriers between passes underutilize the GPU
• Requires some rework of our rendering architecture to make this bulletproof
Continue to Push High End Graphics on iOS
Metal 2 for Game Rendering

Summary
Metal 2 for Game Rendering

Summary

Leverage multi-core CPUs
Metal 2 for Game Rendering

Summary

Leverage multi-core CPUs

Control memory and GPU concurrency for advanced performance needs
Metal 2 for Game Rendering

Summary

Leverage multi-core CPUs

Control memory and GPU concurrency for advanced performance needs

Move CPU work to the GPU
Metal 2 for Game Rendering

Summary

Leverage multi-core CPUs

Control memory and GPU concurrency for advanced performance needs

Move CPU work to the GPU

Control tile memory on A11 for high performance and extended playtime
More Information
