Modern Rendering with Metal

Jaap van Muijden, GPU Software
Srinivas Dasari, GPU Software
Advanced Rendering Techniques
GPU-Driven Pipelines
Simpler GPU Families
Advanced Rendering Techniques

Jaap van Muijden, GPUSW
Deferred
Tiled Deferred
Tiled and Clustered Forward
Visibility Buffer
Deferred Rendering
2-pass approach
Deferred Rendering

2-pass approach

Geometry → GBuffer → Lighting

Mesh data

Lit scene
Deferred Rendering

2-pass approach

Geometry → GBuffer → Lighting → Lit scene
Deferred Rendering with Metal

- Single pass
  - Geometry
  - Lighting

- System memory
Deferred Rendering with Metal

- Single pass
  - Geometry
  - Lighting

System memory
Deferred Rendering with Metal

- Single pass
  - Geometry
  - Lighting
- System memory
  - Depth
Deferred Rendering with Metal

- Single pass
  - Geometry
  - Lighting

- System memory
  - Depth
  - Normal
  - Albedo
  - Roughness
Deferred Rendering with Metal

Single pass
Geometry

Lighting

System memory
Depth
Normal
Albedo
Roughness
Deferred Rendering with Metal

- Geometry
- Lighting
- System memory
  - Depth
  - Normal
  - Albedo
  - Roughness
- Lit scene

Single pass
Render Pass Setup
Render Pass Setup

Render Pass Descriptor

Render pass
Render Pass Setup

Render Pass Descriptor

Attachments define output
Render Pass Setup

Render Pass Descriptor

Attachments define output

- Texture: Render Target
Render Pass Setup

Render Pass Descriptor

Attachments define output
- Texture: Render Target
- Load actions
Render Pass Setup

Render Pass Descriptor

Attachments define output

- Texture: Render Target
- Load actions
- Store actions
Render Pass Setup

Render Pass Descriptor

Attachments define output
- Texture: Render Target
- Load actions
- Store actions

Render Command Encoder
func setupDeferred() {
    // geometry pass
    geoRpd = MTLRenderPassDescriptor()

    geoRpd.depthAttachment.texture = depthTexture
    geoRpd.depthAttachment.loadAction = .clear
    geoRpd.depthAttachment.storeAction = .store

    geoRpd.colorAttachments[0].texture = albedoTexture
    geoRpd.colorAttachments[0].loadAction = .dontcare
    geoRpd.colorAttachments[0].storeAction = .store

    ... // lighting pass
    lgtRpd = MTLRenderPassDescriptor()

    lgtRpd.colorAttachments[0].texture = litSceneTexture
    lgtRpd.colorAttachments[0].loadAction = .clear
    lgtRpd.colorAttachments[0].storeAction = .store
}
func setupDeferred()
{
    // geometry pass
    geoRpd = MTLRenderPassDescriptor()
    geoRpd.depthAttachment.texture = depthTexture
    geoRpd.depthAttachment.loadAction = .clear
    geoRpd.depthAttachment.storeAction = .store
    geoRpd.colorAttachments[0].texture = albedoTexture
    geoRpd.colorAttachments[0].loadAction = .dontcare
    geoRpd.colorAttachments[0].storeAction = .store
    ...

    // lighting pass
    lgtRpd = MTLRenderPassDescriptor()
    lgtRpd.colorAttachments[0].texture = litSceneTexture
    lgtRpd.colorAttachments[0].loadAction = .clear
    lgtRpd.colorAttachments[0].storeAction = .store
}
func setupDeferred()
{
    // geometry pass
    geoRpd = MTLRenderPassDescriptor()

    geoRpd.depthAttachment.texture = depthTexture
    geoRpd.depthAttachment.loadAction = .clear
    geoRpd.depthAttachment.storeAction = .store
    geoRpd.colorAttachments[0].texture = albedoTexture
    geoRpd.colorAttachments[0].loadAction = .dontcare
    geoRpd.colorAttachments[0].storeAction = .store

    // lighting pass
    lgtRpd = MTLRenderPassDescriptor()

    lgtRpd.colorAttachments[0].texture = litSceneTexture
    lgtRpd.colorAttachments[0].loadAction = .clear
    lgtRpd.colorAttachments[0].storeAction = .store
}
func setupDeferred()
{
    // geometry pass
    geoRpd = MTLRenderPassDescriptor()
    geoRpd.depthAttachment.texture = depthTexture
    geoRpd.depthAttachment.loadAction = .clear
    geoRpd.depthAttachment.storeAction = .store
    geoRpd.colorAttachments[0].texture = albedoTexture
    geoRpd.colorAttachments[0].loadAction = .dontcare
    geoRpd.colorAttachments[0].storeAction = .store

    // lighting pass
    lgtRpd = MTLRenderPassDescriptor()
    lgtRpd.colorAttachments[0].texture = litSceneTexture
    lgtRpd.colorAttachments[0].loadAction = .clear
    lgtRpd.colorAttachments[0].storeAction = .store
}
func setupDeferred()
{
    // geometry pass
    geoRpd = MTLRenderPassDescriptor()
    geoRpd.depthAttachment.texture            = depthTexture
    geoRpd.depthAttachment.loadAction         = .clear
    geoRpd.depthAttachment.storeAction        = .store
    geoRpd.colorAttachments[0].texture        = albedoTexture
    geoRpd.colorAttachments[0].loadAction     = .dontcare
    geoRpd.colorAttachments[0].storeAction    = .store
    ...

    // lighting pass
    lgtRpd = MTLRenderPassDescriptor()
    lgtRpd.colorAttachments[0].texture        = litSceneTexture
    lgtRpd.colorAttachments[0].loadAction     = .clear
    lgtRpd.colorAttachments[0].storeAction    = .store
}
func setupDeferred()
{
    // geometry pass
    geoRpd = MTLRenderPassDescriptor()
    geoRpd.depthAttachment.texture = depthTexture
    geoRpd.depthAttachment.loadAction = .clear
    geoRpd.depthAttachment.storeAction = .store
    geoRpd.colorAttachments[0].texture = albedoTexture
    geoRpd.colorAttachments[0].loadAction = .dontcare
    geoRpd.colorAttachments[0].storeAction = .store
    ...

    // lighting pass
    lgtRpd = MTLRenderPassDescriptor()
    lgtRpd.colorAttachments[0].texture = litSceneTexture
    lgtRpd.colorAttachments[0].loadAction = .clear
    lgtRpd.colorAttachments[0].storeAction = .store
}
func render(cmdBuffer: MTLCommandBuffer)
{
    // geometry
    let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
    for mesh in _scene.meshes {
        // set state
        geoPass.drawIndexedPrimitives(...) // draw scene object
    }
    geoPass.endEncoding()

    // lighting
    let lgtPass = commandBuffer.makeRenderCommandEncoder(descriptor: lgtRpd)
    for light in _scene.lights {
        lgtPass.setFragmentTexture(albedoTexture ...) // bind gBuffer textures
        lgtPass.drawIndexedPrimitives(...) // draw light volume
    }
    lgtPass.endEncoding()
}
func render(cmdBuffer: MTLCommandBuffer) {
  // geometry
  let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
  for mesh in _scene.meshes {
    // set state
    geoPass.drawIndexedPrimitives(...) // draw scene object
  }
  geoPass.endEncoding()

  // lighting
  let lgtPass = commandBuffer.makeRenderCommandEncoder(descriptor: lgtRpd)
  for light in _scene.lights {
    lgtPass.setFragmentTexture(albedoTexture ...) // bind gBuffer textures
    lgtPass.drawIndexedPrimitives(...) // draw light volume
  }
  lgtPass.endEncoding()
}
func render(cmdBuffer: MTLCommandBuffer) {

    // geometry
    let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
    for mesh in _scene.meshes {
        // set state
        geoPass.drawIndexedPrimitives(...) // draw scene object
    }
    geoPass.endEncoding()

    // lighting
    let lgtPass = commandBuffer.makeRenderCommandEncoder(descriptor: lgtRpd)
    for light in _scene.lights {
        lgtPass.setFragmentTexture(albedoTexture ...) // bind gBuffer textures
        lgtPass.drawIndexedPrimitives(...) // draw light volume
    }
    lgtPass.endEncoding()
}

func render(cmdBuffer: MTLCommandBuffer) {

    // geometry
    let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
    for mesh in _scene.meshes {
        // set state
        geoPass.drawIndexedPrimitives(...) // draw scene object
    }
    geoPass.endEncoding()

    // lighting
    let lgtPass = commandBuffer.makeRenderCommandEncoder(descriptor: lgtRpd)
    for light in _scene.lights {
        lgtPass.setFragmentTexture(albedoTexture ...) // bind gBuffer textures
        lgtPass.drawIndexedPrimitives(...) // draw light volume
    }
    lgtPass.endEncoding()
}
}
func render(cmdBuffer: MTLCommandBuffer)
{
    // geometry
    let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
    for mesh in _scene.meshes {
        // set state
        geoPass.drawIndexedPrimitives(...) // draw scene object
    }
    geoPass.endEncoding()

    // lighting
    let lgtPass = commandBuffer.makeRenderCommandEncoder(descriptor: lgtRpd)
    for light in _scene.lights {
        lgtPass.setFragmentTexture(albedoTexture ...) // bind gBuffer textures
        lgtPass.drawIndexedPrimitives(...) // draw light volume
    }
    lgtPass.endEncoding()
}
Programmable Blending

Multiple passes

Geometry

Lighting

System memory

*Depth*  
*Normal*  
*Albedo*  
*Roughness*

*Lit*  
*Scene*
Programmable Blending

Multiple passes

Geometry

Lighting

Scene

Lighting

Geometry

Lit

Roughness

Albedo

Depth

Memory
Programmable Blending
Merge passes

Geometry

Lighting
Programmable Blending

Merge passes

Single pass

Geometry

Lighting
Programmable Blending

Merge passes

Single pass
Geometry

Lighting

Tile memory

System memory
fragment float4 Shade(LightingVertexInput in
  depth2d <float, access::read> depth_tex
  texture2d <float, access::read> color_tex
  texture2d <float, access::read> normal_tex
  texture2d <float, access::read> rough_tex,
  ...
){
  float4 color = color_tex.read(in.pixelPos);
  uint normal = normal_tex.read(in.pixelPos);
  uint roughness = rough_tex.read(in.pixelPos);
  float depth = depth_tex.read(in.pixelPos);

  return runLightingModel(color, normal, roughness, depth, ...);
}
fragment float4 Shade(LightingVertexInput in, [[stage_in]],

depth2d <float, access::read> depth_tex [[texture (0)]],
texture2d <float, access::read> color_tex [[texture (1)]],
texture2d <float, access::read> normal_tex [[texture (2)]],
texture2d <float, access::read> rough_tex [[texture (3)]],
...
)
{
    float4 color = color_tex.read(in.pixelPos);
    uint normal = normal_tex.read(in.pixelPos);
    uint roughness = rough_tex.read(in.pixelPos);
    float depth = depth_tex.read(in.pixelPos);

    return runLightingModel(color, normal, roughness, depth, ...);
}

fragment float4 Shade(LightingVertexInput in
    [[stage_in]],
    depth2d <float, access::read> depth_tex [[texture (0)]],
    texture2d <float, access::read> color_tex [[texture (1)]],
    texture2d <float, access::read> normal_tex [[texture (2)]],
    texture2d <float, access::read> rough_tex [[texture (3)]],
    ...)
{
    float4 color = color_tex.read(in.pixelPos);
    uint normal = normal_tex.read(in.pixelPos);
    uint roughness = rough_tex.read(in.pixelPos);
    float depth = depth_tex.read(in.pixelPos);

    return runLightingModel(color, normal, roughness, depth, ...);
}
fragment float4 Shade(LightingVertexInput in [[stage_in]],
    depth2d <float, access::read> depth_tex [[texture (0)]],
    texture2d <float, access::read> color_tex [[texture (1)]],
    texture2d <float, access::read> normal_tex [[texture (2)]],
    texture2d <float, access::read> rough_tex [[texture (3)]],
    ...
) {
    float4 color = color_tex.read(in.pixelPos);
    uint normal = normal_tex.read(in.pixelPos);
    uint roughness = rough_tex.read(in.pixelPos);
    float depth = depth_tex.read(in.pixelPos);

    return runLightingModel(color, normal, roughness, depth, ...);
}
fragment float4 Shade(LightingVertexInput in
    float4 color
    float4 normal
    float roughness
    float depth
    ...)
{
    return runLightingModel(color, normal, roughness, depth, ...);
}
fragment float4 Shade(LightingVertexInput in [[stage_in]],
    float4 color [[color (0)]],
    float4 normal [[color (1)]],
    float roughness [[color (2)]],
    float depth [[color (3)]],
    ...
)
{
    return runLightingModel(color, normal, roughness, depth, ...);
}
fragment float4 Shade(LightingVertexInput in
                      [[stage_in]],
                      float4 color
                      [[color (0)]],
                      float4 normal
                      [[color (1)]],
                      float roughness
                      [[color (2)]],
                      float depth
                      [[color (3)]],
                      ...
)
{
  return runLightingModel(color, normal, roughness, depth, ...);
}
Programmable Blending
Don’t store transient attachments

```python
goRpd = MTLRenderPassDescriptor()
goRpd.colorAttachments[0].texture = albedoTexture
goRpd.colorAttachments[0].loadAction = .dontcare
goRpd.colorAttachments[0].storeAction = .store
```
Programmable Blending

Don’t store transient attachments

```swift
geoRpd = MTLRenderPassDescriptor()
geoRpd.colorAttachments[0].texture = albedoTexture
geoRpd.colorAttachments[0].loadAction = .dontcare
geoRpd.colorAttachments[0].storeAction = .dontcare
```
Programmable Blending

Don’t allocate transient textures either

```
.storageMode = .memoryless
```

- Single pass
  - Geometry
  - Lighting
- Tile memory
- System memory
# Deferred Rendering

<table>
<thead>
<tr>
<th></th>
<th>Material/Light Separation</th>
<th>Many Lights</th>
<th>Transparency</th>
<th>Anti-Aliasing</th>
<th>Material Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiled Deferred</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiled Forward</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster Forward</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visibility Buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tiled Deferred Rendering

Mesh data → GBuffer → Light cull → Light tiles → Lit scene
Tiled Deferred Rendering
Cull lights against geometry depth bounds
Tiled Deferred Rendering
Cull lights against geometry depth bounds
Tiled Deferred Rendering
Cull lights against geometry depth bounds
Tiled Deferred Rendering with Metal

Multiple passes:
- Geometry
- Lighting

System memory:
- Depth, GBuffer
- Lit scene

Diagram shows the process of rendering with multiple passes through geometry and lighting stages, utilizing system memory for depth and GBuffer data, leading to a lit scene.
Tiled Deferred Rendering with Metal

- Multiple passes
  - Geometry (compute)
  - Light culling (compute)
  - Lighting (per tile)

System memory
- Depth, GBuffer
- Lit scene
Tiled Deferred Rendering with Metal

Geometry

Light culling (compute)

Lighting (per tile)

Multiple passes

System memory

Depth, GBuffer

Light lists

Lit scene
Tile Shaders
Merge tile-scoped compute into the render pass

Single pass

Geometry
Light culling (compute)
Lighting (per tile)

System memory

Light lists
Lit scene
Persisting Threadgroup Memory

- Single render pass
- Tile memory
- Threadgroup memory
- System memory

- Geometry
- Light culling (compute)
- Lighting (per tile)

- Light list

Lit scene
func setupDeferredTiledOnDevice(device: MTLDevice)
{
    // setup tile shader descriptor
    tileCullDesc = MTLTileRenderPipelineDescriptor()

    // setup color attachments
    tileCullDesc.colorAttachments[0].PixelFormat = .rgba16Uint // albedo
    tileCullDesc.colorAttachments[1].PixelFormat = .r32Float // linear depth
    ... 

    tileCullDesc.tileFunction = lightCull
    tilePpl = device.makeRenderPipelineState(descriptor: tileCullDesc)

    // setup merged pass
    mrgRpd = MTLTileRenderPipelineDescriptor()
    mrgRpd.threadgroupMemoryLength = MemoryLayout<LightList>.size
    ... 
}
func setupDeferredTiledOnDevice(device: MTLDevice)
{

    // setup tile shader descriptor
    tileCullDesc = MTLTileRenderPipelineDescriptor()

    // setup color attachments
    tileCullDesc.colorAttachments[0].PixelFormat = .rgba16Uint // albedo
    tileCullDesc.colorAttachments[1].PixelFormat = .r32Float   // linear depth
    ...

    tileCullDesc.tileFunction = lightCull
    tilePpl = device.makeRenderPipelineState(descriptor: tileCullDesc)

    // setup merged pass
    mrgRpd = MTLTileRenderPipelineDescriptor()
    mrgRpd.threadgroupMemoryLength = MemoryLayout<LightList>.size
    ...
}

func setupDeferredTiledOnDevice(device: MTLDevice) {
  // setup tile shader descriptor
  tileCullDesc = MTLTileRenderPipelineDescriptor()
  // setup color attachments
  tileCullDesc.colorAttachments[0].pixelFormat = .rgba16UInt // albedo
  tileCullDesc.colorAttachments[1].pixelFormat = .r32Float   // linear depth
  ...
  tileCullDesc.tileFunction = lightCull
  tilePpl = device.makeRenderPipelineState(descriptor: tileCullDesc)
  // setup merged pass
  mrgRpd = MTLTileRenderPipelineDescriptor()
  mrgRpd.threadgroupMemoryLength = MemoryLayout<LightList>.size
  ...
}
func setupDeferredTiledOnDevice(device: MTLDevice)
{
    // setup tile shader descriptor
    tileCullDesc = MTLTileRenderPipelineDescriptor()

    // setup color attachments
    tileCullDesc.colorAttachments[0].pixelFormat = .rgba16UInt // albedo
    tileCullDesc.colorAttachments[1].pixelFormat = .r32Float // linear depth

    tileCullDesc.tileFunction = lightCull
    tilePpl = device.makeRenderPipelineState(descriptor: tileCullDesc)

    // setup merged pass
    mrgRpd = MTLTileRenderPipelineDescriptor()
    mrgRpd.threadgroupMemoryLength = MemoryLayout<LightList>.size

    ...
}
```swift
func setupDeferredTiledOnDevice(device: MTLDevice)
{
    // setup tile shader descriptor
tileCullDesc = MTLTileRenderPipelineDescriptor()

    // setup color attachments
tileCullDesc.colorAttachments[0].pixelFormat = .rgba16UInt // albedo
tileCullDesc.colorAttachments[1].pixelFormat = .r32Float // linear depth
...

tileCullDesc.tileFunction = lightCull
tilePpl = device.makeRenderPipelineState(descriptor: tileCullDesc)

    // setup merged pass
    mrgRpd = MTLTileRenderPipelineDescriptor()
    mrgRpd.threadgroupMemoryLength = MemoryLayout<LightList>.size
    ...
}
```
func setupDeferredTiledOnDevice(device: MTLDevice)
{
    // setup tile shader descriptor
    tileCullDesc = MTLTileRenderPipelineDescriptor()

    // setup color attachments
    tileCullDesc.colorAttachments[0].PixelFormat = .rgba16Uint // albedo
    tileCullDesc.colorAttachments[1].PixelFormat = .r32Float   // linear depth
...

    tileCullDesc.tileFunction = lightCull
    tilePpl = device.makeRenderPipelineState(descriptor: tileCullDesc)

    // setup merged pass
    mrgRpd = MTLTileRenderPipelineDescriptor()
    mrgRpd.threadgroupMemoryLength = MemoryLayout<LightList>.size
...
}
func render(cmdBuffer: MTLCommandBuffer) {
    // merged pass
    let encoder = cmdBuffer.makeRenderCommandEncoder(descriptor: rpd)
    // meshes
    for mesh in _scene.meshes {
        ...
        encoder.drawIndexedPrimitives(...) // scene object
    }

    // tile shader
    encoder.setRenderPipelineState(tileCullDesc)
    encoder.setTileBuffer(sceneLights, offset:0 atIndex:0)
    encoder.setThreadgroupMemoryLength(MemoryLayout<LightList>.size, offset:0 atIndex:0)
    encoder.dispatchThreadsPerTile(MTLSizeMake(encoder.tileWidth, encoder.tileHeight, 1))
    // lighting
    encoder.drawIndexedPrimitives(...)
    encoder.endEncoding()
}
func render(cmdBuffer: MTLCommandBuffer) {
    // merged pass
    let encoder = cmdBuffer.makeRenderCommandEncoder(descriptor: rpd)

    // meshes
    for mesh in _scene.meshes {
        ...
        encoder.drawIndexedPrimitives(...) // scene object
    }

    // tile shader
    encoder.setRenderPipelineState(tileCullDesc)
    encoder.setTileBuffer(sceneLights, offset: 0 atIndex: 0)
    encoder.setThreadgroupMemoryLength(MemoryLayout<LightList>.size, offset: 0 atIndex: 0)
    encoder.dispatchThreadsPerTile(MTLSizeMake(encoder.tileWidth, encoder.tileHeight, 1))

    // lighting
    encoder.drawIndexedPrimitives(...) // scene object
    encoder.endEncoding()}
}
func render(cmdBuffer: MTLCommandBuffer)
{
    // merged pass
    let encoder = cmdBuffer.makeRenderCommandEncoder(descriptor: rpd)

    // meshes
    for mesh in _scene.meshes {
        ...
        encoder.drawIndexedPrimitives(...) // scene object
    }

    // tile shader
    encoder.setRenderPipelineState(tileCullDesc)
    encoder.setTileBuffer(sceneLights, offset:0 atIndex:0)
    encoder.setThreadgroupMemoryLength(MemoryLayout<LightList>.size, offset:0 atIndex:0)
    encoder.dispatchThreadsPerTile(MTLSizeMake(encoder.tileWidth, encoder.tileHeight, 1))
    // lighting
    encoder.drawIndexedPrimitives(...)
    encoder.endEncoding()
}
func render(cmdBuffer: MTLCommandBuffer)
{
    // merged pass
    let encoder = cmdBuffer.makeRenderCommandEncoder(descriptor: rpd)
    // meshes
    for mesh in _scene.meshes  {
        ...
        encoder.drawIndexedPrimitives(...) // scene object
    }

    // tile shader
    encoder.setRenderPipelineState(tileCullDesc)
    encoder.setTileBuffer(sceneLights, offset:0 atIndex:0)
    encoder.setThreadgroupMemoryLength(MemoryLayout<LightList>.size, offset:0 atIndex:0)
    encoder.dispatchThreadsPerTile(MTLSizeMake(encoder.tileWidth, encoder.tileHeight, 1))

    // lighting
    encoder.drawIndexedPrimitives(...)
    encoder.endEncoding()
}
func render(cmdBuffer: MTLCommandBuffer) {
    // merged pass
    let encoder = cmdBuffer.makeRenderCommandEncoder(descriptor: rpd)
    // meshes
    for mesh in _scene.meshes {
        ... 
        encoder.drawIndexedPrimitives(...) // scene object
    }

    // tile shader
    encoder.setRenderPipelineState(tileCullDesc)
    encoder.setTileBuffer(sceneLights, offset:0 atIndex:0)
    encoder.setThreadgroupMemoryLength(MemoryLayout<LightList>.size, offset:0 atIndex:0)
    encoder.dispatchThreadsPerTile(MTLSizeMake(encoder.tileWidth, encoder.tileHeight, 1))
    // lighting
    encoder.drawIndexedPrimitives(...)
    encoder.endEncoding()}
}
kernel void CullLights(device Light *all_lights [[buffer(0)]], ... 
    threadgroup uint32_t &active_light_list [[threadgroup(0)]],
    threadgroup float2 &depth_bounds [[threadgroup(1)]])
{
    active_light_mask = 0; // clear light list
    for (uint i = tid; i < MAX_LIGHTS; ++i) {
        if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, ...) 
            active_light_list = (1u << i);
    }
}

fragment float4 Shade(VertexInputs stage_in [[stage_in]],
    float4 albedo [[color(0)]], ... 
    device Light *all_lights [[buffer(0)]],
    threadgroup uint32_t &active_light_list [[threadgroup(0)]])
{
    // light pixel
}
kernel void CullLights(device Light *all_lights [[buffer(0)]], …
    threadgroup uint32_t &active_light_list [[threadgroup(0)]],
    threadgroup float2 &depth_bounds [[threadgroup(1)]])
{
    active_light_mask = 0; // clear light list
    for (uint i = tid; i < MAX_LIGHTS; ++i) {
        if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, …)
            active_light_list = (1u << i);
    }
}

fragment float4 Shade(VertexInputs stage_in [[stage_in]],
    float4 albedo [[color(0)]], …
    device Light *all_lights [[buffer(0)]],
    threadgroup uint32_t &active_light_list [[threadgroup(0)]])
{
    // light pixel
}
kernel void CullLights(device Light *all_lights [[buffer(0)]], …
    threadgroup uint32_t &active_light_list [[threadgroup(0)]],
    threadgroup float2 &depth_bounds [[threadgroup(1)]])
{
    active_light_mask = 0; // clear light list
    for (uint i = tid; i < MAX_LIGHTS; ++i) {
        if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, …)
            active_light_list = (1u << i);
    }
}

fragment float4 Shade(VertexInputs stage_in [[stage_in]],
    float4 albedo [[color(0)]], …
    device Light *all_lights [[buffer(0)]],
    threadgroup uint32_t &active_light_list [[threadgroup(0)]])
{
    // light pixel
}
kernel void CullLights(device Light *all_lights         [[buffer(0)]], ... 
    threadgroup uint32_t &active_light_list [[threadgroup(0)]], 
    threadgroup float2 &depth_bounds         [[threadgroup(1)]]
{
    active_light_mask = 0; // clear light list
    for (uint i = tid; i < MAX_LIGHTS; ++i) {
        if (IntersectLightWithTileFrustum(all_lights[i], depth_bounds, ...)
            active_light_list = (1u << i);
    }
}

fragment float4 Shade(VertexInputs stage_in            [[stage_in]], 
    float4 albedo                           [[color(0)]], ... 
    device Light *all_lights                [[buffer(0)]], 
    threadgroup uint32_t &active_light_list [[threadgroup(0)]])
{
    // light pixel
}
Forward Rendering with Light Lists

- Single render pass
- Geometry
- Light culling (tile shading)
- Lighting
- Forward

Tile memory
Threadgroup memory
System memory

Light list
Forward Rendering with Light Lists

Single render pass

Tile memory

Threadgroup memory

System memory

- Geometry
- Light culling (tile shading)
- Lighting
- Forward

Light list
# Tiled Deferred Rendering

<table>
<thead>
<tr>
<th></th>
<th>Material/Light Separation</th>
<th>Many Lights</th>
<th>Transparency</th>
<th>Anti-Aliasing</th>
<th>Material Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred</td>
<td>🟢</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiled Deferred</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td></td>
</tr>
<tr>
<td>Tiled Forward</td>
<td>Louisiana</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster Forward</td>
<td>Louisville</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visibility Buffer</td>
<td>Vermont</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tiled Forward Rendering

- Single render pass
- Tile memory
- Threadgroup memory
- System memory

- Geometry
- Light culling (tile shading)
- Lighting
- Forward

Light list
Tiled Forward Rendering

Single render pass

Tile memory

Threadgroup memory

System memory

Light culling (tile shading)

Forward

Light list
Tiled Forward Rendering

- Single render pass
- Depth only
- Light culling (tile shading)
- Forward

Tile memory

Threadgroup memory

System memory

Light list (tile shading)
Tiled Forward Rendering

- Single render pass
- Depth only
- Light culling (tile shading)
- Forward
- Tile memory
- Threadgroup memory
- System memory
- Light list
Clustered Light Culling

Subdivide tiles into depth
Clustered Light Culling
Subdivide tiles into depth
Clustered Light Culling
Generate a 3D light list
Clustered Forward Rendering
No depth pass needed
<table>
<thead>
<tr>
<th>Forward Rendering</th>
<th>Material/Light Separation</th>
<th>Many Lights</th>
<th>Transparency</th>
<th>Anti-Aliasing</th>
<th>Material Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiled Deferred</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tiled Forward</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cluster Forward</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Visibility Buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Visibility Buffer

Minimal GBuffer

Multiple passes

Geometry

Lighting

System memory

Depth
Normal
Albedo
Roughness

Lit scene
Visibility Buffer
Minimal GBuffer

- Multiple passes
  - Geometry
  - Reconstruction
  - Lighting

System memory
- Depth
- PrimitiveID
- Barycentrics
- Lit scene
Primitive ID and Barycentric Coordinates

- float3 [[barycentric_coord]]
  - barycentric coordinates for current fragment

- uint [[primitive_id]]
  - primitive ID of the current primitive
fragment GBufferOutput GenerateVisibility(VertexToFragment in, constant uint& mesh_id, float3 barycentrics, uint prim_id) {
    GBufferOutput out;
    out.barycentrics = barycentrics.xy;
    out.geometry_ids = (prim_id & 0xFFFF) + (mesh_id << 16);
    out.depth = in.position.z;
    return out;
}
fragment GBufferOutput GenerateVisibility(VertexToFragmnet in, 
  constant uint& mesh_id, 
  float3 barycentrics, 
  uint prim_id)
{
  GBufferOutput out;
  out.barycentrics = barycentrics.xy;
  out.geometry_ids = (prim_id & 0xFFFF) + (mesh_id << 16);
  out.depth = in.position.z;
  return out;
}
<table>
<thead>
<tr>
<th></th>
<th>Material/Light Separation</th>
<th>Many Lights</th>
<th>Transparency</th>
<th>Anti-Aliasing</th>
<th>Material Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiled Deferred</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiled Forward</td>
<td>✅</td>
<td></td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Cluster Forward</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Visibility Buffer</td>
<td>✅</td>
<td></td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
</tbody>
</table>
GPU Driven Pipelines

Srinivas Dasari, GPUSW
Large scenes require complex rendering operations
Render Loop

Large scenes require complex rendering operations

- Frustum culling
Render Loop

Large scenes require complex rendering operations

- Frustum culling
- Occlusion culling
Render Loop

Large scenes require complex rendering operations

• Frustum culling
• Occlusion culling
• LOD selection
Traditional CPU Driven Render Loop

- **Previous frame**
  - CPU: Encode occluders
  - GPU: Encode occlusion tests

- **Current frame**
  - CPU:
    - Encode occluders
    - Frustum culling
    - LOD selection
    - Occlusion culling
    - Encode draw
  - GPU:
    - Render pass

- System memory:
  - Occluder data
  - Command buffer
  - Occluder data
  - Command buffer
Traditional CPU Driven Render Loop

- Encoder occluders
- Encode occlusion tests
- Frustum culling
- LOD selection
- Occlusion culling
- Encode draw

Previous frame

CPU
- Occluder data
- Command buffer
- Occluder data

GPU
- Render pass
- Command buffer

System memory
Traditional CPU Driven Render Loop

- **CPU**
  - Encode occluders
  - Encode occlusion tests
  - Frustum culling
  - LOD selection
  - Occlusion culling
  - Encode draw

- **GPU**
  - Occlusion render pass

- **System memory**
  - Occluder data
  - Command buffer
  - Occluder data
  - Command buffer

- **Current frame**
  - Render pass
Traditional CPU Driven Render Loop

- Encode occluders
- Encode occlusion tests
- Frustum culling
- LOD selection
- Occlusion culling
- Encode draw

Previous frame

CPU

Occluder data

GPU

Command buffer

System memory

Occluder data

Current frame

Encode occluders

Frustum culling

Occlusion render pass

Render pass

Command buffer
Traditional CPU Driven Render Loop

- Command buffer
- Frustum culling
- LOD selection
- Occlusion culling
- Encode draw

CPU
- Occluder data
- Encode occluders
- Encode occlusion tests
- Frustum culling
- LOD selection
- Occlusion culling

GPU
- Occluder data
- Occlusion render pass
- Encode draw
- Render pass

System memory
- Command buffer
- Occluder data
- Command buffer
Traditional CPU Driven Render Loop

- CPU
  - Encode occluders
  - Encode occlusion tests
  - Frustum culling
  - LOD selection
  - Occlusion culling
  - Encode draw

- GPU
  - Render pass

- System memory
  - Occluder data
  - Command buffer
  - Occluder data

- Previous frame
- Current frame
Serial Execution on the CPU

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Frustum cull Object 0</th>
<th>Frustum cull Object 1</th>
<th>Frustum cull Object 2</th>
<th>Frustum cull Object 3</th>
<th>Frustum cull Object 4</th>
<th>Frustum cull Object 5</th>
<th>Frustum cull Object 6</th>
<th>Frustum cull Object 7</th>
</tr>
</thead>
</table>
Parallel Execution on the CPU

Thread 0
- Frustum cull
  - Object 0
  - Object 1
  - Object 2
  - Object 3

Thread 1
- Frustum cull
  - Object 4
  - Object 5
  - Object 6
  - Object 7
Parallel Execution on the CPU

- Frustum cull
- Select LOD
- Occlusion cull

Thread 0
- Object 0
- Object 0
- Object 0

Thread 1
- Object 1
- Object 1
- Object 1
Parallel Execution on the GPU

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GPU Driven Render Loop

Current frame

GUI
- Compute pass
  - Frustum culling occluders
- Render pass
  - Render occluders
- Compute pass
  - Generate occluder data
- Compute pass
  - Frustum culling
  - LOD selection
  - Occlusion culling
  - Encode draws
- Render pass
  - Render scene

System memory
- Occluder draw commands
- Occluder data
- Scene draw commands
- Scene data
GPU Driven Render Loop

Current frame

GPU

Compute pass
Frustum culling occluders
Encode occluder draws

Render pass
Render occluders

Compute pass
Generate occluder data

Compute pass
Frustum culling
LOD selection
Occlusion culling
Encode draws

Render pass
Render scene

System memory

Occluder draw commands
Occluder data
Scene draw commands
Scene data
GPU Driven Render Loop

- **Scene draw commands**
- **Render pass**
  - Frustum culling occluders
  - Encode occluder draws
- **Render occluders**
- **Compute pass**
  - Generate occluder data
- **Render pass**
  - Frustum culling
  - LOD selection
  - Occlusion culling
  - Encode draws
- **Render scene**

- **System memory**
  - Occluder draw commands
  - Occluder data
  - Scene draw commands
  - Scene data
GPU Driven Render Loop

Current frame

GPU
- Compute pass
  - Frustum culling occluders
  - Encode occluder draws
- Render pass
  - Render occluders
- Compute pass
  - Generate occluder data
- Compute pass
  - Frustum culling
  - LOD selection
  - Occlusion culling
  - Encode draws
- Render pass
  - Render scene

System memory
- Occluder draw commands
- Occluder data
- Scene draw commands

Scene data
GPU Driven Render Loop
GPU Driven Render Loop

- Scene data
- System memory
- Compute pass: Frustum culling occluders, Encode occluder draws
- Render pass: Render occluders
- Compute pass: Generate occluder data
- Compute pass: Frustum culling, LOD selection, Occlusion culling, Encode draws
- Render pass: Frustum culling occluders, Encode occluder draws
- Render scene

Current frame
Metal Building Blocks

Argument Buffers
- Make scene data available on the GPU
- Describe complex data structures

Indirect Command Buffers
- Encode draws on the GPU
Scene Object Model Example

- **Scene**
  - **Meshes**
    - Mesh 0
    - Mesh 1
    - ...
  - **Materials**
  - **Models**
    - Model 0
    - Model 1
    - ...

- **Meshes**
  - **Mesh**
    - Index buffer
    - Vertex buffer

- **Materials**
  - Model 0
  - Model 1
  - ...
    - Roughness
    - Intensity
    - Surface texture
    - Specular texture
    - Pipeline state object

- **Models**
  - **Model**
    - Array of mesh indices
    - Array of material indices
Scene Object Model Example

- **Scene**
  - **Meshes**
    - Mesh 0
    - Mesh 1
    - ...
  - **Materials**
  - **Models**
    - Model 0
    - Model 1
    - ...

- **Meshes**
  - **Mesh**
    - Index buffer
    - Vertex buffer

- **Models**
  - **Model**
    - Array of mesh indices
    - Array of material indices

- **Materials**
  - **Model**
  - **Material**
    - Roughness
    - Intensity
    - Surface texture
    - Specular texture
    - Pipeline state object
Scene Object Model Example

Meshes
- Mesh 0
- Mesh 1
- ...

Materials
- Mesh 0
- Mesh 1
- ...

Models
- Model 0
- Model 1
- ...

Index buffer
Vertex buffer
Array of mesh indices
Array of material indices
Roughness
Intensity
Surface texture
Specular texture
Pipeline state object
Scene Object Model Example

- Scene
  - Meshes
    - Mesh 0
    - Mesh 1
    - ...
  - Materials
  - Models
    - Model 0
    - Model 1
    - ...

Meshes
- Index buffer
- Vertex buffer

Materials
- Roughness
- Intensity
- Surface texture
- Specular texture
- Pipeline state object

Models
- Array of mesh indices
- Array of material indices

Model
Scene Object Model Example

```
Scene
  Meshes
  | Mesh 0
  | Mesh 1
  | ...
  Materials
  Models
  | Model 0
  | Model 1
  | ...
  Models
  | Array of mesh indices
  | Array of material indices
  Materials
  | Model 0
  | Model 1
  | ...
  Materials
  | Roughness
  | Intensity
  | Surface texture
  | Specular texture
  | Pipeline state object
```
Scene Object Model Example

Scene
- Meshes
  - Mesh 0
  - Mesh 1
  - ...
- Materials
- Models
  - Model 0
  - Model 1
  - ...

Meshes
- Mesh
  - Index buffer
  - Vertex buffer

Models
- Model
  - Array of mesh indices
  - Array of material indices

Materials
- Model
  - Material
    - Roughness
    - Intensity
    - Surface texture
    - Specular texture
    - Pipeline state object
Scene Object Model with Argument Buffers

Mesh argument buffer
- Mesh 0
- Mesh 1
- Mesh 2
- ...

Material argument buffer
- Material 0
- Material 1
- Material 2
- ...

Model argument buffer
- Model 0
- Model 1
- Model 2
- ...

Scene argument buffer
- Meshes
- Materials
- Models
// Accessing Argument Buffers in a Shader

struct Material
{
    float                 roughness;
    float                 intensity;
    texture2d<float>      surfaceTexture;
    texture2d<float>      specularTexture;
    render_pipeline_state pipelineState;
    ...
};

struct Scene
{
    device Mesh           *meshes;
    device Material       *materials;
    device Model          *models;
    ...
};
// Accessing Argument Buffers in a Shader

struct Material
{
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    render_pipeline_state pipelineState;
    ...
};

struct Scene
{
    device Mesh *meshes;
    device Material *materials;
    device Model *models;
    ...
};
// Accessing Argument Buffers in a Shader

struct Material
{
  float roughness;
  float intensity;
  texture2d<float> surfaceTexture;
  texture2d<float> specularTexture;
  render_pipeline_state pipelineState;
  ...
};

struct Scene
{
  device Mesh *meshes;
  device Material *materials;
  device Model *models;
  ...
};
// Accessing Argument Buffers in a Shader

struct Material
{
    float roughness;
    float intensity;
    texture2d<float> surfaceTexture;
    texture2d<float> specularTexture;
    render_pipeline_state pipelineState;
    ...
};

struct Scene
{
    device Mesh *meshes;
    device Material *materials;
    device Model *models;
    ...
};
kernel void EncodeDraw(device Scene &scene         [[buffer(0)]],
   device CommandArgs &cmd_args  [[buffer(1)]],
   const uint draw_id         [[thread_position_in_grid]])
{
    // Get the model for the object being processed
    device Model* model = scene.models[draw_id];

    // See if it is frustum culled ...
    bool culled = isFrustumCulled(model, ...);

    // If the model is not culled ...
    if (!culled)
    {
        // Get the lod to use for the model
        uint lod = getLOD(model, ...);
kernel void EncodeDraw(device Scene &scene [[buffer(0)]],
                      device CommandArgs &cmd_args [[buffer(1)]],
                      const uint draw_id [[thread_position_in_grid]])
{
  // Get the model for the object being processed
  device Model* model = scene.models[draw_id];

  // See if it is frustum culled ...
  bool culled = isFrustumCulled(model, ...);

  // If the model is not culled ...
  if (!culled)
  {
    // Get the lod to use for the model
    uint lod = getLOD(model, ...);
kernel void EncodeDraw(device Scene &scene, device CommandArgs &cmd_args, const uint draw_id) {
    // Get the model for the object being processed
    device Model* model = scene.models[draw_id];

    // See if it is frustum culled ...
    bool culled = isFrustumCulled(model, ...);

    // If the model is not culled ...
    if (!culled)
    {
        // Get the lod to use for the model
        uint lod = getLOD(model, ...);
kernel void EncodeDraw(device Scene &scene [[buffer(0)]],
                        device CommandArgs &cmd_args [[buffer(1)]],
                        const uint draw_id [[thread_position_in_grid]])
{
    // Get the model for the object being processed
    device Model* model = scene.models[draw_id];

    // See if it is frustum culled ...
    bool culled = isFrustumCulled(model, ...);

    // If the model is not culled ...
    if (!culled)
    {
        // Get the lod to use for the model
        uint lod = getLOD(model, ...);
    }
}
kernel void EncodeDraw(device Scene &scene           
    [[buffer(0)]],
    device CommandArgs &cmd_args [[buffer(1)]],
    const uint draw_id           [[thread_position_in_grid]])
{
    // Get the model for the object being processed
    device Model* model = scene.models[draw_id];

    // See if it is frustum culled ...
    bool culled = isFrustumCulled(model, ...);

    // If the model is not culled ...
    if (!culled)
    {
        // Get the lod to use for the model
        uint lod = getLOD(model, ...);
kernel void EncodeDraw(device Scene &scene          
    device CommandArgs &cmd_args            
    const uint draw_id)
{
    // Get the model for the object being processed
    device Model* model = scene.models[draw_id];

    // See if it is frustum culled ...
    bool culled = isFrustumCulled(model, ...);

    // If the model is not culled ...
    if (!culled)
    {
        // Get the lod to use for the model
        uint lod = getLOD(model, ...);
// Get the mesh for the lod from mesh indices
uint mesh_id = model->meshIndices[lod];
Mesh *mesh = scene.meshes[mesh_id];

// Get the material for the lod from material indices
uint material_id = scene.models->materialIndices[lod];
Material *material = scene.materials[material_id];

...
// Get the mesh for the lod from mesh indices
uint mesh_id = model->meshIndices[lod];
Mesh *mesh = scene.meshes[mesh_id];

// Get the material for the lod from material indices
uint material_id = scene.models->materialIndices[lod];
Material *material = scene.materials[material_id];

...
Encoding Draw Commands

Scene

Object 0

Object 1

Object N

System memory

Indirect command buffer

Cmd 0
- Set render pipeline state
- Set vertex buffer
- Set fragment buffer
- Draw primitives

Cmd 1
- Set render pipeline state
- Set vertex buffer
- Set fragment buffer
- Draw primitives

Cmd N
- Set render pipeline state
- Set vertex buffer
- Set fragment buffer
- Draw primitives
... 

// Get a command to encode from the indirect command buffer
render_command cmd(cmd_args.cmd_buffer, draw_id);

// Encode PSO, required vertex buffers and the material
cmd.set_render_pipeline_state(material->pipelineState);
cmd.set_vertex_buffer(getMeshUniforms(mesh), 0);
cmd.set_vertex_buffer(getMeshVertexData(mesh), 1);
cmd.set_fragment_buffer(material, 0);

// Encode the draw
cmd.draw_indexed_primitives(primitive_type::triangle, ...);
// Get a command to encode from the indirect command buffer
render_command cmd(cmd_args.cmd_buffer, draw_id);

// Encode PSO, required vertex buffers and the material
cmd.set_render_pipeline_state(material->pipelineState);
cmd.set_vertex_buffer(getMeshUniforms(mesh), 0);
cmd.set_vertex_buffer(getMeshVertexData(mesh), 1);
cmd.set_fragment_buffer(material, 0);

// Encode the draw
cmd.draw_indexed_primitives(primitive_type::triangle, ...);
// Get a command to encode from the indirect command buffer
render_command cmd(cmd_args.cmd_buffer, draw_id);

// Encode PSO, required vertex buffers and the material
cmd.set_render_pipeline_state(material->pipelineState);
cmd.set_vertex_buffer(getMeshUniforms(mesh), 0);
cmd.set_vertex_buffer(getMeshVertexData(mesh), 1);
cmd.set_fragment_buffer(material, 0);

// Encode the draw
cmd.draw_indexed_primitives(primitive_type::triangle, ...);
// Get a command to encode from the indirect command buffer
render_command cmd(cmd_args.cmd_buffer, draw_id);

// Encode PSO, required vertex buffers and the material
    cmd.set_render_pipeline_state(material->pipelineState);
    cmd.set_vertex_buffer(getMeshUniforms(mesh), 0);
    cmd.set_vertex_buffer(getMeshVertexData(mesh), 1);
    cmd.set_fragment_buffer(material, 0);

// Encode the draw
    cmd.draw_indexed_primitives(primitive_type::triangle, ...);
// Executing the ICB

// Create an indirect command buffer for the occluder draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw]
let icb_occluders = device.newIndirectCommandBuffer(with: desc, maxCommandCount: count, ...)!

// Compute pass for frustum culling occluders, encoding occluder draws into the ICB
computeEncoder.dispatchThreadgroups(...)

// Optimize the Indirect Command Buffer
blitEncoder.optimizeCommandsInBuffer(icb_occluders, with: NSRange(0, count))

// Execute icb_occluders for generating occluder data
renderEncoder.executeCommandsInBuffer(in: icb_occluders, withRange: NSMakeRange(0, count))
// Executing the ICB

// Create an indirect command buffer for the occluder draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw]
let icb_occluders = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, ...)!

// Compute pass for frustum culling occluders, encoding occluder draws into the ICB
computeEncoder.dispatchThreadgroups(...)

// Optimize the Indirect Command Buffer
blitEncoder.optimizeCommandsInBuffer(icb_occluders, with: NSRange(0, count))

// Execute icb_occluders for generating occluder data
renderEncoder.executeCommandsInBuffer(in: icb_occluders, withRange:NSMakeRange(0, count))
// Executing the ICB

// Create an indirect command buffer for the occluder draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw]
let icb_occluders = device.newIndirectCommandBuffer(with: desc, maxCommandCount: count, ...)!

// Compute pass for frustum culling occluders, encoding occluder draws into the ICB
computeEncoder.dispatchThreadgroups(...)

// Optimize the Indirect Command Buffer
blitEncoder.optimizeCommandsInBuffer(icb_occluders, with: NSRange(0, count))

// Execute icb_occluders for generating occluder data
renderEncoder.executeCommandsInBuffer(in: icb_occluders, withRange:NSMakeRange(0, count))
// Executing the ICB

// Create an indirect command buffer for the occluder draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw]
let icb_occluders = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, ...)!

// Compute pass for frustum culling occluders, encoding occluder draws into the ICB
computeEncoder.dispatchThreadgroups(...)

// Optimize the Indirect Command Buffer
blitEncoder.optimizeCommandsInBuffer(icb_occluders, with: NSRange(0, count))

// Execute icb_occluders for generating occluder data
renderEncoder.executeCommandsInBuffer(in: icb_occluders, withRange:NSMakeRange(0, count))
/* Executing the ICB */

/* Create an indirect command buffer for the occluder draws */
let desc = MLTIIndirectCommandBufferDescriptor()
desc.commandTypes = [.draw]
let icb_occluders = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, ...)

/* Compute pass for frustum culling occluders, encoding occluder draws into the ICB */
computeEncoder.dispatchThreadgroups(...)

/* Optimize the Indirect Command Buffer */
blitEncoder.optimizeCommandsInBuffer(icb_occluders, with: NSRange(0, count))

/* Execute icb_occluders for generating occluder data */
renderEncoder.executeCommandsInBuffer(in: icb_occluders, withRange:NSMakeRange(0, count))
// Compute dispatch for additional occluder data
computeEncoder.dispatchThreadgroups(...)

// Create an indirect command buffer for scene execution draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encoding draws
computeEncoder.dispatchThreadgroups(...)

// Optimize the generated indirect command buffer
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(0, count))

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, withRange:NSMakeRange(0, count))
// Compute dispatch for additional occluder data
computeEncoder.dispatchThreadgroups(...)

// Create an indirect command buffer for scene execution draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)!  

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encoding draws
computeEncoder.dispatchThreadgroups(...)

// Optimize the generated indirect command buffer
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(0, count))

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, withRange:NSMakeRange(0, count))
// Compute dispatch for additional occluder data
computeEncoder.dispatchThreadgroups(...)

// Create an indirect command buffer for scene execution draws
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)!

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encoding draws
computeEncoder.dispatchThreadgroups(...)

// Optimize the generated indirect command buffer
blitEncoder.optimizeCommandsInBuffer(icb, with: NSRange(0, count))

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, withRange:NSMakeRange(0, count))
Sparse Encoding

System memory

Indirect command buffer

Draw 0  Draw 2  Draw 4

Scene

Object 0  Object 1  Object 2  Object 3  Object 4  ...  Object N
Packed Encoding

System memory

Indirect command buffer

Scene

Object 0 Object 1 Object 2 Object 3 Object 4 ... Object N
Indirect Range Execution

- Indirect range buffer
  - location
  - length

- System memory
  - length

- Indirect command buffer

- Scene
  - Object 0
  - Object 1
  - Object 2
  - Object 3
  - Object 4
  - ... Object N
kernel void EncodeDraw(device Scene &scene [[buffer(0)]],
    device CommandArgs &cmd_args [[buffer(1)]],
    device atomic_uint *range [[buffer(2)]],
    const uint draw_id [[thread_position_in_grid]])
{
    // Get model for the object being processed
    device Model* model = scene.models[draw_id];
    ...

    // If the model is not culled...
    if (!culled)
    {
        ...
        // Get a command to encode from the indirect command buffer.
        // atomic add increments the range of the commands to execute
        render_command cmd(cmd_args.cmd_buffer, atomic_fetch_add_explicit(range, 1, ...));
        ...
    }
kernel void EncodeDraw(device Scene &scene [[buffer(0)]],
    device CommandArgs &cmd_args [[buffer(1)]],
    device atomic_uint *range [[buffer(2)]],
    const uint draw_id [[thread_position_in_grid]])
{
    // Get model for the object being processed
    device Model* model = scene.models[draw_id];
    ...

    // If the model is not culled ...
    if (!culled)
    {
        ...
        // Get a command to encode from the indirect command buffer.
        // atomic add increments the range of the commands to execute
        render_command cmd(cmd_args.cmd_buffer, atomic_fetch_add_explicit(range, 1, ...));
        ...
    }
kernel void EncodeDraw(device Scene &scene          
device CommandArgs &cmd_args          
device atomic_uint *range          
const uint draw_id          
{
    // Get model for the object being processed
    device Model* model = scene.models[draw_id];
    ...

    // If the model is not culled ...
    if (!culled)
    {
        ...
        // Get a command to encode from the indirect command buffer.
        // atomic add increments the range of the commands to execute
        render_command cmd(cmd_args.cmd_buffer, atomic_fetch_add_explicit(range, 1, ...));
        ...
    }
kernel void EncodeDraw(device Scene &scene          
    device CommandArgs &cmd_args [[buffer(1)]],
    device atomic_uint *range    [[buffer(2)]],
    const uint draw_id           [[thread_position_in_grid]])
{
    // Get model for the object being processed
    device Model* model = scene=models[draw_id];
    ...

    // If the model is not culled ...
    if (!culled)
    {
        ...
        // Get a command to encode from the indirect command buffer.
        // atomic add increments the range of the commands to execute
        render_command cmd(cmd_args.cmd_buffer, atomic_fetch_add_explicit(range, 1, ...));
        ...
    }
// Executing ICB with Indirect Range

// Create an Indirect Command Buffer for scene execution
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount: count, options...)!

// Create indirect range buffer
let rangeBuffer = device.makeBuffer(...);

// Set the rangeBuffer as the kernel argument
computeEncoder.setBuffer(rangeBuffer, ...)

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encode draws
computeEncoder.dispatchThreadgroups(...)

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer: rangeBuffer, ...)
// Executing ICB with Indirect Range

// Create an Indirect Command Buffer for scene execution
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)!

// Create indirect range buffer
let rangeBuffer = device.makeBuffer(...);

// Set the rangeBuffer as the kernel argument
computeEncoder.setBuffer(rangeBuffer, ...)

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encode draws
computeEncoder.dispatchThreadgroups(...)

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer:rangeBuffer, ...)
// Executing ICB with Indirect Range

// Create an Indirect Command Buffer for scene execution
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)
!

// Create indirect range buffer
let rangeBuffer = device.makeBuffer(...);

// Set the rangeBuffer as the kernel argument
computeEncoder.setBuffer(rangeBuffer, ...)

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encode draws
computeEncoder.dispatchThreadgroups(...)

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer:rangeBuffer, ...)
// Executing ICB with Indirect Range

// Create an Indirect Command Buffer for scene execution
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)

// Create indirect range buffer
let rangeBuffer = device.makeBuffer(...);

// Set the rangeBuffer as the kernel argument
computeEncoder.setBuffer(rangeBuffer, ...)

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encode draws
computeEncoder.dispatchThreadgroups(...)

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer:rangeBuffer, ...)
// Executing ICB with Indirect Range

// Create an Indirect Command Buffer for scene execution
let desc = MLTIndirectCommandBufferDescriptor()
desc.commandTypes = [.drawIndexed]
let icb = device.newIndirectCommandBuffer(with: desc, maxCommandCount:count, options...)

// Create indirect range buffer
let rangeBuffer = device.makeBuffer(...);

// Set the rangeBuffer as the kernel argument
computeEncoder.setBuffer(rangeBuffer, ...)

// Compute dispatch for frustum culling, LOD selection, occlusion culling and encode draws
computeEncoder.dispatchThreadgroups(...)

// Execute the draws for the scene
renderEncoder.executeCommandsInBuffer(in: icb, indirectBuffer:rangeBuffer, ...)
Encoding Compute Dispatches

Allows GPU to build compute dispatches

Compute ICBs can be reused or modified

Both render and compute can now be driven on the GPU
Per Patch Tessellation Factors

GPU thread 0

- Frustum cull
  - Object 0

- Select LOD
  - Object 0

- Occlusion cull
  - Object 0

- Process patches
  - Patch 0
    - Object 0
  - Patch 1
    - Object 0
  - ... (up to Patch N)
    - Object 0

...
Per Patch Tessellation Factors

Culling dispatch

Tessellation factor generation dispatch

GPU thread 0

GPU thread 1

GPU thread N

GPU thread N

Frustum cull

Select LOD

Occlusion cull

Encode tessfactor dispatch

Process Patch 0

Process Patch 1

Process Patch N
Per Patch Tessellation Factors

Current frame

Compute pass
- Frustum culling
- LOD selection
- Occlusion culling
- Encode draws

Render pass
- Render scene

GPU
... System memory

Scene data
Occluder data
Scene draw commands
... Draws
Per Patch Tessellation Factors

Current frame

GPU

Compute pass

Frustum culling
LOD selection
Occlusion culling
Encode tessfactor generation
Encode draws

Compute pass

Generate tessfactors

Render pass

Render scene

System memory

Occluder data
Tessfactor generation commands
Tessfactors

Scene draw commands

Scene data

Per Patch Tessellation Factors

Current frame

GPU

Compute pass

Frustum culling
LOD selection
Occlusion culling
Encode tessfactor generation
Encode draws

Compute pass

Generate tessfactors

Render pass

Render scene

System memory

Occluder data
Tessfactor generation commands
Tessfactors

Scene draw commands

Scene data
Simpler GPU Families
Metal Features Available on All Platforms
Metal Features Available on All Platforms

New to iOS and tvOS

• Setting pipeline states on Indirect Command Buffers
• Sourcing Indirect Command Buffer ranges from buffers
• 16-bit depth textures
Metal Features Available on All Platforms

New to iOS and tvOS
• Setting pipeline states on Indirect Command Buffers
• Sourcing Indirect Command Buffer ranges from buffers
• 16-bit depth textures

New to macOS
• Rendering without render pass attachments
• Command buffer timing
• Casting between sRGB and non-sRGB texture views
GPU Family

Replaces Metal Feature Set queries

Easier to query GPU capabilities

• 4 families, focus on cross-platform commonality
• Hierarchical feature instances within each family
• Separate version queries
• Few device queries for optional features
<table>
<thead>
<tr>
<th>Apple Family</th>
<th>GPU</th>
<th>iPhone</th>
<th>iPad</th>
<th>Apple TV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple 1</td>
<td>Apple A7</td>
<td>iPhone 5s</td>
<td>iPad mini 3, iPad Air</td>
<td></td>
</tr>
<tr>
<td>Apple 2</td>
<td>Apple A8</td>
<td>iPhone 6, iPhone 6 Plus, iPod Touch</td>
<td>iPad mini 4, iPad Air 2</td>
<td>Apple TV</td>
</tr>
<tr>
<td>Apple 3</td>
<td>Apple A9</td>
<td>iPhone 6S, iPhone 6S Plus, iPhone 7, iPhone 7 Plus, iPhone SE</td>
<td>iPad Pro (2015), iPad Pro (2016), iPad (2018), iPad Pro (2017)</td>
<td>Apple TV 4K</td>
</tr>
<tr>
<td></td>
<td>Apple A10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple 4</td>
<td>Apple A11</td>
<td>iPhone 8, iPhone 8 Plus, iPhone X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple 5</td>
<td>Apple A12</td>
<td>iPhone XS, iPhone XS Max, iPhone XR</td>
<td>iPad mini (2019), iPad Air (2019), iPad Pro (2018)</td>
<td></td>
</tr>
</tbody>
</table>
## Mac Family

<table>
<thead>
<tr>
<th>GPU</th>
<th>Mac</th>
<th>MacBook</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mac 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nvidia GeForce GT</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mac 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Iris Plus Graphics 6xx</td>
<td>iMac Pro (2017+)</td>
<td></td>
</tr>
<tr>
<td>AMD Radeon, Radeon Pro, FirePro</td>
<td>Mac Pro (late 2013)</td>
<td></td>
</tr>
</tbody>
</table>
### Common Family

<table>
<thead>
<tr>
<th>Common 1</th>
<th>Universally supported features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common 2</td>
<td>Indirect Draw/Dispatch, Counting Occlusion Queries, Tessellation, Read/Write Buffer Arguments, Arrays of Textures/Samplers, Compressed Volume Textures, Metal Performance Shaders, and more</td>
</tr>
<tr>
<td>Common 3</td>
<td>Stencil Feedback, MSAA Depth/Stencil Resolve, Programmable Sample Positions, Invariant Vertex Position, Indirect Stage-In, Indirect Command Buffers, Quad-scoped Shuffle/Broadcast, Cube Texture Arrays, Read/Write Texture Arguments, Attachment-less Render Passes, Layered Rendering, Multi-Viewport Rendering, Argument Buffers, Pipelined Compute, Indirect Tessellation, Heap Placement, Texture Swizzle, and more</td>
</tr>
</tbody>
</table>
## iPad Apps for Mac Family

<table>
<thead>
<tr>
<th>iPad Apps for Mac Family</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOSMac 1</td>
<td>Common 2, BC Pixel Formats, Managed Textures, Cube Texture Arrays, Read/Write Textures Tier 1, Layered Rendering, Multiple Viewports/Scissors, Indirect Tessellation</td>
</tr>
<tr>
<td>iOSMac 2</td>
<td>Common 3, BC Pixel Formats, Managed Textures</td>
</tr>
</tbody>
</table>
// Determine if Mac2 family features are available

if #available(macOS 10.15, iOS 13, tvOS 13, *) {
    if self.device.supportsVersion(.version3_0) {
        if self.device.supportsFamily(.familyMac2) {
            // Enable features that require Mac family 2 as defined by Metal 3
        }
    }
} else {
    // Fallback on earlier Metal versions
}

} else {
    // Fallback to feature set API on earlier OS versions
    if self.device.supportsFeatureSet(.featureSet_macOS_GPUFamily2_v1) {
        ...
    }
}
// Determine if Mac2 family features are available

if #available(macOS 10.15, iOS 13, tvOS 13, *) {
    if self.device.supportsVersion(.version3_0) {
        if self.device.supportsFamily(.familyMac2) {
            // Enable features that require Mac family 2 as defined by Metal 3
        }
    }
    else {
        // Fallback on earlier Metal versions
    }
} else {
    // Fallback to feature set API on earlier OS versions
    if self.device.supportsFeatureSet(.featureSet_macOS_GPUFamily2_v1) {
        ...
    }
}
// Determine if Mac2 family features are available

if #available(macOS 10.15, iOS 13, tvOS 13, *) {
    if self.device.supportsVersion(.version3_0) {
        if self.device.supportsFamily(.familyMac2) {
            // Enable features that require Mac family 2 as defined by Metal 3
        }
    }
    else {
        // Fallback on earlier Metal versions
    }
} else {
    // Fallback to feature set API on earlier OS versions
    if self.device.supportsFeatureSet(.featureSet_macOS_GPUFamily2_v1) {
        ...
    }
}
// Determine if Mac2 family features are available

if #available(macOS 10.15, iOS 13, tvOS 13, *) {
    if self.device.supportsVersion(.version3_0) {
        if self.device.supportsFamily(.familyMac2) {
            // Enable features that require Mac family 2 as defined by Metal 3
        }
    }
}
else {
    // Fallback on earlier Metal versions
}
} else {
    // Fallback to feature set API on earlier OS versions
    if self.device.supportsFeatureSet(.featureSet_macOS_GPUFamily2_v1) {
        ...
    }
}
// Determine if Mac2 family features are available

if #available(macOS 10.15, iOS 13, tvOS 13, *) {
    if self.device.supportsVersion(.version3_0) {
        if self.device.supportsFamily(.familyMac2) {
            // Enable features that require Mac family 2 as defined by Metal 3
        }
    }
} else {
    // Fallback on earlier Metal versions
}
} else {
    // Fallback to feature set API on earlier OS versions
    if self.device.supportsFeatureSet(.featureSet_macOS_GPUFamily2_v1) {
        ...
    }
}
## Optional Features and Limits

<table>
<thead>
<tr>
<th>Features</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth24Stencil8</td>
<td>MSAA Sample Counts</td>
</tr>
<tr>
<td>Raster Order Groups</td>
<td>Threadgroup Memory Length</td>
</tr>
<tr>
<td>Read-Write Texture Tier 2</td>
<td>Linear Texture Alignment</td>
</tr>
<tr>
<td>Programmable Sample Positions</td>
<td>Argument Buffer Sampler Counts</td>
</tr>
</tbody>
</table>
Rendering Technique Support by Family

Deferred Shading
  • All families
  • Programmable Blending supported by Apple 1 and later

Tile Deferred/Forward
  • Common 2 and later
  • Tile Shading supported by Apple 3 and later

Visibility Buffer
  • Mac 1 and 2
GPU-Driven Pipelines by Family

Argument Buffers
• All families

Render/Compute Indirect Command Buffers
• Common 2 and later
Summary
Summary

Optimize modern rendering techniques with Metal
• Programmable blending and tile shading on Apple family
• Barycentric coordinates and query LOD on Mac family
Summary

Optimize modern rendering techniques with Metal
• Programmable blending and tile shading on Apple family
• Barycentric coordinates and query LOD on Mac family

Move render loop from CPU to GPU
• Argument Buffers and Indirect Command Buffers
Summary

Optimize modern rendering techniques with Metal
• Programmable blending and tile shading on Apple family
• Barycentric coordinates and query LOD on Mac family

Move render loop from CPU to GPU
• Argument Buffers and Indirect Command Buffers

Choose from common and specialized features with GPU Family
More Information

developer.apple.com/wwdc19/601

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal Lab</td>
<td>Tuesday, 12:00</td>
</tr>
<tr>
<td>Metal Lab</td>
<td>Friday, 9:00</td>
</tr>
<tr>
<td>Delivering Optimized Metal Apps and Games</td>
<td>Wednesday, 11:00</td>
</tr>
</tbody>
</table>