#WWDC19

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

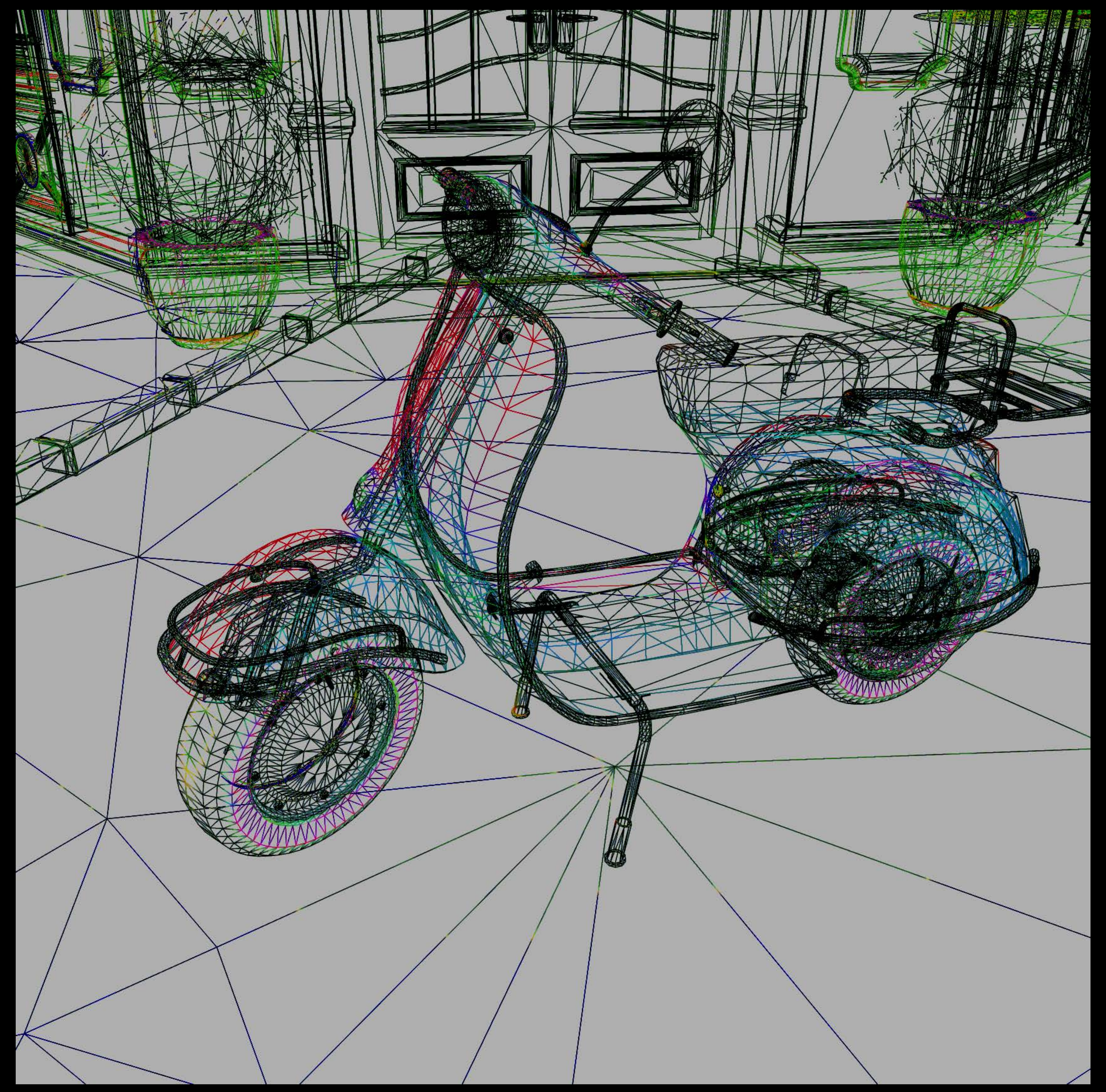
Deferred

Tiled Deferred

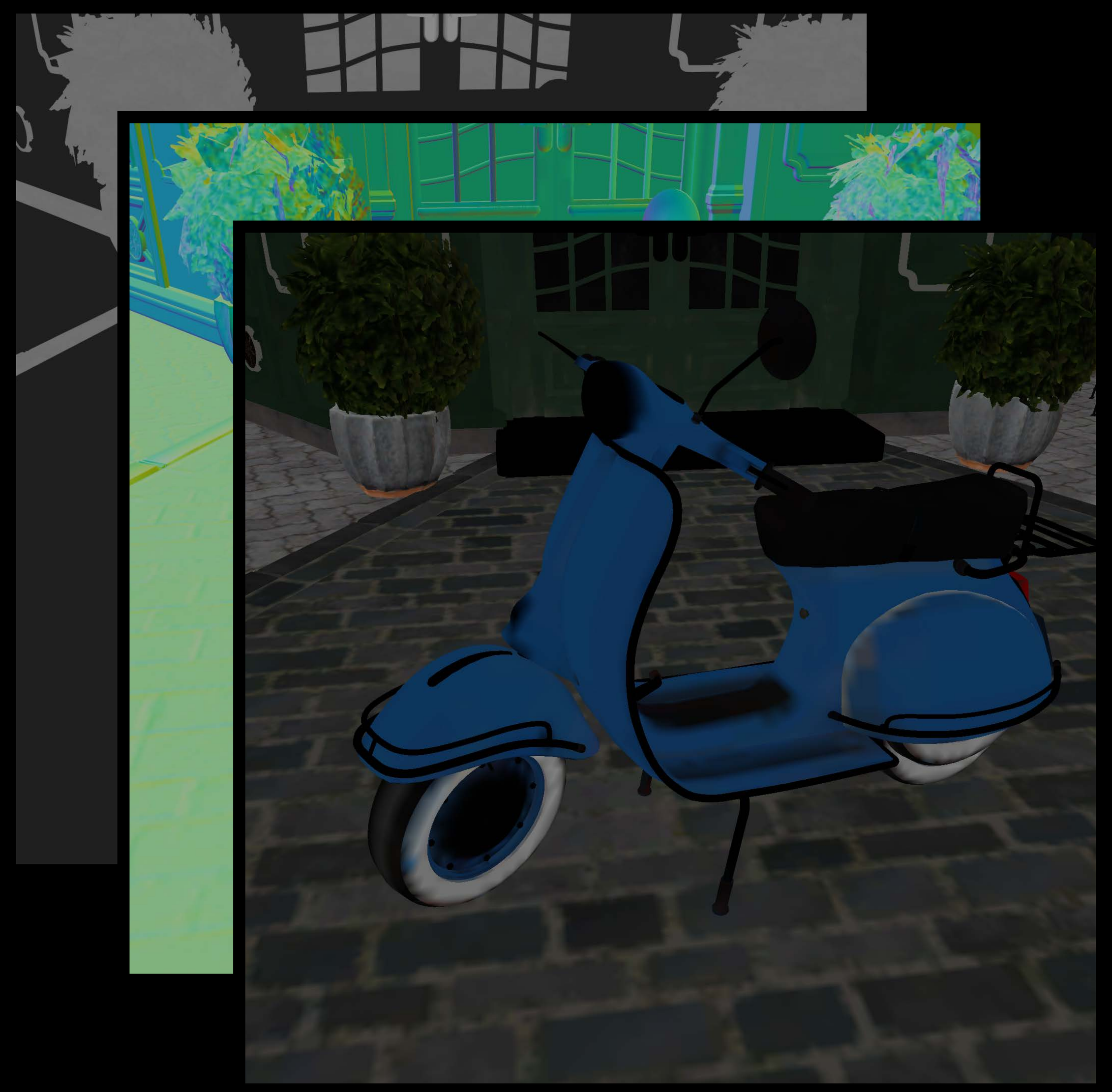
Tiled and Clustered Forward

Visibility Buffer

Deferred Rendering
2-pass approach



Geometry



Lighting

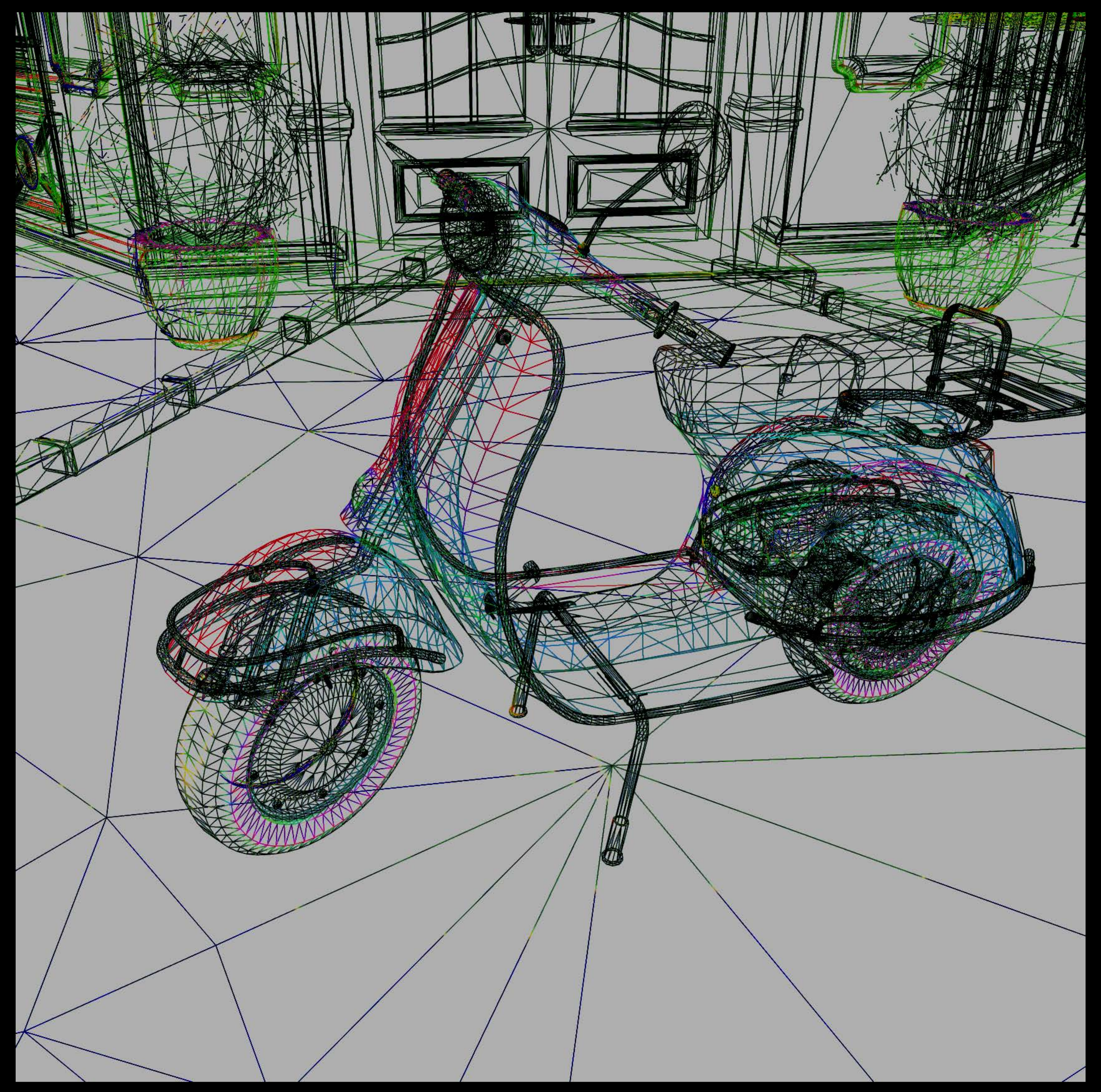
Mesh data

GBuffer

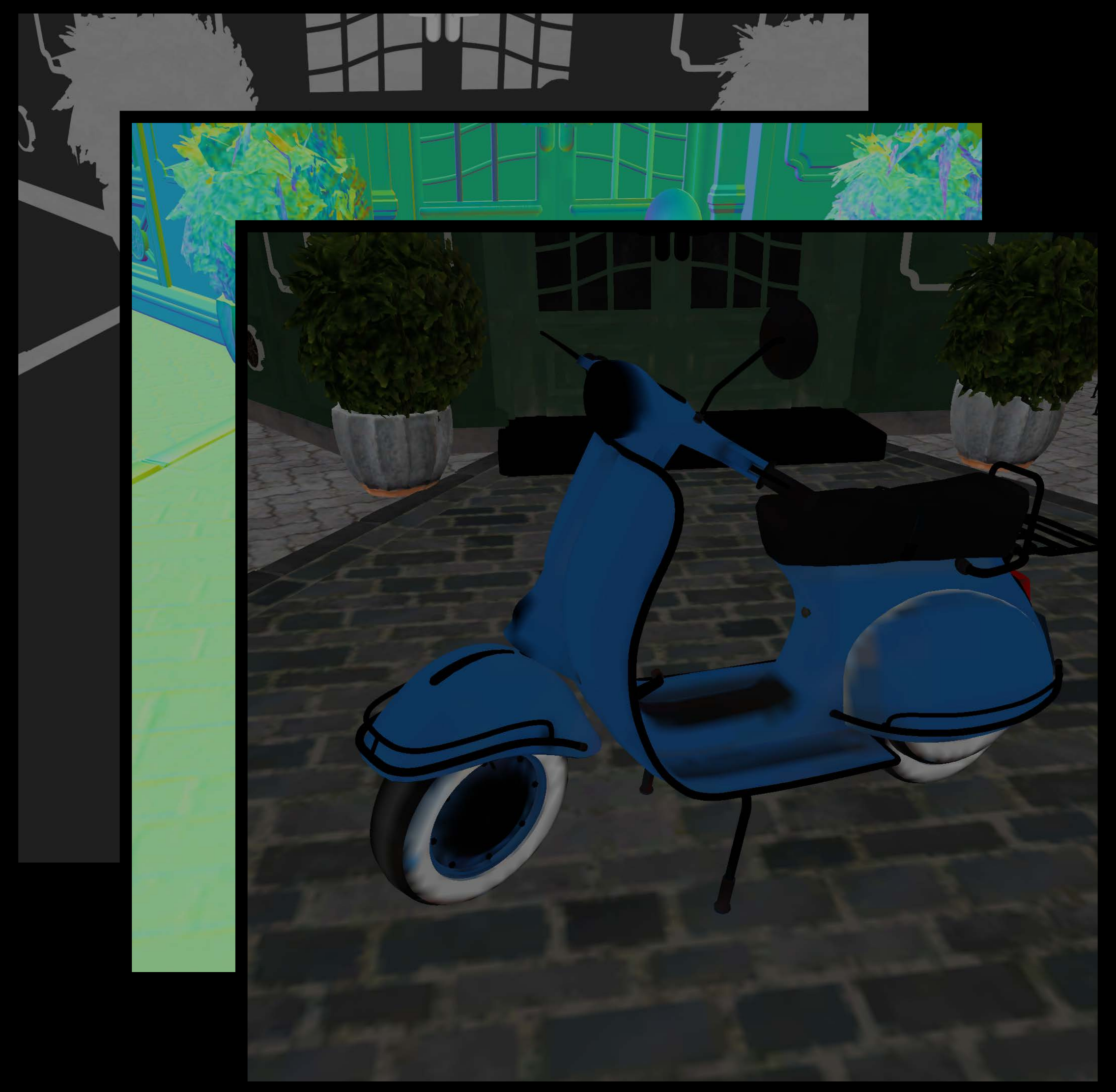
Lit scene

Deferred Rendering

2-pass approach



Geometry



Lighting

Mesh data

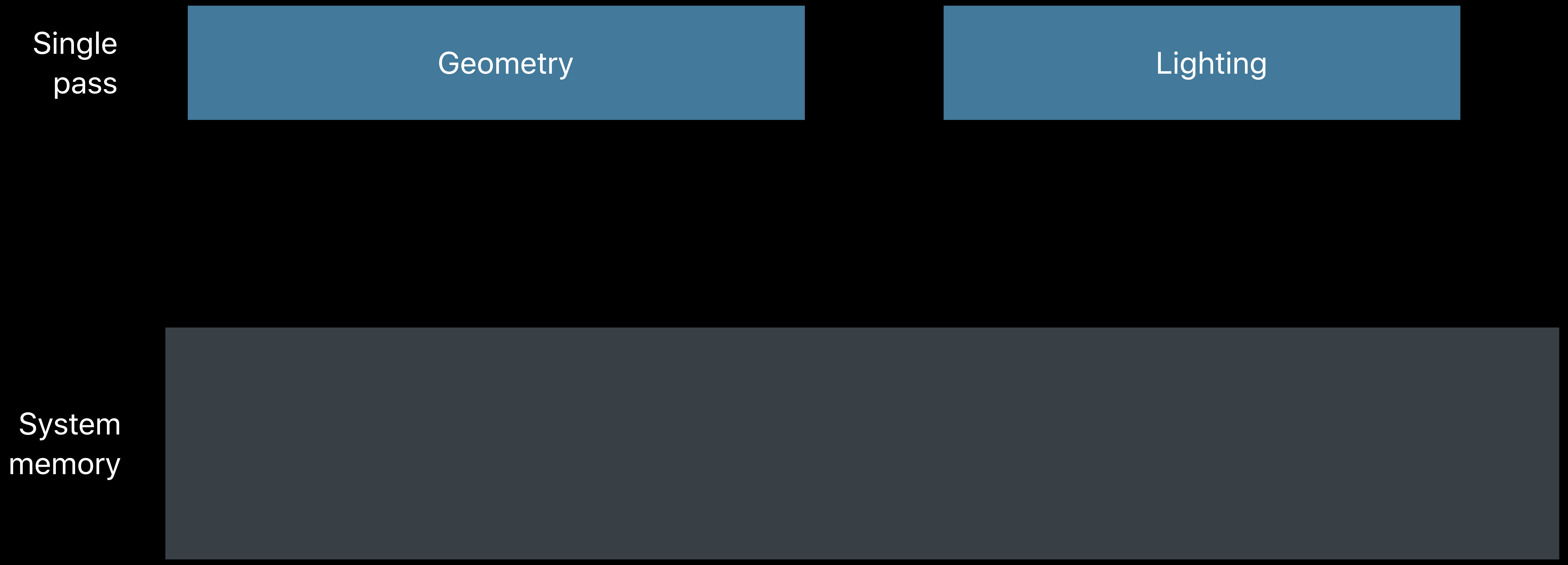
GBuffer

Lit scene

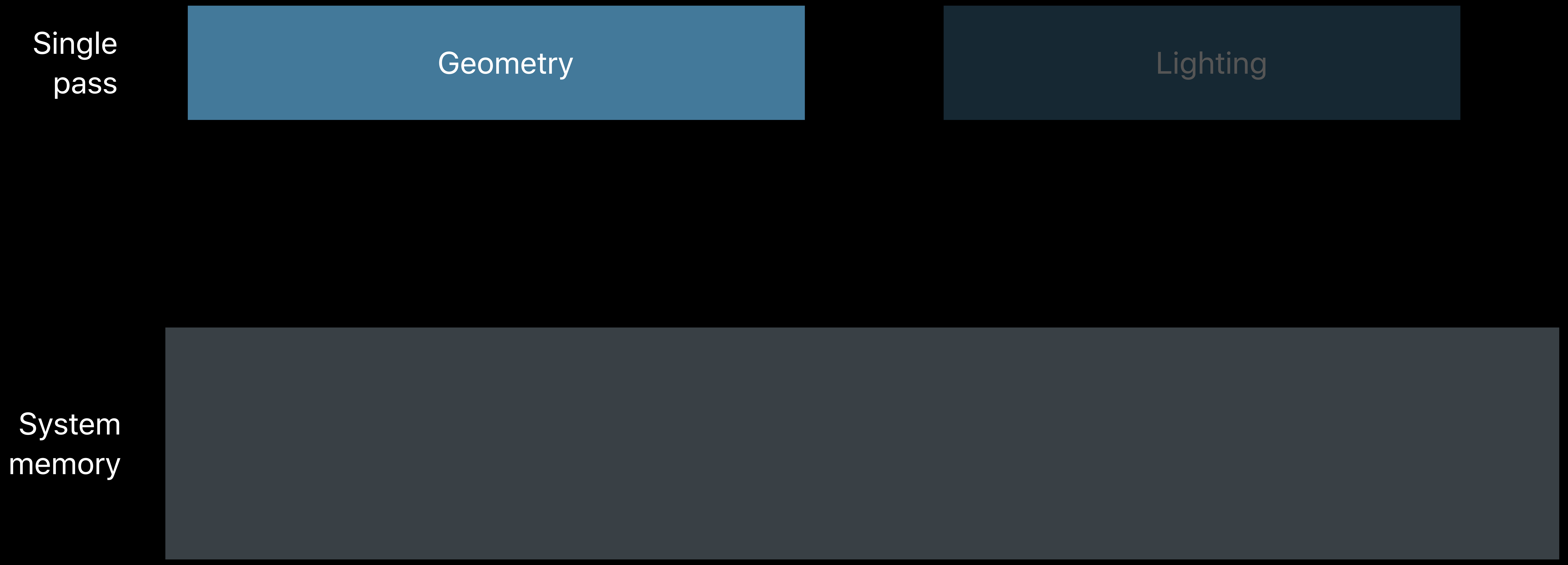
Deferred Rendering

2-pass approach

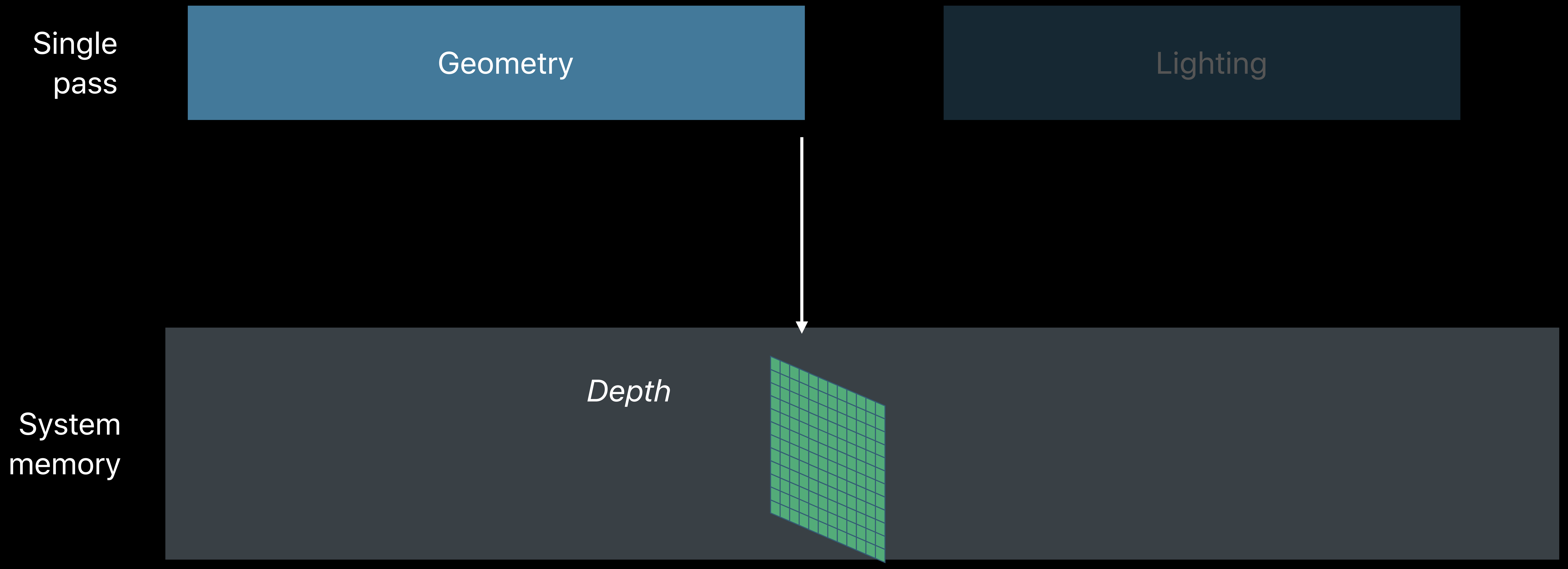
Deferred Rendering with Metal



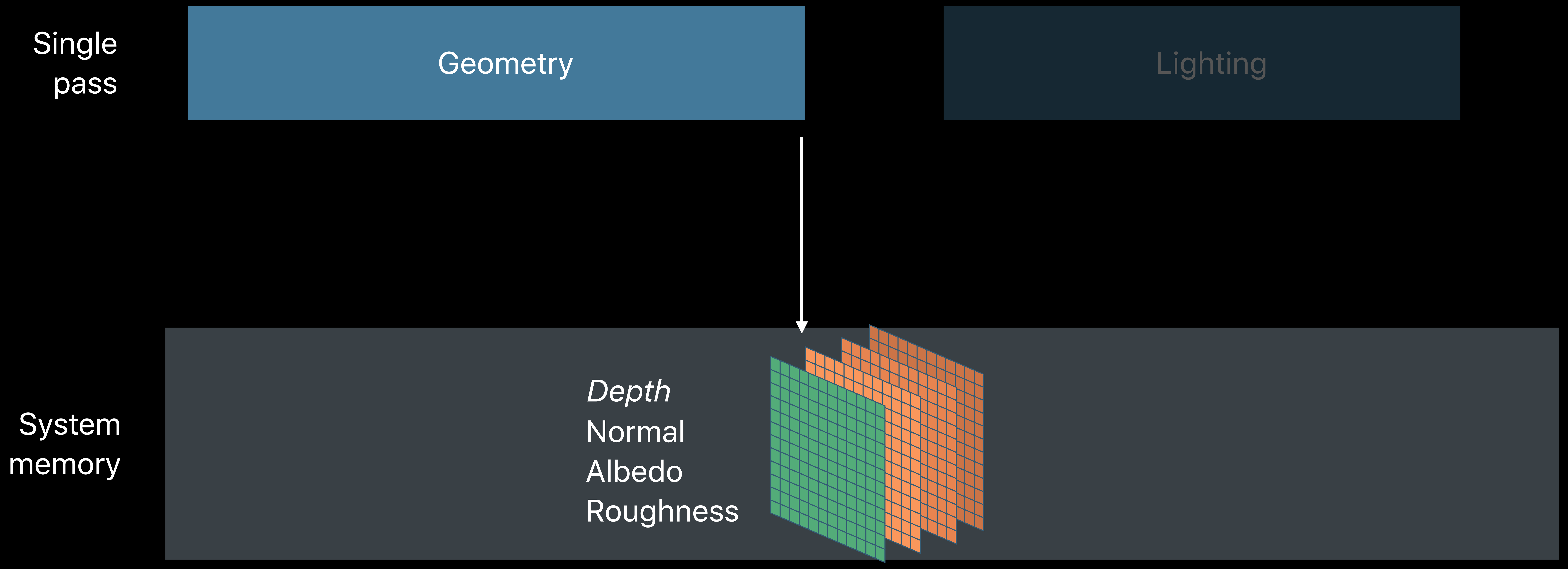
Deferred Rendering with Metal



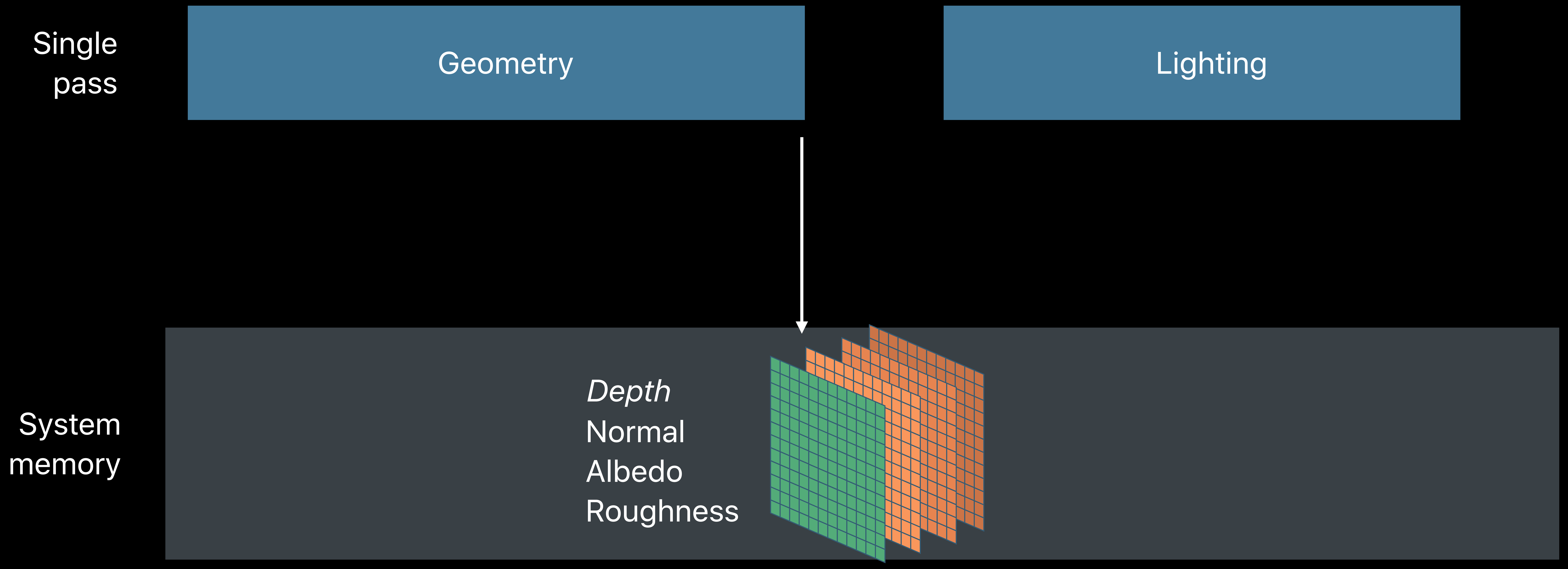
Deferred Rendering with Metal



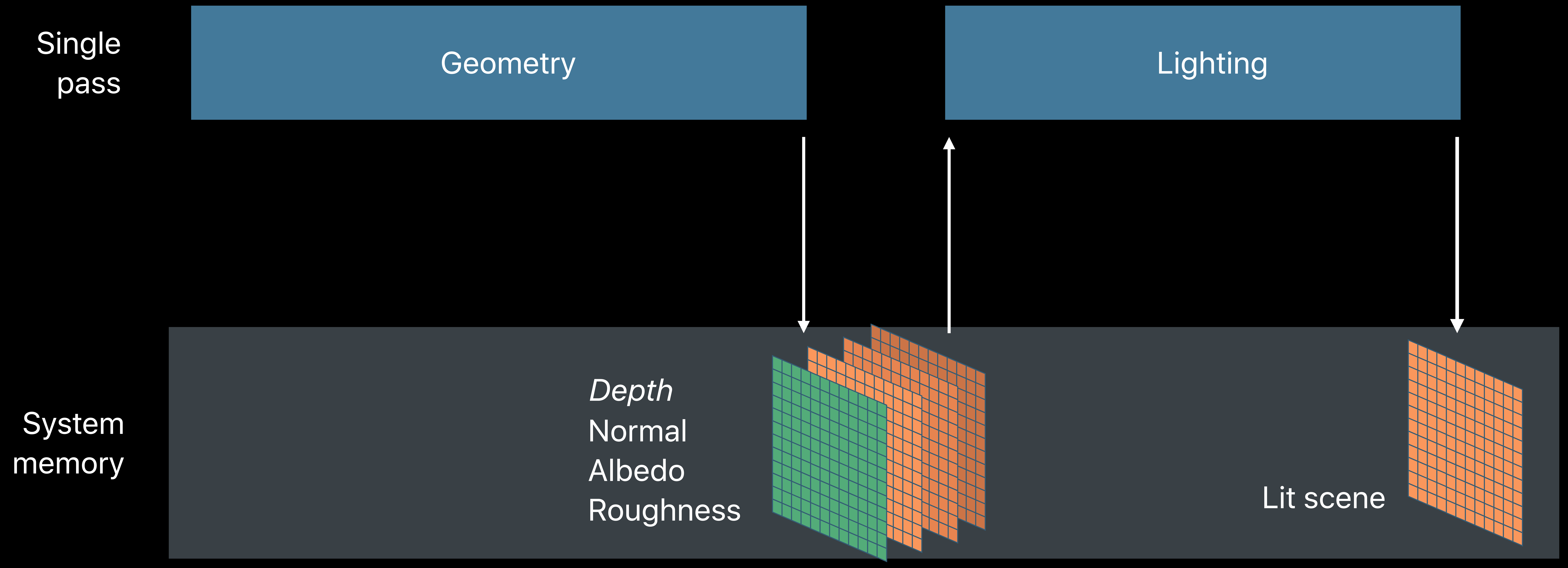
Deferred Rendering with Metal



Deferred Rendering with Metal



Deferred Rendering with Metal



Render Pass Setup

Render Pass Setup

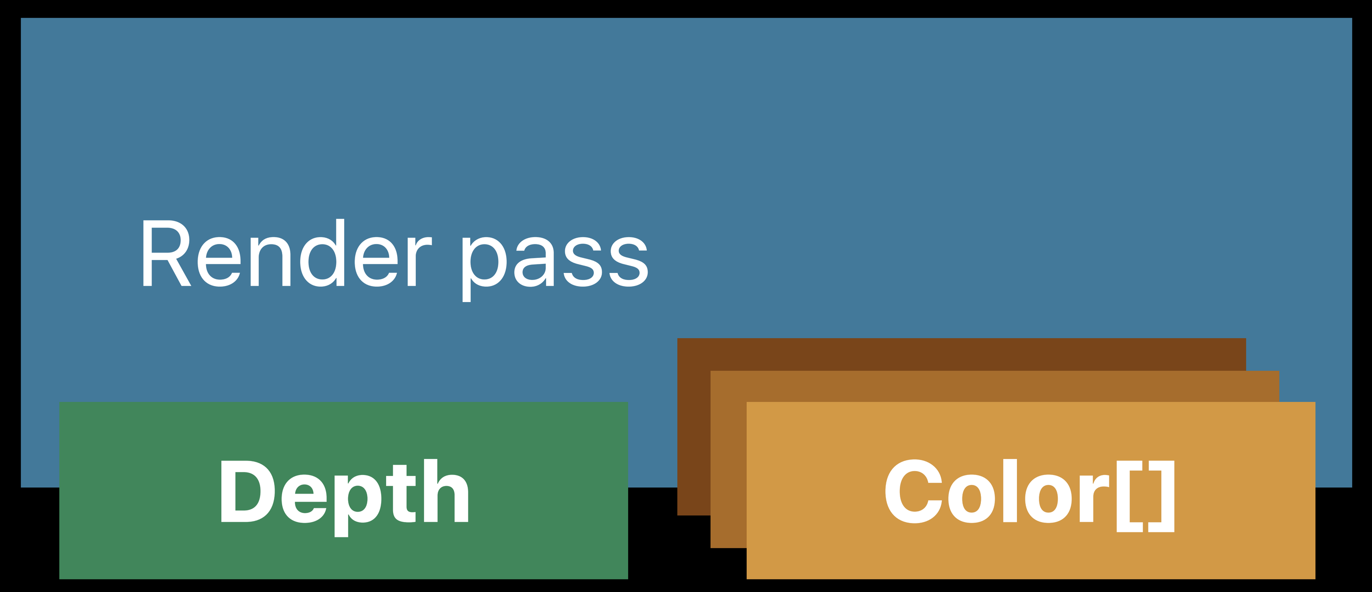
Render Pass Descriptor

Render pass

Render Pass Setup

Render Pass Descriptor

Attachments define output

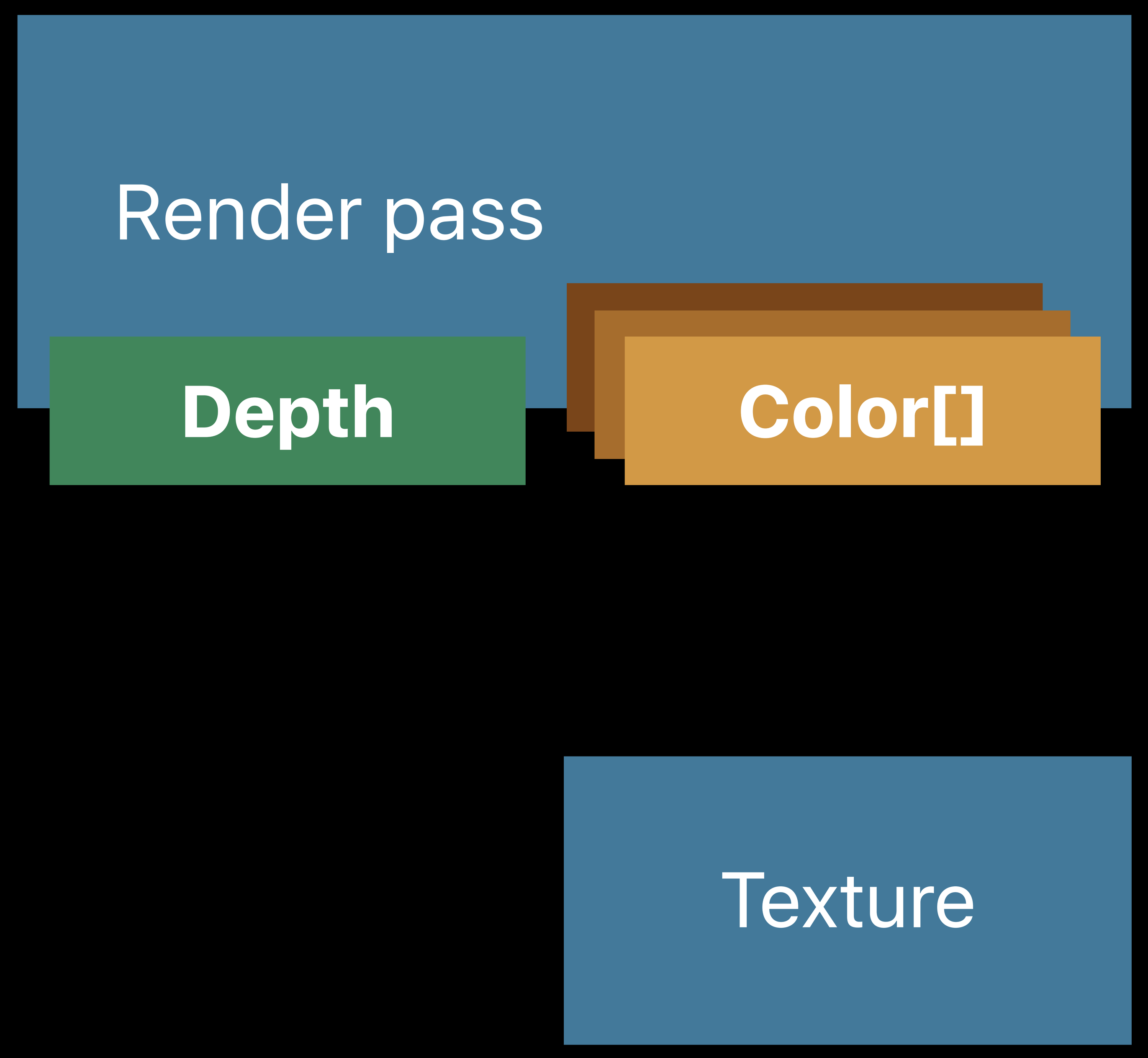


Render Pass Setup

Render Pass Descriptor

Attachments define output

- Texture: Render Target

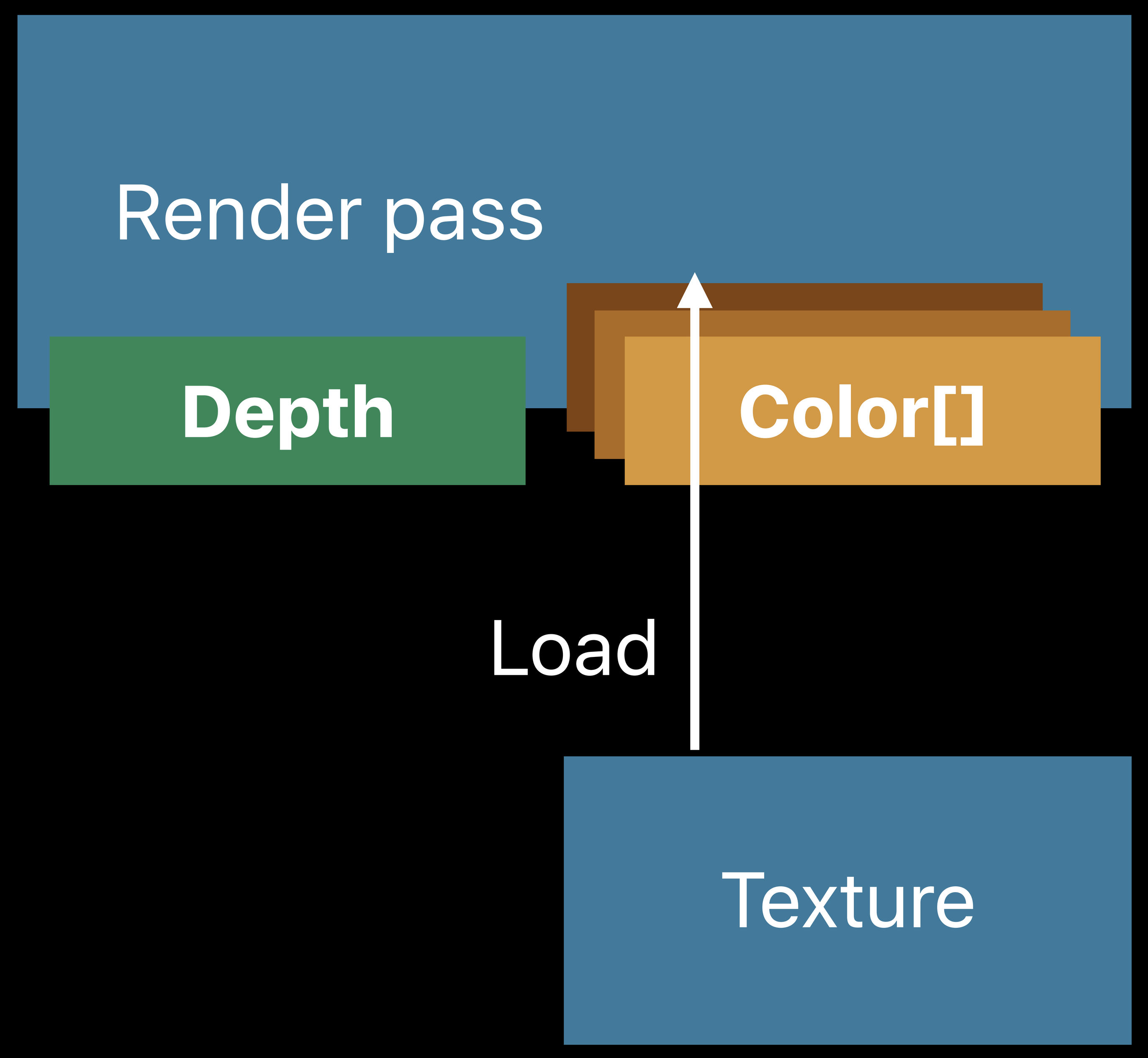


Render Pass Setup

Render Pass Descriptor

Attachments define output

- Texture: Render Target
- Load actions

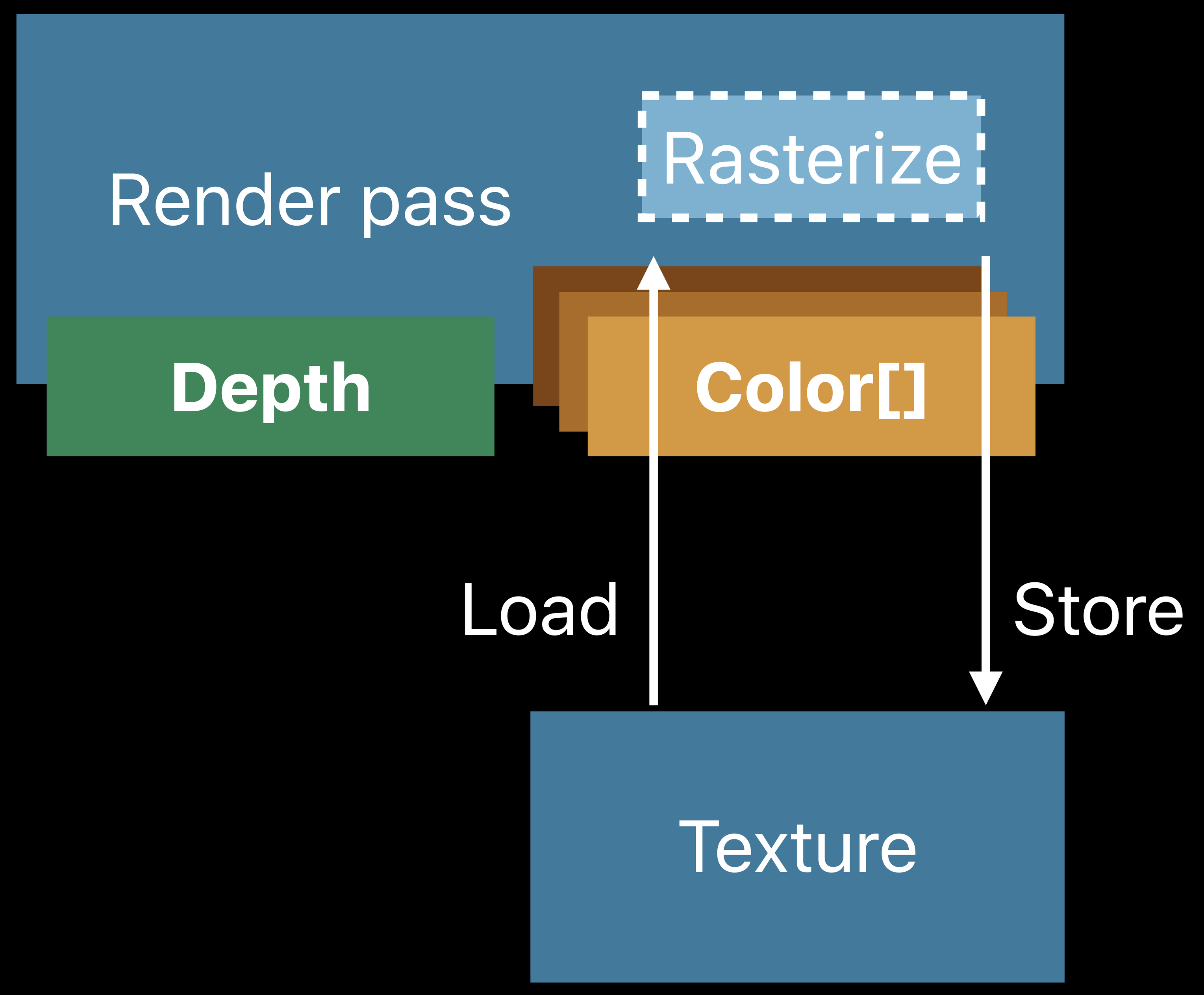


Render Pass Setup

Render Pass Descriptor

Attachments define output

- Texture: Render Target
- Load actions
- Store actions



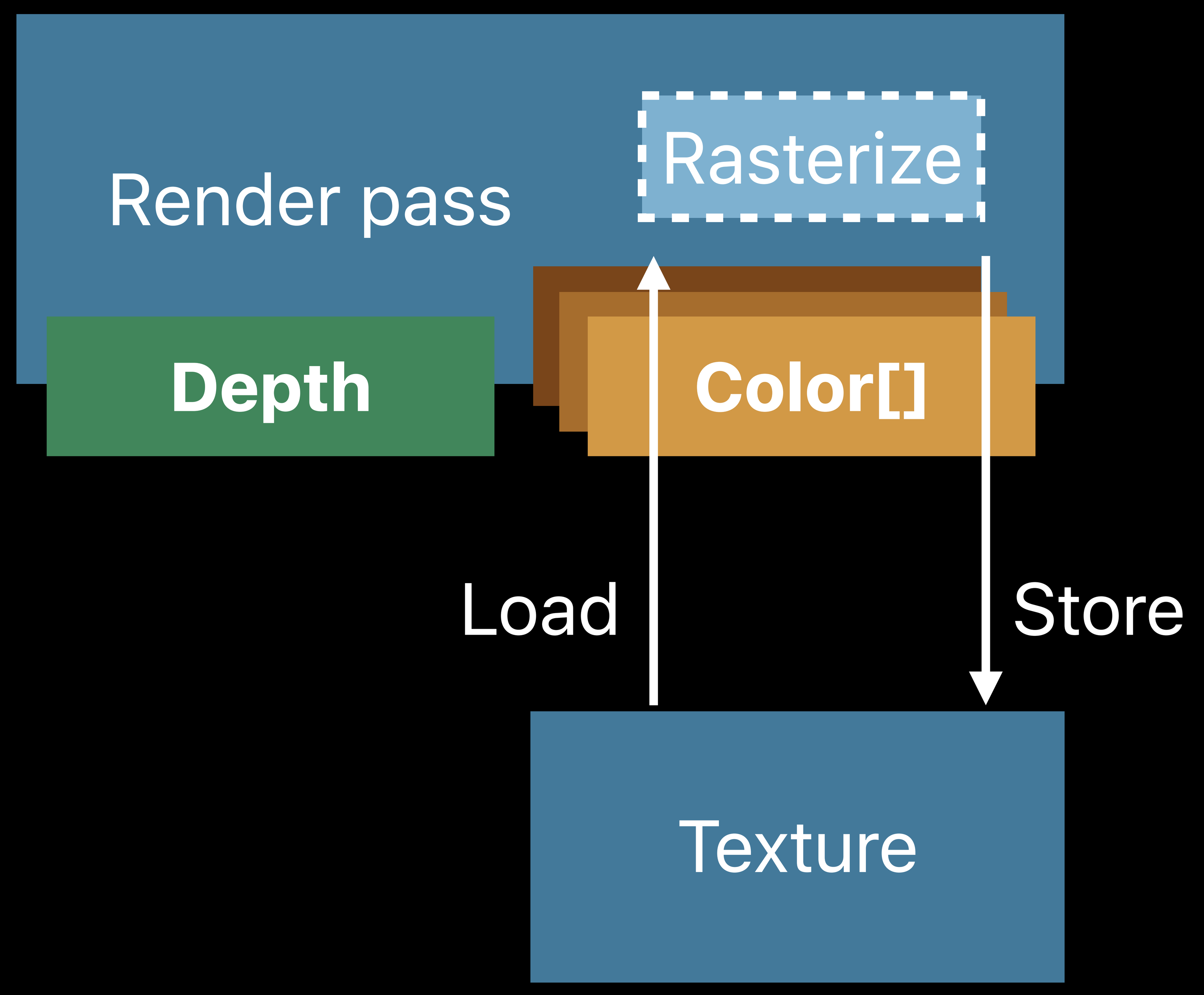
Render Pass Setup

Render Pass Descriptor

Attachments define output

- Texture: Render Target
- Load actions
- Store actions

Render Command Encoder



```swift
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
}
```

```swift
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
}
```

```swift
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
}
```

```swift
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()
}
```

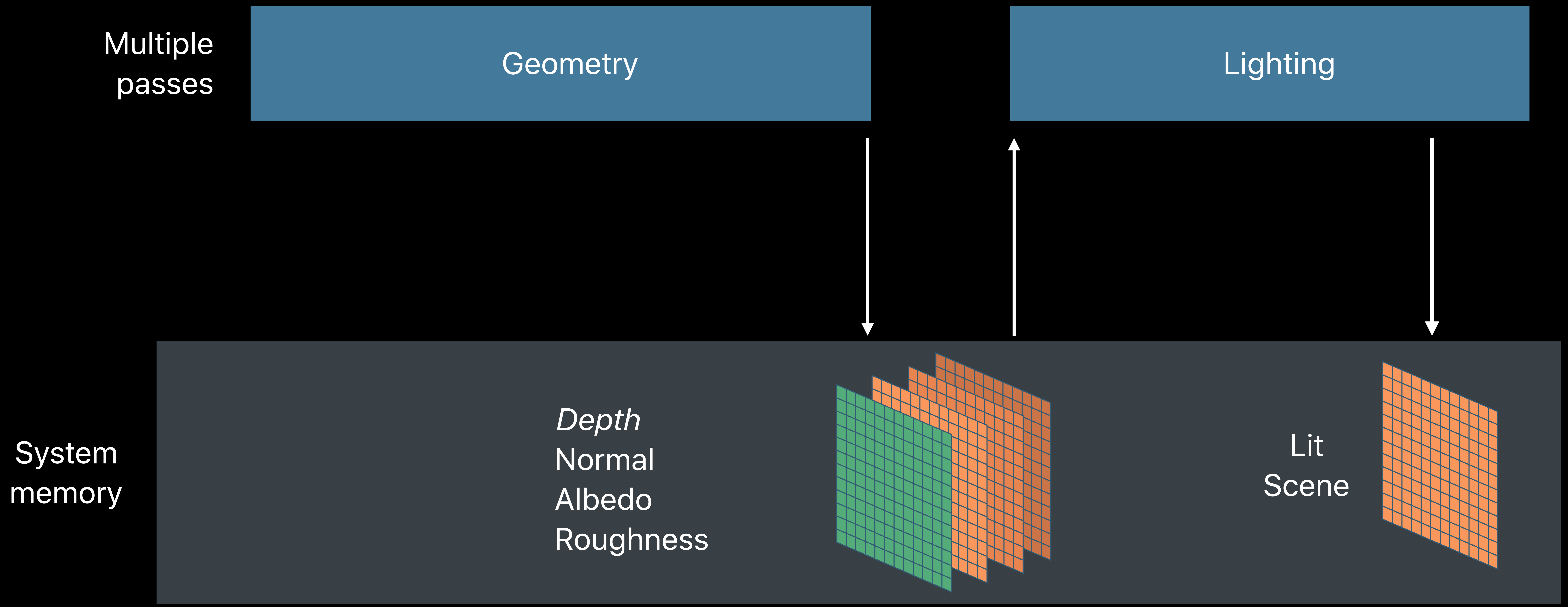
```swift
func render(cmdBuffer: MTLCommandBuffer)
{
  // geometry
  let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
  for mesh in _scene.meshes {
    // set state
    geoPass.drawIndexedPrimitives(...) // draw scene object
  }
  geoPass.endEncoding()

  // 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()
}
```

```swift
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()
}
```

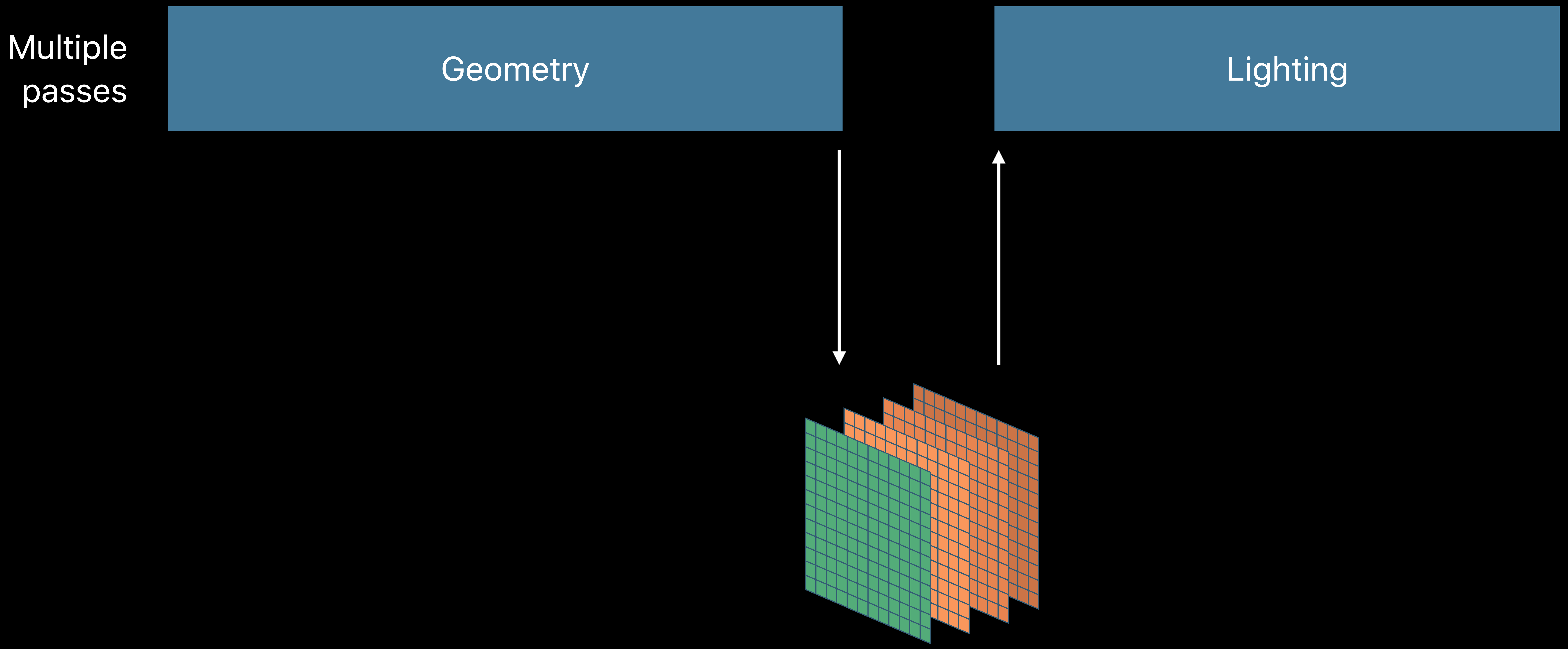
```swift
func render(cmdBuffer: MTLCommandBuffer)
{
  // geometry
  let geoPass = commandBuffer.makeRenderCommandEncoder(descriptor: geoRpd)
  for mesh in _scene.meshes {
    // set state
    geoPass.drawIndexedPrimitives(...) // draw scene object
  }
  geoPass.endEncoding()

  // 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()
}
```

```swift
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



Programmable Blending



Programmable Blending

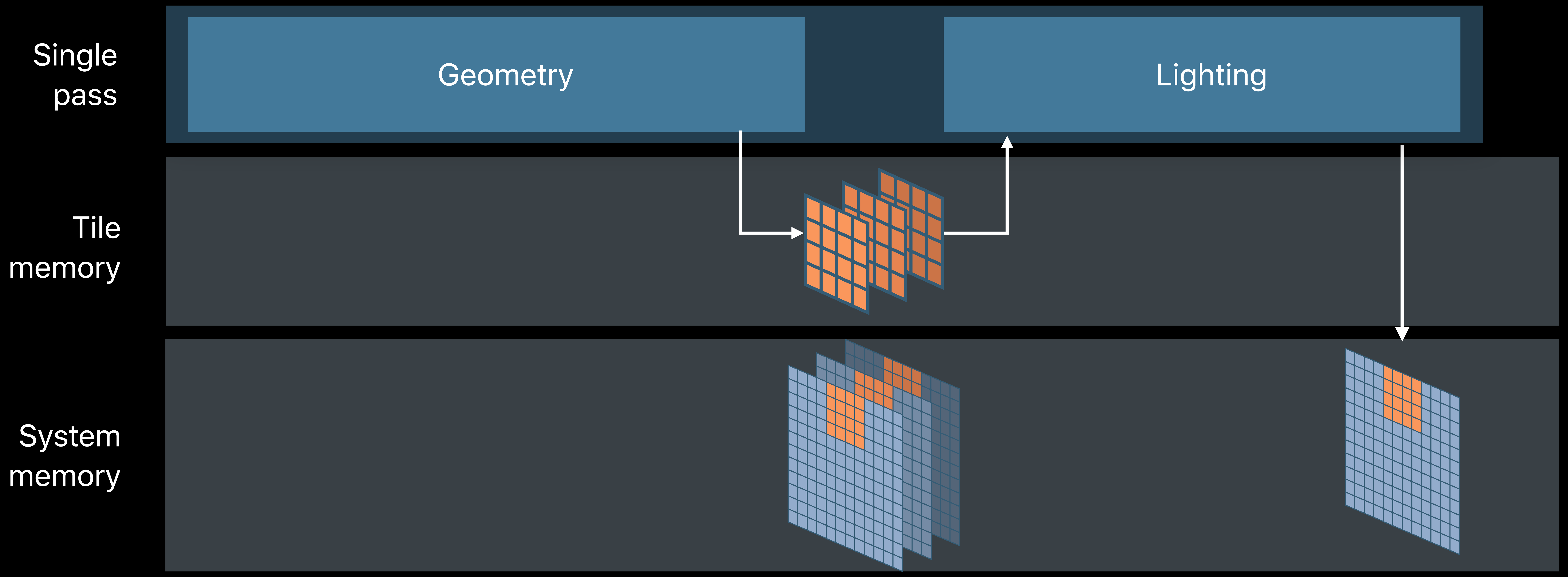
Merge passes

Programmable Blending

Merge passes

Programmable Blending

Merge passes



```
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                    [[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]],
                      float4 color                              [[color (0)]],
                      float4 normal                             [[color (1)]],
                      float  roughness                          [[color (2)]],
                      float  depth                              [[color (3)]],
                      ...)
{




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

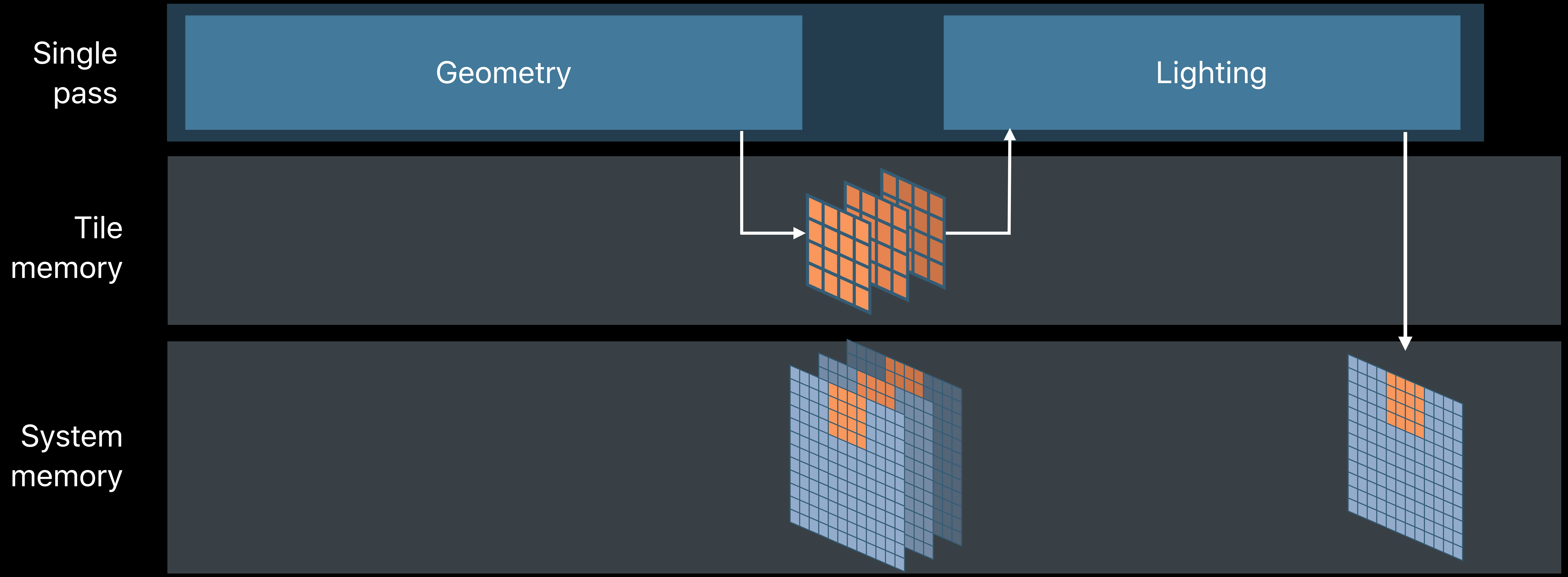
```
fragment float4 Shade(LightingVertexInput in                    [[stage_in]],
                      float4 color                               [[color (0)]],
                      float4 normal                              [[color (1)]],
                      float  roughness                           [[color (2)]],
                      float  depth                               [[color (3)]],
                      ...)
{





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

```
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

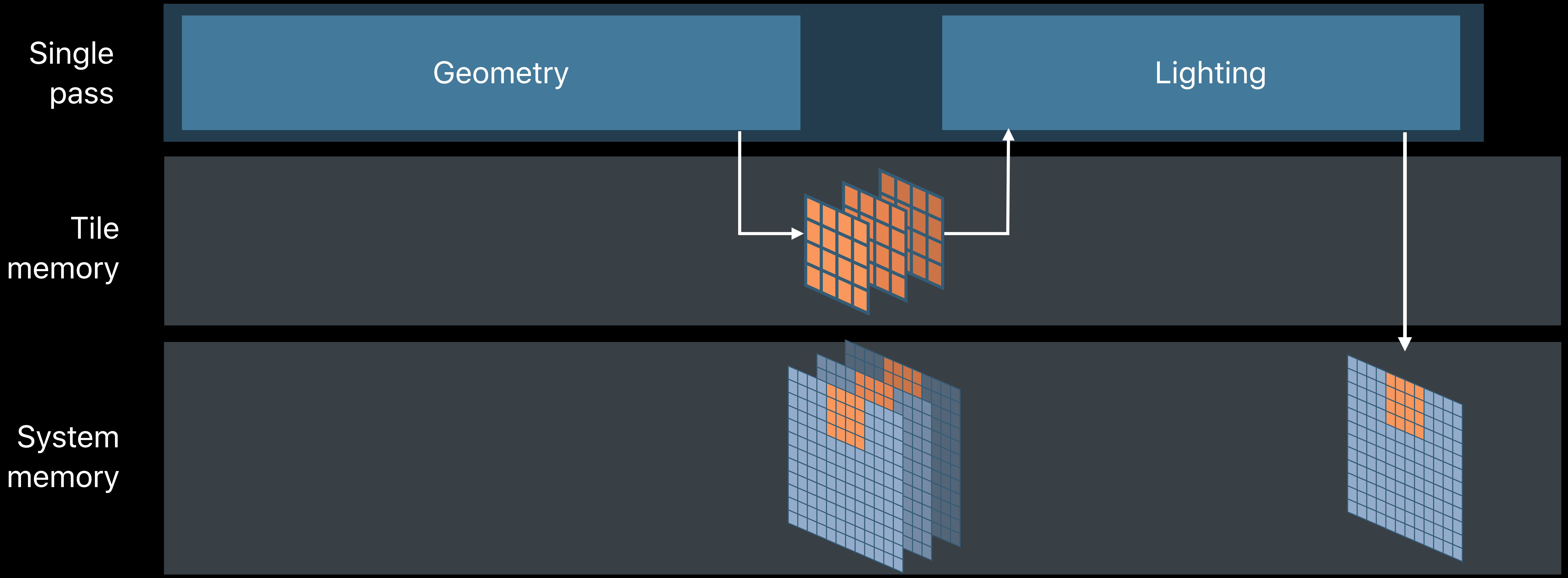
```
geoRpd = MTLRenderPassDescriptor()
geoRpd.colorAttachments[0].texture         = albedoTexture
geoRpd.colorAttachments[0].loadAction      = .dontcare
geoRpd.colorAttachments[0].storeAction     = .store
```



Programmable Blending

Don’t store transient attachments

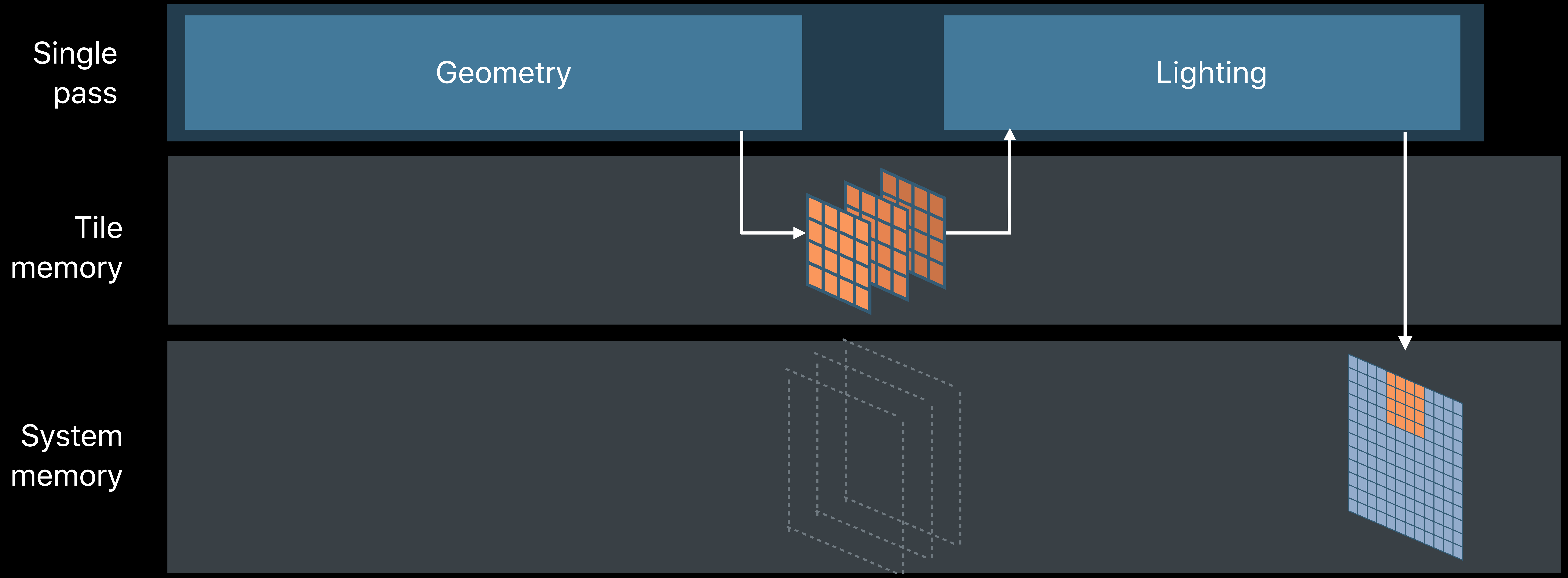
```
geoRpd = MTLRenderPassDescriptor()
geoRpd.colorAttachments[0].texture        = albedoTexture
geoRpd.colorAttachments[0].loadAction     = .dontcare
geoRpd.colorAttachments[0].storeAction    = .dontcare
```



Programmable Blending

Don’t allocate transient textures either

```
.storageMode = .memoryless
```



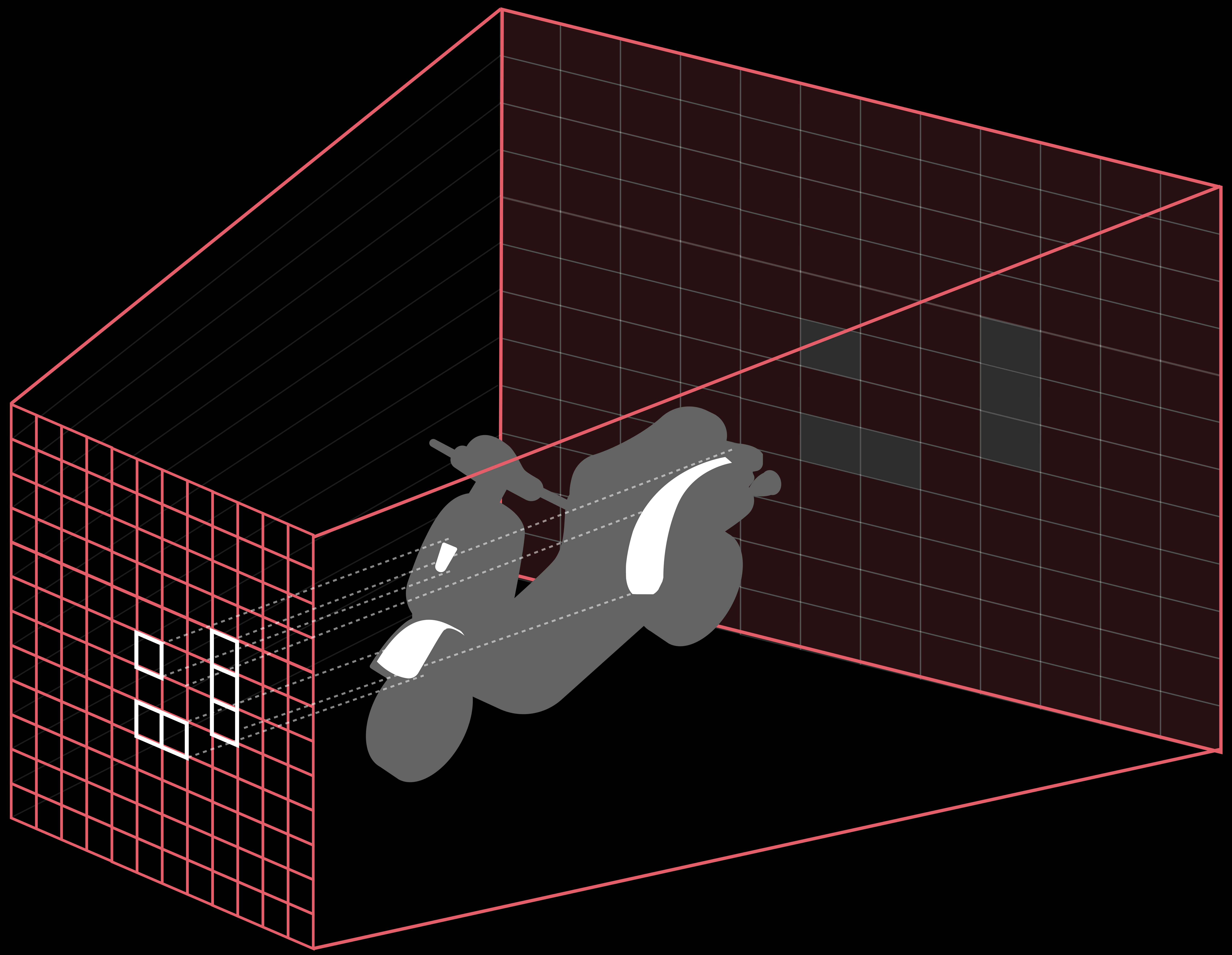
Deferred Rendering

	Material/Light Separation	Many Lights	Transparency	Anti-Aliasing	Material Complexity
Deferred	✓				
Tiled Deferred					
Tiled Forward					
Cluster Forward					
Visibility Buffer					

Tiled Deferred Rendering

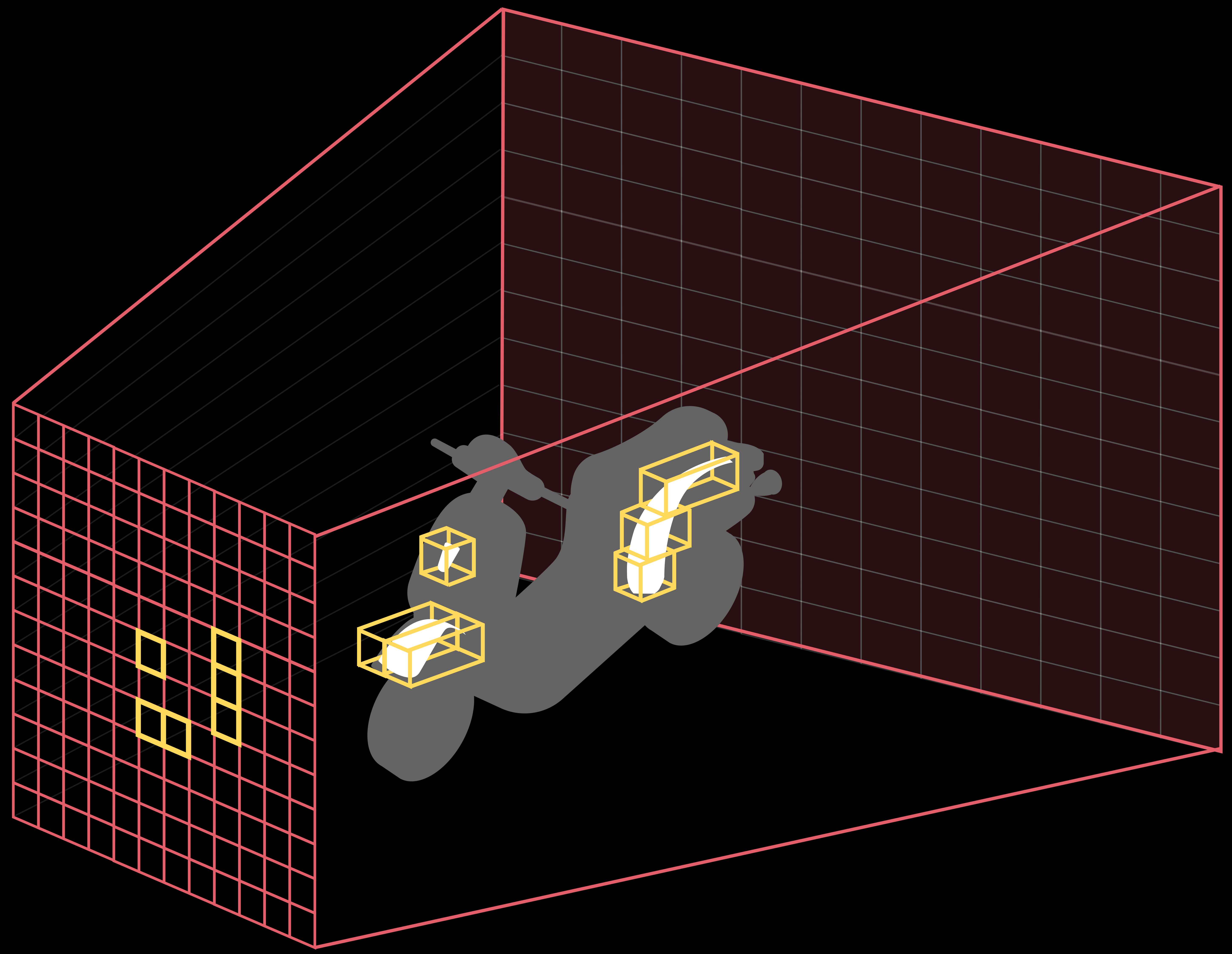
Tiled Deferred Rendering

Cull lights against geometry depth bounds



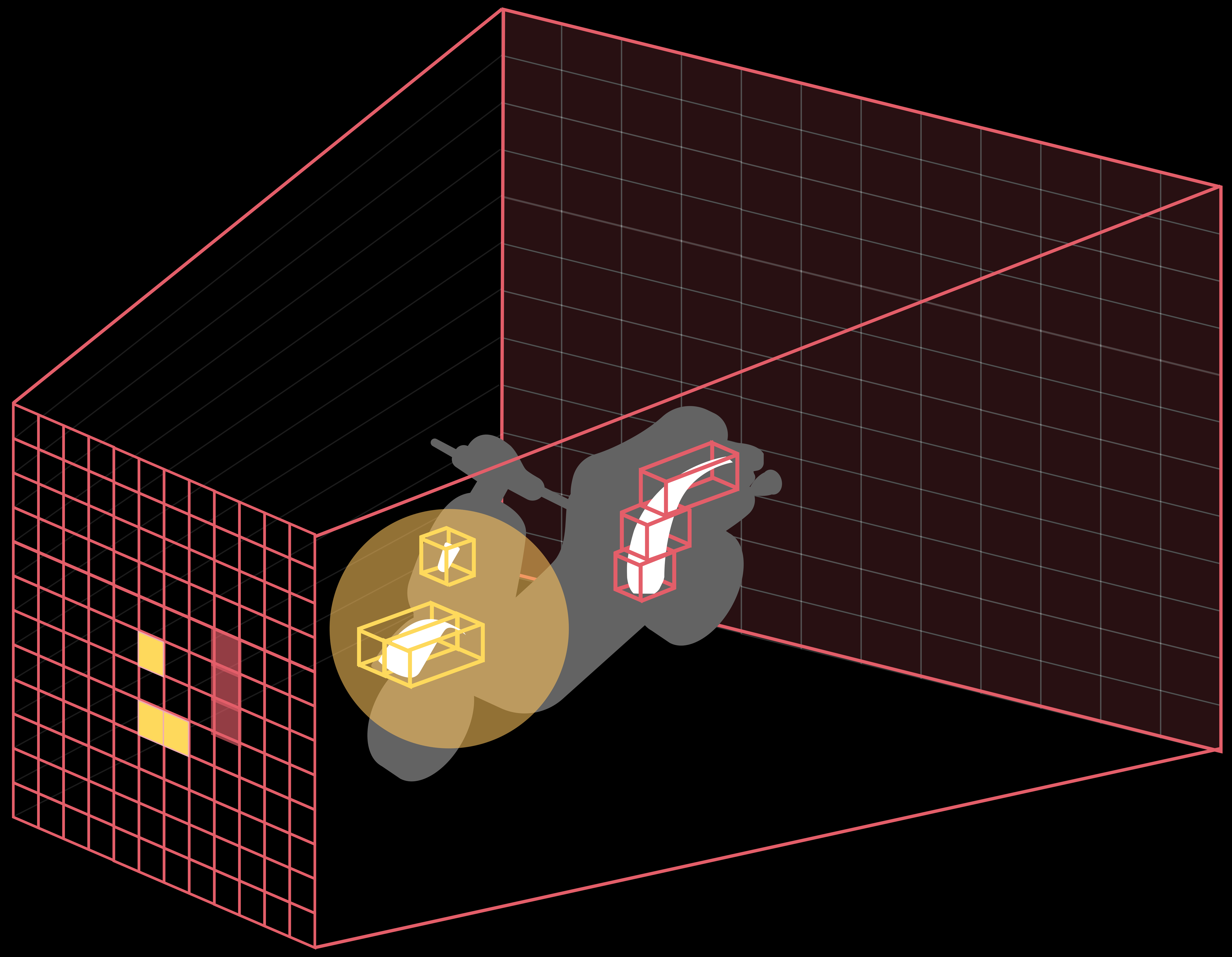
Tiled Deferred Rendering

Cull lights against geometry depth bounds

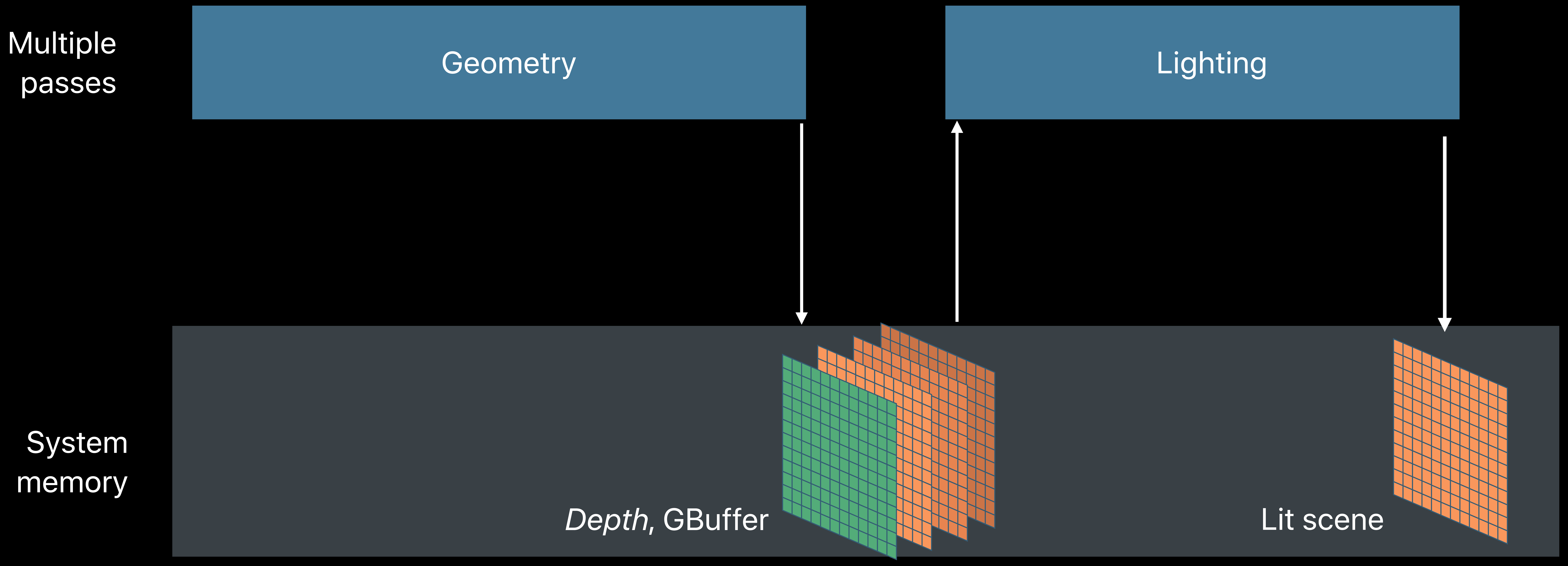


Tiled Deferred Rendering

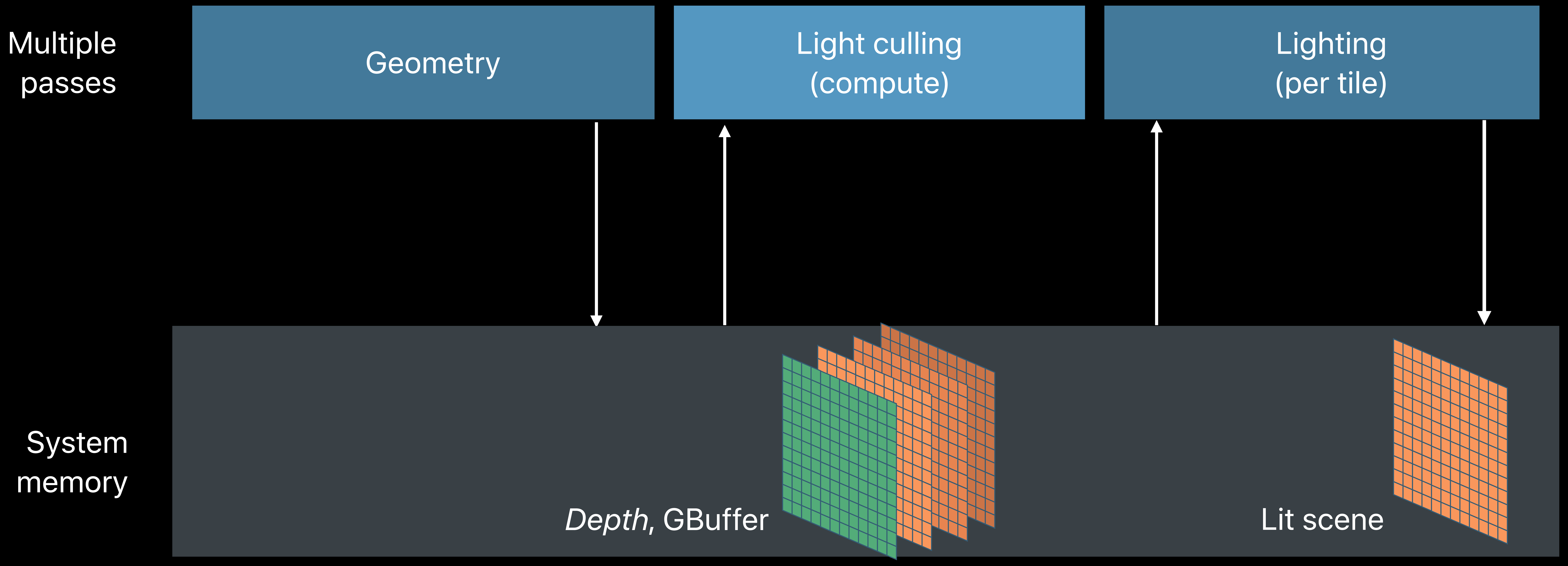
Cull lights against geometry depth bounds



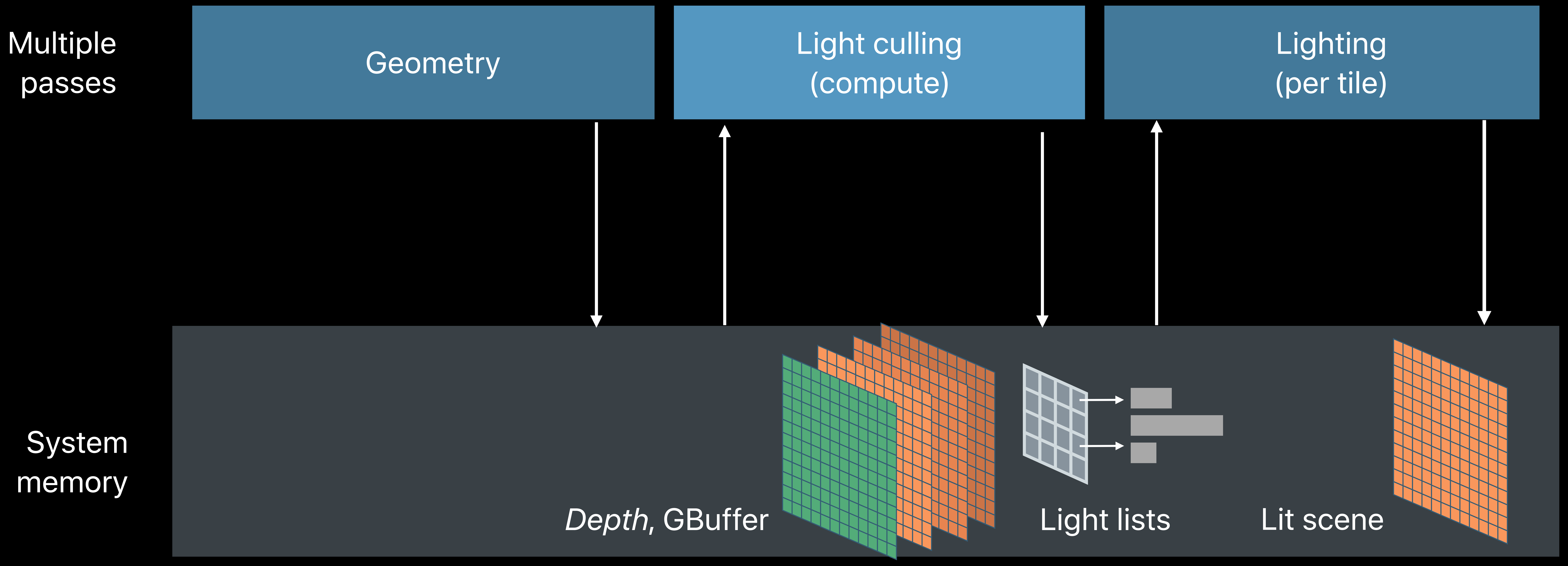
Tiled Deferred Rendering with Metal



Tiled Deferred Rendering with Metal

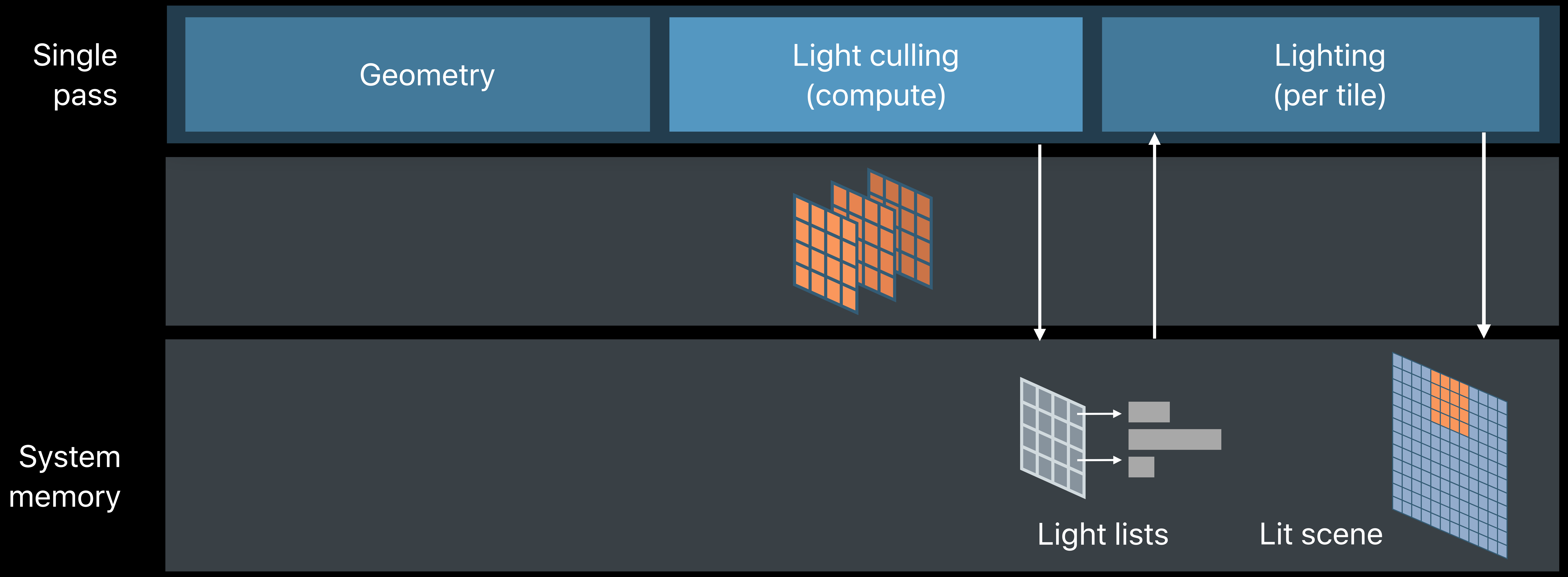


Tiled Deferred Rendering with Metal

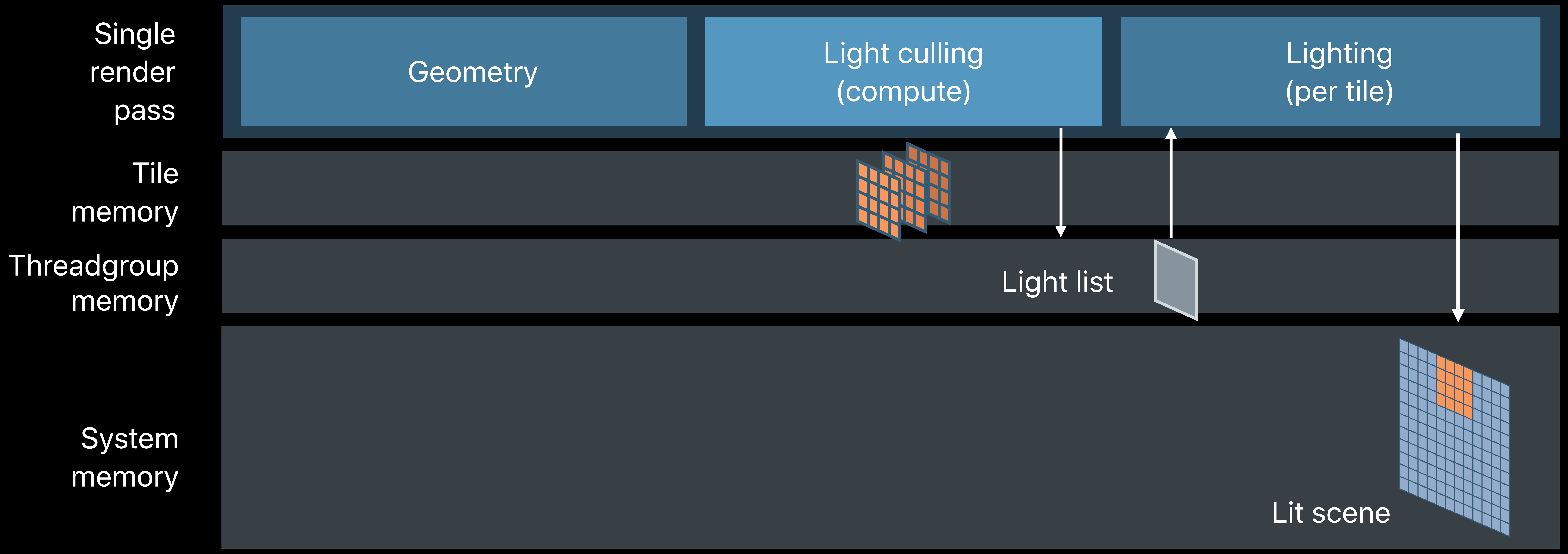


Tile Shaders

Merge tile-scoped compute into the render pass



Persistent Threadgroup Memory



```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
  ...
}
```

```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
  ...
}
```

```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
  ...
}
```

```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
  ...
}
```

```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
  ...
}
```

```swift
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()
}
```

```swift
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()
}
```

```swift
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()
}
```

```swift
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()
}
```

```swift
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()
}
```

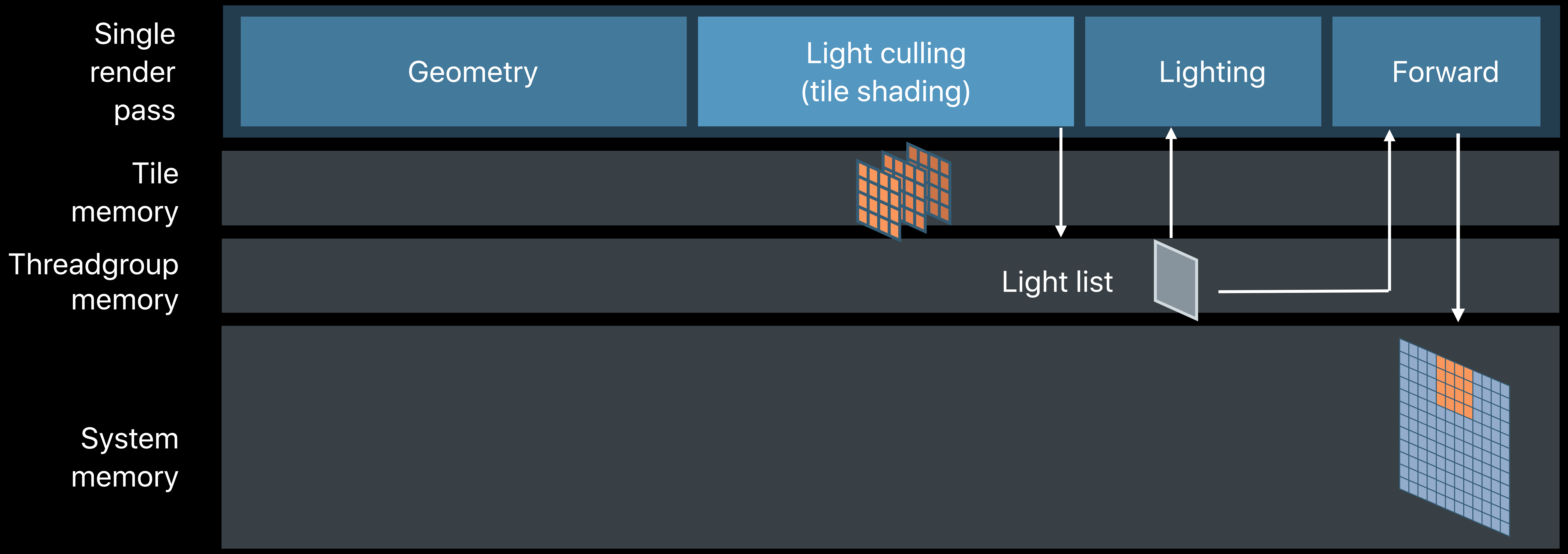
```
kernel void CullLights(device Light *all_lights                 [[buffer(0)]], …
                       threadgroup uint32_t &active_light_list [[threadgroup(0)]],
                       threadgroup float2 &depth_bounds        [[threadgroup(1)]])
{
  active_light_mask = 0; // clear light list
  for (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

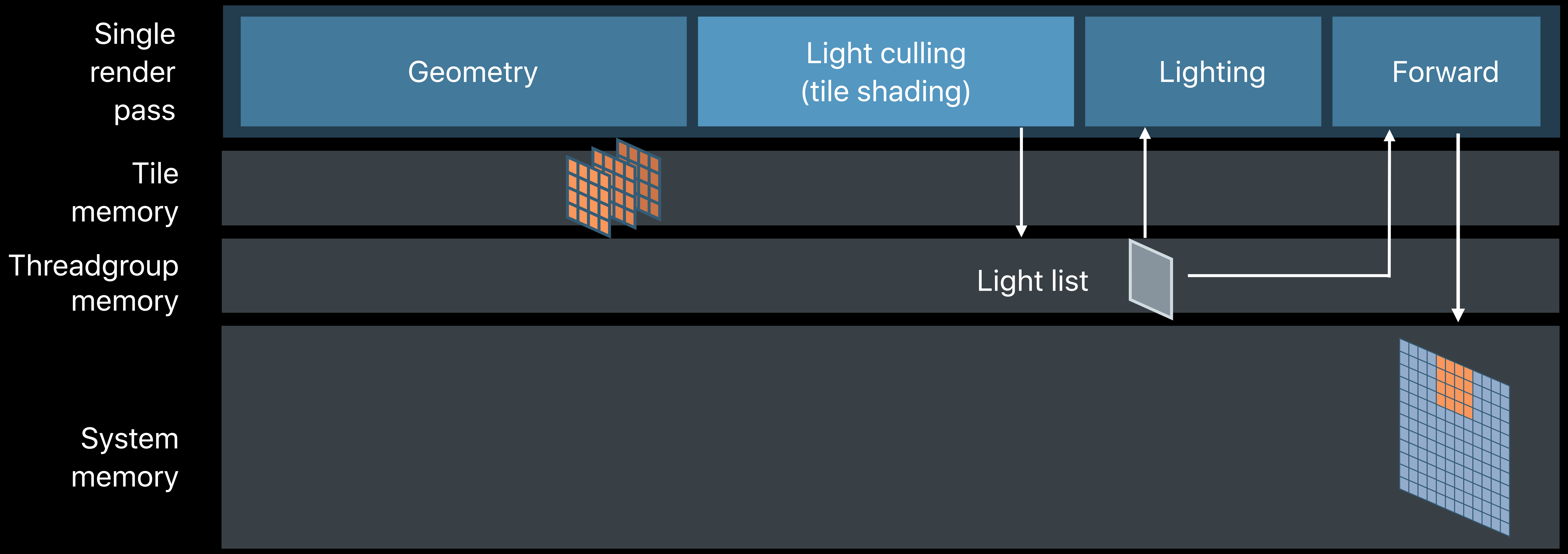


Forward Rendering with Light Lists

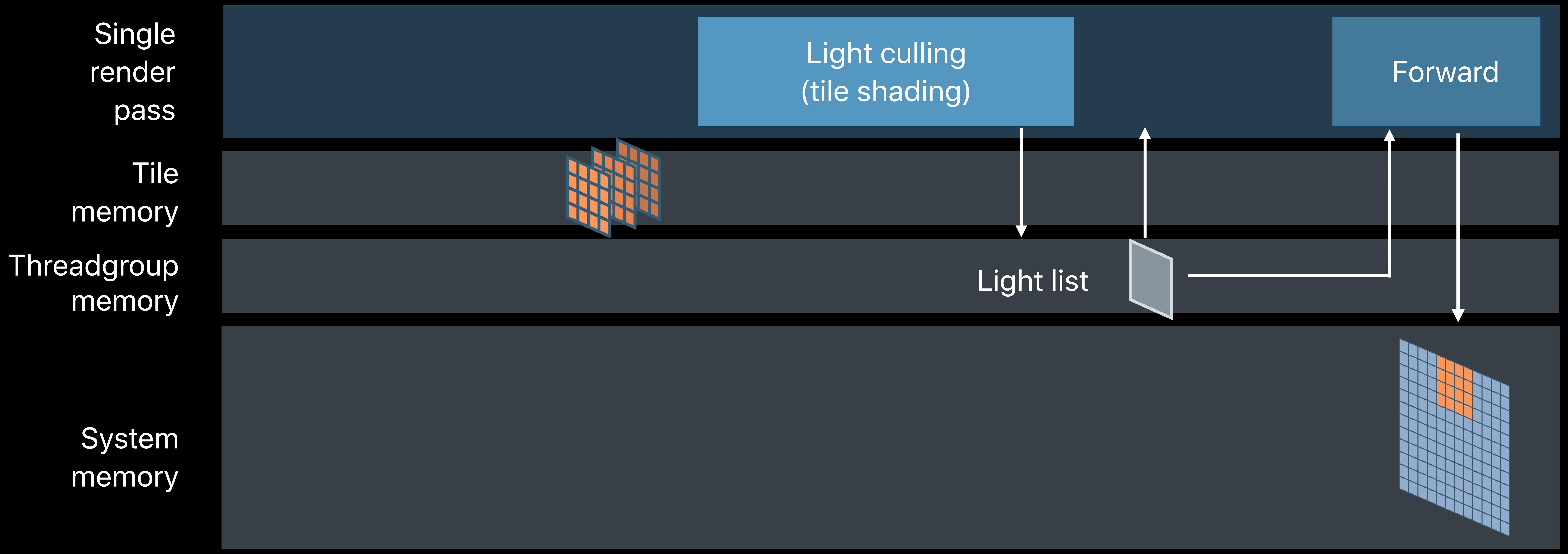
Tiled Deferred Rendering

	Material/Light Separation	Many Lights	Transparency	Anti-Aliasing	Material Complexity
Deferred	✓				
Tiled Deferred	✓	✓	✓		
Tiled Forward					
Cluster Forward					
Visibility Buffer					

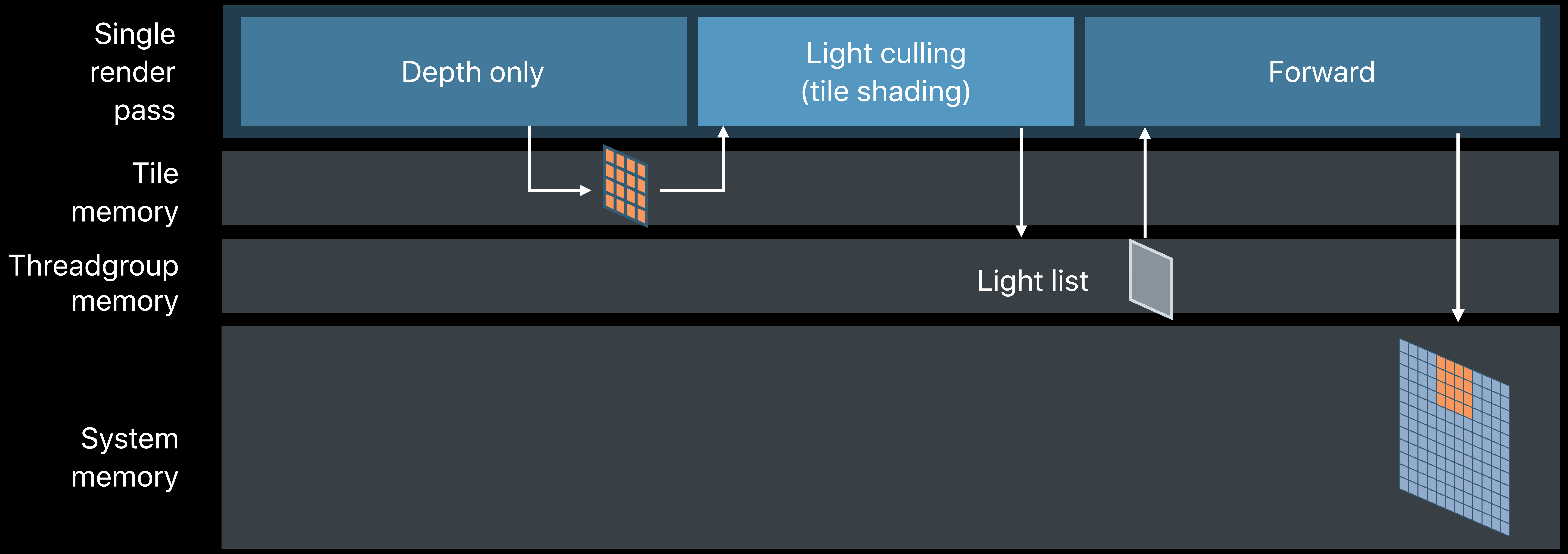
Tiled Forward Rendering



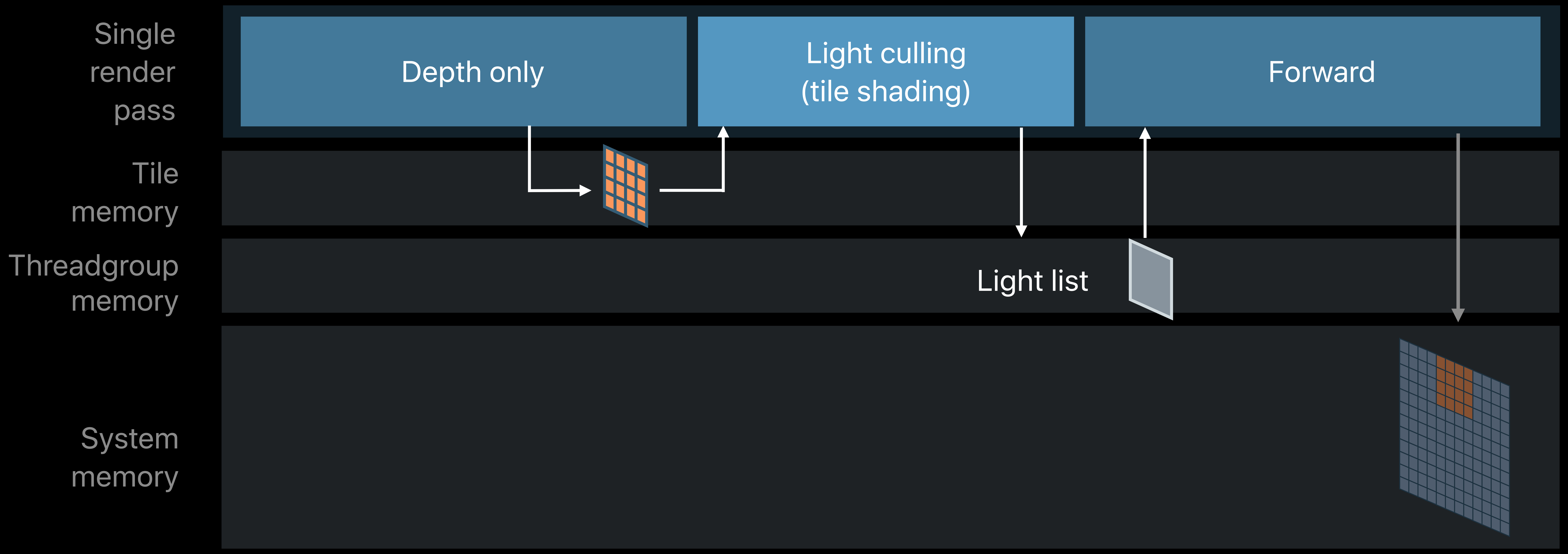
Tiled Forward Rendering



Tiled Forward Rendering

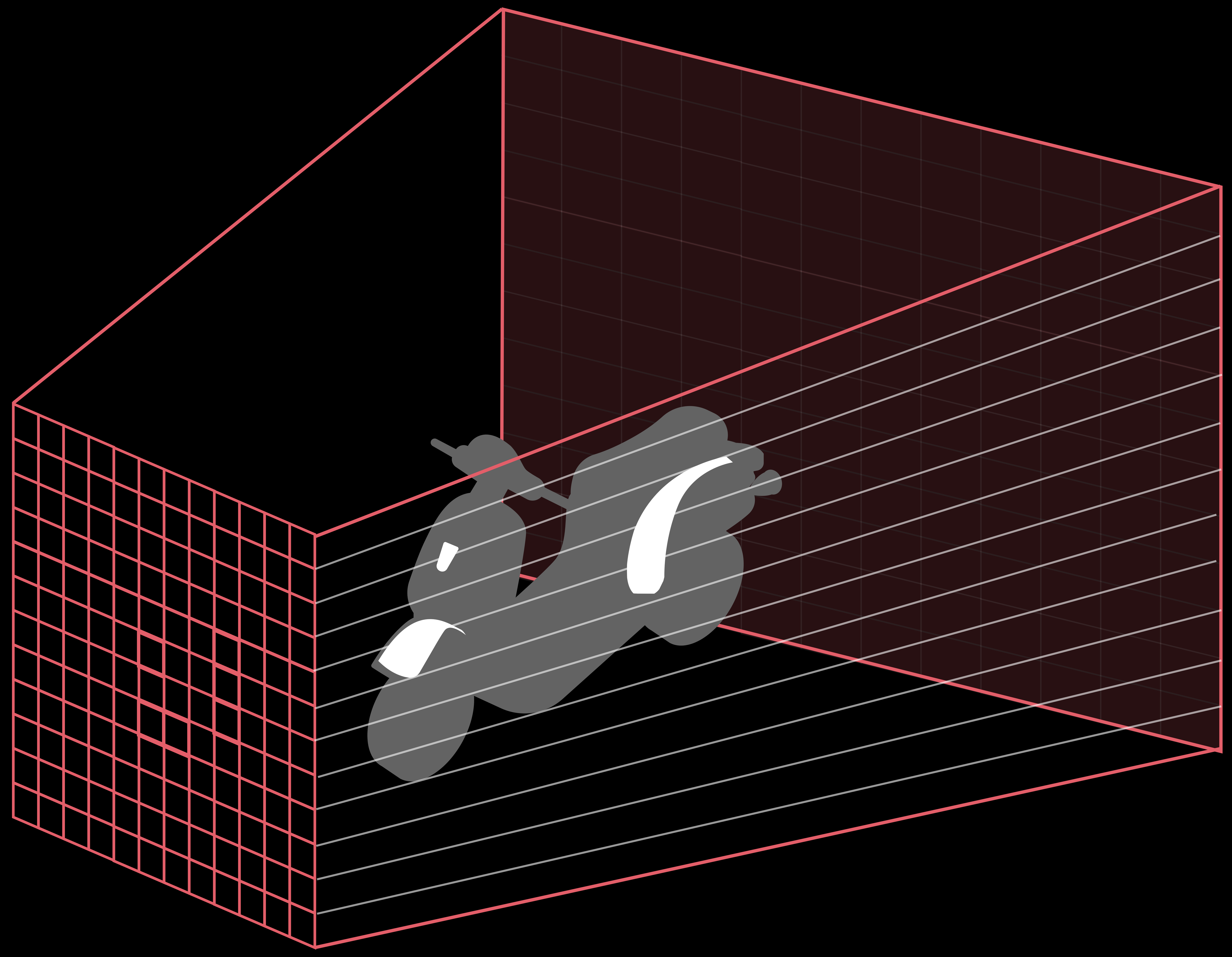


Tiled Forward Rendering



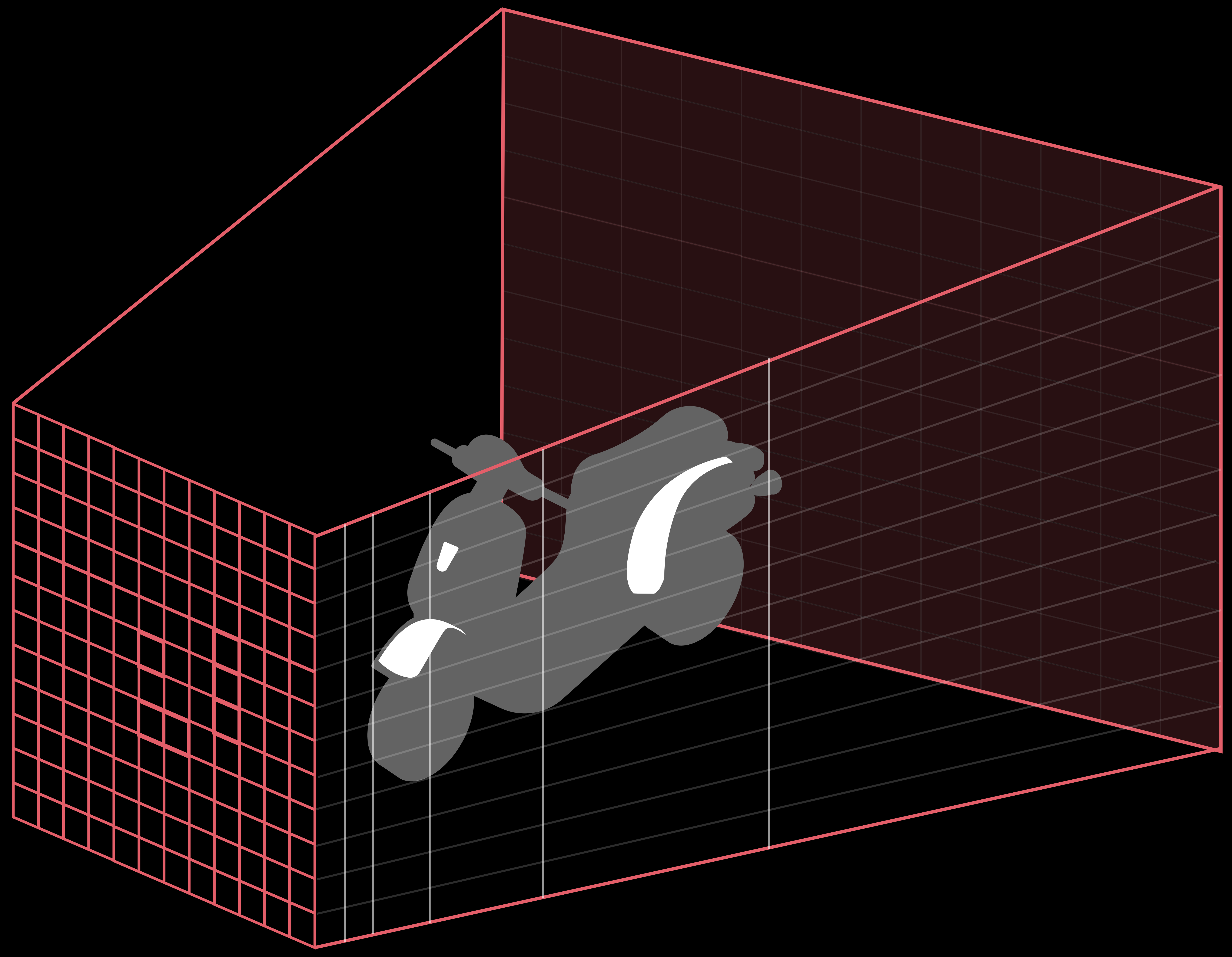
Clustered Light Culling

Subdivide tiles into depth



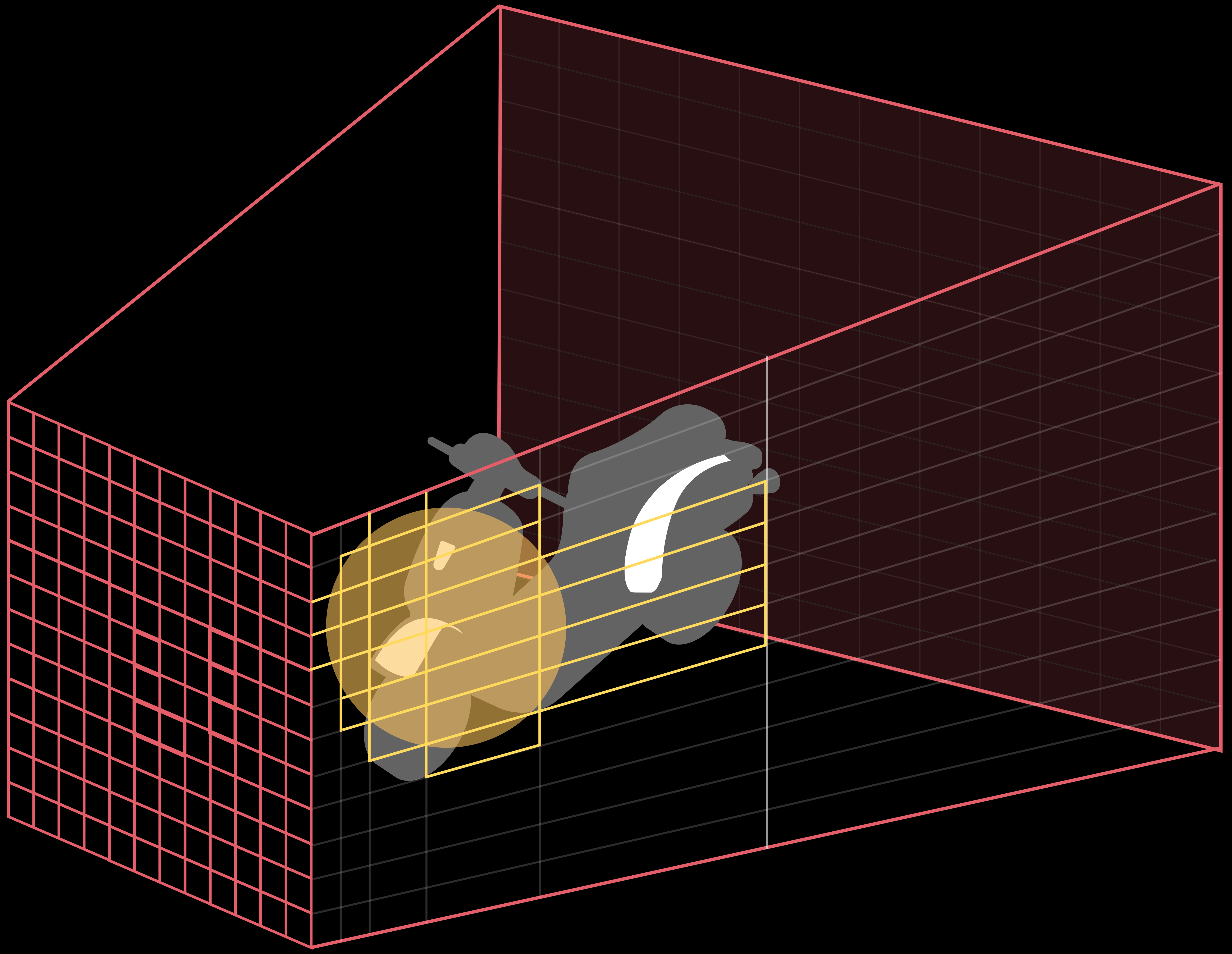
Clustered Light Culling

Subdivide tiles into depth



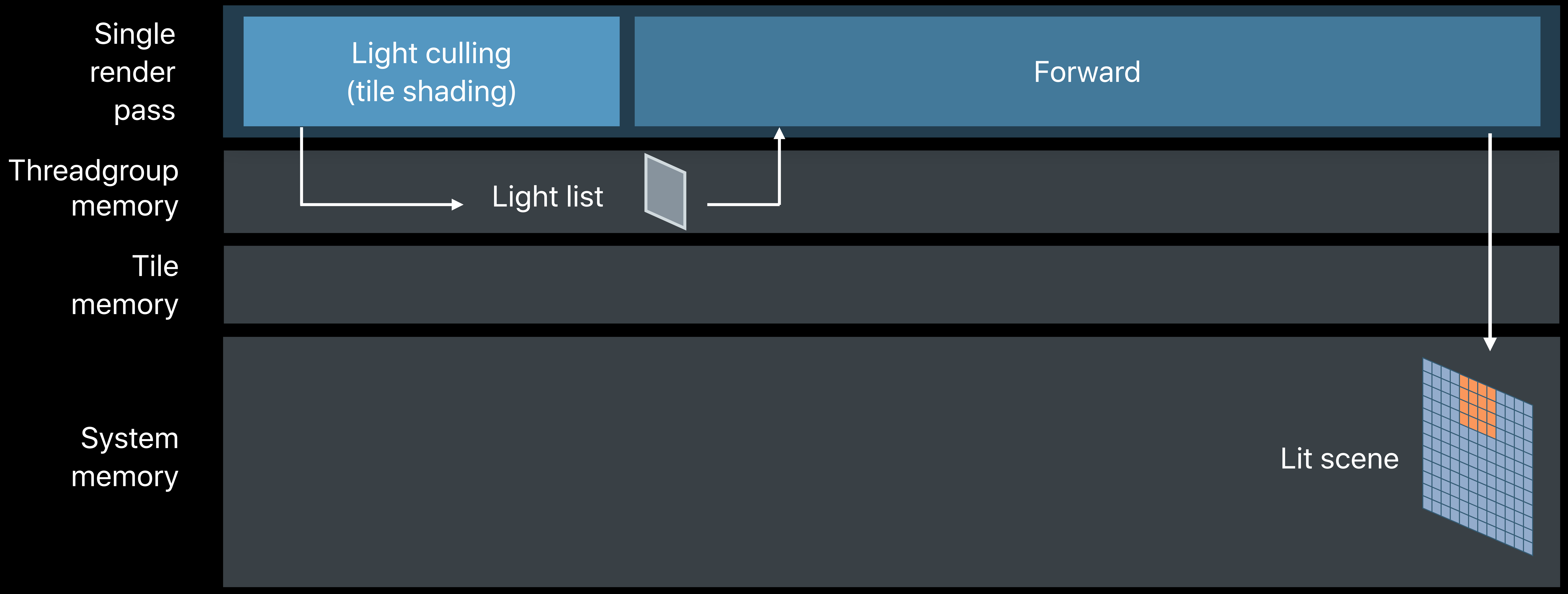
Clustered Light Culling

Generate a 3D light list



Clustered Forward Rendering

No depth pass needed

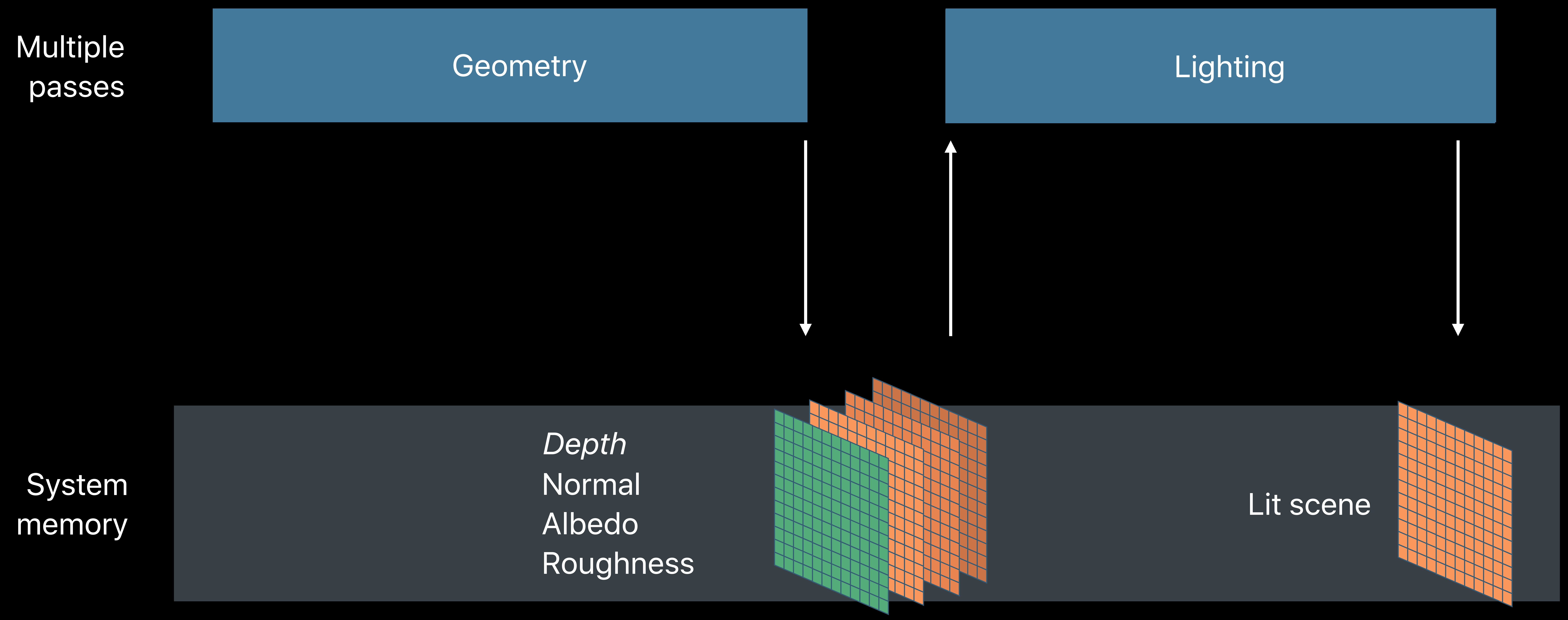


Forward Rendering

	Material/Light Separation	Many Lights	Transparency	Anti-Aliasing	Material Complexity
Deferred	✓				
Tiled Deferred	✓	✓	✓		
Tiled Forward	✓		✓	✓	✓
Cluster Forward	✓	✓	✓	✓	✓
Visibility Buffer					

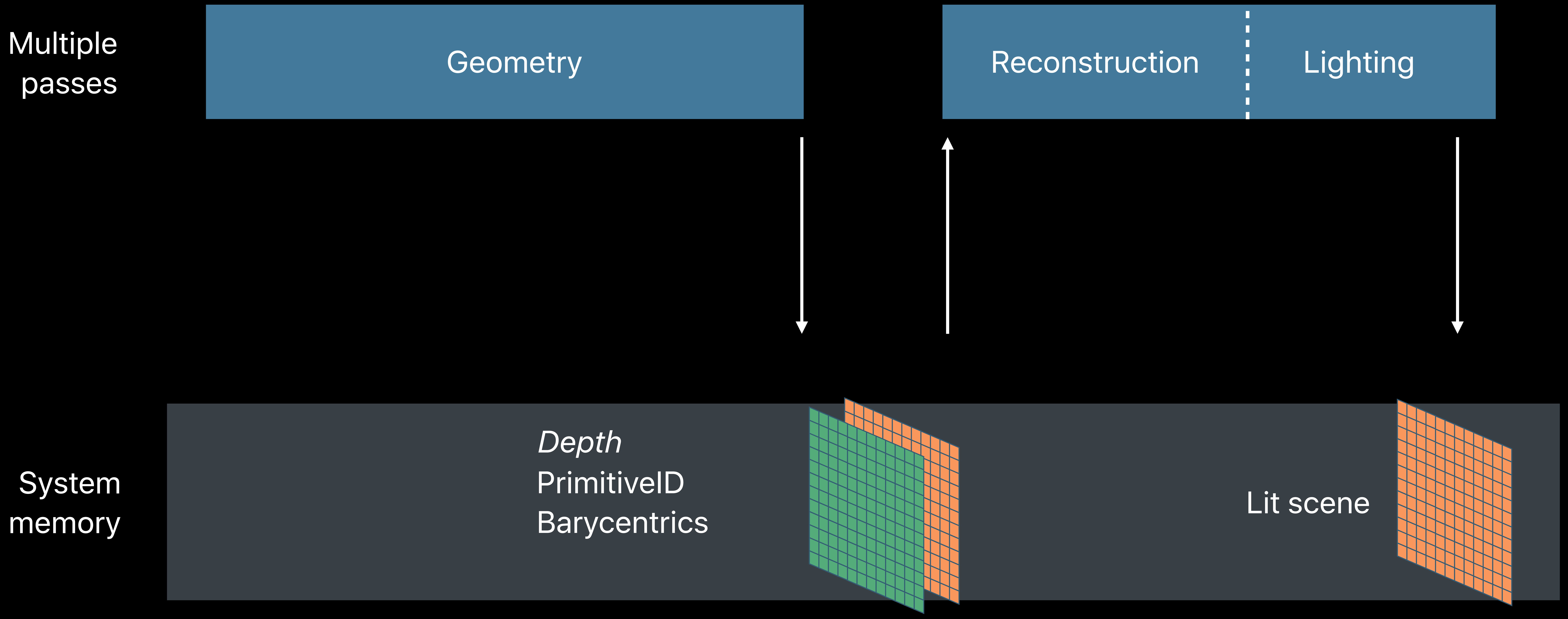
Visibility Buffer

Minimal GBuffer

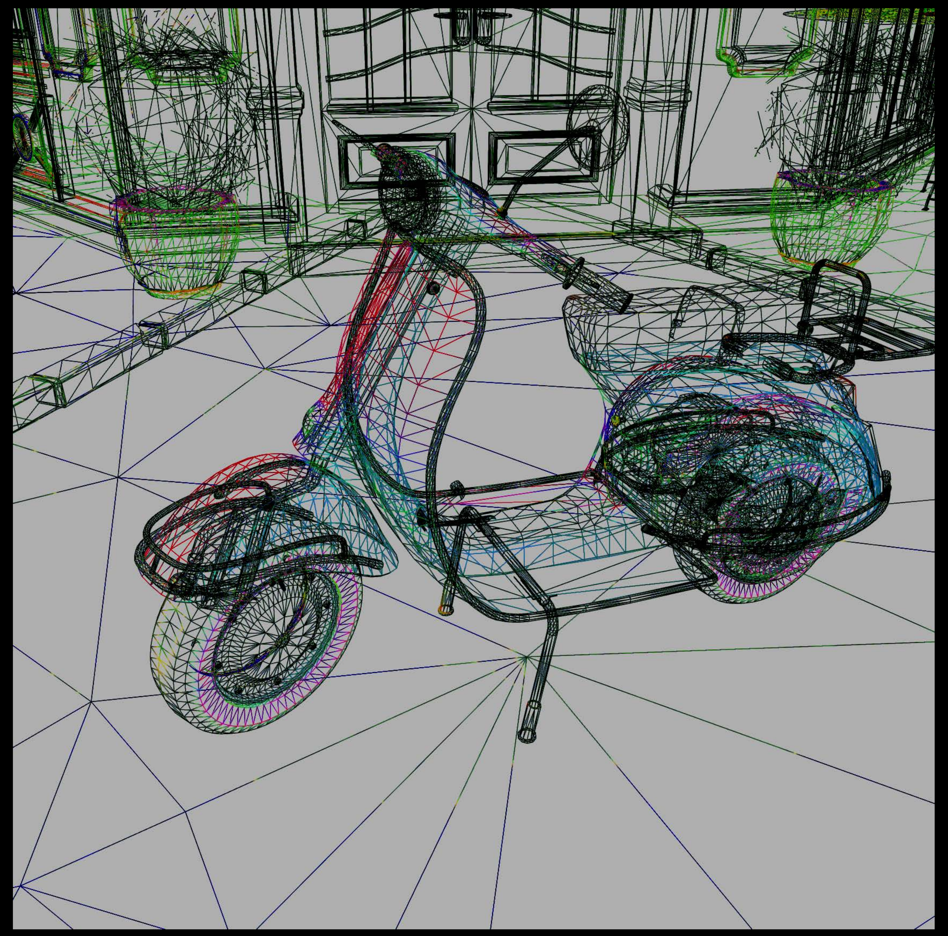


Visibility Buffer

Minimal GBuffer



Visibility Buffer



Geometry

Barycentrics

Primitive ID

Geometry reconstruction

Material shading

Lighting

Geometry

Lit scene

Primitive ID and Barycentric Coordinates

NEW

```
float3 [[barycentric_coord]]
```

- barycentric coordinates for current fragment

```
uint [[primitive_id]]
```

- primitive ID of the current primitive

```
fragment GBufferOutput GenerateVisibility(VertexToFragment in    [[stage_in]],
                                          constant uint& mesh_id [[buffer(0)]],
                                          float3 barycentrics    [[barycentric_coord]],
                                          uint prim_id           [[primitive_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(VertexToFragment in    [[stage_in]],
                                          constant uint& mesh_id [[buffer(0)]],
                                          float3 barycentrics    [[barycentric_coord]],
                                          uint prim_id           [[primitive_id]])
{
    GBufferOutput out;
    out.barycentrics  = barycentrics.xy;
    out.geometry_ids  = (prim_id & 0xFFFF) + (mesh_id << 16);
    out.depth         = in.position.z;
    return out;
}
```

Visibility Buffer

	Material/Light Separation	Many Lights	Transparency	Anti-Aliasing	Material Complexity
Deferred	✓				
Tiled Deferred	✓	✓	✓		
Tiled Forward	✓		✓	✓	✓
Cluster Forward	✓	✓	✓	✓	✓
Visibility Buffer	✓	✓			


Le Chevalier


LIBRAIRIE


LIBRAIRIE


LIBRAIRIE

GPU Driven Pipelines

Srinivas Dasari, GPUSW

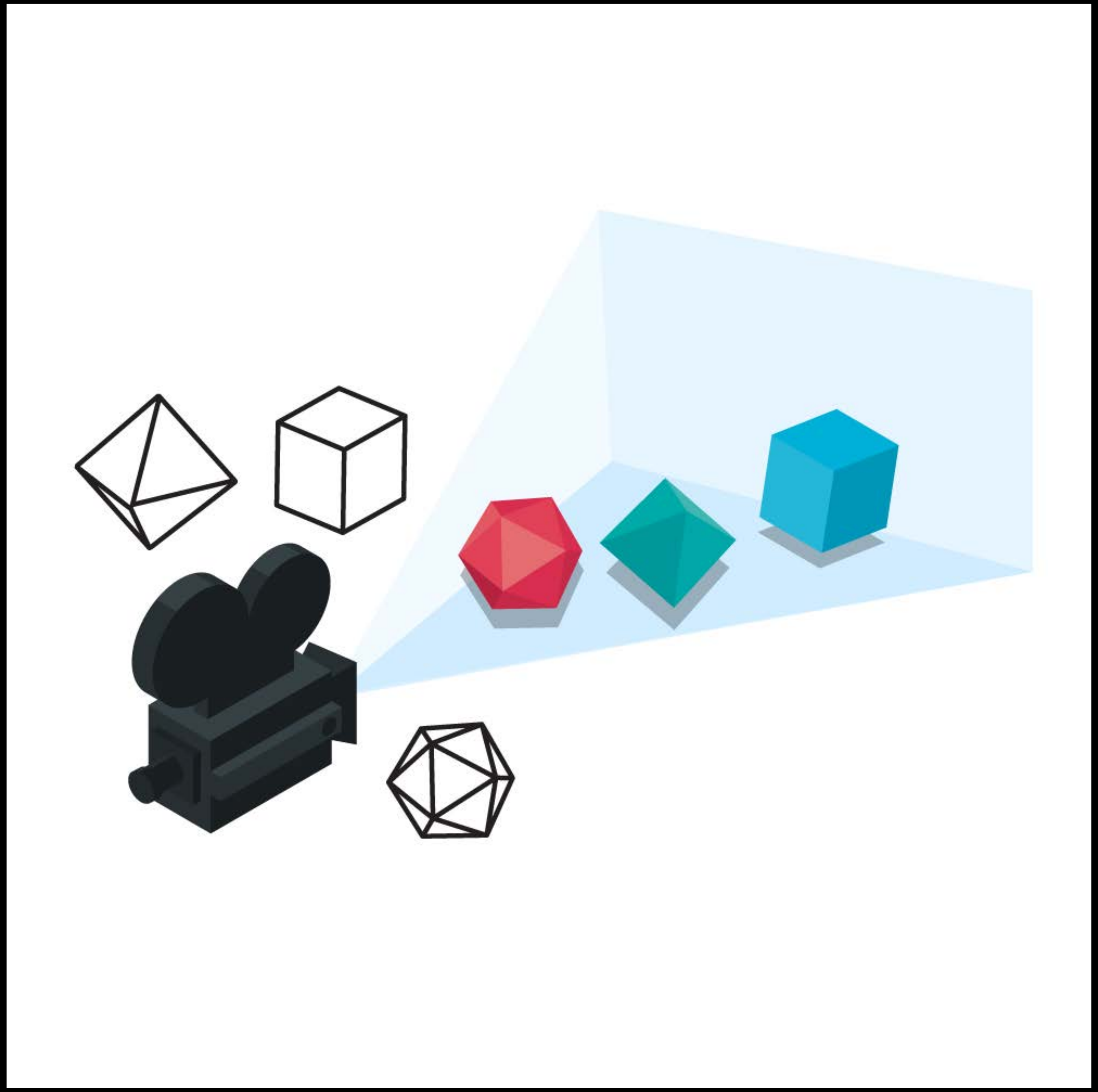
Render Loop

Large scenes require complex rendering operations

Render Loop

Large scenes require complex rendering operations

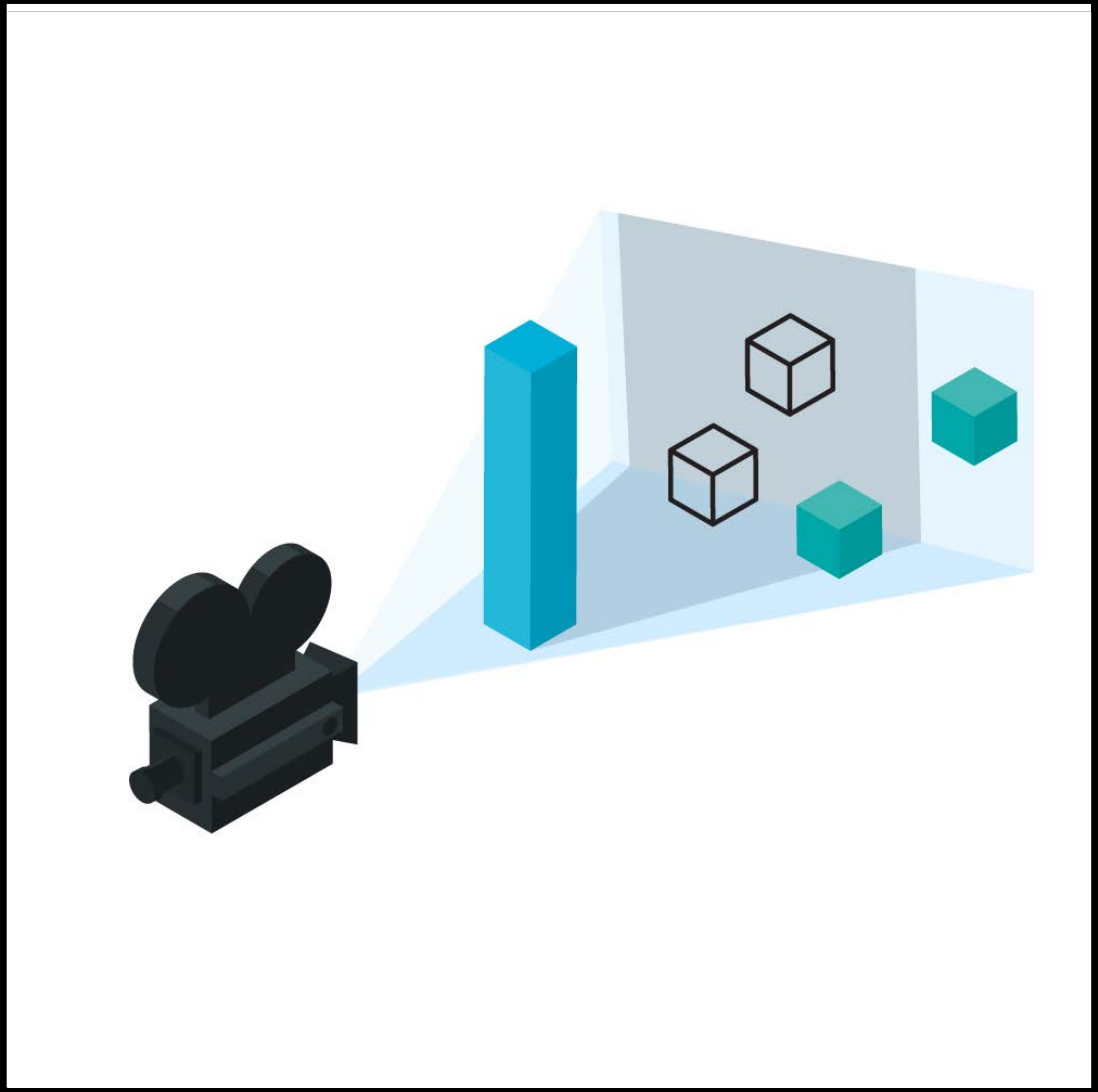
- Frustum culling



Render Loop

Large scenes require complex rendering operations

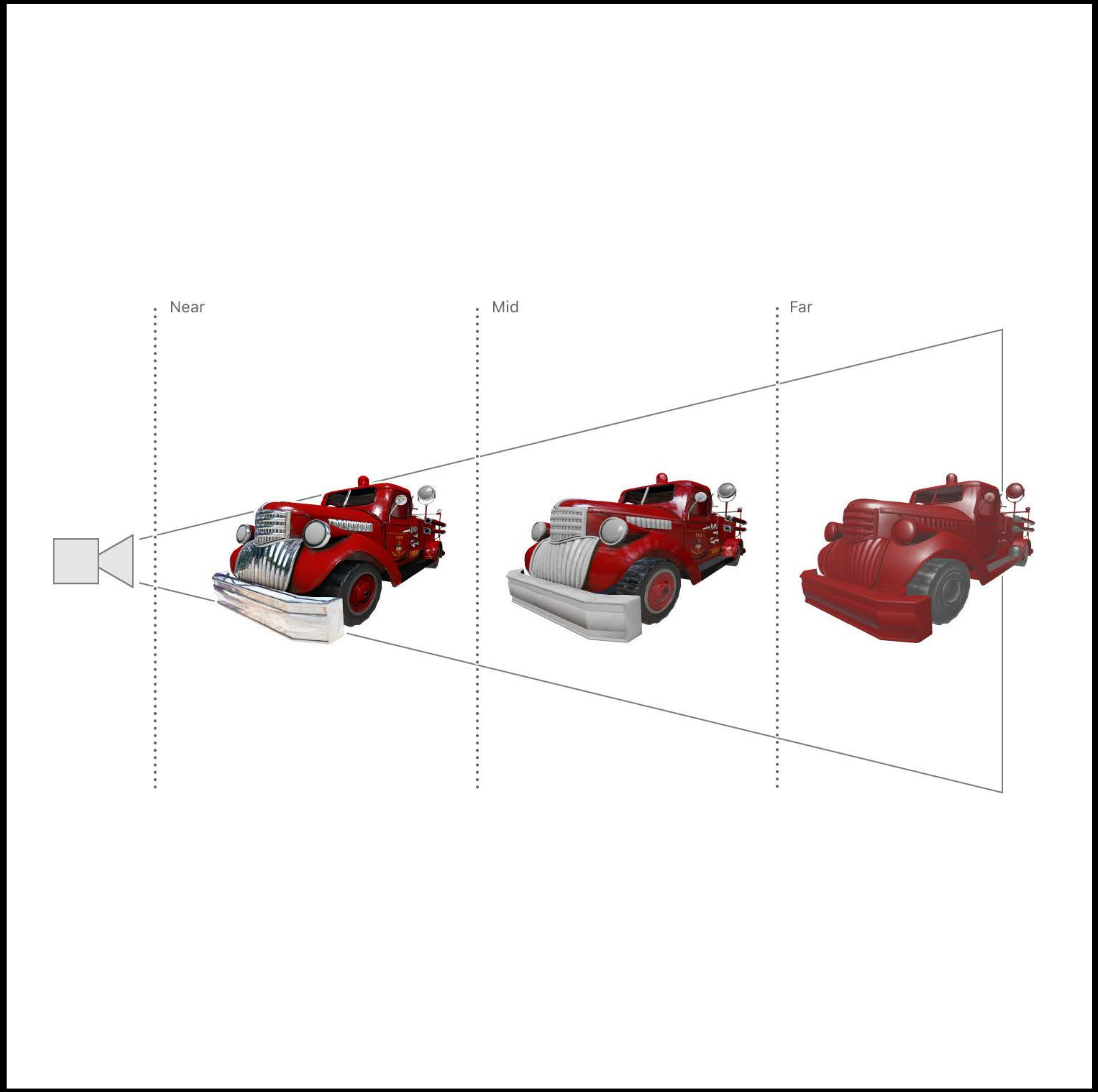
- Frustum culling
- Occlusion culling



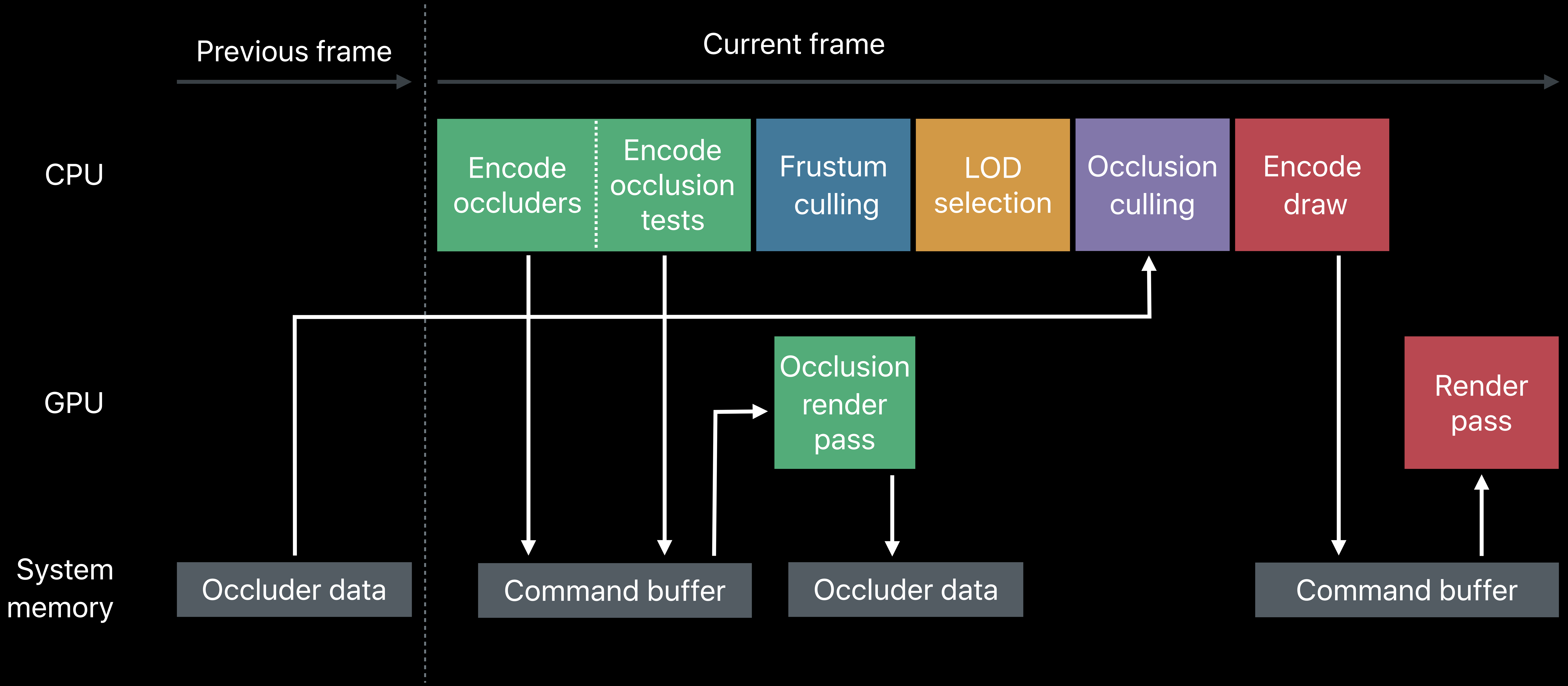
Render Loop

Large scenes require complex rendering operations

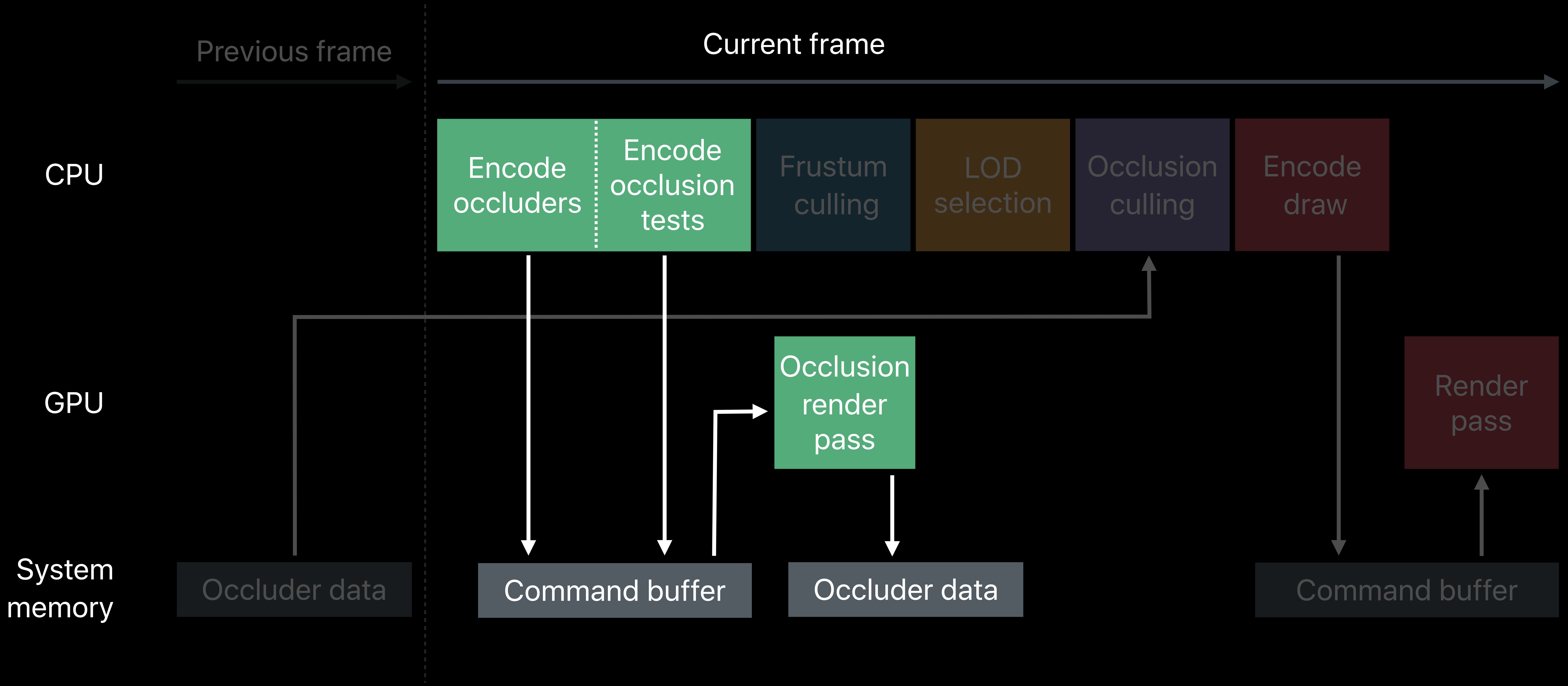
- Frustum culling
- Occlusion culling
- LOD selection



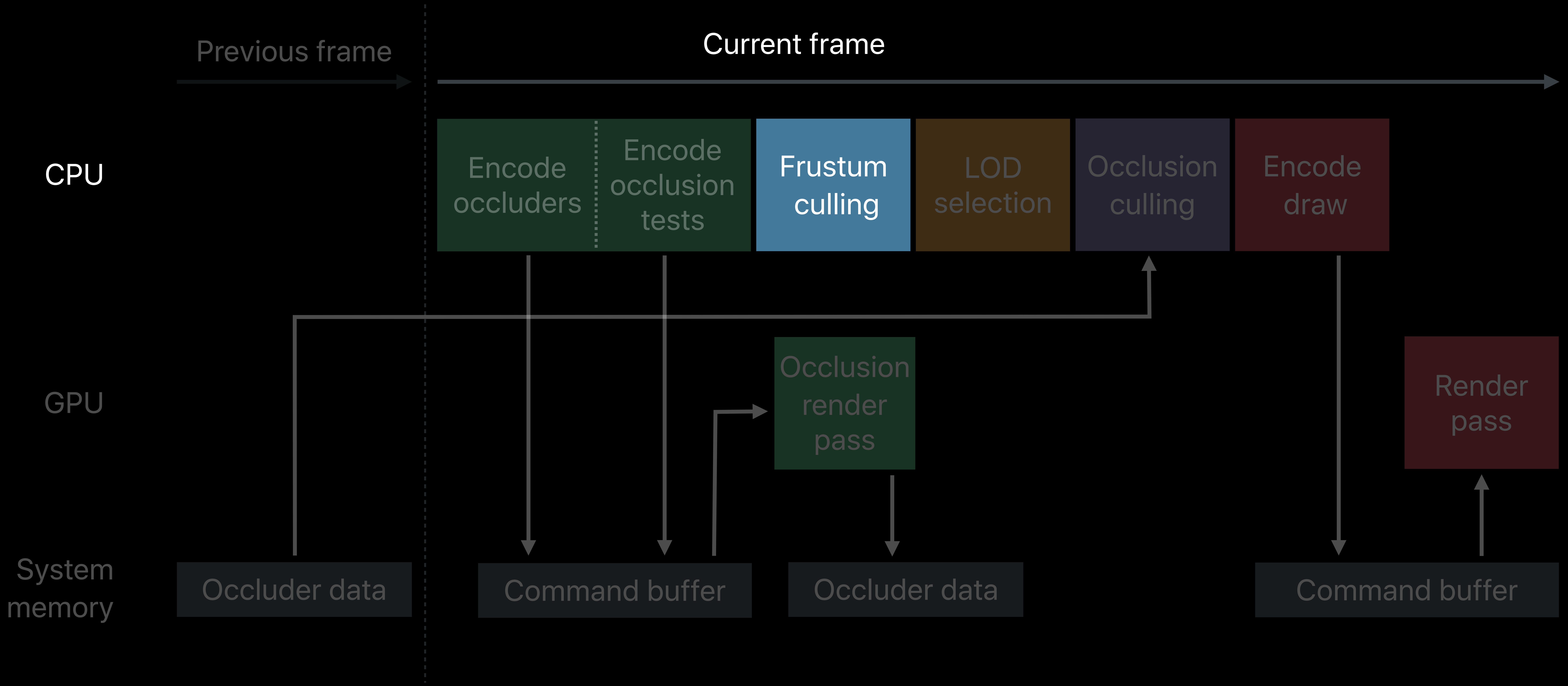
Traditional CPU Driven Render Loop



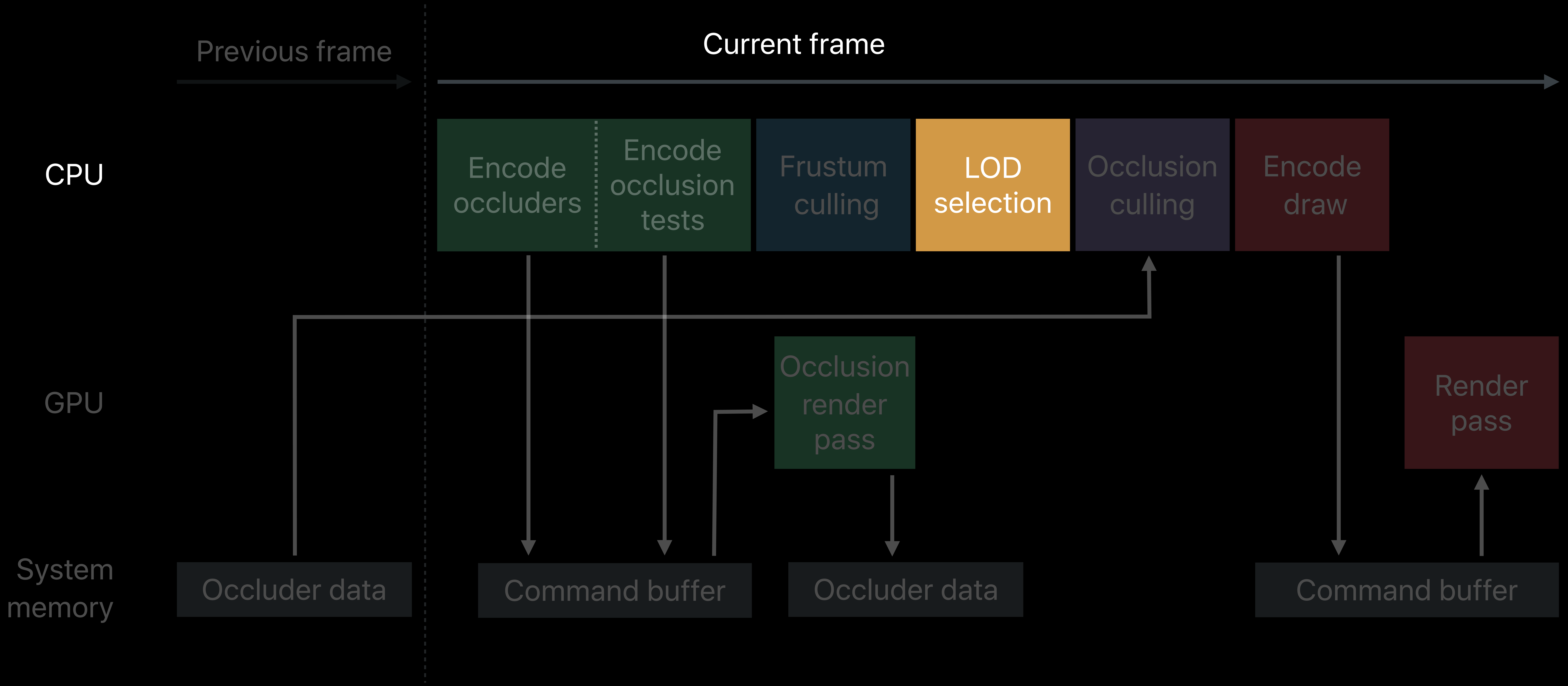
Traditional CPU Driven Render Loop



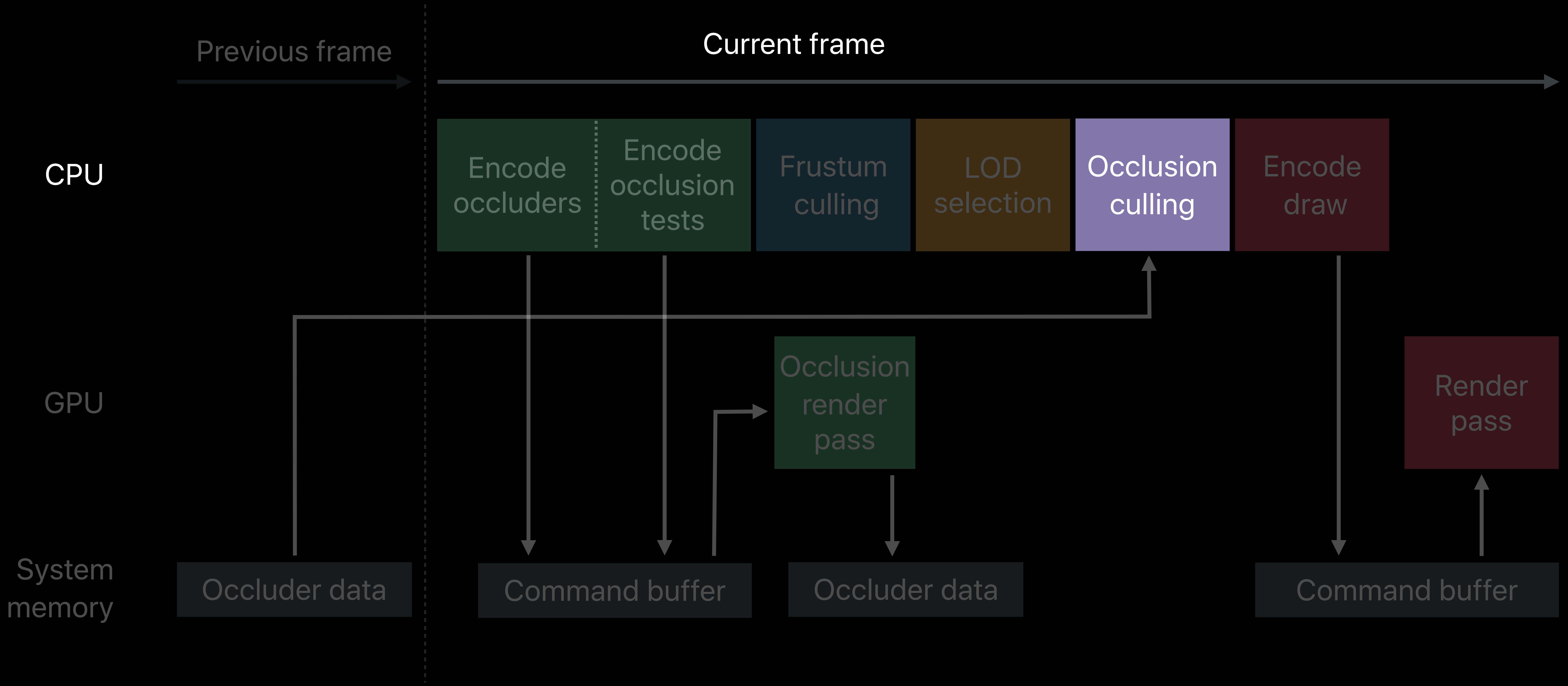
Traditional CPU Driven Render Loop



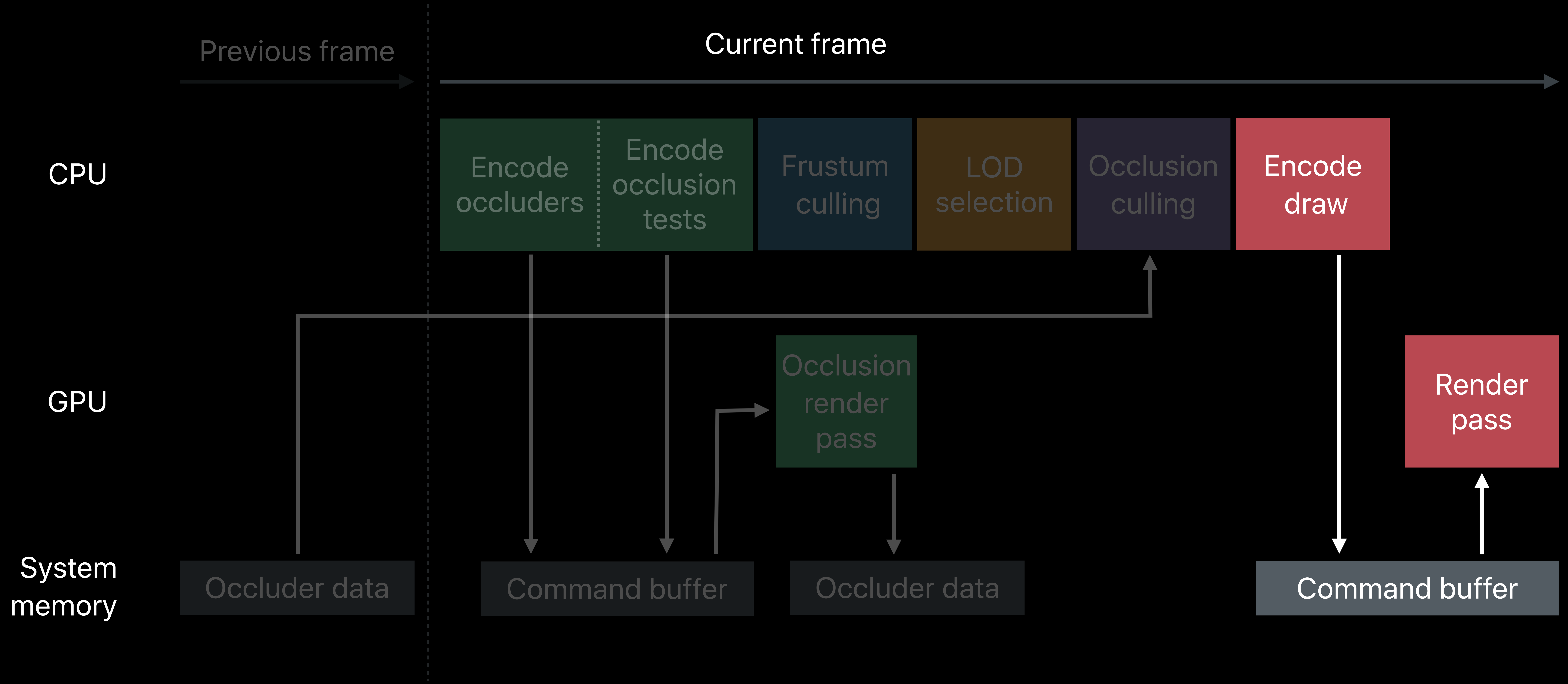
Traditional CPU Driven Render Loop



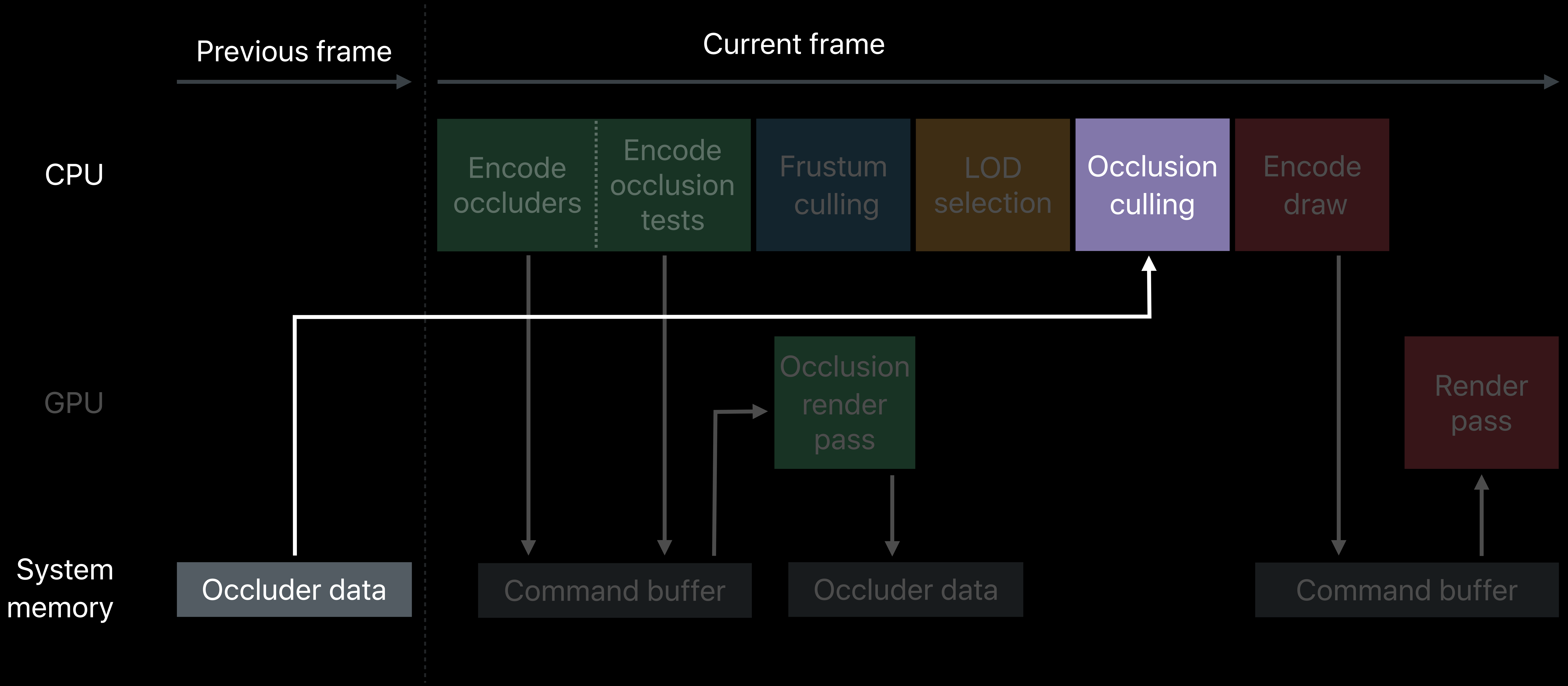
Traditional CPU Driven Render Loop



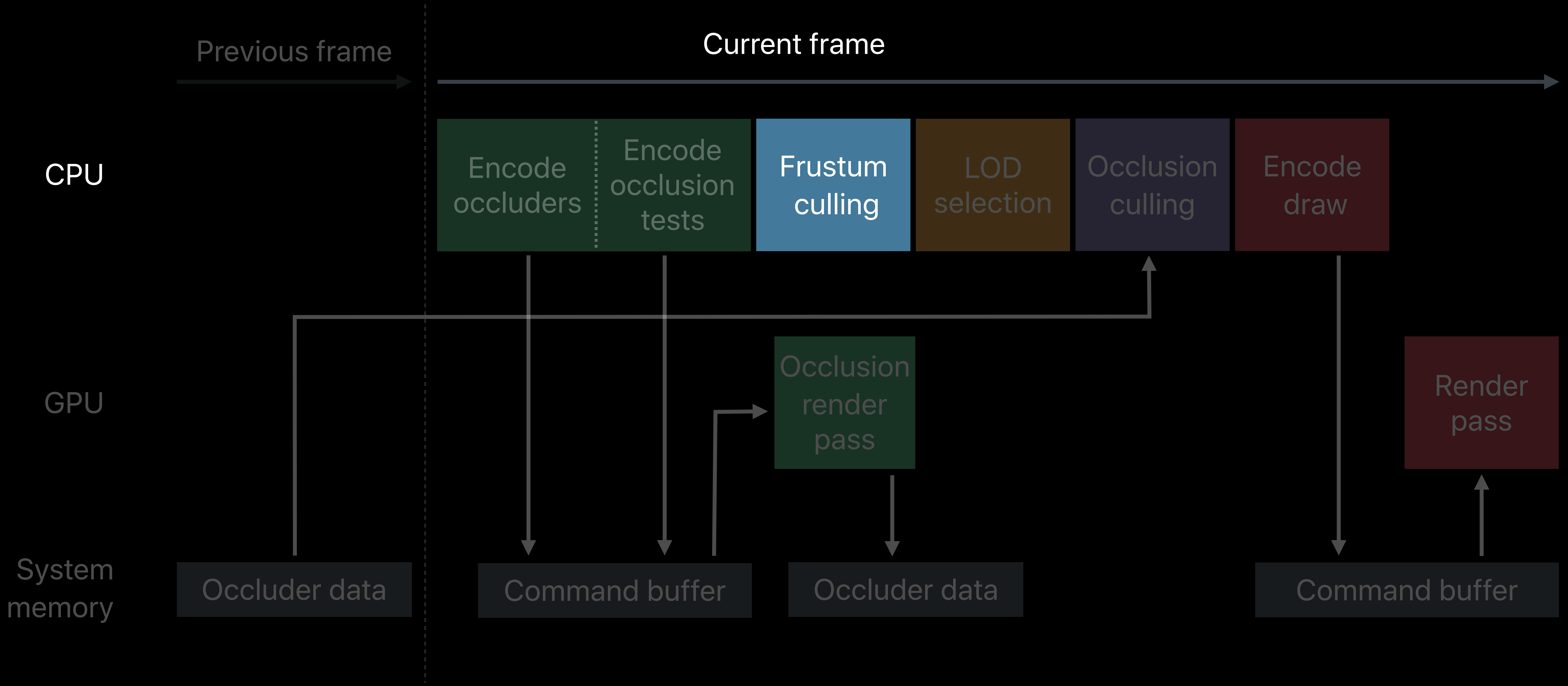
Traditional CPU Driven Render Loop



Traditional CPU Driven Render Loop



Traditional CPU Driven Render Loop



Serial Execution on the CPU

Parallel Execution on the CPU

Parallel Execution on the CPU

Parallel Execution on the GPU

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

GPU Driven Render Loop

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 Object Model Example

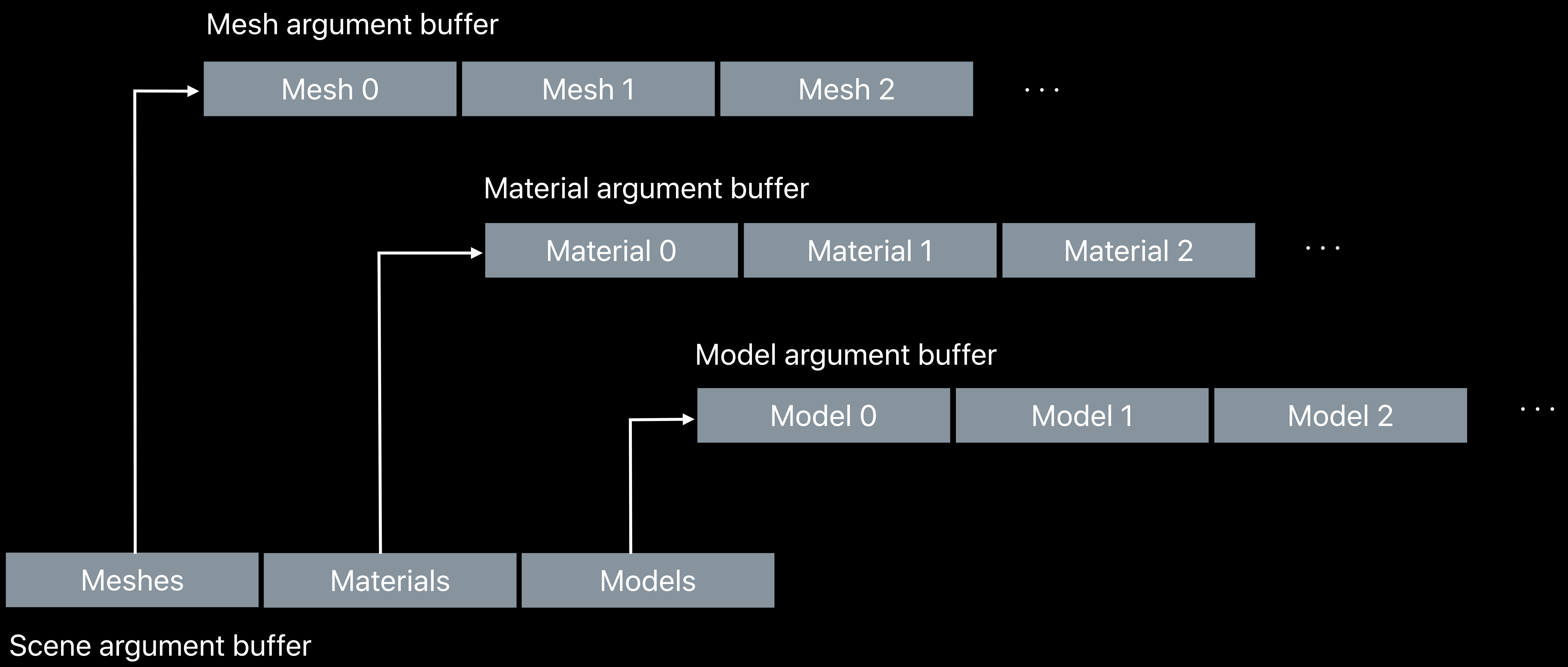
Scene Object Model Example

Scene Object Model Example

Scene Object Model Example

Scene Object Model Example

Scene Object Model with Argument Buffers



```
// 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           [[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            [[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, ...);
```

```
// 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

```
        ...

        // 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, ...);
  }
}
```

```swift
// 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)
```

```swift
// 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)
```

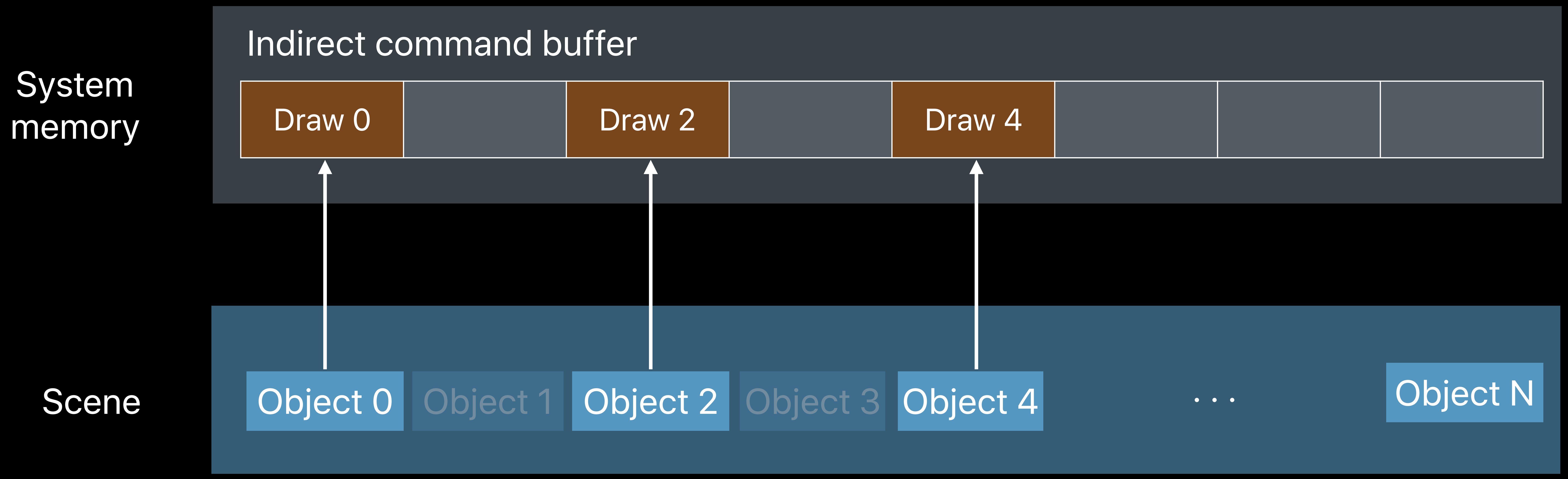
```swift
// Executing the ICB

// 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)
```

```swift
// 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)
```

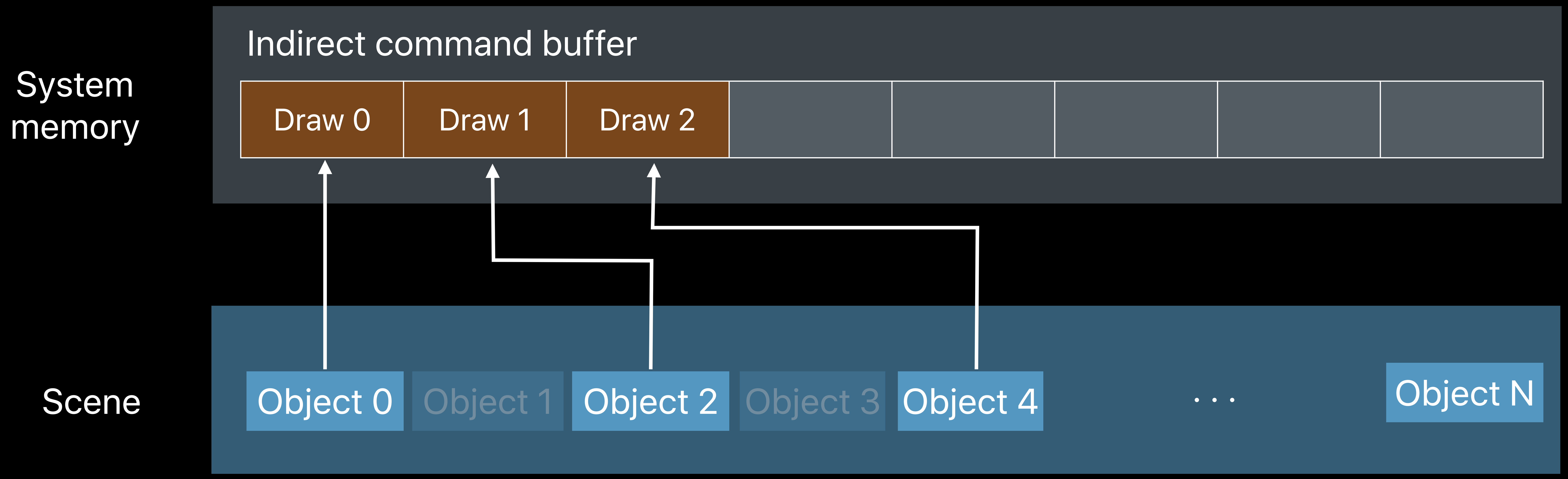
```swift
// Compute dispatch for additional occluder data
computeEncoder.dispatchThreadgroups(...)

// 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)
```

```swift
// 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)
```

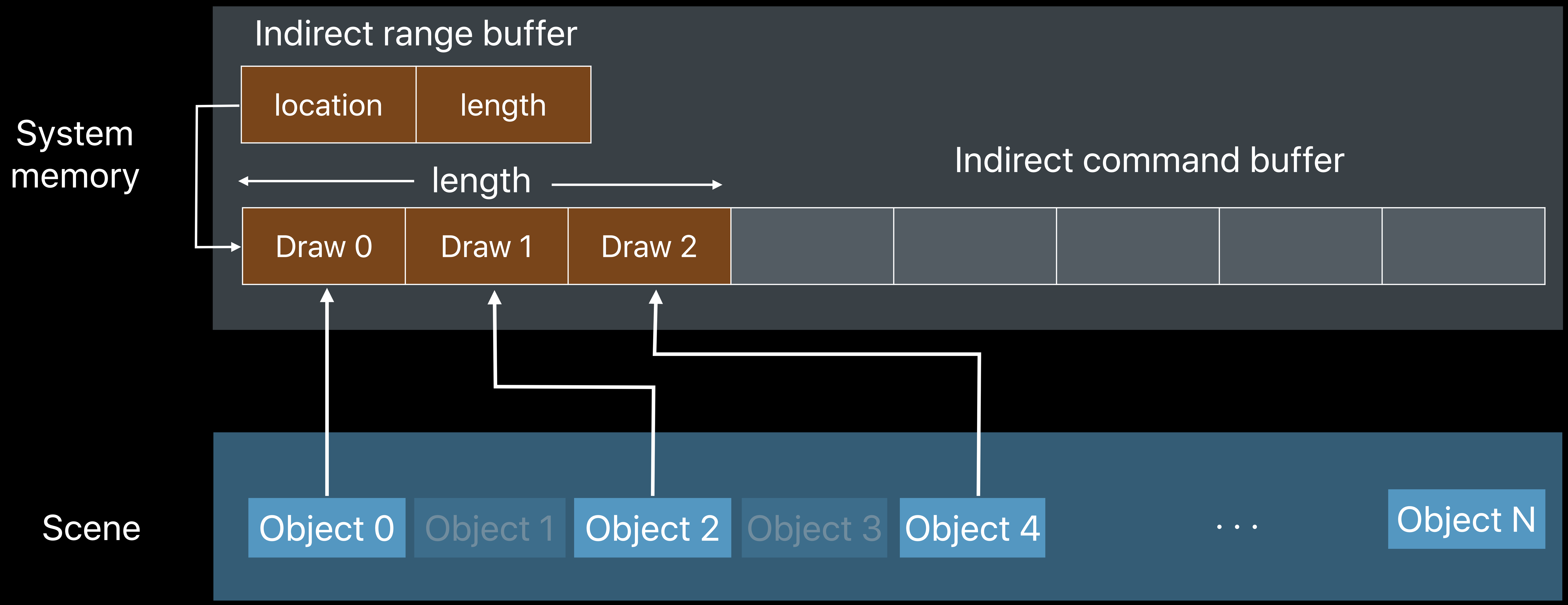
Sparse Encoding



Packed Encoding



Indirect Range Execution



```
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              [[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, ...));
    ...
```

```swift
// 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, ...)
```

```swift
// 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, ...)
```

```swift
// 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, ...)
```

```swift
// 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, ...)
```

```swift
// 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

NEW

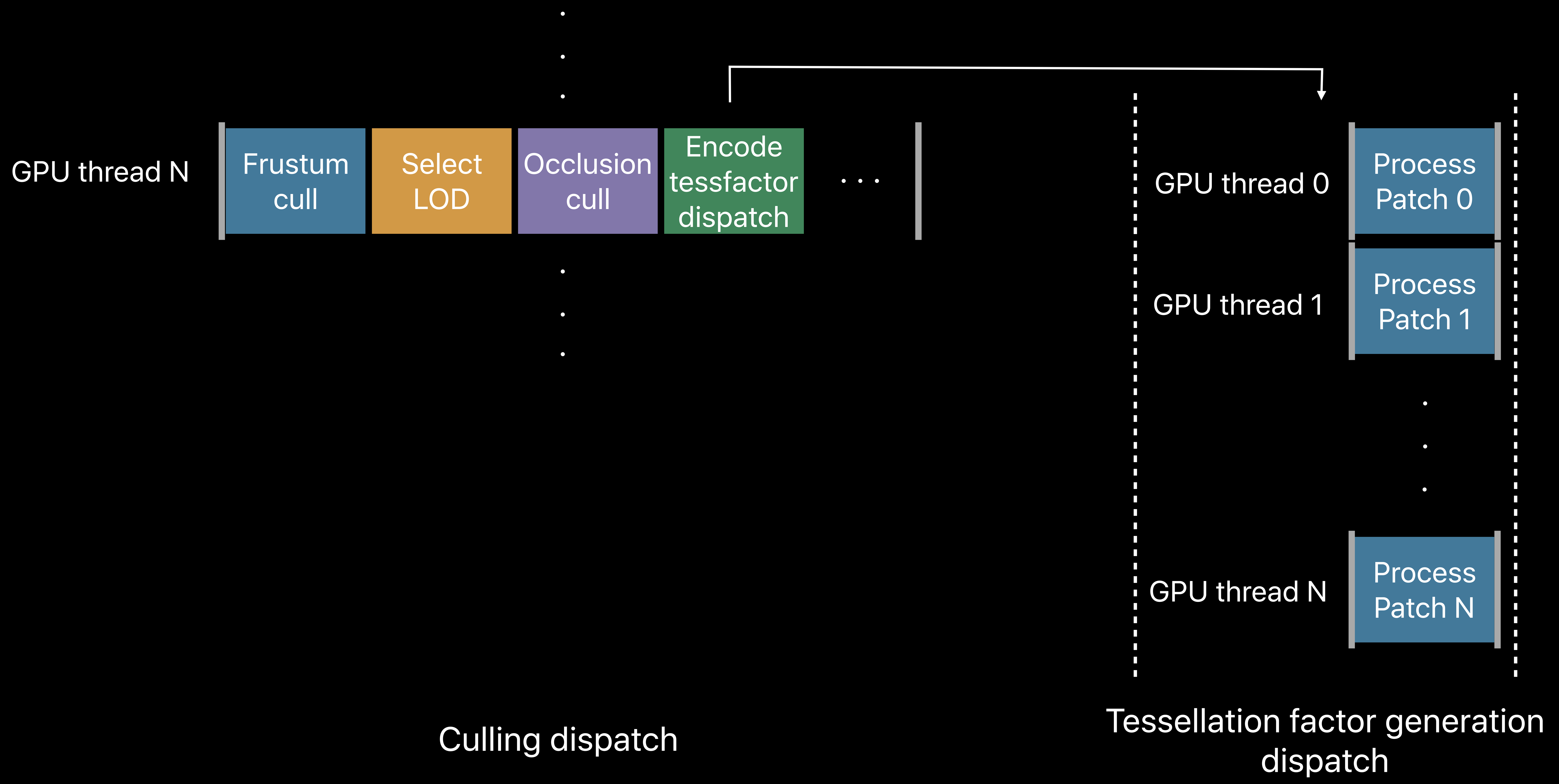
Allows GPU to build compute dispatches

Compute ICBs can be reused or modified

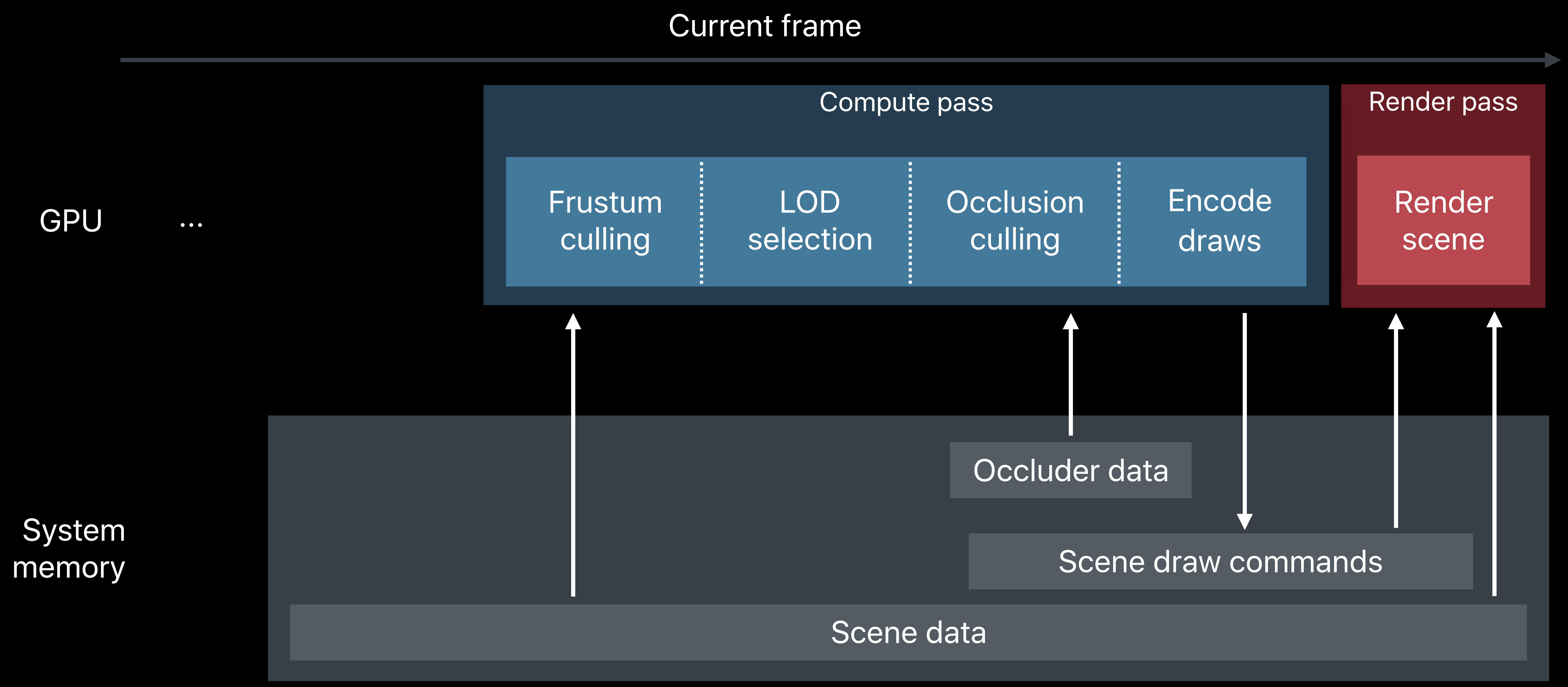
Both render and compute can now be driven on the GPU

Per Patch Tessellation Factors

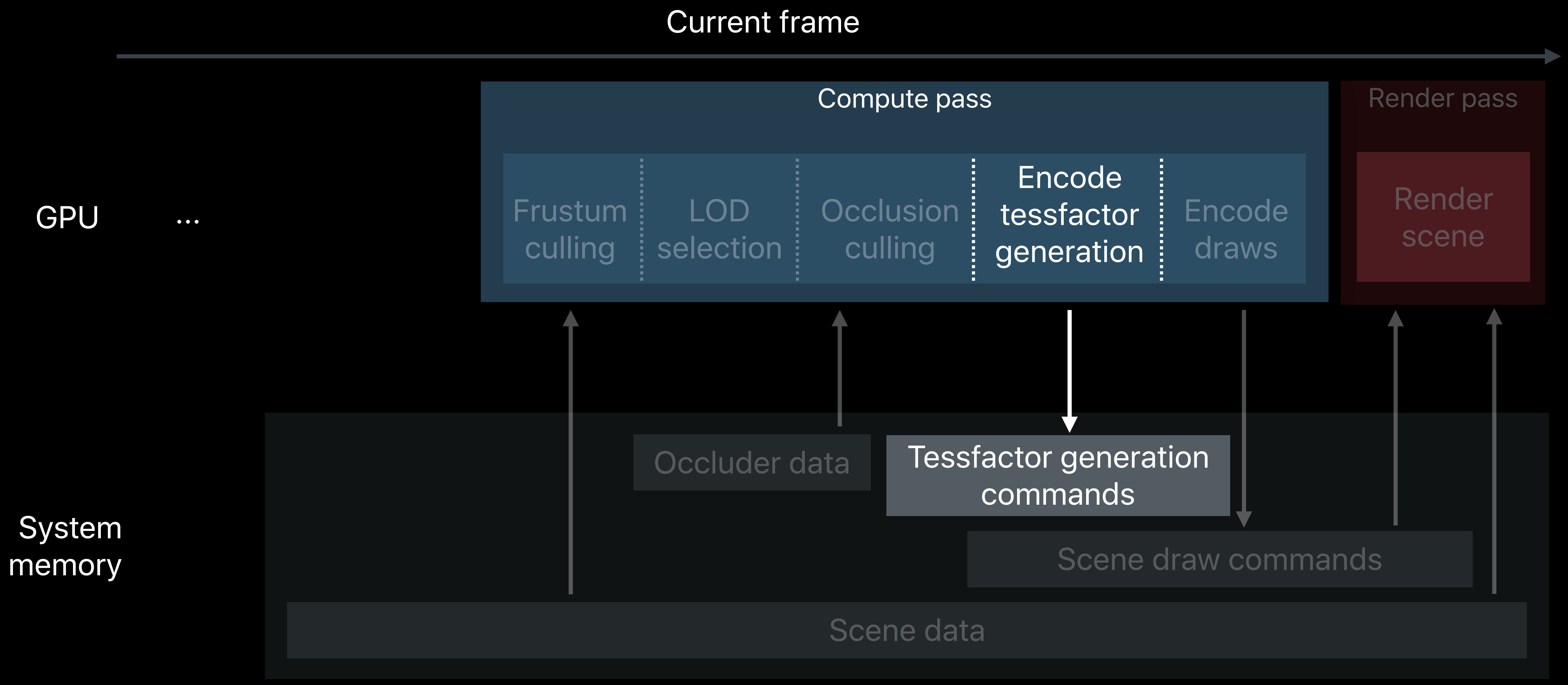
Per Patch Tessellation Factors



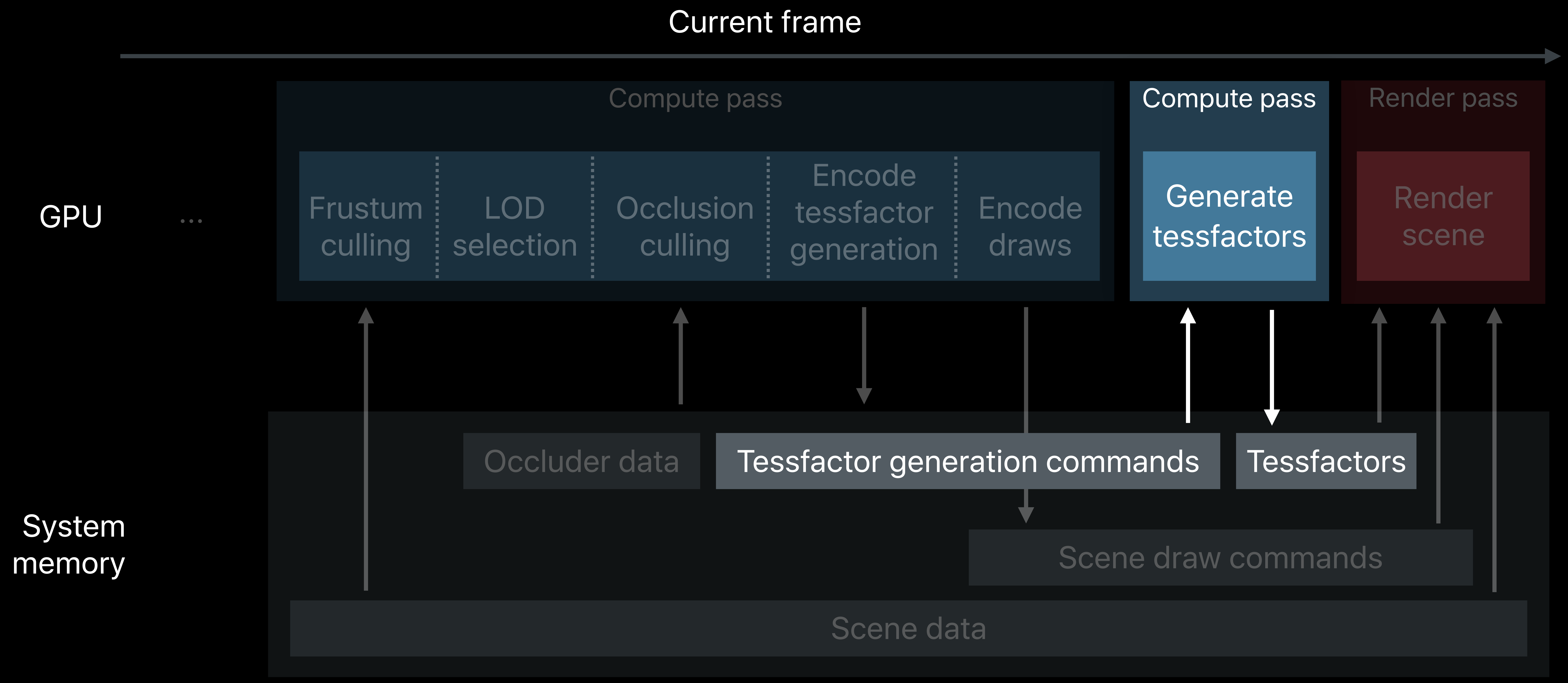
Per Patch Tessellation Factors



Per Patch Tessellation Factors



Per Patch Tessellation Factors




BOULANGERIE


BOULANGERIE


Restaurant
Le Petit Coin
MENU


Restaurant
MENU


Restaurant


Restaurant
Le Petit Coin

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

Apple Family

NEW

	GPU	iPhone	iPad	Apple TV
Apple 1	Apple A7	iPhone 5s	iPad mini 3, iPad Air	
Apple 2	Apple A8	iPhone 6, iPhone 6 Plus iPod Touch	iPad mini 4, iPad Air 2	Apple TV
Apple 3	Apple A9 Apple A10	iPhone 6S, iPhone 6S Plus iPhone 7, iPhone 7 Plus, iPhone SE	iPad Pro (2015), iPad Pro (2016), iPad (2018), iPad Pro (2017)	Apple TV 4K
Apple 4	Apple A11	iPhone 8, iPhone 8 Plus, iPhone X		
Apple 5	Apple A12	iPhone XS, iPhone XS Max, iPhone XR	iPad mini (2019), iPad Air (2019), iPad Pro (2018)	

Mac Family

NEW

	GPU	Mac	MacBook
Mac 1	Intel HD Graphics 4000 Intel Iris, Iris Pro Intel Iris Graphics 6100 Nvidia GeForce GT	iMac (2012+) Mac mini (2012+)	MacBook Air (2012+) MacBook Pro (2012+) MacBook (2015+)
Mac 2	Intel HD Graphics 5xx Intel Iris Plus Graphics 6xx AMD Radeon, Radeon Pro, FirePro	iMac (2014+) iMac Pro (2017+) Mac Pro (late 2013)	MacBook (2016+) MacBook Pro (2016+)

Common Family

NEW

Common 1	Universally supported features
Common 2	Indirect Draw/Dispatch, Counting Occlusion Queries, Tessellation, Read/Write Buffer Arguments, Arrays of Textures/Samplers, Compressed Volume Textures, Metal Performance Shaders, and more
Common 3	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

iPad Apps for Mac Family

NEW

iOSMac 1	Common 2, BC Pixel Formats, Managed Textures, Cube Texture Arrays, Read/Write Textures Tier 1, Layered Rendering, Multiple Viewports/Scissors, Indirect Tessellation
iOSMac 2	Common 3, BC Pixel Formats, Managed Textures

```swift
// 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) {
        …
    }
}
```

```swift
// 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) {
        …
    }
}
```

```swift
// 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) {
        …
    }
}
```

```swift
// 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) {
        …
    }
}
```

```swift
// 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

Features	Limits
Depth24Stencil8	MSAA Sample Counts
Raster Order Groups	Threadgroup Memory Length
Read-Write Texture Tier 2	Linear Texture Alignment
Programmable Sample Positions	Argument Buffer Sampler Counts

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

Metal Lab	Tuesday, 12:00
Metal Lab	Friday, 9:00
Delivering Optimized Metal Apps and Games	Wednesday, 11:00

WWDC19