Protocol

MTLRenderCommandEncoder

An interface that encodes a render pass into a command buffer, including all its draw calls and configuration.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.1+ | macOS 10.11+ | tvOS | visionOS 1.0+

```
protocol MTLRenderCommandEncoder : MTLCommandEncoder
```

Mentioned in

- Understanding the Metal 4 core API
- Improving CPU performance by using argument buffers
- Improving rendering performance with vertex amplification
- Sampling GPU data into counter sample buffers
- Setting up a command structure

Overview

The `MTLRenderCommandEncoder` protocol defines an interface that configures and encodes a render pass. Use a render pass to draw a scene, or a component within a scene, to its render *attachments*, the outputs of a render pass. You can use various approaches to render to those outputs, including techniques that apply the following:

- Primitive drawing
- Mesh drawing
- Ray tracing
- Tile shaders dispatching

To create an `MTLRenderCommandEncoder` instance, call the `makeRenderCommandEncoder(descriptor:)` of an `MTLCommandBuffer` instance, or the `makeRenderCommandEncoder()` method of an `MTLParallelRenderCommandEncoder` instance.

To configure the render pass for your first drawing commands, start with a pipeline state by passing an `MTLRenderPipelineState` instance to the encoder's `setRenderPipelineState(_:)` method. You create the pipeline states your render pass needs, typically ahead of time, by calling one or more `MTLDevice` methods (see Pipeline state creation).

Tip

Avoid visual stutter by creating pipeline states at a noncritical time, such as during launch, because of the time it can take to make them.

Set any other applicable parts of the encoder's configuration by calling the methods on the Render pass configuration page. For example, you may need to configure the pass's viewport, its scissor rectangle, and the settings for depth and stencil tests.

Assign resources, such as buffers and textures, for the shaders that depend on them. For more information, see the shader-specific pages in the resource preparation section, such as Vertex shader resource preparation commands and Fragment shader resource preparation commands. If your shaders access resources through an argument buffer, ensure the pass makes those resources *resident* by loading those resources resident in GPU memory. You do this by calling the methods on the Argument buffer resource preparation commands page.

Encode drawing commands by calling the methods on the Render pass drawing commands page after you configure the state and resources the commands depend on. The encoder maintains its current state and applies them to all subsequent draw commands. For drawing commands that need different states or resources, reconfigure the render pass appropriately and then encode those draw commands. Repeat the process for each batch of drawing commands that depend on the same render pass configuration and resources.

When you finish encoding the render pass's commands, finalize it into the command buffer by calling the encoder's `endEncoding()` method.

Topics

Encoding configuration commands

Manage the render pass's overall state.

Render pass configuration

Set a render pass's pipeline state, attachment actions, viewports, and so on, that affect subsequent drawing commands.

Encoding resource preparation commands

Load buffers, textures, and other resources into GPU memory for each shader type, including mesh, object, vertex, fragment, and tile shaders.

- Mesh and object shader resource preparation commands

  Assign resources to mesh and object shaders, including buffers, textures, acceleration structures, sampler states, and function tables.

- Vertex shader resource preparation commands

  Assign resources to vertex shaders, including buffers, textures, acceleration structures, sampler states, and function tables.

- Fragment shader resource preparation commands

  Assign resources to fragment shaders, including buffers, textures, acceleration structures, sampler states, and function tables.

- Tile shaders resource preparation commands

  Assign resources to tile shaders, including buffers, textures, acceleration structures, sampler states, and function tables.

- Argument buffer resource preparation commands

  Load individual resources and multiple resources within a heap into GPU memory so that they're available to shaders through argument buffers.

Encoding draw commands

Render your content with draw commands that invoke the shaders of the pass's current pipeline state.

- Render pass drawing commands

  Render an image by drawing meshes, primitives, tessellation patches, and by dispatching tile shaders.

Encoding resource synchronization commands

Protect against data hazards for resources with a `hazardTrackingMode` property that's `MTLHazardTrackingMode.untracked` with memory fences and barriers.

```
func waitForFence(any MTLFence, before: MTLRenderStages)
```

Encodes a command that instructs the GPU to pause before starting one or more stages of the render pass until a pass updates a fence.

Required

```
func updateFence(any MTLFence, after: MTLRenderStages)
```

Encodes a command that instructs the GPU to update a fence after one or more stages, which signals passes waiting on the fence.

Required

```
func memoryBarrier(resources: [any MTLResource], after: MTLRenderStages
, before: MTLRenderStages)
```

Creates a memory barrier that enforces the order of write and read operations for specific resources.

```
func memoryBarrier(scope: MTLBarrierScope, after: MTLRenderStages,
before: MTLRenderStages)
```

Creates a memory barrier that enforces the order of write and read operations for specific resource types.

Required

Encoding commands that run indirect command buffers

Run a command that tells the GPU to run other commands in a buffer, typically for commands your app needs frequently.

```
func executeCommandsInBuffer(any MTLIndirectCommandBuffer, range: Range
<Int>)
```

Encodes a command that runs a range of commands from an indirect command buffer (ICB).

```
func executeCommandsInBuffer(any MTLIndirectCommandBuffer, indirect
Buffer: any MTLBuffer, offset: Int)
```

Encodes a command that runs an indirect range of commands from an indirect command buffer (ICB).

Encoding data sampling commands

Sample realtime data from the GPU’s hardware as it runs the render pass.

```
func sampleCounters(sampleBuffer: any MTLCounterSampleBuffer, sample
Index: Int, barrier: Bool)
```

Encodes a command that samples hardware counters during the render pass and stores the data into a counter sample buffer.

Required

Deprecated

Replace older symbols in this group with their newer equivalents.

Deprecated symbols

Review unsupported symbols and their replacements.

Instance Methods

`func setColorAttachmentMap(MTLLogicalToPhysicalColorAttachmentMap?)`

Sets the mapping from logical shader color output to physical render pass color attachments.

Required

`func setDepthTestBounds(ClosedRange<Float>)`

Configures the range for depth bounds testing.

`func setVertexBuffer((any MTLBuffer)?, offset: Int, attributeStride: Int, index: Int)`

Required

`func setVertexBufferOffset(offset: Int, attributeStride: Int, index: Int)`

Required

`func setVertexBuffers([(any MTLBuffer)?], offsets: [Int], attributeStrides: [Int], range: Range<Int>)`

`func setVertexBytes(UnsafeRawPointer, length: Int, attributeStride: Int, index: Int)`

Required

Relationships

Inherits From

`MTLCommandEncoder`, `NSObjectProtocol`

See Also

Encoding a render pass

`protocol MTL4RenderCommandEncoder`

Encodes a render pass into a command buffer, including all its draw calls and configuration.

`struct MTL4RenderEncoderOptions`

Custom render pass options you specify at encoder creation time.

`enum MTLTriangleFillMode`

Specifies how to rasterize triangle and triangle strip primitives.

`enum MTLWinding`

The vertex winding rule that determines a front-facing primitive.

`enum MTLCullMode`

The mode that determines whether to perform culling and which type of primitive to cull.

`enum MTLPrimitiveType`

The geometric primitive type for drawing commands.

`enum MTLIndexType`

The index type for an index buffer that references vertices of geometric primitives.

`enum MTLDepthClipMode`

The mode that determines how to deal with fragments outside of the near or far planes.

`enum MTLVisibilityResultMode`

The mode that determines what, if anything, the GPU writes to the results buffer, after the GPU executes the render pass.

`enum MTLVisibilityResultType`

This enumeration controls if Metal accumulates visibility results between render encoders or resets them.