API Collection

Render passes

Encode a render pass to draw graphics into an image.

Overview

A render pass processes and rasterizes geometry to one or more output attachments using a render pipeline.

Render passes consist of:

- A set of input resources, such as buffers and textures
- A render pipeline that configures the GPU that does work with the input resources
- Draw commands
- Vertex shaders, the GPU core functions that process and transform a scene's geometry
- An optional tessellation stage that adds fine details to a scene's geometry
- Fragment shaders, the GPU core functions that produce the final color values for each pixel
- Optional outputs that can include color, depth, and stencil attachments, and their load and store operations

See the Customizing render pass setup sample for implementation details.

Topics

Encoding a render pass

`protocol` `MTL4RenderCommandEncoder`

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

`protocol` `MTLRenderCommandEncoder`

An interface that 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.

Encoding a render pass in parallel

`protocol` `MTLParallelRenderCommandEncoder`

An instance that splits up a single render pass so that it can be simultaneously encoded from multiple threads.

`enum` `MTLLoadAction`

Types of actions performed for an attachment at the start of a rendering pass.

`enum` `MTLStoreAction`

Types of actions performed for an attachment at the end of a rendering pass.

`struct` `MTLStoreActionOptions`

Options that modify a store action.

Configuring a render command encoder

`class` `MTL4RenderPassDescriptor`

Describes a render pass.

`class` `MTLRenderPassDescriptor`

A group of render targets that hold the results of a render pass.

`class` `MTLRenderPassAttachmentDescriptor`

A render target that serves as the output destination for pixels generated by a render pass.

`class` `MTLRenderPassColorAttachmentDescriptorArray`

An array of render pass color attachment descriptor objects.

`class` `MTLRenderPassColorAttachmentDescriptor`

A color render target that serves as the output destination for color pixels generated by a render pass.

`struct` `MTLClearColor`

An RGBA value used for a color pixel.

`class` `MTLRenderPassDepthAttachmentDescriptor`

A depth render target that serves as the output destination for depth pixels generated by a render pass.

`enum` `MTLMultisampleDepthResolveFilter`

Filtering options for controlling an MSAA depth resolve operation.

`class` `MTL4RenderPipelineColorAttachmentDescriptorArray`

An array of color attachment descriptions for a render pipeline.

`class` `MTLTileRenderPipelineColorAttachmentDescriptorArray`

An array of color attachment descriptors for the tile render pipeline.

`class` `MTLRenderPassStencilAttachmentDescriptor`

A stencil render target that serves as the output destination for stencil pixels generated by a render pass.

`enum` `MTLMultisampleStencilResolveFilter`

Constants used to control the multisample stencil resolve operation.

`class` `MTLRenderPassSampleBufferAttachmentDescriptorArray`

An array of sample buffer attachments for a render pass.

`class` `MTLRenderPassSampleBufferAttachmentDescriptor`

A description of where to store GPU counter information at the start and end of a render pass.

`class` `MTLLogicalToPhysicalColorAttachmentMap`

Allows you to easily specify color attachment remapping from logical to physical indices.

`struct` `MTLDispatchThreadsIndirectArguments`

Render pipeline states

`protocol` `MTLRenderPipelineState`

An interface that represents a graphics pipeline configuration for a render pass, which the pass applies to the draw commands you encode.

`class` `MTL4RenderPipelineDescriptor`

Groups together properties to create a render pipeline state object.

`class` `MTLRenderPipelineDescriptor`

An argument of options you pass to a GPU device to get a render pipeline state.

`class` `MTLRenderPipelineFunctionsDescriptor`

A collection of functions for updating a render pipeline.

`class` `MTL4MeshRenderPipelineDescriptor`

Groups together properties you use to create a mesh render pipeline state object.

`class` `MTLMeshRenderPipelineDescriptor`

An object that configures new render pipeline state objects for mesh shading.

`class` `MTLPipelineBufferDescriptor`

The mutability options for a buffer that a render or compute pipeline uses.

`class` `MTLPipelineBufferDescriptorArray`

An array of pipeline buffer descriptors.

`class` `MTL4RenderPipelineColorAttachmentDescriptor`

`class` `MTLRenderPipelineColorAttachmentDescriptor`

A color render target that specifies the color configuration and color operations for a render pipeline.

`class` `MTLRenderPipelineColorAttachmentDescriptorArray`

An array of render pipeline color attachment descriptor objects.

`class` `MTL4TileRenderPipelineDescriptor`

Groups together properties you use to create a tile render pipeline state object.

`class` `MTLTileRenderPipelineDescriptor`

An object that configures new render pipeline state objects for tile shading.

`class` `MTLTileRenderPipelineColorAttachmentDescriptor`

A description of a tile-shading render pipeline's color render target.

`struct` `MTLPipelineOption`

Options that determine how Metal prepares the pipeline.

`class` `MTL4RenderPipelineBinaryFunctionsDescriptor`

Allows you to specify additional binary functions to link to each stage of a render pipeline.

`class` `MTL4RenderPipelineDynamicLinkingDescriptor`

Groups together properties that provide linking properties for render pipelines.

Dynamic render pipeline states

`struct` `MTLViewport`

A 3D rectangular region for the viewport clipping.

`struct` `MTLScissorRect`

A rectangle for the scissor fragment test.

`struct` `MTLVertexAmplificationViewMapping`

An offset applied to a render target index and viewport index.

`struct` `MTLQuadTessellationFactorsHalf`

The per-patch tessellation factors for a quad patch.

`struct MTLTriangleTessellationFactorsHalf`

The per-patch tessellation factors for a triangle patch.

Render pass inputs

`class MTLVertexDescriptor`

An instance that describes how to organize and map data to a vertex function.

`class MTLVertexAttributeDescriptor`

An object that determines how to store attribute data in memory and map it to the arguments of a vertex function.

`class MTLVertexAttributeDescriptorArray`

An array of vertex attribute descriptor instances.

`class MTLVertexBufferLayoutDescriptor`

An object that configures how a render pipeline fetches data to send to the vertex function.

`class MTLVertexBufferLayoutDescriptorArray`

An array of vertex buffer layout descriptor instances.

`let MTLBufferLayoutStrideDynamic: Int`

Render pass outputs

`protocol MTLDrawable`

A displayable resource that can be rendered or written to.

`typealias MTLDrawablePresentedHandler`

A block of code invoked after a drawable is presented.

Depth testing

{} Calculating primitive visibility using depth testing

Determine which pixels are visible in a scene by using a depth texture.

`protocol MTLDepthStencilState`

A depth and stencil state instance that specifies the depth and stencil configuration and operations used in a render pass.

`class` `MTLDepthStencilDescriptor`

An instance that configures new `MTLDepthStencilState` instances.

`class` `MTLStencilDescriptor`

An object that defines the front-facing or back-facing stencil operations of a depth and stencil state object.

Rasterization settings

Rendering at different rasterization rates

Configure a rasterization rate map to vary rasterization rates depending on the amount of detail needed.

Creating a rasterization rate map

Define the rasterization rates for each part of your render target.

Rendering with a rasterization rate map

Create offscreen textures to hold intermediate rasterized data.

Scaling variable rasterization rate content

Use the rate map data to scale the content to fill your destination texture.

`class` `MTLRasterizationRateMapDescriptor`

An object that you use to configure new rasterization rate maps.

`protocol` `MTLRasterizationRateMap`

A compiled read-only instance that determines how to apply variable rasterization rates when rendering.

`typealias` `MTLCoordinate2D`

A coordinate in the viewport.

`func` `MTLCoordinate2DMake(Float, Float) -> MTLCoordinate2D`

Returns a new 2D point with the specified coordinates.

Optimizing techniques

Specifying drawing and dispatch arguments indirectly

Use indirect commands if you don't know your draw or dispatch call arguments when you encode the command.

- Rendering to multiple viewports in a draw command
  Select viewports and their corresponding scissor rectangles in your vertex shader.
- Rendering to multiple texture slices in a draw command
  Select a destination texture slice in your vertex shader.

Advanced multisampling

- Positioning samples programmatically
  Configure the position of samples when rendering to a multisampled render target.
- Storing data a pass makes with custom sample positions for a subsequent pass
  Inform Metal when your app uses programmable sample positions for its depth render targets or copies MSAA depth data.

Applying rendering techniques

- Drawing a triangle with Metal 4
  Render a colorful, rotating 2D triangle by running draw commands with a render pipeline on a GPU.
- Customizing render pass setup
  Render into an offscreen texture by creating a custom render pass.
- Setting load and store actions
  Set actions that define how a render pass loads and stores a render target.
- Improving rendering performance with vertex amplification
  Run draw commands that render to different outputs using the same vertex data multiple times.

See Also

Command encoders

- Compute passes
  Encode a compute pass that runs computations in parallel on a thread grid, processing and manipulating Metal resource data on multiple cores of a GPU.

- Machine-learning passes

  Add machine-learning model inference to your Metal app's GPU workflow.

- Blit passes

  Encode a block information transfer pass to adjust and copy data to and from GPU resources, such as buffers and textures.

- Indirect command encoding

  Store draw commands in Metal buffers and run them at a later time on the GPU, either once or repeatedly.

- Ray tracing with acceleration structures

  Build a representation of your scene's geometry using triangles and bounding volumes to quickly trace rays through the scene.