API Collection

Textures

Create and manage typed data your app uses to exchange information with its shader functions.

Overview

`MTLTexture` instances can serve as input and output resources to shader functions, as well as render pass destinations, or *render attachments*. Unlike buffers, textures define the underlying pixel type and structure. Textures can:

- Store 1-, 2-, or 3-dimensional data
- Contain several faces or layers
- Work as an array of texture data

Apps typically use textures to render details onto the surfaces in a scene or a 3D model. You can also use textures for post-processing pipelines, such as adding an advanced visual effect to an image before presenting it to a display.

Although textures can consume large amounts of memory, they also offer strategies, such as texture compression, that can save storage and memory bandwidth. Apple silicon GPUs support memoryless textures for transient render attachments that only need to exist for the duration of a render pass.

Topics

Texture basics

Understanding color-renderable pixel format sizes

Know the size limits of color render targets in Apple GPUs based on the target's pixel format.

Optimizing texture data

Optimize a texture's data to improve GPU or CPU access.

`protocol MTLTexture`

A resource that holds formatted image data.

`enum MTLTextureCompressionType`

`class MTLTextureDescriptor`

An instance that you use to configure new Metal texture instances.

`class MTKTextureLoader`

An object that creates textures from existing data in common image formats.

`class MTLSharedTextureHandle`

A texture handle that can be shared across process address space boundaries.

`enum MTLPixelFormat`

The data formats that describe the organization and characteristics of individual pixels in a texture.

Texture samplers

{} Creating and sampling textures

Load image data into a texture and apply it to a quadrangle.

`protocol MTLSamplerState`

An instance that defines how a texture should be sampled.

`class MTLSamplerDescriptor`

An object that you use to configure a texture sampler.

`struct MTLSamplePosition`

A subpixel sample position for use in multisample antialiasing (MSAA).

`enum MTLSamplerReductionMode`

Configures how the sampler aggregates contributing samples to a final value.

Texture mipmapping

- Improving texture sampling quality and performance with mipmaps

  Avoid texture-rendering artifacts and reduce the GPU's workload by creating smaller versions of a texture.

- Creating a mipmapped texture

  Decide whether a texture that you're creating needs mipmaps.

- Copying data into or out of mipmaps

  Specify which mipmaps that the data transfer affects.

- Generating mipmap data

  Create your mipmaps either when you author content or at runtime.

- Adding mipmap filtering to samplers

  Specify how the GPU samples mipmaps in your textures.

- Restricting access to specific mipmaps

  Set the range of mipmap levels that a sampler can access.

- Predicting which mips the GPU samples with level-of-detail queries

  Determine in advance which mipmap levels the GPU requires to sample a texture.

- Dynamically adjusting texture level of detail

  Defer generating or loading larger mipmaps until that level of detail is needed.

Sparse textures

- Managing sparse texture memory

  Take direct control of memory allocation for texture data by using sparse textures.

- Creating sparse heaps and sparse textures

  Allocate memory for sparse textures by creating a sparse heap.

- Converting between pixel regions and sparse tile regions

  Learn how a sparse texture's contents are organized in memory.

- Assigning memory to sparse textures

  Use a resource state encoder to allocate and deallocate sparse tiles for a sparse texture.

Reading and writing to sparse textures

Decide how to handle access to unmapped texture regions.

Estimating how often a texture region is accessed

Use texture access patterns to determine when you need to map a texture region.

`class MTLResourceStatePassDescriptor`

A configuration for a resource state pass, used to create a resource state command encoder.

`class MTLResourceStatePassSampleBufferAttachmentDescriptor`

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

`class MTLResourceStatePassSampleBufferAttachmentDescriptorArray`

An array of sample buffer attachments for a resource state pass.

`protocol MTLResourceStateCommandEncoder`

An encoder that encodes commands that modify resource configurations.

`struct MTLMapIndirectArguments`

The data layout for mapping sparse texture regions when using indirect commands.

Texture loading

`class MTKTextureLoader`

An object that creates textures from existing data in common image formats.

`struct Error`

Errors returned by the texture loader.

`struct Option`

Keys and values used to specify loading options.

`typealias Callback = ((any MTLTexture)?, (any Error)?) -> Void`

The signature for the block executed after an asynchronous loading operation for a single texture has completed.

`typealias ArrayCallback = ([any MTLTexture], (any Error)?) -> Void`

The signature for the block executed after an asynchronous loading operation for multiple textures has completed.

See Also

Resources

- Resource fundamentals

  Control the common attributes of all Metal memory resources, including buffers and textures, and how to configure their underlying memory.

- Buffers

  Create and manage untyped data your app uses to exchange information with its shader functions.

- Memory heaps

  Take control of your app’s GPU memory management by creating a large memory allocation for various buffers, textures, and other resources.

- Resource loading

  Load assets in your games and apps quickly by running a dedicated input/output queue alongside your GPU tasks.

- Resource synchronization

  Prevent multiple commands that can access the same resources simultaneously by coordinating those accesses with barriers, fences, or events.