API Collection

Resource loading

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

Overview

Metal 3 adds input/output command queues and buffers that make the most of a device's storage hardware, including flash storage and the unified memory architecture of Apple silicon, when available. When you run a dedicated input/output queue alongside your GPU tasks, you can synchronize them with Metal shared events. With this approach, you can minimize load screen times by fetching the essential assets first and streaming the rest as you need them. You can also start multiple input/output command buffers to load different asset batches and later cancel the ones you don't need. Ensure that time-sensitive assets, such as sound effects, load with lower latency by running those command buffers on higher-priority queues that you create.

First, create `MTLIOCommandQueue` instances by configuring an `MTLIOCommandQueueDescriptor` instance and passing it to an `MTLDevice` instance's `makeIOCommandQueue(descriptor:)` method.

Swift | Objective-C

```
// Create a Metal I/O command queue.
let commandQueueDescriptor = MTLIOCommandQueueDescriptor()

commandQueueDescriptor.type = .concurrent
commandQueueDescriptor.priority = .normal

let ioCommandQueue = try device.makeIOCommandQueue(descriptor:
                                                   commandQueueDescriptor)
```

For each queue, create one or more `MTLIOCommandBuffer` instances by calling the queue's `makeCommandBuffer()` or `makeCommandBufferWithUnretainedReferences()` method. For each command buffer, load the assets you want by calling any of the `MTLIOCommandBuffer` protocol's load methods. For example:

- The `load(_:offset:size:sourceHandle:sourceHandleOffset:)` method loads an asset into an `MTLBuffer`.
- The `load(_:slice:level:size:sourceBytesPerRow:sourceBytesPerImage:destinationOrigin:sourceHandle:sourceHandleOffset:)` method loads an asset into an `MTLTexture`.
- The `loadBytes(_:size:sourceHandle:sourceHandleOffset:)` method loads an asset, such as an audio file, into a CPU-accessible memory buffer.

Swift Objective-C

```
// Create a Metal I/O command buffer.
let ioCommandBuffer = ioCommandQueue.makeCommandBuffer()

// Encode a command that loads a texture.
ioCommandBuffer.load(texture,
                     slice: 0,
                     level: 0,
                     size: textureSize,
                     sourceBytesPerRow: bytesPerRow,
                     sourceBytesPerImage: bytesPerImage,
                     destinationOrigin: origin,
                     sourceHandle: fileHandle,
                     sourceHandleOffset: 0)

// Encode a command that loads a buffer.
ioCommandBuffer.load(buffer,
                     offset: 0,
                     size: bufferSize,
                     sourceHandle: fileHandle,
                     sourceHandleOffset: 0)

// Submit the command buffer to run.
ioCommandBuffer.commit()
```

For each asset, create an `MTLIOFileHandle` instance using the input/output command buffer's load methods. To create a file handle for your asset, call an `MTLDevice` instance's `make`

IOHandle(url:) or makeIOHandle(url:compressionMethod:) method.

Swift Objective-C

```
func createHandleForFile(at url: URL, with device: MTLDevice) -> MTLIOFileHandle? {
    return try? device.makeIOHandle(url: url)
}
```

> **Note**
>
> You must create each file handle using the same MTLDevice instance that created the MTLIOCommandQueue and MTLIOCommandBuffer instances that load the files.

To help minimize your app's storage footprint, compress your assets at development time. First, create a new compression context with the MTLIOCreateCompressionContext function. Then, add data for an asset to the compression context using the MTLIOCompressionContextAppendData(_:_:_:) function. Finally, call the MTLIOFlushAndDestroyCompressionContext(_:) function to save the context to a compressed file that you add to your project.

Topics

I/O command queues

`protocol` MTLIOCommandQueue

A command queue that schedules input/output commands for reading files in the file system, and writing to GPU resources and memory.

`class` MTLIOCommandQueueDescriptor

A configuration template you use to create a new input/output command queue.

`enum` MTLIOPriority

Designates the priority for a new input/output command queue.

`enum` MTLIOCommandQueueType

Designates the queue type for a new input/output command queue.

`protocol` MTLIOScratchBufferAllocator

A protocol your app implements to provide scratch memory to an input/output command queue.

`protocol` **MTLIOScratchBuffer**

A protocol your app implements that wraps a Metal buffer instance to serve as scratch memory for an input/output command queue.

I/O command buffers

`protocol` **MTLIOCommandBuffer**

A command buffer that contains input/output commands that work with files in the file systems and Metal resources.

`protocol` **MTLIOFileHandle**

Represents a raw or compressed file, such as a resource asset file in your app's bundle.

`typealias` **MTLIOCommandBufferHandler**

A convenience type that defines the signature of an input/output command buffer's completion handler.

`enum` **MTLIOStatus**

Represents the state of an input/output command buffer.

`enum` **Code**

The error codes for creating an input/output file handle.

`let` **MTLIOErrorDomain**`: String`

The domain for input/output command queue errors.

Asset compression

`func` **MTLIOCreateCompressionContext**`(String, MTLIOCompressionMethod, Int) -> MTLIOCompressionContext?`

Creates a compression context that you use to compress data into a single file.

`enum` **MTLIOCompressionMethod**

The compression codecs that Metal supports for input/output handles.

`func` **MTLIOCompressionContextDefaultChunkSize**`() -> Int`

Returns a compression chunk size you can use as a default for creating a compression context.

`typealias` **MTLIOCompressionContext**

A pointer that represents the state of a file compression session in progress.

`func MTLIOCompressionContextAppendData(MTLIOCompressionContext, UnsafeRawPointer, Int)`

Adds data to a compression context.

`func MTLIOFlushAndDestroyCompressionContext(MTLIOCompressionContext) -> MTLIOCompressionStatus`

Finishes compressing and saves the file that a compression context represents.

`enum MTLIOCompressionStatus`

Represents the final state of a compression context.

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.

- Textures

  Create and manage typed 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 synchronization

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