Article

Understanding the Metal 4 core API

Discover the features and functionality in the Metal 4 foundational APIs.

Overview

Metal 4 improves runtime performance and memory efficiency through its underlying implementations, while making it easier to adapt your apps and games from other platforms, such as DirectX and Vulkan.

Metal 4 introduces new types for existing concepts and several new ones, including:

- Command queues
- Command buffers
- Command encoders
- Command allocators
- Argument tables
- Texture view pools
- Next generation barriers

Metal 4 introduces several types with the `MTL4` prefix that are completely independent from the original `MTL` types they replace, such as `MTL4CommandQueue` versus `MTLCommandQueue`. Other types are communal to all versions of Metal.

Metal 4	Metal
`MTL4CommandQueue`	`MTLCommandQueue`
`MTL4CommandBuffer`	`MTLCommandBuffer`

Metal 4	Metal
MTL4RenderCommandEncoder	MTLRenderCommandEncoder
MTL4ComputeCommandEncoder	MTLComputeCommandEncoder MTLBlitCommandEncoder MTLAccelerationStructureCommandEncoder

At runtime, your app can detect whether the current system supports Metal 4. For devices that support Metal 4, you can create an `MTL4CommandQueue`, otherwise, create an `MTLCommandQueue`. The type of queue you create with determines which family of types you work with. For more information, see Work submission.

You can incrementally adopt Metal 4 over time, which is convenient for larger projects. Portions of your app can individually switch to submitting work to an `MTL4CommandQueue` instance. When applicable, an app can synchronize the work it sends to an `MTL4CommandQueue` with other parts of the app that send work to `MTLCommandQueue` instances. For more information, see Resource synchronization.

Command queues

Metal 4 introduces a new command queue protocol, `MTL4CommandQueue`, which reduces CPU runtime and memory overhead by sending work to the GPU when you commit a command buffer. This means your app can submit work from any thread. You create a Metal 4 command queue by calling an `MTLDevice` factory method, such as `makeMTL4CommandQueue()`.

Metal 4 command queues can commit multiple command buffers, as a group. Apps can encode subsets of GPU work to multiple command buffers — each on a separate worker thread. When the worker threads finish encoding to their respective command buffers, you send the command buffers to the GPU as a whole by committing it to an `MTL4CommandQueue` instance with one of its methods, such as `commit:count:`. This is similar to how you use an `MTLParallelRenderCommandEncoder`, but different in that you can also apply other types of work in addition to rendering.

You can synchronize work between command queues with `MTLSharedEvent` instances. Shared events work with any combination of `MTLCommandQueue` and `MTL4CommandQueue` instances. This interoperability makes it easier for you to:

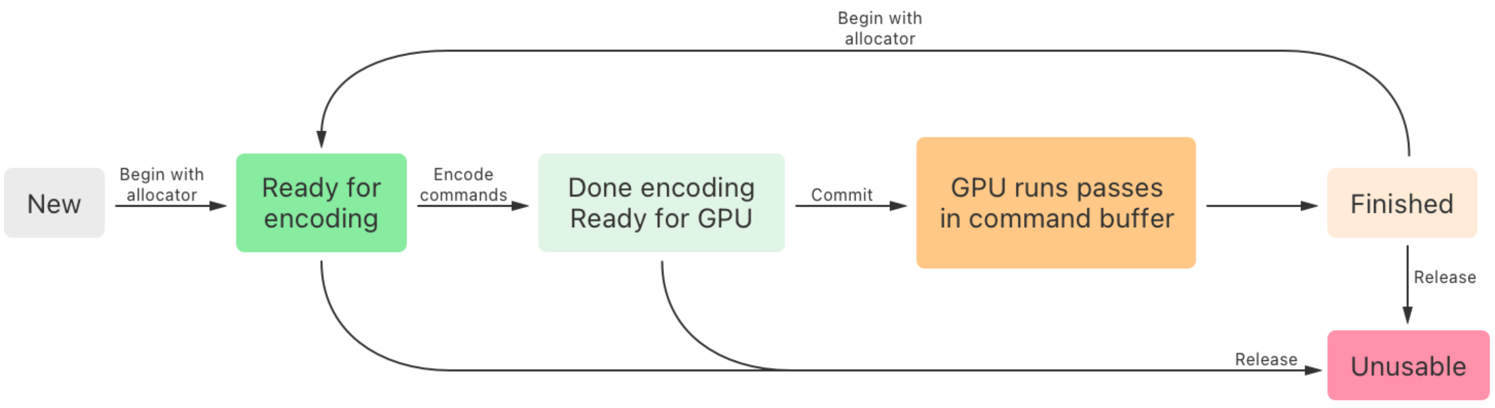
- Coordinate work between your app’s Metal 4 queues and existing Metal code.
- Transition to Metal 4 over time and incrementally adopt its features.

You can synchronize work within the same queue by adding a barrier, as Resource synchronization describes.

Command buffers

Metal 4 introduces `MTL4CommandBuffer`, which is more efficient and works differently than `MTLCommandBuffer` in the following ways:

- You create a Metal 4 command buffer by calling an `MTLDevice` factory method, such as `makeCommandBuffer()`, instead of from a queue.
- You submit a command buffer to any `MTL4CommandQueue` instance that belongs to the same device by calling one its methods, such as `commit:count:`, unlike `MTLCommandBuffer` which has its own `commit()` method.
- You can reuse and repurpose each command buffer indefinitely by starting over, encoding new commands, and committing it again, instead of allocating a new buffer.
- Unlike the default behavior of `MTLCommandBuffer`, you may need to consider a resource's retain count because each `MTL4CommandBuffer` instance doesn't create strong references to resources. This is similar to creating an `MTLCommandBuffer` with the `makeCommandBufferWithUnretainedReferences()` method of an `MTLCommandQueue`.



After committing a command buffer to a queue, you can use it again by calling its `beginCommandBuffer(allocator:)` method. You can then encode commands to the buffer as if it were a new instance. This is different from previous versions of Metal that require you to create a new transient, single-use command buffer when you need to commit more work to a queue.

Command allocators

The *command allocator* is a companion type that provides memory for command buffers. You associate a command allocator with one command buffer at a time by calling its `beginCommandBuffer(allocator:)` method. When you finish encoding commands to a command buffer, you can apply the allocator to another command buffer by first calling the current command buffer's endCommandBuffer() method and then another command buffer's beginCommandBuffer(allocator:) method.

Each allocator manages the memory that your app needs to encode commands into the command buffer that you associate with it. Like command buffers, you create each new MTL4CommandAllocator instance by calling a factory method of an MTLDevice, such as makeCommandAllocator().

Your app can manage the memory that it requires by using a command allocator for each frame's work. When the GPU finishes the work for that frame, call the reset() method to release the memory for reuse.

Apps can render frames by reusing a series of allocators, one for each frame it might have in flight at the same time to begin working on the next frame.

For example, the sample code project, Drawing a triangle with Metal 4 (Hello Triangle), works with three frames at the same time:

Swift   Objective-C

```swift
/// The number of frames the renderer works with at the same time.
let kMaxFramesInFlight = 3

/// A class that renders each of the app's video frames.
class Metal4Renderer {

    /// The Metal device the renderer draws with by sending commands to it.
    let device: MTLDevice

    /// A command queue the app uses to send command buffers to the Metal device.
    let commandQueue: MTL4CommandQueue

    /// An array of allocators that store commands for each frame
    /// while the app encodes them and the GPU runs them.
    var commandAllocators: [MTL4CommandAllocator] = []

    /// A command buffer the app reuses to render each frame.
    let commandBuffer: MTL4CommandBuffer

    // ...
}

/// Draws a frame of content to a view's drawable.
/// - Parameter view: A view with a drawable that the renderer draws into.
```

```
func renderFrameToView(_ view: MTKView) {

    // ...

    // Get the next allocator in the rotation.
    let frameIndex: Int = Int(frameNumber) % kMaxFramesInFlight
    let frameAllocator = commandAllocators[frameIndex]

    // Prepare to use or reuse the allocator by resetting it.
    frameAllocator.reset()

    // Prepare to use or reuse the command buffer for the frame's commands.
    commandBuffer.beginCommandBuffer(allocator: frameAllocator)

    // ...
}
```

At any point, each in-flight frame is in a different part of its life cycle.

- The current frame is what the app displays until the GPU finishes rendering the next frame.
- Meanwhile, the GPU is rendering the first future frame from the most recent command buffers that the app submits to the GPU.
- The app encodes the second future frame — either on the CPU or GPU — and submits the frame when other frames advance to the next stage in their life cycle.

Command encoders

The *command encoder*, `MTL4CommandEncoder`, is a base protocol for other work-specific protocols that Metal provides, including:

- `MTL4MachineLearningCommandEncoder`
- `MTL4RenderCommandEncoder`
- `MTL4ComputeCommandEncoder`

The base command encoder protocol defines a different interface and default behavior than its earlier counterpart, `MTLCommandEncoder`. The most important difference with Metal 4 encoders is that they don't have methods that bind individual buffers, textures, and heaps. Instead, you configure the resource bindings in an argument table and then bind that table to one or more pipeline stages with a command encoder.

Use `MTL4MachineLearningCommandEncoder` to encodes inference commands that apply Core ML models into a command buffer, alongside your app's rendering and computation workloads. For more information, see Machine-learning passes.

The `MTL4RenderCommandEncoder` protocol is the equivalent to its earlier counterpart, `MTLRenderCommandEncoder`, and has most of the same rendering methods. `MTL4RenderCommandEncoder` differs from `MTLRenderCommandEncoder` by removing methods that manage resource bindings and residency sets, and methods that configure store-action options and tessellation. Instead, `MTL4RenderCommandEncoder` gives you the ability to:

- Add a residency set to either an encoder's command buffer, or the command queue you submit that command buffer to.
- Create an argument table, configure it with bindings to resources, and then assign it to an encoder that refers to those resources.
- Apply mesh shader techniques to replace tessellation functionality.

> **Note**
>
> Store-action options (see `MTLStoreActionOptions`) aren't available because they don't apply to Apple silicon GPUs.

`MTL4RenderCommandEncoder` also supports encoding a render pass across command buffers by:

- Suspending the work at the end of one render encoder
- Resuming the work after the beginning of the next render encoder in the sequence

This technique conceptually replaces the `MTLParallelRenderCommandEncoder` protocol and simplifies encoding a render pass in parallel with multiple threads because each thread can have its own render encoder instead of tying all of them to a single render encoder.

The `MTL4ComputeCommandEncoder` protocol is a new type that combines the functionality of its three predecessors:

- `MTLBlitCommandEncoder`
- `MTLComputeCommandEncoder`
- `MTLAccelerationStructureCommandEncoder`

Argument tables

Metal 4 introduces an *argument table* type that stores bindings to resources, such as data buffers, textures, and samplers, on an encoder's behalf. Argument tables can reduce your app's memory footprint because:

- Metal 4 encoders don't require memory for storing the binding tables for every resource type, at every stage.
- Each table consumes only the memory it needs to store its resource bindings.

Each `MTL4ArgumentTable` instance stores a list for each resource type, which your app creates and maintains.

- Create or reuse an `MTL4ArgumentTableDescriptor` instance.
- Configure how many bindings of each type it stores by configuring its properties, including `maxBufferBindCount` and `maxTextureBindCount`.
- Create an argument table by passing the descriptor instance to an `MTLDevice` factory method, such as `makeArgumentTable(descriptor:)`.
- Add or update bindings to the argument table by calling its methods, such as `setResource(_:bufferIndex:)` and `setSamplerState(_:index:)`.

Assign an argument table to one or more stages of a command encoder, and then the commands you encode with it can refer to the resources in the argument table's lists, such as textures and data buffers. You can also apply a single argument table to the stages of multiple encoders at the same time.

As your app adds render or dispatch work to a command buffer by calling an encoder's methods, the encoder looks up the resources that the method needs from the encoder's argument table.

The design adds flexibility for reducing your app's CPU and memory overhead. For example, in Metal 4 you can create a single argument table that stores bindings to resources that apply to multiple encoders, and then reuse that argument table indefinitely. This is different from previous Metal versions, where each encoder instance manages its own resource binding tables. This approach is efficient compared to previous Metal encoder types, where each encoder instance manages its own resource binding tables. In Metal 4, the memory and runtime savings add up with each communal resource your encoders share, and each time you assign the argument table to a new encoder.

> **Tip**
>
> Create and configure separate argument tables for your app's disparate types of work so that each table only manages the communal resources for similar or overlapping tasks.

Barriers

Earlier versions of Metal support tracking data hazards for textures and heaps you create with hazard tracking (see the `hazardTrackingMode` property of the `MTLTextureDescriptor` and `MTLHeapDescriptor` types, respectively).

In Metal 4, the framework considers all resources untracked. You need to synchronize pipeline stages that can concurrently access a resource if any of the shaders in these pipelines modify it. For example, apps commonly encode a pass that writes to a communal buffer that a later pass needs to read from to do its work, such as rendering to a texture.

One of the most efficient ways to synchronize work between two or more passes is to add a *barrier*. A barrier tells the GPU that it needs to avoid a race condition by delaying the start of a pipeline stage until a previous stage finishes, so that it's safe to access the results of that stage. For example, if an app encodes a compute pass that produces data that a subsequent render pass consumes in its fragment shader, the app needs to add a barrier between the compute pass's dispatch stage and the render pass's fragment stage. In that scenario, the barrier signals to the GPU that it needs to wait before running the fragment stage of the render pass's pipeline until the compute pass's dispatch stage finishes modifying the communal resources.

Texture view pools

Metal 4 introduces the `MTLTextureViewPool` protocol which creates lightweight texture views that can reduce your app's memory footprint compared to creating the equivalent instances of `MTLTexture`. Each `MTLTexture` instance is a heavyweight type that stores a texture's underlying data and metadata. Each texture also has an implicit *texture view*, which is the default format interpretation of the texture's underlying data. With a texture view pool, you can create lightweight texture views that interpret and access a texture's underlying data with a different format that its original. For example, you can create `MTLTexture` instance with its `pixelFormat` equal to `MTLPixelFormat.rgba32Uint`, and then create a new texture view of the same texture that interprets the underlying data as if its pixel format is `MTLPixelFormat.rg11b10Float`.

Every texture view has a unique `MTLResourceID`, which includes:

- Texture views you create with an `MTLTextureViewPool` instance's methods, which is the return value of those methods
- Implicit texture views that Metal assigns to each `MTLTexture` you create, which you can access with a texture's `gpuResourceID`)` property

The resource IDs that a texture pool creates is part of a contiguous range of values that belong to that pool. For example, for a texture view pool that has 20 texture views, you can get the resource ID of the fifth texture view by adding 4 to the first texture view's resource ID. Similarly, you can get the resource ID of the last (twentieth) texture view by adding 19 to the first texture view's resource ID.

You can reuse a resource ID within a texture view pool, such as when you no longer need it, by reassigning the index of that pool with another view of any texture.

A texture view pool has a contiguous range of `MTLResourceID` values that you can manage by creating lightweight texture view, each of which gets its own resource ID. You can repurpose any

ID in the pool to another view when you no longer need the view that it currently represents.

See Also

Essentials

{} Drawing a triangle with Metal 4

Render a colorful, rotating 2D triangle by running draw commands with a render pipeline on a GPU.

{} Performing calculations on a GPU

Use Metal to find GPUs and perform calculations on them.

{} Using Metal to draw a view’s contents

Create a MetalKit view and a render pass to draw the view’s contents.