Article

Using the Metal 4 compilation API

Control when and how you compile an app's shaders.

Overview

Metal 4 introduces a modular shader compilation API workflow that gives you more control over when, where, and how you compile your shaders. A major part of that workflow comes from a compiler protocol, which represents the compiler instances your app creates at runtime. The compiler protocol gives you the ability to monitor and control the shader compilation process.

For example, an app can use a compiler instance to reduce or eliminate runtime delays from compilation tasks, which can have an unpredictable duration at runtime, by deciding the best time to run those tasks. This reduction is useful for large-scale apps that have compilation tasks needing a non-trivial amount of time. You can build strategies that balance your app's size and performance with the time it takes to compile shaders during development, your app's runtime, or both.

Improve your app's performance

Metal 4 also introduces other shader compilation features that can improve your app's performance by helping you only build the pipeline configurations it needs, including:

Function descriptors
: Define a more modular workflow that separates compilation during development versus at runtime

Flexible render pipeline states
: Add the ability to compile most of the shader ahead of time and optionally compile other parts once your app knows those details

Color-attachment mapping

Configures where each of your pipeline's outputs send their data without recompiling the entire shader

You can incrementally adopt the Metal 4 compilation workflow over time. For you example, you might start by integrating your app's most critical pipelines with one or more dedicated compiler instances, and then iteratively migrate the remaining pipelines at convenient times.

As you convert more pipelines to Metal 4, you can increasingly leverage *harvesting* workflows, which are the techniques and APIs that capture, store, and reuse the pipeline states your app compiles. You can serialize a harvesting set into a binary archive, or to a pipeline script which you can precompile to a binary archive during development. Harvesting pipelines states can:

- Reduce the time your app spends compiling pipeline states at runtime
- Create opportunities for broad compatibility and longevity because both Metal 3 and 4 can load binary archives

Create a dedicated compiler

Control of the compilation process with the `MTL4Compiler` protocol, which provides direct, fine-grained control over shader compilation tasks. With Metal 4, your app can explicitly oversee more aspects of the compilation processes with a dedicated compiler instance, including the ability to:

- Decide when to compile each shader, improving predictability
- Create a dispatch queue, optionally with a specific quality of service (QoS)
- Monitor the progress of each compilation task and when it finishes
- Harvest shaders and pipeline states your app compiles on-device and save them to binary archives
- Save shaders and pipeline states that your app compiles on-device with an `MTL4PipelineDataSetSerializer`

Compilation requests you run on that dispatch queue, such as with `dispatch_sync` or `dispatch_async`, run at the queue's QoS. If the app runs a compilation directly, without using a dispatch queue, such as on the main thread, the framework runs the compilation on the same QoS as the app's calling thread.

Choose synchronous or asynchronous compilation

You can compile your shaders synchronous or asynchronously. *Synchronous* calls block the caller's thread until the compiler finishes, but *asynchronous* calls return immediately and finish running on a background thread or queue.

Smaller projects or early prototypes typically compile their shaders with synchronous calls for simplicity because it's easier to understand, implement, and avoid concurrency issues. But, compiling synchronously can affect the app's runtime performance and responsiveness, especially when shader compilation tasks take more time.

Larger projects, such as AAA games and professional graphics tools, typically compile their shaders with asynchronous calls because of their size and sensitivity to runtime performance. This approach avoids introducing long stalls that can impede high-priority tasks on the CPU, such as UI responsiveness, because you can choose a Quality-of-Service (QoS) level. Compiling asynchronously does add additional complexity because your app needs to manage the background work and synchronize it with foreground tasks.

Schedule compilation tasks

The schedule model in Metal 4 divides compilation work into different quality-of-service (QoS) levels, including `userInteractive` and `background` (see `DispatchQoS`). You create one or more queues, each with their own QoS level, and then submit each compilation task to a queue with an appropriate QoS for that task.

The `MTL4Compiler` protocol applies multithreading in macOS and iOS by default, which automatically scales to the hardware device it's running on. However, iOS conserves energy and memory by applying limitations to background compilation threads. You can check how your app's concurrency and memory consumption behaves by profiling your app on physical devices and tune how it applies multithreading and its QoS settings accordingly.

The system can help your app avoid compiler tasks blocking more critical tasks by raising or lowering a task's priority based on your app's needs. This approach solves scenarios that exhibit *priority inversion*, which is when a low-priority task indirectly blocks another task with a higher priority, by promoting the relevant, individual tasks on a lower-priority queue to a higher priority.

For example, if an app submits a task on a queue with a high QoS setting, such as `userInitiated`, but it's waiting on another task already in flight on another queue with a lower QoS, such as `background`, the system automatically promotes the in-flight task to the higher QoS, including `userInitiated`.

> **Note**
>
> In Metal 3, you can configure whether compilation tasks apply multithreading by setting the `shouldMaximizeConcurrentCompilation` property of an `MTLDevice` instance.

Save compile time and memory with flexible pipeline state properties

Staring in Metal 4, you can reduce your app's shader compilation time by defining and creating a generic, or *unspecialized*, pipeline state one time, such as at launch. After launch, your app can create multiple, task-specific *specialized* pipeline states from the the unspecialized state for each pair of a fragment shader with a vertex or mesh shader. For example, if your app creates multiple pipeline states that are mostly identical except for a few specific configuration details, you can create an unspecialized pipeline with these steps:

1. Create a pipeline descriptor.
2. Configure the descriptor properties that the app's pipeline states have in common with concrete values.
3. Configure the properties that vary across the various pipeline states to their unspecialized value, such as `MTLPixelFormat.unspecialized` for the `pixelFormat` property.
4. Create an unspecialized render pipeline state by passing the descriptor to a compiler instance's render pipeline factory method.

When your app needs to create a pipeline state that replaces unspecialized values with specific ones, it can create one from the unspecialized pipeline state by:

1. Configuring a new descriptor that provides concrete values for the properties that the unspecialized pipeline state's descriptor doesn't define.
2. Compiling that descriptor and the unspecialized pipeline state into specialized pipeline state.

Creating specialized pipeline states from unspecialized pipeline ones can improve your app's runtime performance because it doesn't need to recompile the common code each time. The technique can also save memory at runtime, especially with shaders that have more code in their main body, because it doesn't need to recreate communal state or recompile the shader's main body for each pipeline state specialization.

Improve GPU runtime performance with color-attachment mapping

Metal 4 introduces the ability to modify the way a render pass maps its pipeline's logical outputs to the specific, physical outputs on the GPU. This flexibility means you can create a render pipeline state that works with various render encoders, even if they each define their outputs differently. Instead of creating a series of nearly identical pipeline states that only differ by their output configurations, you can create a single pipeline state that *inherits* the logical-to-physical output mapping from each encoder you apply it to. This technique:

- Improves CPU runtime performance because your app only needs to compile one pipeline state instead of many
- Can improve GPU performance by consolidating render commands with various pipelines states into a single pass

See Also

Shader compilation and libraries

Shader libraries

Manage and load your app's Metal shaders.

Using function specialization to build pipeline variants

Create pipelines for different levels of detail from a common shader source.