Article

Developing Metal apps that run in Simulator

Prototype and test your Metal apps in Simulator.

Overview

In Xcode, you can write iOS, tvOS, and visionOS apps that use Metal and test them in Simulator, gaining the benefits of hardware acceleration on the Mac during development of your app. If you use frameworks built on top of Metal, such as Core Animation and UIKit, you also experience improved performance when testing your apps in Simulator.

Use Simulator to rapidly prototype and test application behavior, and to develop the basic rendering or compute capabilities of your app. Use a hardware device instead of Simulator to design your final Metal workflow. Design your app to run on actual hardware, and test on real devices to tune its performance.

Get the default device instance in Simulator

In Simulator, call the `MTLCreateSystemDefaultDevice()` function to get the default device instance, just as you do when running on device in iOS, tvOS, or visionOS. This returns a device instance that connects to Simulator.

Treat Simulator as a special device

Simulator doesn't try to exactly simulate the GPU from the iOS, tvOS, or visionOS device you're simulating. For example, if you're simulating iPhone 15, Simulator doesn't try to emulate the capabilities of an A16 Bionic graphics chip. Instead, Simulator translates any calls you make and directs them to the selected GPU on the host Mac.

Sometimes, this translation means that Simulator may support fewer features or different implementation limits than an actual Apple GPU. Simulator provides a device instance with capabilities similar to an Apple family 2 GPU (`MTLGPUFamily.apple2`), as described in the Metal Feature Set Tables. Because Simulator doesn't support many of the features in more recent Apple family GPUs, consider whether using Simulator is worth the effort for your app based on the Metal features you use. Test the `MTLDevice` instance at runtime to determine exactly which features it supports and what its limits are.

In some cases, like those described in the sections below, Metal doesn't provide an API that you can use to detect the limitations at runtime. In those situations, conditionalize your app's behavior for Simulator, as the following code shows:

Swift Objective-C

```
#if os(macOS) || targetEnvironment(simulator)
    msaaTextureDescriptor.sampleCount = 4
    msaaTextureDescriptor.storageMode = MTLStorageMode.private
#else
    msaaTextureDescriptor.sampleCount = 2
    msaaTextureDescriptor.storageMode = MTLStorageMode.shared
#endif
```

For more code examples, see Supporting Simulator in a Metal app.

Texture limitations

When working with textures in Simulator, follow these recommendations:

- Create a texture that shares storage with a buffer that uses private storage (see `MTLStorageMode.private`) with the `makeTexture(descriptor:offset:bytesPerRow:)` method. Call `minimumLinearTextureAlignment(for:)` to determine the alignment requirements for the texture data; the alignment requirements are different in Simulator. When setting the texture's `usage` property, you can't include `renderTarget` as one of the uses.
- Create depth, stencil, and MSAA textures only with private storage modes.
- Don't use a sample count of 2 for MSAA textures.
- Use only unified depth and stencil texture formats. Simulator doesn't support separate depth and stencil formats.
- Don't use the following pixel formats: `MTLPixelFormat.r8Unorm_srgb`, `MTLPixelFormat.b5g6r5Unorm`, `MTLPixelFormat.a1bgr5Unorm`, `MTLPixelFormat.abgr4Unorm`, `MTLPixelFormat.bgr5A1Unorm`, or any XR10 or YUV formats.

- Don’t render to textures with an `MTLPixelFormat.rgb9e5Float` format.
- Don’t write to sRGB textures in Simulator.

Constant buffer limitations

If you use buffers that use the constant address space as arguments to your shaders, follow these recommendations when running in Simulator:

- Don’t create constant buffers larger than 64 kilobytes.
- When you set arguments for the render or compute command, align constant buffer offsets to 256 bytes. Generally, iOS requires an alignment of 4 bytes. This difference may mean that you might need to arrange your data differently when running in Simulator.
- Don’t use more than 14 constant buffers as arguments to a render or compute pipeline.

Other limitations

Additional limitations that require you to do things differently when running in Simulator include:

- You can create heaps only with private storage modes.
- Simulator doesn’t support programmable blending. When creating render pipelines, you can’t read from color attachments in your fragment shader’s arguments. This may mean, for example, that you may have to perform additional render passes when running in Simulator.
- You can’t use Xcode to analyze the memory footprint or measure performance when running in Simulator.

Prototype your app with Simulator

Consider the following guiding principles to use Simulator effectively in your Metal app development process:

Use Simulator to prototype and iterate on your app’s workflow and behavior. For example, when prototyping a game, you only care about how the game plays, not whether the pixels match what renders on device or if the game uses the same approach to render its content. Similarly, in other Metal apps, you might need to iterate on your app’s user experience. Simulator lets you test app behavior without needing a device.

Don’t use Simulator to design your iOS, tvOS, or visionOS rendering engine. The features that Simulator supports is significantly different from the features Apple GPUs support. To get best performance and battery life on devices with Apple GPUs, you need to use Metal features that Simulator doesn’t support. To develop, test, and profile those code paths, you need to run on a device.

Decide whether to provide long-term support for Simulator. Maintaining a separate Metal path for Simulator takes time and effort. A large game development team can have many game designers and engine developers. Supporting Simulator lets designers work in Simulator to perfect gameplay while engineers work with devices to design the game engine and tune its performance. On a smaller team, you might find that your time is better spent focusing on device support rather than devoting resources to keep your game running in Simulator.

For more information about Simulator, see Devices and Simulator. For more information about the differences between testing on device and testing in Simulator, see Testing in Simulator versus testing on hardware devices.

See Also

Developer tools

{} Supporting Simulator in a Metal app

Configure alternative render paths in your Metal app to enable running your app in Simulator.

{} Capturing Metal commands programmatically

Invoke a Metal frame capture from your app, then save the resulting GPU trace to a file or view it in Xcode.

Logging shader debug messages

Print debugging messages that a shader generates using shader logging.

Improving your game's graphics performance and settings

Fix performance glitches and develop default settings for smooth experiences on Apple platforms using the powerful suite of Metal development tools.

Metal debugger

Debug and profile your Metal workload with a GPU trace.

Metal developer workflows

Locate and fix issues related to your app's use of the Metal API and GPU functions.

GPU counters and counter sample buffers

Retrieve runtime data from a GPU device by sampling one or more of its counters.

Metal debugging types

Create capture managers and capture scopes, and review a GPU device's log after it runs a command buffer.