Sample Code

Using Metal to draw a view's contents

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

Download

iOS 12.0+ | iPadOS 12.0+ | macOS 10.14+ | tvOS 12.1+ | Xcode 11.0+

Overview

In this sample, you'll learn the basics of rendering graphics content with Metal. You'll use the MetalKit framework to create a view that uses Metal to draw the contents of the view. Then, you'll encode commands for a render pass that erases the view to a background color.

> Note
>
> MetalKit automates windowing system tasks, loads textures, and handles 3D model data. See MetalKit for more information.

Prepare a MetalKit view to draw

MetalKit provides a class called `MTKView`, which is a subclass of `NSView` (in macOS) or `UIView` (in iOS and tvOS). `MTKView` handles many of the details related to getting the content you draw with Metal onto the screen.

An `MTKView` needs a reference to a Metal device object in order to create resources internally, so your first step is to set the view's `device` property to an existing `MTLDevice`.

```
_view.device = MTLCreateSystemDefaultDevice();
```

Other properties on `MTKView` allow you to control its behavior. To erase the contents of the view to a solid background color, you set its `clearColor` property. You create the color using the `MTLClearColorMake(_:_:_:_:)` function, specifying the red, green, blue, and alpha values.

```
_view.clearColor = MTLClearColorMake(0.0, 0.5, 1.0, 1.0);
```

Because you won't be drawing animated content in this sample, configure the view so that it only draws when the contents need to be updated, such as when the view changes shape:

```
_view.enableSetNeedsDisplay = YES;
```

Delegate drawing responsibilities

`MTKView` relies on your app to issue commands to Metal to produce visual content. `MTKView` uses the delegate pattern to inform your app when it should draw. To receive delegate callbacks, set the view's `delegate` property to an object that conforms to the `MTKViewDelegate` protocol.

```
_view.delegate = _renderer;
```

The delegate implements two methods:

- The view calls the `mtkView(_:drawableSizeWillChange:)` method whenever the size of the contents changes. This happens when the window containing the view is resized, or when the device orientation changes (on iOS). This allows your app to adapt the resolution at which it renders to the size of the view.

- The view calls the `draw(in:)` method whenever it's time to update the view's contents. In this method, you create a command buffer, encode commands that tell the GPU what to draw and when to display it onscreen, and enqueue that command buffer to be executed by the GPU. This is sometimes referred to as drawing a frame. You can think of a frame as all of the work that goes into producing a single image that gets displayed on the screen. In an interactive app, like a game, you might draw many frames per second.

In this sample, a class called `AAPLRenderer` implements the delegate methods and takes on the responsibility of drawing. The view controller creates an instance of this class and sets it as the view's delegate.

Create a render pass descriptor

When you draw, the GPU stores the results into *textures*, which are blocks of memory that contain image data and are accessible to the GPU. In this sample, the `MTKView` creates all of the textures you need to draw into the view. It creates multiple textures so that it can display the contents of one texture while you render into another.

To draw, you create a *render pass*, which is a sequence of rendering commands that draw into a set of textures. When used in a render pass, textures are also called *render targets*. To create a render pass, you need a render pass descriptor, an instance of `MTLRenderPassDescriptor`. In this sample, rather than configuring your own render pass descriptor, ask the MetalKit view to create one for you.

```
MTLRenderPassDescriptor *renderPassDescriptor = view.currentRenderPassDescriptor;
if (renderPassDescriptor == nil)
{
    return;
}
```

A render pass descriptor describes the set of render targets, and how they should be processed at the start and end of the render pass. Render passes also define some other aspects of rendering that are not part of this sample. The view returns a render pass descriptor with a single color attachment that points to one of the view's textures, and otherwise configures the render pass based on the view's properties. By default, this means that at the start of the render pass, the render target is erased to a solid color that matches the view's `clearColor` property, and at the end of the render pass, all changes are stored back to the texture.

Because a view's render pass descriptor might be `nil`, you should test to make sure the render pass descriptor object is non-`nil` before creating the render pass.

Create a render pass

You create the render pass by encoding it into the command buffer using a `MTLRenderCommandEncoder` object. Call the command buffer's `makeRenderCommandEncoder(descriptor:)` method and pass in the render pass descriptor.

```
id<MTLRenderCommandEncoder> commandEncoder = [commandBuffer renderCommandEncoderWit
```

In this sample, you don't encode any drawing commands, so the only thing the render pass does is erase the texture. Call the encoder's `endEncoding` method to indicate that the pass is complete.

```
[commandEncoder endEncoding];
```

Present a drawable to the screen

Drawing to a texture doesn't automatically display the new contents onscreen. In fact, only some textures can be presented onscreen. In Metal, textures that can be displayed onscreen are managed by *drawable objects*, and to display the content, you present the drawable.

`MTKView` automatically creates drawable objects to manage its textures. Read the `currentDrawable` property to get the drawable that owns the texture that is the render pass's target. The view returns a `CAMetalDrawable` object, an object connected to Core Animation.

```
id<MTLDrawable> drawable = view.currentDrawable;
```

Call the `present(_:)` method on the command buffer, passing in the drawable.

```
[commandBuffer presentDrawable:drawable];
```

This method tells Metal that when the command buffer is scheduled for execution, Metal should coordinate with Core Animation to display the texture after rendering completes. When Core Animation presents the texture, it becomes the view's new contents. In this sample, this means that the erased texture becomes the new background for the view. The change happens alongside any other visual updates that Core Animation makes for onscreen user interface elements.

Commit the command buffer

Now that you've issued all the commands for the frame, commit the command buffer.

```
[commandBuffer commit];
```

See Also

Essentials

Understanding the Metal 4 core API

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

{} 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.