Sample Code

Creating a custom Metal view

Implement a lightweight view for Metal rendering that's customized to your app's needs.

Download

iOS 12.0+ | iPadOS 12.0+ | macOS 10.13+ | tvOS 12.0+ | Xcode 12.0+

Overview

While MetalKit's `MTKView` provides significant functionality, allowing you to quickly get started writing Metal code, sometimes you want more control over how your Metal content is rendered. This sample app demonstrates how to create a simple Metal view derived directly from an `NSView` or `UIView`. It uses a `CAMetalLayer` object to hold the view's contents.

Configure the sample code project

This sample has targets for iOS, tvOS, and macOS. There are significant differences between apps that use UIKit and AppKit. Because of these differences, this sample creates two different classes. The iOS and tvOS versions of the sample use the `AAPLUIView` class, and the macOS version uses the `AAPLNSView` class. Both are derived from a common `AAPLView` class.

This sample provides a number of options you can enable when building the app, such as whether to animate the view's contents or handle updates through system events. You control these options by changing the preprocessor definitions in the `AAPLConfig.h` file.

```
// When enabled, rendering occurs on the main application thread.
// This can make responding to UI events during redraw simpler
// to manage because UI calls usually must occur on the main thread.
// When disabled, rendering occurs on a background thread, allowing
```

```
// the UI to respond more quickly in some cases because events can be
// processed asynchronously from potentially CPU-intensive rendering code.
#define RENDER_ON_MAIN_THREAD 1

// When enabled, the view continually animates and renders
// frames 60 times a second.  When disabled, rendering is event
// based, occurring when a UI event requests a redraw.
#define ANIMATION_RENDERING   1

// When enabled, the drawable's size is updated automatically whenever
// the view is resized. When disabled, you can update the drawable's
// size explicitly outside the view class.
#define AUTOMATICALLY_RESIZE  1

// When enabled, the renderer creates a depth target (i.e. depth buffer)
// and attaches with the render pass descritpr along with the drawable
// texture for rendering.  This enables the app properly perform depth testing.
#define CREATE_DEPTH_BUFFER   1
```

Configure the view with a Metal layer

For Metal to render to the view, the view must be backed by a `CAMetalLayer`.

All views in UIKit are layer backed. To indicate the type of layer backing, the view implements the `layerClass` class method. To indicate that your view should be backed by a `CAMetalLayer`, you must return the `CAMetalLayer` class type.

```
+ (Class) layerClass
{
    return [CAMetalLayer class];
}
```

In AppKit, you make the view layer backed by setting the view's `wantsLayer` property.

```
self.wantsLayer = YES;
```

This triggers a call to the view's `makeBackingLayer` method, which returns a CAMetalLayer object.

```
- (CALayer *)makeBackingLayer
{
```

```
    return [CAMetalLayer layer];
}
```

Render to the view

To render to the view, create an `MTLRenderPassDescriptor` object that targets a texture provided by the layer. The `AAPLRenderer` class stores the render pass descriptor in the `_drawableRenderPassDescriptor` instance variable. Most of the properties of this descriptor are set up automatically when you initialize the renderer. The code configures the render pass to clear the contents of the texture, and to store any rendered contents to the texture when the render pass completes.

```
_drawableRenderDescriptor = [MTLRenderPassDescriptor new];
_drawableRenderDescriptor.colorAttachments[0].loadAction = MTLLoadActionClear;
_drawableRenderDescriptor.colorAttachments[0].storeAction = MTLStoreActionStore;
_drawableRenderDescriptor.colorAttachments[0].clearColor = MTLClearColorMake(0, 1, 1
```

You also need to set the texture that the render pass renders into. Each time the app renders a frame, the renderer obtains a `CAMetalDrawable` from the Metal layer. The drawable provides a texture for Core Animation to present onscreen. The renderer updates the render pass descriptor to render to this texture:

```
id<CAMetalDrawable> currentDrawable = [metalLayer nextDrawable];

// If the current drawable is nil, skip rendering this frame
if(!currentDrawable)
{
    return;
}

_drawableRenderDescriptor.colorAttachments[0].texture = currentDrawable.texture;
```

The rest of the rendering code is similar to that found in other Metal samples. For an explanation of a typical rendering path, see Drawing a triangle with Metal 4.

Implement a render loop

To animate the view, the sample sets up a display link. The display link calls the view at the specified interval, synchronizing updates to the display's refresh interval. The view calls the renderer object to render a new frame of animation.

`AAPLUIView` creates a `CADisplayLink` in the `setupCADisplayLinkForScreen` method. Because you need to know which screen the window is on before creating the display link, you call this method when UIKit calls your view's `didMoveToWindow()` method. UIKit calls this method the first time the view is added to a window and when the view is moved to another screen. The code below stops the render loop and initializes a new display link.

```
- (void)setupCADisplayLinkForScreen:(UIScreen*)screen
{
    [self stopRenderLoop];

    _displayLink = [screen displayLinkWithTarget:self selector:@selector(render)];

    _displayLink.paused = self.paused;

    _displayLink.preferredFramesPerSecond = 60;
}
```

`AAPLNSView` uses a `CVDisplayLink` instead of a `CADisplayLink` because `CADisplayLink` is not available on macOS. `CVDisplayLink` and `CADisplayLink` API look different, but, in principle, have the same goal, which is to allow callbacks in sync with the display. `AAPLNSView` creates a `CVDisplayLink` in the `setupCVDisplayLinkForScreen` method. The `setupCVDisplayLinkForScreen` method is called from `viewDidMoveToWindow()`, which AppKit calls immediately after loading the view. If the view is moved to another screen, AppKit also calls `viewDidMoveToWindow`, and like the previous code for UIKit, the AppKit view must recreate the display link for the new screen.

```
- (BOOL)setupCVDisplayLinkForScreen:(NSScreen*)screen
{
#if RENDER_ON_MAIN_THREAD

    // The CVDisplayLink callback, DispatchRenderLoop, never executes
    // on the main thread. To execute rendering on the main thread, create
    // a dispatch source using the main queue (the main thread).
    // DispatchRenderLoop merges this dispatch source in each call
    // to execute rendering on the main thread.
    _displaySource = dispatch_source_create(DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0, dis
    __weak AAPLView* weakSelf = self;
    dispatch_source_set_event_handler(_displaySource, ^(){
        @autoreleasepool
        {
            [weakSelf render];
        }
```

```
    });
    dispatch_resume(_displaySource);

#endif // END RENDER_ON_MAIN_THREAD

    CVReturn cvReturn;

    // Create a display link capable of being used with all active displays
    cvReturn = CVDisplayLinkCreateWithActiveCGDisplays(&_displayLink);

    if(cvReturn != kCVReturnSuccess)
    {
        return NO;
    }

#if RENDER_ON_MAIN_THREAD

    // Set DispatchRenderLoop as the callback function and
    // supply _displaySource as the argument to the callback.
    cvReturn = CVDisplayLinkSetOutputCallback(_displayLink, &DispatchRenderLoop, (__

#else // IF !RENDER_ON_MAIN_THREAD

    // Set DispatchRenderLoop as the callback function and
    // supply this view as the argument to the callback.
    cvReturn = CVDisplayLinkSetOutputCallback(_displayLink, &DispatchRenderLoop, (__

#endif // END !RENDER_ON_MAIN_THREAD

    if(cvReturn != kCVReturnSuccess)
    {
        return NO;
    }

    // Associate the display link with the display on which the
    // view resides
    CGDirectDisplayID viewDisplayID =
        (CGDirectDisplayID) [self.window.screen.deviceDescription[@"NSScreenNumber"]

    cvReturn = CVDisplayLinkSetCurrentCGDisplay(_displayLink, viewDisplayID);

    if(cvReturn != kCVReturnSuccess)
    {
```

```
        return NO;
    }

    CVDisplayLinkStart(_displayLink);

    NSNotificationCenter* notificationCenter = [NSNotificationCenter defaultCenter];

    // Register to be notified when the window closes so that you
    // can stop the display link
    [notificationCenter addObserver:self
                           selector:@selector(windowWillClose:)
                               name:NSWindowWillCloseNotification
                             object:self.window];

    return YES;
}
```

The macOS version of this code performs a few additional steps. After creating the display link, it sets the callback and a parameter to pass to the callback. If you want rendering to happen on the main thread, it passes a dispatch source object; otherwise, it passes a reference to the view itself. Finally, it tells the display link which display the window is located on, and sets a notification to be called when the window is closed.

See Also

Render workflows

{} Using Metal to draw a view's contents

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

{} Drawing a triangle with Metal 4

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

{} Selecting device objects for graphics rendering

Switch dynamically between multiple GPUs to efficiently render to a display.

{} Customizing render pass setup

Render into an offscreen texture by creating a custom render pass.

{} Calculating primitive visibility using depth testing

Determine which pixels are visible in a scene by using a depth texture.

{} Encoding indirect command buffers on the CPU

Reduce CPU overhead and simplify your command execution by reusing commands.

{} Implementing order-independent transparency with image blocks

Draw overlapping, transparent surfaces in any order by using tile shaders and image blocks.

{} Loading textures and models using Metal fast resource loading

Stream texture and buffer data directly from disk into Metal resources using fast resource loading.

{} Adjusting the level of detail using Metal mesh shaders

Choose and render meshes with several levels of detail using object and mesh shaders.

{} Creating a 3D application with hydra rendering

Build a 3D application that integrates with Hydra and USD.

{} Culling occluded geometry using the visibility result buffer

Draw a scene without rendering hidden geometry by checking whether each object in the scene is visible.

{} Improving edge-rendering quality with multisample antialiasing (MSAA)

Apply MSAA to enhance the rendering of edges with custom resolve options and immediate and tile-based resolve paths.

{} Achieving smooth frame rates with a Metal display link

Pace rendering with minimal input latency while providing essential information to the operating system for power-efficient rendering, thermal mitigation, and the scheduling of sustainable workloads.