Article

Managing your game window for Metal in macOS

Set up a window and view for optimally displaying your Metal content.

Overview

With Metal, apps can leverage a GPU to quickly render complex scenes and run computational tasks in parallel. Your results accumulate into a `CAMetalLayer` that you can display onscreen using a window. By configuring your window correctly, your app can achieve optimal results by engaging direct-to-display for its Metal drawable. And, when not in full-screen mode, your game's content displays as expected in a window that works in familiar ways for people using macOS.

When a Metal drawable is direct-to-display, the hardware composites it directly to the display at a very low performance cost with a high-quality upscaling or downscaling algorithm. This means your app can present the drawable to the display at high speed with all the details taken care of.

To enable direct-to-display, your app needs to run in full-screen mode, displaying an opaque `CAMetalLayer` layer and RGB content, and run on a Mac with Apple silicon. There may be other edge case conditions depending on the hardware and system software, but if you set up your window in this way, the drawable is direct-to-display in most situations. All of the RGB formats supported by Metal layers are capable of drawing direct-to-display content. You can enable the Metal HUD or use instruments to verify that your drawable goes direct-to-display.

Choose the screen for displaying your window

To set up a window for displaying your game in macOS, begin by selecting a screen where you want to display your game. The computer running your game may be equipped with more than one monitor. You can use the `NSScreen` class to discover what `screens` are connected to a computer running macOS and what screen the user designated as the main screen. The `main` is optimal for displaying a game. You can get a reference to the main screen like so:

```
NSScreen *screen = [NSScreen mainScreen];
```

The NSScreen object includes detailed information such as the resolution, bit depth, dimensions and location of the screen, its color space, and other details. If you need more information about a screen or want to interact with the screen beyond the functionality offered by the NSScreen AppKit API, see the Quartz Display Services API. Quartz Display Services is a lower-level API that provides direct access to features in the macOS window server for configuring and controlling display hardware.

Pick a style for your window

AppKit in macOS has a concept called *window style* that you store as a set of flags describing the layout of the frame and controls decorating the outside edges of a window. Use the following value for windows displaying Metal game content:

```
NSWindowStyleMask style= NSWindowStyleMaskClosable
                       | NSWindowStyleMaskTitled
                       | NSWindowStyleMaskMiniaturizable
                       | NSWindowStyleMaskResizable;
```

Here are the parts of the style above and the functionality each adds to your window:

`closable`
: A person can close the window.

`titled`
: The window displays a title bar.

`miniaturizable`
: The window contains a miniaturized (yellow dot) control with which a person can collapse the window into the Dock.

`resizable`
: A person can resize the window. Note that allowing resizing doesn't mean you need to resize all your render targets dynamically. In a later section, this article discusses how to adjust your `CAMetalLayer`'s `drawableSize` to match the pixels onscreen whenever someone resizes your window.

Choose the content size of your window and Metal view

When you create a new window, you specify its coordinates in points relative to the coordinates of a screen. *Points* are an abstract measurement quantity that don't correspond to any actual pixel sizes. There are roughly 72 points per inch. For more information on points, see High Resolution Guidelines for OS X.

The system manages most issues related to pixel sizes and display resolution for you, and optimizes a person's experience depending on the device hardware they're using. Your app need only be concerned with setting up its window as described here.

For your initial window size, you can use any point size that you think looks suitable on displays where you expect people to use your app:

```
NSRect contentRect = NSMakeRect(0, 0, 1280, 720);
```

The size you choose when creating your window isn't important, but center your window in the screen:

```
contentRect.origin.x = (screen.frame.size.width - contentRect.size.width) / 2;
contentRect.origin.y = (screen.frame.size.height - contentRect.size.height) / 2;
```

After you create your window, you can get more information about the actual resolution AppKit uses to render your view by using the `convertPointToBacking(_:)` method. For example, if you call `convertPointToBacking(_:)` with the size of your window, it returns the actual pixel dimensions of the window's content onscreen. This can be useful to know when converting locations in your window to actual screen pixel positions, but you don't need to be concerned about these details when creating your window.

Leave all of the AppKit settings for managing a window and the `CALayer` instance it depends on at their defaults so your app looks consistent on the rest of the system.

Create the window

In macOS, the AppKit framework represents windows using the `NSWindow` class, so you can easily add additional functionality to your `NSWindow`. Subclass `NSWindow` like so:

```
@interface GameWindow: NSWindow
@end

@implementation GameWindow
@end
```

Then you can allocate and initialize your window as follows:

```
GameWindow *window = [[GameWindow alloc] initWithContentRect:contentRect
                                                   styleMask:style
                                                     backing:NSBackingStoreBuffered
                                                       defer:NO
                                                      screen:screen];
```

You already set up the contentRect, mask, and screen parameters in the previous sections. The NSWindow.BackingStoreType.buffered parameter tells AppKit that you want a window that has a CALayer backing store. Later, you replace the CALayer with the CAMetalLayer so the window displays your content.

Add the window

For a better user experience, after creating your new window, set the following properties on your window:

- Set the minSize to prevent the user from accidentally resizing the window too small:

  ```
  window.minSize = NSMakeSize(640, 360);
  ```

- If your game retains a reference to a window, set the isReleasedWhenClosed property to NO. This prevents the system from releasing your NSWindow object when someone closes the window: `window.releasedWhenClosed = NO`;.

> **Note**
>
> If your window is closed and not showing, you can display it again by calling the window's setIsVisible(_:) and makeKeyAndOrderFront(_:) methods. For an example, see the "Make the window visible and present in front" section below.

Display your Metal content in your new view

When you want Metal to render a layer's contents, use a CAMetalLayer. You set up a CAMetalLayer and use it to replace the CALayer in the view you set up previously. Begin by creating a new CAMetalLayer:

```
CAMetalLayer *metalLayer = [[CAMetalLayer alloc] init]
```

Then, configure the following settings:

1. Associate your CAMetalLayer with a metal device using the default `metalLayer.device = MTLCreateSystemDefaultDevice();`.
2. Make the layer opaque. An opaque layer can provide direct-to-display contents under the right conditions:

   ```
   metalLayer.opaque = YES;
   ```

3. Choose a resolution for your CAMetalLayer layer. The pixel resolution of your CAMetalLayer determines the size of the drawable produced to fill the layer.

Support resizing dynamically whenever possible and keep in mind the following considerations when deciding on a pixel resolution for your CAMetalLayer:

- The pixel resolution of the CAMetalLayer doesn't have to match the view's backing size (the screen's resolution).
- Render 2D UI that matches the view's backing size and render 3D in a different render target with a size appropriate for the device. Then upscale it to the final drawable using a custom render pass or MetalFX: `metalLayer.drawableSize = [view convertSizeToBacking:view.frame.size];`.
- However, for some games, it might be more convenient to render the drawable at an arbitrary size: `metalLayer.drawableSize = NSMakeSize(3840, 2160);`.
- Depending on your situation, the drawable aspect ratio might not always match the view aspect ratio. In that case, you can ask Core Animation to preserve the aspect ratio for you using `metalLayer.contentsGravity = kCAGravityResizeAspect` and `metalLayer.backgroundColor = CGColorGetConstantColor(kCGColorBlack);`.
- Regardless, the macOS compositor efficiently sends the drawable directly to the display, even if the drawable size doesn't match the monitor size.

After you set up your CAMetalLayer, you can activate it by replacing the layer's CALayer with your new CAMetalLayer:

```
view.layer = metalLayer;
```

At this point, your window is ready to start rendering Metal content.

Make the window visible and present in front

Up until now, everything you've done is offscreen. To present your new window and Metal view onscreen, call the following APIs:

```
[window setIsVisible:true];
[window makeKeyAndOrderFront:nil];
```

If someone happens to close your window and you previously set `window.releasedWhenClosed = NO;` when creating your window, you can use these APIs to present your window onscreen again.

Use the entire window

Lastly, set your window to full-screen mode by calling the `toggleFullScreen(_:)` method:

```
[window toggleFullScreen:nil];
```

You can call this same method again to switch out of full-screen mode. For more information about AppKit and full-screen mode, see Mac App Programming Guide: Implementing the Full-Screen Experience.

Note that when using the `toggleFullScreen(_:)` method, you don't specify what full screen means, and you let the system decide what it means in a way that's most familiar to the user. When you call `toggleFullScreen(_:)`, the system adjusts your window's size to what it considers to be full screen.

Important

Use `toggleFullScreen(_:)` to switch to full screen so your app's full-screen mode works in a way consistent with other apps that use full-screen mode. This keeps control over the window in the user's hands, and AppKit takes responsibility for finding the optimal position and size for your window on the screen that you selected. Avoid customizing any aspects of what `toggleFullScreen(_:)` does.

Add code so your window can handle resizing

To keep the size of your `CAMetalLayer` instance's `drawableSize` in sync with the part of the screen the window is drawing to, set up a `windowDidResize(_:)` method on the window's delegate as described below. Call this method every time your window is resized, including times when your app calls `toggleFullScreen(_:)`.

By adding a `NSWindowDelegate` to your `NSWindow` subclass (`GameWindow`), you can respond to resizing events for the window. These can occur in response to user actions, when properties of the display change, or when your application resizes the window. Use the `NSWindowDelegate` protocol to define your own delegate class capable of responding to `windowDidResize(_:)` events as follows:

```
@interface GameWindowDelegate: NSObject<NSWindowDelegate>
@end

@implementation GameWindowDelegate {
}

-(void)windowDidResize:(NSNotification *)notification {
// Automatically resize the view.
// Resize the Metal layer using the view
// size here. You can use any other size if necessary.
    NSWindow window = notification.object; // 1
    NSView *view = window.contentView; // 2
    CAMetalLayer *metalLayer = (CAMetalLayer *)view.layer; // 3
    metalLayer.drawableSize = [view convertSizeToBacking:view.frame.size]; // 4
}

@end
```

Here's what's going on in the above statements:

- Receive the `NSWindow` object as the object property of the `NSNotification` object.
- Get a reference to the view you created in the "Create the window" section.
- Retrieve a reference to the `CAMetalLayer` you created in the "Display your Metal content in your new view" section.
- Reset the `CAMetalLayer` instance's `drawableSize` property to the actual pixel size of the screen you intend to draw to by calling your `NSView` instance's `convertToBacking(_:)` method, which the "Choose the content size of your window and Metal view" section covers. Note these directions set the drawable to match the size and resolution of the display, but it's not necessary. If the drawable doesn't match the size and resolution of the display, it scales automatically as it presents on the display.

The net result is that whenever you resize the window, the system resets the `CAMetalLayer` instance's `drawableSize` property to the actual pixel resolution of the display your window is drawing to.

Set the delegate for your `GameWindow` to an instance of your class that conforms to the `NSWindowDelegate` protocol by assigning it to the window's `delegate` property.

```
GameWindowDelegate *windowDelegate = [[GameWindowDelegate alloc] init];
if (windowDelegate != NULL) {
    window.delegate = windowDelegate;
}
```

After you set up your window delegate, it begins receiving method calls to your windowDidResize(_:) method whenever the size of your window changes. When your app calls toggleFullScreen(_:), AppKit recomputes the size of your window and calls your delegate's windowDidResize(_:) method.

Add code to prevent exposing your window class to unintentionally consuming key events

Interactions between the AppKit framework and the Game Controller framework can, under some conditions, expose the NSWindow APIs to unintentionally consuming key events that need to go to GCController. To avoid this possibility, add a key down handler that does nothing on your GameWindow class:

```
@interface GameWindow : NSWindow
@end

@implementation GameWindow

- (void)keyDown:(NSEvent *)event
{
}

@end
```

See Also

Presentation

Managing your Metal app window in iPadOS

Set up a window that handles dynamically resizing your Metal content.

Adapting your game interface for smaller screens

Make text legible on all devices the player chooses to run your game on.

- Onscreen presentation

  Show the output from a GPU’s rendering pass to the user in your app.

- HDR content

  Take advantage of high dynamic range to present more vibrant colors in your apps and games.