Article

Managing your Metal app window in iPadOS

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

Overview

A scene represents a single instance of your app's UI. You can choose whether people can create multiple scenes for your app. Typically, Metal apps and games support only one scene because they need priority access to the available resources on a device. On iPadOS 26 and later, people can always resize your app's scenes if they have enabled multitasking.

Apps that don't adopt the scene-based life cycle log a warning at startup on iOS 26 and iPadOS 26 and must be updated. In the next major release, the scene-based life cycle is required when building with the latest SDK.

> **Important**
>
> Because `UIRequiresFullScreen` is deprecated, you can no longer opt out of iPad multitasking and dynamic resizing.

For more information on migrating your iPad app, see TN3192: Migrating your iPad app from the deprecated UIRequiresFullScreen key and TN3187: Migrating to the UIKit scene-based life cycle.

Create the window

Manage windows on iPad by using `UIWindowScene` for UIKit and `Scene` for SwiftUI. To configure a `UIWindow` under a scene you assign a content view controller and embed your Metal view inside the controller.

To configure scene support for your Metal project:

1. Open the Xcode project.
2. Select the project in the Project navigator.
3. Select the app target.
4. Navigate to the General tab.
5. In the Deployment Info section, select “Scene manifest”.
6. Add the `UIApplicationSceneManifest` key if it doesn’t already exist.
7. Configure the dictionary value for your project.

Swift Objective-C

```
<key>UIApplicationSceneManifest</key>
<dict>
    <key>UIApplicationSupportsMultipleScenes</key>
    <false/>
    <key>UISceneConfigurations</key>
    <dict>
        <key>UIWindowSceneSessionRoleApplication</key>
        <array>
            <dict>
                <key>UISceneConfigurationName</key>
                <string>Default Configuration</string>
                <key>MyCustomSceneDelegateClass</key>
                <string>$(PRODUCT_MODULE_NAME).SceneDelegate</string>
                <key>UISceneStoryboardFile</key>
                <string>Main</string>
                <key>UISceneClassName</key>
                <string>UIWindowScene</string>
            </dict>
        </array>
    </dict>
</dict>
```

To provide dynamic scene configurations for complex scenes that require fine-grained control, implement `application(_:configurationForConnecting:options:)` for UIKit and `UIApplicationDelegateAdaptor` for apps that uses the SwiftUI life cycle. This allows for providing dynamic scene configurations:

Swift Objective-C

```swift
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    func application(
        _ application: UIApplication,
        configurationForConnecting connectingSceneSession: UISceneSession,
        options: UIScene.ConnectionOptions
    ) -> UISceneConfiguration {

        // Each scene configuration has a unique, app-specific configuration
        // name that you use to identify the scene. The configuration
        // name corresponds to entries in the `Info.plist` scene manifest.
        var configurationName: String!

        // An activity type distinguishes which scene to create.
        switch options.userActivities.first?.activityType {
        case "com.apple.gallery.openInspector":
            // Create a photo inspector window scene.
            configurationName = "Inspector Configuration"
        default:
            // Create a default gallery window scene.
            configurationName = "Default Configuration"
        }

        return UISceneConfiguration(
            name: configurationName,
            sessionRole: connectingSceneSession.role
        )
    }

    // The system calls this delegate when a person dismisses a scene
    // session, like when closing a window.
    //
    // If the system ends a session while the app wasn't running, it
    // calls application(_:didDiscardSceneSessions:) shortly after
    // calling `application(_:didFinishLaunchingWithOptions:)`.
    //
    // Use this method to release scene-specific resources for the
    // scene, as they won't return.
    func application(_ application: UIApplication,
                     didDiscardSceneSessions sceneSessions: Set<UISceneSession>) {
```

```
    }
}
```

For more information on dynamic configuration, see TN3187: Migrating to the UIKit scene-based life cycle. For more information on adding scene support to your app, see Specifying the scenes your app supports.

Choose the content size and style of your window

After adding scene support, configure the initial size and style of your window's scenes. When your app creates or restores an instance of your user interface, the system calls `scene(_:willConnectTo:options:)`. This delegate method provides a window scene that you use to configure size contraints and style. For Metal apps and games, this is typically a single scene.

Use `UISceneSizeRestrictions` to constrain the minimum size you want, and to handle aspect ratio changes:

Swift | Objective-C

```swift
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(_ scene: UIScene,
               willConnectTo session: UISceneSession,
               options connectionOptions: UIScene.ConnectionOptions) {

        guard let windowScene = scene as? UIWindowScene else { return }
        windowScene.sizeRestrictions?.minimumSize.width = 640.0
    }
}
```

In SwiftUI, use the `windowResizability(_:)` modifier to allow your scene's content to provide sizing information:

```swift
@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
```

```
                .frame(minWidth: 640, minHeight: 360)
        }
        .windowResizability(.contentMinSize)
    }
}
```

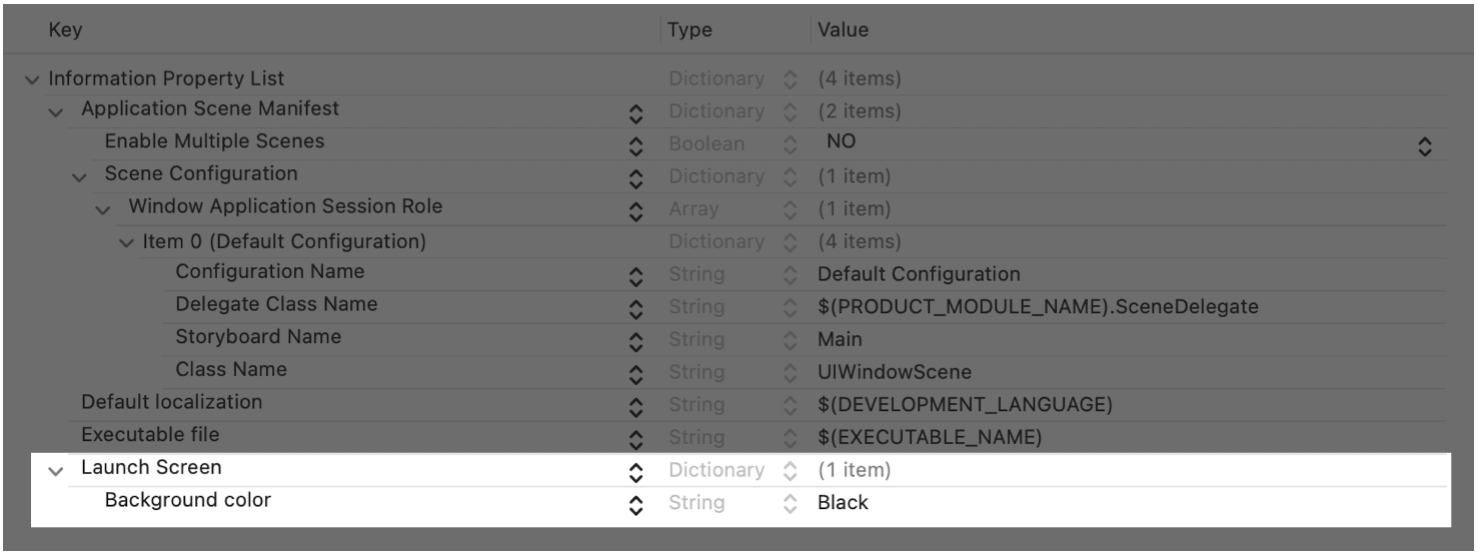
To get the display scale, access `displayScale` from `UITraitCollection` and perform necessary updates in `viewIsAppearing(_:)`. To calculate the pixel values you use for updating the size of `MTLDrawable`, multiply the view's frame and the `contentsScale` of your `CAMetalLayer`:

Swift | Objective-C

```
guard let metalLayer = view.layer as? CAMetalLayer else {
    return
}
// Get the scale that matches the window's display scale.
let screenScale = metalLayer.contentsScale

// Calculate the drawable size in pixels.
let sizeInPixels = CGSize(width: view.frame.width * screenScale,
                          height: view.frame.height * screenScale)
metalLayer.drawableSize = sizeInPixels
```

When a person resizes a window, it's possible that the Metal view only renders to a portion of the window. In this case, add a launch screen with a black background color to letterbox the presentation, then configure the content gravity property for your view so drawable content scales uniformly.



Swift Objective-C

```
override func viewDidLoad() {
    super.viewDidLoad()
    guard let metalLayer = view.layer as? CAMetalLayer else { return }

    metalLayer.isOpaque = true
    metalLayer.backgroundColor = UIColor.black.cgColor

    // For a game, set the content gravity so the aspect ratio of your
    // drawable scene scales uniformly to avoid squishing your content.
    metalLayer.contentsGravity = .resizeAspect
}
```

Handle window resizing

When resizing a window, the system sets `isInteractivelyResizing` and calls the scene delegate `windowScene(_:didUpdateEffectiveGeometry:)` to allow for an app to handle window size changes. When a window resizes, continue rendering at the existing render target size until a person stops resizing the window, at which point you can update the new render target size. Don't query the window size while a person is resizing a window. Instead, track the state in your renderer and then perform the necessary render size update when the person finishes resizing the window. For more information on responding to scene size changes, see TN3187: Migrating to the UIKit scene-based life cycle.

If you use MetalKit, your app receives the `mtkView(_:drawableSizeWillChange:)` delegate view callback:

Swift Objective-C

```
func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {

    /// Your code that responds to drawable size or orientation changes.

    /// Update the projection matrix with the new aspect size.
    let aspect = Float(size.width / size.height)
    projectionMatrix = matrix_perspective_right_hand(fovyRadians: 65.0 * (.pi / 180.
                                                     aspect: aspect,
                                                     nearZ: 0.1,
                                                     farZ: 100.0)
```

```
}

func matrix_perspective_right_hand(fovyRadians: Float,
                                   aspect: Float,
                                   nearZ: Float,
                                   farZ: Float) -> matrix_float4x4 {
    let ys = 1 / tanf(fovyRadians * 0.5)
    let xs = ys / aspect
    let zs = farZ / (nearZ - farZ)

    return matrix_float4x4(columns: (
      simd_float4(xs,  0,        0,  0),    // Column 0
      simd_float4( 0, ys,        0,  0),    // Column 1
      simd_float4( 0,  0,       zs, -1),    // Column 2
      simd_float4( 0,  0, nearZ * zs, 0)    // Column 3
    ))
}
```

Your CAMetalLayer views receive a UIView life cycle call to layoutSubviews() and related property updates — contentScaleFactor, frame, and bounds. Use the related properties to update the MTLDrawable size by getting the window scene's bounds from coordinateSpace and multiplying it by the contentsScale of your CAMetalLayer:

Swift Objective-C

```
func resizeDrawable(scaleFactor: CGFloat) {
    var newSize = self.bounds.size
    newSize.width *= scaleFactor
    newSize.height *= scaleFactor

    if newSize.width <= 0 || newSize.height <= 0 {
        return
    }

    if let metalLayer = layer as? CAMetalLayer {
        if newSize.width == metalLayer.drawableSize.width &&
           newSize.height == metalLayer.drawableSize.height {
            return
        }

        metalLayer.drawableSize = newSize
    }
```

```
    delegate?.drawableResize(newSize)
}
```

Handle moving a window between displays

In iPad, you use `UITraitCollection` to assist with providing a flexible windowing environment that allows your app to render and move windows between multiple displays. To eliminate the need to manually register for trait changes, use Automatic trait tracking to observe the values you need from your specific views. In some cases, you might use `UIScreen` to access a trait that `UITraitCollection` doesn't provide, like `nativeScale`.

When your app's scene geometry changes — like when moving between screens — the `UIWindowSceneDelegate` calls the `windowScene(_:didUpdateEffectiveGeometry:)` method to inspect the window geometry and perform necessary updates:

Swift | Objective-C

```
func windowScene(
    _ windowScene: UIWindowScene,
    didUpdateEffectiveGeometry previousGeometry: UIWindowScene.Geometry) {

    let geometry = windowScene.effectiveGeometry
    let sceneSize = geometry.coordinateSpace.bounds.size

    // Perform necessary updates after the scene geometry changes.
    if sceneSize != previousSceneSize {
        previousSceneSize = sceneSize
    }
}
```

For more information on supporting multiple displays in iPadOS, see Presenting content on a connected display. For more information on managing your Metal app window in macOS, see Managing your game window for Metal in macOS.

Lock interface orientation for device rotation

Some Metal apps and games might need to lock the interface orientation so the screen geometry remains locked when a person rotates the device. To lock the orientation, call `setNeedsUpdateOfPrefersInterfaceOrientationLocked()` in your view controller and check whether the

interface is already locked with the `previousEffectiveGeometry` parameter of `windowScene(_:didUpdateEffectiveGeometry:)`:

Swift Objective-C

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var myGameInstance = MyGame()

    func windowScene(
        _ windowScene: UIWindowScene,
        didUpdateEffectiveGeometry previousGeometry: UIWindowScene.Geometry) {

        let wasLocked = previousGeometry.isInterfaceOrientationLocked
        let isLocked = windowScene.effectiveGeometry.isInterfaceOrientationLocked

        if wasLocked != isLocked {
            myGameInstance.pauseIfNeeded(isInterfaceOrientationLocked: isLocked)
        }
    }
}
```

For more information on locking your app's orientation, see TN3192: Migrating your iPad app from the deprecated UIRequiresFullScreen key.

See Also

Presentation

- Managing your game window for Metal in macOS
  Set up a window and view for optimally displaying 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.