Sample Code

Selecting device objects for graphics rendering

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

Download

macOS 10.13+ | Xcode 12.4+

Overview

macOS supports systems that have multiple GPUs and displays. An example is a MacBook Pro with a low-power integrated GPU, a high-performance discrete GPU, a powerful external GPU, and additional displays. Metal apps must carefully select a GPU that maximizes efficiency and performance for a given display. They should also gracefully respond to any GPU or display changes, such as when the user disconnects an external GPU or moves a window between displays.

Getting started

Not all Mac computers have both an integrated GPU and a discrete GPU. To check the GPUs in your Mac, choose Apple menu > About this Mac, press the System Report button, and select Graphics/Displays on the left. The GPUs are listed under Video Card. MacBook Pro computers with two GPUs have an Automatic Graphics Switching option, turned on by default, that allows the system to automatically switch between the two GPUs. To toggle the Automatic Graphics Switching state, choose Apple menu > System Preferences and click Energy Saver. The Automatic Graphics Switching checkbox is shown at the top.

Optionally, you may connect an external GPU to your Mac via Thunderbolt 3, and you may also connect an external display to your external GPU. For this system setup, your Mac must be running macOS 10.13.4 or later. Connecting an external GPU allows the sample to run the code described in Handle External GPU Notifications.

The sample provides these interactive UI controls:

- **Device Selection Mode.** Allow the sample to automatically select the best device for the display, or indicate that you want to manually select a device.
- **Manual Device Selection.** Manually select a device from a list of available devices.

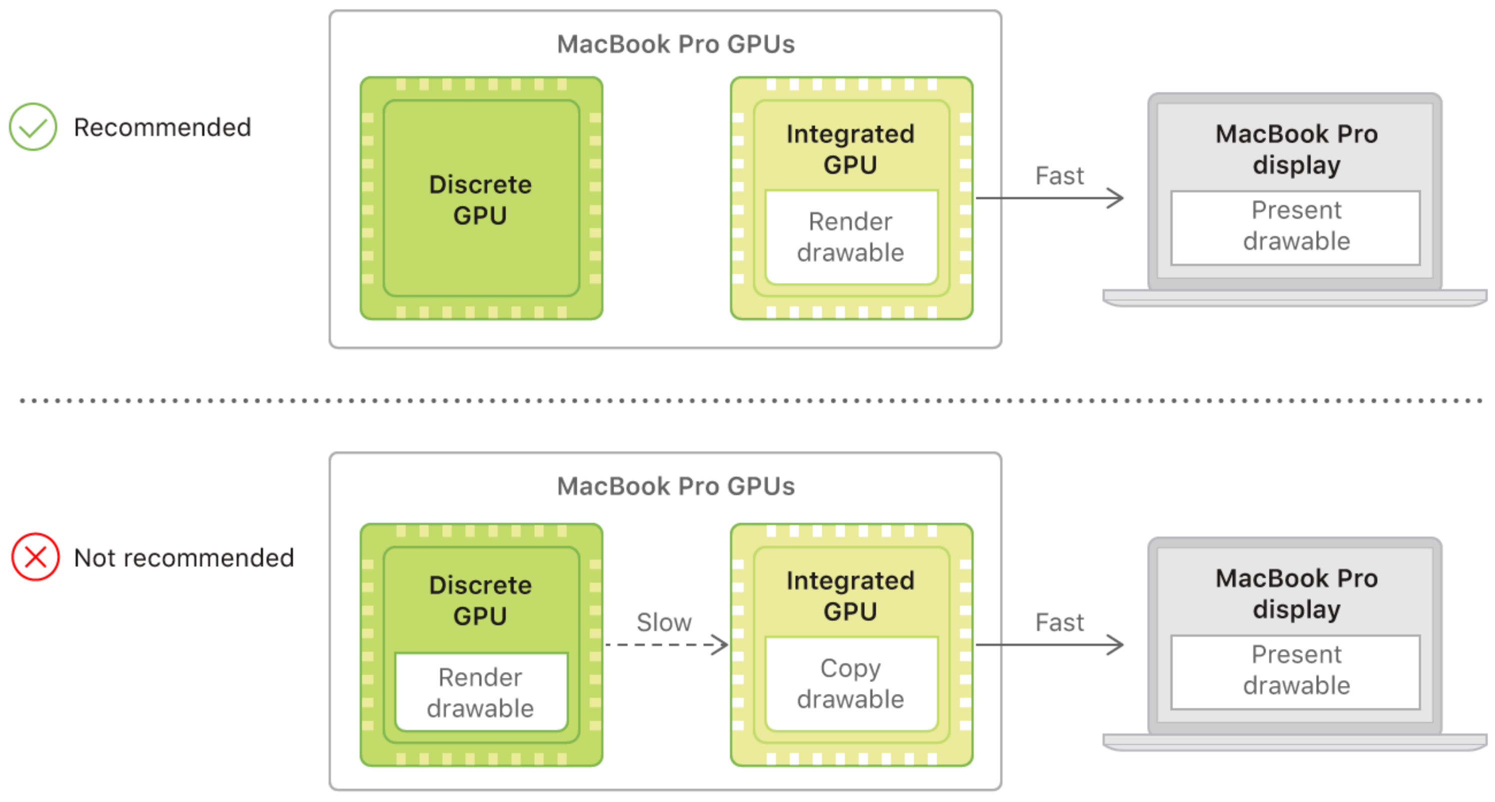
Furthermore, the Device Driving Display label indicates which device is currently driving the display.

Drawables, displays, and GPUs

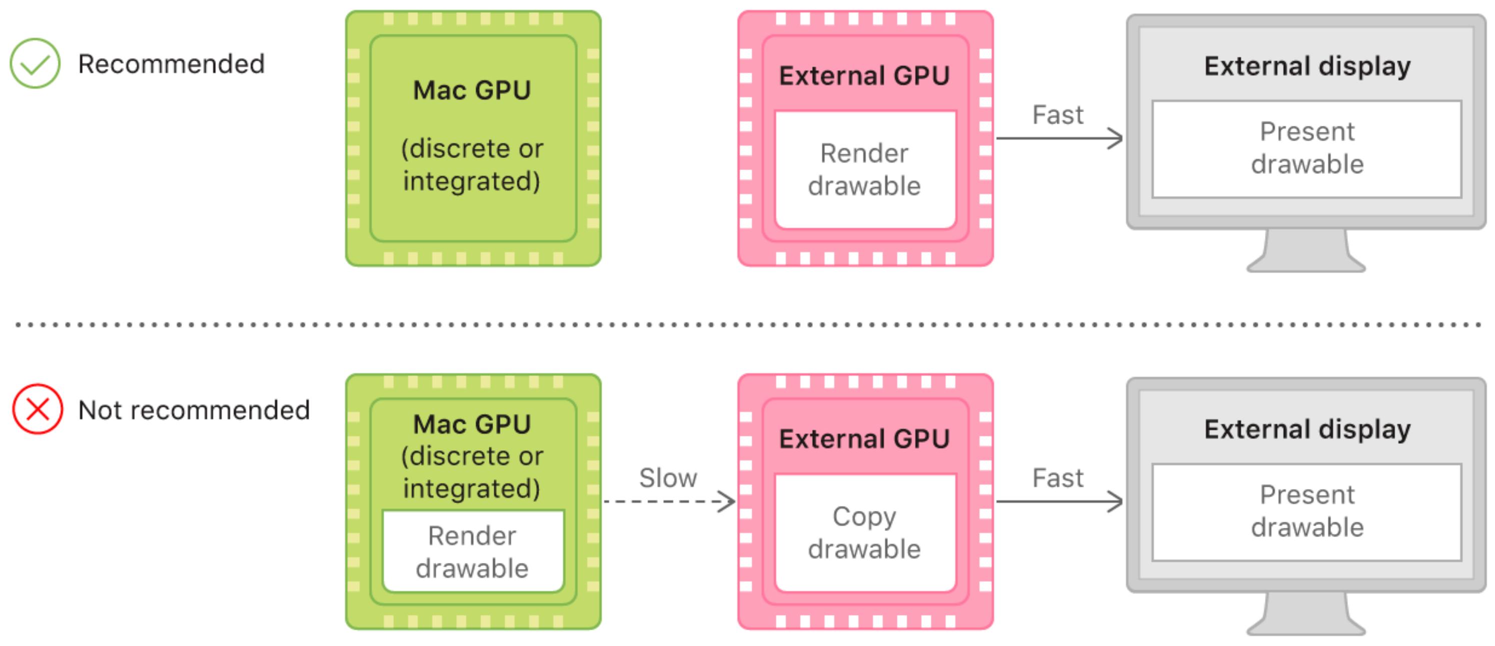
Each view in your app is shown on a single display, and each display is driven by a single GPU. To show graphics content in your view, the view's display presents a rendered drawable from the display's driving GPU.

If your app renders with a GPU that isn't driving your view's display, the system must copy the drawable from the rendering GPU to the displaying GPU before presenting it. This transfer can be expensive because the bandwidth between GPUs is limited by the bus that connects them. This expense is more severe with external GPUs because their Thunderbolt 3 bus has much less bandwidth than an internal PCI Express bus.

The fastest path to present a drawable is to render that drawable with the GPU that drives your view's display. An example is a MacBook Pro with a discrete GPU and an integrated GPU, where the integrated GPU can drive the built-in display under certain conditions, such as thermal state, battery life, or an app's needs.



Another example is a Mac connected to an external GPU, where the external GPU drives an external display.



Transition smoothly between devices

The sample's view controller manages all Metal devices, with each device representing a different GPU. When the sample runs the `viewDidLoad` method, the view controller initializes a new `AAPLRenderer` for each device available to the system. The sample uses only one device at a time, but it initializes a renderer for each device in order to preload and mirror the app's Metal resources across all devices. Therefore, when the app switches between GPUs at runtime, the sample transitions smoothly between devices because equivalent resources are already available and loaded on each device. This preloading and mirroring strategy avoids significant delays that would otherwise exist if the sample needed to load resources at the time of the switch.

> **Note**
>
> Preloading and mirroring resources allows you to transition smoothly between devices, but it also increases your app's total memory usage. You must carefully determine which resources should be preloaded and mirrored, and which resources should instead be loaded only when your app switches between devices.

Set the optimal device for the view's display

After the view appears, the sample gets the `CGDirectDisplayID` value of the display in which the view appears. The sample uses this identifier to get the Metal device that drives the display.

```
// Get the display ID of the display in which the view appears
CGDirectDisplayID viewDisplayID = (CGDirectDisplayID) [_view.window.screen.deviceDes

// Get the Metal device that drives the display
id<MTLDevice> newPreferredDevice = CGDirectDisplayCopyCurrentMetalDevice(viewDispla
```

The sample sets this device for the view controller's `MTKView`, and chooses the `AAPLRenderer` associated with that same device to perform the app's rendering. This setup ensures that the system renders with the device that drives the display, and it avoids copying any drawables from one GPU to another.

Handle display change notifications

To keep up to date with the optimal device for the view's display, the sample registers for two system notifications:

- `NSApplicationDidChangeScreenParametersNotification`. macOS posts this notification when a display configuration changes. An example is when the user connects or disconnects an external display from the system. Another example is when the GPU driving the display changes, such as when Automatic Graphics Switching is enabled and the system switches between discrete and integrated GPUs to drive the display.
- `NSWindowDidChangeScreenNotification`. The system posts this notification when any window, including the window containing the app's view, moves to a different display.

```
// Register for the NSApplicationDidChangeScreenParametersNotification, which trigge
// when the system's display configuration changes
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(handleScreenChanges:)
                                             name:NSApplicationDidChangeScreenParame
                                           object:nil];

// Register for the NSWindowDidChangeScreenNotification, which triggers when the wir
// changes screens
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(handleScreenChanges:)
                                             name:NSWindowDidChangeScreenNotificatio
                                           object:nil];
```

In both cases, the system calls the sample's `handleScreenChanges:` method to handle the notification. The sample then chooses the optimal device for the view's display by selecting the `AAPLRenderer` object that corresponds to the device driving the display.

Set a GPU eject policy

By default, macOS fully relaunches an app when an external GPU being used by the app is removed from the system. An app typically handles a relaunch by:

1. Saving as much state as possible when the system calls the app's `application:willEncodeRestorableState:` method, before macOS quits the app.

2. Restoring any saved state when the system calls the app's `application:didDecodeRestorableState:` method, after macOS relaunches the app.

The sample avoids this app relaunch routine by instead opting in to handling the external GPU removal itself, without macOS needing to quit and relaunch the app. The sample's `Info.plist` file has a `GPUEjectPolicy` key with a `wait` value, which indicates that the app explicitly handles the removal of an external GPU by responding to the corresponding notification posted by Metal.

Register for external GPU notifications

The sample calls the `MTLCopyAllDevicesWithObserver` function to obtain all Metal devices available to the system. This method allows the sample to supply an `MTLDeviceNotificationHandler` block that's executed when an external GPU is added or removed from the system. This handler provides two arguments:

- `device`. The device that was added or removed.

- `notifyName`. A value that describes the event that triggered the notification.

```
MTLDeviceNotificationHandler notificationHandler;

AAPLViewController * __weak controller = self;
notificationHandler = ^(id<MTLDevice> device, MTLDeviceNotificationName name)
{
    [controller markHotPlugNotificationForDevice:device name:name];
};

// Query all supported metal devices with an observer, so the app can receive notifi
// when external GPUs are added to or removed from the system
id<NSObject> metalDeviceObserver = nil;
NSArray<id<MTLDevice>> * availableDevices =
    MTLCopyAllDevicesWithObserver(&metalDeviceObserver,
                                  notificationHandler);
```

Respond to external GPU notifications

The notification handler can execute on any thread. However, all UI updates must occur on the main thread and the app's state changes must be explicitly made thread-safe. To comply with these thread requirements, the view controller protects access to the `_hotPlugEvent` and `_hotPlugDevice` instance variables with a `@synchronized` directive. (The `@synchronized` directive is a convenient way to create mutex locks in Objective-C code.)

The sample sets these instance variables in the `markHotPlugNotificationForDevice:name:` method when a notification occurs.

```
- (void)markHotPlugNotificationForDevice:(nonnull id<MTLDevice>)device
                                    name:(nonnull MTLDeviceNotificationName)name
{
    @synchronized(self)
    {
        if ([name isEqualToString:MTLDeviceWasAddedNotification])
        {
            _hotPlugEvent = AAPLHotPlugEventDeviceAdded;
        }
        else if ([name isEqualToString:MTLDeviceRemovalRequestedNotification])
        {
            _hotPlugEvent = AAPLHotPlugEventDeviceEjected;
        }
        else if ([name isEqualToString:MTLDeviceWasRemovedNotification])
        {
            _hotPlugEvent = AAPLHotPlugEventDevicePulled;
        }

        _hotPlugDevice = device;
    }
}
```

The sample reads these instance variables on the main thread and handles the notification in the `handlePossibleHotPlugEvent` method.

```
- (void)handlePossibleHotPlugEvent
{
    AAPLHotPlugEvent hotPlugEvent;
    id<MTLDevice> hotPlugDevice;

    @synchronized(self)
    {
        hotPlugEvent = _hotPlugEvent;
        hotPlugDevice = _hotPlugDevice;
        _hotPlugDevice = nil;
    }

    if(hotPlugDevice)
    {
        switch (hotPlugEvent)
        {
            case AAPLHotPlugEventDeviceAdded:
                [self handleMTLDeviceAddedNotification:hotPlugDevice];
                break;
            case AAPLHotPlugEventDeviceEjected:
            case AAPLHotPlugEventDevicePulled:
                [self handleMTLDeviceRemovalNotification:hotPlugDevice];
                break;
        }
    }
}
```

When a device that represents an external GPU is added to the system, the `handlePossible HotPlugEvent` method adds the device to the `_supportedDevices` array and initializes a new `AAPLRenderer` for the device. When such a device is removed from the system, the same method removes the device from the `_supportedDevices` array and destroys its associated

AAPLRenderer. If the removed device was being used for rendering, the sample switches to another device and renderer.

Update per-frame state and data

MetalKit calls the draw(in:) method for the sample to render each frame. Within this method, the sample calls the handlePossibleHotPlugEvent method to handle device additions or removals on the main thread. Such actions include updating UI related to these device events and completing any additional state changes that must be executed atomically on a single thread.

The sample then calls the drawFrameNumber:toView: to begin rendering a new frame for the current renderer. To ensure continuous rendering that enables seamless switching between different renderers, the sample stores any nonrendering state separate from the renderers themselves. Then, for each frame, the sample passes any necessary nonrendering state to a specific AAPLRenderer instance. In this case, the sample passes the current frame number, _frameNumber, to the renderer so it can calculate the position and rotation of the sample's 3D model.

Deregister from notifications

After the view disappears, the sample explicitly deregisters itself from any previous display or device notifications. Otherwise, the system's notification center and Metal can't release the sample's view controller.

```
- (void)viewDidDisappear
{
    [[NSNotificationCenter defaultCenter] removeObserver:self
                                                     name:NSApplicationDidChangeScree
                                                   object:nil];

    [[NSNotificationCenter defaultCenter] removeObserver:self
                                                     name:NSWindowDidChangeScreenNoti
                                                   object:nil];

    MTLRemoveDeviceObserver(_metalDeviceObserver);
}
```

Note

The sample can't defer the deregistration process to the view controller's `dealloc` method. When the `dealloc` method is executed, the system's notification center and Metal still have references to the view controller that prevent it from being destroyed.

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.

{} Customizing render pass setup

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

{} Creating a custom Metal view

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

{} 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.