iOS Specific Optimizations

This page details optimizations which are unique to iOS deployment. For more information on optimizing for mobile devices, see the Practical Guide to Optimization for Mobiles.

Most of the functions in the **UnityEngine** namespace are implemented in C/C++. Calling a C/C++ function from a Mono script involves a performance overhead. You can use iOS Script Call optimization (menu: **Edit->Project Settings->Player**) to save about 1 to 4 milliseconds per frame. The options for this setting are:-

- **Slow and Safe** - the default Mono internal call handling with exception support.
- **Fast and Exceptions Unsupported** - a faster implementation of Mono internal call handling. However, this doesn't support exceptions and so should be used with caution. An app that doesn't explicitly handle exceptions (and doesn't need to deal with them gracefully) is an ideal candidate for this option.

Setting the Desired Framerate

Unity iOS allows you to change the frequency with which your application will try to execute its rendering loop, which is set to 30 frames per second by default. You can lower this number to save battery power but of course this saving will come at the expense of frame updates. Conversely, you can increase the framerate to give the rendering priority over other activities such as touch input and accelerometer processing. You will need to experiment with your choice of framerate to determine how it affects gameplay in your case.

If your application involves heavy computation or rendering and can maintain only 15 frames per second, say, then setting the desired frame rate higher than fifteen wouldn't give any extra performance. The application has to be optimized sufficiently to allow for a higher framerate.

To set the desired framerate, open the XCode project generated by Unity and open the `AppController.mm` file. The line

```
#define kFPS 30
```

...determines the the current framerate, so you can just change to set the desired value. For example, if you change the define to:-

```
#define kFPS 60
```

...then the application will attempt to render at 60 FPS instead of 30 FPS.

The Rendering Loop

When iOS version 3.1 or later is in use, Unity will use the **CADisplayLink** class to schedule the rendering loop. Versions before 3.1 need to use one of several fallback methods to handle the loop. However, the fallback methods can be activated even for iOS 3.1 and later by changing the line

```
#define USE_DISPLAY_LINK_IF_AVAILABLE 1
```

...and changing it to

```
#define USE_DISPLAY_LINK_IF_AVAILABLE 0
```

Fallback Loop Types

Apple recommends the system timer for scheduling the rendering operation on iOS versions before 3.1. This approach is good for applications where performance is not critical and favours battery life and correct processing of events over rendering performance. However, better rendering performance is often more important to games, so Unity provides several scheduling methods to tweak the performance of the rendering loop:-

- **System Timer:** this is the standard approach suggested by Apple. It uses the **NSTimer** class to schedule rendering and has the worst rendering performance but guarantees to process all input events.
- **Thread:** a separate thread is used to schedule rendering. This offers better rendering performance than the NSTimer approach, but sometimes could miss touch or accelerometer events. This method of scheduling is also the easiest to set up and is the default method used by Unity for iOS versions before 3.1.
- **Event Pump:** this uses a **CFRunLoop** object to dispatch events. It gives better rendering performance than the NSTimer approach and also allows you to set the amount of time the OS should spend processing touch and accelerometer events. This option must be used with care since touch and accelerometer events will be lost if there is not enough processor time available to handle them.

The different fallback loop types can be selected by changing defines in the AppController.mm file. The significant lines are the following:-

```
#define FALLBACK_LOOP_TYPE NSTIMER_BASED_LOOP
```

```
#define FALLBACK_LOOP_TYPE THREAD_BASED_LOOP
#define FALLBACK_LOOP_TYPE EVENT_PUMP_BASED_LOOP
```

The file should have all but one of these lines commented out. The uncommented line selects the rendering loop method that will be used by the application.

If you want to prioritize rendering over input processing with the NSTimer approach you should locate and change the line

```
#define kThrottleFPS 2.0
```

...in AppController.mm. Increasing this number will give higher priority to rendering. The result of changing this value varies among applications, so it is best to try it for yourself and see what happens in your specific case.

If you use the Event Pump rendering loop then you need to tweak the `kMillisecondsPerFrameToProcessEvents` constant precisely to achieve the desired responsiveness. The `kMillisecondsPerFrameToProcessEvents` constant allows you to specify exactly how much time (in milliseconds) you will allow the OS to process events. If you allocate insufficient time for this task then touch or accelerometer events might be lost, and while the application will be fast, it will also be less responsive.

To specify the amount of time (in milliseconds) that the OS will spend processing events, locate and change the line

```
#define kMillisecondsPerFrameToProcessEvents 7.0
```

...in AppController.mm.

Tuning Accelerometer Processing Frequency

If accelerometer input is processed too frequently then the overall performance of your game may suffer as a result. By default, a Unity iOS application will sample the accelerometer 60 times per second. You may see some performance benefit by reducing the accelerometer sampling frequency and it can even be set to zero for games that don't use accelerometer input. You can change the accelerometer frequency from the **Other Settings** panel in the iOS Player Settings.

Page last updated: 2012-07-30