Specialized Debugging Workflows

This chapter focuses on often encountered but more specialized debugging scenarios and highlights the Xcode debugging tools used to work with them.

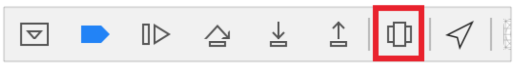
Debugging View Hierarchies

Some bugs are immediately visible to the eye because they are problems with the views your app uses in the UI. For instance, misplaced graphics on the screen; the wrong picture associated with an item; pictures, labels, or text that are incorrectly clipped or placed—these are all examples of issues with an app's view hierarchy.

The Xcode debugger provides the ability to inspect and understand the view hierarchy.

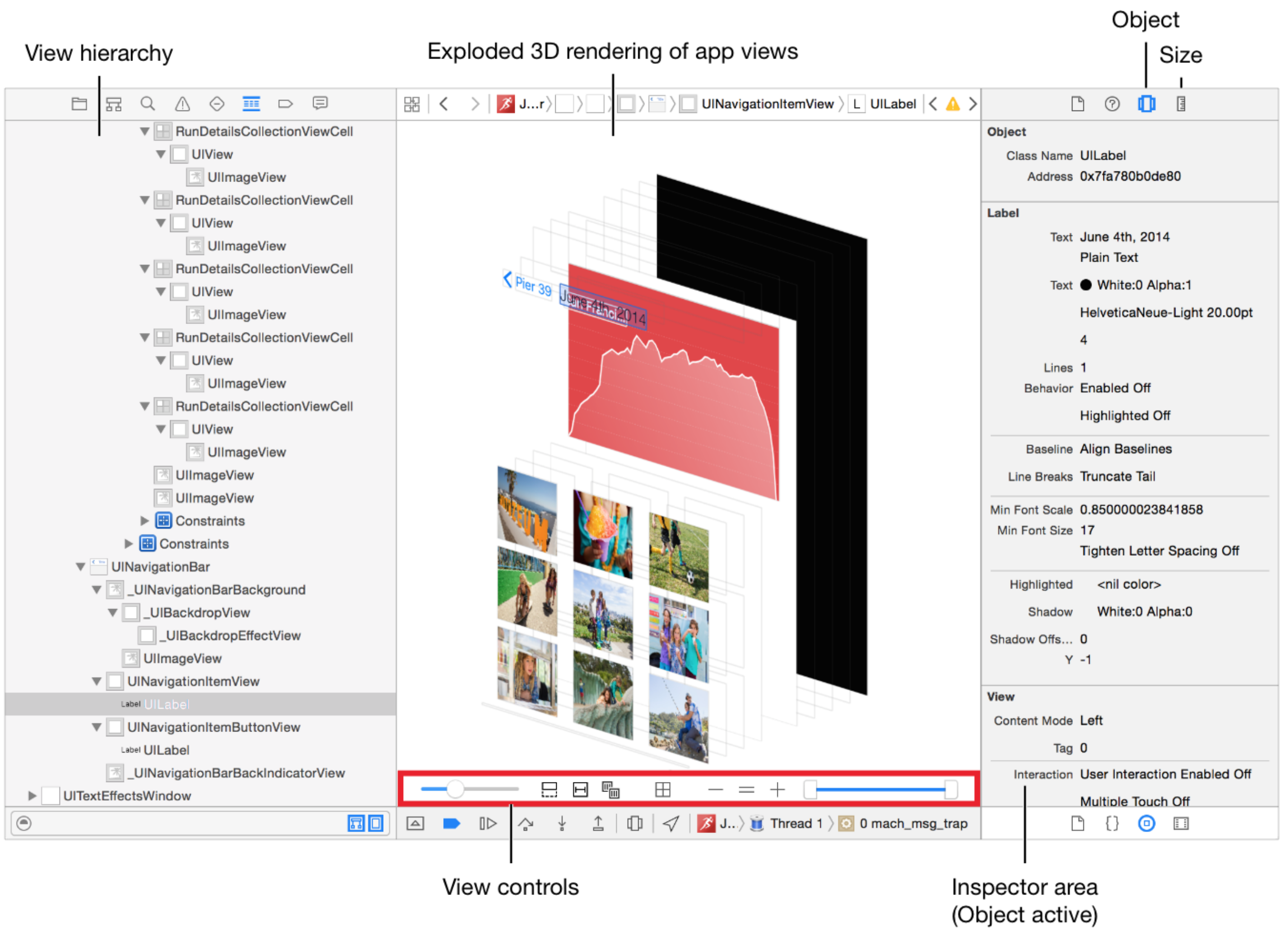
Basic Operation

With your app running in the debugger, click the Debug View Hierarchy button in the debug bar.



Note: You can also invoke the view debugger by choosing View UI Hierarchy from the process view options menu in the debug navigator, or by choosing Debug > View Debugging > Capture View Hierarchy.

When you debug view hierarchies, Xcode pauses your app in its current state and reconfigures the main window. The tools to debug view hierarchies are integrated throughout the Xcode main window. The main source editor window shows an interactive 3D model of the currently selected window in the main editor. The debug navigator process view changes to display your app's view hierarchy as a hierarchical list. In the Xcode utility pane, the inspector selector bar now includes buttons for an object inspector and size inspector, all pertaining to your app's views.



Object

Class Name	UILabel
Address	0x7fa780b0de80

Label

Text	June 4th, 2014
	Plain Text
Text	● White:0 Alpha:1
	HelveticaNeue-Light 20.00pt
	4
Lines	1
Behavior	Enabled Off
	Highlighted Off
Baseline	Align Baselines
Line Breaks	Truncate Tail
Min Font Scale	0.850000023841858
Min Font Size	17
	Tighten Letter Spacing Off
Highlighted	<nil color>
Shadow	White:0 Alpha:0
Shadow Offs...	0
Y	-1

View

Content Mode	Left
Tag	0
Interaction	User Interaction Enabled Off
	Multiple Touch Off

The view of the currently selected window in the main editor is an active representation of all the views in the window. To rotate the window being displayed in the main editor, drag it. The views are exploded into three dimensions so that you can visualize the layer hierarchy and see the relationships between views in that hierarchy. A two-finger drag allows you to shift the position of the view display in the main editor; this becomes very useful when you magnify the views to examine details of the view hierarchy. Clicking an object in the exploded view hierarchy populates the object and size inspectors in the utility pane, and indicates the selection in the hierarchical list displayed in the debug navigator.

At the bottom of the main editor there are additional controls for manipulating the exploded view display. The left slider allows you to adjust the spacing between the views so you can see individual view objects more easily. The right slider allows you to filter the views from back-to-front or front-to-back so you can home in on a particular view in a complex hierarchy.

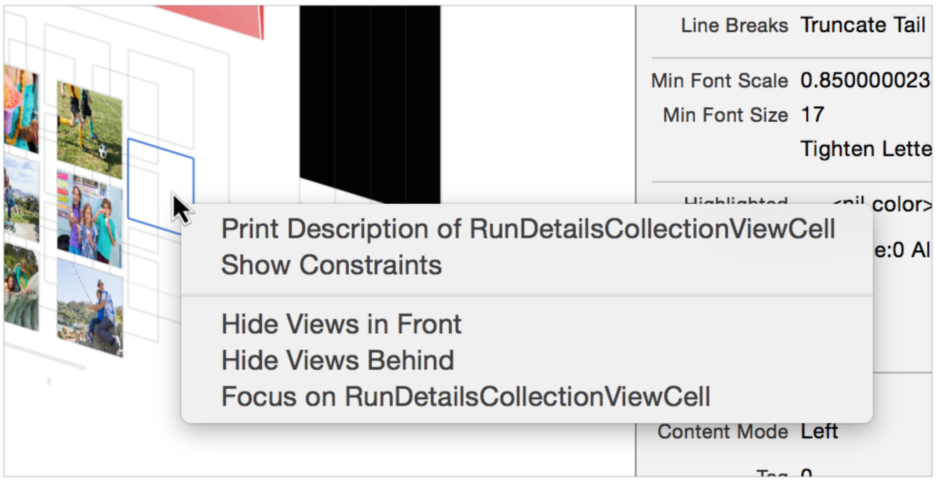
A set of control buttons is located between the two sliders. From left to right:

- Use the Clipping button to reveal clipped content of all views that are displayed in the main editor.
- Click the Show Constraints button to display Auto Layout constraints of the currently selected view layer in the main editor. When you turn on Show Constraints and click in a view object, other view objects are dimmed to allow you to see the constraints associated with this object more clearly.

Note: The size inspector enables you to examine dimensions and constraints values.

- Use the Reset button to return the main editor display to the default 2D orientation.
- Click the View Mode button to select whether to display view Contents, Wireframes, or Wireframes and Contents in the main editor.
- The zoom controls allow you to increase and decrease the displayed magnification, or return the display to standard size. You can also zoom in and out using pinch gestures on a trackpad. If you prefer using a mouse, holding down the Option key and using the scroll wheel is another way to zoom.

Control-clicking on a view in the main editor produces a menu with several actions you can apply to the selected view object.

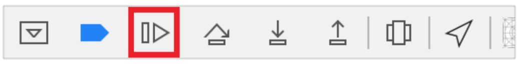

Line Breaks Truncate Tail
Min Font Scale 0.850000023
Min Font Size 17
Tighten Lette
Print Description of RunDetailsCollectionViewCell
Show Constraints
Hide Views in Front
Hide Views Behind
Focus on RunDetailsCollectionViewCell
Content Mode Left

- **Print Description of {viewObject}**. Performs a `po` command on the selected view object and outputs to the Xcode console.
- **Show Constraints**. Enables the display of constraints for the selected view object. This works together with the Size inspector to show constraints details of the view object.
- **Hide Views in Front**, **Hide Views Behind**. These two commands simplify the display to allow easier access to views at the same level in the hierarchy for inspection
- **Focus on {viewObject}**. Simplifies the view to just the sub-view hierarchy containing the selected view object.

  This is the same as double-clicking a view: It removes the superview tree and focuses on that view and its subview tree. To return to the full view hierarchy display, double-click outside the displayed view or click "`x`" in the debug navigator's "Focused" indicator).

Tip: If you select a view or constraint in 3D from the main window, then pressing Shift-Command-D highlights it in the debug navigator. This is a keyboard short-cut for the menu command Navigate > Reveal in Debug Navigator.

Click the Continue button in the debug bar to exit the view debugger and continue running your app in Xcode.

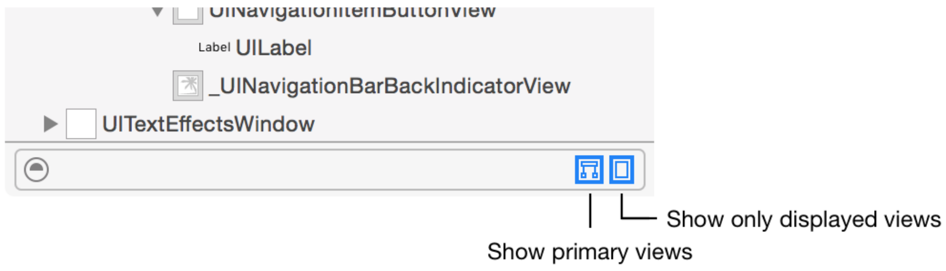


The View Hierarchy Display

When the view debugger is active, the debug navigator presents all the view objects in the active window as a hierarchical list. This display lets you clearly identify parent, child, and sibling views in the view hierarchy, and can be an easier way to find and select a view object than manipulation in the main editor. Clicking a view in the editor pane sets it to be a secondary selection, indicated with a light color in the view hierarchy. The primary selection remains highlighted with a deeper color. These different selection indications help you explore the relationship between views in a complex view hierarchy by selecting a primary in the view hierarchy then selecting other views in the editor pane. The secondary selection becomes the active selection in the object and size inspectors.

Due to the complexity of views, you can also use the main editor jump bar to navigate the view hierarchy.

The filter bar under the view hierarchy has two filter buttons enabled by default.


UINavigationItemButtonView
Label UILabel
_UINavigationBarBackIndicatorView
UITextEffectsWindow
Show only displayed views
Show primary views

- **Show primary views.** This filter hides view objects that are secondary elements of system view implementations, not under app control; it simplifies system views and hides the internal complexity.

  For example, a `UIButton` instance is constructed from a background view and a `UILabel` instance. When this filter is enabled, you will see only show one view, a `UIButton` instance. If the filter is disabled, you also see the additional subviews when rotated in 3D. Using this option removes clutter that isn't helpful in normal use, but lets you display the full complexity of details when needed.

- **Show only displayed views.** This filter hides views that have been hidden by the app using the view attributes. It is particularly useful to disable "Show only displayed views" if you don't see a view you were expecting to see, which can happen when the visible or "is hidden" attributes are set incorrectly. Disabling this filter allows you to see the view object, see its address in memory, and find it in your source code.

Use the text filter in the filter bar to find views by type or name.

Click a view object in the view hierarchy to select it in the editor pane and list its attributes in the inspectors. The view hierarchy in the debug navigator pane shows the relationship of each view to its parent, chlld, and sibling view objects.

The Object and Size Inspectors

When you select objects in the main editor or in the debug navigator view hierarchy, the object attributes are loaded into the Object inspector and the Size inspector in the Xcode utility pane. The Object inspector presents the class and actual address in memory of the view object, as well as any other attributes associated with it such as whether it contains a label, text, and the specific attributes of those object entities. The Size inspector presents the sizing information associated with the view object as well as its Auto Layout constraints.

The Assistant Editor

Set the assistant editor set to Automatic mode to view the source of a selected view object. When you click a custom view object, the assistant editor displays your custom implementation file. When you select a view object supplied by an operating system framework, the assistant editor displays the interface file.

Using the View Debugger

Using the exploded display of view objects in the main editor, the view hierarchy listing in the debug navigator, the information presented in the Object and Size inspectors, and the assistant editor's ability to show associated implementation files for objects in the view hierarchy, you can obtain a great deal of information about how your app's view objects interact. This information enables you to take what you know about the relationships of view objects in your app and quickly correct display problems based on incorrect view object interactions and Auto Layout constraint issues.

For a demonstration of using the Xcode view debugging tools effectively, see these video presentations:

- WWDC 2014 Debugging in Xcode 6: Debugging User Interfaces in Xcode
- WWDC 2016 Debugging in Xcode 8: Visual Debugging with Xcode

Note on compatibility: View debugging is available when developing for iOS, tvOS, and macOS. It is supported on iOS 8 and later, all tvOS versions, and 64-bit targets on macOS.

Debugging App Extensions

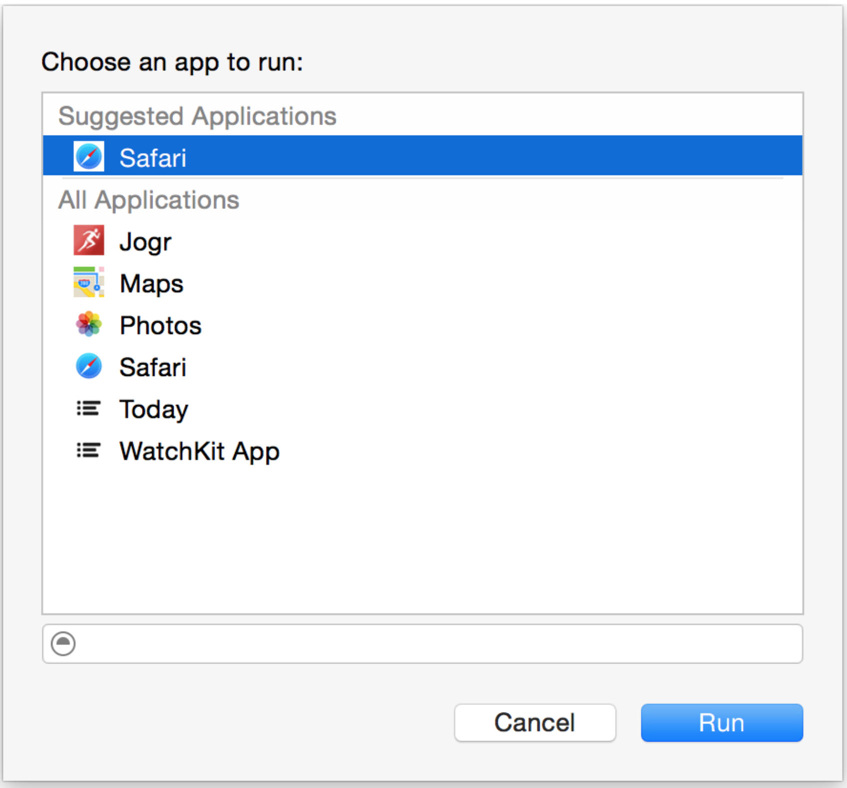
Using Xcode to debug an app extension is a lot like using Xcode to debug any other process, but with one important difference: In your extension scheme's Run phase, you specify a *host app* as the executable. Upon accessing the extension through that specified host's UI, the Xcode debugger attaches itself to the extension.

Note: You must code sign your containing app and its contained app extensions.

All the targets in your Xcode project must be code signed in the same way. For example, during testing you can employ ad hoc code signing or use your developer certificate, but you must use the same approach for all the targets in your project. For submission to the App Store, use your distribution certificate for all the targets.

Build and Run for Debugging with Ask On Launch

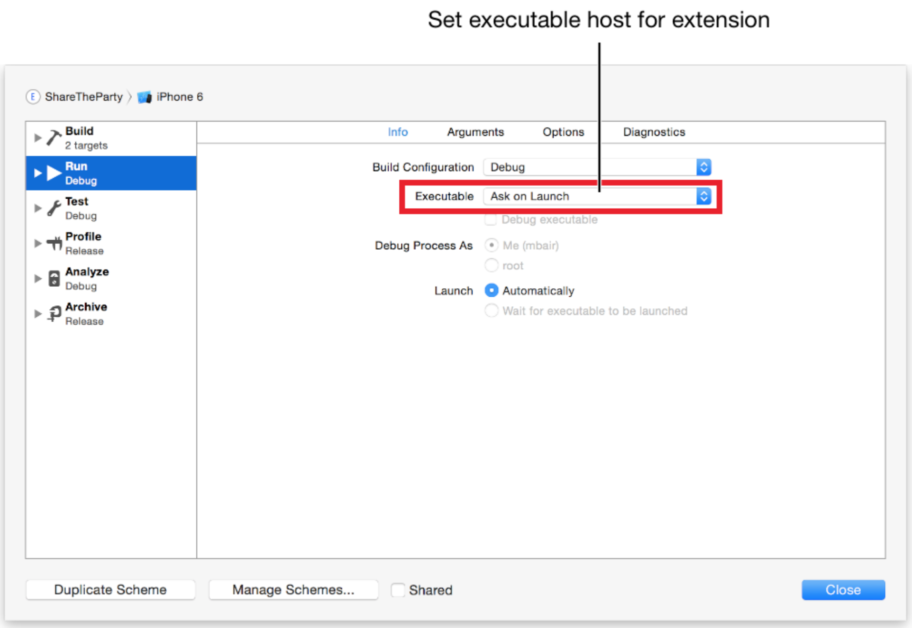
By default, the scheme in an Xcode app extension template uses the Ask On Launch option for the executable. With this option, each time you build and run your project you're prompted to pick a host app.


Choose an app to run:

Suggested Applications
- Safari

All Applications
- Jogr
- Maps
- Photos
- Safari
- Today
- WatchKit App

If you want to instead specify a host to use every time, open the scheme editor and click the Info tab to view the app extension scheme's Run phase.



Build Configuration: Debug
Executable: Ask on Launch
Debug executable
Debug Process As: [✓ Me (mbair)] root
Launch: [✓ Automatically] Wait for executable to be launched

To attach the Xcode debugger to your app extension:

1. Enable the app extension’s scheme by choosing Product > Scheme > MyExtensionName or by clicking the scheme pop-up menu in the Xcode toolbar and choosing MyExtensionName.
2. Click the Build and Run button to tell Xcode to launch your specified host app.

   The Debug navigator indicates it is waiting for you to invoke the app extension.
3. Invoke the app extension by way of the host app’s UI.

   The Xcode debugger attaches to the extension’s process, sets active breakpoints, and lets the extension execute. At this point, you can use the same Xcode debugging features that you use to debug other processes.

Note: Before you build and run your app extension project, make sure the extension’s scheme is selected.

If you instead build and run using the *containing app* scheme, Xcode does not attach to your app extension unless you invoke it from the containing app, which is an unusual scenario and might not be what you want.

If you access your app extension from a host app different from the one specified in the scheme, the Xcode debugger does not attach to the extension.

Enabling App Extensions

For a custom keyboard in iOS, use Settings to enable the app extension (Settings > General > Keyboard > Keyboards).

In OS X, Xcode handles the step of enabling an app extension so you can access it from a host app for testing and debugging.

For an OS X Today widget, use the Widget Simulator to test and debug it.

Xcode registers a built app extension for the duration of the debugging session on OS X. This means that if you want to install the development version of your extension on OS X, you need to use the Finder to copy it from the build location to a location such as the Applications folder.

Note: In the Xcode debug console logs, an app extension’s binary might be associated with the value of the `CFBundleIdentifier` property instead of the value of the `CFBundleDisplayName` property.

Performance Monitoring

Because app extensions must be responsive and efficient, it’s a good idea to watch the debug gauges in the debug navigator while you’re running your extension. The debug gauges show how your extension uses the CPU, memory, and other system resources while it runs. If you see evidence of performance problems, such as an unusual spike in CPU usage, you can you can often fix it on the spot.

Debugging Watch Apps and App Extensions

The methodology for debugging watchOS app extensions and apps follows the same patterns as debugging for iOS apps, libraries, and app extensions. However, you cannot debug a watchOS 1 app extension in a project that also has a watchOS 2 app built in the same iOS app. In this situation, Xcode prefers the watchOS 2 app when both are present. This means that you need to remove the watchOS 2 app from the iOS app bundle to enable debugging for the watchOS 1 app extension.

After debugging your watchOS 2 app, use the following procedure to enable debugging for the watchOS 1 app extension.

1. Select your project in the project navigator to open the project editor.
2. Select the iOS app target.
3. Click the Build Phases tab.
4. In the Target Dependencies section, select the watchOS 2 app.
5. Click Delete (–) to remove the watchOS 2 app as a build dependency of the iOS app.
6. In the Embed Watch Content section, select the watchOS 2 app.
7. Click Delete (–) to remove the watchOS 2 app.
8. Hold down the Option key.
9. Choose Product > Clean Build Folder

After completing these steps, click Run to debug the watchOS 1 app extension. When you’re done debugging, remember to reverse this procedure in order to restore the watchOS 2 app target to the iOS app’s build when building for release.

Thread and Queue Debugging

The debug navigator’s display of threads and queues helps you fix problems related to concurrency and control flow.

Background Information

To take advantage of multicore processor’s potential power, threads are used to partition app operations and distribute them onto different cores for more efficient, smoother, faster execution. However, writing threaded code is challenging; threads are a low-level tool that must be managed manually.

Both macOS and iOS have adopted a more asynchronous and higher-level approach to the execution of concurrent tasks called *Grand Central Dispatch (GCD)*. Rather than creating and managing threads directly, apps need only define tasks, enqueue them, and then let the operating system perform them. The work is divided into multiple parts that are queued and dequeued for execution when a thread becomes available, leaving the determination of availability and management of the threads to the operating system. If you want to learn more about concurrent coding and Grand Central Dispatch, see *Grand Central Dispatch (GCD) Reference*, as well as *Threading Programming Guide* and *Concurrency Programming Guide*.

The debug navigator process display views are designed to help debug problems in apps that are caused by control flow and the interactions of GCD queues. You should be familiar with the debug navigator’s display of the backtrace. If you would like to review a description of the backtrace and learn how the debug navigator presents it to you, see Examining the Backtrace in the Debug Navigator in the *Quick Start*.

The GCD Scenario

Consider a typical debugging scenario in an app that uses GCD:

- Your app has used GCD to enqueue blocks to be dequeued for execution later, as system resources permit.
- You set a breakpoint to inspect operation of a routine that is called in an queued block.
- When the app pauses at the breakpoint, the backtrace appears, organized by threads, and you start to track how your app arrived at this point.

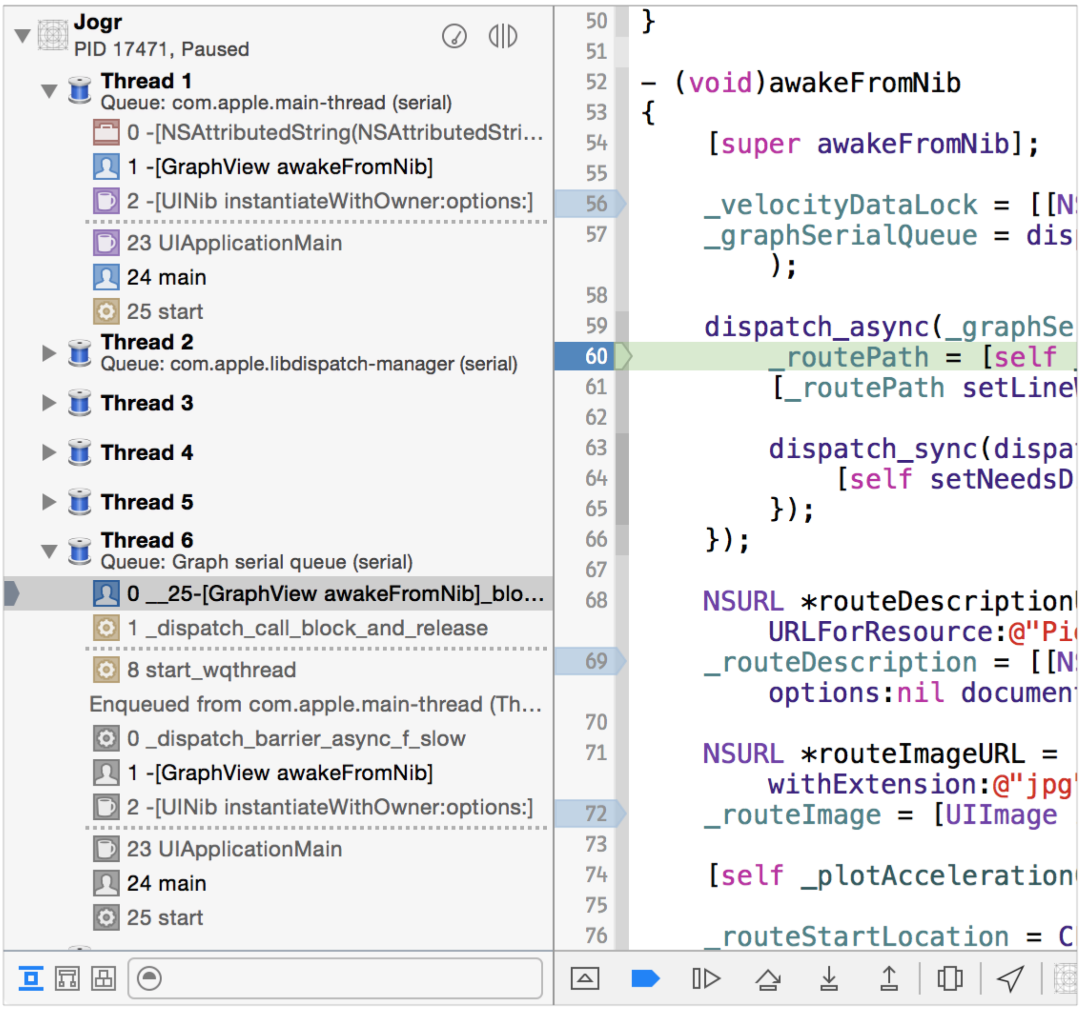
If the backtrace contains only the live code at the time it hit the breakpoint, the control flow of the app is difficult to determine because the dequeuing thread may have already been terminated. You would need to find all call sites of the function, set breakpoints appropriately, and run again, replicating the actions that cause the problem to surface. With some deep thought about the logic of your code, you deduce where the block was enqueued from and investigate what might be causing the problem. This can be a time-consuming process.

To help reduce the amount of work you need to do, Xcode splices the recorded backtrace of enqueued blocks into the backtrace, including those enqueued within other dequeued blocks. Doing this allows you to trace the control flow back to the origin with much greater ease; you can see the whole control flow in the backtrace listing. It might still be difficult to find the cause for a particular problem, because concurrent code executing on one thread might be blocked by code that hasn't been dequeued yet and the code that hasn't been dequeued yet is blocked by the executing code. You need to know how the blocks are organized in their queues for execution to determine exactly what’s happening.

Queues view reorganizes the view of blocks and threads by emphasizing the relationship between blocks and queues. Using this view you see all the queues that have been defined and loaded with both executing and queued blocks. You can more easily find whether a queue has been overloaded with too many blocks, and what block might have a lock on data that some other block needs. You have the opportunity to evaluate whether the blocks are all necessary or whether they are the result of too many enqueuing events to a single queue, and you can consider whether some operations need to be distributed to other queues, and so forth.

Threads View

In the debug navigator's default mode, threads view displays the stack frames, both live and recorded. The stack frame indicator icons are colored for the live stack frames and gray for recorded stack frames. The threads view shows the backtrace (ultimately the stack frames) organized by parent thread. This view is most useful in serially organized code operation but has its limitations when you are using async blocks.



```
Jogr
PID 17471, Paused
  Thread 1
  Queue: com.apple.main-thread (serial)
    0 -[NSAttributedString(NSAttributedStri…
    1 -[GraphView awakeFromNib]
    2 -[UINib instantiateWithOwner:options:]
    23 UIApplicationMain
    24 main
    25 start
  Thread 2
  Queue: com.apple.libdispatch-manager (serial)
  Thread 3
  Thread 4
  Thread 5
  Thread 6
  Queue: Graph serial queue (serial)
    0 __25-[GraphView awakeFromNib]_blo…
    1 _dispatch_call_block_and_release
    8 start_wqthread
    Enqueued from com.apple.main-thread (Th…
    0 _dispatch_barrier_async_f_slow
    1 -[GraphView awakeFromNib]
    2 -[UINib instantiateWithOwner:options:]
    23 UIApplicationMain
    24 main
    25 start
```

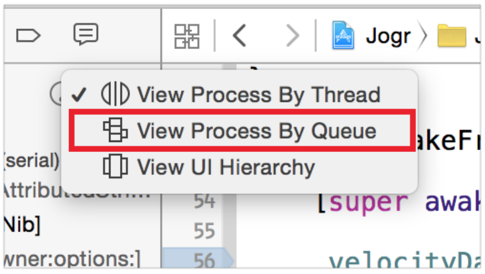
```
}

- (void)awakeFromNib
{
    [super awakeFromNib];

    _velocityDataLock = [[N
    _graphSerialQueue = dis
        );

    dispatch_async(_graphSe
        _routePath = [self 
        [_routePath setLine

        dispatch_sync(dispa
            [self setNeedsD
        });
    });

    NSURL *routeDescription
        URLForResource:@"Pi
    _routeDescription = [[N
        options:nil documen

    NSURL *routeImageURL = 
        withExtension:@"jpg
    _routeImage = [UIImage

    [self _plotAcceleration

    _routeStartLocation = C
```

In this example, the app has been paused by a breakpoint in the method `awakeFromNib` inside a block that was queued asynchronously. You can see how the historical backtrace was spliced into the display of Thread 6 to show the enqueuing event and complete the backtrace.

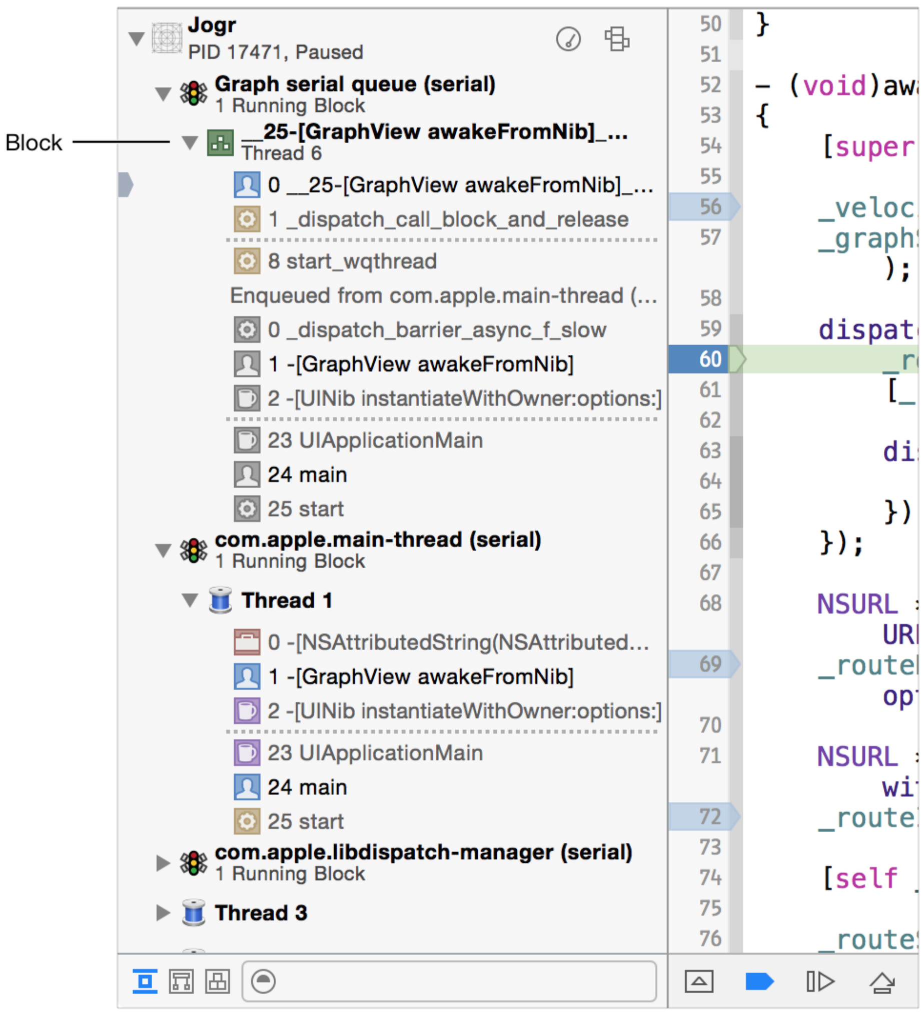
- Clicking on a stack frame for which Xcode has symbols displays the source code associated in the source editor.
- Clicking on a stack frame for which there are no symbols displays disassembly in the source editor.
- Gray icons indicate historical data that's been recorded but is no longer in memory. You can't interact with this data or inspect stack frame variables.

Queues View

The Process View Option menu allows you to switch the debug navigator to queues view.



In this mode, the debug navigator shows the backtrace organized by queue. This view emphasizes the relationship between blocks and queues: you can see which blocks are executing on which queue, and which blocks are pending.



```
Jogr
PID 17471, Paused
  Graph serial queue (serial)
  1 Running Block
    __25-[GraphView awakeFromNib]_...
    Thread 6
      0 __25-[GraphView awakeFromNib]_...
      1 _dispatch_call_block_and_release
      8 start_wqthread
      Enqueued from com.apple.main-thread (...
      0 _dispatch_barrier_async_f_slow
      1 -[GraphView awakeFromNib]
      2 -[UINib instantiateWithOwner:options:]
      23 UIApplicationMain
      24 main
      25 start
  com.apple.main-thread (serial)
  1 Running Block
    Thread 1
      0 -[NSAttributedString(NSAttributed...
      1 -[GraphView awakeFromNib]
      2 -[UINib instantiateWithOwner:options:]
      23 UIApplicationMain
      24 main
      25 start
  com.apple.libdispatch-manager (serial)
  1 Running Block
  Thread 3
```

```
}

- (void)aw
{
    [super

    _veloc
    _graph
          );

    dispat
          _r
          [_

          di

          })
    });

    NSURL
          URl
    _route
          op

    NSURL
          wi
    _route

    [self _

    _route
```

The example above has the same context as the previous threads view image. Now in queues view, you see the `awakeFromNib` block in the “Graph serial queue” that is executing on thread 6. Each block in the queue, whether executing or pending, can be disclosed (as this one is) to show its backtrace. This view mode allows you to answer questions such as these:

- How many blocks have I submitted to a queue?
- Have I oversaturated the queue?
- Is an active block on a queue “stuck” because it is waiting for an unlock that can only come from a pending block?
- If there are too many blocks on a queue, are there some that are not needed? Can the distribution of blocks be better arranged to improve the app’s responsiveness?

For a demonstration of using the debug navigator threads and queues views to solve performance and blocking issues, see the first segment of the video presentation: WWDC 2014 Debugging in Xcode 6: Queue Debugging.

Memory Graph Debugging

You use memory graph debugging to help find and fix leaked and abandoned memory. It is a special mode of debugger operation, similar to view debugging, that pauses app execution when invoked. Memory graph debugging displays objects present on the heap and their connection to/relationship with references that are keeping them alive in memory.

There are several command-line tools already available for memory analysis on macOS (leaks, heap, vmmap, and so forth). They tend to be underutilized because they are 'outside' the standard Xcode development and debugging workflow, it takes more of an effort to put them to use. Such tools are also not available to the iOS, tvOS, and watchOS device developers.

Memory graph debugging brings data from such analytic tools directly into the Xcode debugging workflow, making it more accessible and easier to take advantage of.

Background and scope of capabilities

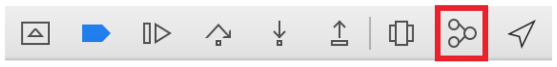
Identifying abandoned memory is often difficult; solving its causes can be challenging. Current tools like Allocations can help to identify undesirable memory growth over time, but in an ARC world solving memory growth has become a graph problem. Memory graph debugging brings the equivalent of `leaks --trace <pointer>` to Xcode debugging with a progressive disclosure UI. When debugging apps with large object graphs, this helps you answer the question “what’s holding on to my view controller that’s still allocated?”

You can quickly pause, find how many of your classes are allocated, and then summon a Quick Look or `po` an object in the console enabling you to obtain its statistics in a quick, convenient way.

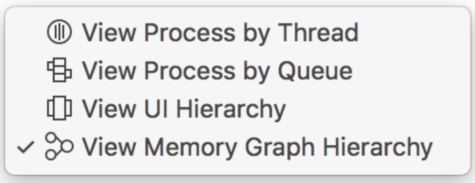
Memory graph debugging isn't trying to answer the questions of “How is my process using its memory overall?” or “What memory is counting towards my jetsam limits?” While these are important and difficult questions, the current scope of memory graph debugging focuses on leaks and abandoned memory.

Navigating the heap

To start looking for leaks and abandoned memory with your app running in Xcode, click Debug Memory Graph in the debug bar

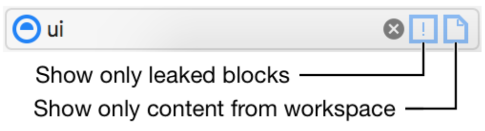


or select View Memory Graph Hierarchy in the debug navigator process view selector menu.

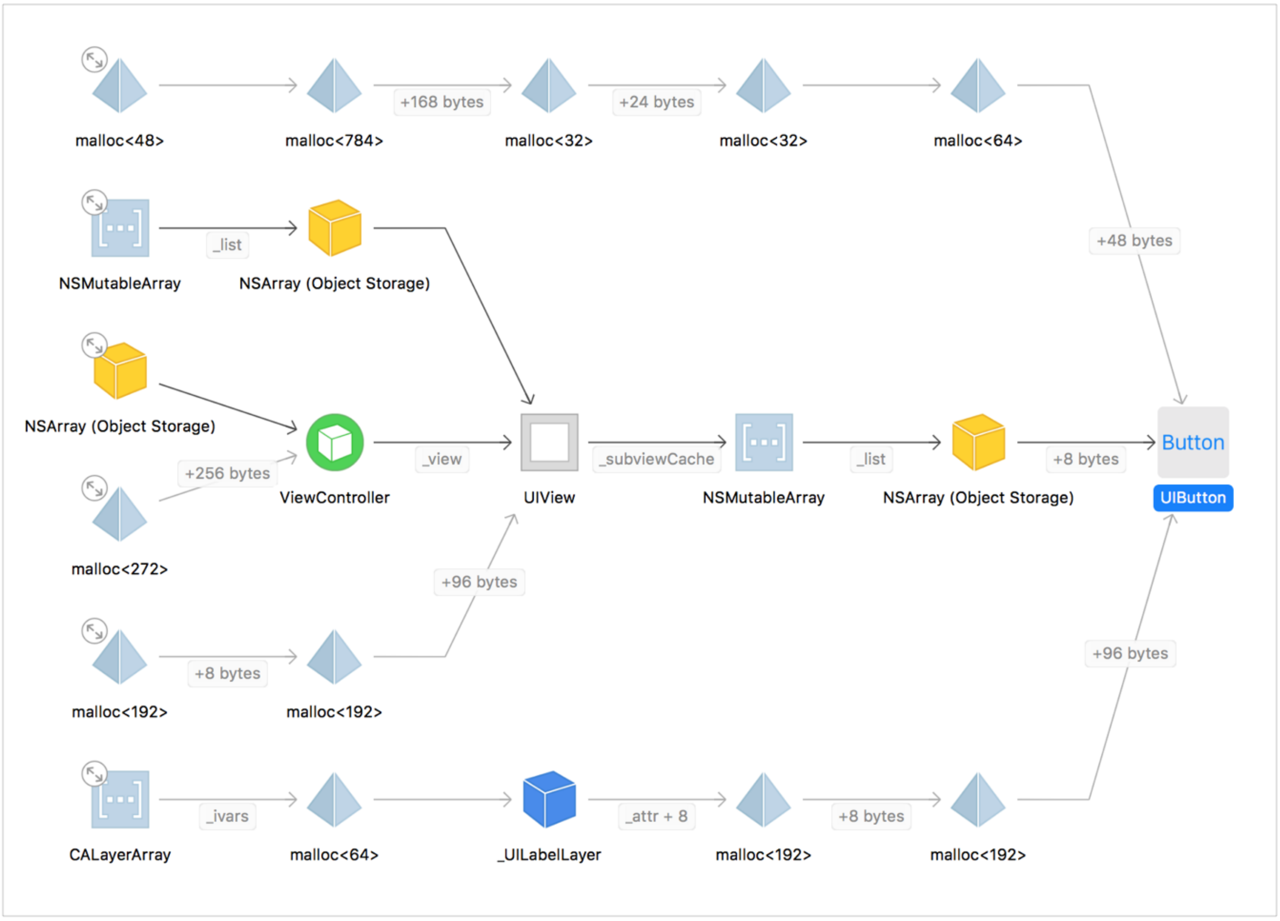


Doing this causes your app to pause. The debug navigator process view switches to show the app's heap contents, much like the command-line tool heap would show. The process view groups them into a hierarchy: Module -> Type -> Instance. Modules are sorted to put higher-level App code and frameworks at the top of the display. Types are sorted in descending order of instance counts.

The filter bar in the debug navigator allows you to filter based on module name, class name, or address. In addition, you can filter to show only leaked blocks and show only content from the workspace.



Selecting an instance in the navigator view show a graphical representation intended to help answer the question "why is this object still present in my app's heap?" by displaying the references that point to it.



You can determine from this display and from the backtrace these two situations:

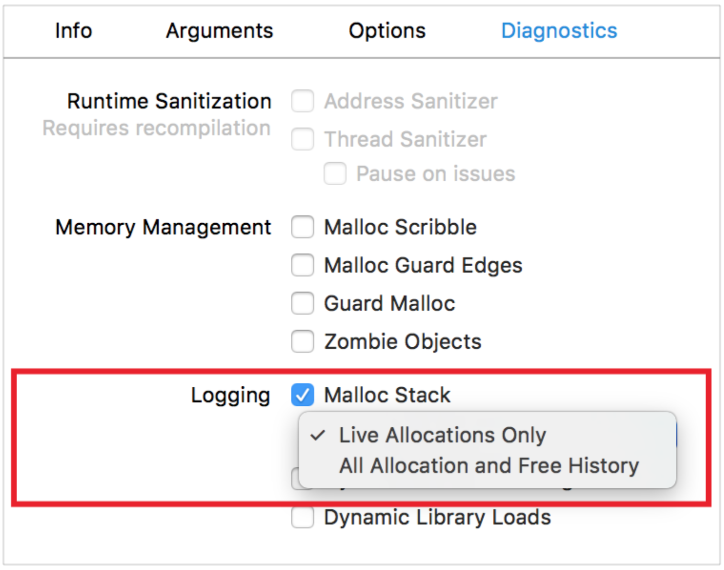
- Leaked objects — unreferenced memory that can't be used again
- Abandoned memory — referenced memory that won't be used again and should be freed

Getting backtraces

Backtraces will appear in the inspector for most memory blocks if Malloc Stack Logging is enabled in the target process.

There are two different stack logging modes you can enable:

- All Allocations (`MallocStackLogging=1`) — the stack traces of all `malloc` / `free` calls are logged to disk
- Live Allocations Only (`MallocStackLogging=lite`) — the stack traces only for currently allocated blocks and thus has much lower overhead than traditional stack logging



Runtime Sanitization
Requires recompilation
- [ ] Address Sanitizer
- [ ] Thread Sanitizer
  - [ ] Pause on issues

Memory Management
- [ ] Malloc Scribble
- [ ] Malloc Guard Edges
- [ ] Guard Malloc
- [ ] Zombie Objects

Logging
- [✓] Malloc Stack
  - ✓ Live Allocations Only
  - All Allocation and Free History
- [ ] Dynamic Library Loads

Leaked objects

Strong reference cycles are the cause for most leaks in code compiled with Automatic Reference Counting (ARC), as is true for many non-ARC leaks as well. Selecting a leaked instance will show its relationships to other leaked memory to aid you in finding the cause of the cycle. Usually the fix will be to break a reference along the chain, for example capturing 'self' weakly in a ^block rather than strongly.

When there are multiple references between nodes, the reference will show a count bubble that when clicked will display a popover listing the individual references. The inspector content or contextual menu options (Print Description, Quick Look) may also help in determining which objects are being leaked from which code-paths.

If no cycle is immediately visible when selecting a leak but other objects are shown to the left, it's likely that the other cycles or leaks to the left are strongly referencing the selected object and that these should be investigated first. If no other objects show up when a leak is selected, then using Instruments' Leaks template may be helpful for diagnosing non-ARC or __bridging retain/release issues.

Referenced objects

Most memory referenced by other objects, with paths that eventually lead all the way to 'roots' of your process — Stack memory for an active thread, or global variables in writable __DATA sections of framework/App binaries.

Identifying Abandoned Memory isn't easy, but a good starting point is to take a look at the instance counts of types you define and investigate cases where there are more instances present than you'd expect — e.g. "should there really be 15 instances of MyViewController in memory at once?"

When you select an object, it will initially appear at the right side of the graph and show incoming references from objects on the left. All incoming references are along the shortest path to some root node in the application. Darker references are known to be "strong" and more certain, while the lighter references are more "conservative" and may simply appear to be references. References are also selectable and will provide some additional information like instance variable names and offsets where available.

Fixing abandoned memory is about cutting the incoming references at the correct point, searching for the object that no longer needs to be holding a strong reference to your selected object — or potentially looking upstream to remove a reference to some of the objects that should be referencing your object.

As you're investigating a specific path, you can double-click on a node to focus on that branch and fade out other paths or right-click a node to use the "Focus Node" option of the contextual menu.

How it works

The Memory Graph Debugger in Xcode is built upon a memory graph file format (file extension `.memgraph`) designed to abstract away the architecture-specific aspects of finding malloc blocks in a target process, understanding the runtime metadata versions of the target, and scanning for references. When you summon the Memory Graph Debugger, Xcode reaches out to the debugged device, pauses the target process, and acquires its memory graph file via a service running with the same architecture as your target process. The graph debugger then searches from the roots to identify unreferenced memory (leaks) and indexes the graph to provide the heap navigator, object root analysis (for referenced blocks), and cycle analysis (for unreferenced blocks).

These `.memgraph` files contain most of the information that `leaks`, `heap`, and `vmmap` require, and can be manually acquired from the command line by running the following on the target device:

```
$ leaks --outputGraph=/tmp/ <process>
```

`leaks`, `heap`, and `vmmap` also support being run against a `.memgraph` file in place of their <process> argument, and `sysdiagnose` now outputs a `.memgraph` file when invoked targeting a single process.

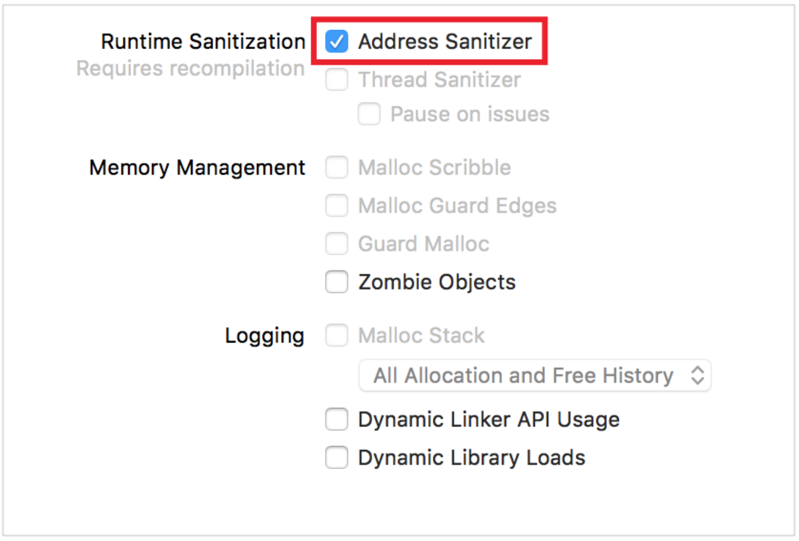
Using the Address Sanitizer

Objective-C and C code is susceptible to memory corruption issues such as stack and heap buffer overruns and use-after-free issues. When these memory violations occur, your app can crash unpredictably or display odd behavior. Memory corruption issues are difficult to track down because the crashes and odd behavior are often hard to reproduce. Also, in many situations, a problem can appear to be triggered by code that is located far from where the memory corruption that actually caused it occurred.

The address sanitizer is designed to help with problems of this kind.

You enable the address sanitizer in the build scheme for Run and Test actions using the Diagnostics tab:

1. From the Scheme toolbar menu, choose Edit Scheme.
2. In the left column, select Run.
3. Click the Diagnostics tab.
4. Select the Address Sanitizer checkbox.



Runtime Sanitization Requires recompilation	✓ Address Sanitizer
	☐ Thread Sanitizer
	☐ Pause on issues
Memory Management	☐ Malloc Scribble
	☐ Malloc Guard Edges
	☐ Guard Malloc
	☐ Zombie Objects
Logging	☐ Malloc Stack
	All Allocation and Free History
	☐ Dynamic Linker API Usage
	☐ Dynamic Library Loads

5. Click the Close button.

Note: You can also set address sanitizer to run with the Test action.

Once address sanitizer is enabled, Xcode recompiles your app the next time it runs and adds instrumentation to catch memory violations immediately and halt the app. You can inspect the problem right at the place where it occurs this way. Other diagnostic information is provided as well, such as the relationship between the faulty address and a valid object on the heap and allocation/deallocation information, which helps you pinpoint and fix the problem quickly.

Adding instrumentation to your app’s code does have a performance penalty, but address sanitizer is efficient enough to be used regularly with interactive apps. For more information and a demo of address sanitizer in action, see this video presentation: WWDC 2015: Advanced Debugging and the Address Sanitizer.

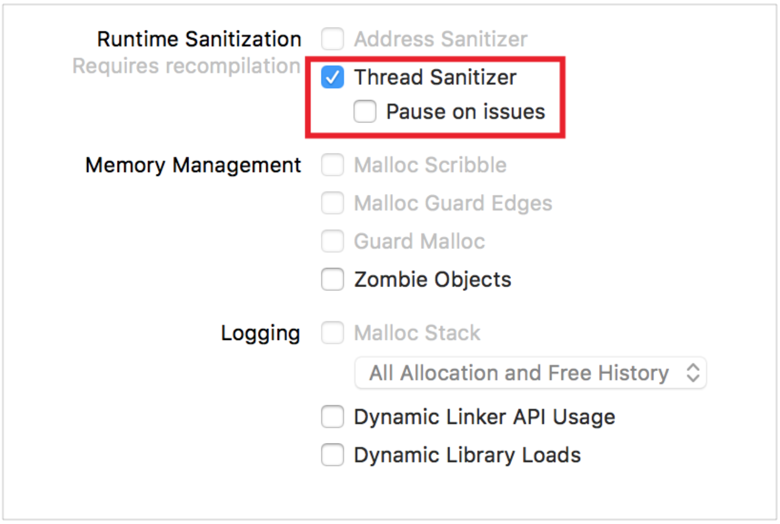
Using the Thread Sanitizer

The thread sanitizer combines compiler instrumentation and run time monitoring to help find and understand data races and other concurrency bugs in Swift and Objective-C.

Note: Thread Sanitizer is supported on 64-bit macOS and 64-bit simulators.

You enable the thread sanitizer in the build scheme for Run and Test actions using the Diagnostics tab:

1. From the Scheme toolbar menu, choose Edit Scheme.
2. In the left column, select Run.
3. Click the Diagnostics tab.
4. Select the Thread Sanitizer checkbox.



Runtime Sanitization
Requires recompilation
- [ ] Address Sanitizer
- [✓] Thread Sanitizer
  - [ ] Pause on issues

Memory Management
- [ ] Malloc Scribble
- [ ] Malloc Guard Edges
- [ ] Guard Malloc
- [ ] Zombie Objects

Logging
- [ ] Malloc Stack
  - All Allocation and Free History
- [ ] Dynamic Linker API Usage
- [ ] Dynamic Library Loads

5. Click the Close button.

When enabled, the thread sanitizer checks for uninitialized mutexes, thread leaks, unsafe calls in signal handlers, and data races.

Like the address sanitizer, running an app with the thread sanitizer introduces a small performance penalty, which should be negligible when interacting with most apps. For more information and a demo of thread sanitizer in action, see this video presentation: WWDC 2016: Thread Sanitizer and Static Analysis.

Debugging Metal and OpenGL ES

Xcode provides tools for debugging, analyzing, and tuning code that uses Metal and OpenGL ES. These tools are useful during all stages of development. The FPS Debug Gauge and GPU report summarize your app’s GPU performance every time you run it from Xcode, allowing you to quickly spot performance issues while designing and building your renderer. When you find a trouble spot, you can capture a frame and use Xcode’s GPU Frame Debugger interface to pinpoint rendering problems and solve performance issues.

This section reviews the workflow and user interface basics of the GPU Frame Debugger in Xcode. A detailed look at the GPU Frame Debugger is available in Xcode OpenGL ES Tools Overview.

Workflow Basics

For a detailed look at your app's GPU graphics usage, you capture the sequence of commands used to render a single frame of animation. Xcode offers several ways to begin a frame capture:

- **Manual capture.** While running your app in Xcode, click the Camera icon ("Capture GPU Frame") in the debug bar or choose Debug > Capture GPU Frame.

> **Note:** The Capture GPU Frame button appears only if your project links against the Metal, OpenGL ES, or SpriteKit framework. You can choose whether it appears for other projects by editing the active scheme.

- **Breakpoint action.** Choose Capture GPU Frame as an action for any breakpoint. When the debugger reaches a breakpoint with this action, Xcode automatically captures a frame. If you use this action with an OpenGL ES Error breakpoint while developing your app, you can use the GPU Frame Debugger to investigate the causes of OpenGL ES errors whenever they occur.
- **OpenGL ES event marker.**Programmatically trigger a frame capture by inserting an event marker in the OpenGL ES command stream programmatically. When the OpenGL ES client reaches this marker, it finishes rendering the frame, and then Xcode automatically captures the entire sequence of commands used to render that frame.

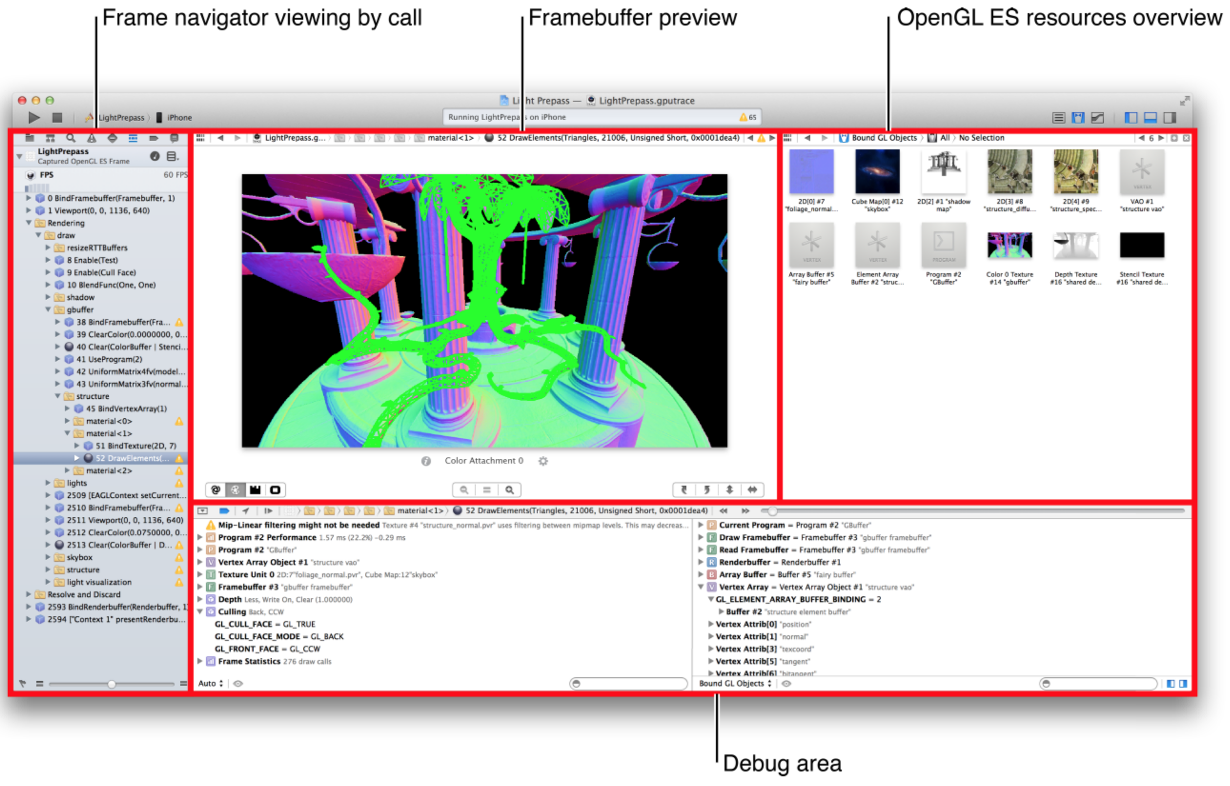
After Xcode captures the frame with any of these methods, it displays the GPU Frame Debugger interface. You use this interface to inspect the sequence of graphics commands that render the frame and examine graphics resources.

The GPU Frame Debugger UI

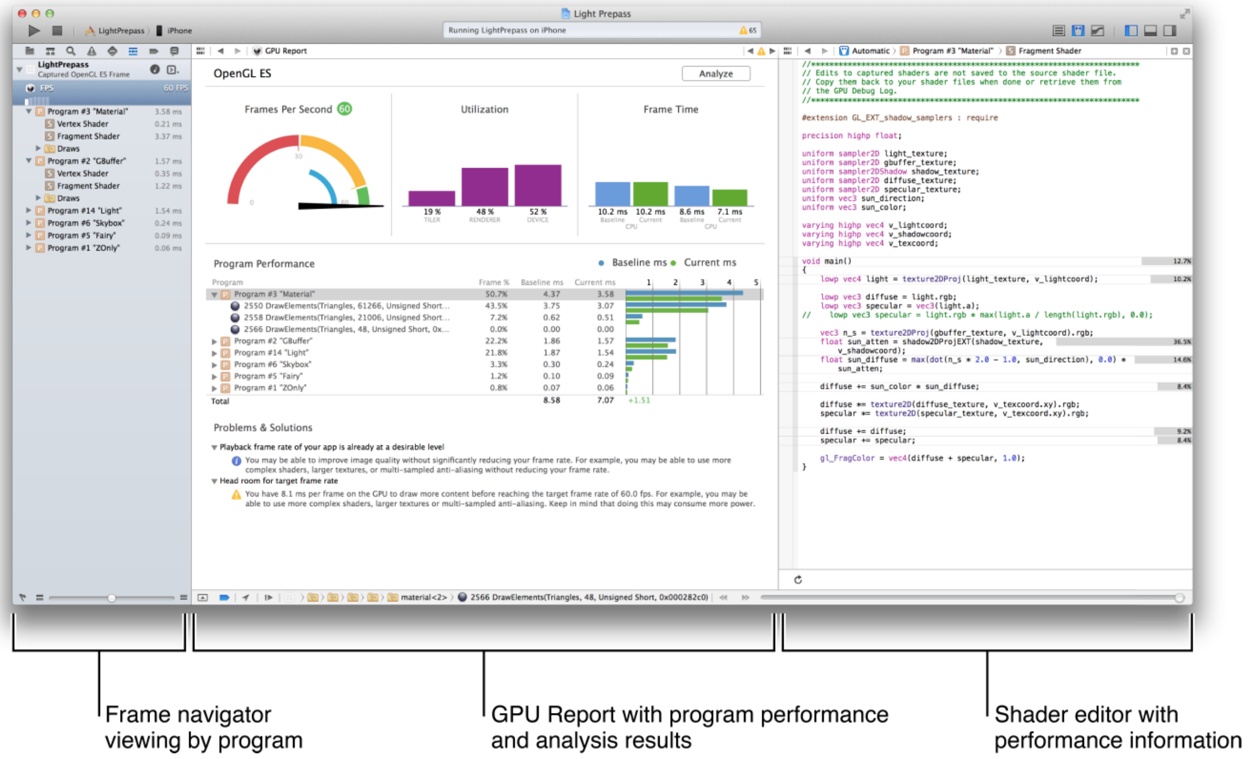
After Xcode captures a frame, it reconfigures the user interface for OpenGL ES or Metal debugging. The GPU Frame debugger interface modifies several areas of the Xcode workspace window in order to provide information about the GPU graphics rendering process, as shown below. The frame debugger does not use the inspector or library panes, so you may want to hide the Xcode utility area during GPU Frame debugging in order to increase the available space for inspection and debugging.

Examples of the GPU Frame Debugger UI

This figure below shows the UI presented when examining draw calls and resources.



The figure below shows the UI presented when examining shader program performance and analysis results.



LightPrepass
Captured OpenGL ES Frame

FPS	60 FPS
Program #3 "Material"	3.58 ms
Vertex Shader	0.21 ms
Fragment Shader	3.37 ms
Draws	
Program #2 "GBuffer"	1.57 ms
Vertex Shader	0.35 ms
Fragment Shader	1.22 ms
Draws	
Program #14 "Light"	1.54 ms
Program #6 "Skybox"	0.24 ms
Program #5 "Fairy"	0.09 ms
Program #1 "ZOnly"	0.06 ms

OpenGL ES

Program Performance

● Baseline ms ● Current ms

Program	Frame %	Baseline ms	Current ms
Program #3 "Material"	50.7%	4.37	3.58
2550 DrawElements(Triangles, 61266, Unsigned Short…	43.5%	3.75	3.07
2558 DrawElements(Triangles, 21006, Unsigned Short…	7.2%	0.62	0.51
2566 DrawElements(Triangles, 48, Unsigned Short, 0x…	0.0%	0.00	0.00
Program #2 "GBuffer"	22.2%	1.86	1.57
Program #14 "Light"	21.8%	1.87	1.54
Program #6 "Skybox"	3.3%	0.30	0.24
Program #5 "Fairy"	1.2%	0.10	0.09
Program #1 "ZOnly"	0.8%	0.07	0.06
Total		8.58	7.07

+1.51

Problems & Solutions

▼ Playback frame rate of your app is already at a desirable level

You may be able to improve image quality without significantly reducing your frame rate. For example, you may be able to use more complex shaders, larger textures, or multi-sampled anti-aliasing without reducing your frame rate.

▼ Head room for target frame rate

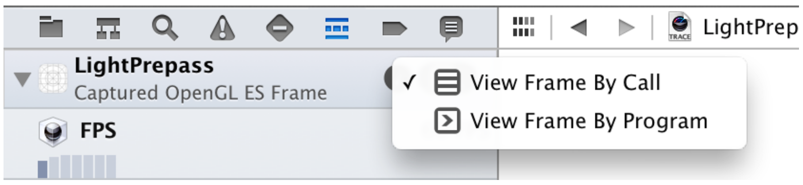
You have 8.1 ms per frame on the GPU to draw more content before reaching the target frame rate of 60.0 fps. For example, you may be able to use more complex shaders, larger textures or multi-sampled anti-aliasing. Keep in mind that doing this may consume more power.

material<2> 〉 2566 DrawElements(Triangles, 48, Unsigned Short, 0x000282c0)

```
//************************************************************************
// Edits to captured shaders are not saved to the source shader file.
// Copy them back to your shader files when done or retrieve them from
// the GPU Debug Log.
//************************************************************************

#extension GL_EXT_shadow_samplers : require

precision highp float;

uniform sampler2D light_texture;
uniform sampler2D gbuffer_texture;
uniform sampler2DShadow shadow_texture;
uniform sampler2D diffuse_texture;
uniform sampler2D specular_texture;
uniform vec3 sun_direction;
uniform vec3 sun_color;

varying highp vec4 v_lightcoord;
varying highp vec4 v_shadowcoord;
varying highp vec4 v_texcoord;

void main()                                                             12.7%
{
    lowp vec4 light = texture2DProj(light_texture, v_lightcoord);       10.2%

    lowp vec3 diffuse = light.rgb;
    lowp vec3 specular = vec3(light.a);
//    lowp vec3 specular = light.rgb * max(light.a / length(light.rgb), 0.0);

    vec3 n_s = texture2DProj(gbuffer_texture, v_lightcoord).rgb;
    float sun_atten = shadow2DProjEXT(shadow_texture,                   36.5%
        v_shadowcoord);
    float sun_diffuse = max(dot(n_s * 2.0 - 1.0, sun_direction), 0.0) *  14.6%
        sun_atten;

    diffuse += sun_color * sun_diffuse;                                   8.4%

    diffuse *= texture2D(diffuse_texture, v_texcoord.xy).rgb;
    specular *= texture2D(specular_texture, v_texcoord.xy).rgb;

    diffuse += diffuse;                                                   9.2%
    specular += specular;                                                 8.4%

    gl_FragColor = vec4(diffuse + specular, 1.0);
}
```

Navigator Area

In the frame debugger interface, the debug navigator has been replaced by the frame navigator. This navigator shows the commands that render the captured frame, organized sequentially or according to their associated shader program. Use the Frame View Options pop-up menu at the top of the frame navigator to switch between view styles.



View Frame by Call

Use this option to view the captured frame by call when you want to study OpenGL ES commands in sequence to pinpoint errors, diagnose rendering problems, or identify common performance issues. In this mode, the frame navigator lists commands in the order your app called them.

Clicking a command in the list navigates to that point in the command sequence, affecting the contents of other areas of the frame debugger interface and showing the effects of the calls up to that point on the attached device's display.

View Frame by Program

Use this option to view the captured frame by program when you want to analyze the GPU time spent on each shader program and draw command.

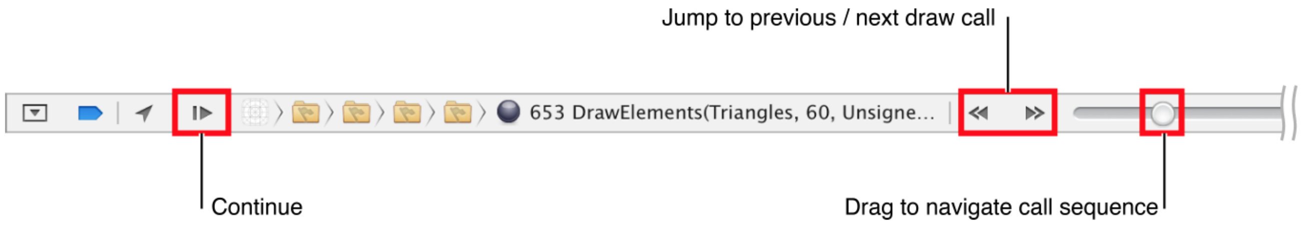
Clicking a program or shader shows the corresponding GLSL source code in the primary editor. Clicking a command navigates to that point in the frame capture sequence.

Editor Area

When working with a frame capture, you use the primary editor to preview the framebuffer being rendered to, and the assistant editor to examine GPU graphics resources and edit GLSL shader programs. By default, the assistant editor shows a graphical overview of all resources currently owned by the GPU graphics context. You can use the assistant editor's jump bar to show only those resources bound for use as of the call selected in the frame navigator, or to select an individual resource for further inspection. You can also double-click a resource in the overview to inspect it. When you select a resource, the assistant editor changes to a format suited for tasks appropriate to that resource's type.

Debug Area

The debug bar provides multiple controls for navigating the captured sequence of commands. You can use its menus to follow the hierarchy shown in the frame navigator and choose a command, or you can use the arrows and slider to move back and forth in the sequence. Press the Continue button to end frame debugging and return to running your app.


Jump to previous / next draw call
653 DrawElements(Triangles, 60, Unsigne...
Continue
Drag to navigate call sequence

The frame debugger has no debug console. Instead, Xcode offers multiple variables views, each of which provides a different summary of the current state of the GPU graphics rendering process. Use the pop-up menu to choose between the available variables views:

The All GL Objects View

The All GL Objects view lists the same OpenGL ES resources as the graphical overview in the assistant editor. Unlike the graphical overview, however, this view can provide more detailed information about a resource when you expand its disclosure triangle. Expanding the listing for a shader program shows its status, attribute bindings, and the currently bound value for each uniform variable.

The Bound GL Objects View

The Bound GL Objects view behaves identically to the All GL Objects view, but lists only resources currently bound for use as of the selected OpenGL ES command in the frame navigator.

The GL Context View

The GL Context view lists the entire state vector of the OpenGL ES renderer, organized into functional groups. When you select a call in the frame navigator that changes OpenGL ES state, the changed values appear highlighted.

The Context Info View

The Context Info view lists static information about the OpenGL ES renderer in use: name, version, capabilities, extensions, and similar data. You can look through this data instead of writing your own code to query renderer attributes such as `GL_MAX_TEXTURE_IMAGE_UNITS` and `GL_EXTENSIONS`.

The Auto View

The Auto view automatically lists a subset of items normally found in the other variables views and other information appropriate to the selected call in the frame navigator. For example:

- If the selected call results in an OpenGL ES error or if Xcode has identified possible performance issues with the selected call, the view lists the errors or warnings and suggested fixes for each.
- If the selected call changes part of the OpenGL ES context state or its behavior is dependent on context state, the view automatically lists relevant items from the GL Context view.

- If the selected call binds a resource or makes use of bound resources such as vertex array objects, programs, or textures, the view automatically lists relevant items from the Bound GL Objects view.
- If a draw call is selected, the view lists program performance information, including the total time spent in each shader during that draw call and, if you've changed and recompiled shaders since capturing the frame, the difference from the baseline time spent in each shader. (Program performance information is available only when debugging on an OpenGL ES 3.0–capable device.)

In addition, this view lists aggregate statistics about frame rendering performance, including the number of draw calls and frame rate.

Using Alternative Toolchains

When you activate an alternative toolchain, Xcode uses the version of LLDB supplied by that toolchain for debugging. In this way, the debugger matches the installed compiler and any other toolchain components with respect to any language changes: the debugger is able to inspect any new or different runtime structures created by the alternate toolchain and also understands any new syntax in the expression parser.

Other than the Toolchain button located in the Xcode window toolbar (see Viewing and Switching Toolchains), you should experience no changes in UI or debugger workflow.

Important: If you use an alternative toolchain, you might encounter regressions because the alternative tools won't have been fully qualified by Apple for official release.