Debugging with Xcode

Developer

Contents

About Debugging with Xcode 4
Prerequisites 4
See Also 4

Quick Start 5
Getting Ready to Debug 7
Using Breakpoints 9
Controls in the Debugger 16
    Stepping Controls in the Debug Bar 17
    Inspecting Variables with Variables View 18
    Understanding the Console 21
Examining the Backtrace in the Debug Navigator 22
Cycling Through the Debugging Process 27

Debugging Tools 28
General Notes 29
    The Five Parts of Debugging and the Debugging Tools 30
    LLDB and the Xcode Debugger 30
    Xcode Toolbar Controls 31
    Xcode Debug and Product Menus 32
Debug Area 32
    Debug Bar – Process Controls 32
    Variable View – Inspecting Variables 34
    Console — Command Line Input/Output 40
Breakpoints and the Breakpoint Navigator 41
    Breakpoints 41
    Breakpoint Navigator 43
Source Editor 48
The Debug Navigator 53
    Debug Gauges 54
    Process View Display 55
Debugging the View Hierarchy 58
OpenGL ES Debugger 58
Debugging Options in the Scheme Editor 59

Specialized Debugging Workflows 65
Debugging View Hierarchies 65
Basic Operation 65
The View Hierarchy Display 67
The Object and Size Inspectors 68
The Assistant Editor 68
Using the View Debugger 68
Debugging App Extensions 69
Build and Run for Debugging with Ask On Launch 70
Enabling App Extensions 72
Performance Monitoring 72
Thread and Queue Debugging 72
Background Information 73
The GCD Scenario 73
Threads View 75
Queues View 76
Debugging OpenGL ES and Metal Graphics 78
Workflow Basics 78
The OpenGL ES Frame Debugger UI 79

Quick Look Data Types 83

Document Revision History 86

About Debugging with Xcode

Finding and eliminating problems in your code is a critical part of the development process. The Xcode debugger is preset with useful features for general purpose debugging and runs automatically when your application is launched. The debugger helps you:

- Identify and locate the problem
- Examine the control flow and data structures of running code to find the cause
- Devise a solution and edit your code accordingly
- Run the revised app and confirm that the fix works

Prerequisites

You should be familiar with app design and programming concepts. Some familiarity with Xcode is also recommended; see *Xcode Overview*.

See Also

Every year, several sessions at the Apple Worldwide Developer Conference are devoted to debugging that expand upon the material in this guide and add to it with useful techniques. These sessions are available for you in the Apple developer libraries at the Apple Developer website and are easy to find by filtering on "debug."

The following recent WWDC presentations focus on using the Xcode debugger:

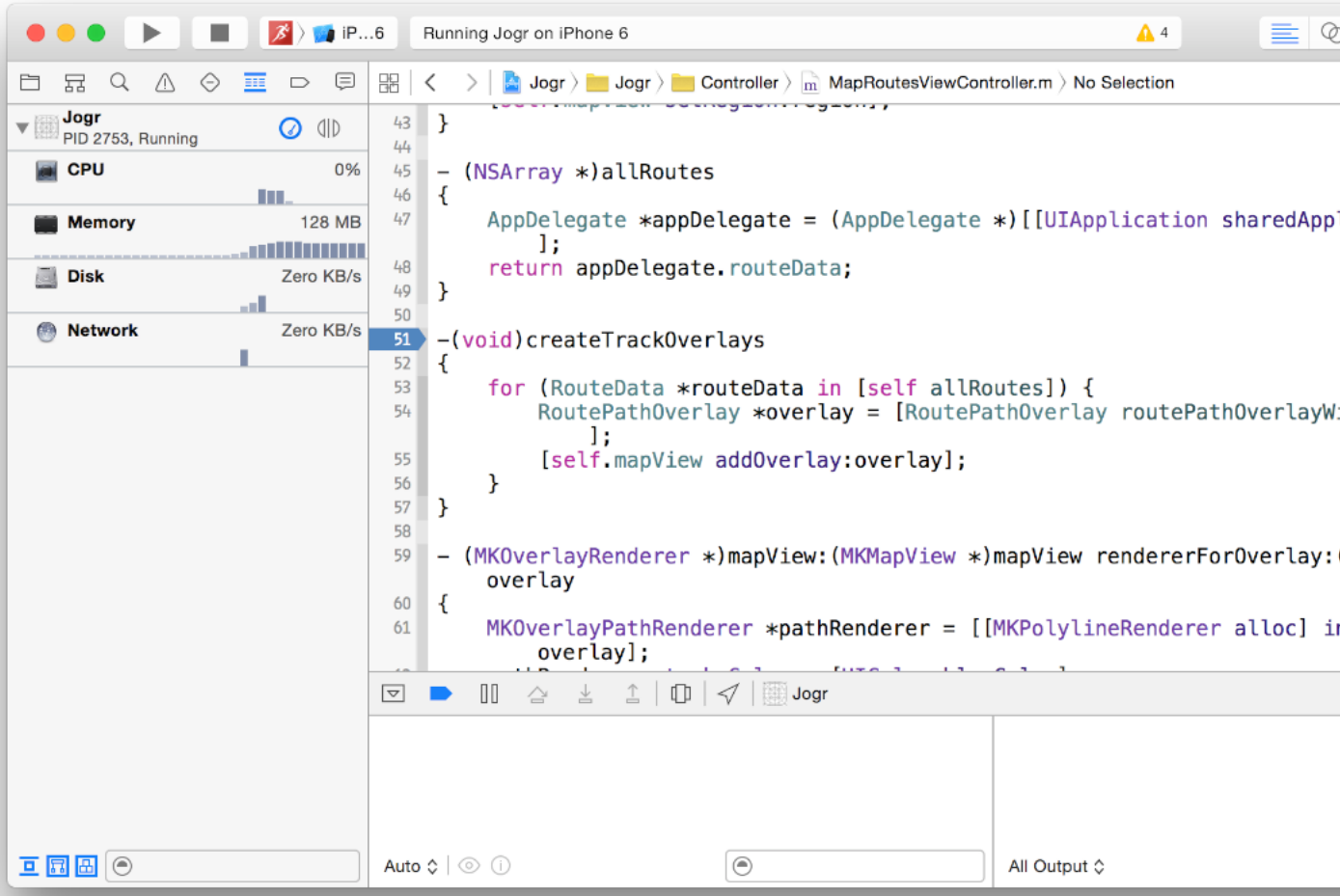
- WWDC 2014: Debugging with Xcode 6: Learn how apps enqueue work, explore and fix user interfaces, add custom Quick Look support.
- WWDC 2013: Debugging with Xcode: Detect and fix performance problems using the Xcode graphical debugger.
- WWDC 2013: Advanced Debugging with LLDB: Debug using Terminal and the Xcode graphical debugger.

A good primer on debugging in general is *The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems* by David J. Agans.

Quick Start

Imagine this situation: You've got a successful app to which you are adding a new feature. After you click the Run button in the workspace toolbar and your app builds successfully, Xcode runs your app and starts a debugging session.

While your app is running, the Xcode window adjusts for debugging by opening the debug area at the bottom and the debug navigator at the left. In the debug navigator, the debug gauges help monitor your program's use of system resources. You start to exercise your app and use some of the new code you just added to display a picture or a graph ... and nothing is displayed.



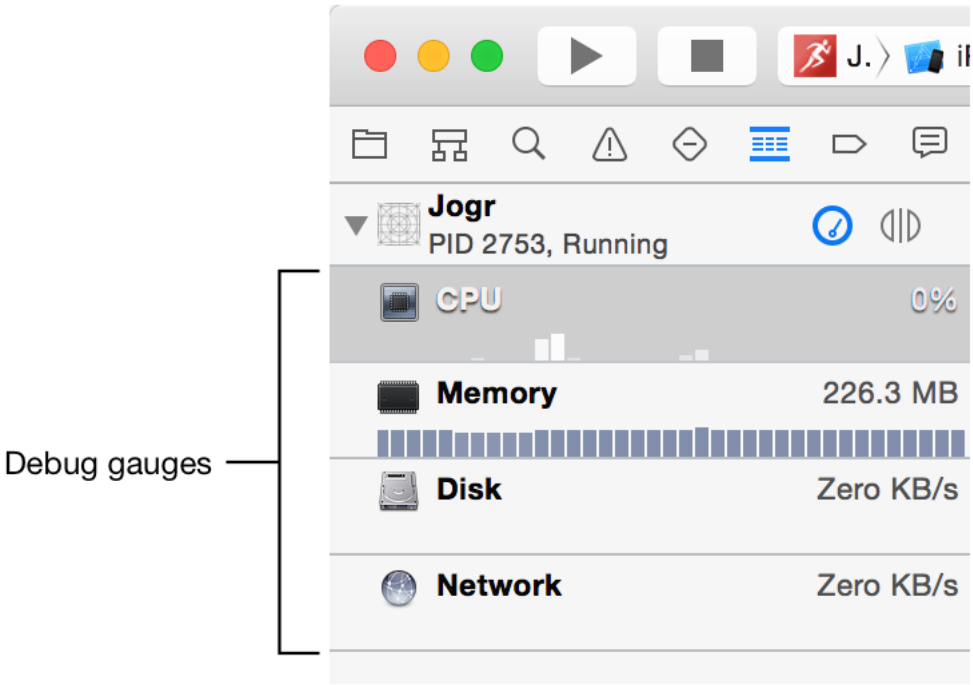
What should you do now?

Getting Ready to Debug

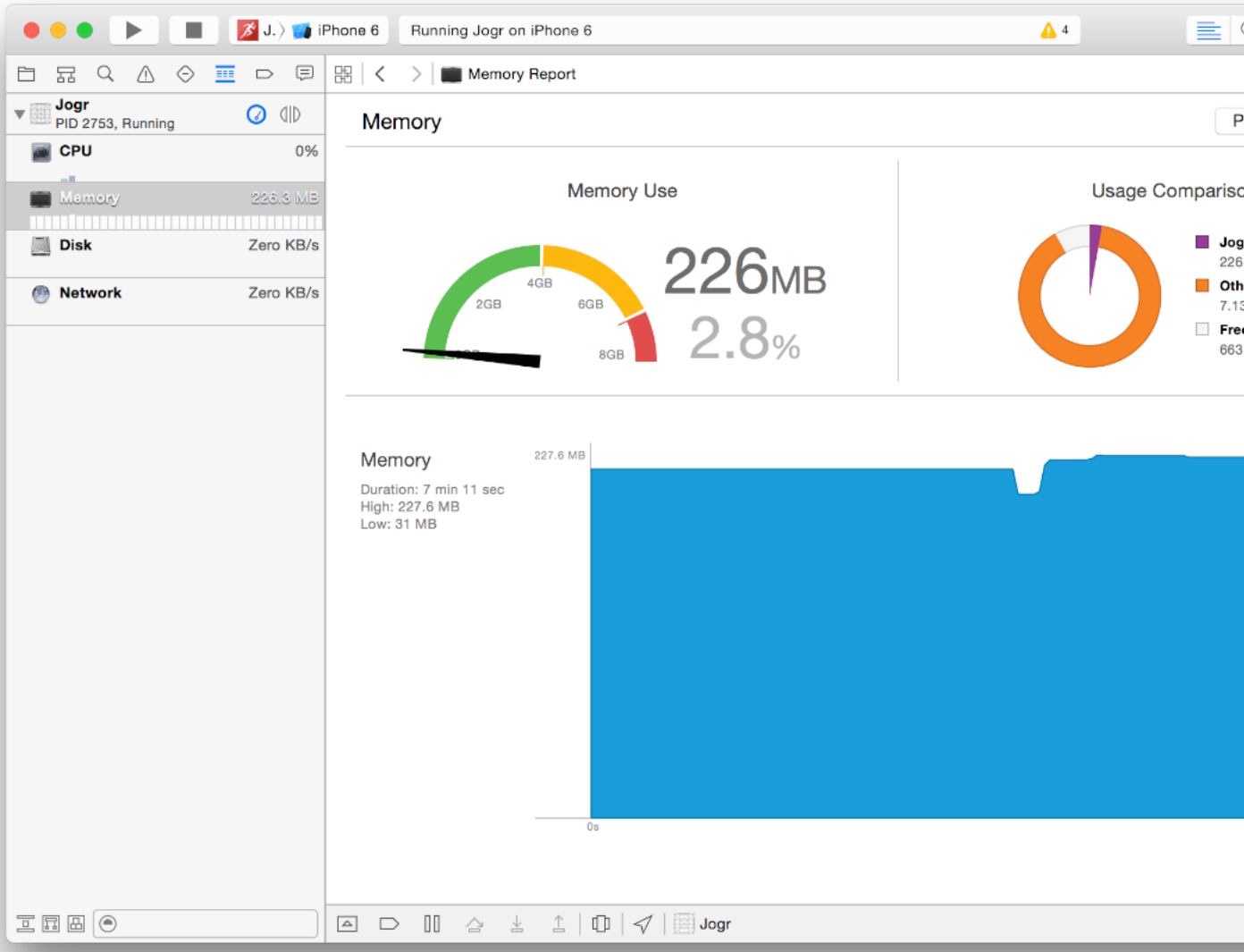
Two of the most important things to do in debugging are to analyze the logic, or control flow, of the code, and to be sure the app presents correct data. Apps with logic problems exhibit unexpected behavior: nothing happens (as in the hypothetical situation described above), the wrong thing happens, or the app crashes. Code that performs incorrect operations often presents the wrong data to the user; imagine pressing buttons on a calculator to add *1* and *1* and obtaining the result *20*. You debug when you see an issue, to make sure the app behaves as planned and the right results are being shown to the user.

In your hypothethical situation, your new code didn't seem to be executed. You don't know yet whether the code wasn't hit, or whether it did but didn't produce an output. But you know that something is amiss—you have a bug.

Other problem situations are more difficult to spot. Exercising an app in the debugger gives you the opportunity to monitor its use of system resources using the debug gauges. Pay attention to the shape of the graph, and think about what you should expect to see as the app runs and various parts of the code are called into play. If you see unexpected spikes in the CPU gauge or a constantly rising pattern in the memory gauge, for example, these might indicate more subtle kinds of problems.



Clicking a gauge opens up a more detailed report for a higher-resolution view of the situation.



In both the case of your new code not doing what you expect and of unexpected indications in the debug gauges, you likely do not immediately know the cause of the problem. However, you already have some information:

- The app behaved correctly before you started adding the new feature.
- The app behaved correctly up to the point where you used the app's UI to run the new code.
- You know what new code you added to the source.

To further isolate the cause, you can set *breakpoints* in the code.

Using Breakpoints

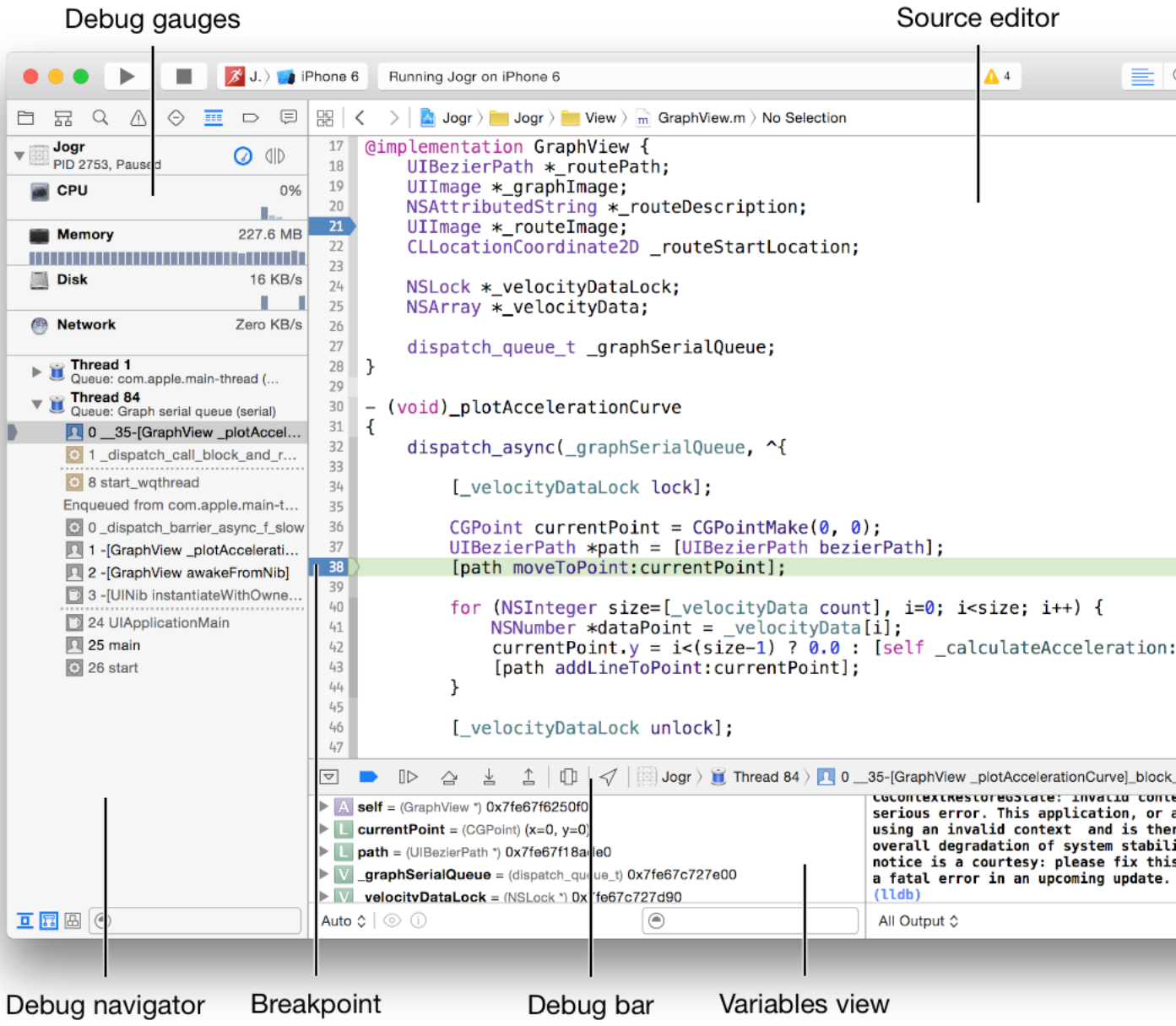
Breakpoints are a versatile solution for inspecting a running app.

A breakpoint interrupts normal execution so that you can run your app step by step with the debug bar controls to examine the flow of control and state of variables at every line of code. When your code encounters a breakpoint, the debugger pauses the app, switches to the Xcode main window with your source positioned for you to examine, and populates the variables view and the debug navigator process view with tools for looking at the state of the paused app. Breakpoints have several features you can use to modify their behavior and make it easier for you to gather information. You can add breakpoints to your source before you run the app or you can add them to the code while it is running.

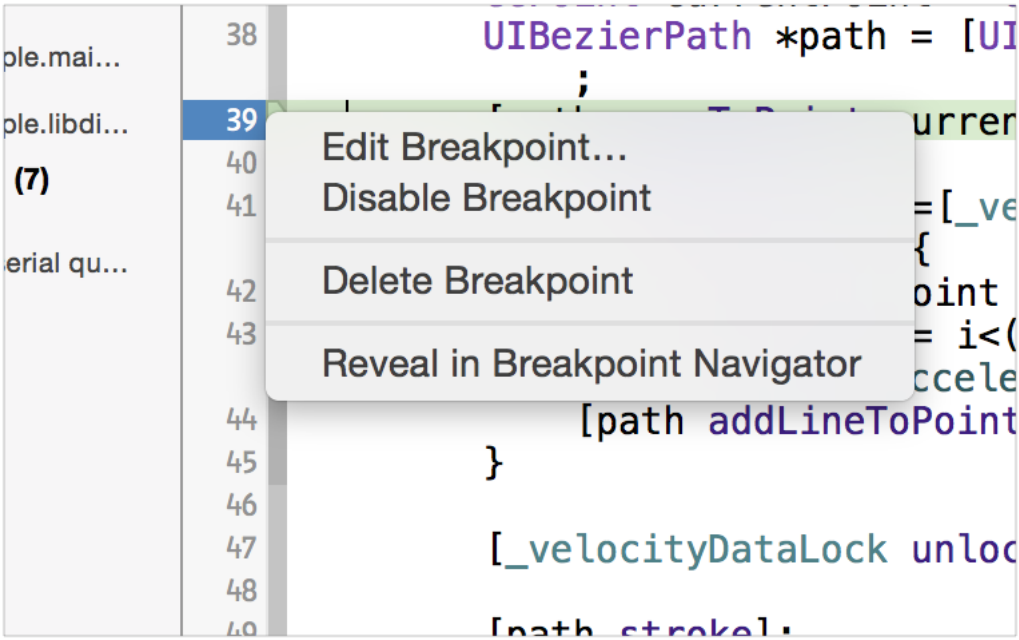
To insert a breakpoint:

1. Position the source in the source editor.
2. Click the source editor gutter next to the line of source where you the debugger to pause your app.

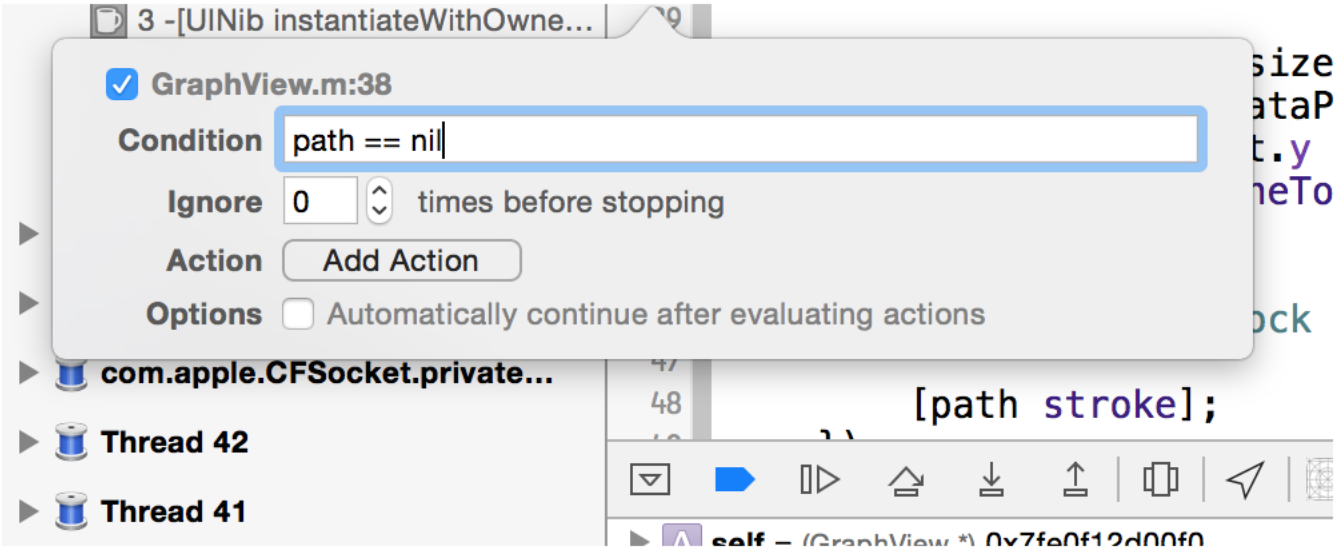
A file and line breakpoint is created, and it is enabled by default.



You use the controls in the debug bar to step in various ways, as you'll see in Controls in the Debugger (page 16). Stepping through a section of code manually is the high resolution way to examine your code, and it can also be time consuming. Once you have put a breakpoint in your code, you can set it to operate in different ways to increase the efficiency of your debugging. Control-click the breakpoint to show the context sensitive menu, then choose Edit Breakpoint.

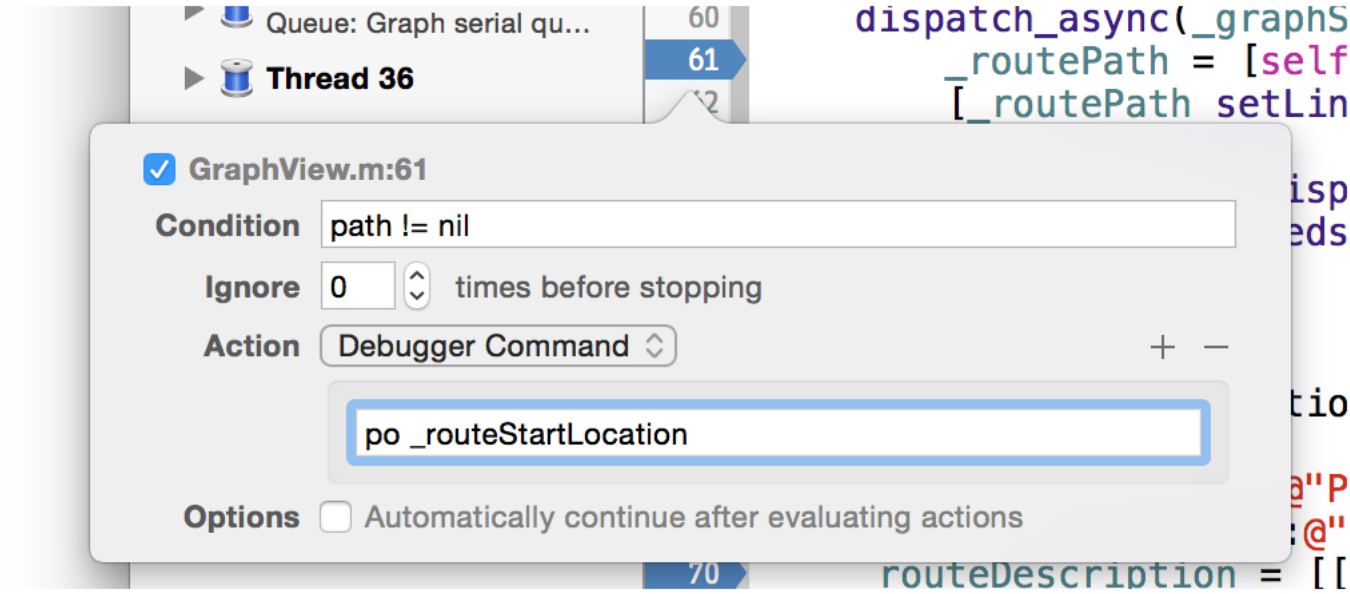


You can configure breakpoints to halt the program based on a condition. Because your app is running when breakpoints are active, you can use any code or variable in your app that is in scope at the point of the source where the breakpoint is activated to test for a condition.



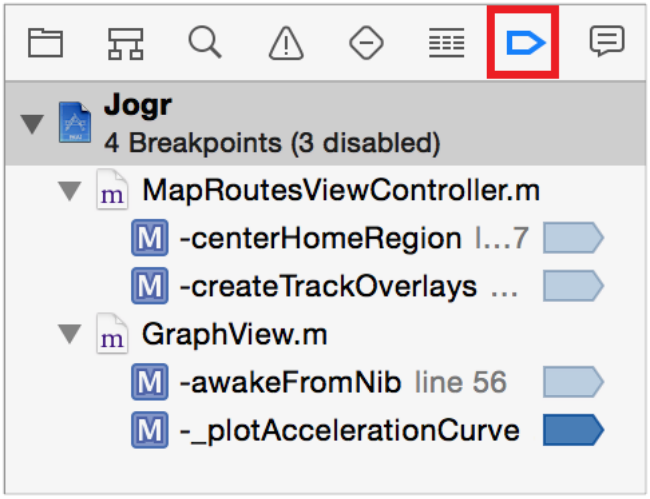
Breakpoint conditions are useful in many ways. For one example, say your new feature relies on the state of a variable. You notice that when a problem has occurred and you inspect a the variable with the app paused at a breakpoint, that variable always has a particular unusual state—nil or some other unusual value. Once you know that, you can set up a breakpoint condition to monitor the variable and pause the app only when the variable has that value.

Another useful capability of breakpoints is the ability to trigger an action to be performed.



Given the above situation where a loop is incrementing and a variable needs to achieve a particular state, maybe you are interested to know what the value of the variable is each time the loop executes. You can set the breakpoint action to print the variable description to the console at each pass through the loop using the LLDB command `po` (`po` is the provided LLDB abbreviation for the "print object" command).

Breakpoints in your code are managed with the breakpoint navigator.

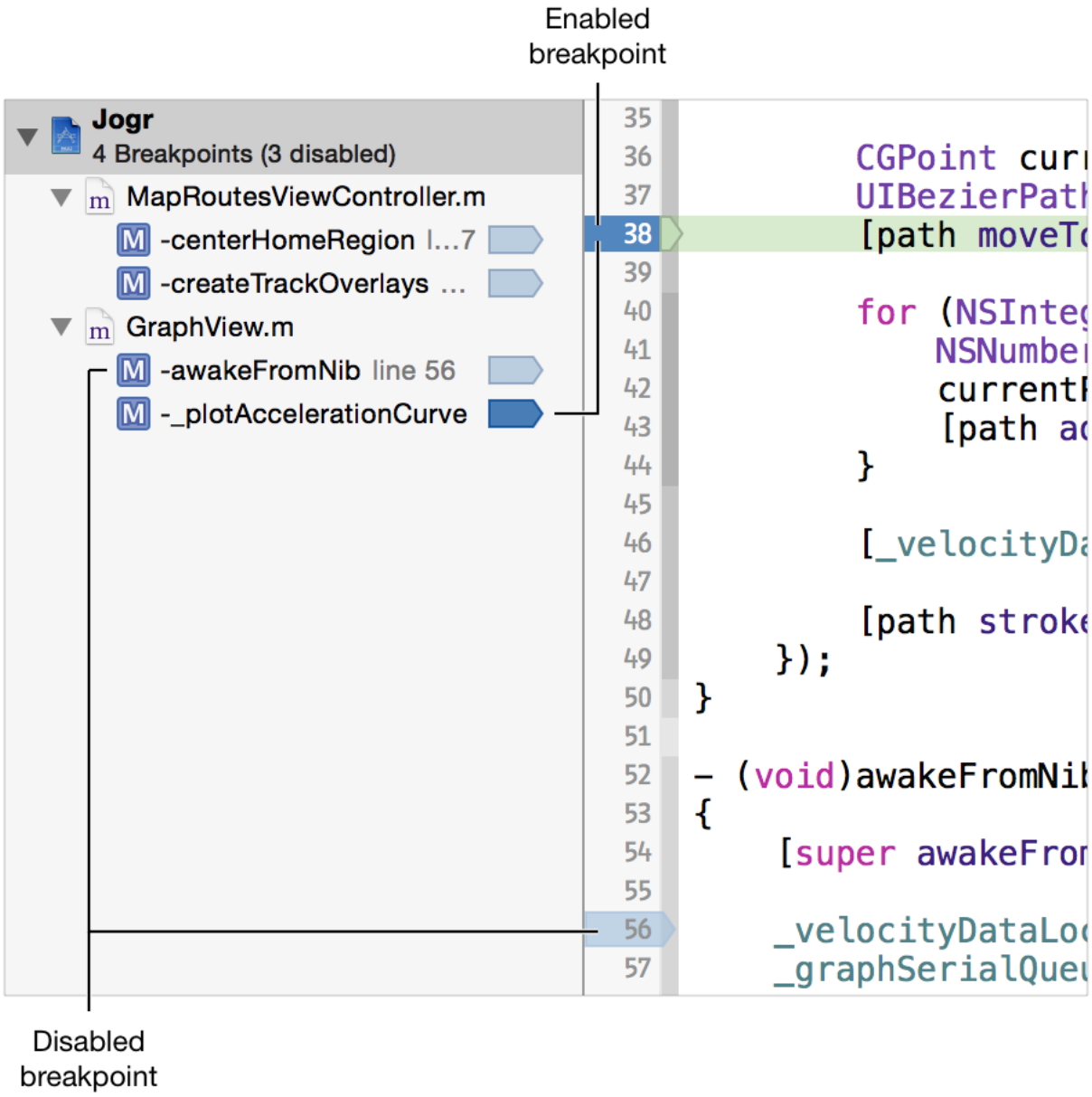


Using the breakpoint navigator, you can see all the breakpoints set in your code, edit breakpoints, enable and disable them, and change their scope of operation in the Xcode context. To enable or disable a breakpoint, click its indicator in the breakpoint navigator or in the source editor; a dimmed indicator indicates the breakpoint is disabled.

To delete a breakpoint when you finish using it, do one of the following:

- Drag it out of the source editor gutter.
- Select it in the breakpoing navigator and press Delete.

- Control-click it (in the source editor or the breakpoint navigator) and choose Delete Breakpoint.

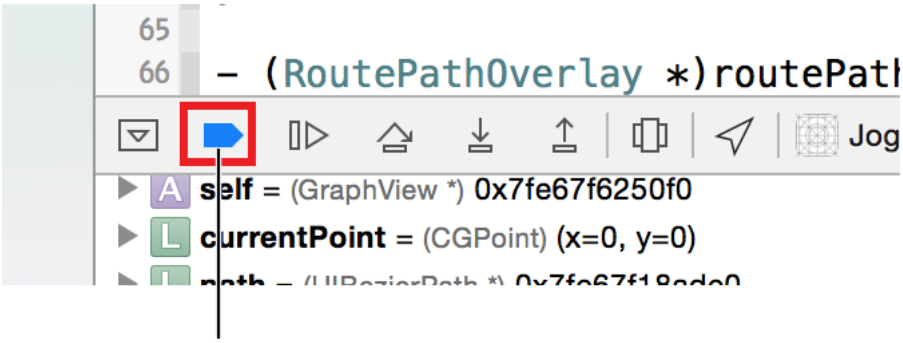


You can enable or disable multiple breakpoints at the same time. You might want to do that if, for example, you identify several problems located in different parts of the code.

Clicking a breakpoint row in the breakpoint navigator moves that source into the source editor at the breakpoint location.

There are times when you have placed a set of breakpoints for debugging a problem but temporarily need to run your app without having it pause so that you can reach the state at which the problem you're debugging is likely to occur. To deactivate or activate all breakpoints, click the Breakpoint activation button in the debug bar.

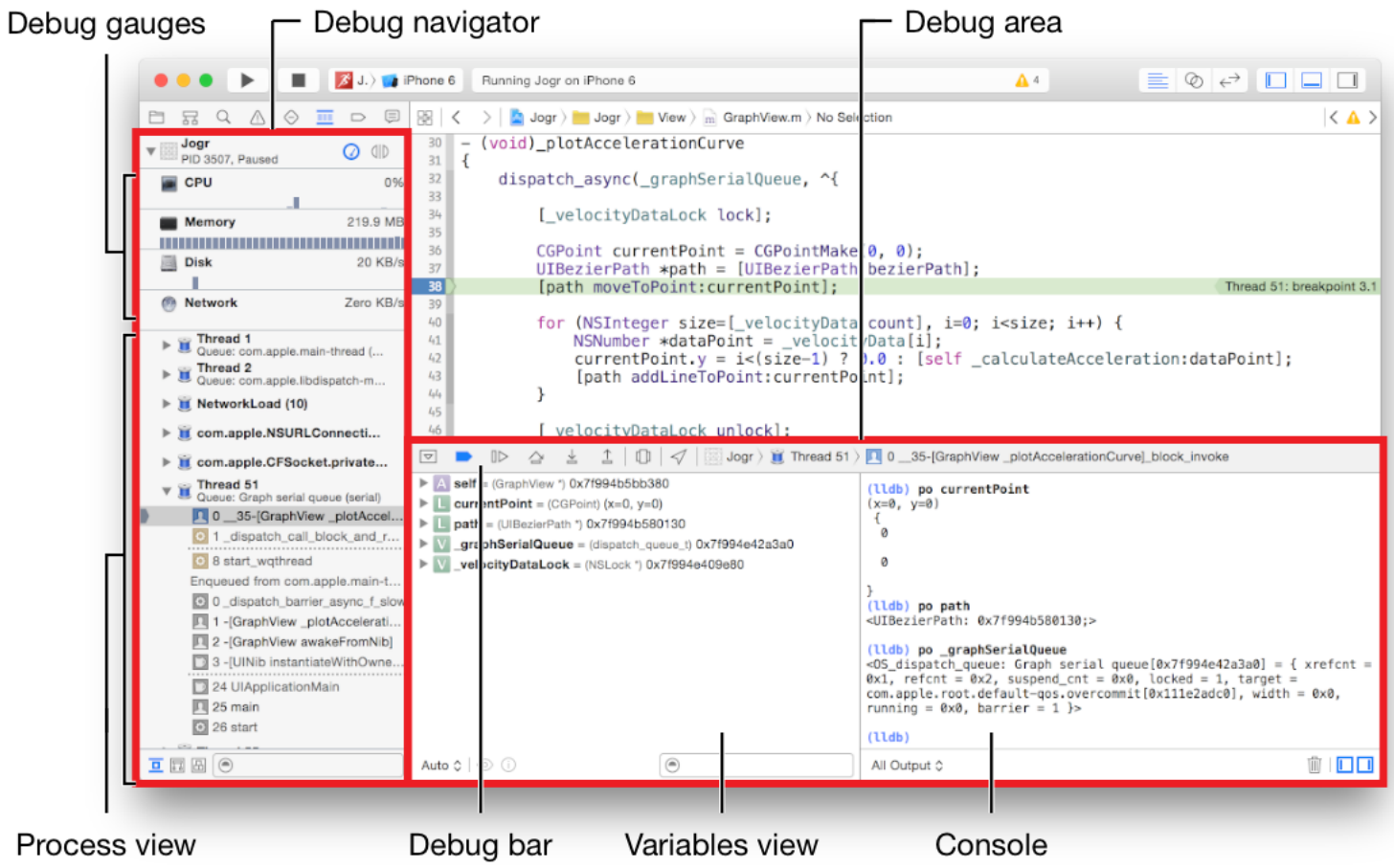
> **Note:** The Breakpoint activation button does not change the enabled/disabled setting of the breakpoints. All breakpoints remain in place and, when activated, are enabled or disabled as they were before being deactivated.



Breakpoint activation button

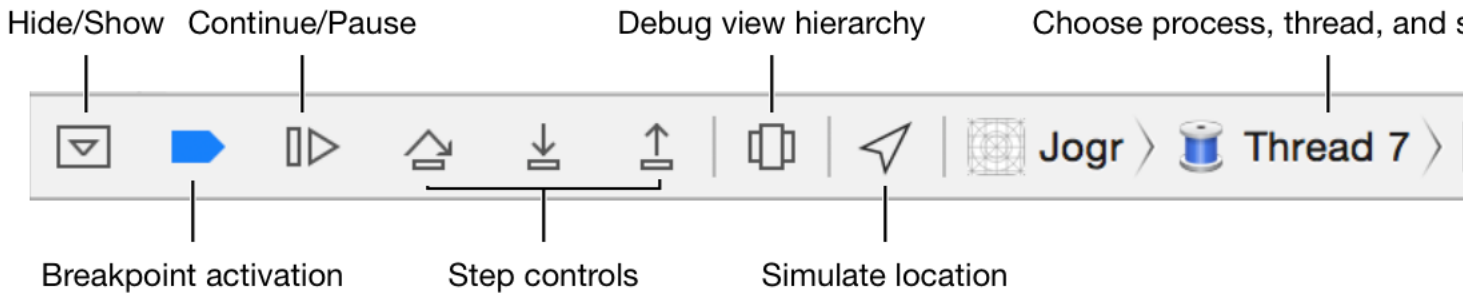
Controls in the Debugger

The debug navigator helps you examine the app's control flow. When the app is paused by a breakpoint, the debug gauges display the last values generated by the app's execution. Below the debug gauges is the process view, which can be set to show your app's run state organized by threads or queues.



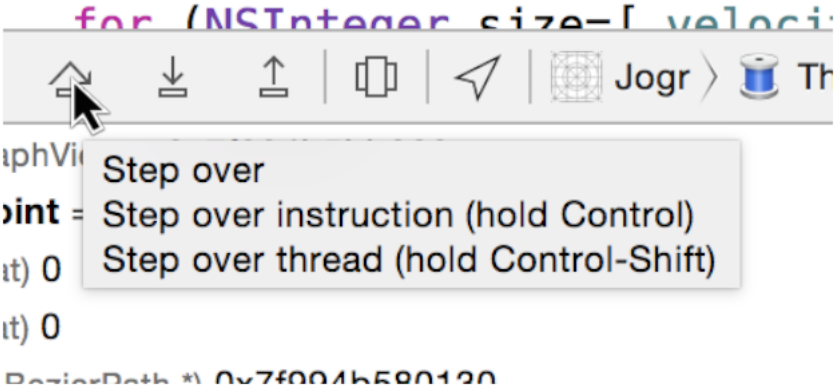
The debug area at the bottom of the Xcode main window contains three panes: The debug bar, the variables view, and the console.

Stepping Controls in the Debug Bar



The debug bar contains controls that hide and show the debug area, the breakpoint activation button, and the continue/pause button. When the app has been paused by a breakpoint, click the Continue buttion to resume the app.

Next to the continue/pause button is a set of three buttons that allow stepping your app. Their basic operation is to step over a source instruction, step into an instruction, and step out of an instruction. If you hold the mouse over these buttons, alternate stepping modes appear, accessed by using modifier keys when clicking:



You set a breakpoint before the line of code you suspect is causing the problem. This pauses the app so you can inspect the variables. Then you step the app to see how the variable states change, stepping over, into, and out of lines of code as needed. Once you've completed the inspection, you click the continue/pause button to resume the app's normal operation.

> **Note:** The debug bar's stepping controls are also available in the Debug menu and have keyboard equivalents to make their use efficient and easy. The default key bindings can be customized using Xcode Preferences > Key Bindings.

The other buttons in the debug bar are more specialized in nature. The button to debug view hierarchies is discussed in Debugging View Hierarchies (page 65). To the right of these two buttons, the process/thread/stack frame jump bar enables you to go directly to stack frames in the paused app, a useful alternative way to move around that works in conjunction with the debug navigator process view display.

In the next section you'll see how the debug area's variable view allows you to inspect variables, but often the best way to debug a problem is to see as much of your source surrounding the problem area as possible. That's when you use the Hide/Show button to hide the debug area, leaving just the debug bar available for stepping in the source editor. You can open the debug area again by pressing the Hide/Show button again.

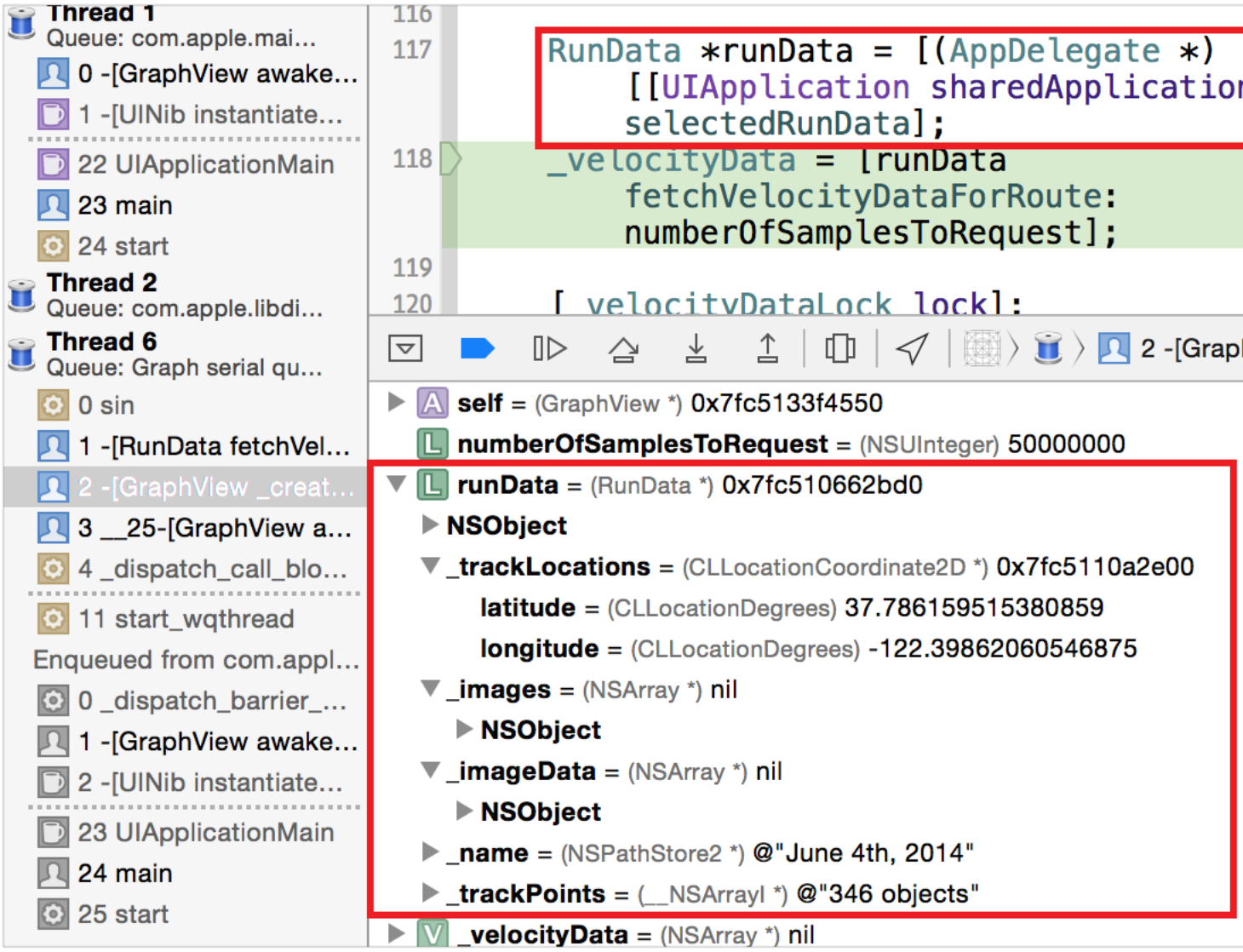
Inspecting Variables with Variables View

You use the variables view in the debug area to inspect variables and determine their state.

As you can see from the figure, the variable view pane lists each variable available in the context of the breakpoint where the app is paused. Next to the disclosure triangles is a set of icons that indicate the variable's kind, then the variable name. The name is followed by the variable summary representing the current value. As you use the stepping controls in the debug bar to execute your app line by line, you can see the variable values that result from each operation in the source.

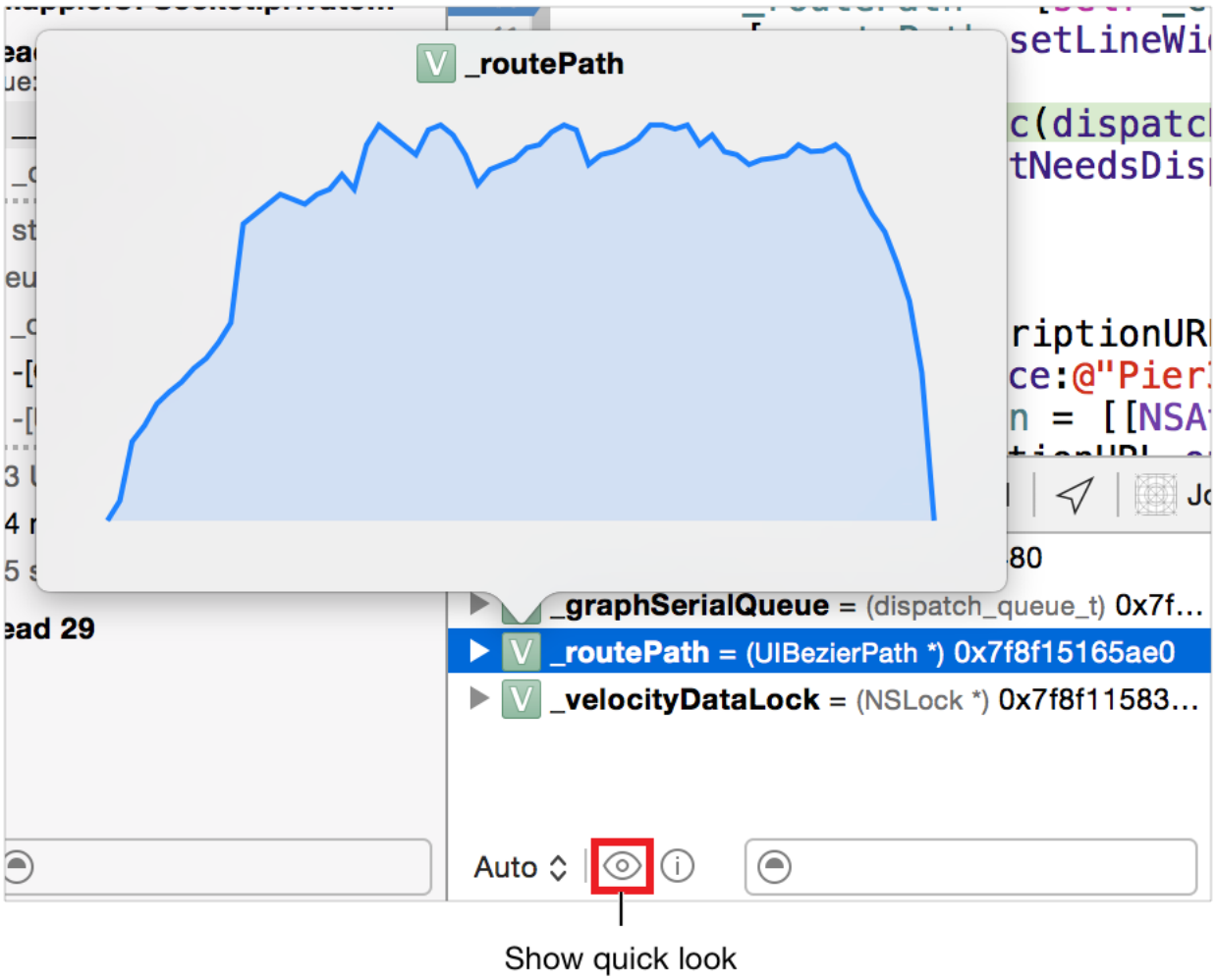
At the lower-left corner of the variables view pane is a filter popup menu, set by default to Auto. This setting restricts the variables in the view to those normally considered interesting for most debugging, but you can set the variables view to show all variables (including globals, statics, and registers as well) or just local variables.

For complex variable structures and objects, you can drill down into all the components of the variable by clicking the disclosure triangle. This allows you to see every detail of the variable's components at once, and how they might change as you step your app.



Next to the filter popup menu are the Quick Look button,  , and the Print Description button,  . You select a variable in the list and click these buttons to display them in popup windows. The Quick Look button produces a graphical rendering with dimension and/or value information, depending upon the type of the

selected variable. The quick look graphical rendering is particularly useful when you are trying to see a complex object and how it is being drawn or rendered. Below is an example of the Quick Look button being used to display a `UIBezierPath` object in the variables view:



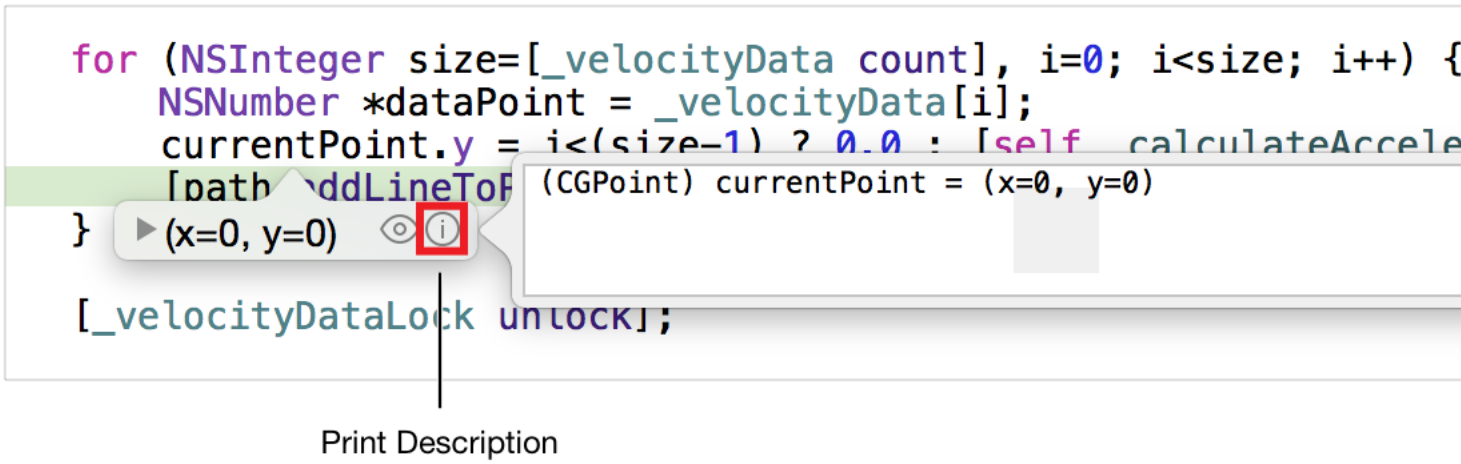
Note: Quick Look displays both system provided object types and your own object types. See Quick Look Data Types (page 83) for more information.

The Print Description button presents textual information about the variable, equivalent to using the `po` command in the console. When you click the Print Description button, it sends the `po` output to the console as well as shows it in the popover window.

For many developers, the best way to work is to see as much of the source as possible when paused in the debugger. You can hide the debug area and work directly in the source editor, using Quick Look and Print Description buttons to inspect the variables. Hold the mouse over a variable in the source editor for a moment

and a popup window appears with the same information that the variable would be displayed with in variables view. You can open the disclosure triangle, again just as in variables view, to see the components of a complex variable, and so forth. The popup window also has the Quick Look and Print Description buttons available right there as well: clicking them has the same effect as clicking them in variables view.

For example, this figure shows a `CGPoint` variable being inspected with the Print Description button:



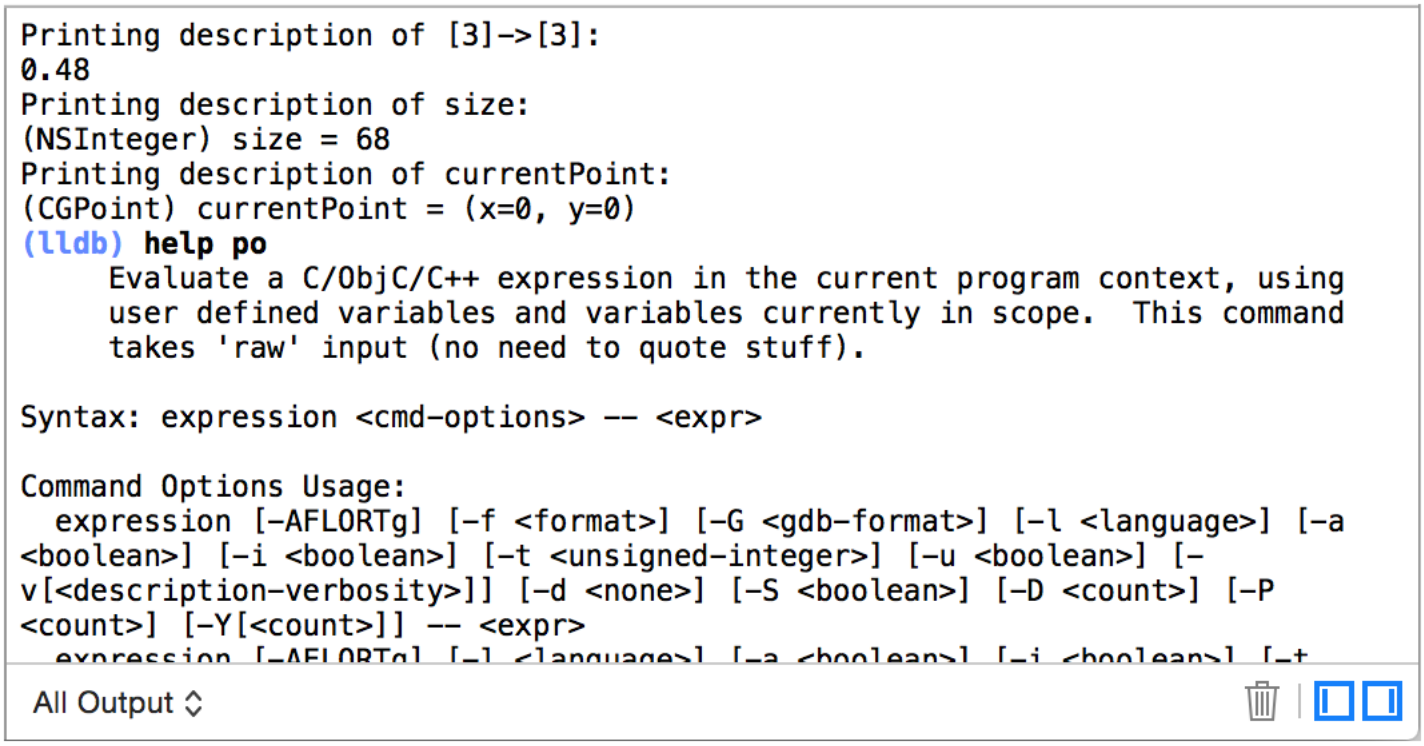
As you might expect, the Print Description button output displays in both a popup window and in the console.

Most of the time, you use the variables view, quick look, and print description tools to inspect the variable values when you're debugging, making sure that they are as expected. However, there are times when you might want to try changing a variable value on the fly to see if that corrects a problem without having to go through a full build cycle. This is easily done in the source editor using the popup window on a variable: hold the mouse over a variable until the popup window appears, drill down into the variable using the disclosure triangle to the variable you want, then double-click on the value. Remember that changing the value of a variable on the fly is often a bit risky as you're changing the state of a running app, which might have side effects, but for some kinds of debugging it can provide the information you're looking for.

Understanding the Console

The Xcode debugger area includes the console view. The console view is a Terminal-like command line environment that records output from the app and the debugger while the debugging session is running. You use the console to collect printed variable values as you run, break, and inspect variables. The values in the console view persist throughout a debugging session, you can see a listing of outputs created there during your entire session. When you finish a session and re-run the app for another session, the console is cleared so it always starts empty at the beginning of a debugging session. However, you can review the console's contents from previous debugging sessions by going to the Report navigator and checking the contents of the debug sessions stored there.

The Xcode debugger uses LLDB, a low-level debugging engine, to perform its functions. The console enables you to use the LLDB command-line interface directly, you can use any LLDB commands in the console view to supplement or extend the Xcode debugging experience.



LLDB is a powerful debugging environment with many capabilities. Learn the basics of using LLDB by reading *LLDB Quick Start Guide*.

Examining the Backtrace in the Debug Navigator

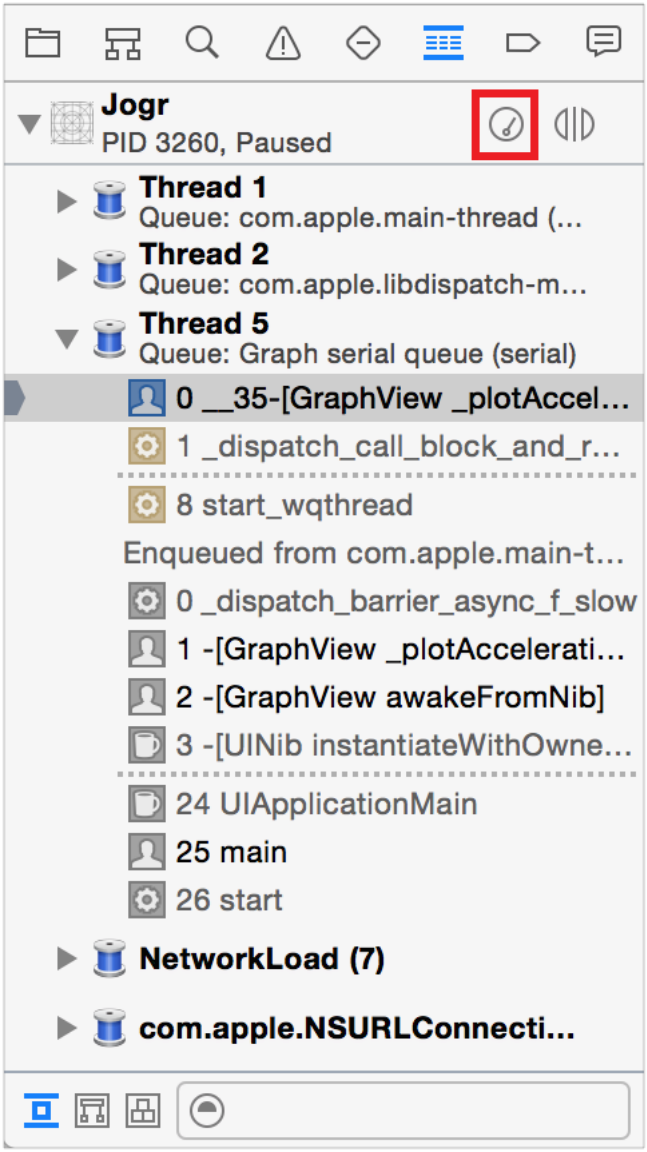
To recap the situation described earlier in this chapter, you launched your app using the Run button. You tried to invoke the new code that you added and nothing happened. Now you've added a breakpoint where the new code should be started from and tried to invoke the code again. The Xcode debugger has paused your app at that location and populated the variables view in the debug area and the debug navigator with the current state of the app.

You'll likely first want to take a look at the variables and their values. The variables view, quick look, and print description tools give you that first look into the state of the app. The next question that needs to be addressed is "how did the code get here?" To answer this question, you use the debug navigator process view pane.

When the app stops at a breakpoint, the debugger "unwinds" the program flow and presents that to you in the debug navigator process view pane. Some definitions will make this clear:

- A *stack frame* is an instance of an invocation of a method.
- Unwinding means "looks at the method that contains your breakpoint and follows the stack frame pointers back to the call site." The debugger does this until it reaches the beginning of the thread.
- The sequence of stack frames unwound in this fashion is known as the *backtrace*.

By default this view is organized by threads with the current stack frame highlighted, corresponding to the source in the source editor where the program counter is positioned at the breakpoint. (The debug gauges have been hidden in this example of the debug navigator using the highlighted button, allowing you to concentrate on the process view showing the backtrace.)



The backtrace allows you to understand the control flow in your app. It is displayed in the debug navigator's process view pane with each stack frame identified by an icon. The icons tell you where in the compiled code the stack frame is from. There are icons for user code, the `Foundation` framework, `AppKit` or `UIKit` frameworks, the `Graphics` frameworks, and so forth. The debugger retrieves and splices in this full history of execution, even including stack frames that are no longer in memory.

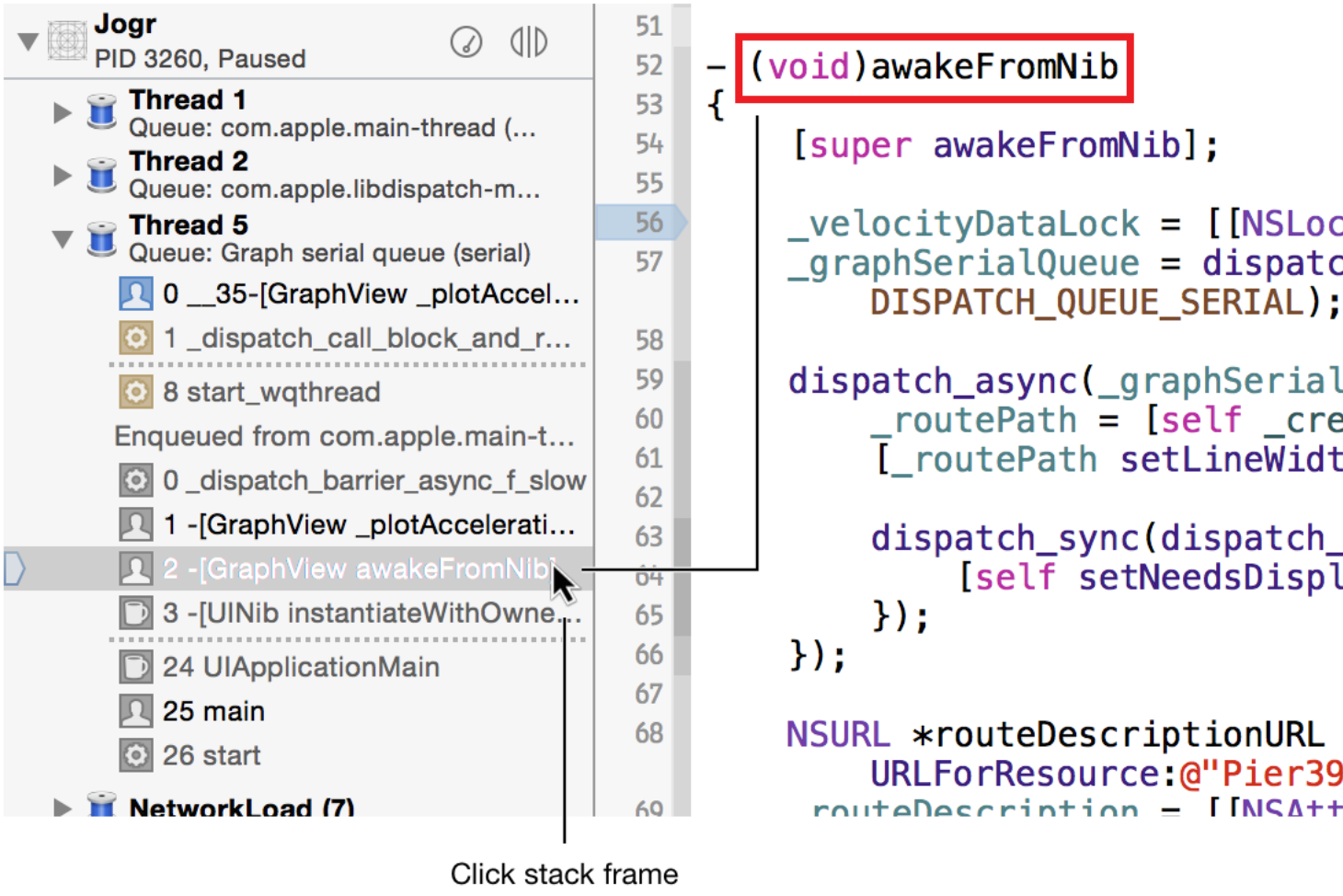
With this view of the backtrace, you can see how your app came to its current state at the breakpoint pause. Looking at the illustration above:

1. The app is paused in your source code at `_plotAccel`, which is executing in Thread 5.
2. This code has been called by a system function that is part of a dispatched asynchronous callback.
3. Some system stack frames are elided from the view—the dotted line represents them. You can show these stack frames by deselecting the left-most button in the filter bar at the bottom of the debug navigator.
4. The stack frame below the dotted line shows the start of the worker thread that called the code.
5. Below that, you can see where the block was asynchronously enqueued in the recorded backtrace.

   The recorded backtrace shows that this block was enqueued during `awakeFromNib`, possibly near the start of the app.

As you click on the stack frames in the backtrace, the source editor jumps to that point in the source or in the decompiled binary executable. For stack frames of your code, you can see your source code. If the frame is in memory, you can see variable values; if the stack frame is no longer in memory—that is, it's a recorded stack frame—the variable values are no longer available.

Note: In the debug navigator process view, if the icon next to a stack frame is colored, the stack frame is in memory. Gray icons next to a stack frame indicate that the stack frame is no longer in memory, and the debug navigator has retrieved the stack frame by analyzing the recorded call history; the debugger cannot re-create the variable values from the recorded call history.



Using the debug navigator and the backtrace, you can see whether the app's code has moved along the expected path to the breakpoint or whether it diverged from where you expected it to be. By jumping to user code in the backtrace when you have a problem, you can set new breakpoints that stop the app at earlier points and see where it went awry by examining the variables along the way.

If your new code doesn't run or display because a conditional was not fulfilled correctly, find the right point in the backtrace, set a breakpoint that will illuminate the problem, click Run/Pause to resume normal operation, and then perform the action that should run the new code again. This time stop at the earlier point in the program flow and inspect variables, then step the app until you find the cause of the problem, examining the variables along the way.

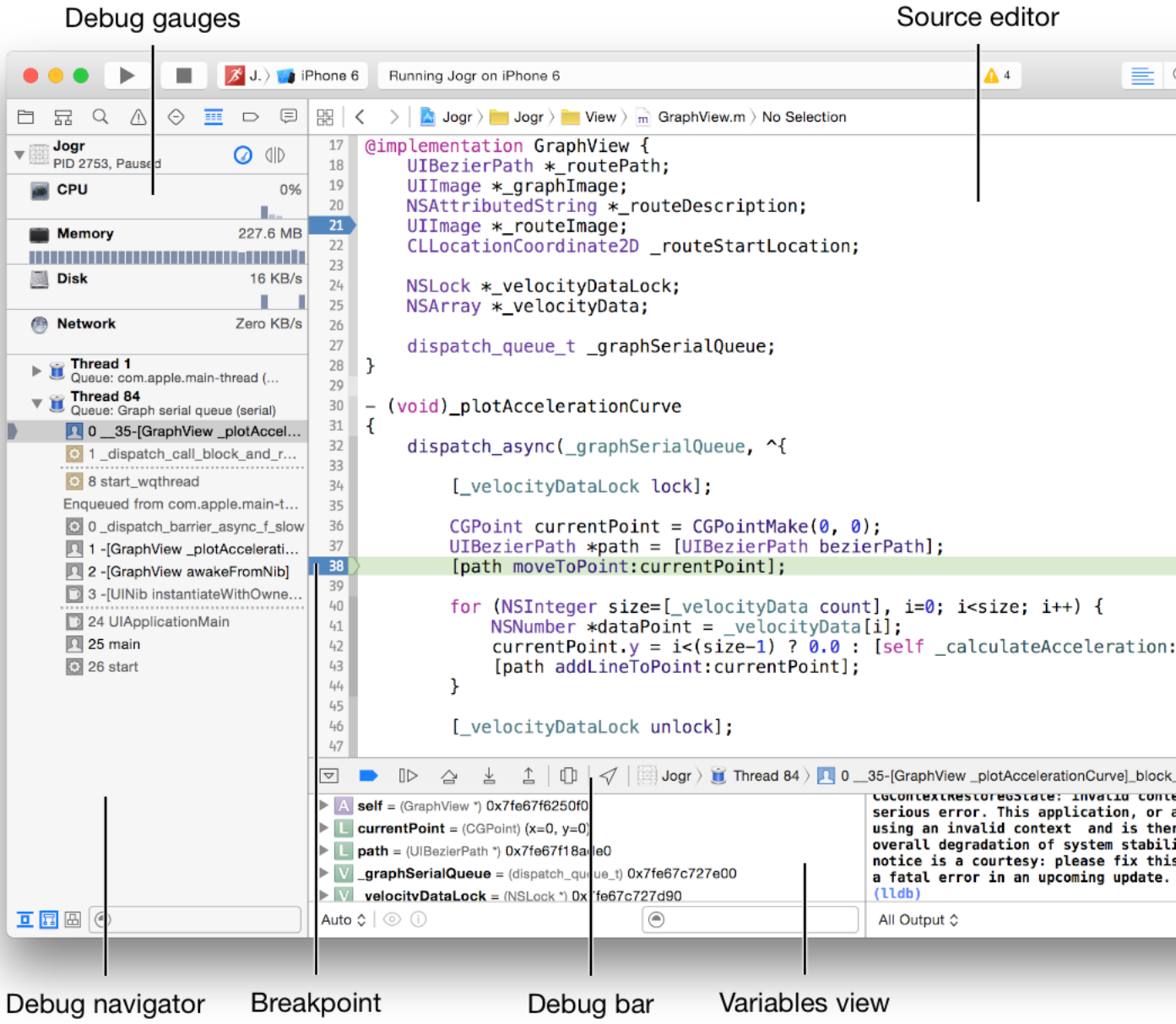
Cycling Through the Debugging Process

Debugging is an iterative effort. You discover a problem, you set breakpoints to help you locate where in the source it is happening, you inspect the backtrace and variables to assess their state and the cause of the problem, and you devise a solution or fix. With a potential solution in hand, you make a change to the source and re-run the app to see if the problem is solved. If testing shows that the problem persists, you repeat this cycle until you've put the right fix in place.

When a problem is solved, you can disable or delete the breakpoints you used in its investigation. You don't want them to get in the way as you move on to the next problem.

Debugging Tools

The Xcode debugging tools are integrated throughout the Xcode main window but are primarily located in the Debug area, the debug navigator, the breakpoint navigator, and the source editor. The debugging UI is dynamic; it reconfigures as you build and run your application. To customize how Xcode displays portions of the UI, choose `Xcode Preferences > Behaviors`.



The illustration above shows the default layout of the Xcode debugger with the app paused at a breakpoint.

General Notes

Here are a few notes about debugging in general and some basic information about Xcode as you begin to read this chapter.

The Five Parts of Debugging and the Debugging Tools

There are five parts to the debugging workflow:

- **Discover.** Identify a problem.
- **Locate.** Determine where in the code the problem is happening.
- **Inspect.** Examine the control flow and data structures of the running code to find the cause of the problem.
- **Fix.** Apply your insight into the cause of the problem to devise a solution and edit the code to suit.
- **Confirm.** After editing, run the app and check it with the debugger to be sure the fix does the job.

The division of labor in these five parts of debugging are not necessarily reflected in the specifics of the debugging tools, although some tools are more pointed at discovery (for instance, the debug gauges), some are particularly useful for dealing with locations of interest in your code (breakpoints), and others are more specific to inspection (the debug area's variables view and the debug navigator's process view).

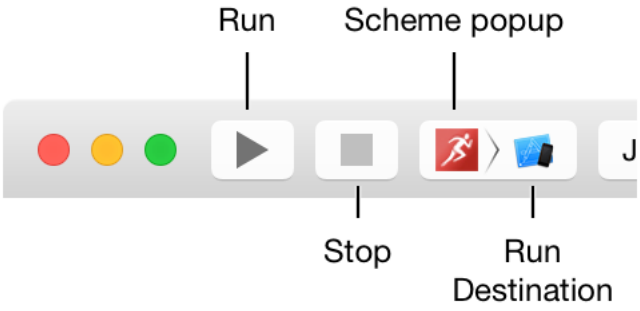
Think of the debugging tools as addressing the five parts of debugging more as a function of how you use them. For instance, you use the Quick Look feature to inspect the state of a graphical object as you work through a problem, but you can also think of it as being a discovery tool, using it to see how a complex graphic is "assembled" as you iterate through a set of drawing calls. In other words, how you put a tool to work for you often determines which part of the debugging effort it addresses, and it is the combination of what the tool does, what data using it uncovers, and your creative insight into the context of the situation that delivers success to your debugging efforts.

LLDB and the Xcode Debugger

The Xcode debugger uses services and functions provided by LLDB, the underlying command-line debugger that is part of the LLVM compiler development suite. LLDB is tightly integrated with the compiler, which enables it to provide the Xcode debugger with deep capabilities in a user-friendly environment. The Xcode debugger provides all the functionality needed for most debugging situations, but a little familiarity with LLDB can be helpful. For a basic introduction to LLDB, see *LLDB Quick Start Guide*.

Xcode Toolbar Controls

The Xcode toolbar contains the most basic controls you need to start debugging.

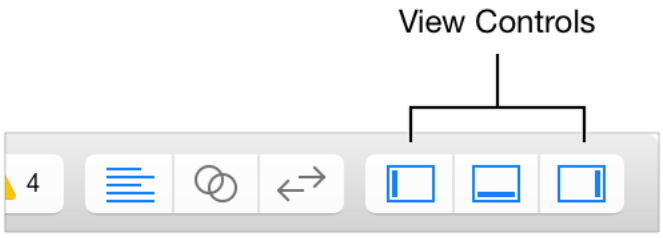


Run button. Click to build and run. Click and hold to select other Product actions (Run, Test, Profile, Analyze) from a menu. Using the Shift key modifies the menu to a "Build for" operation. The default operation is to build and run, which starts the debugger as well.

Stop button. Click to stop the current running task or app.

Scheme menu. Xcode schemes control the build process based on the settings they contain for the Product action you choose and the target build settings. For most uses, the defaults created with a project suffice, but there are useful debugging options configurable in the scheme editor's Run action. A look at these options is provided in Debugging Options in the Scheme Editor (page 59).

Run destination. Choose from this menu the OS X or iOS device (or simulator) the build and run operation will execute on.



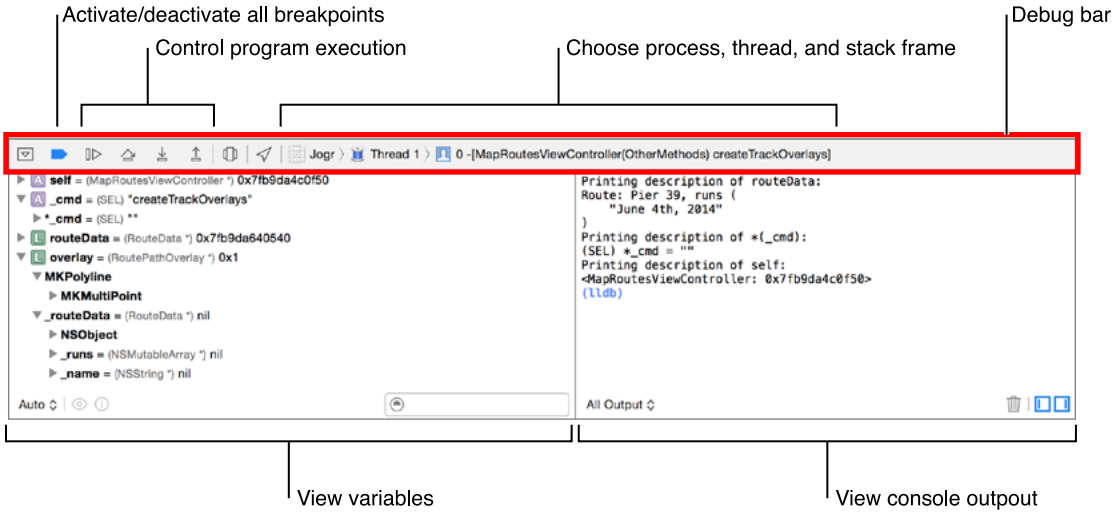
View controls. These three buttons control the visibility of the navigator, debug, and utility areas in the Xcode main window. They have matching commands with keyboard shortcuts in the View menu.

Xcode Debug and Product Menus

In addition to the areas and controls located in the Xcode main window panels and toolbar, the Xcode main menu bar includes the Product and Debug menus. These menus provides a convenient, configurable keyboard mapping for many of the more common commands used during debugging sessions. The Debug menu provides convenient keyboard stepping commands for use when you pause your app and analyze what happens line by line. It also gives you access to some of the less-used debugging functions, such as the `Debug > Attach to Process` command, which allows you to target an app that is already running with the debugger. As you become proficient with the Xcode debugger, you'll likely make more use of these menu shortcuts.

Debug Area

The debug area opens at the bottom of the Xcode main window below the source editor when you build and run your app. It contains the debug bar, the variables view, and the console.



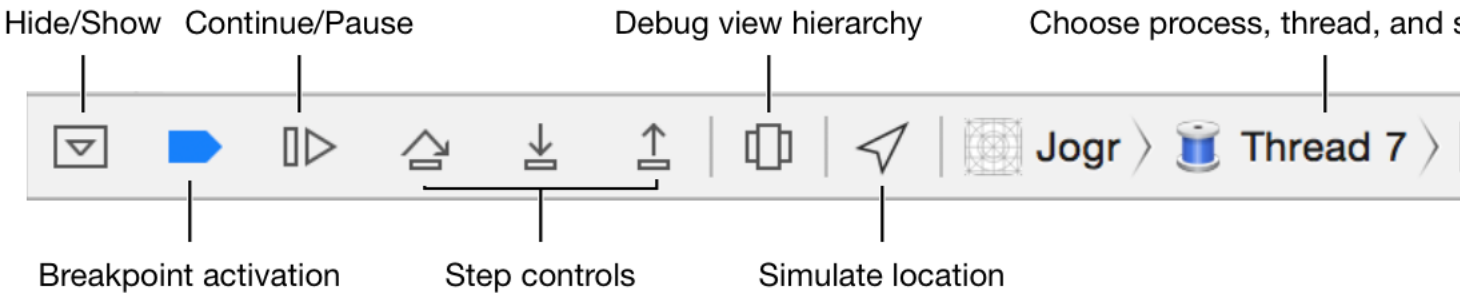
To hide or show the debug area click the center button ( ) in the view controls group located in the main window toolbar.

Debug Bar – Process Controls

When you build and run a program in Xcode, the debug bar appears at the bottom of the editor pane. The debug bar includes buttons to:

- Open or close the debug area
- Activate or deactivate all breakpoints
- Pause or resume execution of your code

- Step over; that is, execute the current line of code and, if the current line is a function or method, step out to the next line in the current file
- Step in; that is, execute the current line of code and, if the current line is a routine, jump to the first line of that routine
- Step out of a jumped-to routine; that is, complete the current routine and step to the next routine or back to the calling routine.



Debug area hide/show button. The action of this button differs from the view controls in the Xcode main window toolbar in that when you run a debugging session, it controls the view of both variable view and console panels but leaves the debug bar accessible. This is useful when you are working primarily in the source editor while debugging and want to maximize the amount of space you have to view your source code.

Breakpoint activation button. This button acts as a toggle to deactivate and activate all breakpoints in your app simultaneously. It's useful when you know you need to let the app run normally, without pausing at any breakpoints, for a while to reach a state where you can start debugging a problem.

Continue/Pause button. You can suspend the execution of your app by clicking the Pause button, which toggles between  to pause and to continue. More commonly, however, you set a breakpoint to pause your app at a planned location.

Step controls. When your app is paused, the currently executing line of code is highlighted in green. You can step through execution of your code using step over (), step into (), and step out (). Step over will execute the current line of code, including any methods. If the current line of code calls a method, step into starts execution at the current line, and then stops when it reaches the first line of the called method. Step out executes the rest of the current method or function.

The stepping controls have alternative operations for working with disassembly or threads. You use the Control and Control-Shift modifier keys to call these alternatives. Press Control to step by assembly language instruction instead of by statement (the step icons change to show a dot rather than a line under the arrow) or Control-Shift to step into or over the active thread only while holding other threads stopped (the step icons show a dashed rather than solid line below the arrow).

Debug view hierarchy button. Click to investigate the relationship of view objects both in a 3D rendering and in a hierarchical list in the debug navigator. See Debugging the View Hierarchy (page 58) for more information.

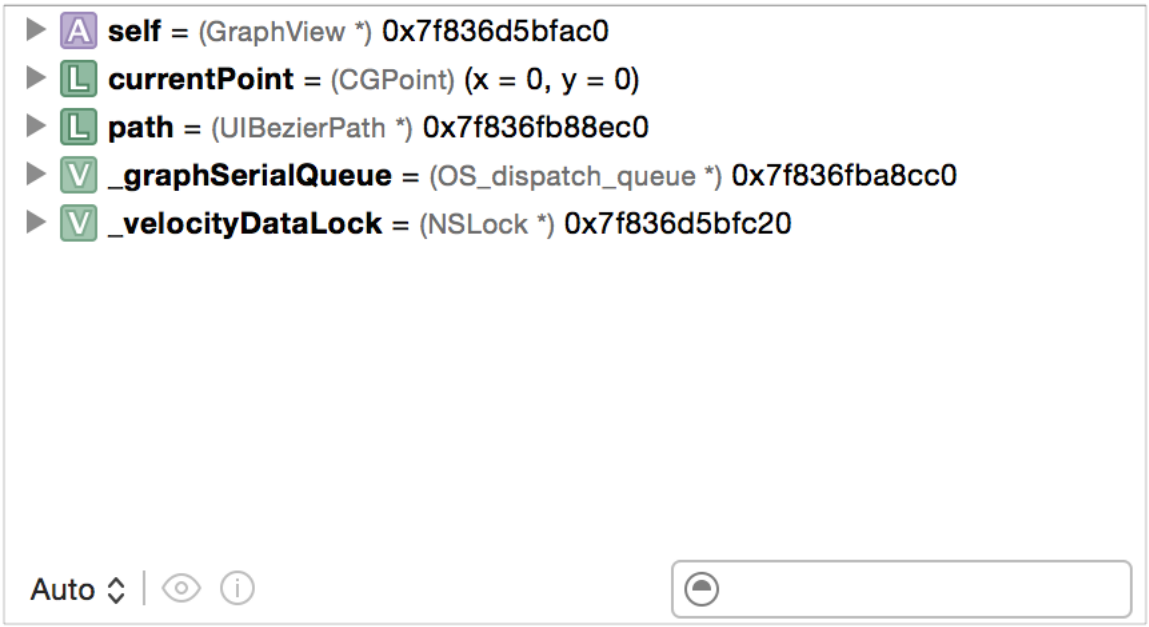
Simulate location button. Clicking this button presents a menu of locations to choose from. You choose one to tell a device or the simulator to report that location to your app as it runs.

Process/Thread/Stack frame jump bar. When your app is paused, this jump bar provides a convenient way to navigate through all processes and threads, and lets you jump to specific stack frames for the purpose of inspecting the program flow or setting breakpoints. Picking a stack frame from the jump bar will bring the source, or disassembly if the source file isn't available, into the source editor.

Variable View – Inspecting Variables

The variable view provides the primary way to inspect the state of variables when an application is paused.

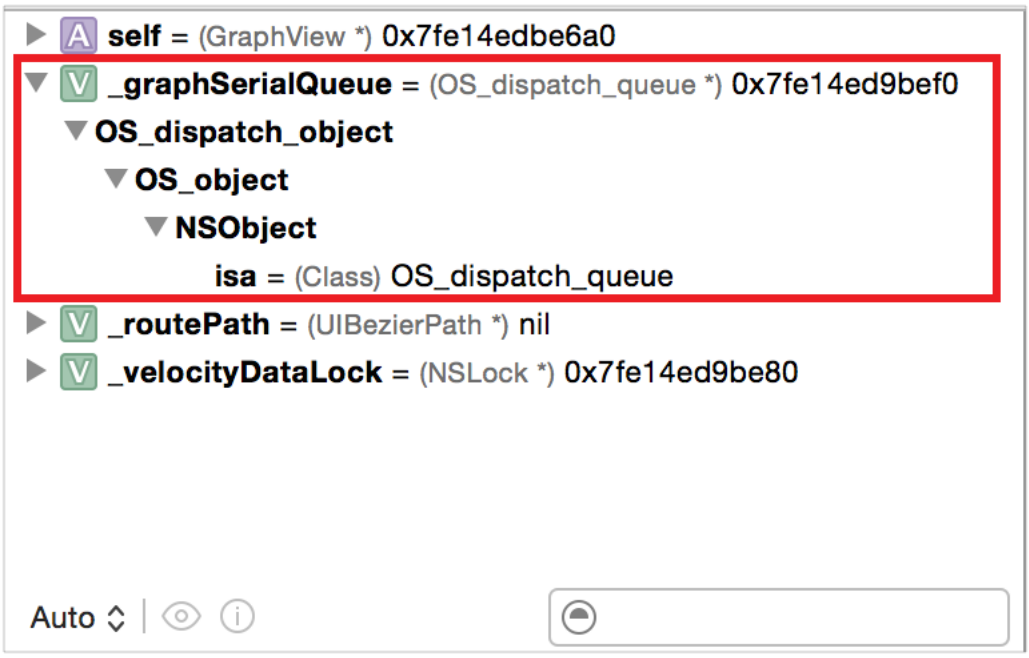
Variable list. The variable view lists each variable available in the context where the app is paused. Each variable at the top level takes up a row in the list. Disclosure triangles are at the far left of the row, followed by an identifying icon, the variable name, type, and value.



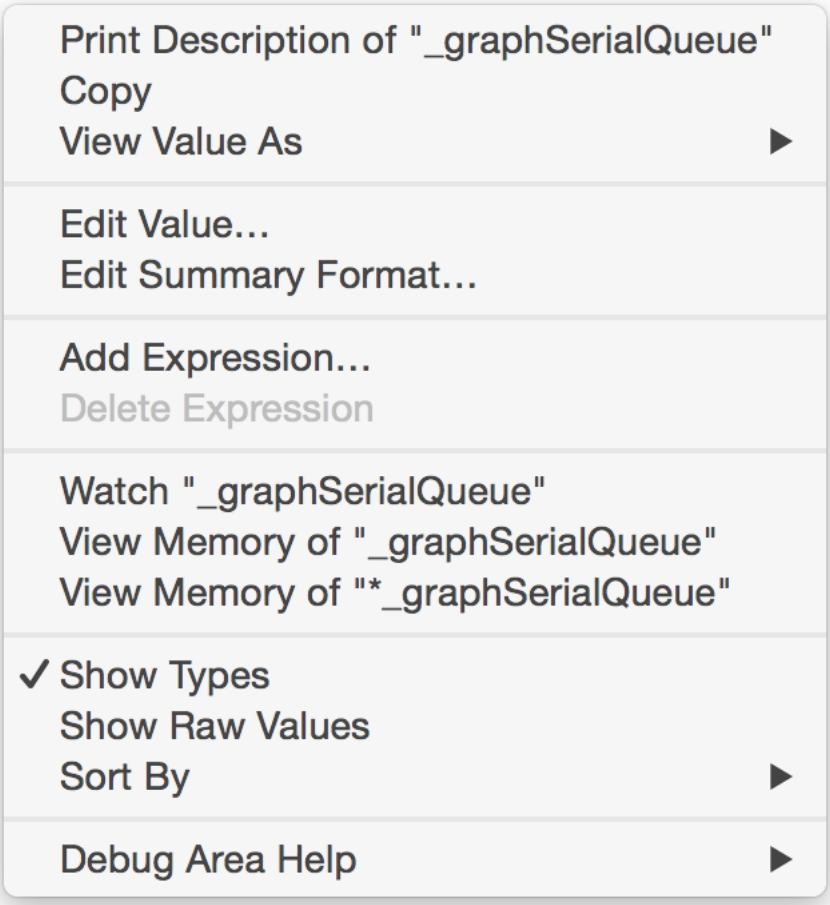
The icons used in the variable list allow easy recognition of the variable kind at a glance.

The variable name is followed by the variable value as generated by the data formatter for the variable type. As you use the stepping controls in the debug bar to execute your app line by line, you can see the variable values that result from each operation in the source.

Clicking the disclosure triangle on the left, you "open up" the variable to inspect its component parts and their values, as in the red boxed variable below.



Variable view menu. Right- or Control-click a variable in the list to display commands that act on the variable. Some of the commands duplicate other ways to obtain the same behavior.



Print Description of "{variable}". An equivalent to using `po` in the console or using the Print Description button in a tool tip.

Copy. Copies the selected variable's representation displayed in variable view, allowing you to paste it as text.

View Value As. You can use this command to cast a variable's type to another type and apply that data formatter in variable view for display. Clicking on the command presents a menu of standard types. The menu includes a Custom Type option, which presents a pop-up editor that you use to input a custom data format.

Edit Value. This command puts the selected variable's value into an edit field so you can type in a specific value. Alternatively, you can double-click the variable value in the list directly to achieve the same function.

Edit Summary Format. Presents an editor allowing you to change the data formatter by entering a new formatter representation for the variable.

Add Expression. Allows you to add an expression to the variable list for the debugger to evaluate and present a result in variable view. An option in the editor allows you to add the expression to the variable list in all stack frames.

Watch "{variable}". Creates a watchpoint that reports the value of a variable as your app runs. Watchpoints are managed in the breakpoint navigator.

View Memory of "{variable}". Using this command changes the display in the source editor to a hex editor display based on the address of `{variable}`.

View Memory of "*{variable}". Similar in function to the preceding, this command changes the display in the source editor to a hex editor display based on the address pointed to by `{variable}`.

> **Note:** For both of these View Memory modes, return to the source listing by pressing the back button, by choosing the stack frame in the debug bar's jump bar, by clicking on the stack frame in the debug navigator, or by clicking the breakpoint in the breakpoint navigator.

Show Types and **Show Raw Values.** Allow you to adjust how much information is presented for each variable in the variable view list.

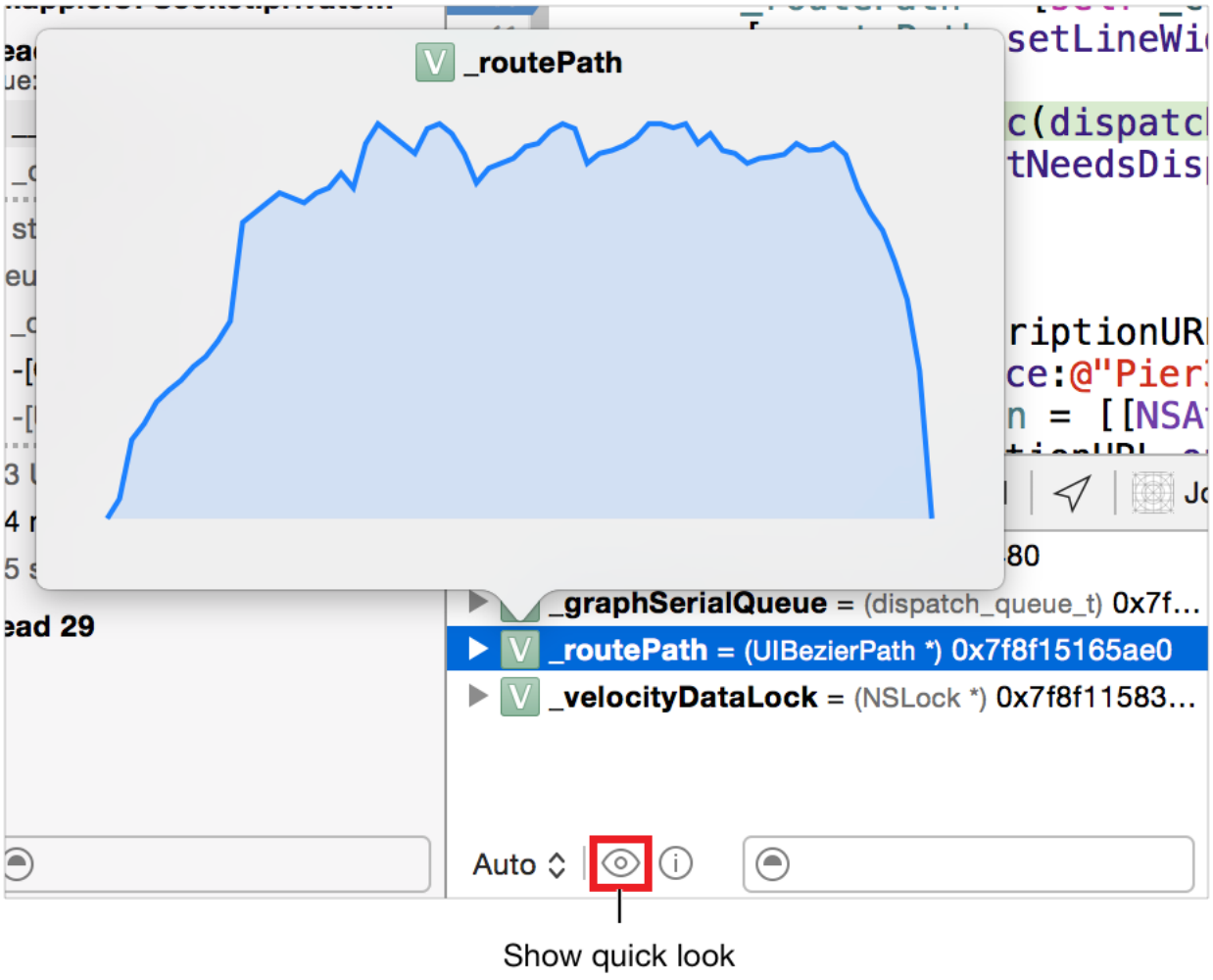
Sort By. Allows you to choose to show variables by their order of appearance in the source or by name, in ascending or descending order (when showing by name).

View control menu. The variable view can display both variables and registers. You specify which items to display using the pop-up menu in the lower-left corner. By default, it is set to Auto. The options are:

- **Auto** displays only the variables you're most likely to be interested in, given the current context.
- **Local** displays local variables.
- **All** displays all variables and registers.

Filter bar. Use the search field to filter the items displayed in the variables pane.

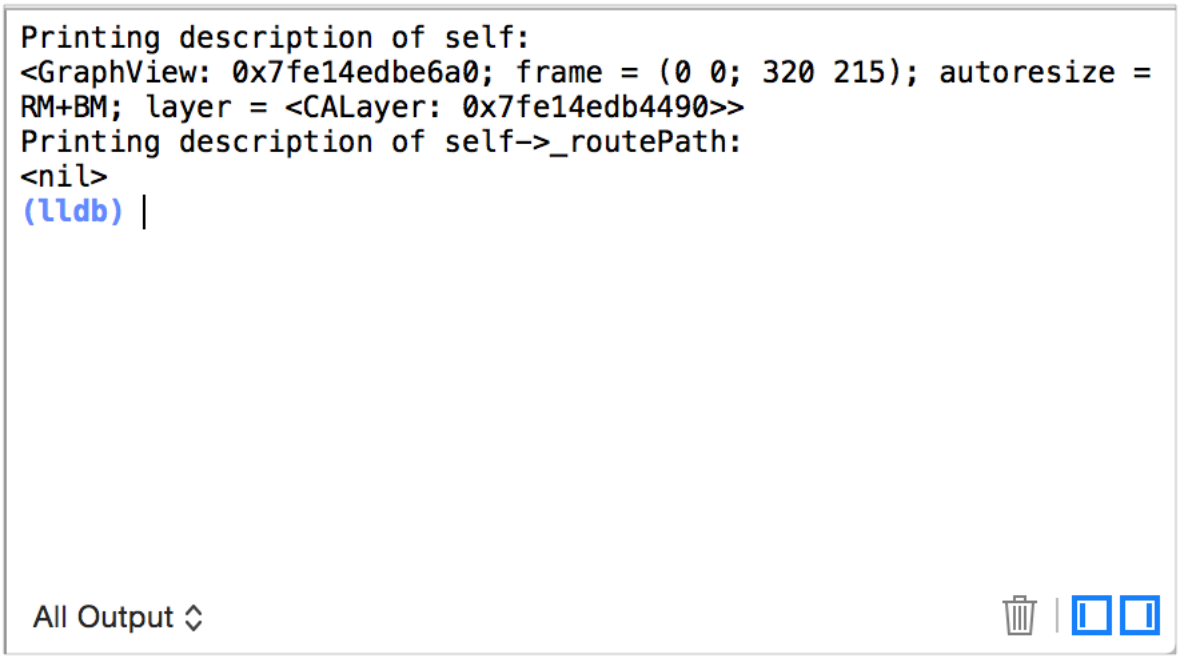
Quick Look button. Use Quick Look to produce a visual rendering depending upon the type of the selected variable. The quick look graphical rendering is particularly useful when you are trying to see a complex object and how it is being drawn or rendered. For example, this illustration shows a `UIBezierPath` object in the variables view:



Print Description button. Select a variable in the list and click the Print Description button, ⓘ, to print textual information about the variable to the console pane. It is equivalent to using the LLDB `po` command.

Console — Command Line Input/Output

The debug area console is a command line environment that you can use to interact with an app or with LLDB.



Command line environment. When the app is running during a debugging session, it can read from the console using `stdin` and output to the console using `stdout` and `stderr`. Logging with `NSLog()` is sent to the console as well. When an app is paused, input typed in the console is interpreted by LLDB; output from LLDB is sent to the console as well.

> **Note:** Running an app within the Xcode environment but without attaching to the debugger is also possible, in which case the console is still accessible to an app as `stdin`, `stdout`, and `stderr`.

Console output persists throughout a debugging session, you can see a listing of outputs created for the entire session. When you finish a session and re-run the app, the console is cleared. You can review the console contents from previous debugging sessions by going to the Report navigator and checking the contents of the debug sessions stored there.

Output control. You specify the type of output the console displays with the pop-up menu in the lower-left corner of the console pane:

- **All Output** displays target and debugger output.
- **Debugger Output** displays debugger output only.

- **Target Output** displays target output only.

All output is the console default.

Clear console button. The Trash button at the lower right allows you to clear the console at any time in your session. (The full console output is logged to the Reports navigator.)

Debug area view controls. Click the buttons at the lower right of the console to control the display of the debug area. They allow you to hide or show either the variable view or the console.

Breakpoints and the Breakpoint Navigator

Breakpoints

A breakpoint is a mechanism to pause an app during execution at a predetermined location in order to inspect the state of variables and the app's control flow. There are several different kinds of breakpoints, and they can be edited to add conditions, actions, and scripts. Although you can use the debugger to pause an app at any time, it's helpful to set breakpoints before and even while your app is running so that you can pause it at known points where you have determined problems might be occuring.

Kinds of breakpoints. The most commonly used breakpoint is file and line dependent: After you create and enable a file and line breakpoint, your app pauses every time it encouters that location in the code. You can set several options for a breakpoint, such as a condition, the number of times to pass the breakpoint before it's triggered, or an action to perform when the breakpoint is triggered. Conditions and actions are the most commonly used breakpoint options.

In addition to file and line breakpoints, you can create exception breakpoints, which are triggered when a specific type of exception is thrown or caught, and symbolic breakpoints, which are triggered when a specific method or function is called.

A test failure breakpoint is a specialized type of breakpoint used in debugging Xcode tests; see *Testing with Xcode* for details on its creation and use.

An OpenGL ES error breakpoint is a specialized type of breakpoint used in the Open GL ES debugging tools, it is a derivative of a symbolic breakpoint. See *OpenGL ES Programming Guide for iOS* for details on its creation and use.

You can also create watchpoints. Watchpoints are breakpoints that don't stop execution, they report the state of the variable they're configured for each time that variable is used. Set a watchpoint by Control-clicking the variable in the debug area variable view and choosing "Watch `{variable name}`" from the popup menu. There are a limited number of watchpoints available depending upon the hardware the app is running on.

Adding and enabling breakpoints. The most commonly used file and line breakpoints for methods and functions are created in the source editor; see Source Editor (page 48) for details. Exception and symbolic breakpoints are created with the Add (+) button at the bottom of the breakpoint navigator by choosing Add Exception Breakpoint or Add Symbolic Breakpoint from the pop-up menu. An editor is displayed that allows you to set the parameters for these types of breakpoints.

Editing breakpoints. To set breakpoint options, Control-click the breakpoint for which you want to set options, either in the breakpoint navigator or the source editor, then choose Edit Breakpoint from the shortcut menu. The Condition field lets you specify an execute condition for the breakpoint. When you specify a condition, the breakpoint is triggered only if the condition (for example, `i == 24`) is true. You can use any variables that are in the current scope for that breakpoint.

> **Function return types in expressions:** Expressions used in the breakpoint editor, just like expressions used with the LLDB expression parser, need to be the correct return type before the expression parser can evaluate them. For Objective-C, debug information often supplies this information automatically; in other cases Xcode can obtain that information from the Objective-C runtime for you. Otherwise you need to cast the function return type in the expression.
>
> For Swift, since you can overload function return types, the return type is a part of the mangled name and is generally available even without debug information, as long as the Swift modules are available.
>
> A good strategy to use when constructing conditional expressions to use is to first stop the app at the breakpoint unconditionally and then try your conditional expression in LLDB using the console. If it works, you can put it into the condition and you're set. If your attempt doesn't work, the compiler diagnostics are visible—you use them to fix up the expression for use.

Breakpoint behavior. What Xcode does when a breakpoint is encountered and your app is paused can be configured by the settings in Xcode Preferences > Behaviors using the Running > Pauses options.

Breakpoint scope. Breakpoints have scope; that is, they have a context in which they are defined and operate. By default, a new breakpoint is local to the workspace you have open. In this case, if you add the project containing that breakpoint to another workspace, the breakpoint is not copied to the new workspace.

You can assign breakpoints to two other scopes:

- A breakpoint with User scope appears in all of your projects and workspaces, wherever that source file is used.
- A breakpoint set to a specific project can be seen whenever you open that project, regardless of which workspace the project is in.

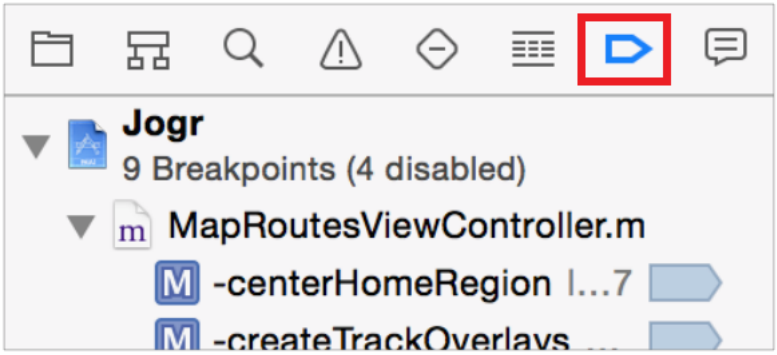
> **Note:** The default scope changes to the project if the project is not part of a workspace.

Breakpoints can be shared with other users. To share a breakpoint with other users, in the breakpoint navigator select the breakpoint you want to share, Control-click the breakpoint, and choose Share Breakpoint from the shortcut menu. Xcode moves shared breakpoints into their own category in the breakpoint navigator; all users with access to the project will be able to use them.

To change the scope of a breakpoint, use the breakpoint navigator. Control-click the breakpoint and choose the scope from the Move Breakpoint To menu item. The scopes of breakpoints are shown in the breakpoint navigator.

Breakpoint Navigator

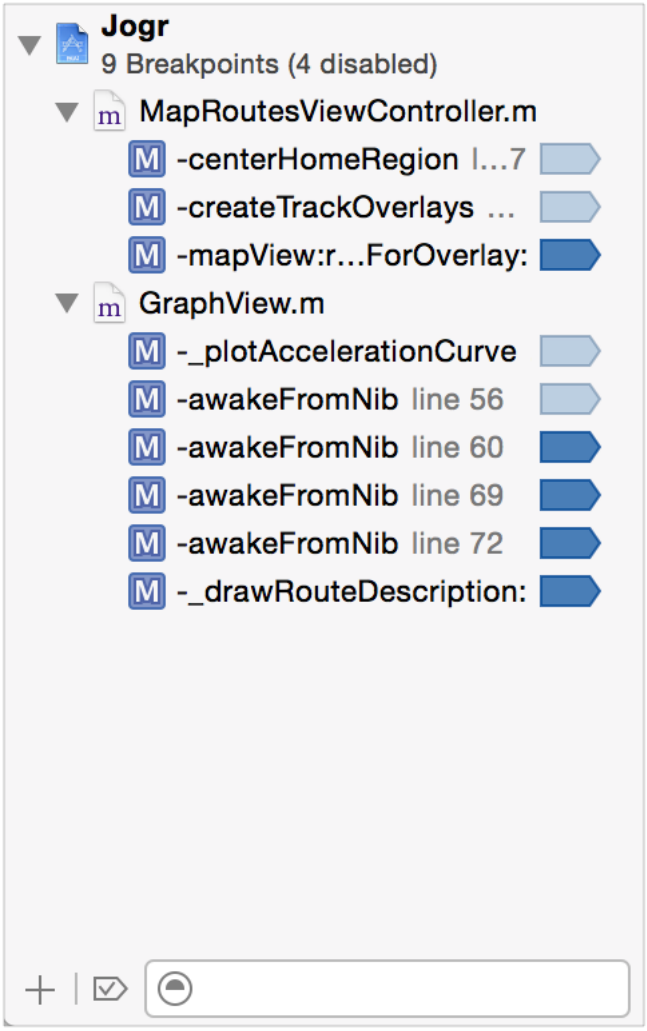
The breakpoint navigator helps you manage the breakpoints in your projects and workspaces.



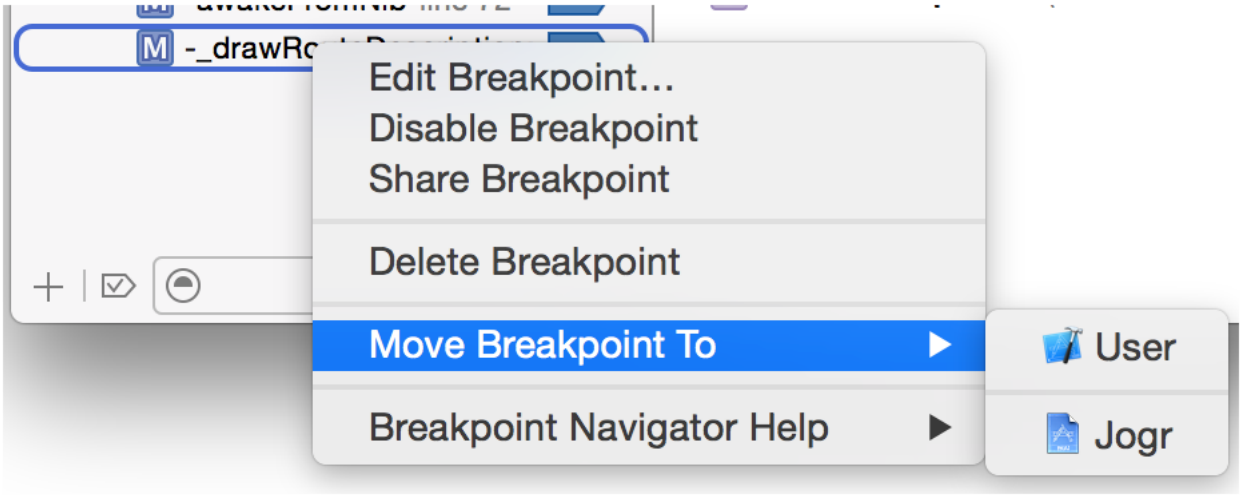
Open the breakpoint navigator by clicking the breakpoint-styled button in the Xcode navigators panel.

Hierarchical list organization. Breakpoints are organized in a hierarchy by scope, then by containing file. The list includes the symbol name of the function or method that a breakpoint is contained in as well as the line number in the source file.

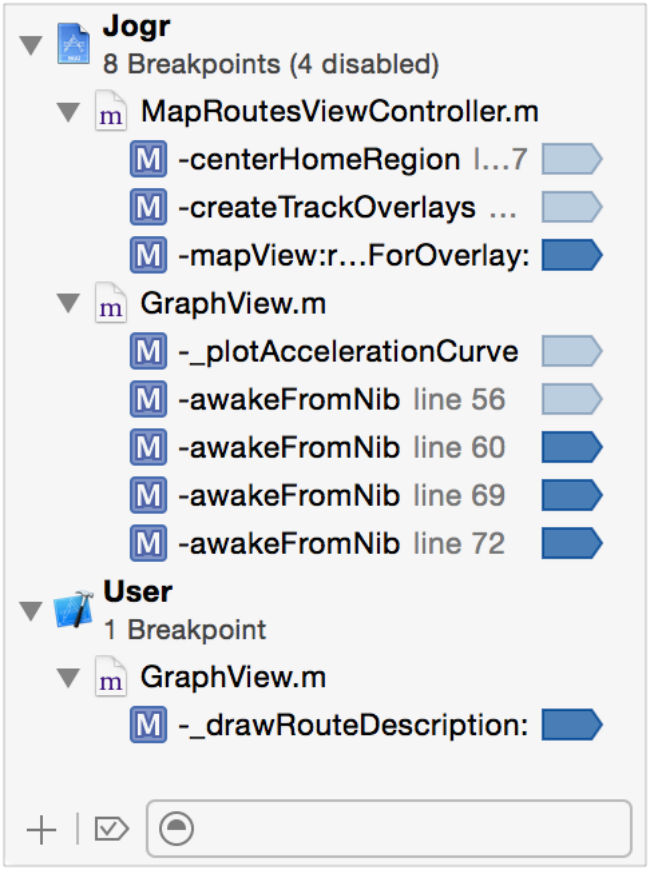
A breakpoint indicator symbol to the right of a breakpoint location—the same as the breakpoint indicators used in the source editor—is either dark or light blue to show whether the breakpoint is enable or disabled, respectively. This example shows nine breakpoints defined in two files:



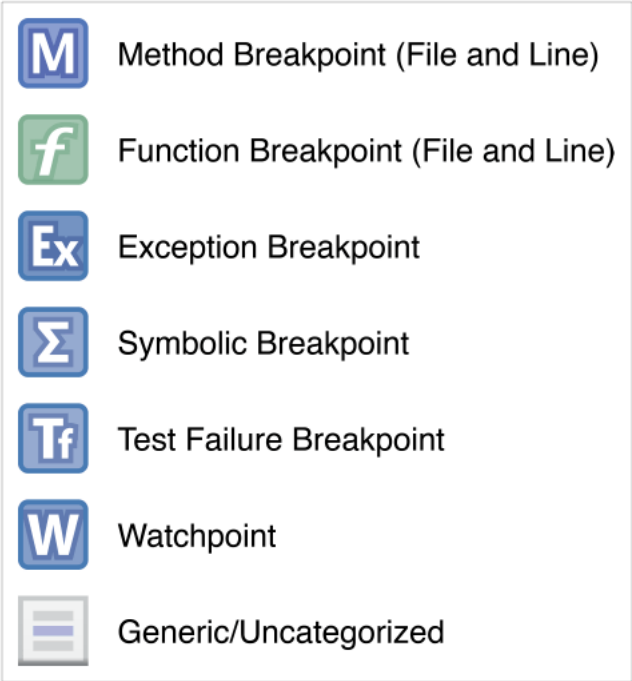
All these breakpoints were created and left at the defaults so they are all set to project scope. To change the scope of a breakpoint, Control-click the breakpoint and use Move Breakpoint To > User or Jogr, the two available choices since this is a project not contained in a workspace.



The result of that change is shown in the hierarchy like this:



Similar to variables listed in variable view, the several kinds of breakpoints listed in the breakpoint navigator are tagged with icons to make it easy to recognize them.



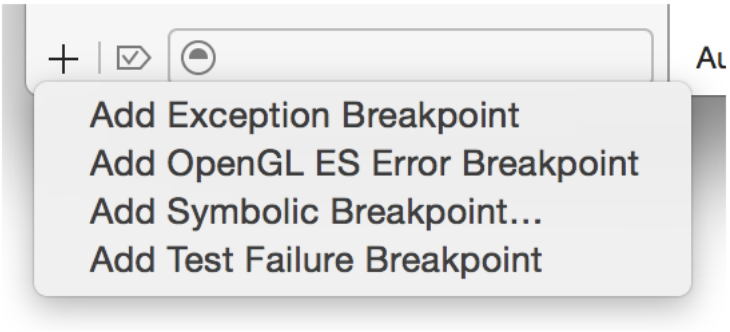
Note: Xcode categorizes breakpoints that you create in the source editor gutter as "method" or "function" based on the context of the source code they are a part of. A breakpoint placed at an inappropriate location is categorized as "generic/uncategorized"—these are usually unreachable points in the source code file, such as between functions or methods, placed by accident.

Disable/enable breakpoints. You click on the breakpoint indicators individually to disable or re-enable them. The color changes from dark to light when disabled to indicate the state.

Note: Clicking the Breakpoint Activation button in the debug bar globally toggles all breakpoints in all files and scopes to be activated or deactivated per their individual enabled/disabled state. When deactivated globally, the indicators in the breakpoint navigator are dark and light gray, reflecting the individual state of each breakpoint.

Breakpoint editing. Control-click the breakpoint in the breakpoint navigator and choose Edit Breakpoint from the pop-up menu to display the breakpoint editor. Choosing Edit Breakpoint displays the breakpoint editor, allowing you to set conditions, add actions, and so forth, as mentioned in the Breakpoints (page 41) section.

Filter bar. To add an exception, symbolic, or other type of breakpoint to your code, click the Add (+) button at the lower right.



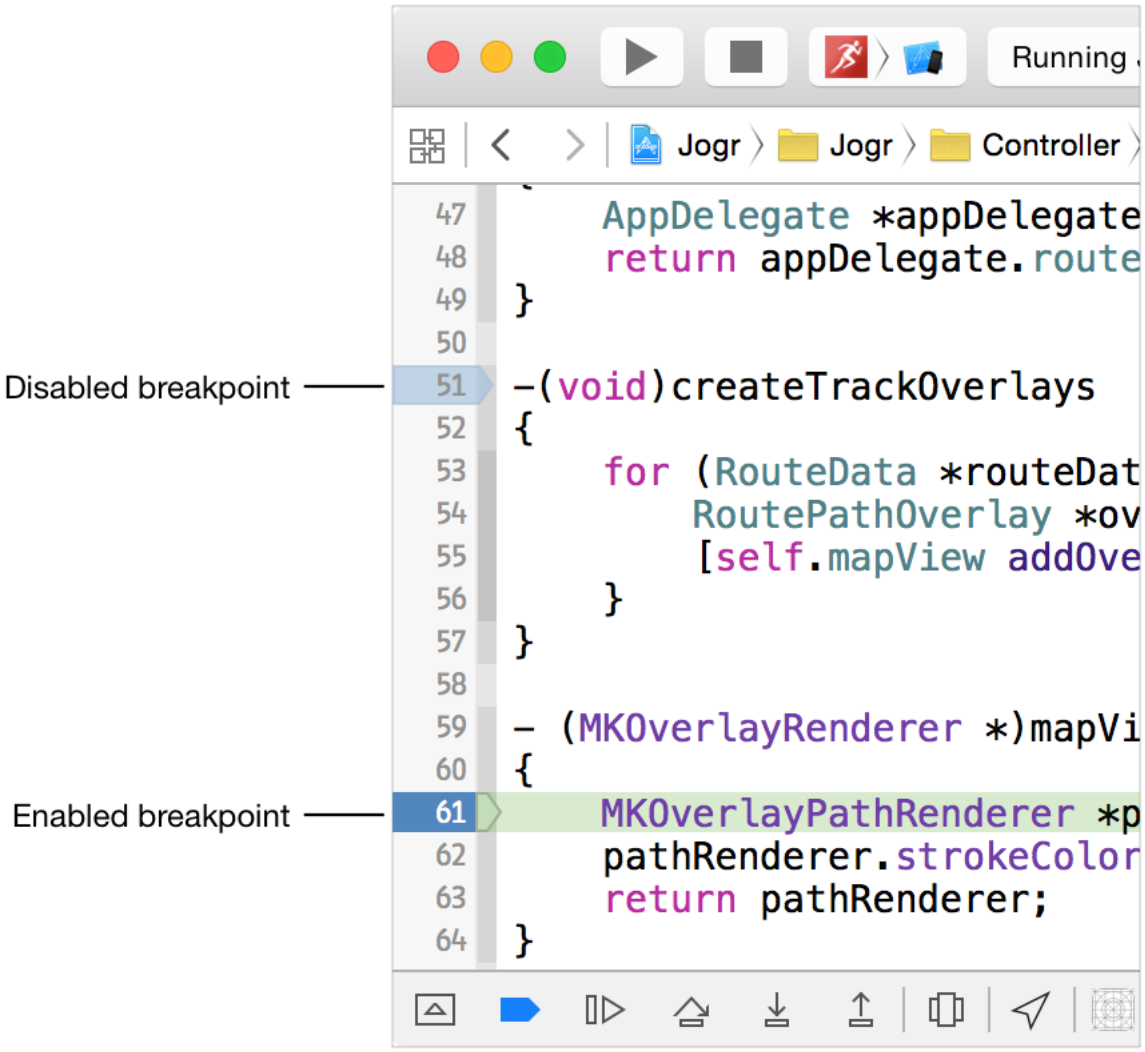
Click the "enabled breakpoints" filter button, , to show only the enabled breakpoints in your project. Enabling this filter is handy if you have a lot of breakpoints defined but are working with only a small set of them in an enabled state.

Type into the text filter to limit the view of breakpoints in all scopes and files to ones with text that matches. Using the text filter is a fast way to find a breakpoint in a function or method when you have have a large project with many files and defined breakpoints.

For more information on using the breakpoint navigator, see *Breakpoint Navigator Help* .

Source Editor

In addition to displaying and editing your source code, the source editor plays a vital role in debugging operations. The source editor is where you set breakpoints, and often where you manage them dynamically during a debugging session. Having your code first and foremost in front of you can also be the best way to work through a problem—you can minimize the debug area and debug navigator to use the source editor as the debugging interface.

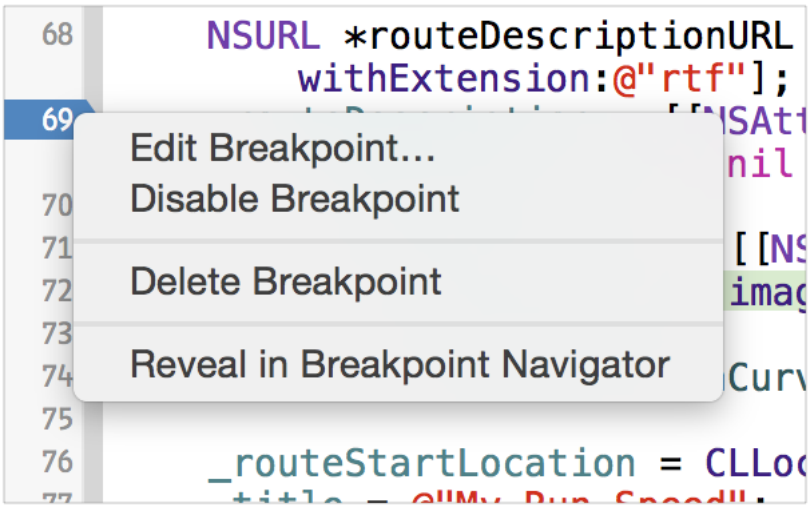


The illustration above shows the source editor, with debug area and debug navigator collapsed, and the app paused by an enabled breakpoint. The green pointer and highlight on line 61 in the source indicates the location of the instruction pointer and the current line of code to be executed.

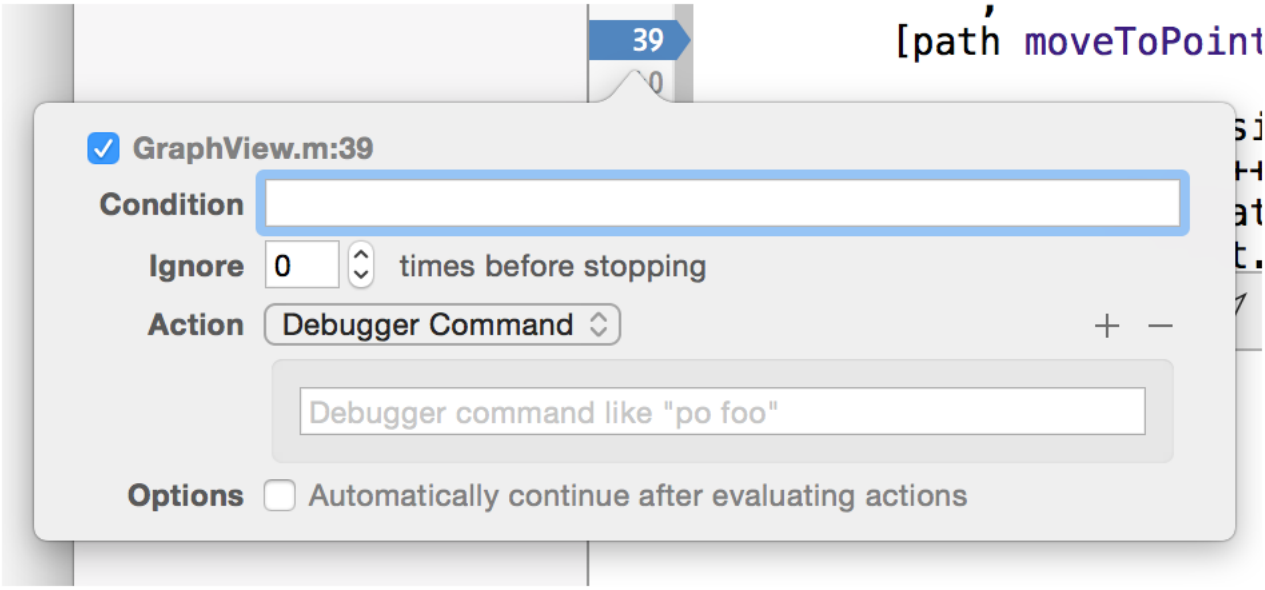
> **Note:** You can manipulate the instruction pointer in the source editor. Control-click in the gutter (not on a breakpoint), choose "Continue to Here," and Xcode executes the source code up to that point. This is often useful if your code needs to execute an extensive loop from where you wanted to break for inspection before reaching the next point that you want to inspect again.
>
> You can similarly drag the instruction pointer to a location; Xcode warns you that moving the instruction pointer this way can cause problems up to and including a crash. What this operation does is simply move the instruction pointer to a different location, enabling you to start from a different location. This kind of instruction pointer manipulation is only infrequently useful for testing very specific situations, for example, when you've accidentally stepped over a line in the source and need to force it to be executed.

Create/delete breakpoints. Click the gutter next to the line where you want execution to pause. When you add a breakpoint, Xcode automatically enables it. When an enabled breakpoint is encountered during execution, the app pauses and a green indicator shows the position of the program counter. You can create breakpoints at any time, including when your app is running in a debugging session. If a breakpoint is not in the ideal location, drag the indicator to a better location. To delete a breakpoint, drag the indicator out of the gutter.

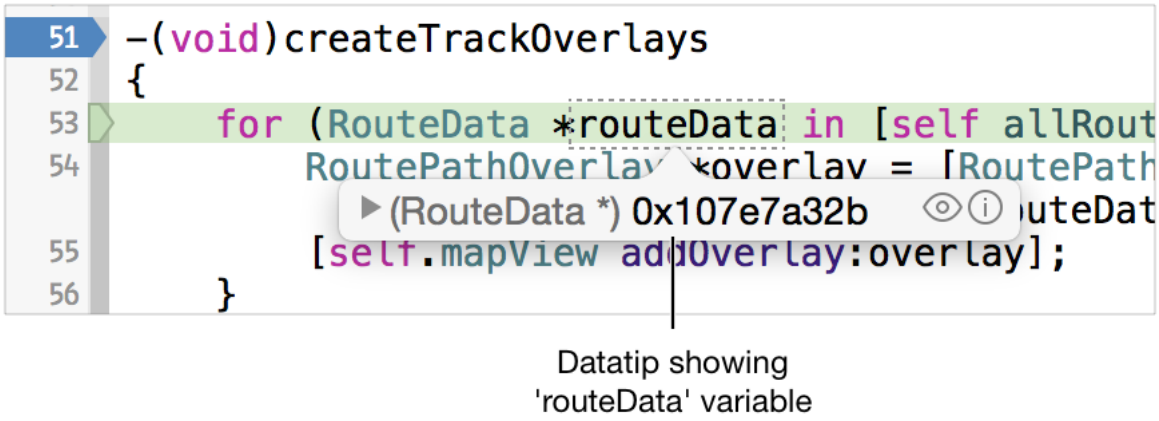
Disable/enable breakpoints. You disable a breakpoint in the source editor by clicking the breakpoint indicator, even while your app is running; this allows you to adjust set locations where the app will be paused as you work through a problem. Disabled breakpoints display a dimmed blue indicator. Enable a breakpoint by clicking the indicator again.



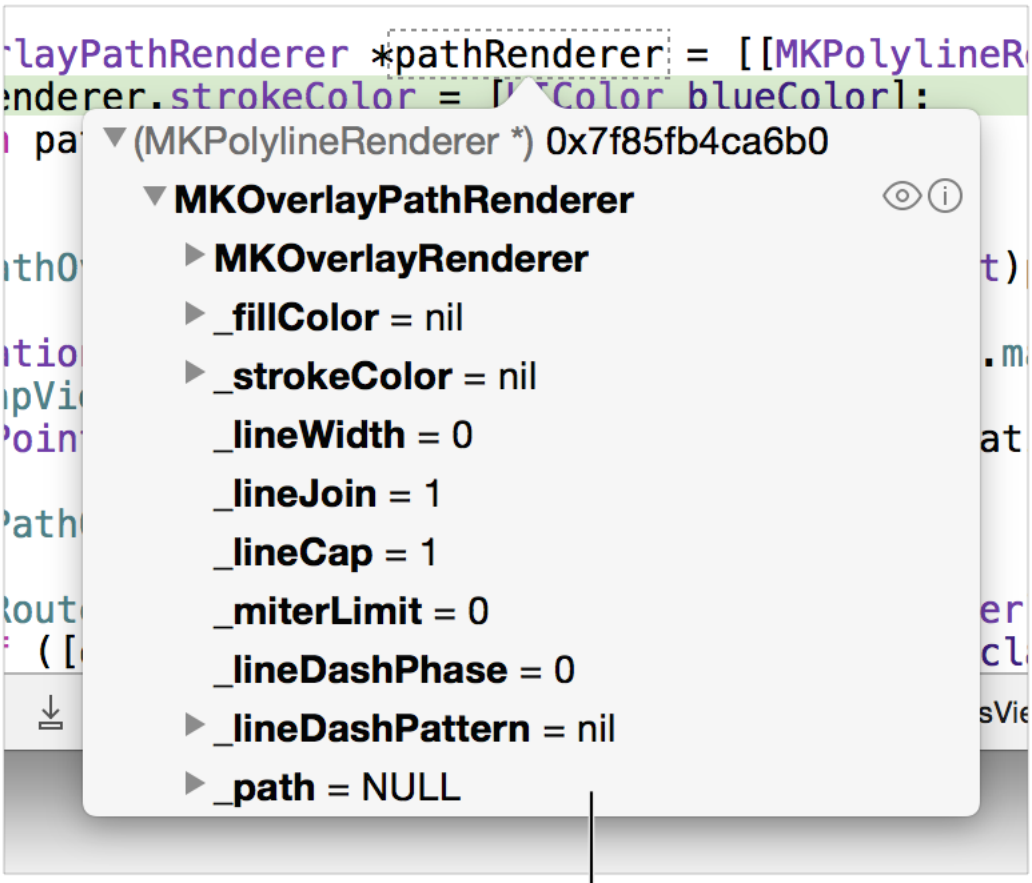
Edit breakpoints. Control-click a breakpoint indicator to display a command menu and choose Edit Breakpoint to open the breakpoint editor and set conditions, add actions, and so forth, as mentioned in the Breakpoints (page 41) section.



Tip: Command-option click on a breakpoint to open the breakpoint editor without having to choose Edit Breakpoint from a menu.



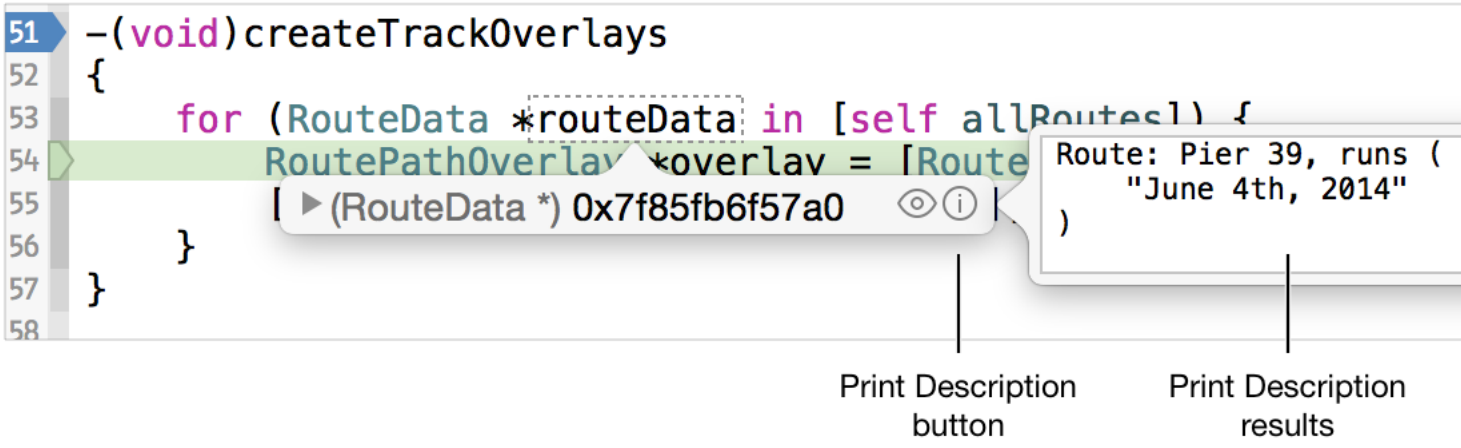
Datatips. You can use the source editor alone during a debugging session with the debug area hidden, showing only the debug bar, to give you a larger view of your source code and the flow of your program as you step.



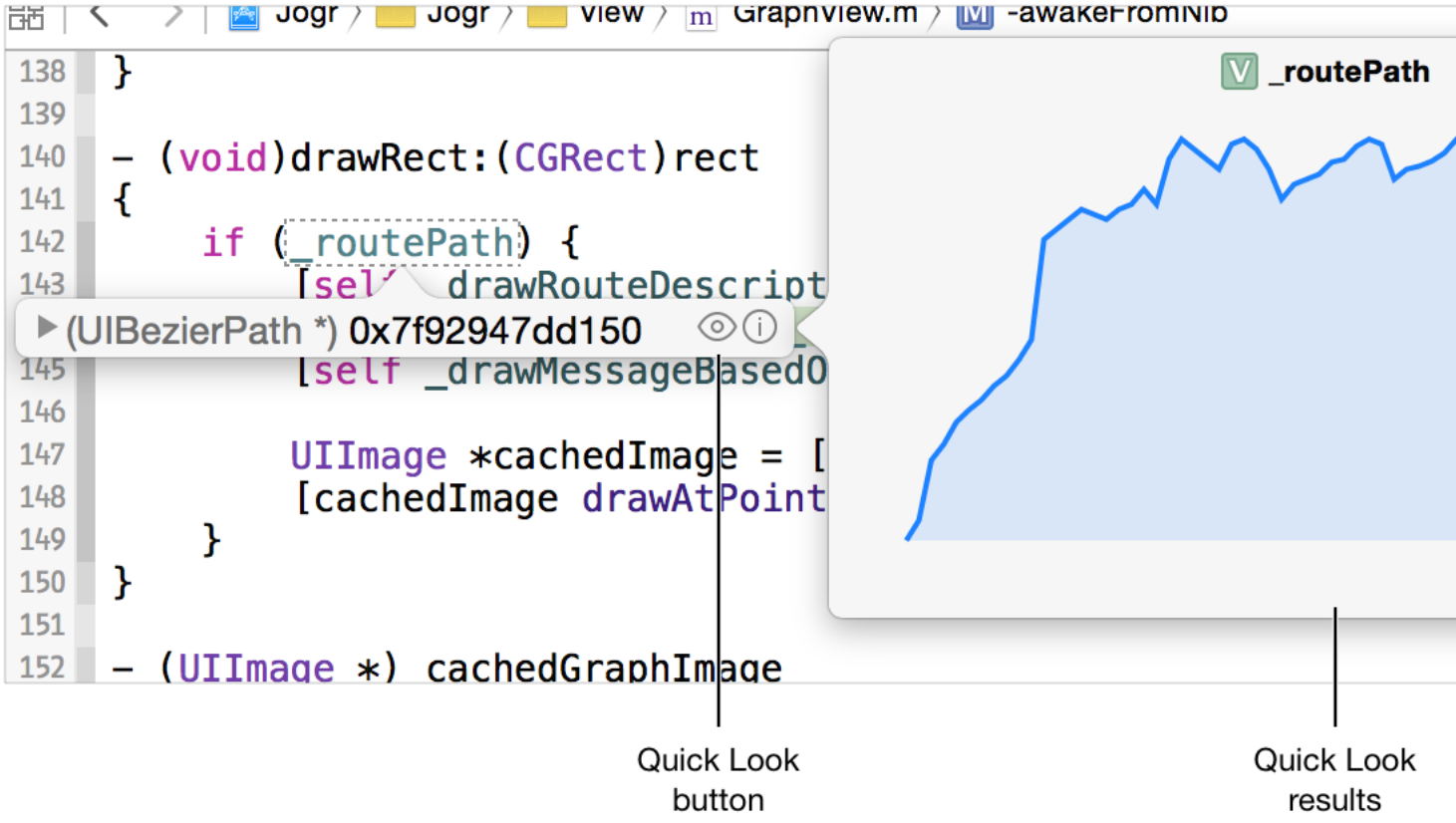
Datatip with
'pathRenderer' disclosed

You can inspect variables directly in the source editor with a datatips popup window, which shares the same display layout as the debug area variable view. Both Quick Look and Print Description buttons are immediately available from the datatips popup window, and operate entirely within the source editor.

Print Description button. In this example, you hold the mouse over the `routeData` variable to display a datatip. When the Print Description button is clicked, results are displayed just as they would be in the console in a popover window.

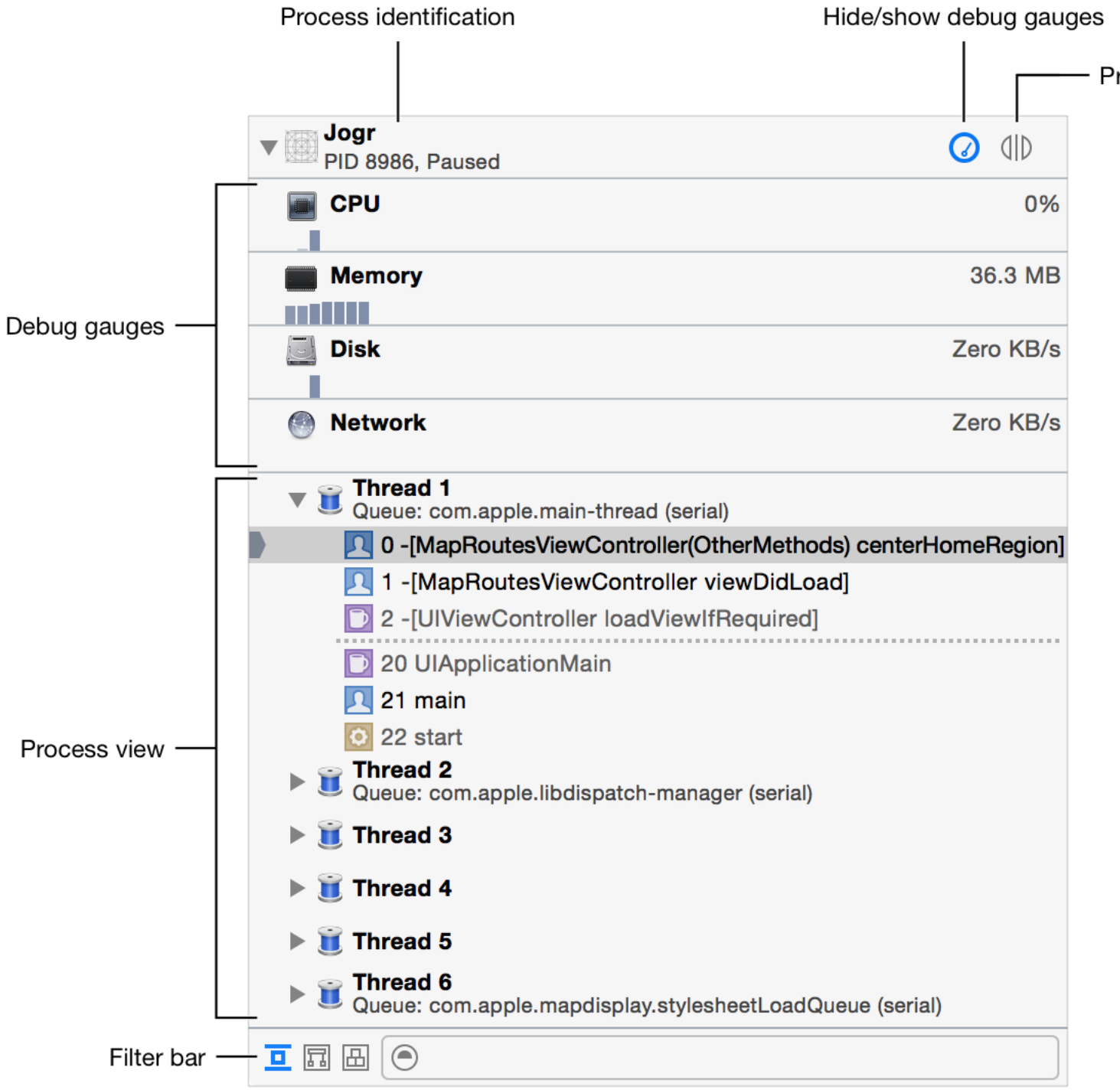


Quick Look button. The Quick Look feature works similarly, this time electing the `_routePath` variable in the example.



The Debug Navigator

The debug navigator has two main parts, debug gauges and process view display, as well as view controls and a filter bar. It provides tools for discovering issues in your running app and presents the call stack for inspection when the app is paused.



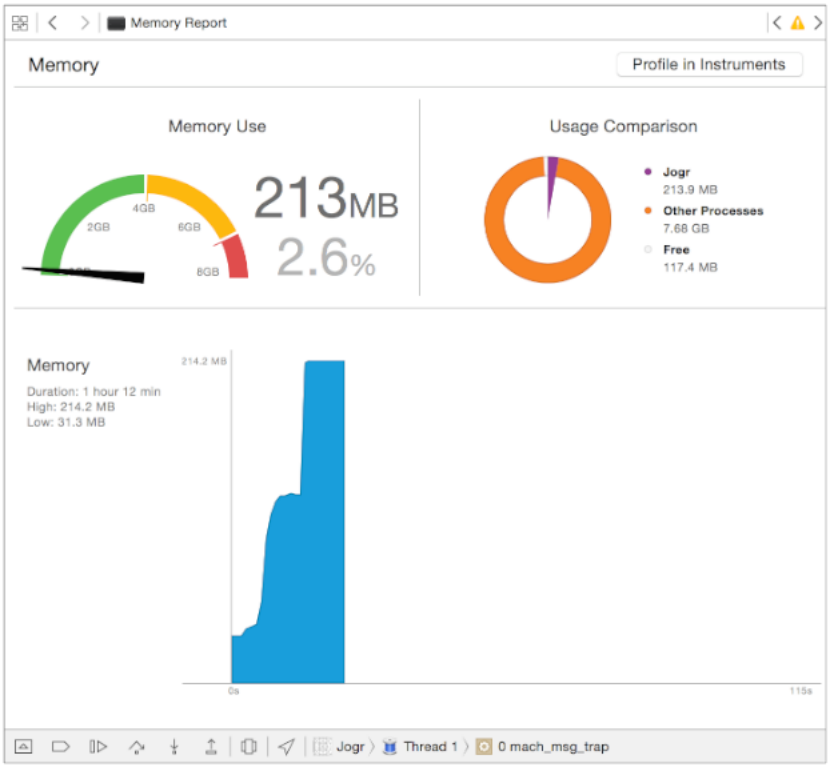
Debug Gauges

The debug gauges monitor your running app. Debug gauges constantly change as your app runs—they are presented as a histogram running from left to right in time. Spikes in the histogram show activity.

There are seven types of debug gauges: CPU, Memory, Energy, Disk I/O, Network I/O, GPU, and iCloud. They are displayed depending on platform and app capabilities. CPU, Memory, Disk I/O, and Network I/O are available for all app development. The Energy gauge is available in OS X apps when running Xcode 6.3 and greater. The iCloud and GPU gauges are available when your app target has been subscribed to associated capabilities with the project editor; the GPU gauge is available on iOS only.

The debug gauges provide insight about how your app is performing in its use of system resources. Depending on the capabilities of your app and the characteristics of its destination, gauges can report your app's impact on the system and other running apps. Observe the debug gauges for a running app to become familiar with the gauges' normal variation and the standard behaviors of the app over time. When you make changes to the app, look for differences in the debug gauges readings and look into any new behavior that seems anomalous or indicative of a problem compared to the app's previous running behavior.

Click a debug gauge as your app runs to display a live detail report in the main editor. The detail report of each type of gauge differs in the specifics but all follow a similar pattern. As an example, here is a Memory Report shown when you click the Memory debug gauge, captured from a paused app:



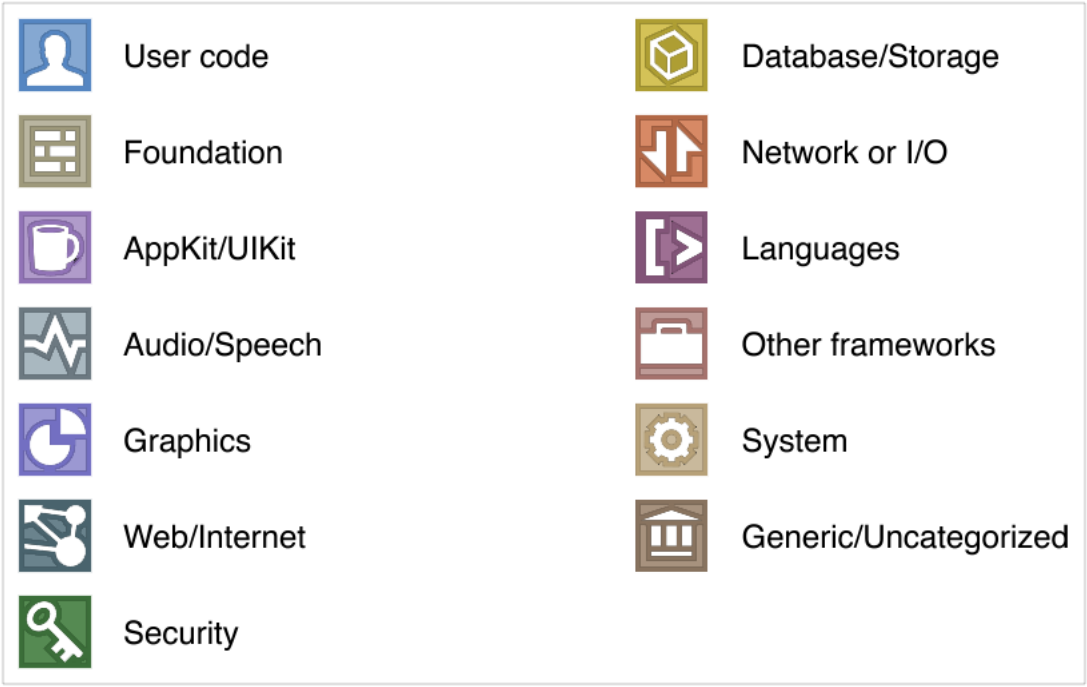
> **Note:** When your app is paused or stops at a breakpoint, the last state of the debug gauges freezes and the detail report of each gauge at that moment can be accessed. Live indication stops.

You can hide the debug gauges to have more room for the process view display by clicking the Hide/Show debug gauges button ( ).

Further information on using the debug gauges can be found in Debug Navigator Help.

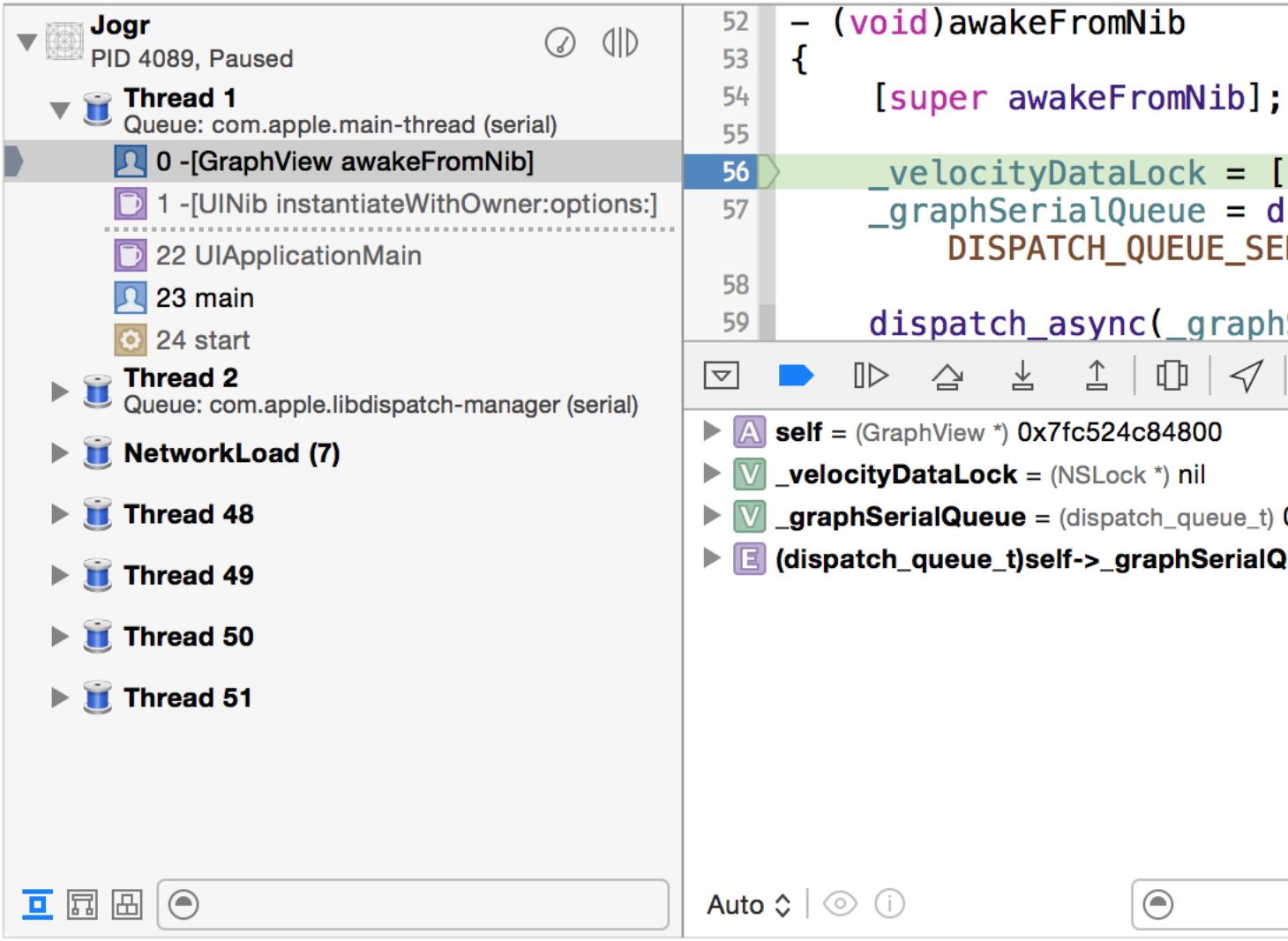
Process View Display

The process view display of the debug navigator displays the backtrace of your paused app organized by either thread or Grand Central Dispatch queues. With this tool you can debug the control flow of Swift or C-based code as well as OpenGL frames. The backtrace is displayed with each stack frame identified by an icon. The icons tell you where in the compiled code the stack frame is from:



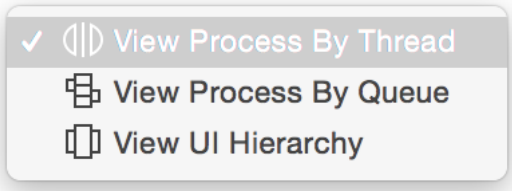
These icons are displayed in different colors to help you differentiate them easily. When they are colored gray, it means the stack frame indicated is from the recorded backtrace and is not live in memory in the context of the paused app.

When your app pauses at a breakpoint, by default the backtrace is organized by threads with the current stack frame highlighted and the corresponding source in the source editor showing the program counter positioned at the breakpoint. In this example, the app has paused at a breakpoint placed at line 56 in the source file in the `awakeFromNib` method. You can see the stack frame selected in the debug navigator, the source and breakpoint in the source editor, and the variable list in the debug area.



When you select another stack frame, information about the stack frame is displayed in the source editor and in the debug area. If source is not available, the source editor displays the code as decompiled assembly instructions. For a brief description of how to read the backtrace, see Examining the Backtrace in the Debug Navigator (page 22) in the *Quick Start* chapter.

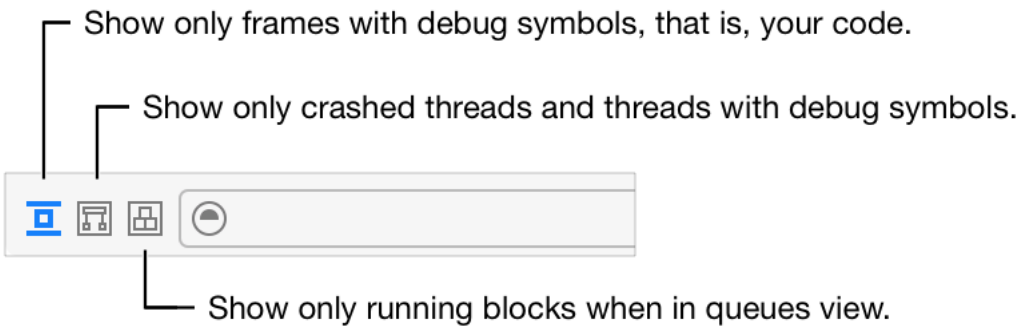
The process view options button ( ) enables you to switch the process view display between thread organization and queues organization with a menu:



Selecting View Process by Queue changes the backtrace organization to show stack frames below the dispatch queue that created them. This is useful when looking for bugs that might be caused by queue interactions and blocking. See the discussion of how to use the queue display in the next chapter, Thread and Queue Debugging (page 72).

> **Note:** The View UI Hierarchy option is also available from this menu. This option changes the debug navigator to work with bugs in the view hierarchy. It is discussed in the next chapter, see Debugging View Hierarchies (page 65).

The debug navigator filter bar enables you to to remove extraneous information and focus on your own code during a debugging session.



- **Show only frames with debug symbols** hides threads that may not be relevant to debugging your code by suppressing the display of stack frames without debug symbols. Examples of such threads include the heartbeat and dispatch management threads and any threads that are idle and not currently executing any specific application code.

- **Show only crashed threads** displays only crashed threads and threads with debug symbols. It collapses the list to help you focus your debugging efforts by hiding calls that are far removed from your code. The presence of hidden symbols is indicated by a dotted line in the stack frame.
- **Show only running blocks** suppresses the display of non-running blocks when the debug navigator is displaying queues.

Debugging the View Hierarchy

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, labels and text that are incorrectly clipped or placed—these are all examples of issues with an app's view hierarchy.

Using the Xcode debugger, you can inspect the view hierarchy in detail, using a hierarchical listing of view objects, an exploded 3D rendering of your app's view hierarchy, and inspectors for object attributes and sizing. The view hierarchy tools are linked to your source code as well. These tools help you to a speedy resolution of this class of bugs.

To read a full discussion of using the Xcode debugger to debug a view hierarchy, see Debugging View Hierarchies (page 65) in the next chapter, *Specialized Debugging Workflows*.

OpenGL ES Debugger

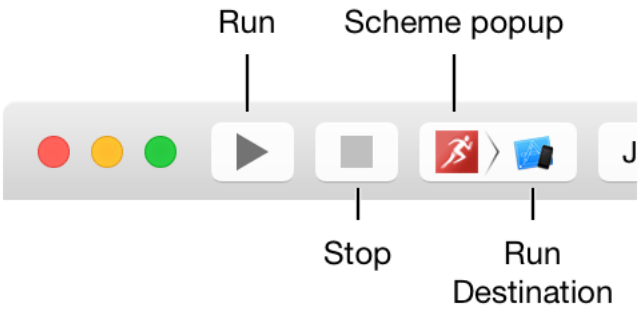
With the Xcode debugger, you have tools for debugging, analyzing, and tuning OpenGL ES and Metal applications that are useful during all stages of development. The FPS Debug Gauge and GPU report summarize your app's GPU performance every time you run it from Xcode, allowing you to quickly spot performance issues while designing and building your renderer. Once you've found a trouble spot, you can capture a frame and use Xcode's OpenGL ES Frame Debugger interface to pinpoint rendering problems and solve performance issues.

A larger description of the features and workflow used with the OpenGL ES debugger is presented in Debugging OpenGL ES and Metal Graphics (page 78) in the next chapter, *Specialized Debugging Workflows*.

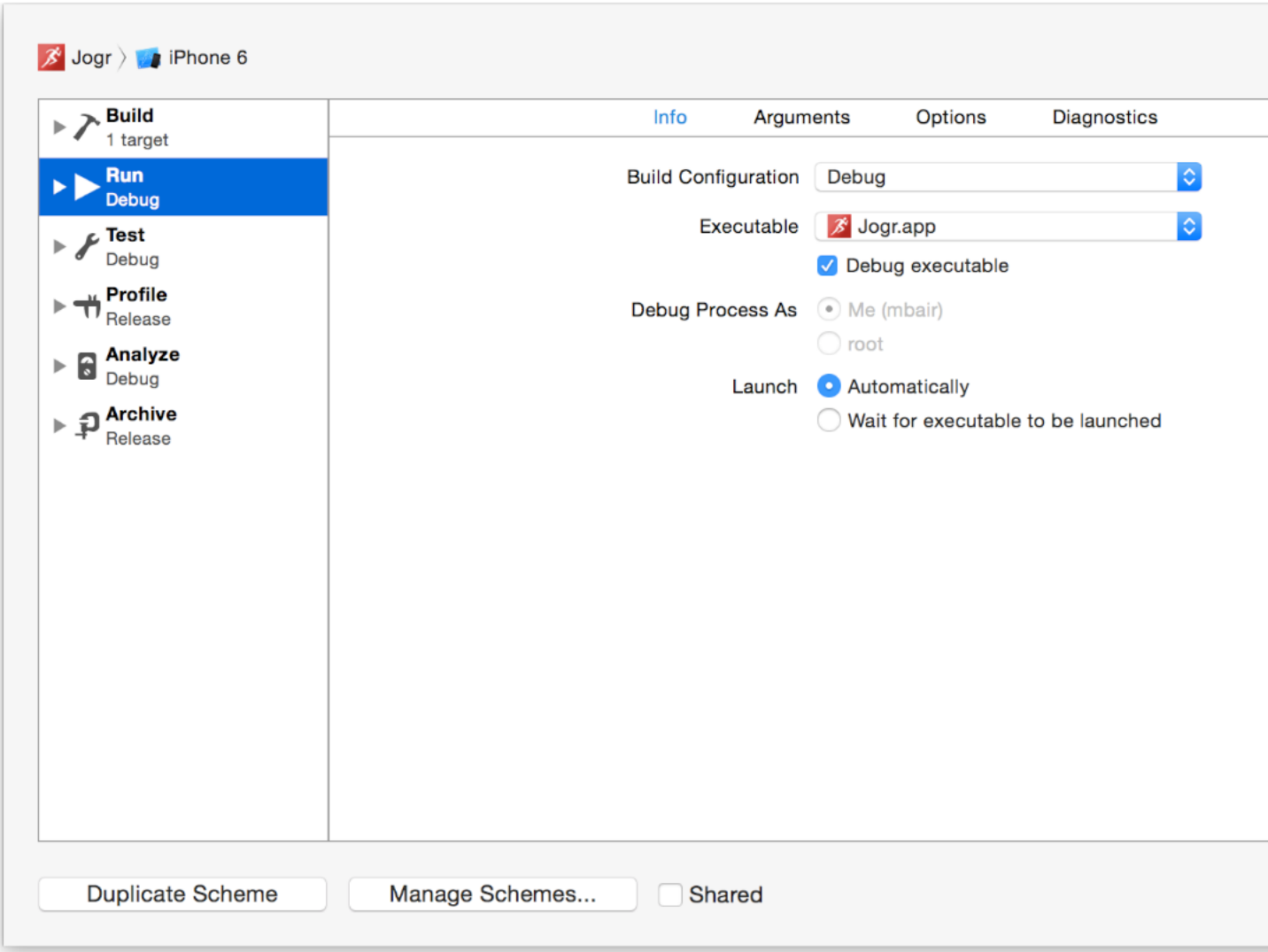
Debugging Options in the Scheme Editor

Xcode schemes control the build process. For most uses, the scheme defaults created when you create a project or target are sufficient for most debugging. However, useful debugging options are configurable in the scheme editor as part of the Run action configuration. Most of the scheme editor options are covered in detail in Scheme Editor Help.

To open the scheme editor by choose Edit Scheme from the Scheme menu in the toolbar.

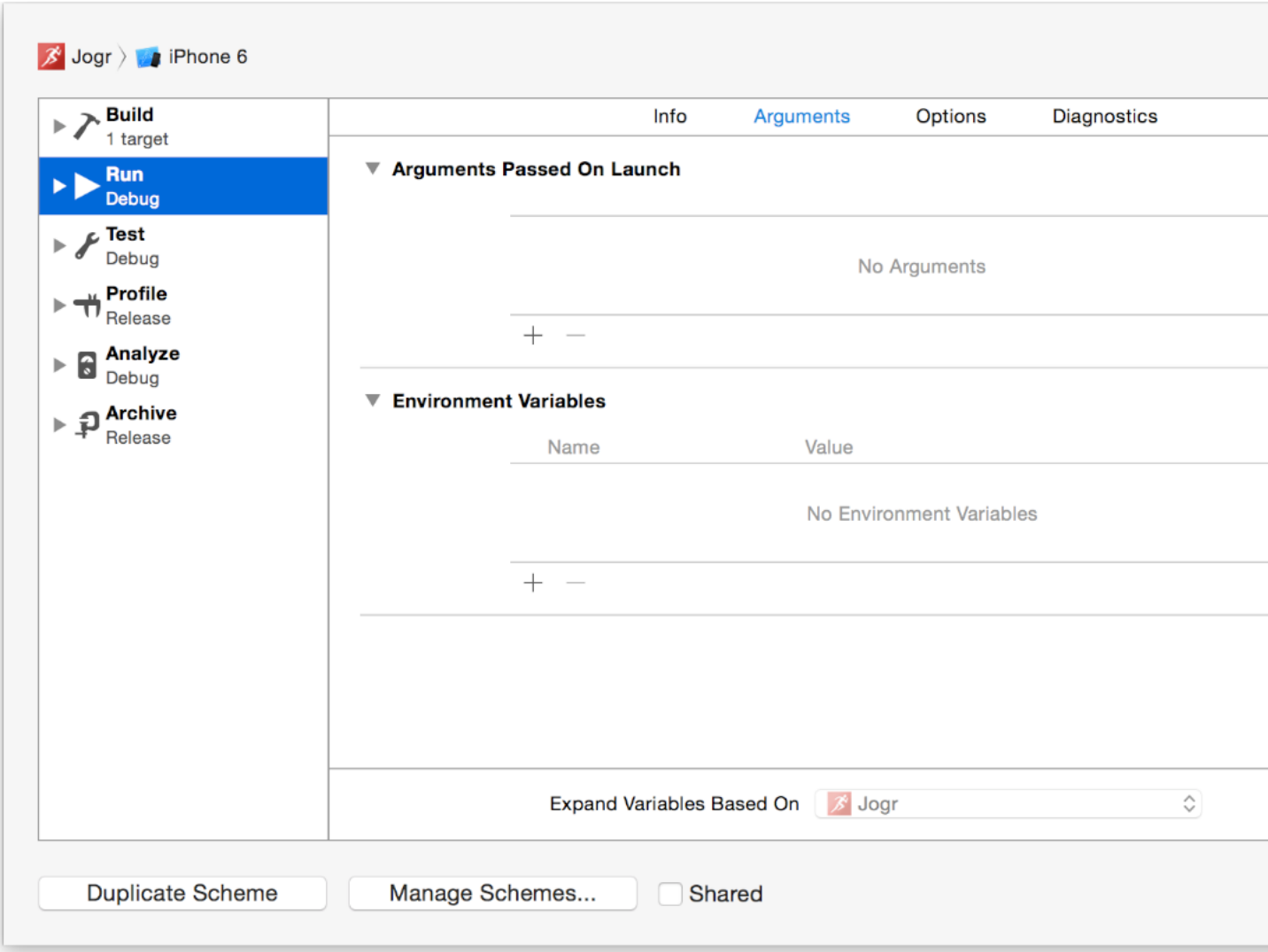


Info panel. The info panel contains settings for the build configuration.

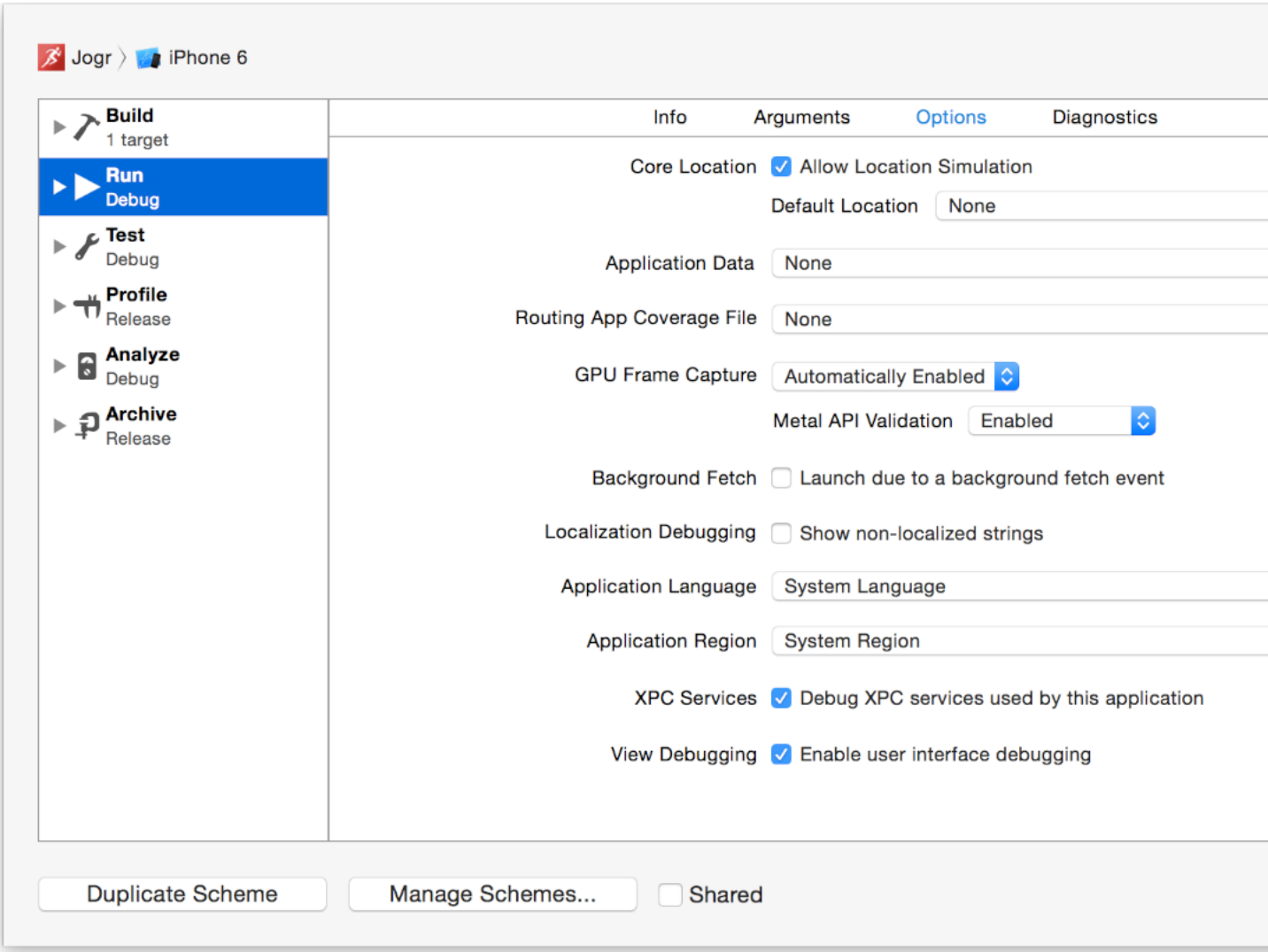


- The "Debug executable" checkbox specifies whether or not you want to run with the debugger enabled. Once running, you can use Debug > Attach to Process on a process that has been launched with debugging disabled if needed.
- Set Debug Process As to root if you are working on code that requires root privileges.

Arguments panel. You can set up arguments to pass on launch and environment variables for your app with this panel.



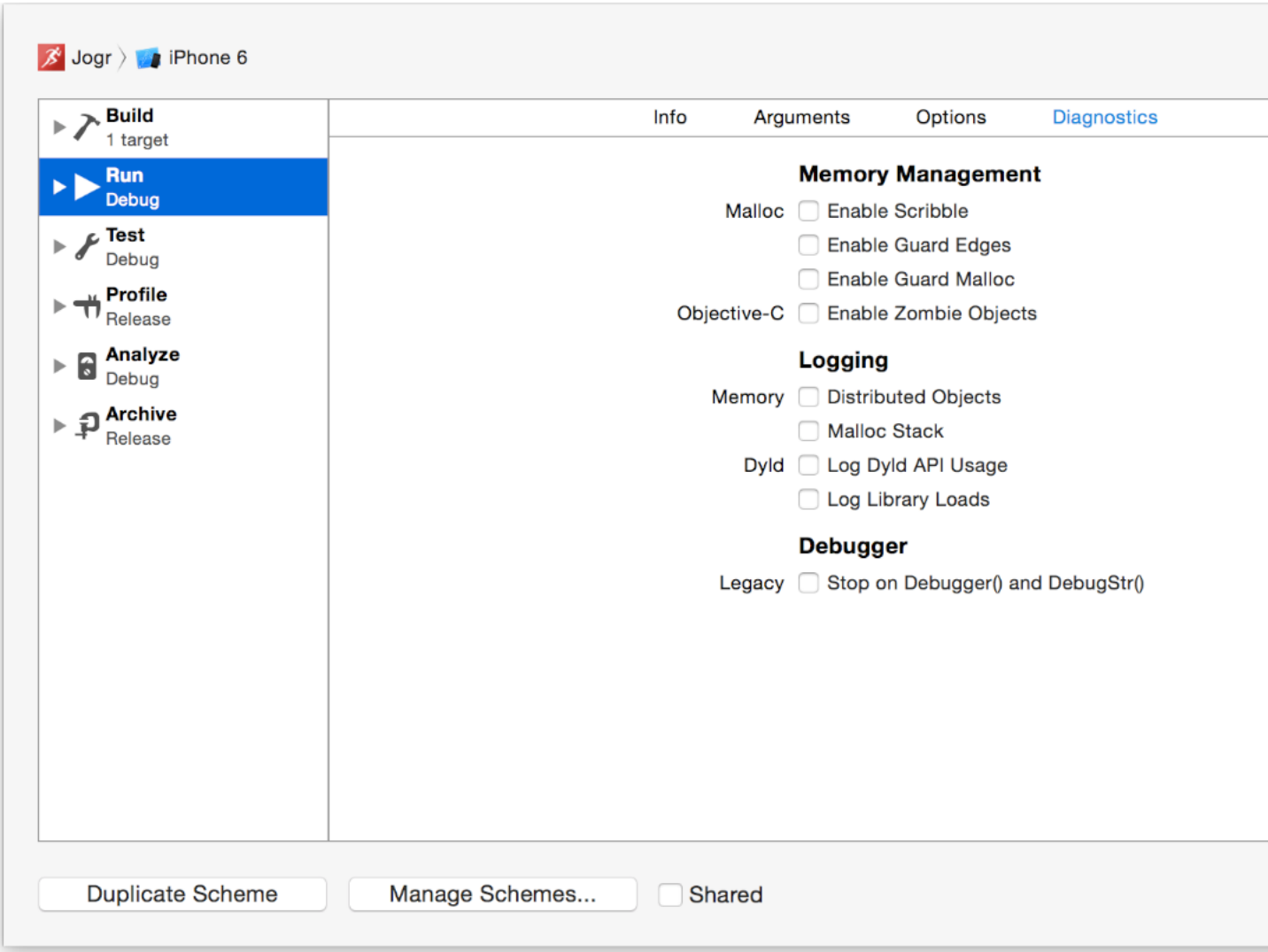
Options panel. Options settings are specific to different technology needs, not just debugging.



- Allow Location Simulation enables the Xcode debugger to provide location simulation services. It is on by default.
- GPU Frame Capture is enabled by default, set to Automatically Enabled, and allows the Xcode debugger to support the OpenGL ES debugging environment. You can disable it, or limit it to either OpenGL ES or Metal technology.
- When you select “Debug XPC services used by this application,” Xcode automatically attaches to and breaks in any XPC services triggered by your app. This makes debugging your XPC services seamless (remember that they are separate processes). This option is on by default.

- The View Debugging option allows a special debugging library used by view hierarchy debugging to be injected into your app context when running in the debugger; it is on by default.

Diagnostics panel. The diagnostics panel allows a variety of additional diagnostic tools to be added to the Run environment.



Select the tools that you want Xcode to use. You can view output from these tools in the debug area console and in the debug log in the reports navigator.

Memory Management options:

- **Enable Scribble.** Initialize allocated memory with `0xAA` and deallocated memory with `0x55`.
- **Enable Guard Edges.** Add guard pages before and after large allocations.

- **Enable Guard Malloc.** Use `libgmalloc` to catch common memory problems such as buffer overruns and use-after-free.
- **Enable Zombie Objects.** Replace deallocated objects with a "zombie" object that traps any attempt to use it. When you send a message to a zombie object, the runtime logs an error and crashes. You can look at the backtrace to see the chain of calls that triggered the zombie detector.

> **Note:** Using Guard Malloc and Zombie Objects diagnostics options disables the memory debug gauge.

Logging options:

- **Distributed Objects.** Enable logging for distributed objects (`NSConnection`, `NSInvocation`, `NSDistantObject`, and `NSConcretePortCoder`).
- **Malloc Stack.** Record stack logs for memory allocations and deallocations.
- **Log DYLD API Usage.** Log dynamic-linker API calls (for example, dlopen).
- **Log Library Loads.** Log dynamic-linker library loads.

Debugger options:

- **Stop on Debugger and DebugStr.** Enable Core Services routines that enter the debugger with a message. These routines send a `SIGINT` signal to the current process.

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.

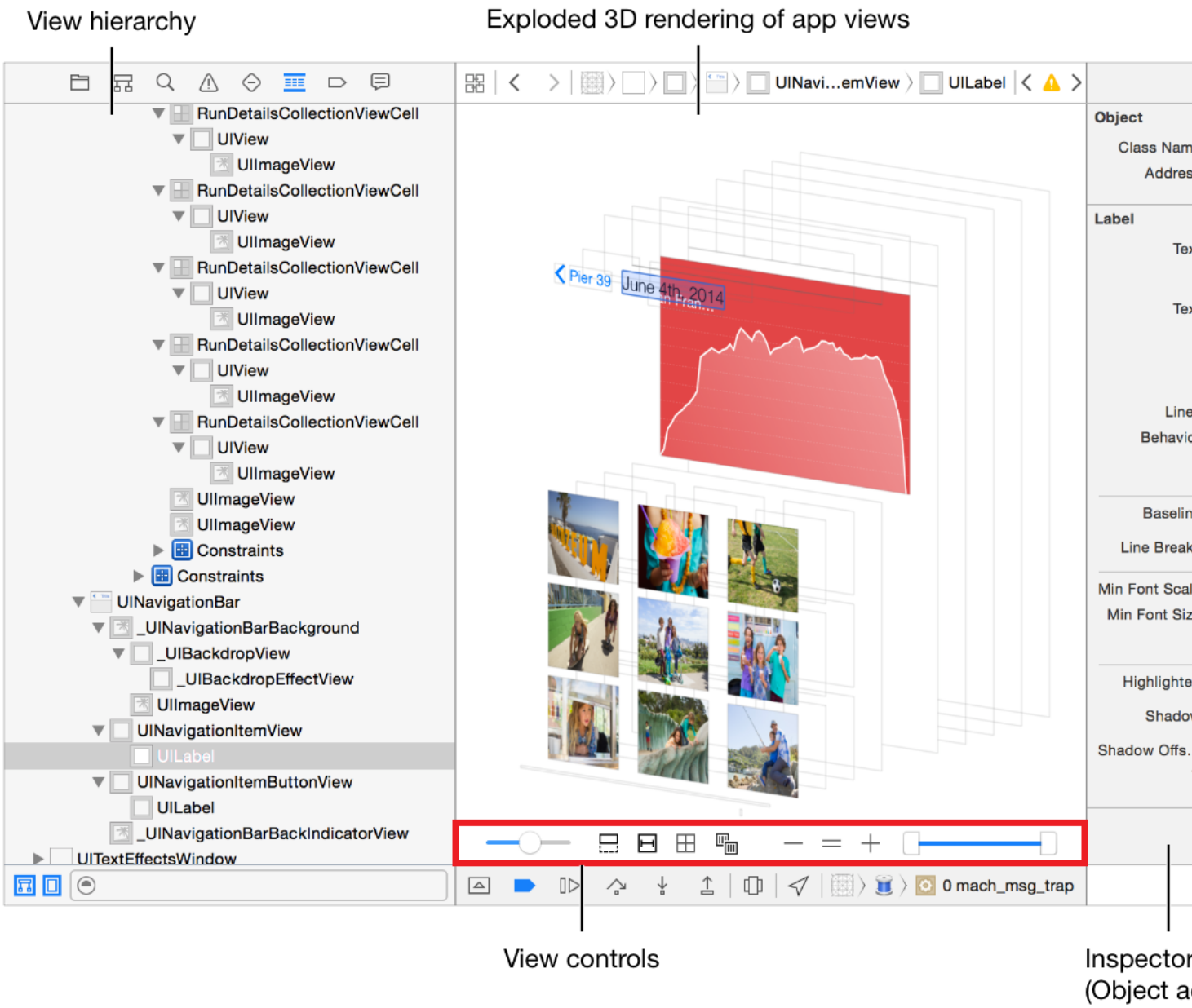
Basic Operation

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 using 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.



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.

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. On the left is Show Primary Views, which filters out view objects that are secondary elements of system view implementations, not under app control. On the right is Show Only Visible Views, which filters out views that have been hidden by the app using the view attributes. It is particularly useful to disable Show Only Visible 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

Use 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 the second segment of this video presentation: WWDC 2014 Debugging in Xcode 6: Debugging User Interfaces in Xcode.

Debugging App Extensions

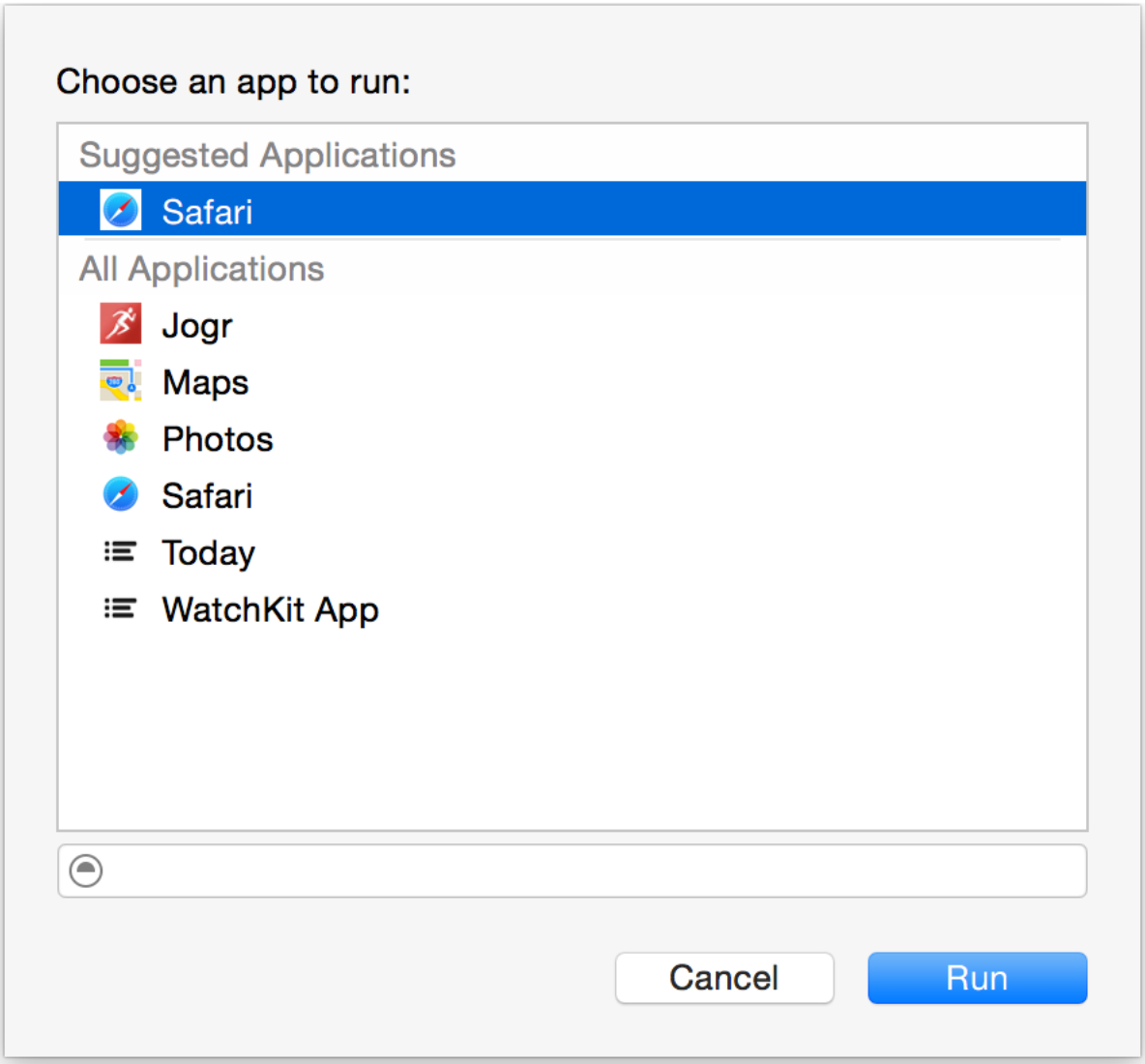
Using Xcode to debug an app extension is a lot like using Xcode to debug any other process, but with one important difference: In your extension scheme's Run phase, you specify a *host app* as the executable. Upon accessing the extension through that specified host's UI, the Xcode debugger attaches to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

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

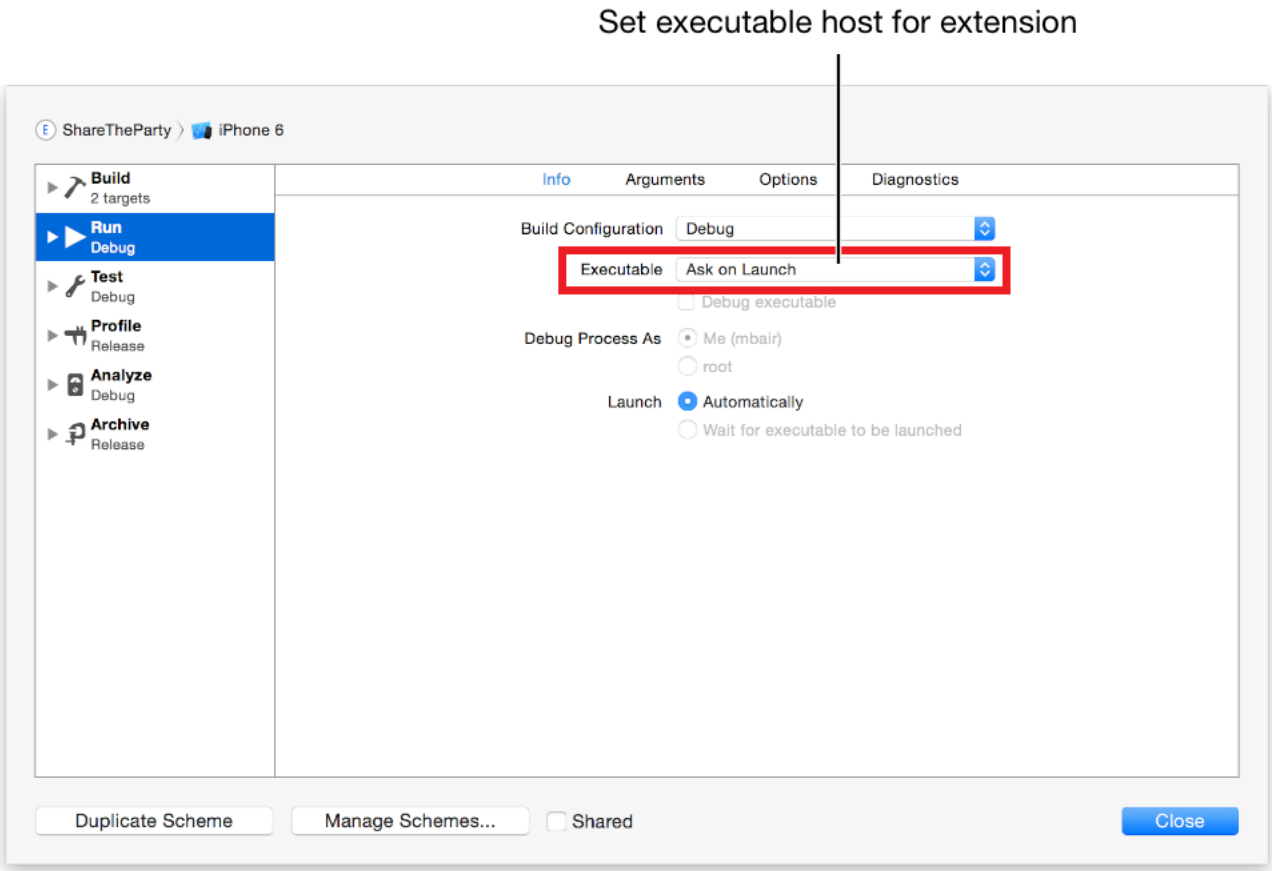
All the targets in your Xcode project must be code signed in the same way. For example, during testing you can employ ad hoc code signing or use your developer certificate, but must use the same approach for all the targets in your project. For submission to the App Store, use your distribution certificate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

Build and Run for Debugging with Ask On Launch

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



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



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, ensure 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.

Thread and Queue Debugging

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

Background Information

To take advantage of multi-core 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 OS X 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 up 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 wish 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 how the debug navigator presents it to you, see Examining the Backtrace in the Debug Navigator (page 22) 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 down how your app arrived at this point.

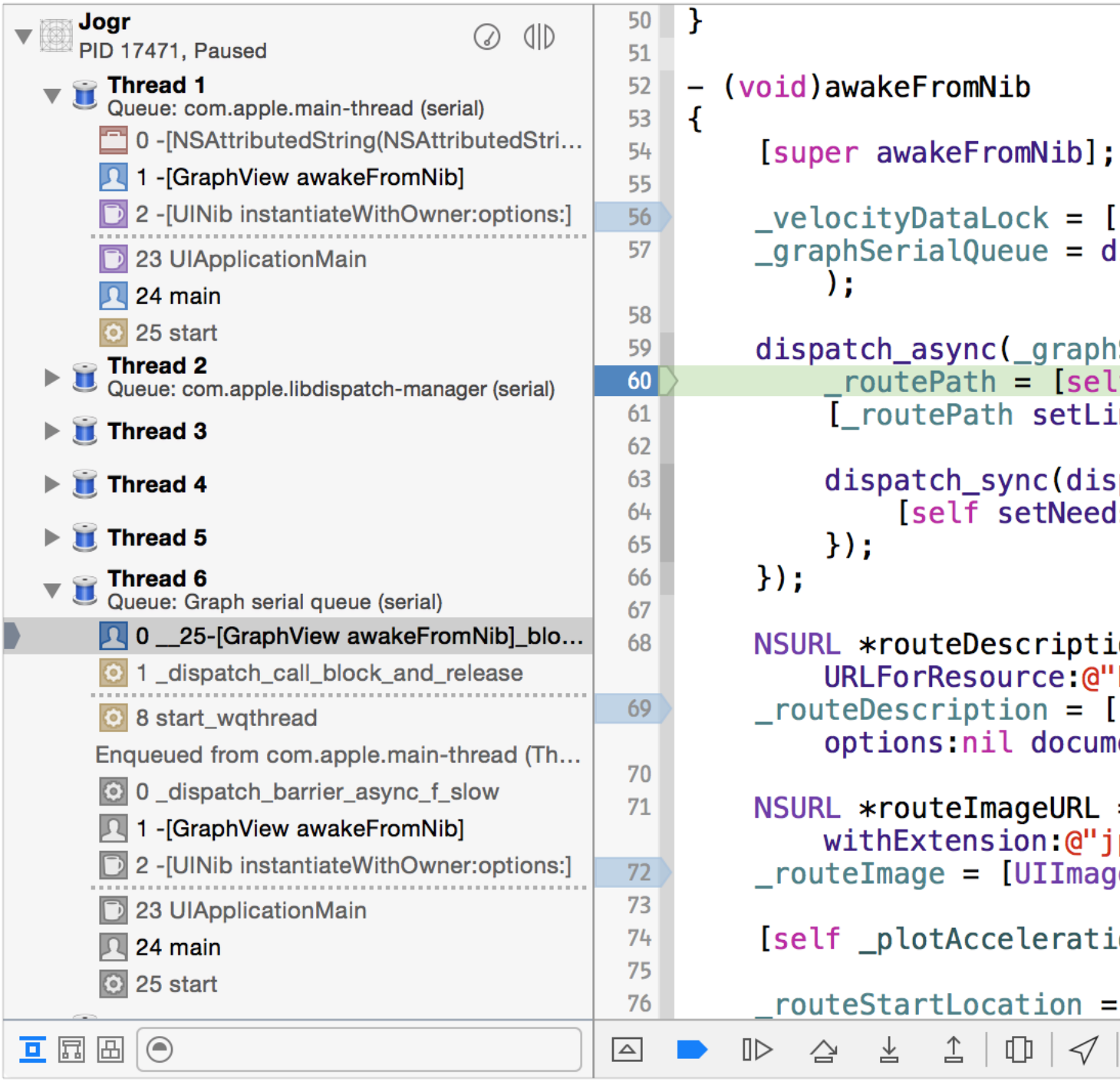
If the backtrace only contains the live code at the time it hit the breakpoint, the control flow of the app is difficult to determine because the dequeuing thread may have already been terminated. You would need to find all call sites of the function, set breakpoints appropriately, and run again, replicating 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 up 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 is most useful in serially organized code operation but has its limitations when you are using async blocks.

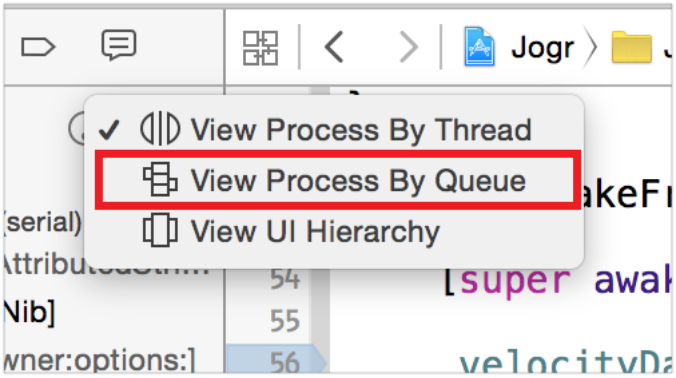


In this example, the app is paused by a breakpoint in the method `awakeFromNib` inside a block that was queued asynchronously. You can see how the historical backtrace was spliced into the display of Thread 6 to show the enqueuing event and complete the backtrace.

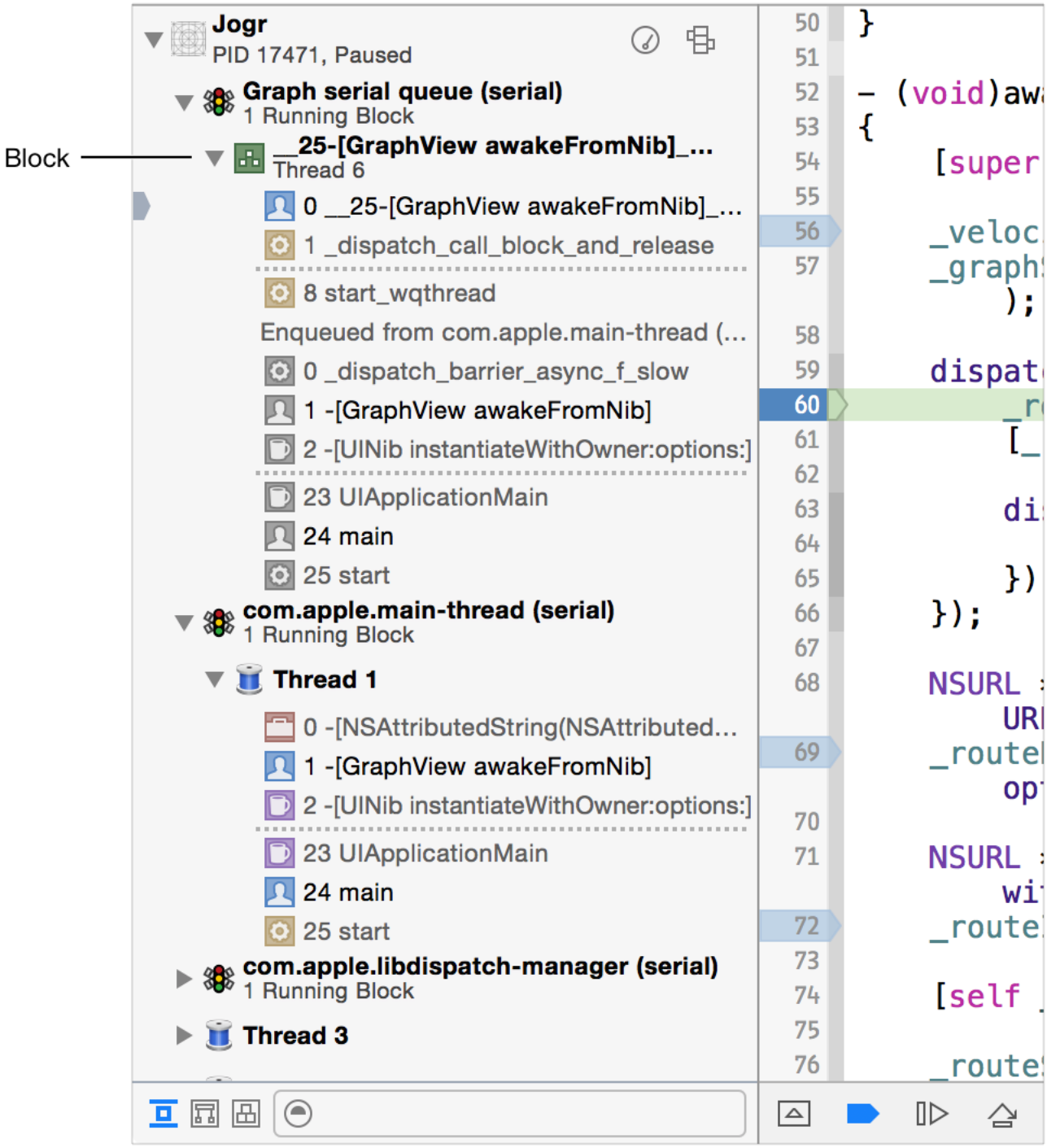
- Clicking on a stack frame for which Xcode has symbols displays the source code associated in the source editor.
- Clicking on a stack frame for which there are no symbols displays disassembly in the source editor.
- Gray icons indicate historical data that's been recorded but is no longer in memory. 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.



This example is 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 like:

- 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 which 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 this video presentation: WWDC 2014 Debugging in Xcode 6: Queue Debugging.

Debugging OpenGL ES and Metal Graphics

Xcode provides tools for debugging, analyzing, and tuning OpenGL ES and Metal applications that 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. Once you find a trouble spot, you can capture a frame and use Xcode's OpenGL ES Frame Debugger interface to pinpoint rendering problems and solve performance issues.

This section reviews the workflow and user interface basics of the OpenGL ES Debugger in Xcode. A detailed look at the OpenGL ES Debugger is available in the "Xcode OpenGL ES Tools Overview" chapter of *OpenGL ES Programming Guide for iOS*.

Workflow Basics

For a detailed look at your app's OpenGL ES usage, you capture the sequence of OpenGL ES 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 in the debug bar or choose Debug > Capture OpenGL ES Frame.

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

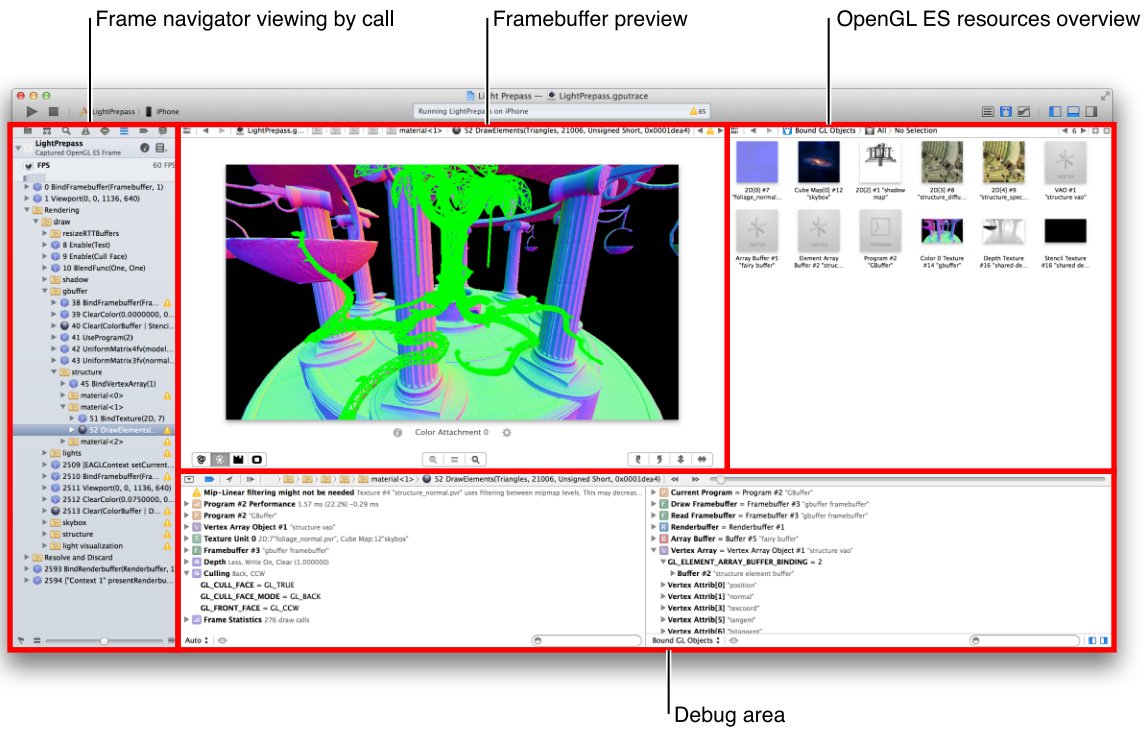
- *Breakpoint action.* Choose Capture OpenGL ES Frame as an action for any breakpoint. When the debugger reaches a breakpoint with this action, Xcode automatically captures a frame. If you use this action with an OpenGL ES Error breakpoint while developing your app, you can use the OpenGL ES Frame Debugger 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, 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 OpenGL ES Frame Debugger interface. You use this interface to inspect the sequence of graphics commands that render the frame and examine graphics resources.

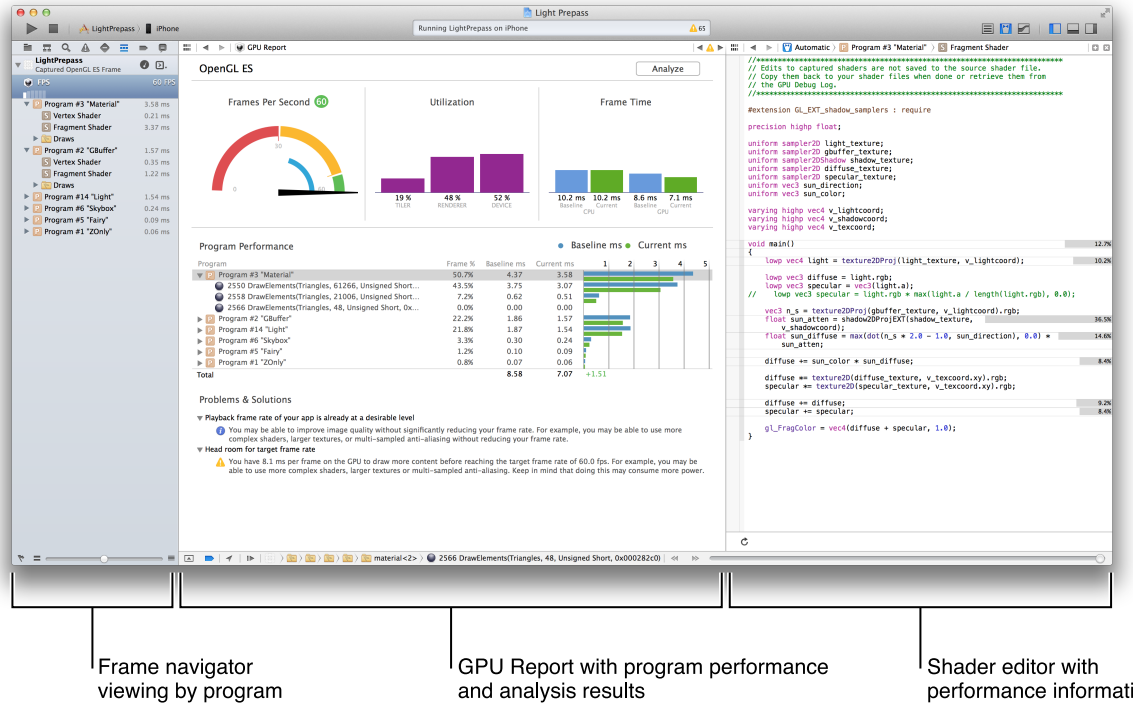
The OpenGL ES Frame Debugger UI

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

Examining draw calls and resources:

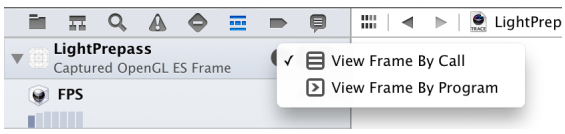


Examining shader program performance and analysis results:



Navigator Area

In the frame debugger interface, the debug navigator is 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.

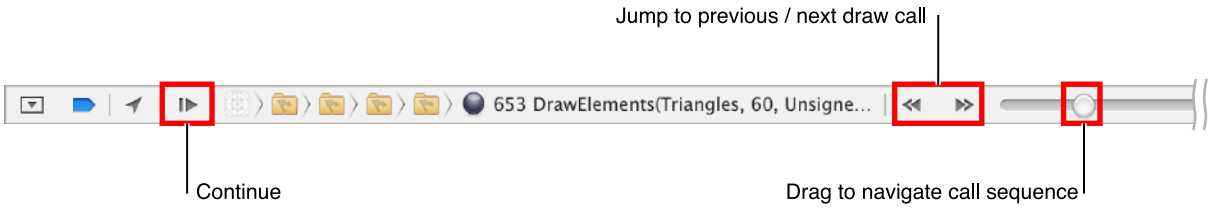
Clicking a program or shader shows the corresponding GLSL source code in the primary editor. Clicking a command navigates to that point in the frame capture sequence.

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 OpenGL ES resources and edit GLSL shader programs. By default, the assistant editor shows a graphical overview of all resources currently owned by the OpenGL ES 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 application.



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 OpenGL ES 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

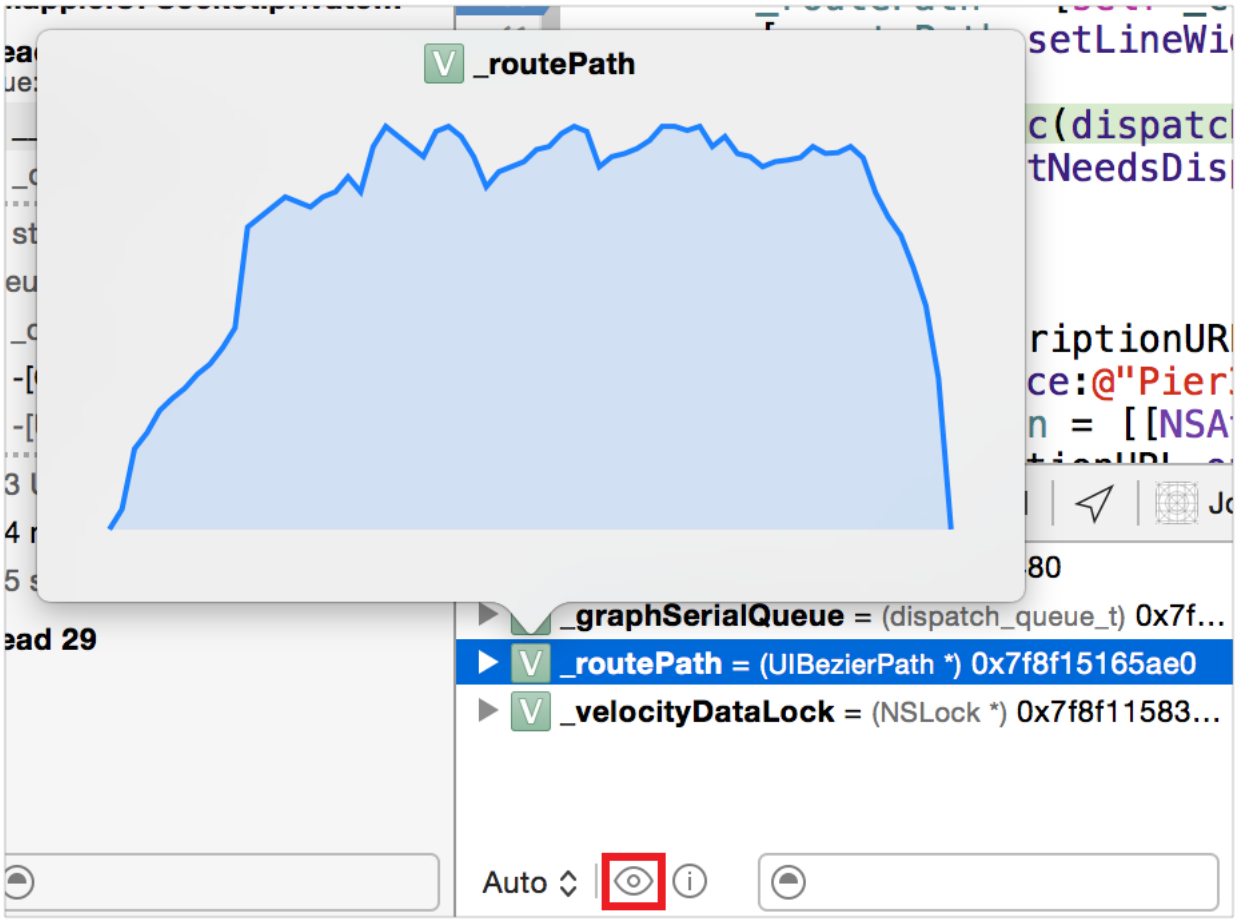
The Auto view automatically lists a subset of items normally found in the other variables views and other information appropriate to the selected call in the frame navigator. For example:

- 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 only available 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.

Quick Look Data Types

The Xcode debugger includes the variables Quick Look feature, a way to view the current state of object variables in your app by displaying their contents graphically in a pop-up display. Quick Look lets you display a graphical rendering of object contents by clicking the Quick Look button in the debugger variables view or by pressing the Space bar with a variable selected.



Not all operating system object types can present quick looks, but many can. The following table lists common object types provided by the operating system that support Quick Look.

Class Category	Object Type

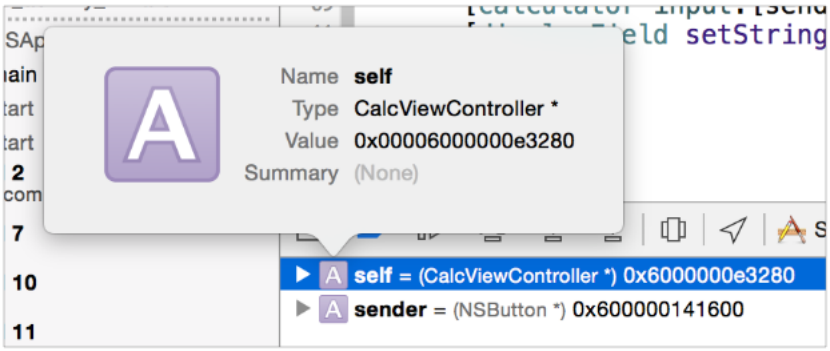
Image classes	`NSImage`
	`UIImage`
	`NSImageView`
	`UIImageView`
	`CIImage`
	`NSBitmapImageRep`

Cursor class	`NSCursor`
Color classes	`NSColor` `UIColor`
BezierPath classes	`NSBezierPath` `UIBezierPath`
Location classes	`CLLocation`
View classes	`NSView` `UIView`
String class	`NSString`
AttributedString class	`NSAttributedString`
Data class	`NSData`
URL class	`NSURL`

Class Category	Object Type

SpriteKit classes	`SKSpriteNode`
	`SKShapeNode`
	`SKTexture`
	`SKTextureAtlas`

You can always try the Quick Look feature if the object type isn't in this list. If Quick Look isn't available for the object type you've selected, Xcode shows the default quick look display for that object.



You can extend Quick Look for use with custom object types by adding a rendering method to the object class. See *Quick Look for Custom Types in the Xcode Debugger* to learn how to implement Quick Look in your custom object classes.

Document Revision History

This table describes the changes to *Debugging with Xcode*.

Date	Notes
2015-06-30	A new book based on Xcode 6 debugging features.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010