Qt DOCUMENTATION

Search

Topics >

Qt Creator Manual > Profiling QML Applications

Profiling QML Applications

You can use the QML Profiler to find causes for typical performance problems in your applications, such as slowness and unresponsive, stuttering user interfaces. Typical causes include executing too much JavaScript in too few frames. All JavaScript must return before the GUI thread can proceed, and frames are delayed or dropped if the GUI thread is not ready.

Another typical cause for similar performance problems is creating, painting, or updating invisible items, which takes time in the GUI thread.

Triggering long-running C++ functions, such as paint methods and signal handlers, also takes time in the GUI thread, but is more difficult to see in the QML Profiler because it does not profile C++ code. To find excessive use of JavaScript, check the frame rate in animations and Scene Graph events, look for gaps, and check whether the application behaves as expected. The JavaScript category displays the run time of functions, which you should try to keep below 16 ms per frame.

To find problems caused by handling invisible items, look for dropped frames and check that you are not using too many short bindings or signal handlers that are updated per frame. You can also visualize Scene Graph overdraw to check scene layout and find items that are never visible to the users because they are located outside the screen or hidden beneath other, visible elements.

If frames get dropped even though JavaScript is not being run, and there are large, unexplained gaps in the timeline, check your custom QQuickItem implementations. You can use Valgrind or other general purpose profilers to analyze C++ code.

You can use *full stack tracing* to trace from the top level QML or JavaScript down to the C++ and all the way to the kernel space. You can view the collected data in the Chrome Trace Format Viewer.

Using QML Profiler

To monitor the performance of an application in the QML Profiler:

1. To be able to profile an application, you must set up QML debugging for the project. For more information, see Setting Up QML Debugging.
2. In the **Projects** mode, select a kit with Qt version 4.7.4 or later.

> **Note:** To profile applications on devices, you must install Qt libraries on them.

3. Select **Analyze** > **QML Profiler** to profile the current application.
4. Select the ▶ (**Start**) button to start the application from the QML Profiler.

Qt DOCUMENTATION

When you start analyzing an application, the application is launched, and the QML Profiler immediately begins to collect data. This is indicated by the time running in the **Elapsed** field.

Data is collected until you select the **Enable Profiling** button. Data collection takes time, and therefore, there might be a delay before the data is displayed.

Do not use application commands to exit the application because data is sent to the QML Profiler when you select the **Enable Profiling** button. The application continues to run for some seconds, after which it is stopped automatically. If you exit the application, the data is not sent.

Select the **Disable Profiling** button to disable the automatic start of the data collection when an application is launched. Data collection starts when you select the button again.

To save all the collected data, select **Analyze** > **QML Profiler Options** > **Save QML Trace**. To view the saved data, select **Load QML Trace**. You can also deliver the saved data to others for examination or load data saved by them.

Specifying Flushing Settings

You can specify flushing settings for the QML Profiler either globally for all projects or separately for each project. To specify global settings, select **Edit** > **Preferences** > **Analyzer**.

To specify custom QML Profiler settings for a particular project, select **Projects** > **Run** and then select **Custom** in **QML Profiler Settings**. To restore the global settings for the project, select **Restore Global**.
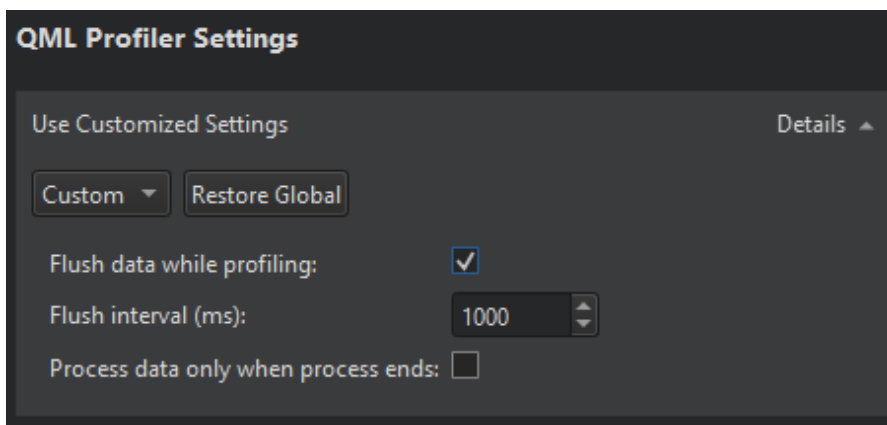
Select the **Flush data while profiling** check box to flush the data periodically instead of flushing all data when profiling stops. This saves memory on the target device and shortens the wait between the profiling being stopped and the data being displayed.

In the **Flush interval** field, set the flush interval in milliseconds. The shorter the interval, the more often the data is flushed. The longer the interval, the more data has to be buffered in the target application, potentially wasting memory. However, the flushing itself takes time, which can distort the profiling results.
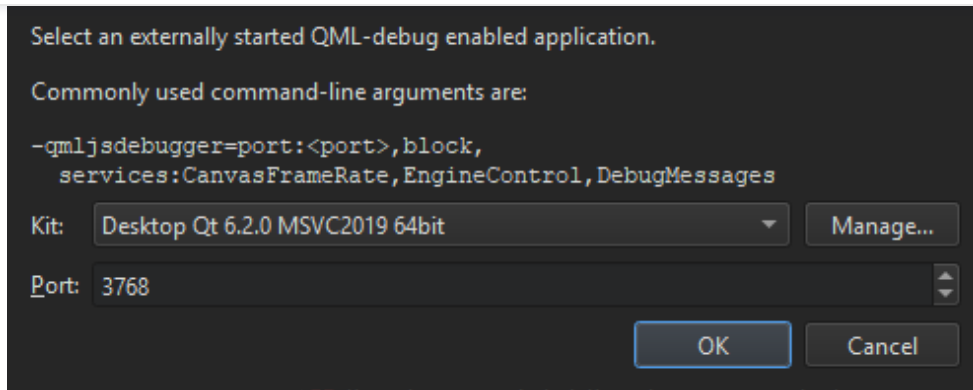
If you have multiple QML engines and you want to aggregate the data produced by all of them into one trace, select the **Process data only when process ends** check box. Otherwise, the profiling stops when one of the engines stops.

Attaching to Running Qt Quick Applications

You can profile Qt Quick applications that are not launched by Qt Creator. However, you must enable QML debugging and profiling for the application in the project build settings. For more information, see Setting Up QML Debugging.

To attach to waiting applications:

1. Select **Analyze** > **QML Profiler (Attach to Waiting Application)**.

Select an externally started QML-debug enabled application.

Commonly used command-line arguments are:

```
-qmljsdebugger=port:<port>,block,
    services:CanvasFrameRate,EngineControl,DebugMessages
```

Kit: Desktop Qt 6.2.0 MSVC2019 64bit   ▼   Manage...

Port: 3768

                                            OK   Cancel

2. In **Kit**, select the kit used to build the application.

3. In **Port**, specify the port to listen to.

4. Select **OK**.

Analyzing Collected Data

The **Timeline** view displays graphical representations of QML and JavaScript execution and a condensed view of all recorded events.

Qt DOCUMENTATION
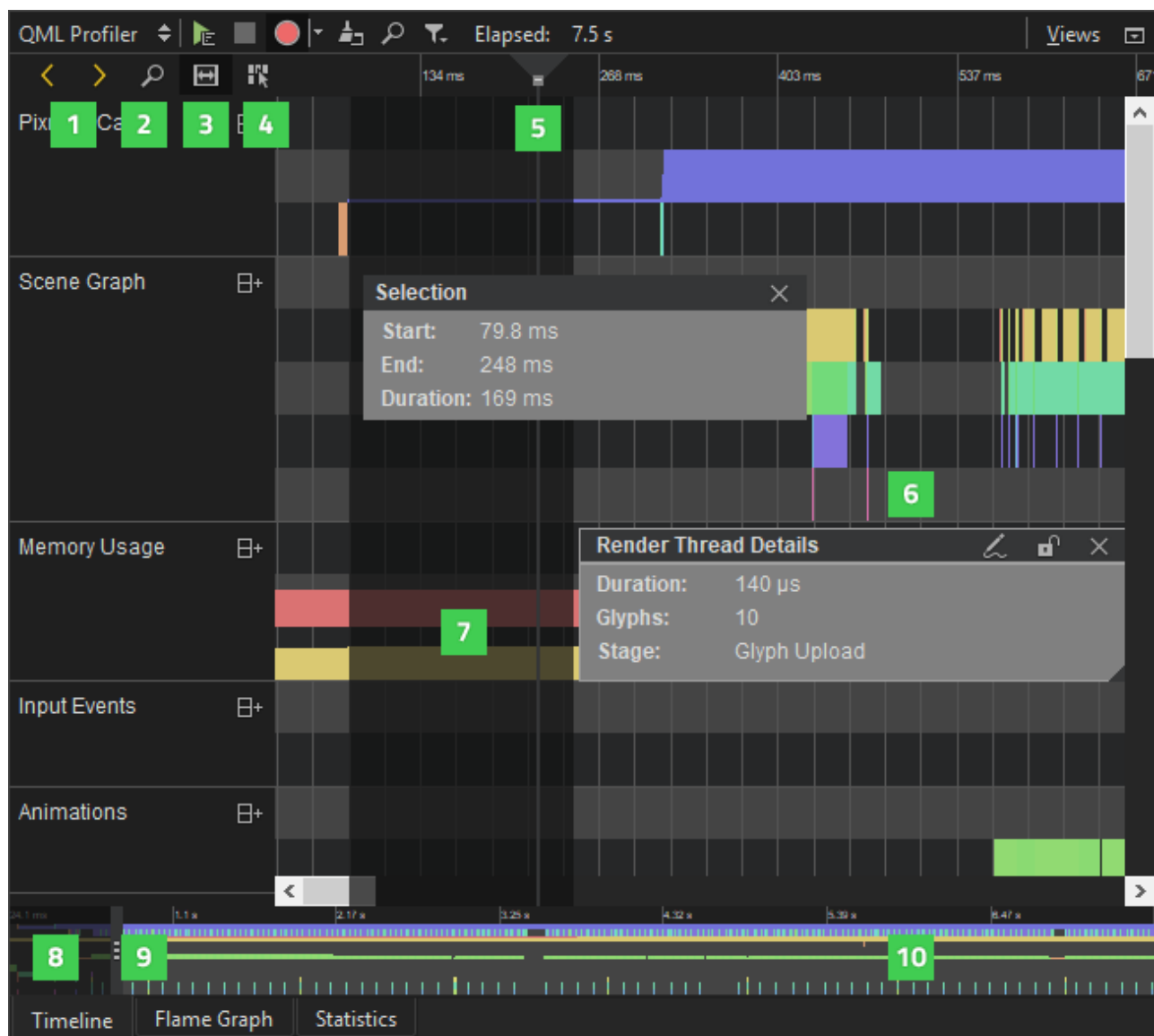
disable the **View Event Information on Mouseover** button (4).

The outline (10) summarizes the period for which data was collected. Drag the zoom range (8) or click the outline to move on the outline. You can also move between events by selecting the **Jump to Previous Event** and **Jump to Next Event** buttons (1).

Select the **Show Zoom Slider** button (2) to open a slider that you can use to set the zoom level. You can also drag the zoom handles (9). To reset the default zoom level, right-click the timeline to open the context menu, and select **Reset Zoom**.

Click the time ruler to add vertical orientation lines (5) to the timeline.

Selecting Event Ranges

You can select an event range (7) to view the frame rate of events and to compare it with the frame rate of similar events. Select the **Select Range** button (3) to activate the selection tool. Then click in the timeline to specify the beginning of the event range. Drag the selection handle to define the end of the range. The length of the range indicates the frame rate of the event.

You can use event ranges also to measure delays between two subsequent events. Place a range between the end of the first event and the beginning of the second event. The **Duration** field displays the delay between the events in milliseconds.

To zoom into an event range, double-click it.

To narrow down the current range in the **Timeline**, **Statistics**, and **Flame Graph** views, right-click the range and select **Analyze Current Range**. To return to the full range, select **Analyze Full Range** in the context menu.

To remove an event range, close the **Selection** dialog.

Understanding the Data

Generally, events in the timeline view indicate how long QML or JavaScript execution took. Move the mouse over them to see details. For most events, they include location in source code, duration and some relevant parts of the source code itself.

You can click on an event to move the cursor in the code editor to the part of the code the event is associated with.

The following types of events are displayed in the timeline view on one or several rows. The availability of event types depends on the version of Qt the application was compiled with and the version of Qt Quick it is using.

Event Category	Description	Minimum Qt Version	Qt Quick Version
Pixmap Cache	Displays the general amount of pixmap data cached, in pixels. In addition, displays a separate event for each picture being loaded, with specifics about its file name and size.	Qt 5.1	Qt Quick 2
Scene Graph	Displays the time when scene graph frames are rendered and some additional timing information for the various stages executed to do so.	Qt 5.1	Qt Quick 2
Memory Usage	Displays block allocations of the JavaScript memory manager. Generally, the memory manager will reserve larger blocks of memory in one piece and later hand them out to the application in smaller bits. If the application requests single blocks of memory surpassing a certain size, the memory manager will allocate those	Qt 5.4	Qt Quick 2
Event Category	separately. Those two modes of operation are shown as events of different colors. The second row displays the actual usage of the allocated memory. This is the amount of JavaScript heap the application has actually requested.	Minimum Qt Version	Qt Quick Version

Qt DOCUMENTATION

			Quick 2
Painting	Displays the time spent painting the scene for each frame.	Qt 4.7.4	Qt Quick 1
Animations	Displays the amount of animations that are active and the frame rate that they are running at. Information about render thread animations is displayed for applications that are built with Qt 5.3 or later. Render thread animations are shown in a separate row then.	Qt 5.0 (Qt 5.3)	Qt Quick 2
Compiling	Displays the time spent compiling the QML files.	Qt 4.7.4	Qt Quick 1 or Qt Quick 2
Creating	Displays the time spent creating the elements in the scene. In Qt Quick 2, creation of elements takes place in two stages. The first stage is for the creation of the data structures, including child elements. The second stage represents the completion callbacks. Not all elements trigger completion callbacks, though. The stages are shown as separate events in the timeline. For Qt Quick 2 applications compiled with versions of Qt before 5.2.1 only the creation of top-level elements is shown, as single events.	Qt 4.7.4 (Qt 5.2.1)	Qt Quick 1 or Qt Quick 2
Binding	Displays the time when a binding is evaluated and how long the evaluation takes.	Qt 4.7.4	Qt Quick 1 or Qt Quick 2
Handling Signal	Displays the time when a signal is handled and how long the handling takes.	Qt 4.7.4	Qt Quick 1 or Qt Quick 2
JavaScript	Displays the time spent executing the actual JavaScript behind bindings and signal handlers. It lists all the JavaScript functions you may be using to evaluate bindings or handle signals.	Qt 5.3	Qt Quick 2
Quick3D	Displays the time spent rendering Qt Quick 3D frames, timing information for frame preparation and synchronization, particle system update times and particle update count, as well as texture and mesh memory allocations and memory consumption.	Qt 6.3	Qt Quick 3D

Analyzing Scene Graph Events

In order to understand the scene graph category, it's important to understand how the Qt Quick scene graph works. See Qt Quick Scene Graph and Qt Quick Scene Graph Default Renderer for a detailed description. The following events are reported in the **Scene Graph** category. Not all events are generated by all render loops. In the Windows and Basic render loops everything runs in the same thread and the distinction between GUI thread and render thread is meaningless.

If you set the environment variable QSG_RENDER_TIMING, you get a textual output of similar, but slightly different timings from the application being profiled. For easier orientation, the differences are listed below.

Event Type	Thread	Render Loop Types	Label in output of QSG_RENDER_TIMING	Description	Caveats
Polish Event Type	GUI Thread	Threaded, Basic, Windows Loop Types	polish Label in output of QSG_RENDER_TIMING	Final touch-up of items before they are rendered using Description QQuickItem::updatePolish().	Versions of Qt prior to Qt 5.4 reCavedtatso polish times for the basic

Qt DOCUMENTATION

Event Type	Thread	Render Loop Types	Label in output of QSG_RENDER_TIMING	Description	Caveats
GUI Thread Wait	GUI	Threaded	lock	Executing slots connected to the QQuickWindow::afterAnimating() signal and then locking the render thread's mutex before waiting on the same mutex at **GUI Thread Sync**. If this starts long before **Render Thread Sync**, there is *free* time in the GUI thread you could be using for running additional QML or JavaScript.	None
GUI Thread Sync	GUI	Threaded	blockedForSync	The time the GUI thread is blocked, waiting for the render thread.	None
Animations	GUI	Threaded, Windows	animations	Advancing animations in the GUI thread. The basic render loop does not drive animations in sync with the rendering. This is why no animation events will be shown when using the basic render loop. Watch the **Animations** category to see animation timing in this case.	None
Render Thread Sync	Render	Threaded, Basic, Windows	Frame rendered ... sync	Synchronizing the QML state into the scene graph using QQuickItem::updatePaintNode().	None
Render	Render	Threaded, Basic, Windows	Frame rendered ... render	Total time spent rendering the frame, including preparing and uploading all the necessary data to the GPU. This is the *gross* render time. Do not confuse it with the *net* **Render Render** time below.	With versions of Qt prior to Qt 5.5, the gross render time and the below breakup of render times may be misaligned by some microseconds due to different, unsynchronized timers being used to measure them. For example **Render Preprocess** might seem to start before **Render Thread Sync** is finished.
Swap	Render	Threaded, Basic, Windows	Frame rendered ... swap	Swapping frames after rendering.	The output of swap times triggered by setting QSG_RENDER_TIMING is incorrect for the basic render loop and versions of Qt prior to Qt 5.4. The QML profiler shows the correct swap times.
Event Type	Thread	Render Loop Types	Label in output of QSG_RENDER_TIMING	Description	Caveats
Render	Render	Threaded	time in renderer	Calling QSGNode::preprocess()	May not be properly

Qt DOCUMENTATION

					gross **Render** step.	prior to Qt 5.5.
Render Update	Render	Threaded, Basic, Windows	time in renderer ... updates		Iterating and processing all the nodes in the scene graph to update their geometry, transformations, opacity, and other state. In the **Render Thread Sync** stage, each node is updated separately with state from the GUI thread. In **Render Update**, all the nodes are combined to create the final scene. This is part of the gross **Render** step.	May not be properly aligned with **Render** with versions of Qt prior to Qt 5.5.
Render Bind	Render	Threaded, Basic, Windows	time in renderer ... binding		Binding the correct framebuffer for OpenGL rendering. This is part of the gross **Render** step.	May not be properly aligned with **Render** with versions of Qt prior to Qt 5.5.
Render Render	Render	Threaded, Basic, Windows	time in renderer ... rendering		The actual process of sending all the data to the GPU via OpenGL. This is part of the gross **Render** step.	May not be properly aligned with **Render** with versions of Qt prior to Qt 5.5.
Material Compile	Render	Threaded, Basic, Windows	shader compiled		Compiling GLSL shader programs.	None
Glyph Render	Render	Threaded, Basic, Windows	glyphs ... rendering		Rendering of font glyphs into textures.	Versions of Qt prior to Qt 5.4 report incorrect times for these events.
Glyph Upload	Render	Threaded, Basic, Windows	glyphs ... upload		Uploading of glyph textures to the GPU.	Versions of Qt prior to Qt 5.4 report incorrect times for these events.
Texture Bind	Render	Threaded, Basic, Windows	plain texture ... bind		Binding a texture in the OpenGL context using glBindTextures.	None
Texture Convert	Render	Threaded, Basic, Windows	plain texture ... convert		Converting the format and downscaling an image to make it suitable for usage as a texture.	None
Texture Swizzle	Render	Threaded, Basic, Windows	plain texture ... swizzle		Swizzling the texture data on the CPU if necessary.	None
Texture Upload	Render	Threaded, Basic, Windows	plain texture ... upload / atlastexture uploaded		Uploading the texture data to the GPU.	None
Texture Mipmap	Render	Threaded, Basic, Windows	plain texture ... mipmap		Mipmapping a texture on the GPU.	None
Texture Delete	Render	Threaded, Basic, Windows	plain texture deleted		Deleting a texture from the GPU that became unnecessary.	None

Qt DOCUMENTATION

The following is the list of events for Qt Quick 3D. Each rendered frame consists of synchronize, prepare, and render phases, which are done in that order. Synchronize happens in scene graph synchronizing phase, while prepare and render happen in scene graph rendering phase.

Setting the environment variable QSG_RENDERER_DEBUG=render will also give additional textual output of render call counts of different rendering passes. These call counts are summed up in the Render Frame event.

Event Type	Thread	Description
Render Frame	Render	Render time of a frame. Also shows the number of draw calls.
Prepare Frame	Render	Time taken for preparing a frame. Resources are allocated and loaded in the prepare phase. The first frame after scene loading usually takes longer than others since most resources are loaded at that time.
Synchronize Frame	Render	Synchronize time of a frame. Synchronize takes care of updating backend values from the frontend. Also manages shared resources between Qt Quick Scene Graph and Qt Quick 3D.
Mesh Load	Render	Load time of a mesh. Shows total memory usage of all meshes. Also shows unloads.
Custom Mesh Load	Render	Load time of a custom mesh. Shows total memory usage of all meshes. Also shows unloads.
Texture Load	Render	Load time of a texture. Shows total memory usage of all textures. Also shows unloads.
Generate Shader	Render	Time for generating a shader for a material.
Load Shader	Render	Time for loading a built-in shader.
Particle Update	GUI	Update time of a particle system. Shows the number of particles updated.
Mesh Memory Consumption	Render	Shows a bar view of total mesh memory consumption.
Texture Memory Consumption	Render	Shows a bar view of total texture memory consumption.

Viewing Statistics

The **Statistics** view displays the number of times each binding, create, compile, JavaScript, or signal event is triggered and the average time it takes. This allows you to examine which events you need to optimize. A high number of occurrences might indicate that an event is triggered unnecessarily. To view the median, longest, and shortest time for the occurrences, select **Extended Event Statistics** in the context menu.

Click on an event to move to it in the source code in the code editor.

Location	Type	Time in Percent	Total Time	Self Time in Percent	Self Time	Calls	Mean Time	Details
<program>		100 %	304 ms	0.00 %	0 ns	1	304 ms	Main program
clocks.qml:59	Creating	78.51 %	238 ms	0.15 %	466 µs	2	119 ms	QtQuick/ListV
Clock.qml:141	Creating	77.87 %	236 ms	77.87 %	236 ms	16	14.8 ms	QtQuick/Text
clocks.qml:0	Compiling	14.19 %	43.1 ms	12.63 %	38.3 ms	1	43.1 ms	clocks.qml
Clock.qml:87	Handling ...	4.28 %	13 ms	0.42 %	1.28 ms	553	23.5 µs	onTriggered: c
Clock.qml:87	JavaScript	3.86 %	11.7 ms	0.70 %	2.13 ms	553	21.2 µs	expression for
Clock.qml:77	JavaScript	3.16 %	9.58 ms	2.21 %	6.72 ms	553	17.3 µs	timeChanged

Caller	Type	Total Time	Calls	Caller Description	Callee	Type	Total Time	Calls	Callee Description
<program>		238 ms	1	Main Program	Clock.qml:141	Creating	235 ms	4	QtQuick/Text
clocks.qml:54	Creating	224 µs	1	QtQuick/Rectangle	Clock.qml:95	Creating	671 µs	4	QtQuick/Image
					Clock.qml:94	Creating	550 µs	4	QtQuick/Image
					clocks.qml:67	Creating	464 µs	4	Clock.qml
					Clock.qml:137	Creating	147 µs	4	QtQuick/Image

The **Callers** and **Callees** panes show dependencies between events. They allow you to examine the internal functions of the application. The **Callers** pane summarizes the QML events that trigger a binding. This tells you what caused a change in a binding. The **Callees** pane summarizes the QML events that a binding triggers. This tells you which QML events are affected if you change a binding.

Click on an event to move to it in the source code in the code editor.

When you select an event in the **Timeline** view, information about it is displayed in the **Statistics** and **Flame Graph** views.

To copy the contents of one view or row to the clipboard, select **Copy Table** or **Copy Row** in the context menu.

JavaScript events are shown in the **Statistics** view only for applications that use Qt Quick 2 and are compiled with Qt 5.3 or later.

Visualizing Statistics as Flame Graphs

The **Flame Graph** view shows a more concise statistical overview of QML and JavaScript execution. In the **Total Time** view, the horizontal bars show the amount of time all invocations of a certain function took together, relative to the total runtime of all JavaScript and QML events. The nesting shows which functions were called by which other ones.
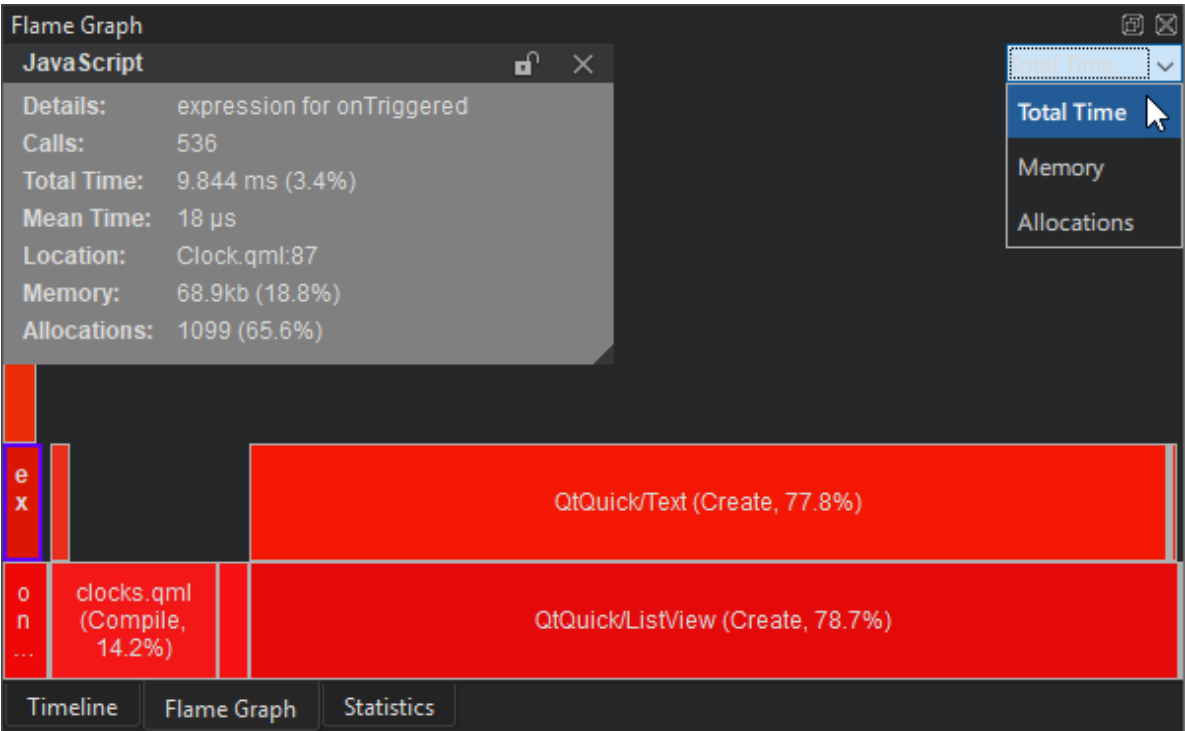
To view the total amount of memory allocated by the functions, select **Memory** in the drop-down menu.

To view the the number of memory allocations performed by the functions, select **Allocations**.

Double-click an item in a view to zoom into it. Double-click an empty area in the view to zoom out again.

Unlike the **Timeline** view, the **Flame Graph** view does not show the time spans when no QML or JavaScript is running at

Qt DOCUMENTATION

Qt The Qt Company

Contact Us

Company

About Us

Investors

Newsroom

Careers

Office Locations

Licensing

Terms & Conditions

Open Source

FAQ

Support

Support Services

Professional Services

Partners

Training

For Customers

Support Center

Downloads

Qt Login

Contact Us

Customer Success

Community

Contribute to Qt

Forum

Wiki

Downloads

Marketplace

© 2022 The Qt Company

Feedback   Sign In

Qt DOCUMENTATION