Using The QML Profiler

Ulf Hermann

The Qt Company

October 8, 2014 / Qt Developer Days 2014
Outline

1. Reasons for Using a Specialized Profiler for Qt Quick
2. The QML Profiler
3. Analysing Typical Problems
4. Live Examples of Profiling and Optimization
5. New Features for the QML Profiler
Outline

1 Reasons for Using a Specialized Profiler for Qt Quick

2 The QML Profiler

3 Analysing Typical Problems

4 Live Examples of Profiling and Optimization

5 New Features for the QML Profiler
Classical optimization workflow

Minimize total time a program will take to run:

- Instrument binary to count and time function calls
- Or use an emulator that keeps track of function calls

Create call statistics to see:

- which functions took most time
- which functions are called most often

Go back and optimize those.

Problematic with Qt Quick applications ...
Profiling JIT-compiled code

- What functions does it call there?
- No useful results on JIT-compiled or interpreted code from general purpose profilers
- No symbolic information available
- No stack unwinding with non-emulating profilers

Profiling QML code with Valgrind
“Long” run time

Single Signal Handler that runs for 40ms
- doesn’t make big dent in statistics
- leads to 2 dropped frames in a row
- might be harmless
- when does it run?

Relate single events on a timescale to pin down problems.
“Many” calls

- Time for each object creation isn’t significant here.
- Number of calls may be more interesting, but ...
- their **distribution over the frames** is most important!
Outline

1. Reasons for Using a Specialized Profiler for Qt Quick
2. The QML Profiler
3. Analysing Typical Problems
4. Live Examples of Profiling and Optimization
5. New Features for the QML Profiler
The QML Profiler

In Analyze mode of Qt Creator

1. start/stop profiling
2. control execution directly or profile external process
3. switch recording on and off while application is running to receive traces for specific time ranges.
4. select event types to be recorded (Qt Creator 3.3+)
5. clear current trace

Save and load trace files from context menu.
Timeline View

- Pixmap Cache: slow loading or large pictures
- Animations, Scene Graph: composition of scene graph
- Memory Usage: JavaScript heap and garbage collector
- Binding, Signal Handling, JavaScript: QML/JavaScript execution times
Events View

- Statistical profile of QML/JavaScript
- For problems that lend themselves to the classical workflow
- Optimize the overall most expensive bits to get a general speedup
Outline

1. Reasons for Using a Specialized Profiler for Qt Quick
2. The QML Profiler
3. Analysing Typical Problems
4. Live Examples of Profiling and Optimization
5. New Features for the QML Profiler
It’s slow. What is wrong?

- **Too much JavaScript** executed in few frames?
  - **All** JavaScript must return before GUI thread advances
  - Frames delayed/dropped if GUI thread not ready
  - Result: Unresponsive, stuttering UI

- **Creating/Painting/Updating** invisible items?
  - Takes time in GUI thread
  - Same effect as “Too much JavaScript”

- **Triggering long running C++ functions?**
  - Paint methods, signal handlers, etc. triggered from QML
  - Also takes time in GUI thread
  - Harder to see in the QML profiler as C++ isn’t profiled
Too much Javascript

- Watch frame rate in Animations and Scene Graph
- Gaps and orange animation events are bad
- JavaScript category shows functions and run time
- Stay under $1000/60 \approx 16\text{ms}$ per frame
Invisible Items

- Check again for dropped frames
- Check for **many short bindings** or signal handlers
  => Too many items updated per frame
- **QSG_VISUALIZE=overdraw** shows scene layout
- Watch for items **outside the screen** or **underneath visible elements**
Long running C++ functions

- Dropped frames, but no JavaScript running?
- Large unexplained gaps in the timeline?
- Check your custom QQuickItem implementations
- Use general purpose profiler to explore the details
Outline

1. Reasons for Using a Specialized Profiler for Qt Quick
2. The QML Profiler
3. Analysing Typical Problems
4. Live Examples of Profiling and Optimization
5. New Features for the QML Profiler
Example 1: Too much JavaScript

Glitch in SameGame example when starting new game

- All items created from one JavaScript function call
- Takes about 100ms
- About 7 dropped frames in a row
- Enough unused CPU time during menu removal animation

Solution:
- Create invisible items during menu animation
- Later only set them visible
- Setting visibility is cheaper than creating items
Example 1: Too much JavaScript

Glitch in SameGame example when starting new game

- All items created from one JavaScript function call
- Takes about 100ms
- About 7 dropped frames in a row
- Enough unused CPU time during menu removal animation

Solution:

- Create invisible items during menu animation
- Later only set them visible
- Setting visibility is cheaper than creating items
Example 1: Too much JavaScript

Glitch in SameGame example when starting new game

- All items created from one JavaScript function call
- Takes about 100ms
- About 7 dropped frames in a row
- Enough unused CPU time during menu removal animation

Solution:

- Create *invisible* items during menu animation
- Later only set them visible
- Setting visibility is cheaper than creating items
Conventions for profiling Qt Creator

- gray color scheme: *profiling* one of the others
- red color scheme: *buggy pre-3.0* as “bad” example
- green color scheme: *v3.3 preview*
- blue color scheme: *patched v3.3 preview*
- Trace files are just loaded into “colored” Qt Creators to trigger activity. Don’t interpret the data.
Example 2: Even more JavaScript

QML Profiler stutters on horizontal resizing.

• Overview always iterates all events to paint itself
• is implemented in JavaScript
• but: only updated on loading and resizing

Solution
• Stretch the code over multiple frames
• Use Timer to trigger deferred JavaScript execution
• onTriggered should not take longer than a frame (around 16ms)
• Downside: Overview painting is “animated” now

1 with potential for further optimization...
Example 2: Even more JavaScript

QML Profiler stutters on horizontal resizing.

- Overview always iterates all events to paint itself
- is implemented in JavaScript
- **but:** only updated on loading and resizing

Solution

1. Stretch the code over multiple frames
2. Use Timer to trigger deferred JavaScript execution
3. onTriggered should not take longer than a frame (around 16ms)

Downside: Overview painting is "animated" now with potential for further optimization...
Example 2: Even more JavaScript

QML Profiler stutters on horizontal resizing.

- Overview always iterates all events to paint itself
- is implemented in JavaScript
- but: only updated on loading and resizing

Solution¹:

- **Stretch** the code over multiple frames
- Use **Timer** to trigger deferred JavaScript execution
- `onTriggered` should not take longer than a frame (around 16ms)
- Downside: Overview painting is “animated” now

¹ with potential for further optimization
Example 3: Painting outside viewport

Slow scrolling if timeline categories expanded

- Coordinate system marks cover a large space in vertical direction
- Can take a long time to paint (up to 10ms)
- Are mostly invisible most of the time.

Solution
- Only paint visible part of coordinate system
- Directly set virtual contentHeight on Flickable
- Painted area "sliding" in virtual contentHeight
- Reduces painting time to about 1 - 2ms

With potential for further optimization...
Example 3: Painting outside viewport

Slow scrolling if timeline categories expanded

- Coordinate system marks cover a large space in vertical direction
- can take a long time to paint (up to 10ms)
- are mostly invisible most of the time.

Solution

- Only paint visible part of coordinate system
- Directly set virtual contentHeight on Flickable
- Painted area "sliding" in virtual contentHeight
- Reduces painting time to about 1 - 2ms

With potential for further optimization...
Example 3: Painting outside viewport

Slow scrolling if timeline categories expanded

- Coordinate system marks cover a large space in vertical direction
- can take a long time to paint (up to 10ms)
- are mostly invisible most of the time.

Solution²:

- Only paint **visible part** of coordinate system
- Directly set virtual contentHeight on Flickable
- Painted area “sliding” in virtual contentHeight
- Reduces painting time to about 1 - 2ms

²with potential for further optimization
Example 4: Expensive C++

Timeline scrolling still slow for some traces
Example 4: Expensive C++

Timeline scrolling still slow for some traces

- Timeline data painted for all categories, no matter how many are visible
- Takes a lot of time, especially in “dense” places.
- Hard to see in QML Profiler, as painting is implemented in C++.
- `QSG_VISUALIZE=overdraw` can help.
Example 4: Expensive C++

Timeline scrolling still slow for some traces

- Timeline data painted for all categories, no matter how many are visible
- Takes a lot of time, especially in “dense” places.
- Hard to see in QML Profiler, as painting is implemented in C++.
- \texttt{QSG\textunderscore VISUALIZE=overdraw} can help.

Solution\textsuperscript{3}:

- Again, only paint \textit{visible} part of timeline.
- Same technique as with coordinate system.

\textsuperscript{3}with potential for further optimization
Example 5: What about the labels?

Hiccup when expanding large categories
Example 5: What about the labels?

Hiccup when expanding large categories

- **Repeater** creates all elements at the same time.
- Use **ListView** to create and delete on demand?
- Potentially save some memory?
Example 5: What about the labels?

Hiccup when expanding large categories

- **Repeater** creates all elements *at the same time*.
- Use **ListView** to create and delete *on demand*?
- Potentially save some memory?

But:

- Labels are *rarely updated*.
- On-demand creation and removal *during scrolling*, when a lot of other code has to run?
- Creation and removal *triggers garbage collector*.

Solution: Probably not worth it *in this case*
Outline

1 Reasons for Using a Specialized Profiler for Qt Quick
2 The QML Profiler
3 Analysing Typical Problems
4 Live Examples of Profiling and Optimization
5 New Features for the QML Profiler
Better Scene Graph Profiling

- Will be included in Professional and Enterprise packages of Qt Creator 3.3
- Visualizes all the timing information available from the scene graph
JavaScript Heap profiler

- UI in Qt Creator 3.2+ (Professional and Enterprise)
- Will be usable with Qt 5.4+
- Tracks page allocations of the memory manager
- Tracks memory allocations on JavaScript heap
- Shows when the garbage collector runs
Selective recording

- Switch off recording of events you’re not interested in
- Reduces amount of data created
- Record longer traces without running into memory bottlenecks
- Smaller trace files, faster loading
Various UI improvements

- Drag&Drop reordering of categories
- Completely hide categories to reduce height of timeline
- Resize rows in timeline