Performance Driven Development on Low-Cost Embedded Hardware

Increasing Return-On-Investment & shortening time-to-market

Nils Christian Roscher-Nielsen
Product Manager, The Qt Company

The Qt Company

Agenda

- Qt Quick 2D Renderer
  - What is it?
  - What pain does it address?
  - Demo
- Qt Quick Compiler
  - Reducing memory consumption
  - Improving start-up time
  - Deployment
- QML Profiler
- Conclusion

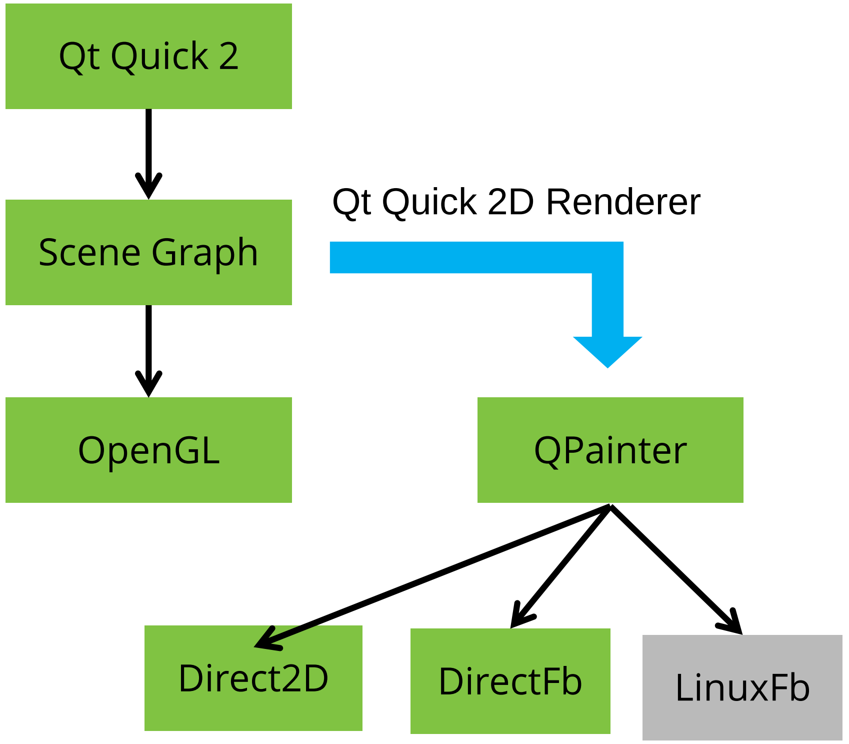
Qt Quick 2D Renderer

No GPU? No Problem

The Qt Company

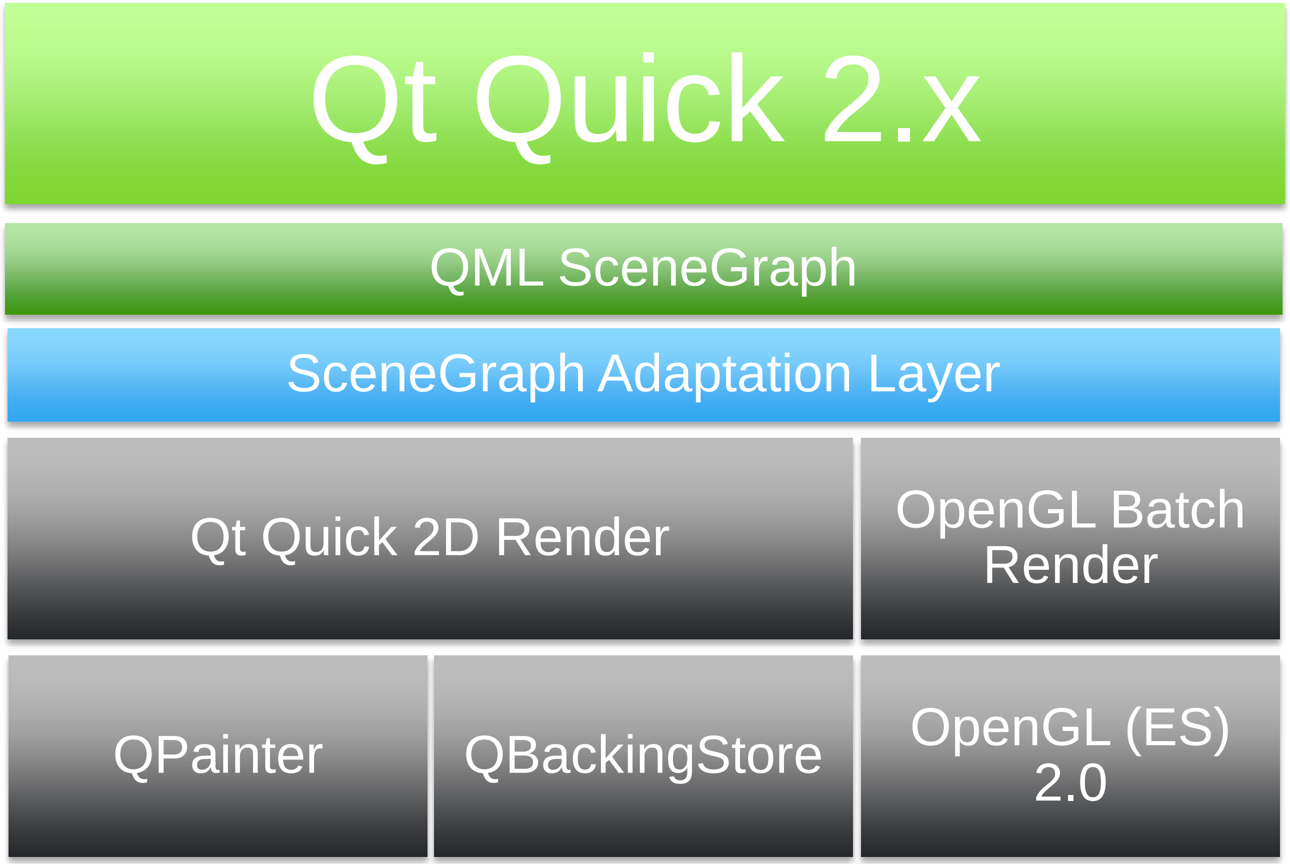
Qt Quick 2D renderer

- Renders Qt Quick without OpenGL
- Can render fully in Software
- Makes use of 2D Hardware acceleration
  - DirectFB (Linux)
  - Direct 2D (Windows)
  - Others possible



New enterprise add-on available from Qt 5.4

What is the Qt Quick 2D Renderer?



Problems Qt Quick 2D Renderer Addresses

- No GPU, or no OpenGL 2.0 support
- No requirement for:
  - Particles
  - 60 FPS animations
  - “Eye Candy”
- Same UI across device portfolio

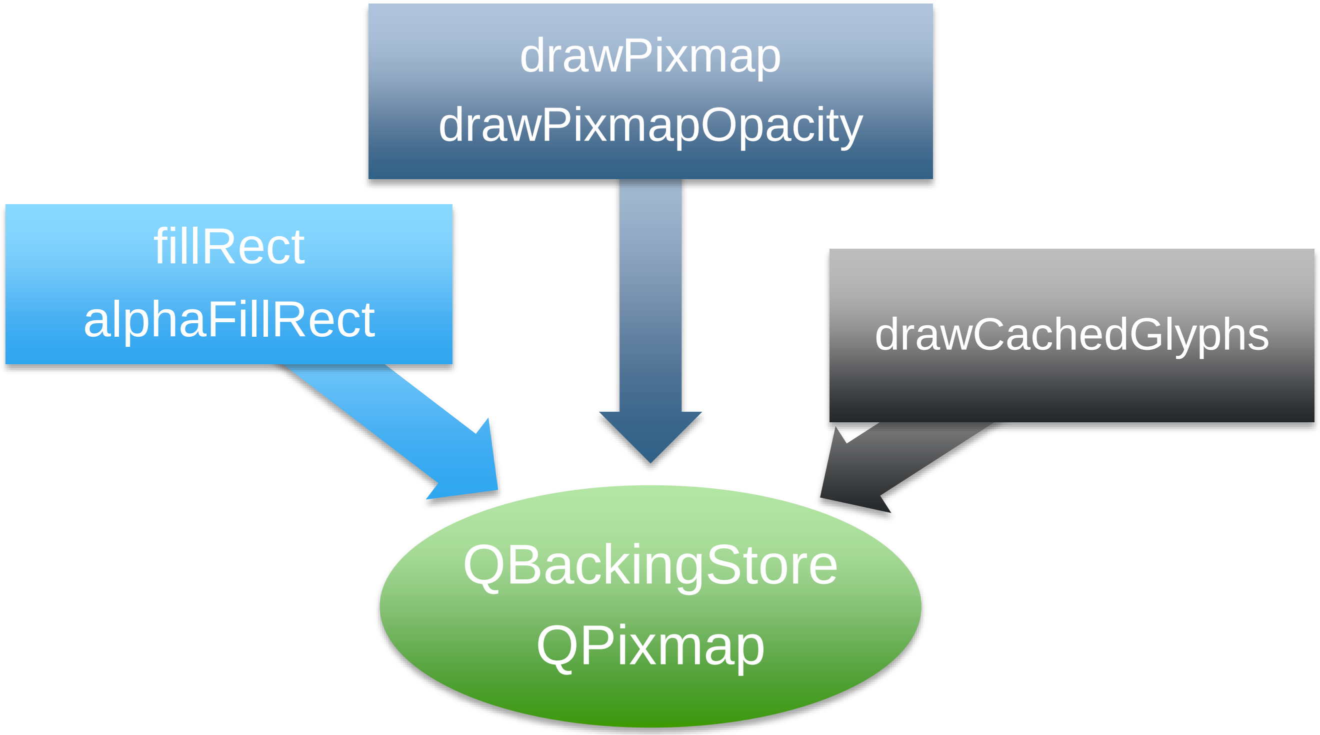
Using the Qt Quick 2D Renderer

- export QMLSCENE_DEVICE=softwarecontext

What about the OpenGL Dependency?

- Qt Quick 2 (QtDeclarative module) depends on OpenGL (ES) 2
- Build Qt 5 with support for OpenGL (desktop or ES2)
- Dummy OpenGL Libraries
  - OpenGL, EGL, and KHR headers
  - Libraries for libGLESv2.so and libEGL.so (only symbols)

2D Hardware Acceleration with QBlittable



Qt Quick 2 Limitations with the 2D Renderer

- ShaderEffect
- Particles
- Sprites
- Custom Items with OpenGL
- RenderControls (Qt 5.4+)

Not Just for Embedded

- Works anywhere QWidget works
  - Uses the same rendering path as QWidgets
  - No dependency on QWidgets Module
- Desktop Platforms
  - Windows (remove OpenGL dependency)
  - Linux (X11 Forwarding)
  - Remote Desktop

Demo time!

Future of 2D renderer

- OpenVG
  - Possible to add support
  - Not planned at the moment
  - Let us know if you are interested
- New Hardware
  - Moving beyond DirectFB
  - Partnering with HW vendors
  - Customer requests
- Many improvements to come
  - Close collaboration with users

Qt Quick Compiler

Optimizing Qt Quick Applications Ahead Of Time

The Qt Company

Background

- Qt Quick applications consist traditionally of a mixture of
  - C++ code
  - Declarative QML files with embedded, imperative JavaScript code
  - Additional resources such as PNG image files

Background

- C++ code gets compiled to native code in target architecture
- PNG image files and other resource get embedded into the resulting binary through the Qt Resource System
- QML source files need to be deployed verbatim to the target device

Background

- On application startup:
  - The QML engine parses QML source code
  - Compiles JavaScript code to native code on the fly:
    - Memory needs to be allocated for the generated code
    - Code generation consumes precious startup time
  - If target architecture is not supported by Just-in-time compiler, execution happens using slower interpreted byte-code

Introducing Qt Quick Compiler

- Allows for compilation of .qml and .js files in Qt Quick applications ahead of time
- Output is portable C++ code that is compiled alongside the application C++ code
- Embedded in the final application binary
- Requires a commercial Qt license

Qt Quick Compiler Outline

1. Reducing memory consumption
2. Improving start-up time
3. Simplifies deployment
4. Conclusion

Reducing Memory Consumption

- Just-in-time code generation requires the allocation of executable memory
- Memory is not shared – two processes loading the same .qml file have to allocate the memory for the embedded code twice.

Reducing Memory Consumption

- Regular application C++ code is compiled into sections in the executable
- Code sections are loaded on-demand from disk using mmap(). Executing the same program twice results in the compiled code being shared in memory.

Memory Consumption: Example Samegame

- Environment: Linux, armv7, release build
- Without Qt Quick Compiler:
    - smaps reports:
        - Private_Clean: 4076 kB
        - Private_Dirty: 12744 kB
- With Qt Quick Compiler:
    - Private_Clean: 4652 kB
    - Private_Dirty: 11976 kB
- ➔~600 kB memory became mmap()'able from disk
- Merely by flipping a switch in the build system

Qt Quick Compiler Outline

1. ~~Reducing memory consumption~~
2. Improving start-up time
3. Deployment
4. Conclusion

Just In Time Compilation on Start-Up

- Six-fold compilation process:
  - Parse QML/JavaScript Source into abstract syntax tree
  - Generate intermediate representation
  - Transform to SSA form
  - Perform optimizations (constant value propagation, etc.)
  - Transform out of SSA, perform register allocation
  - Generate native code from IR
- Time consuming, but important for performance

Qt Quick Compiler Compilation

- Entire compilation process happens at application build time
- Qt Quick Compiler generates C++ code
- Platform compiler optimizes code
- Platform compiler generates native code
- Transparent integration in the application development cycle

Start-up time: Same game example

- Platform: Linux, x86-64
- Counting instructions with callgrind (stable)
- Without Qt Quick Compiler:
  - ~461 Million Instructions for startup
- With Qt Quick Compiler:
  - ~339 Million Instructions for startup
- 27% instructions saved, merely by flipping a switch in the build system

Run-time Performance Implications

- Qt Quick Compiler generated code is about as fast as Just-In-Time compiler in average
  - Where JIT is available
- Just-In-Time compilation not supported on all platforms
- Fallback to byte-code interpreter on PowerPC, MIPS, etc.
- Qt Quick Compiler gives ~2x speed-up
  - Where JIT not available

Qt Quick Compiler Outline

1. ~~Reducing memory consumption~~
2. ~~Improving start-up time~~
3. Deployment
4. Conclusion

Application Deployment

- Qt Quick application need to ship with .qml files that are loaded on start-up and at run-time
- Anyone who has file system access to where your application is installed can see your proprietary source code
- .qml files can be embedded in the binary as resources, but they can still be extracted to plain text from there with little effort

Qt Quick Compiler Usage

Three steps to enable Qt Quick Compiler in your application:

1. Embed your .qml and .js files using the Qt Resource System
2. Convert your application to load your files using qrc:/ URLs
3. Toggle Qt Quick Compiler usage using CONFIG += qtquickcompiler on the command line or using check box in Qt Creator project build settings

- Cmake is also supported

Qt Quick Compiler Usage

- Qt Quick Compiler build system integration transparently removes .qml and .js source code from the Qt Resource System
- Generates C++ code
- No source code shipped with your application

Conclusion

- Port your Qt Quick application to use the Qt Resource system
- Easily toggle use of Qt Quick Compiler in your project
- The more binding expressions and JavaScript code, the greater the benefits of using the compiler

Qt Quick Profiler

Optimizing Qt Quick code performance

The Qt Company

Why do you need a Profiler

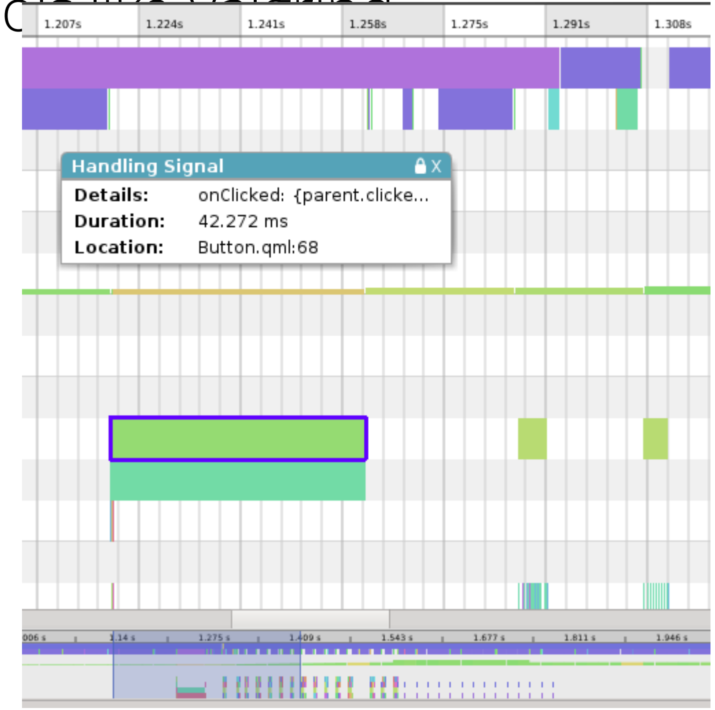
- Classic optimization
  - Minimize the total time a program takes
    - Instrument your binary to count and time function calls
    - Run in an emulator to keep track of function calls
  - Create Call statistics to see
    - Which functions took the most time
    - Which functions are called most often
  - Go back and optimize

- This is not always very helpful in a Qt Quick application

Challenges

- JIT compiled QML makes little sense in tools like Valgrind
  - Which functions are called?
  - No symbolic information available
  - Stack unwinding only with emulating profilers

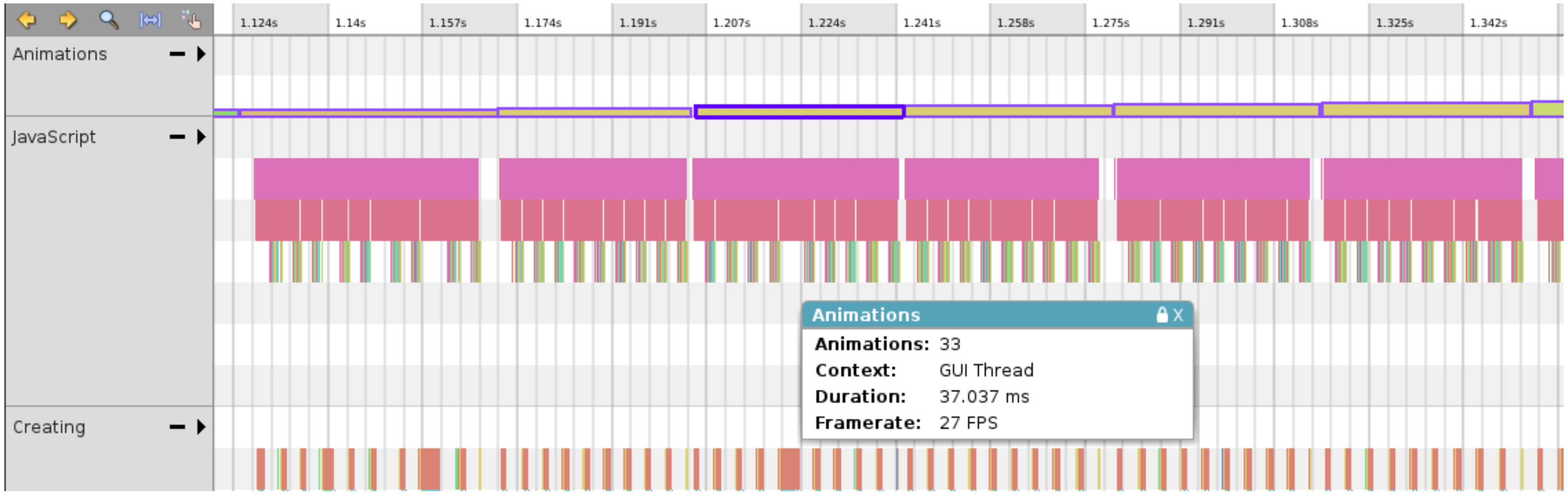
- Mainly statistical information
- 40 ms event handler
  - No big deal statistically
  - When it happens is important



Handling Signal

Details:	onClicked: {parent.clicke...
Duration:	42.272 ms
Location:	Button.qml:68

Challenges
"Many" Calls

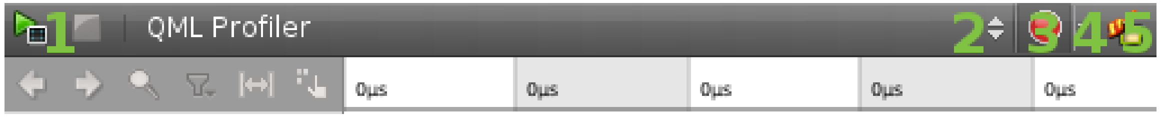


Animations

Animations:	33
Context:	GUI Thread
Duration:	37.037 ms
Framerate:	27 FPS

- Time for each object creation is not very important
- Number of calls a bit more interesting, but ...
- Their distribution over the time frames is most important

The QML Profiler



- Analyze mode in Qt Creator
  1. Start/Stop profiling
  2. Control execution directly or profile external process
  3. Switch recording on and off while the application is running
  4. Select event types to be recorded
  5. Clear current trace
  - Save and load traces from the context menu

Timeline View

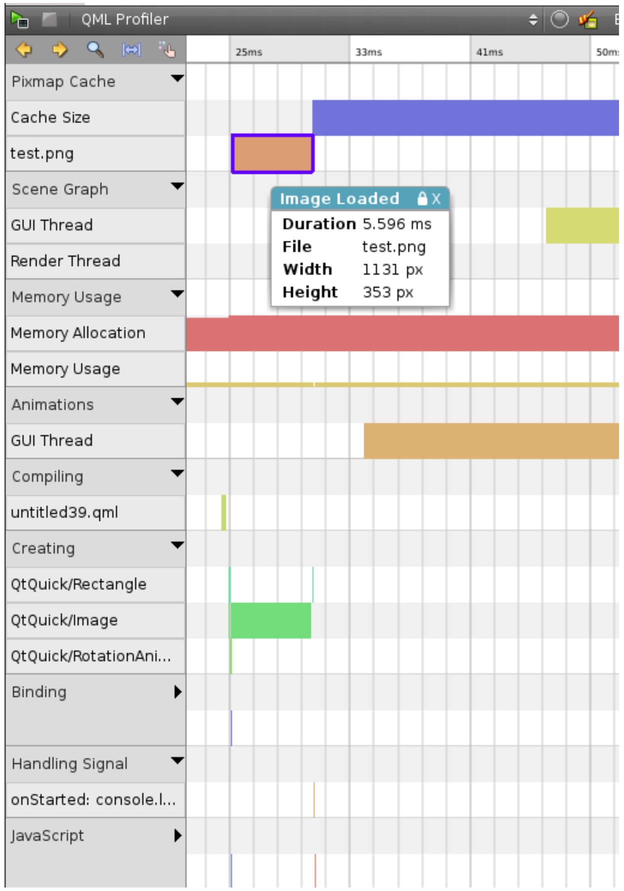
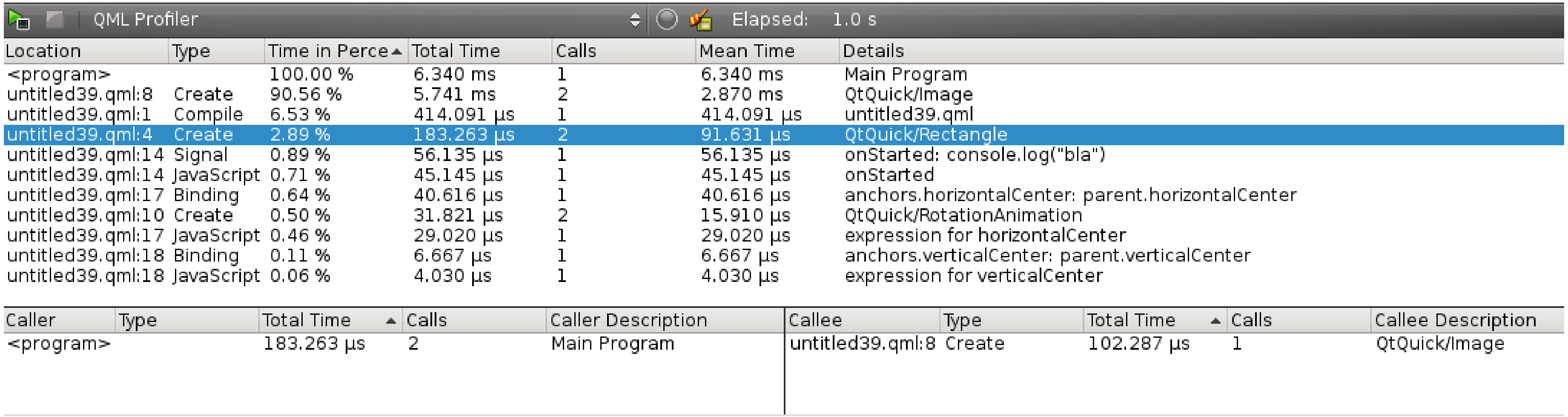


Image Loaded

Duration	5.596 ms
File	test.png
Width	1131 px
Height	353 px

- Pixmap Cache
  - Slow loading or large pictures
- Scene Graph, Animations
  - Composition of the Scene Graph
- Memory usage
  - JavaScript heap and garbage collector
- Binding, Signal Handling, JavaScript, etc.
  - QML and JavaScript execution time

Events View



Location	Type	Time in Percent	Total Time	Calls	Mean Time	Details
<program>		100.00 %	6.340 ms	1	6.340 ms	Main Program
untitled39.qml:8	Create	90.56 %	5.741 ms	2	2.870 ms	QtQuick/Image
untitled39.qml:1	Compile	6.53 %	414.091 µs	1	414.091 µs	untitled39.qml
untitled39.qml:4	Create	2.89 %	183.263 µs	2	91.631 µs	QtQuick/Rectangle
untitled39.qml:14	Signal	0.89 %	56.135 µs	1	56.135 µs	onStarted: console.log("bla")
untitled39.qml:14	JavaScript	0.71 %	45.145 µs	1	45.145 µs	onStarted
untitled39.qml:17	Binding	0.64 %	40.616 µs	1	40.616 µs	anchors.horizontalCenter: parent.horizontalCenter
untitled39.qml:10	Create	0.50 %	31.821 µs	2	15.910 µs	QtQuick/RotationAnimation
untitled39.qml:17	JavaScript	0.46 %	29.020 µs	1	29.020 µs	expression for horizontalCenter
untitled39.qml:18	Binding	0.11 %	6.667 µs	1	6.667 µs	anchors.verticalCenter: parent.verticalCenter
untitled39.qml:18	JavaScript	0.06 %	4.030 µs	1	4.030 µs	expression for verticalCenter

Caller	Type	Total Time	Calls	Caller Description
<program>		183.263 µs	2	Main Program

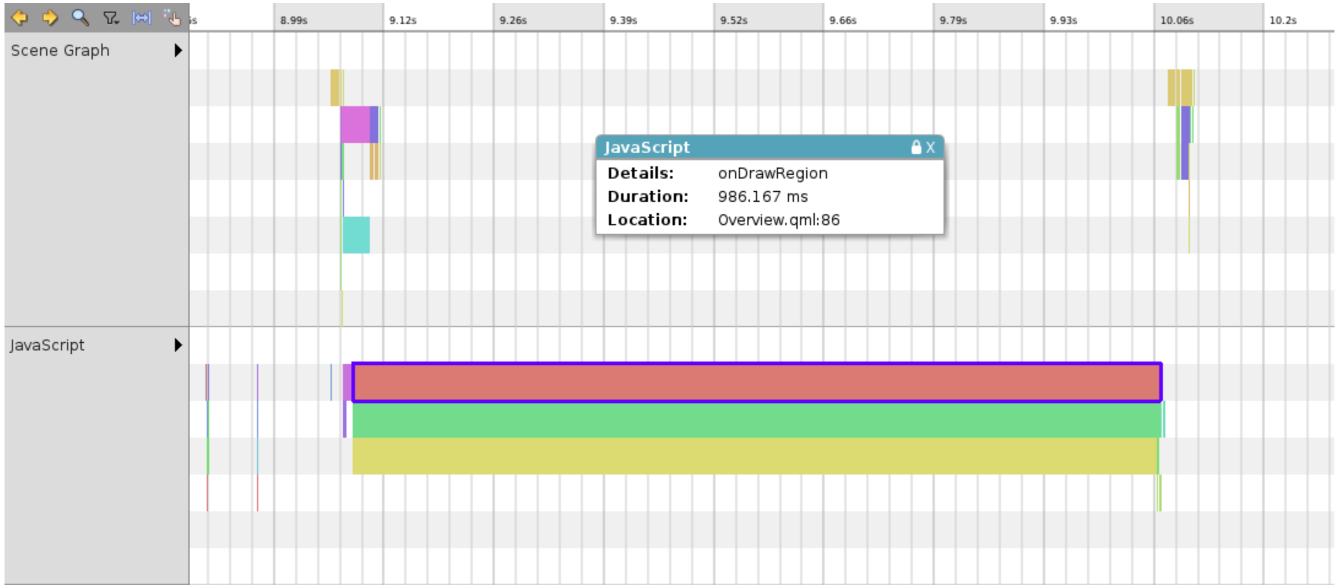
Callee	Type	Total Time	Calls	Callee Description
untitled39.qml:8	Create	102.287 µs	1	QtQuick/Image

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

My application is slow
What is wrong?

- Too much JavaScript in each frame
  - 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

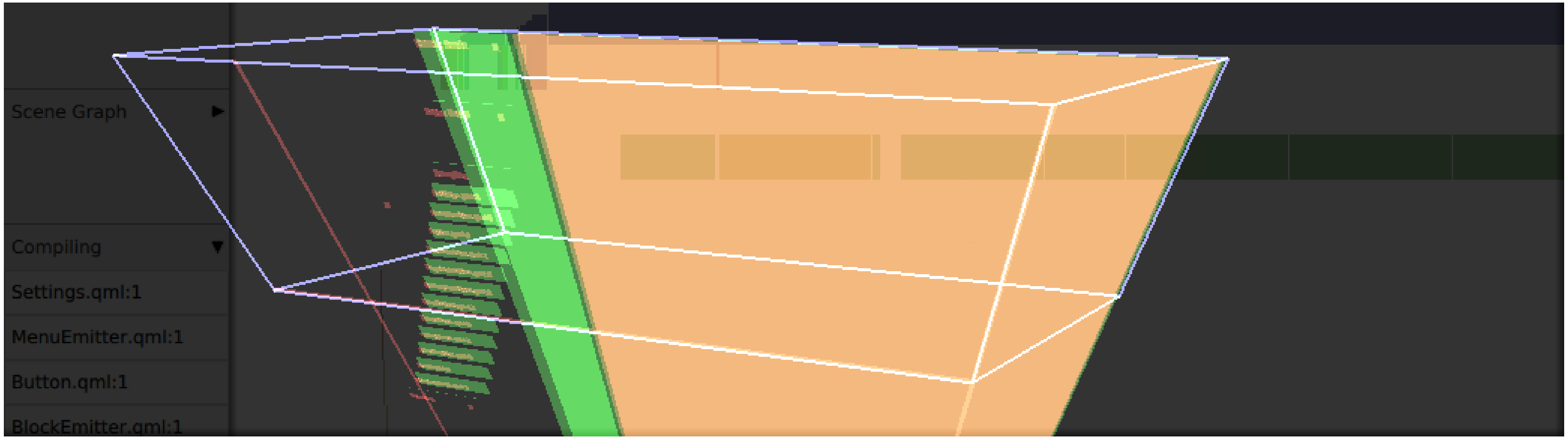


JavaScript

Details: onDrawRegion
Duration: 986.167 ms
Location: Overview.qml:86

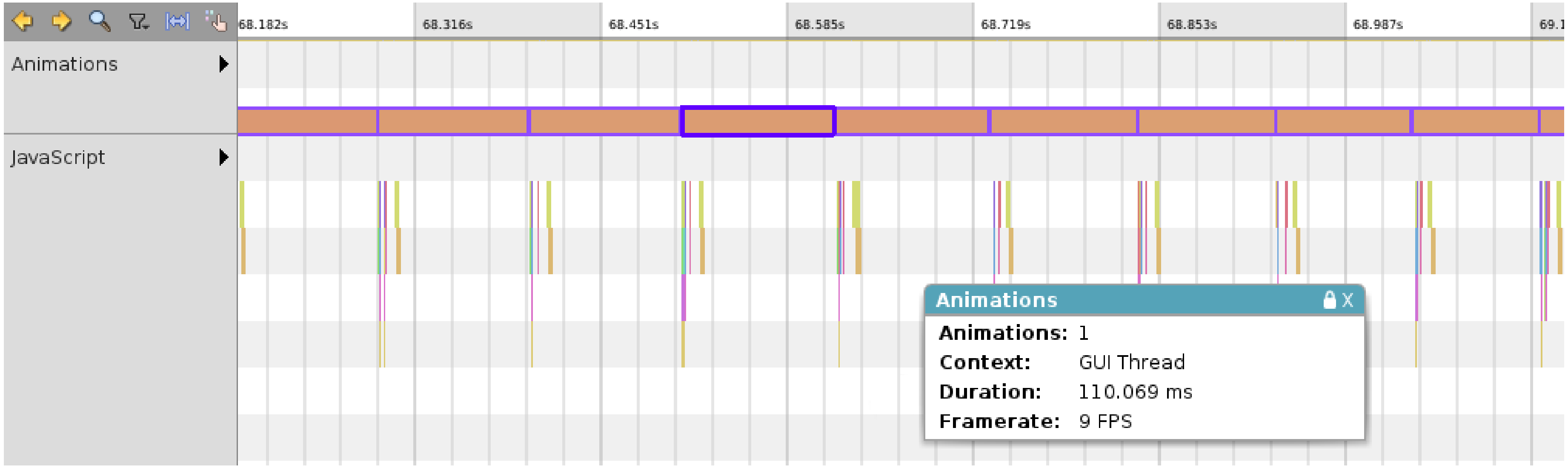
- Watch frame rate in Animations and Scene Graph
- Gaps and orange animation events are bad
- JavaScript category shows functions and run time
- Stay under 1000/60 ≈ 16ms per frame

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
- Are items outside the screen or underneath visible elements

Long Running C++ functions



Animations

Animations: 1
Context: GUI Thread
Duration: 110.069 ms
Framerate: 9 FPS

- 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

Conclusions

- Qt Quick 2D renderer
  - Target more devices, and low-power hardware
- Qt Quick Profiler
  - Improve code performance
  - Ensure smooth animations at all time
- Qt Quick Compiler
  - Shorter application start up time
  - Faster application execution
  - Lower application binary size
  - More secure and simple-to-deployable binaries

Thank you very much for attending today

☺

ありがとうございました

nils.roscher-nielsen@theqtcompany.com
@rosch