Analyzing Performance of QtQuick Applications

Thomas McGuire
KDAB

thomas@kdab.com
Performance: Multiple Aspects

- Startup Duration
- Smooth Rendering / Frames per Second
- Responsiveness
- Boot Duration
- Power Usage
- Memory Usage
Startup Time
## Startup Time - CPU Profiler

<table>
<thead>
<tr>
<th>Function Stack</th>
<th>CPU Time: Total by Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td></td>
</tr>
<tr>
<td>_libc_start_main</td>
<td>634.645ms</td>
</tr>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>QQmlEngine::rootContext</td>
<td>0ms</td>
</tr>
<tr>
<td>QQmlContext::setContextProperty</td>
<td>1.772ms</td>
</tr>
<tr>
<td>Storage::Storage</td>
<td></td>
</tr>
<tr>
<td>Storage::Storage</td>
<td>5.316ms</td>
</tr>
<tr>
<td>loadFonts</td>
<td>7.210ms</td>
</tr>
<tr>
<td>QGuiApplication::QGuiApplication</td>
<td>39.709ms</td>
</tr>
<tr>
<td>MainView::MainView</td>
<td>48.122ms</td>
</tr>
<tr>
<td>MainView::setMainQmlFile</td>
<td>81.629ms</td>
</tr>
<tr>
<td>QQuickView::setSource</td>
<td>412.104ms</td>
</tr>
<tr>
<td>QQuickViewPrivate::execute</td>
<td>412.104ms</td>
</tr>
<tr>
<td>QQmlComponent::QQmlComponent</td>
<td>0.595ms</td>
</tr>
<tr>
<td>QQuickView::continueExecute</td>
<td>411.509ms</td>
</tr>
<tr>
<td>v8::internal::CallIC_Miss</td>
<td>1.771ms</td>
</tr>
<tr>
<td>v8::internal::LoadIC_Miss</td>
<td></td>
</tr>
<tr>
<td>QQmlComponent::create</td>
<td>409.738ms</td>
</tr>
<tr>
<td>QQmlComponent::beginCreate</td>
<td>44.360ms</td>
</tr>
<tr>
<td>QQmlComponentPrivate::beginCreate</td>
<td>44.360ms</td>
</tr>
<tr>
<td>QQmlComponent::completeCreate</td>
<td>365.378ms</td>
</tr>
</tbody>
</table>
• Pay attention to what you measure
  – Cycle count does not include time blocked!
  – Compile in release mode
  – Profile on target device
  – Profile with cold cache

• User code and QML engine code
  – QML engine part opaque
  – high level tooling required
Startup Time - Meet the QML Profiler
· Use Qt 5.4 and QtCreator 3.2
· Enable profiler in settings
  – QMake CONFIG flag
  – run argument
· Record only what you need
1. Compiling
2. Creating
3. Bindings
4. Completion

- JS: `Component.onCompleted`
- C++: `QQuickItem::componentComplete()`
- Text layouting, image loading, creation of Repeater/ListView delegates, ...
• Removing fonts improved startup from 900ms to 200ms
• Completion phase shrunk considerably
Startup Time - Compilation

- Compilation phase fast, small amount of total
- Runs in a separate thread
- QtQuick Compiler pre-compiles files
  - Phase reduced by ~50%
  - Available since Qt 5.3 Enterprise
• Keep bindings simple
• Move complex code to C++
• Use QtQuick compiler if available
function stuff()
{
    console.time("Stuff");
    var a = Math.random(100);
    var b = Math.random(100);
    var c = Math.random(100);
    var sum = 0;
    for (var x = 0; x <= 10000; x++) {
        for (var i = 0; i <= 10000; i++) {
            var d = i*a + i*b + i*c;
            sum += d;
        }
    }
    console.timeEnd("Stuff");
    console.log("SUM= " + sum);
}
· Results
  – Without QtQuick Compiler, Release: 1000ms
  – With QtQuick Compiler, Release: 500ms, 398 instructions (w/o calls)
  – With QtQuick Compiler, Debug: 5000ms, 818 instructions (w/o calls)
  – C++ version, Release: 50 ms, 78 instructions (w/o calls)

· Use QtQuick Compiler if available
· Improvements in simpler code (bindings) \( \sim 15\% \) (*)
· Move complex code to C++
• Not much one can do
• Use fewer elements in QML files
• Make sure custom items are constructed quickly
Use **Loader** to load views later
Startup - Summary

- Profile both C++ and QML
- Know your tools, understand their output
- Move complex JS code to C++
- Use Loaders
- Use QtQuick Compiler when available
Smooth Rendering / Frames per Second
Rendering itself is rarely the culprit!

- High CPU/GPU usage from other processes or threads
- ListView scrolling instantiates new delegates
- Timers in C++ or JS, event handling in C++
- Use a CPU profiler and the QML profiler first to verify!
· See http://qt-project.org/doc/qt-5/qtquick-visualcanvas-scenegraph-renderer.html#performance for general tips to improve render performance

· Useful visualizations with QSG_VISUALIZE
  - batches
  - clip
  - overdraw
  - changes
• QSG_VISUALIZE=overdraw
• No viewport clipping and occlusion culling in renderer!
• Make sure visible is false
• QtCreator Enterprise or `QSG_RENDER_TIMING=1`
• `QSG_RENDER_LOOP=threaded`
• Measures CPU time
• No animations running -> 0 FPS
• GUI Thread

  - **polish**: `QQuickItem::updatePolish()`
    * anchor and text layouting, canvas drawing, ...
  - **animations**: Advancing all animations (binding updates!)
  - **lock**: Posting sync request to render thread
  - **block/sync**: Wait for render thread to call `QQuickItem::updatePaintNode()`
    * Main/GUI thread will block while render thread busy!
Rendering - Measuring Frame Time

- **Render Thread**
  - *framedelta*: 1000 / FPS
  - *sync*: Actual `QQuickItem::updatePaintNode()` call
  - *first render*: CPU render time
  - *final swap*: Swap time

- **Caveat**: swap time + render time >= 16ms with 60 Hz vsync
- **Caveat**: Some drivers wait in first GL call of next frame, not in `glSwapBuffers()`!
Rendering - apitrace

```cpp
#define lowp
#define medium
#define highp

varying highp vec2 coord;
uniform sampler2D src;
void main() {
  int sum = 0;
  for(int i=0;i<20;++i)
    for(int j=0;j<20;++j)
      for(int k=0;k<8;++k)
        sum += i * j * k;
  lowp vec4 tex = texture2D(src, coord);
  gl_FragColor = vec4(tex.rgb, float(sum));
}
```

1239) `glDrawElements(mode = GL_TRIANGLE_STRIP, count = 6, type = GL_UNSIGNED_SHORT, indices = 0x40)`
Rendering - apitrace

- Traces and times OpenGL calls on CPU and GPU
- Shows complete GL state, including buffers and shaders
- Useful when integrating custom items into QtQuick
- Useful when working on the scenegraph renderer itself

**Usage:**

- `apitrace trace` to record
- `qapitrace` to visualize and play back
Responsiveness
Responsiveness

- Usually starts in QtQuick signal handlers like `onClicked` or `onPressed`.
- Mix of JS code, property/binding updates and calls into C++.
- Measure only relevant time period.
- Start with QML Profiler, descent into CPU profiler if needed.
- May load new view.
  - Similar analysis as startup time.
  - Loader: startup time vs reaction time.
Boot Duration
Power Usage
<table>
<thead>
<tr>
<th>Cn</th>
<th>Avg residency (5s)</th>
<th>Long term residency avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>(cpu running)</td>
<td>(3.8%)</td>
</tr>
<tr>
<td>C1</td>
<td>0.0ms (0.0%)</td>
<td>0.0ms</td>
</tr>
<tr>
<td>C2</td>
<td>4.4ms (57.3%)</td>
<td>4.4ms</td>
</tr>
<tr>
<td>C3</td>
<td>10.0ms (31.1%)</td>
<td>10.0ms</td>
</tr>
<tr>
<td>C4</td>
<td>2.3ms (7.7%)</td>
<td>2.3ms</td>
</tr>
</tbody>
</table>

Wakeups per second: **193.6**

Power usage (ACPI estimate): **13.0 W (6.5 hours left)**

Top causes for wakeups:

- **35.2%** `<interrupt>`: i8042
- **28.4%** `<interrupt>`: yenta, 1915@pc1:0000:00:02.0
- **13.6%** `<interrupt>`: ipw2200, Intel 82801DB-ICH4
- **4.6%** `xorg`: do_setitimer (it_real_fn)
- **3.7%** `firefox-bin`: schedule_timeout (process_timeout)
- **3.5%** `xchat`: schedule_timeout (process_timeout)
- **1.6%** `firefox-bin`: schedule_timeout (process_timeout)
- **1.3%** `gnome-terminal`: schedule_timeout (process_timeout)
- **1.1%** `gnome-power-man`: schedule_timeout (process_timeout)
- **1.1%** `emerald`: schedule_timeout (process_timeout)

Suggestion: Enable the CONFIG_USB_SUSPEND kernel configuration option. This option will automatically disable UHCI USB when not in use, and may save approximately 1 Watt of power.
• **powertop** to check for process wakeups and HW power usage
• **QML profiler** to check for unnecessary animations
• **Gammaray** timer top to check for unnecessary timers
Memory Usage
- **massif** to track C++ heap allocations
- **QML Profiler** (enterprise) to track JS memory usage
- QML engine: ?
Thank you!

Questions?

Thomas McGuire - KDAB - thomas@kdab.com