basysKom

QT QUICK ON LOW-END i.MX6 DEVICES

Jeremias Bosch

19.08.2019

Who am I

Jeremias Bosch Dipl. Inf. (FH)

- Since 10+ years at basysKom GmbH
- Senior Developing Project Manager
- Qt Quick, C++
- IoT, Azure, Cloud

basysKom GmbH

- is a Qt Service Partner since 2004
- is located in Darmstadt and Nürnberg
- is employing ~30 people
- is part of the UX Gruppe
- provides services (consulting, training, coaching and development) around Qt
- focuses on technical/industrial applications of Qt (HMI and application development)

Why this talk

We see that our customers tend to select the smallest possible i.MX6 Device still providing a GPU.

You find yourself, as a developer,

- In a price sensitive environment,
- Squeezing the last bit of HMI performance from lower end hardware, such as a SoloX.

This talk will discuss two projects, their KPI’s and the taken technical decisions to fulfill those KPI's.

SoloX Project

400 Mhz (Cortex A9 & M4)

- 400 Mhz because of thermal reasons. In some situations even only 200 Mhz.

Limited GPU

- To reduce impact on CPU.

512 MB Memory

800x480 Display

Qt 5.6.3

QML

Heavy Backend Processes (utilizing ~50% of the CPU)

Webserver (Node.js) running a Web-API parallel to the local Qt HMI to serve an Angular Web-App (provided by the same device)

KPI’s

- Dynamic - Fullscreen - Multipass Shader Component
- "Fast Boot": Power on → Splash < 10 seconds
- 30 fps
- Fullscreen Animations
- > 100 Screens
- < 70 MB Memory usage

Boot Optimizations

System-Level

- Optimize your Kernel
- Do not show a Kernel-Splash image

On Application-Level

- Focus to start your executable as soon as possible
- Load only what is really required to display a first image
- Only load your real application once you displayed that image
- Lazy Loading / Lazy Startup

Lazy startup organization

Design your application to load in stages and modules

- First load the splash screen
- Only after that, start to load the content, logic, dialog system, state machine, IPC (and everything async to each other)

The result: partially loaded HMI during startup

- However, the user will see much earlier that 'something' is happening

Architecture

Caching vs. Load & Destroy

- Caching PRO:
  - No memory fragmentation
  - Fast display times
- Caching CON:
  - Time until everything is cached (this is a strategy point)
  - 'Higher *initial* memory usage'
- Load & Destroy PRO:
  - Lower initial memory usage, lower CPU usage at start
- Load & Destroy CON:
  - High fragmentation lead to higher memory usage over time
  - Loading on demand causes performance issues

Architecture

Caching vs. Load & Destroy

- Page Heap Pro:
  - No structure to keep in mind
  - Caching is easily possible
  - Easy to handle complex screen transitions and concepts
- Page Heap Con:
  - Requires external management
- Page Stack Pro:
  - Part of Qt components/controls
  - Stack allows to push/pop screens in a structured order
- Page Stack Con:
  - Complicated to use in complex situations
  - Caching is possible but more complicated

Architecture

Delegate Reuse

- Delegates are expensive
- Delegates fragment your memory
- Utilize the Complete/Destroy mechanism and "re-parent" the content of your delegates

Shader and Graphics

CPU Usage on Shaders!

- On low-end devices the CPU takes a critical part in handling fragment shaders (i.e. a fullscreen fragment shader on a SoloX takes ~35% CPU time to compute at close to 60 FPS)
- Avoid complex fragment shaders

Using Textures vs. Using Geometry

- For common structures, like rectangles, without gradient, we learned that geometry is the fastest solution
- For complex structures, like circles, a texture can be faster
- For icons, we found it *usually* much better (lower memory impact, faster loading times, higher customer satisfaction) to use white icons and color them with a shader rather than loading colored icons

Dual Lite Project

1,2 Ghz (Cortex A9, Dual Core)

2 GB Memory

1280x800 Display

Qt 5.9.6

QML

Heavy Backend

- Multiple communication frameworks in parallel for
  - Controlling the machine itself
  - Cloud access
  - Providing remote maintenance and supervision options over two separate interfaces
- OPC UA
- Legacy API

KPIs

Smooth User Experience with state of the art transitions

- Complex WYSIWYG-style content editors
- > 100 Screens + Dialogs
- Custom UI framework
  - I18N + VKB support for 36 languages
  - Dynamic styling

Architecture

- Boot-Time had no relevance – everything can be cached and loaded
- Heavy use of page caching and preloading to ensure smooth transitions
- Shader-based coloring and shadow generation for improved performance
- Using 1D instead of 2D textures where possible
  - Unfortunately not supported in QML, but can be simulated using a 2D texture with an 1px width or height
  - Better: Add support for 1D textures by extending the scene graph

Shader and Graphics

How to upload software-rendered dynamic content efficiently?

- QQuickImageProvider can be used, but every update needs to use a new URL since this API is not meant for dynamic content changes and the updated texture will not be visible!
- Make sure to disable image caching!
  - This should be a general remark: Image caching should only be enabled for reoccurring images (e.g. backgrounds for ui controls) and not for anything else, esp. not large background or content images!
- Fillrate can become a problem quickly, so overdraw, esp. with blending enabled, has to be kept as minimal as possible
- Blending is enabled for image elements with image-files that have an alpha channel and for rectangles that use the radius property while anti-aliasing is enabled (which is the default), so use those wisely!

Rendering Performance Issues

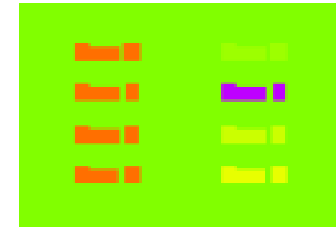
Important Read:

- https://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph-renderer.html

How to investigate your rendering pipeline?

- Visualize/Output the action of QSceneGraph using
  - `QSG_VISUALIZE=batches`
  - `QSG_RENDERER_DEBUG=render`
  - `QSG_RENDER_TIMING=1`

Unexpected high CPU Load might relate to a batching issue



QSG_VISUALIZE=batches

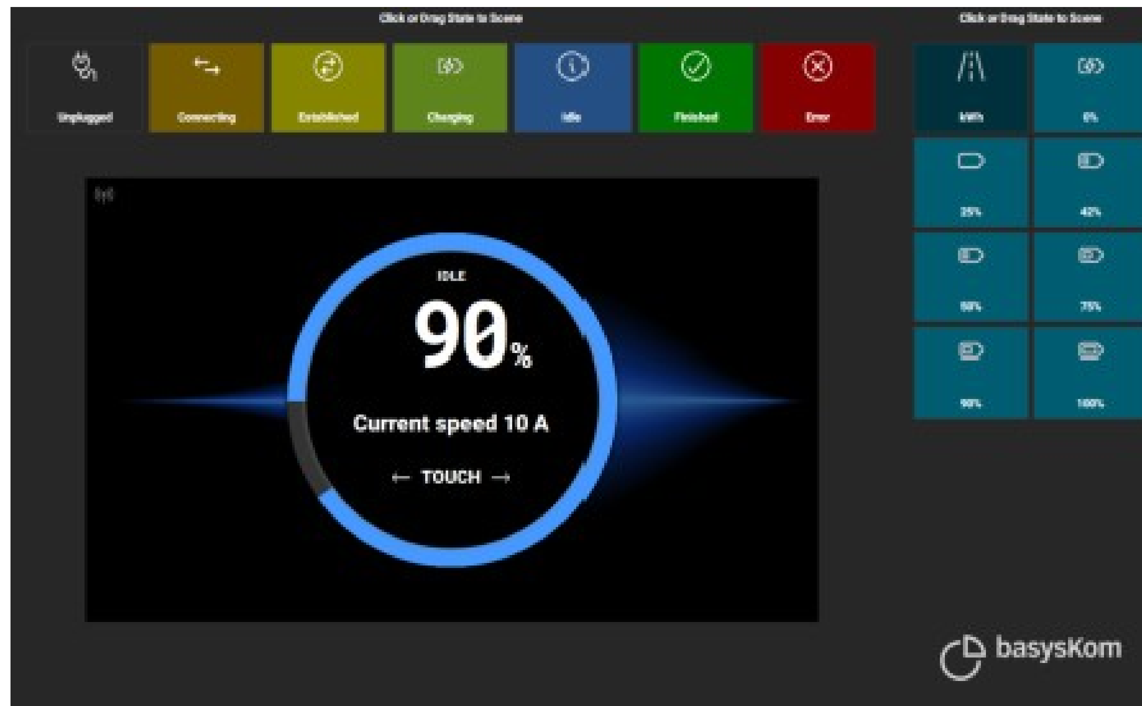
```
Renderer::render() QSGAbstractRenderer(0x318d2910) "rebuild: none"
Rendering:
 -> Opaque: 8 nodes in 1 batches...
 -> Alpha: 70 nodes in 33 batches...
 - 0x4d143670 [retained] [noclip] [opaque] [  merged]  Nodes:    8 …
 - 0x4d1437b0 [retained] [  clip] [ alpha] [unmerged]  Nodes:    1 …
 - 0x4d143bf0 [retained] [  clip] [ alpha] [  merged]  Nodes:    1 …
```

QSG_RENDERER_DEBUG=render

```
QQuickWindowQmlImpl_QML_41(0x2e92a598 active exposed, visibility=QWindow::Visibility(Windowed)…
qt.scenegraph.time.renderer: time in renderer: total=0ms, preprocess=0, updates=0, binding=0, …
qt.scenegraph.time.renderloop: Frame rendered with 'threaded' renderloop in 15ms, sync=0, rend…
qt.scenegraph.time.renderloop: Frame prepared with 'threaded' renderloop, polish=0, lock=0, bl…
qt.scenegraph.time.renderer: time in renderer: total=0ms, preprocess=0, updates=0, binding=0, …
qt.scenegraph.time.renderloop: Frame rendered with 'threaded' renderloop in 16ms, sync=0, rend…
qt.scenegraph.time.renderloop: Frame prepared with 'threaded' renderloop, polish=0, lock=0, bl…
qt.scenegraph.time.renderer: time in renderer: total=0ms, preprocess=0, updates=0, binding=0, …
qt.scenegraph.time.renderloop: Frame rendered with 'threaded' renderloop in 15ms, sync=0, rend…
qt.scenegraph.time.renderer: time in renderer: total=0ms, preprocess=0, updates=0, binding=0, …
```

QSG_RENDER_TIMING=1

Some Demo :)



Come visit our booth!

Conclusion

Yes you can create smooth running modern HMI’s on low-end i.MX6 Devices

- Keep the limitations in mind
- Always test on hardware
- Define KPI’s of what is acceptable performance
  - Attack issues when they first occur and build rules from your learning's
- Caching helps to make the memory usage more predictable