Qt + Embedded Linux + OpenGL = Fun:
The state of open-source Qt on Embedded Linux

Raspberry Pi 2 flavor
with a touch of Freescale Sabre SD

Laszlo Agocs
The Qt Company

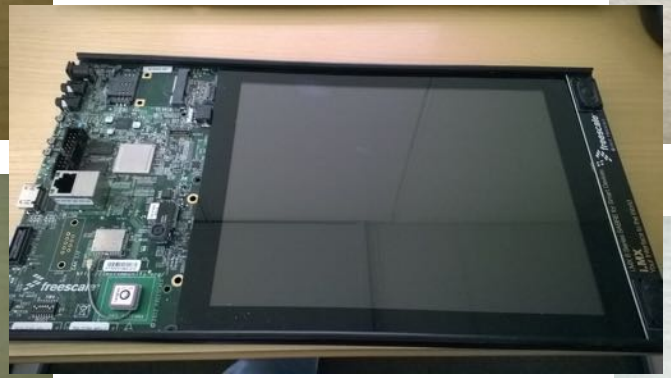
OSDC Nordic 2015

Who I am

- Senior software engineer at The Qt Company, Oslo, Norway
- Mostly focusing on OpenGL, windowing system integration and all sorts of platform work related to graphics & input
- Previously at ARM and Nokia

What do we mean by Embedded here?

- Not microcontrollers
- ARM Cortex-M and alike are out of question
- At least 500 MHz CPU, 256 MB RAM
- GPU with OpenGL ES 2.0 and 128 MB memory
  - These specs mean almost nothing in practice. It all depends on what you want to build.



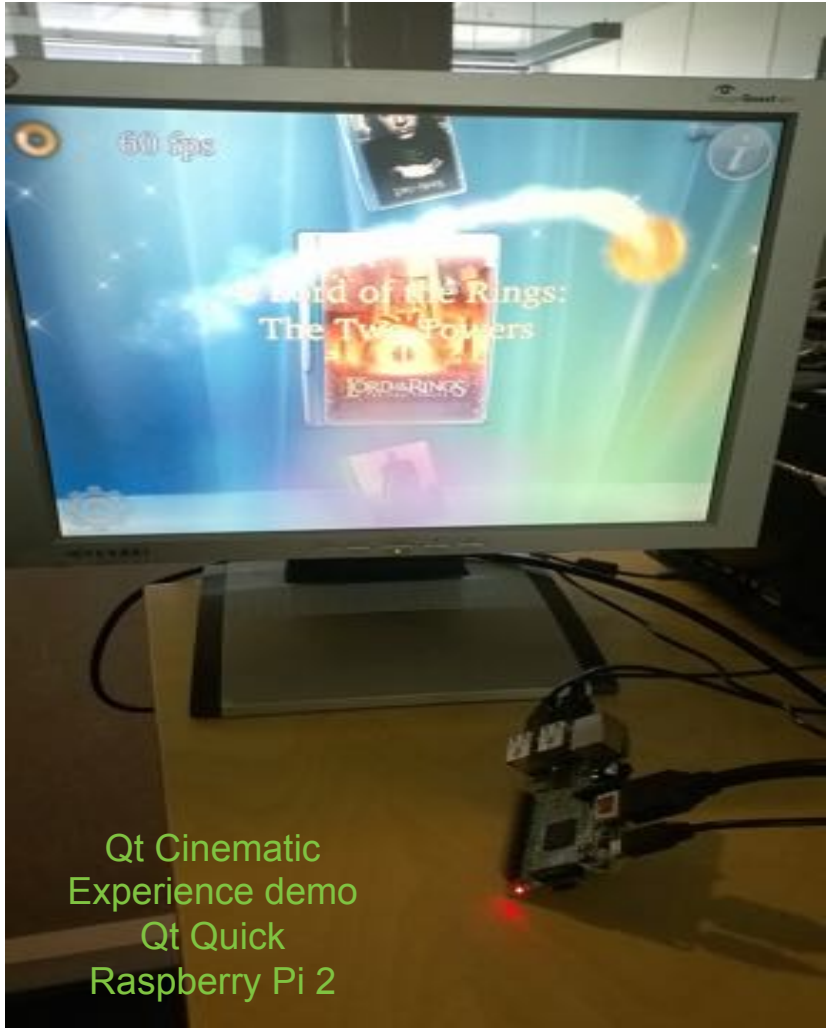
Just a few examples.
Lots of others out there.

- Boundary Devices Nitrogen6_Lite
  - ARM Cortex-A9 1 GHz, 512 MB RAM, Vivante GC880 GPU, Freescale i.MX 6Solo SoC
- Boundary Devices Nitrogen6X
  - ARM Cortex-A9 1GHz quad, 1 GB RAM, Vivante GC2000 GPU, Freescale i.MX 6Quad SoC
- Freescale Sabre SD
  - ARM Cortex-A9 1 GHz dual/quad, 1 GB RAM, Vivante GC2000 GPU, Freescale i.MX 6Dual/Quad SoC
- Silica ArchiTech Tibidabo
  - ARM Cortex-A9 1 GHz quad, 2 GB RAM, Vivante GC2000 GPU, Freescale i.MX 6Quad SoC
- Raspberry Pi 2
  - ARM Cortex-A7 900 MHz quad, 1 GB RAM, Broadcom VideoCore IV GPU, Broadcom BCM2836 SoC
- Hardkernel ODROID-XU3
  - Cortex-A15 2 GHz + Cortex-A7 quad, 2 GB RAM, ARM Mali T-628 GPU, Samsung Exynos5422 SoC
- NVIDIA Jetson TK1
  - Cortex-A15 2.3 GHz quad, 2 GB RAM, Kepler GPU, NVIDIA Tegra K1 SoC

Not quite the UI you want to see in your car, flight entertainment system, TV, set-top box, treadmill, fridge, washing machine, and various other appliances.

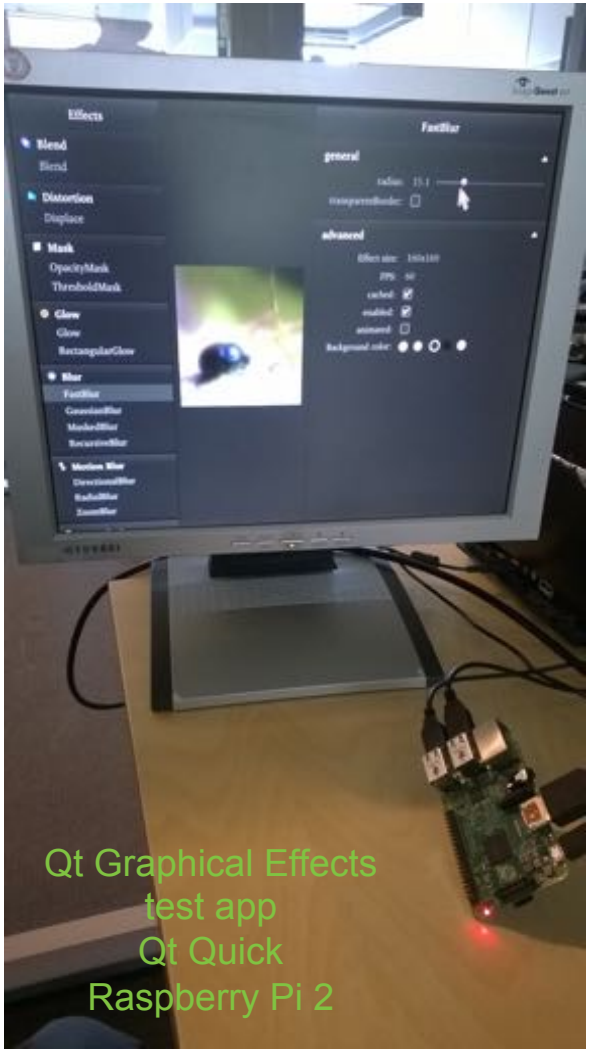
User interface

- Desktop PC style windowing environments not ideal
- Many systems are fine with booting into a single full-screen gui application
- It has to look good. Effects and smooth transitions.
  - GPU essential
- Focus on touch, voice, and fancy future input methods



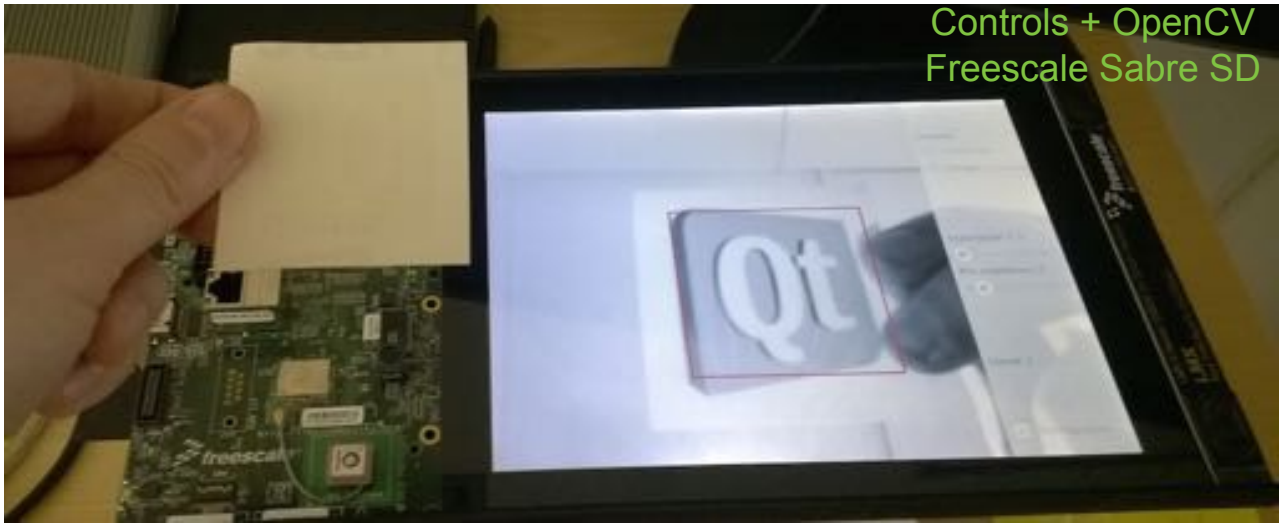
Qt Cinematic
Experience demo
Qt Quick
Raspberry Pi 2

Qt Quick Controls
Flat style
Raspberry Pi 2



Qt Graphical Effects
test app
Qt Quick
Raspberry Pi 2

Qt automotive HMI
demo
Qt Quick + custom 3D



Qt Quick + Camera
via Qt Multimedia +
Controls + OpenCV
Freescale Sabre SD

Why is this hard?

- User interface controls completely customized
  - No native look and feel. No ready-made widgets.
- Often no windowing system
  - No easy drawing, no input events, no nothing.
- EGL, OpenGL (ES), OpenCL
  - It’s all standard, right? (yes, but…)
  - fbdev, kms, x11, wayland will all just work? (yeah, right…)
  - Vendor-specific bits (particularly for fbdev)

What is Qt?

- Cross-platform C++/QML application and UI framework
- Core: Meta object system, JSON, XML, threading, containers, animation framework, etc.
- Gui: Window management, input, OpenGL abstractions, traditional widgets.
- QML: UI specification and programming language. Declarative, JSON-like syntax with support for imperative JavaScript expressions combined with dynamic property bindings.
- Quick: QML types for creating user interfaces, C++ classes to extend it (custom UI elements), all based on a 2D-focused scene graph on OpenGL ES 2.0.
- Multimedia (audio, video, camera), Networking (TCP, SSL, Web Sockets), Connectivity (Bluetooth, NFC), Location (GPS, mapping), Web Engine (Chromium), 3D (Qt 3D, Canvas 3D), Compositor (make your own Wayland compositor), …

Why does Qt rule on Embedded?

- QPA: Platform abstraction layer since Qt 5.0.
- Platform plugins for (Embedded) Linux:
  - linuxfb: Plain /dev/fbX: SW-only rendering, no OpenGL
  - eglfs: Windowing system-less, fullscreen EGL/OpenGL ES platform. #1 on embedded. Qt 5.5 introduces modularized backends: vendor-specific fbdev glues, KMS/DRM, and even X11.
  - xcb: The full, windowed X11 platform. Used on embedded boards where X11 is needed or is the only option.
  - wayland: Run Qt apps with a Wayland compositor.

eglfs: Out of the box support for i.MX6, RPi 1/2, Mali, Beagle*, or anything with KMS

So how do I make an embedded app with Qt?

- Qt Quick, likely with Qt Quick Controls
  - UI described in QML, logic in C++.
  - Perhaps combined with custom OpenGL rendering.
- C++ widgets (QWidget and its subclasses)
  - Usable to some extent on embedded too, likely with custom styling, but it is not the way forward.
- Custom OpenGL rendering
- Custom QPainter-based, non-accelerated rendering

3D

- Plenty of options to integrate raw OpenGL code
  - QOpenGLWindow, QOpenGLWidget
  - Below or above the Qt Quick scene
  - As regular Quick scene items via framebuffer objects
- Qt Canvas 3D (tech preview in Qt 5.4, stable in 5.5)
  - A WebGL-like API for QML's Javascript engine.
  - Use Three.JS or other frameworks and reuse your Web code.
- Qt 3D (tech preview in Qt 5.5)
  - Both C++ and QML APIs.

Multimedia

- The news of Qt Multimedia's death are exaggerated.
  - Actively developed. Challenges due to the number of platforms, backends, and devices.
- Avoid the legacy stuff (widgets). Qt Quick is where it's at.
- Qt 5.5 introduces GStreamer 1.0 support in addition to 0.10.
- When getting started with a new device, there will often be issues.
  - But known-good systems (e.g. i.MX6) are awesome.

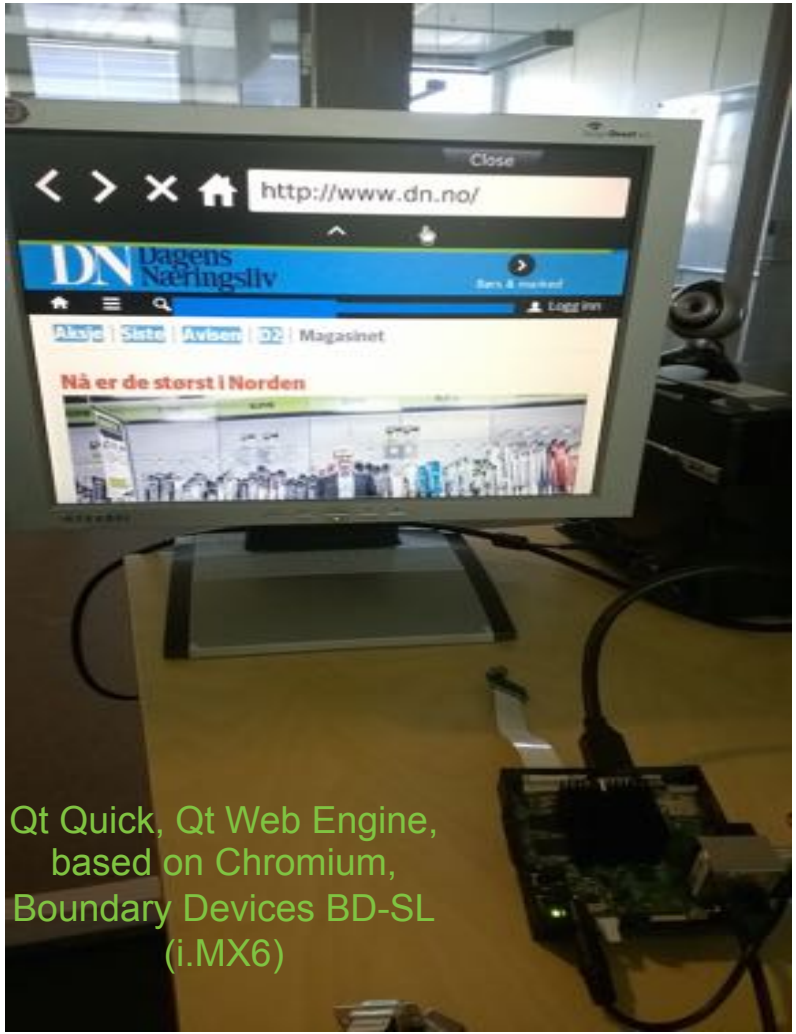
Qt Quick, Qt Multimedia, accelerated video, OpenGL shader effects
Boundary Devices BD-SL (i.MX6)

Camera w/ Qt Multimedia in a Qt Quick scene
Boundary Devices Nit6x_5MP_MIPI camera + BD-SL (i.MX6)

Licensing

- GPLv3, LGPLv2.1, LGPLv3, Commercial
- Qt 5.5: new modules and some existing ones LGPLv3
  - Qt Quick Controls, Qt WebEngine, Qt Location, Qt 3D, Qt Canvas3D
  - Future new stuff expected to follow suit
- Previously commercial-only stuff is now open
  - Touch-friendly “flat” style for Quick Controls. Essential for embedded. Enterprise Controls (industrial gauges etc.) is now available as Qt Quick Extras.

Qt Virtual Keyboard
Qt Quick Controls
(VKB commercial only for now, but who knows what the future brings)



Qt Quick, Qt Web Engine, based on Chromium, Boundary Devices BD-SL (i.MX6)

Getting Qt 5 onto the target device (and the host)

- Take the sources from git and cross-compile
- Use the recipes for Yocto, Buildroot, etc. (cross-compile)
- Take the sources from git and build on the device
  - Fine on boards like the Jetson TK1. Bad idea for lower spec boards.
- Install from the distro repos (on device)

What's the catch?

- Building yourself: No catch, really, but needs some knowledge.
  - The official way.
  - Can use any Qt version or even unreleased code.
  - Full control over what to build and in what configuration.
- Distro provided builds: May or may not provide what you want
  - Almost certainly out of date, probably with a focus on traditional X11 stuff instead of proper graphics/multimedia performance.
- Yocto:
  - meta-qt5 is unofficial and not without issues.
  - Going to get better. Increased focus in the future.

How do I build it?

Raspberry Pi 2

- Quad-core Cortex-A7 @ 900 MHz, 1 GB RAM
- Same Broadcom GPU like on the original Pi
  - EGL and Open GL ES 2.0 supported, but not under X11
    (when using Broadcom's proprietary driver, which we do)
  - Good enough to run fancy Qt Quick UIs at 60 FPS
- Accelerated video with GStreamer 1.0
  - Which Qt 5.5 finally supports
    (when explicitly requested and gstreamer1.0 dev files are available. Default in Qt 5.5 is still 0.10. 1.0 support still stabilizing.)
- Could use Yocto or Buildroot but here we choose Raspbian
  - Easier to get started this way (For now. Expect Yocto support getting more focus in the near future.)

Let’s go…

- Will use Raspbian, a Debian variant, and build and deploy Qt 5.5 manually on top
- We are after Qt Quick and and 60 FPS fluid UIs
- No windowing system, we will use eglfs
- Mouse/keyboard/touch input will come via Qt’s own evdev support (btw Qt 5.5 adds libinput as an option too)
- Who needs boring desktop-ish windows anyway

Get the image and flash it

- http://downloads.raspberrypi.org/raspbian_latest
  - 2015-02-16-raspbian-wheezy.img
- `sudo dd if=2015-02-16-raspbian-wheezy.img of=/dev/sdf bs=4M`
- So we have an image. Boot it up, install additional stuff, if needed.

Cross-compilation

- Get a toolchain
  - git clone https://github.com/raspberrypi/tools
- Now we have gcc and other tools: *tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-{gcc,…}*
- So we can compile. But no sysroot. Have to get the headers and (ARM) libs onto our host.
- Here we will just copy stuff out of the image.
  - sshfs could have been an option too

Sysroot

- `sudo mount -o loop,offset=62914560 2015-02-16-raspbian-wheezy.img sysroot_src`
- Copy *lib, opt/vc, usr/include, usr/lib*
- Bad news: some of the .so links are absolute.
  - Get the script from http://wiki.qt.io/Chromebook2
  - `fixlibs.sh ~/raspi/sysroot`
- We can finally build apps!

Get Qt

- Won’t bother with the full package, enough to get a few core modules to get started.
- *git clone git://code.qt.io/qt/qtbase -b 5.5*
  - Repeat for qtdeclarative, qtquickcontrols, qtgraphicaleffects, qtmultimedia and whatever else needed

Configure Qt

- *~/raspi/qtbase$* ./configure
  - -release
  - -opengl es2
  - -device linux-rasp-pi2-g++
  - -device-option CROSS_COMPILE=~/raspi/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-
  - -sysroot ~/raspi/sysroot
  - -prefix /usr/local/qt5
  - -extprefix ~/raspi/sysroot/usr/local/qt5
  - -hostprefix ~/raspi/qt5-host
  - -opensource -confirm-license -make libs -v

Some explanation

- -device specifies the device makespec. See qtbase/mkspecs/devices. This is where board-specific compiler flags and eglfs settings come from.
- Toolchain and sysroot must be specified.
- Prefixes often not understood. But it's all so simple:
  - -prefix : deployment location on the target (the memory card). E.g. Qt libs will be under /usr/local/qt5/lib on the sd card. Up to you to ensure this remains true when scp/rsync'ing! (or use sshfs)
  - -extprefix : local deployment location. This is where make install will copy the files. Optional, actually, since the default is what we specify here: sysroot+prefix.
  - -hostprefix : this is where host tools are installed. Like the x86 build of qmake that is used when cross-compiling apps.

Troubleshooting

- If some config test fails and aborts, e.g. some .so not found -> most likely messed up the sysroot. Fix it and retry.
- Sanity check configure's output:

```
Support enabled for:
  evdev .................. yes
  OpenGL / OpenVG:
    EGL .................. yes
    OpenGL ............... yes (OpenGL ES 2.0+)
  QPA backends:
    EGLFS ................ yes
        EGLFS Raspberry Pi . Yes
    LinuxFB .............. yes
  udev .................. no    -> ideally yes, see next slide
```

Troubleshooting

- What if some optional dev package is missing and we want it?
  - For example libudev headers and libs are needed to get dynamic input device monitoring (USB mouse/keyboard hotplug)
- `sudo apt-get install libudev-dev` on the device
  - Has nothing to do with libudev-dev on the host.
- With live sshfs mounts this is enough. In our case we need to copy the headers/libs back to our local sysroot dir. Or get the .deb and extract on the host.

Build

- make -j20 or whatever
- make install
- And there we have ~/raspi/sysroot/usr/local/qt5 while host tools (qmake) in ~/raspi/qt5-host
- Building other modules and applications is all the same. Use the qmake from qt5-host.
  - `~/raspi/qtdeclarative$ ../qt5-host/bin/qmake -r && make -j20 && make install`

Deploy

- scp or rsync the stuff from the sysroot.
- ~/raspi/sysroot/usr/local$ `rsync -avz -e ssh qt5 pi@192.168.1.149:/usr/local`
  - (with sshfs this is of course not needed)
- Qt is on the device. And we can build apps on our host PC.
- Let’s build the Qt 5 Cinematic Experience demo: just qmake and scp the necessary files. The result is...

Finally!

Thank you!

http://www.qt.io
https://doc-snapshots.qt.io/qt5-5.5/embedded-linux.html