Qt Graphics Stack

and its usage of Khronos technologies

Laszlo Agocs & Andy Nichols

The Qt Company

Oslo, Norway

17 August 2016

The Qt Company: A Brief Introduction

- Responsible for all Qt operations globally
- Worldwide leader in
  - Qt API development
  - Qt Application Development
  - Design services – UI and UX
- Trusted by over 8,000 customers worldwide
- 20+ years of Qt experience
- 200 in-house Qt experts
- Fast growing
- 27M€ revenue in year 2015

Qt History

1991
Qt conceived by Haavard Nord and Eirik Chambe-Eng on a park bench in Trondheim, Norway.

1995
Troll Tech 1st public release on 20 May -
Qt 0.90 for X11/Linux
» Commercial & open source (FreeQt license)

1996
» Customer #1- European Space Agency
» Qt 1.0 - full X11 support free for free software development plus Windows
» KDE project established with Qt as its underlying library

1998
KDE Free Qt Foundation - guarantees Qt availability for free software development

1999
Qt 2.0 - Qt/X11 open source with QPL (Q Public License)

2000
» New Qt windowing system - Qt/Embedded - (a.k.a. QWS & Qtopia Core)
» Both Qt/X11 & Qt/Embedded under GPL + commercial licenses
» GPL v2 with Qt 2.2

2001
Qt 3.0 - "multiple database environments, multiple languages, multiple monitors" with Mac OS X support & a new Qt Designer GUI builder

2005
Qt 4.0 - Total makeover (a.k.a. compatibility break) under commercial & GPL 2.0 (or later) for all platforms even Windows (Qt 4 dance video published)

Qt History

2006
» Trolltech IPO
» Greenphone developer smartphone
» Qtopia in millions of devices worldwide from Sharp to Motorola

2008
Nokia acquires Trolltech – "Qt Software at Nokia"

2009
Qt Creator launched and Qt 4.5 brought LGPL v2.1

2010
Qt Quick launched, WebKit integration, & Symbian support in Qt 4.7

2011
» Digia acquires Qt commercial licensing business
» Qt Project
» Nokia N9 – Nokia MeeGo "Harmattan" smartphone
» Nokia changes smartphone platform strategy

2012
» Digia acquires all rights to Qt – "Digia, Qt"
» Qt 5.0 - Major overhaul - new modularized codebase, consolidation of QPA (Qt Platform Abstraction), Qt Quick 2, and more support for mobile (WinRT).

2013
Boot to Qt pre-built software stack & Qt WebEngine

2014
"Digia, Qt" becomes its own entity "The Qt Company", subsidiary of Digia

2015
20th anniversary of Qt's 1st public release
» One Qt Site Unification Completed
» 800K+ Qt developers worldwide

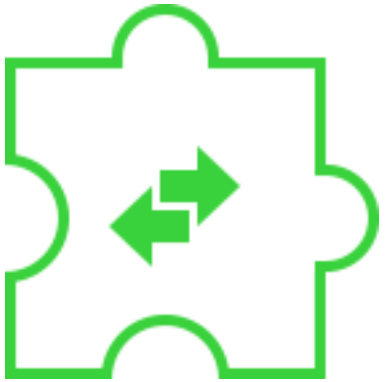
The Leading C++ Cross-Platform Framework

Cross-Platform
Class Library

One Technology for All Platforms

Integrated
Development Tools

Shorter Time-to-Market



Cross-Platform
IDE, Qt Creator

Productive development environment

Used by over 1 million developers in 70+ industries

Proven & tested technology – since 1994

Qt is Used for…

Application Development

on Desktop,

Mobile and Embedded

Creating Powerful Devices

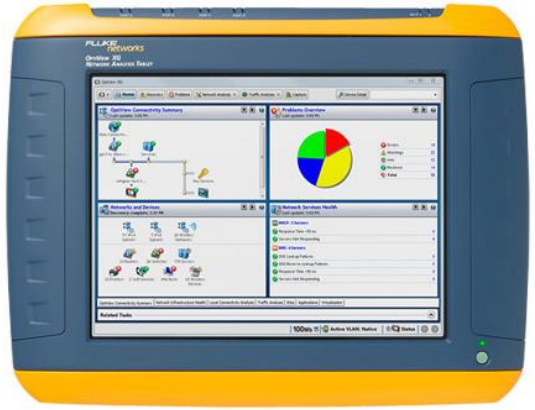
Device GUIs,

Ecosystems and whole SDKs

Where There’s a User Interface, There’s Qt

Automotive IVI

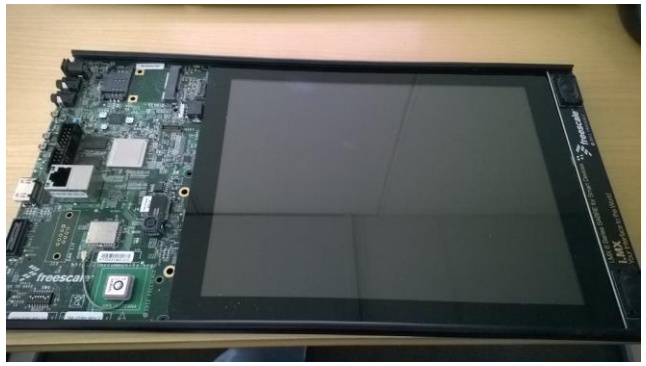
Refrigerators & Coffee Machines



Network Analyzers

- Medical Devices
- Home Automation
- Digital Photo Frames
- Set Top Boxes
- Industrial/UMPCS
- and many, many more …

Embedded Development Boards



Qt UI Offering – Choose the Best of All Worlds

Qt Quick

C++ on the back, declarative UI design (QML) in the front for beautiful, modern touch-based User Experiences.

Built on a 2.5D OpenGL ES 2.0 scenegraph.

Qt Widgets

Customizable C++ UI controls for traditional desktop look-and-feel. Also good for more static embedded UIs for more limited devices / operating systems.

Software rendered (QPainter).

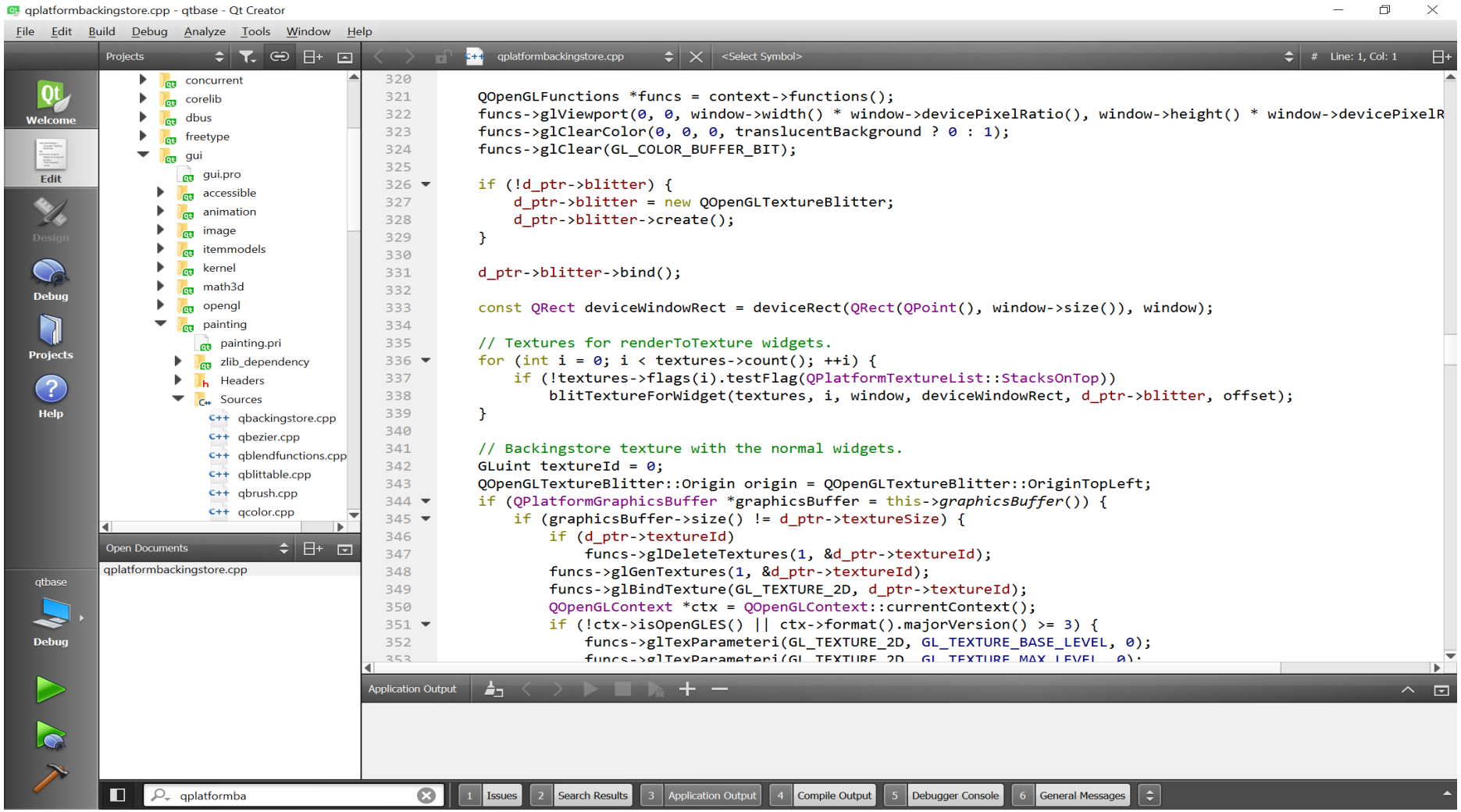
May be composited together with 3D rendering via OpenGL.

Web / Hybrid

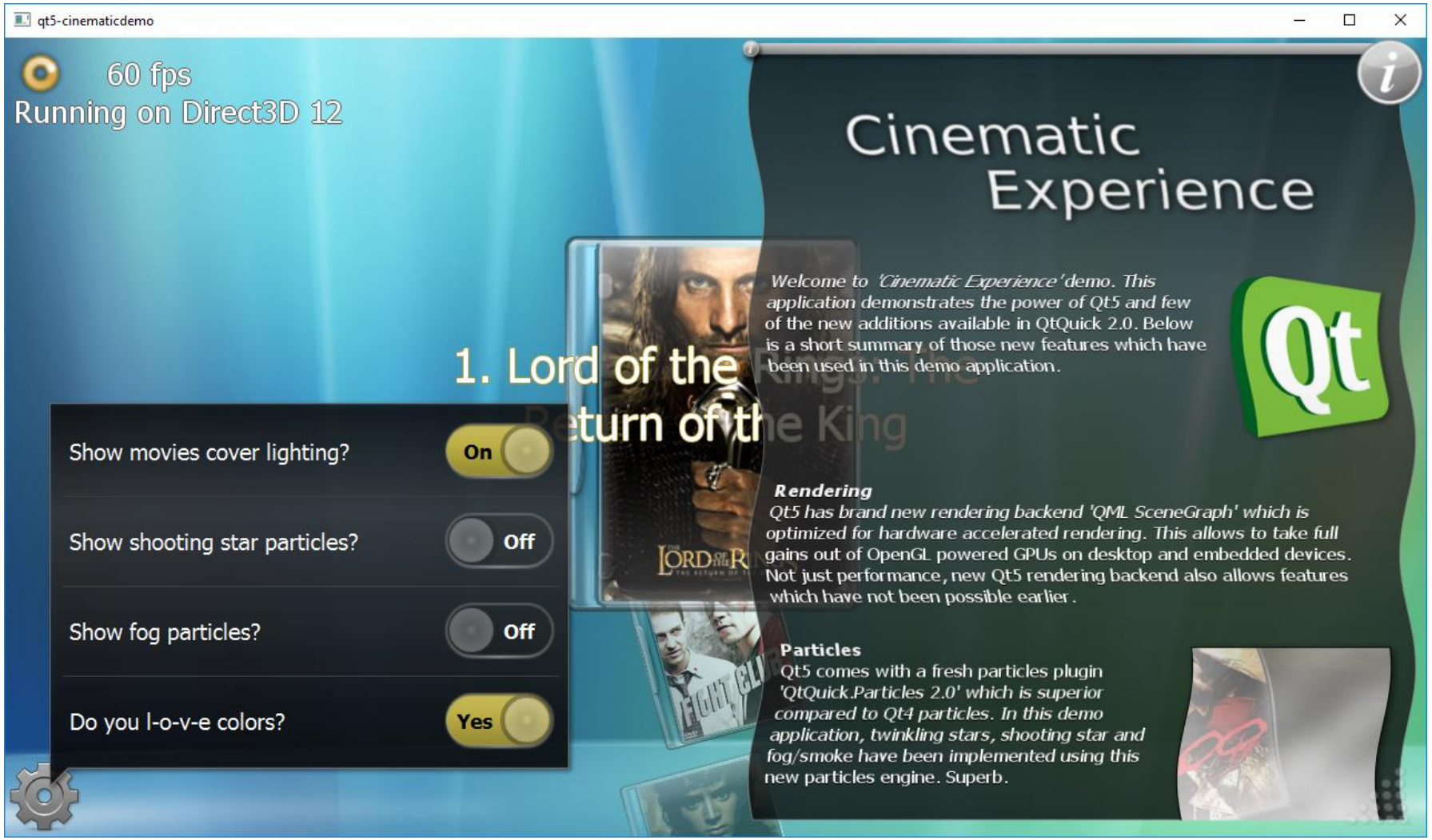
Use HTML5 for dynamic web documents, Qt Quick for native interaction.

Qt WebEngine: built on Chromium. OpenGL or pure software rendering.

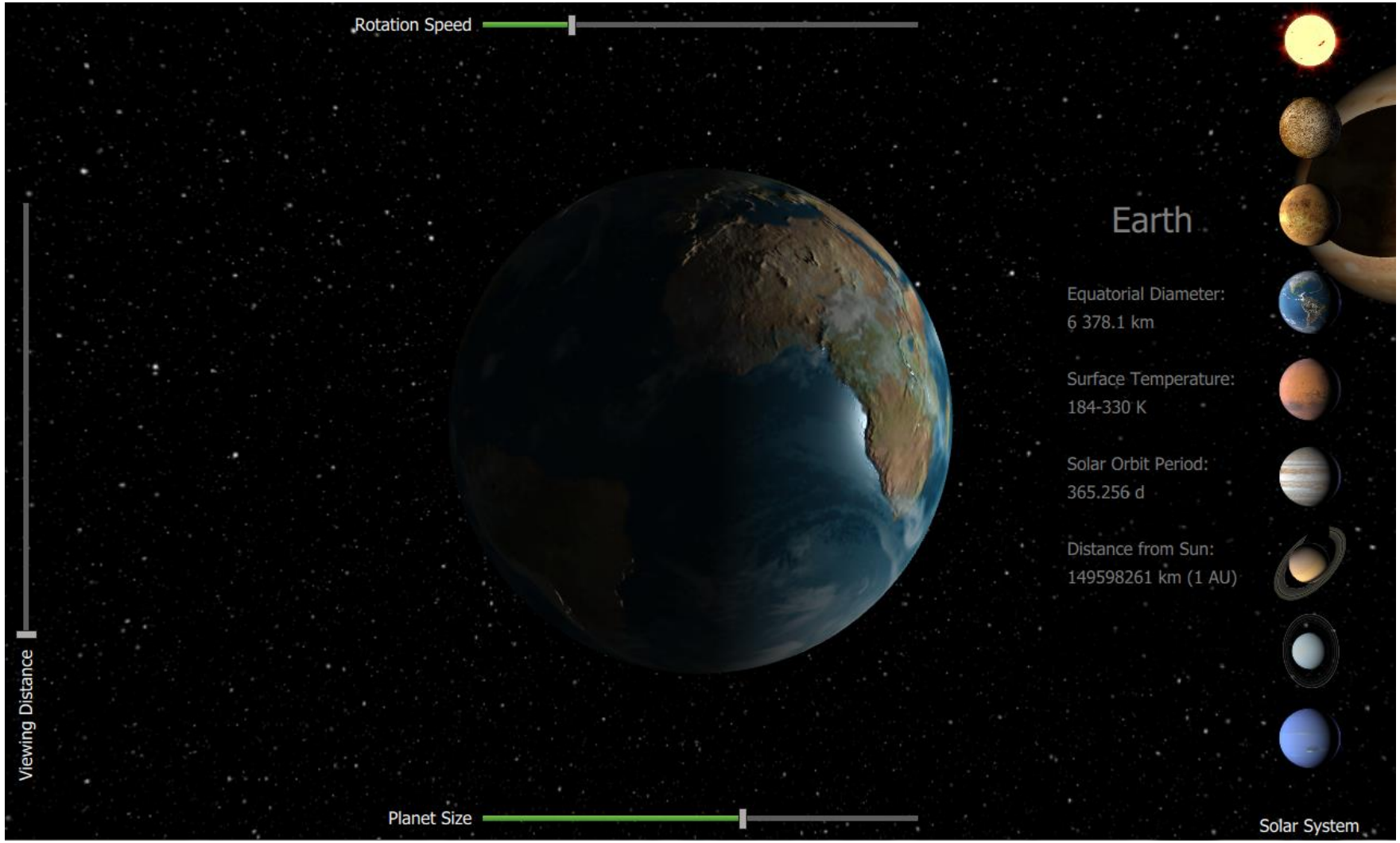
QWidget vs. Qt Quick



QWidget vs. Qt Quick

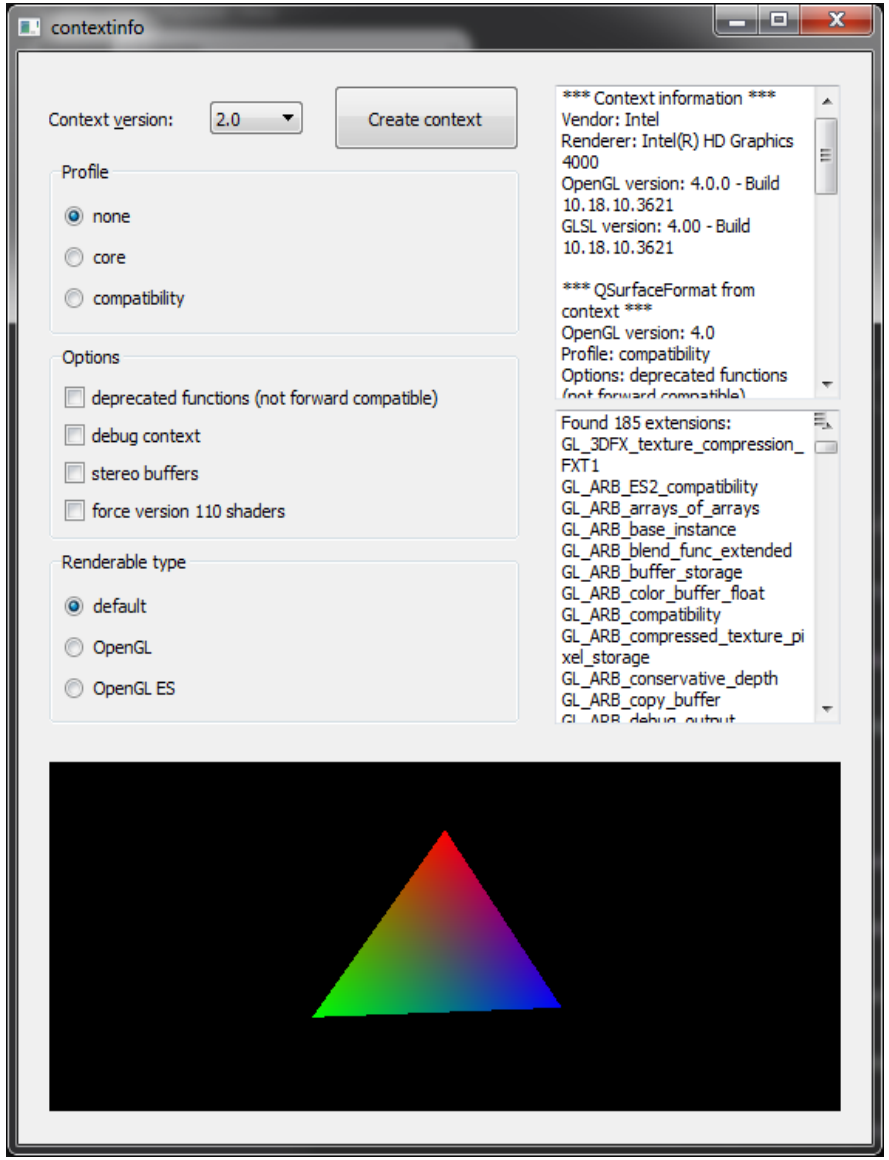


What About True 3D Content?



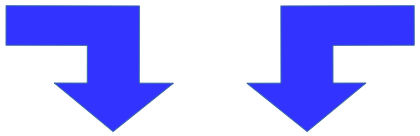
Integrating 3D content

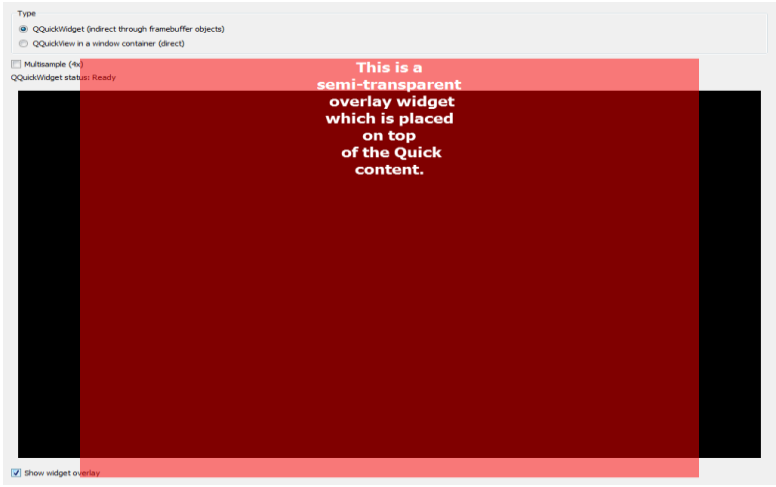
- Plain QWindow
- QWidget + embedded native window
- QWidget + render into FBO and compose
- Qt Quick
  - custom OpenGL rendering as under/overlay
  - FBO item in the scene
- Qt Canvas 3D
- Qt 3D



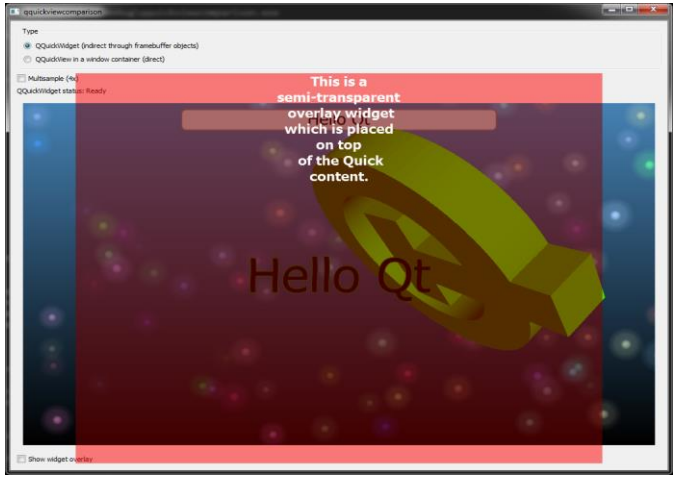
A complex example (Qt Quick + QWidget)

black has alpha == 0





Qt Quick scene with text element, particles and custom OpenGL rendering via an FBO. The whole scene is then rendered into another FBO to get a texture the widget stack can use for composition.



Plain QWidget content, rendered on CPU. Instead of the flushing it to the window, it gets uploaded into an OpenGL texture.

Qt Canvas 3D

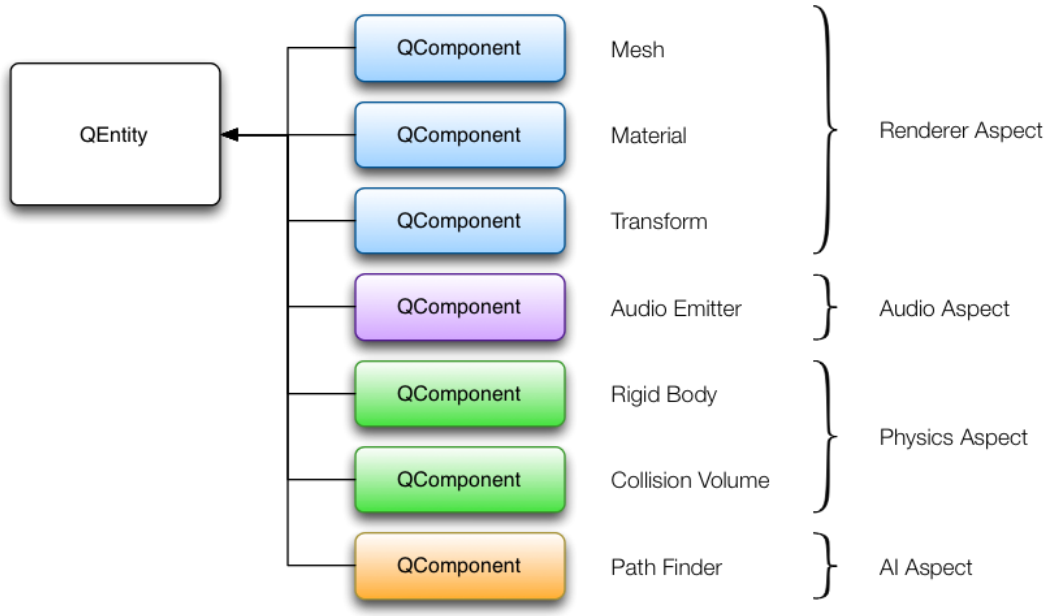
› Since Qt 5.5.

› WebGL + three.js in the Qt Quick JavaScript engine:

WebGL is a nice and productive environment for implementing 3D content
QtQuick is a nice and productive environment for doing 2/2.5D UI
→ Combining these two makes for a very productive environment

Reuse existing code from the web and combine it with a QML-based UI

Qt 3D



- Since Qt 5.7.
- Full 3D engine targeting both C++ and QML.
- Developed mainly by KDAB.
- Extensible entity-component architecture.
- Multi-threaded job system.
- Also targets modern OpenGL, unlike the core of Qt that uses OpenGL ES 2.0 as the baseline for portability.

Qt 3D

› https://www.youtube.com/watch?v=zCBESbHSR1k

› Qt Quick + Qt 3D in-scene via an FBO

› Running on an NVIDIA Jetson TK1 board

glTF in Qt 3D

- Asset transmission and loading format.
- Scene description in JSON,vertex data as binary. Fast to load.
- While experimenting with Qt 3D on embedded devices last year, loading models and scenes turned out be a pain.
- So what if instead of SceneLoader { source: “blah.obj”} we do some asset baking at build time?
- Add a tool and integrate it with qmake. The application’s .pro file can now contain:

```
MODELS += car.3ds house.obj whatever.dae
load(qgltf)
```

And then simply write SceneLoader { source: “:/models/whatever.gltf”} in QML

glTF in Qt 3D

- Conversion using assimp at build time on the host machine.
  - No need to build and deploy assimp as part of Qt 3D on the target device.
- Pack into compressed Qt resource files.
  - Including external textures, shaders, everything.
- Extra goodies: can generate shaders suitable for OpenGL core profile. Or invoke etc1tool to compress textures.
- Potential for adding further build time asset optimizations.
- Future unknown, let’s see.

How Much OpenGL is There in The Core of Qt?

- Windowing, setting up rendering (will talk about this later)
- The Qt Quick scene graph
- Some OpenGL code in widgets to support the QOpenGLWidget/QQuickWidget composition case
- Various OpenGL function resolvers, both for internal and external use:
  - Expose a GLES 2.0 subset, hide some differences, make it work everywhere.
  - Version and profile-based resolvers for modern OpenGL.
- Some API wrappers (QOpenGLBuffer, QOpenGLFramebufferObject, etc.)
  - Mainly for internal use. Some are out of date, others limited.
  - Source of conflicts and misunderstandings in the community.

- **Qt is not a graphics API / shading language/ etc. wrapper**
  - but may provide helpers to make some things easier.

Is Qt Quick OpenGL only?

- Not anymore.
- Qt 5.8 modularizes.
- Built-in software renderer.
  - Think low-cost embedded HW without GPU
- Experimental Direct3D 12 backend.
  - Why?
    - Work started before Vulkan was released.
    - Wanted to try a new, low-level API.
    - Direct3D has anyway been on the radar for a long time now.
- Nothing is as complete as the OpenGL backend however. Remains the default.
  - Particles, distance field text rendering, widget composition interop, etc.
  - All additional 3D modules (Qt 3D, Canvas 3D, Data Visualization) are OpenGL only atm.

Vulkan?

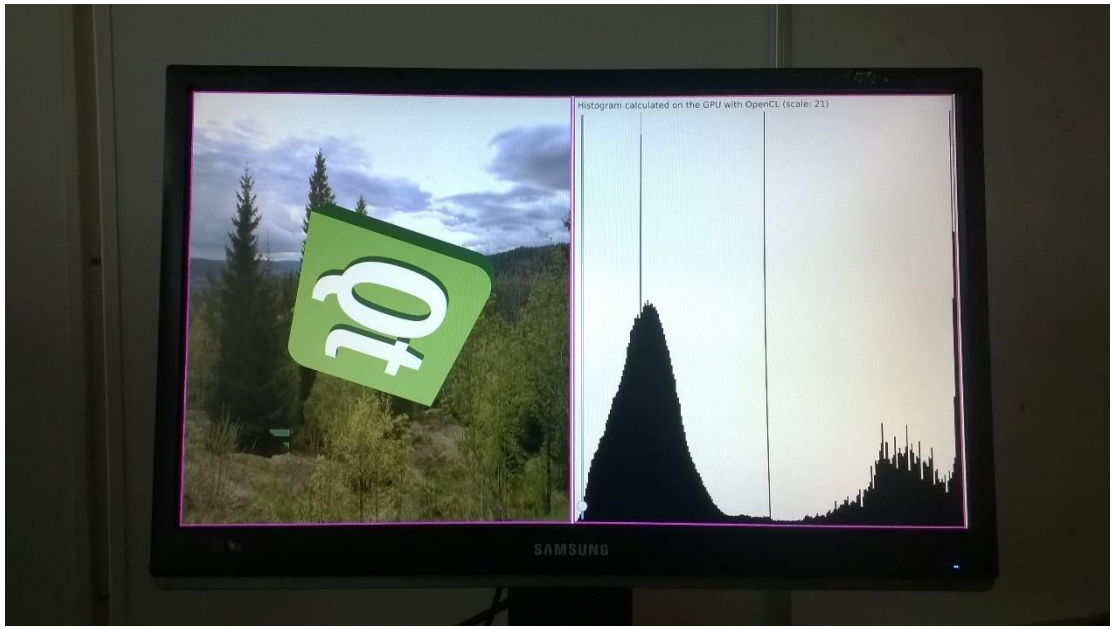
- Not yet.
  - Qt Quick will not be rushed. Limited benefits. Immense amount of work, too little reward. (proof: d3d12)
  - Some basic enablers (for C++ window/widget apps) maybe in 5.9.
- Nothing prevents anyone from integrating Vulkan content into a QWindow or even combine with QWidgets (by embedding a QWindow).
- Qt Quick is a bit problematic, but GL_NV_draw_vulkan_image should work

Some good stuff out there already:

https://github.com/alpqr/qtvulkan

OpenCL?

- Qt itself does not contain anything OpenCL-related.
- Integrating CL-GL interop with Qt Quick is pretty simple:
- https://blog.qt.io/blog/2015/04/20/qt-quick-with-the-power-of-opencl-on-embedded-linux-devices/



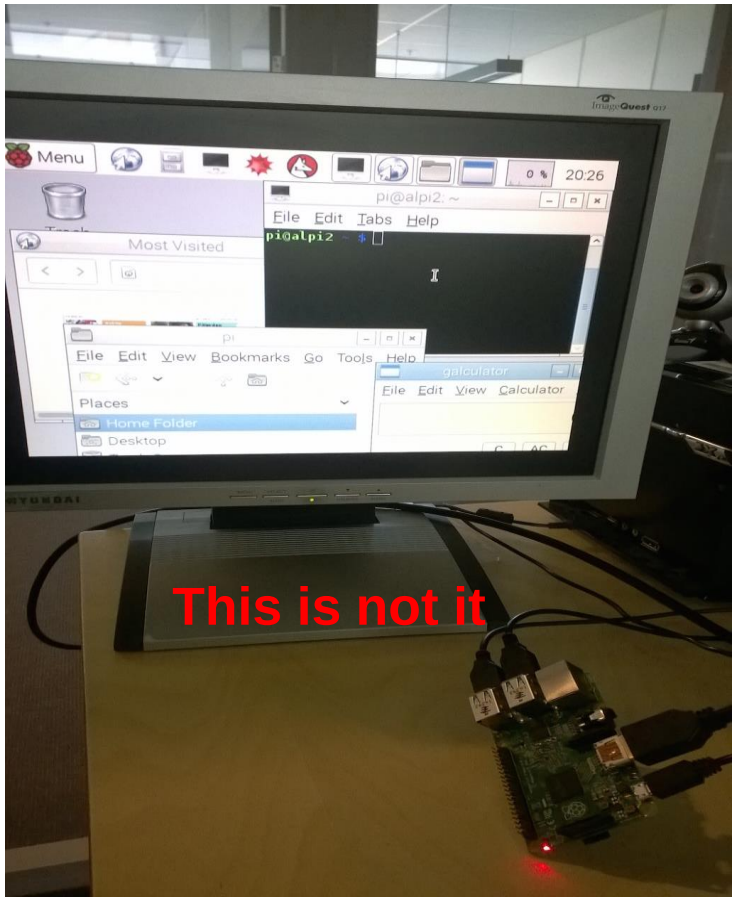
Running on an ODROID-XU3 with a Mali GPU

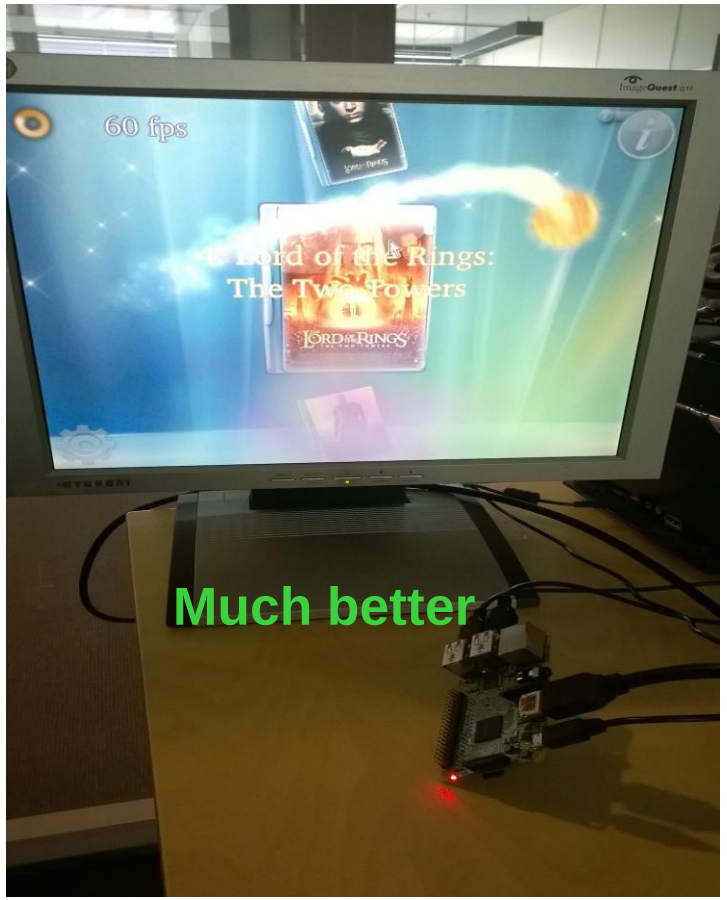
Windowing System Interfaces

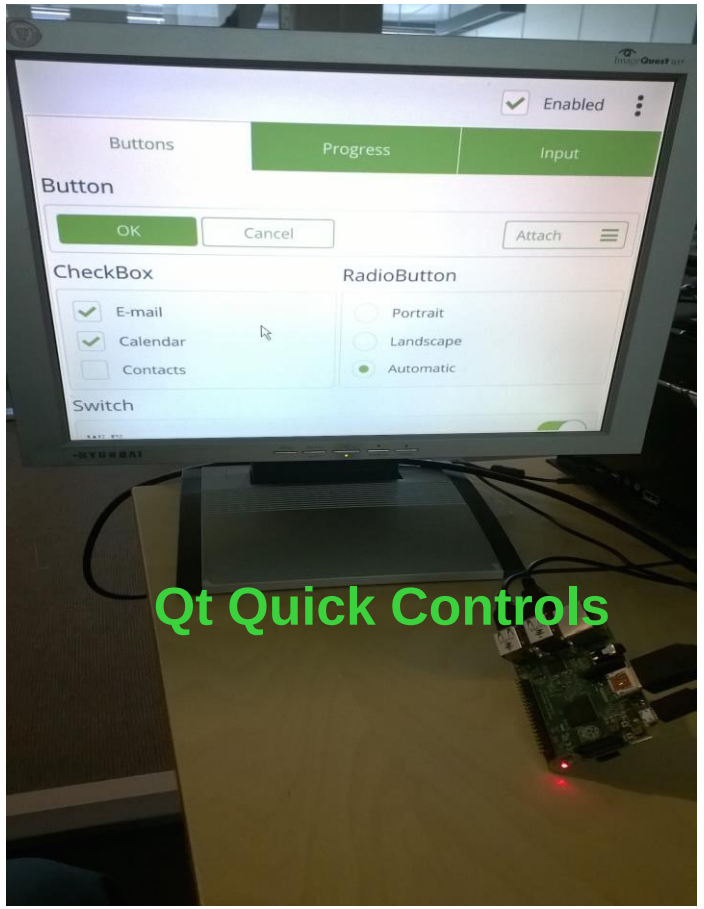
- Setting up rendering for OpenGL or other graphics APIs is non-trivial.
- EGL, GLX, WGL, Apple variants, DXGI, …
- Some platforms even have multiple ones.
  - Qt supports both GLX and EGL on X11.
  - WGL and EGL on Windows, the latter due to supporting ANGLE, the EGL/GLES-to-DXGI/D3D translator
    - And apps may want to choose the WSI and OpenGL implementation themselves in some cases…
    - It’s hard.
- Fortunately Qt hides all this from the application developer. The necessary glue for setting up OpenGL contexts and window surfaces is done in Qt’s platform plugins.
- Modern OpenGL support improved greatly during the lifetime of Qt 5.
  - Creating versioned, profile-based OpenGL contexts, requesting debug context, ARB_ESn_compatibility, …

Windowing Systems on Embedded

› Enter embedded device creation and it all gets even more “fun”. Different needs.







Windowing Systems on Embedded

- Running an ordinary desktop-ish environment is not suitable for device creation.
- Instead, the options are typically:
  - Run the app in fullscreen directly on the framebuffer or KMS/DRM or a vendor-specific composition API.
  - Use Wayland and run a compositor like above and all the other apps as Wayland clients.
  - Use X11 with or without a window manager and customize.
- Some devices may not have a GPU. But they can still
  - output QWidget or software rendered Qt Quick content into the framebuffer
  - or use 2D acceleration for better blitting, when available (DirectFB or vendor-specific APIs)

Windowing Systems on Embedded

- Say we have EGL and OpenGL ES. How do we get something to the screen?
- EGL can give you a window surface for a native window. But what is a native window?
  - For EGL implementations on top of fbdev the native window is either a vendor-specific struct or something created by vendor-specific functions.
  - For graphics stacks built on top of a system compositor, you need to create a layer or similar yourself using the proprietary APIs. (e.g. Dispmanx on the Raspberry Pi)
  - Kernel modesetting + Direct Rendering Manager. Besides trying to be “standard”, this is also great because it has proper connector and output management. (multiple displays can be surprisingly hard on embedded)
  - But the buffer management approach can vary: GBM versus EGLDevice/EGLOutput/EGLStream.
- Qt’s windowing system-less, fullscreen EGL/GLES platform plugin, eglfs, has multiple backends (plugins to the plugin…) to provide support for the above for various boards.
- Vivante (GCnnnn), ARM (Mali), Broadcom (RPi), Mesa (Intel/Nouveau/Radeon), NVIDIA automotive, …

Wayland

- A protocol for creating compositors.
- Weston is a reference implementation for a compositor.
- To create a Wayland-based system, support from the graphics drivers is necessary.
  - Glue.
  - Graphics buffer sharing mechanism.
- QtWayland offers a platform plugin to run Qt apps as Wayland clients and also a library to implement custom compositors with C++ or even QML.
  - Flexibility, customizability, using the full design and animation capabilities Qt Quick offers
- Surprise surprise: multiple backends for the different buffer sharing approaches…
  - EGLImage as used by Mesa and some other vendors
  - EGLStream (NVIDIA)
  - Custom stuff, e.g. Broadcom extensions for RPi

Summary

› Doing graphics and *setting up graphics* is hard.
› Especially when the apps may need to run on multiple platforms.
› Use Qt. Stay sane.

Thank You!

www.qt.io