Towards Qt 6

Lars Knoll
Qt Chief Maintainer and CTO

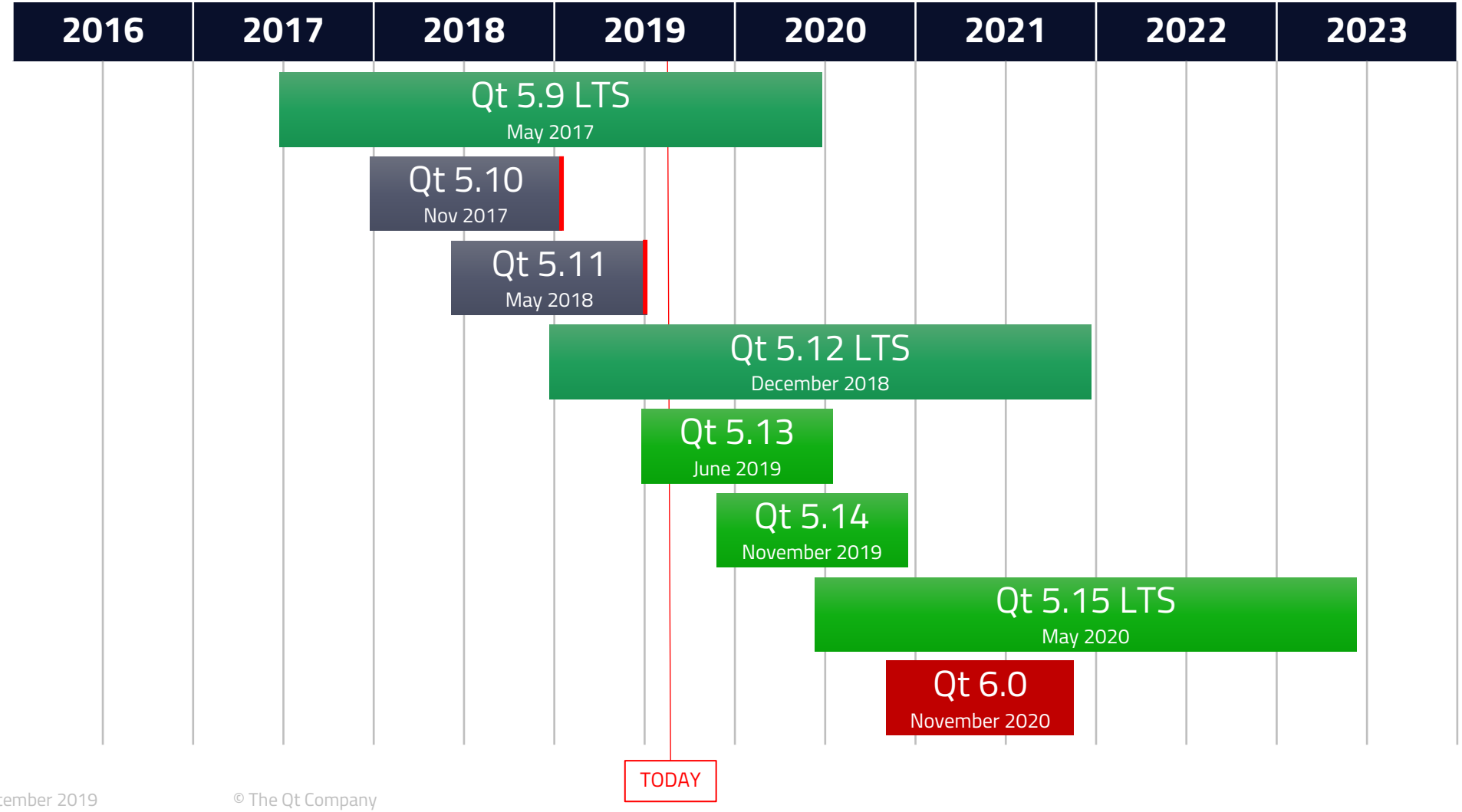
Akademy, 7. September 2019

Motivation

- Qt 5.0 was released in 2012
  - Somewhat in a haste after Nokia
- Early Qt 5 years used to
  - Prove Qt is still viable
  - Expand to Mobile and Embedded
  - Largely expand the functionality

- After 7 years it's time to
  - Adapt to changes in the world around us
  - Clean up and re-architect our core
  - Prepare for what will be important in the years to come

Roadmap to Qt 6



Qt Core values

1. Cross-platform
2. Scalability
3. Ease of use
4. World class APIs, tools and documentation
5. Maintainability, stability and compatibility
6. A large developer ecosystem

A new version of Qt needs to adjust our product to new market demands while keeping those values at the heart of what we're doing.

Goals for Qt 6

- Growth will mostly come from embedded, connected devices
  - Scale down to low end Hardware
  - But also scale up to high-end projects
- Desktop and mobile has lots of the developer mindshare
  - Great support there is a pre-requirement to be able to grow also in other markets
- An integrated development workflow for everybody
  - UX designers
  - Developers
  - QA engineers
  - ...
- Strengthen the ecosystem around Qt
  - Capture new developers
  - Provide good infrastructure

Goals for Qt 6.0

- Qt 6.0 is the place where we can
  - Break Binary Compatibility
  - Break some level of Source Compatibility
- Use the opportunity to
  - Prepare Qt for the requirements coming in 2020 and beyond
  - Clean up our own code base and keep things maintainable
  - Do architectural changes that we can’t do later

Focus areas

- Compatibility with Qt 5
- House cleaning
- Build System
- API quality and simplification
- Graphics
- QML
- Unified Tooling

Compatibility with Qt 5

- Keep porting effort from Qt 5 to Qt 6 minimal
- Break Binary Compatibility
  - Applications will need to recompiled
- Limit source incompatible changes
  - Each change requires effort from our users
- Avoid behavioral changes if possible
  - Unless they can be detected at compile time

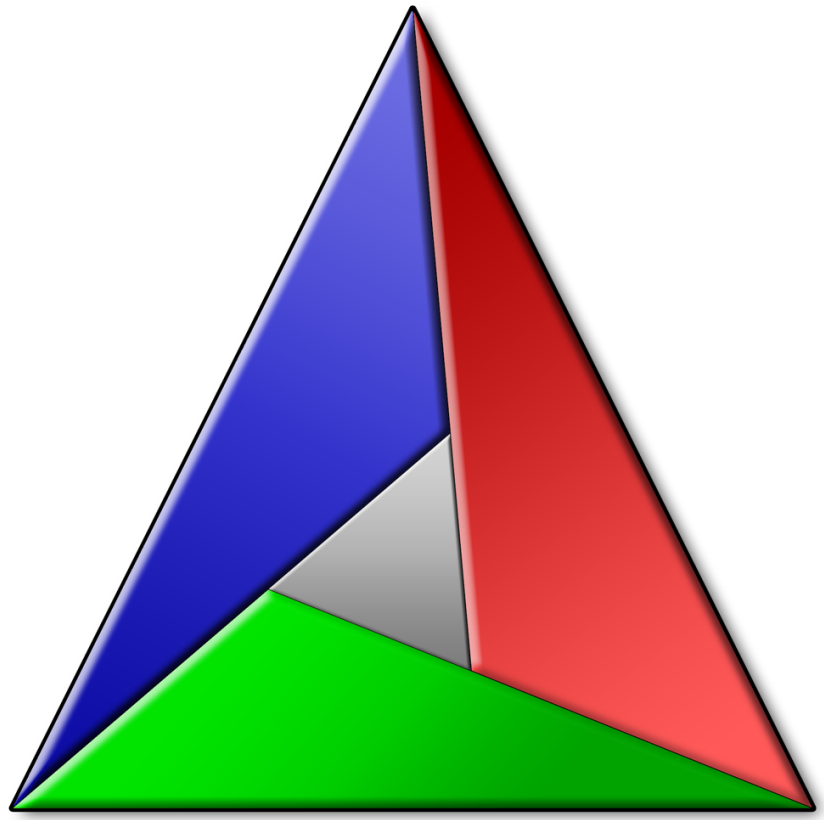
Goal: Allow compiling applications against both 5.15 and 6.0 with minimal workarounds

House cleaning

- Remove all functionality deprecated in Qt 5
  - Modules, Classes and Methods
- Modules that will get removed
  - Qt Script
  - Qt Canvas 3D
  - Qt Quick Controls 1
  - Qt Quick 1
- Some classes removed from Qt 6 **might** live on in a qt5compat module
  - QRegExp -> Qt5::QRegExp
  - QList -> Qt5::QList

CMake

The build system for Qt 6



17

Intuitive and simple APIs

- Qt is known for making C++ easy and accessible
  - Different goal than standard C++
  - One of the main arguments for using Qt
- Ease of use just as important as performance
  - 90% of user code is not performance critical
  - Acceptable: Some extra CPU cycles for simpler APIs
  - Not acceptable: Change in algorithmic complexity
- Ideally have both a simple API and great performance

Unicode throughout

- Source code in UTF-8
- All text handling assumes Unicode (UTF-8, UTF-16 or UTF-32)
  - All conversions from const char * to/from QString will assume UTF-8
- QTextCodec not part of Qt Core anymore
  - Move all encoding conversions (except between UTF-X formats) into own library
- Simplify our String classes
  - Simple string literals in source code
  - QString
  - QStringView
  - QByteArray ASCII only
  - No more QStringLiteral, QLatin1String, QStringRef
- Can QString support both UTF-8 and UTF-16?
  - Or does it break too much source compatibility?

Rethink our containers

- Unify QList and QVector
  - Almost 100% Source Compatible
  - Simplify our API: One less class to worry about
  - Improve performance in most cases
- Base QSet, QMap and QHash on std containers?
  - Better interoperability
  - Keep Qt APIs
  - Simplify our maintenance
- Important to keep an easy to use API

Next generation QML

Performance and memory consumption problems

- QML requires a full JS engine and garbage collector
  - Not well suited for very low-end devices
  - GC leads to some unpredictable performance characteristics
- Object model built on top of Qt Object model
  - Duplicated data structures
  - Huge amount of mallocs
  - Large initialization overhead (runtime, not compile time)
- Binding propagation is immediate, not synced with scene graph
    - Leads to duplicated binding evaluations
- Weak typing and runtime resolution of dependencies
  - Generated code has to be generic

Language problems

- QML Scoping rules are difficult to understand
- Integration with C++ is not ideal
- QML versioning is inconsistent with C++ and intuition
  - Maintaining compatibility in C++ is much easier than in QML
- No private properties / Encapsulation is hard
- Weakly typed
  - Compiler can’t catch many coding errors
  - Refactoring is difficult
- Code model is not fully populated
  - C++ types only become visible to the code model when separately declared in .qmltypes files.
  - Context properties and dynamic type registration need heuristics

Changes for Qt 6

1. Introduce strong typing.
2. Make Javascript an optional feature of QML.
3. Remove QML versioning.
4. Remove the duplication of data structures between QObject and QML
5. Avoid runtime generated data structures
6. Support compiling QML to efficient C++ and native code
7. Support hiding implementation details
8. Better tooling integration

Improve the Qt type system

- Better type information
- Generate type information at compile time
  - Changes to QMetaType and how we register that data
- Additions to meta object system
  - Add data required by QML
- Fast lookup of meta type information
  - No String based lookups for types
  - Get rid of O(inheritance depth) complexity in lookups

→ get rid of QML property cache runtime data structures

New property system

- Real property based syntax
  - No pseudo properties with foo()/getFoo()
- Built-in support for bindings
  - Move support for bindings into Qt Core
  - Make them available for all of Qt
- Easy to use C++ API
- Removes lots of glue code in C++
- Lazy binding evaluation
- Significant reduction in memory allocations for QML
- Extremely fast

Property Syntax

```
struct Item {
    Property<int> width;
    Property<int> height;
    Property<int> border;

    Item() {
        height.setBinding(width);
        border.setBinding([this]() {
            return width / 10;
        });
    }
}
```

Compiled QML

- Compile QML files to native code by default
- With full type information we can generate efficient C++
  - Compile into C++ classes
  - QML property → QProperty
  - QML bindings → C++ bindings

Split current Qt QML

- Qt QML becomes a very small runtime for compiled QML
- Split out other parts
  - Object model → Qt Core
  - Compiler and code model → Own library
  - Item models → Own plugin or Qt Quick
  - Debugging/profiling framework → Partly Qt Core
  - JavaScript runtime → Own library
  - Compiled QML (-> C++/assembly) → QML compiler tool
  - Runtime for dynamic QML (-> bytecode) → Own library

Separate JavaScript module

- Qt QML does not depend on it
- Allow for slightly restricted QML that can be fully compiled to C++
  - Won’t require JavaScript
- JavaScript still required for Canvas, WorkerScript, etc.
  - Move these APIs to a module of it’s own
- Remove all JavaScript dependencies from Qt Quick
  - And dependent libraries
- Extend current QJS* API
  - Turn into full Qt Script replacement

Major changes to the QML language (QML 3)

- Remove context properties
- Simpler lookup/scoping rules
  - Will require explicit usage of the id in a few more places
- Remove QML versioning
  - Becomes unnecessary with better lookup/scoping rules
- Strongly typed
- Compilable subset of JavaScript/TypeScript for bindings and functions
- Add "private" and "public" as qualifiers for properties

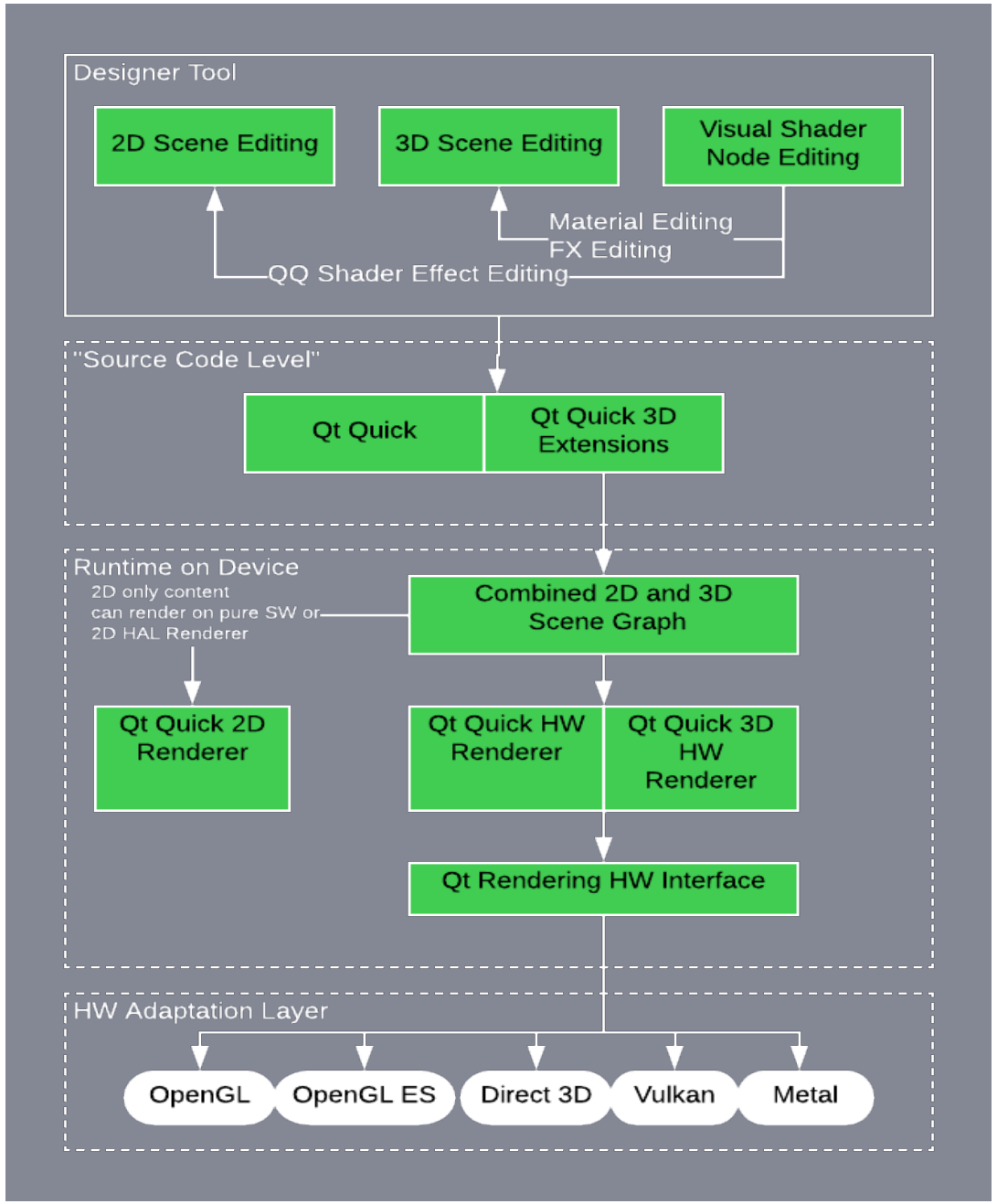
Compatibility with current QML (QML 2)

- Need to differentiate between old and new QML
  - File extension, pragma or import
- QML 2 will require dynamic QML runtime and JavaScript
- Interpreted (JIT'ed) at runtime
- QML 2 and QML 3 can use the same Qt Quick library

Benefits

- Move most initialization work from runtime to compile time
  - Faster application startup
- Significantly faster binding engine
  - Initial benchmarks show 10-100 times faster binding evaluation
- Large reduction in memory consumption
  - Reduce number of dynamic memory allocations
  - Less runtime generated data
- Better C++ integration
- Better Tooling integration
  - Language server

New graphics architecture



Graphics API independence

Qt Rendering Hardware Interface (RHI)

- Abstraction layer for 3D graphics APIs
- Tuned towards the needs of Qt
- Builds on top of Qt Platform Abstraction API
- Supported graphics APIs
  - OpenGL 2.1 or OpenGL ES 2.0 or newer
  - Vulkan 1.0/1.1
  - Direct3D 11.1
  - Metal

- **Qt 5.14:** Technology preview supporting Qt Quick
- **Qt 5.15:** Full support for Qt Quick (2D)
- **Qt 6.0:** Used in all of Qt

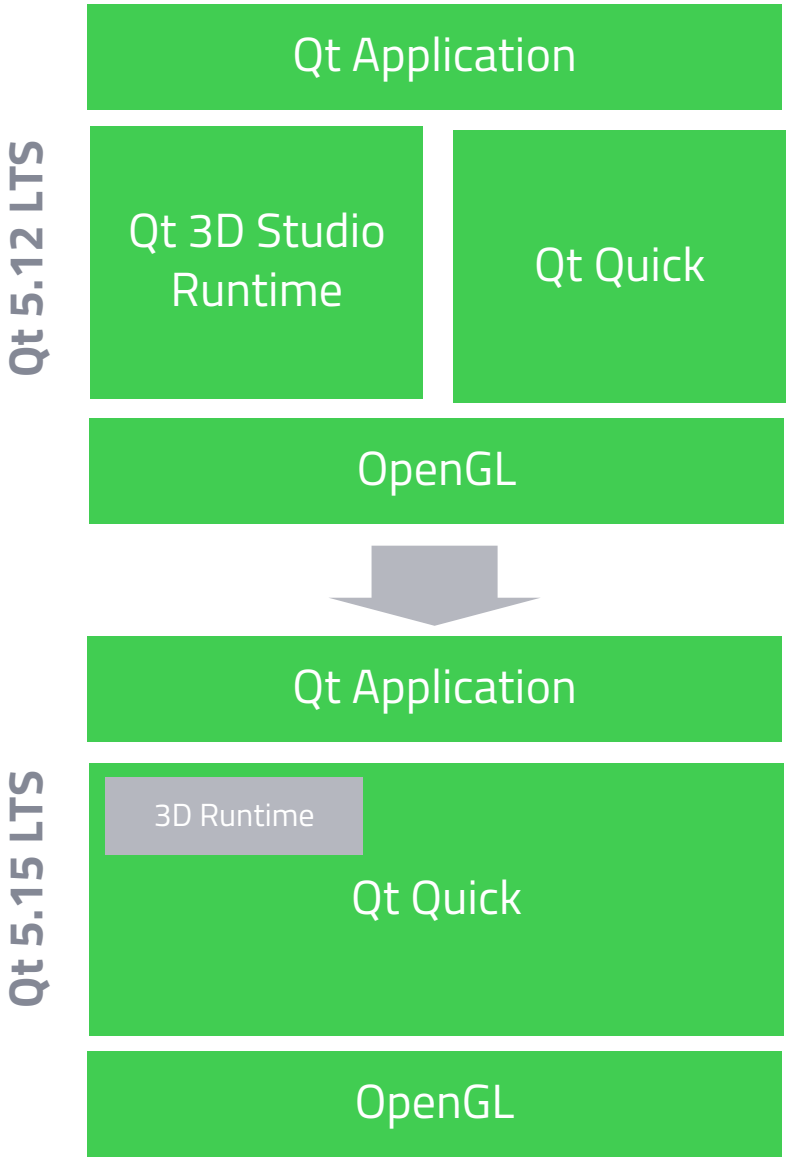
Qt Shader tools

- Write a shader once, use with all graphics APIs
- Vulkan style GLSL
- Create a cross-platform shader package
- Usable with RHI
- Based on top of SPIR-V toolchain
  - Open source
  - Backed by Khronos group

Qt Quick 3D – Unified 2D and 3D

- Introducing a spatial renderer for Qt Quick
  - Use existing functionality of Qt Quick and Qt
  - No parallel implementation as with Qt 3D Studio
- Unified architecture
  - Once scene graph
  - Unified 2D and 3D graphics
- Efficient resource usage and good performance
  - No duplicated resources
  - No need for frame buffer objects in most cases
  - Larger performance improvements over current solutions

- **License: GPLv3**



Qt Quick 3D in Brief

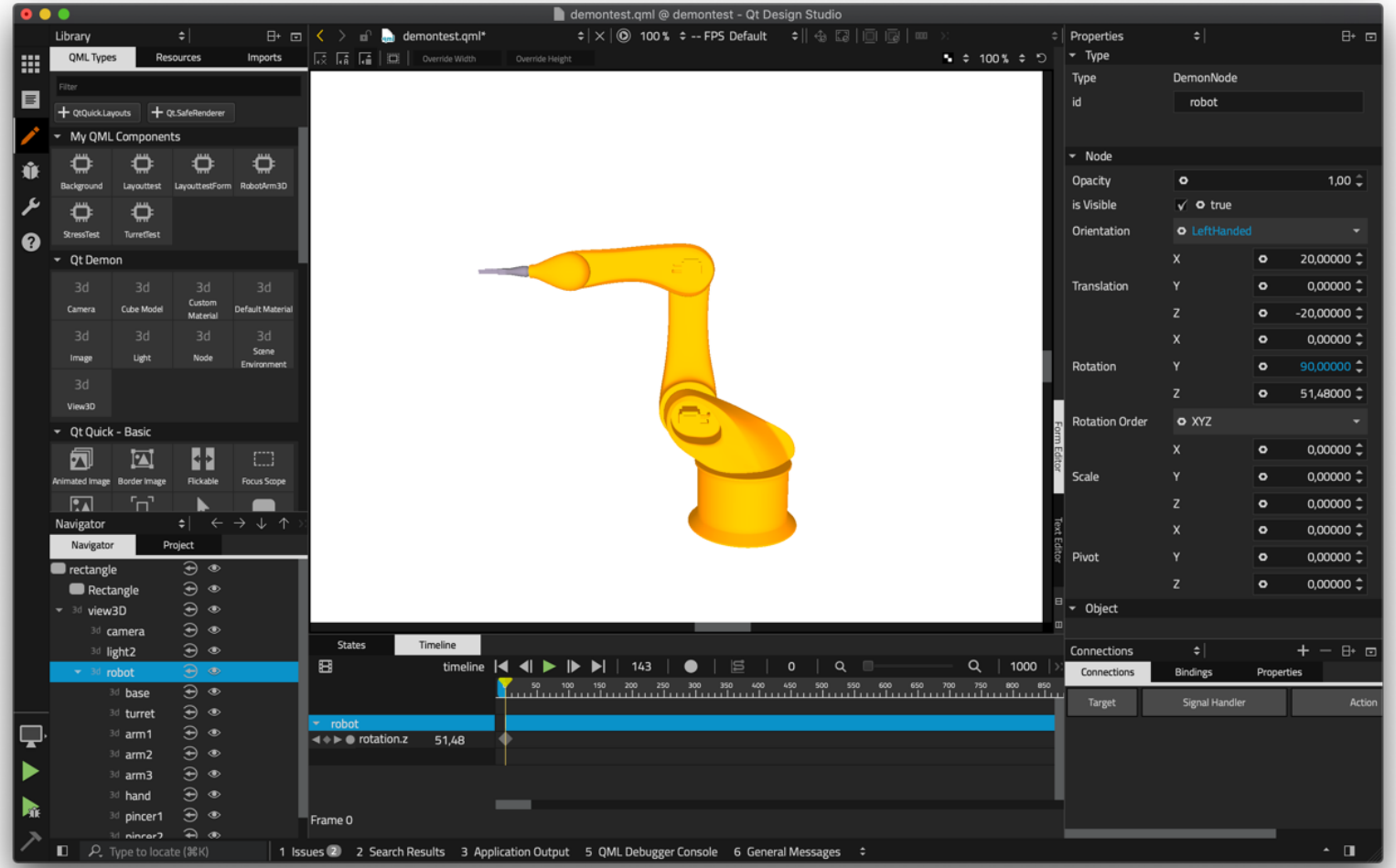
- Build on QML
  - All QML language features available
  - Create re-usable components
  - Unified animations and state transitions for 2D and 3D
- Use with all existing Qt Quick APIs
  - Simply another import
- High-level API for 3D content
  - Focus on ease of use

- **Qt 5.14:** Technology Preview (requires OpenGL)
- **Qt 5.15:** Fully supported (requires OpenGL)
- **Qt 6:** Supported on top of RHI

```
Window {
    id: window
    width: 640
    height: 640
    visible: true
    color: "black"

    Image {
        source: "qt_logo.png"
        x: 50
        SequentialAnimation on y {
            loops: Animation.Infinite
            PropertyAnimation { duration: 3000; to: 400; from: 50 }
        }
    }

    View3D {
        id: layer1
        anchors.fill: parent
        anchors.margins: 50
        camera: camera
        renderMode: View3D.Overlay

        environment: SceneEnvironment {
            probeBrightness: 1000
            backgroundMode: SceneEnvironment.Transparent
            lightProbe: Texture {
                source: "maps/OpenfootageNET_garage-1024.hdr"
            }
        }
        Camera {
            id: camera
            position: Qt.vector3d(0, 200, -300)
            rotation: Qt.vector3d(30, 0, 0)
        }
        Model {
            position: Qt.vector3d(0, 0, 0)
            source: "#Cube"
            materials: [ GlassMaterial {
                }
            ]
            rotation: Qt.vector3d(0, 90, 0)

            SequentialAnimation on rotation {
                loops: Animation.Infinite
                PropertyAnimation { duration: 5000;
                    to: Qt.vector3d(0, 360, 0);
                    from: Qt.vector3d(0, 0, 0) }
            }
        }
    }
}
```

Unified tooling

- **Qt 5:** Two graphical tools to create User Interfaces
  - Qt Design Studio
  - Qt 3D Studio
  - Separate and independent applications
- **Qt 6:** Base all our graphical tooling on Qt Creator
  - Qt Design Studio is Qt Creator with a designer centric UI
  - Merge all 3D design functionality back into Qt Creator/Design Studio
    - Based on QML and Qt Quick 3D



Product structure

Open Source and commercial business

- Qt is a very large product
  - All of Qt for Application Development is FOSS
  - Most of Qt for Device Creation is FOSS
- The Qt Company funds most Qt development
  - 140 engineers working full time on Qt
  - Could need some more to properly cover all parts
- FOSS/commercial licensing has to be balanced to allow
  - Growth of business to fund future development
  - Growth of ecosystem
- Ideal licensing structure
  - LGPLv3 for essential frameworks
  - GPLv3 for Add-ons and tools
  - Commercial licensing for integrations with commercial tools/frameworks

Smaller core product

- Smaller set of core libraries
  - Mostly essentials
  - Core, Network, Widgets, QML, Quick
- Developer and Designer Tools
  - Qt Creator
  - Qt Design Studio
- Deliver add-ons separately
  - More flexible release schedule
- Simplifies creation of Qt releases

Marketplace

- A central rallying point for the Qt ecosystem
- Give everybody a place to publish their own additions to Qt
  - Both free and paid content
- Easy discovery of additional functionality
- Focus on extensions to Qt
  - Extensions to our tools
  - Additional frameworks
  - Content that integrates with Qt
    - Graphical assets (3D models, imagery, styles, etc)
    - AI models, etc.
- Deliver some of Qt's own Add-ons through the Marketplace
  - Decouple release schedules

Thanks!

lars.knoll@qt.io
@larsknoll