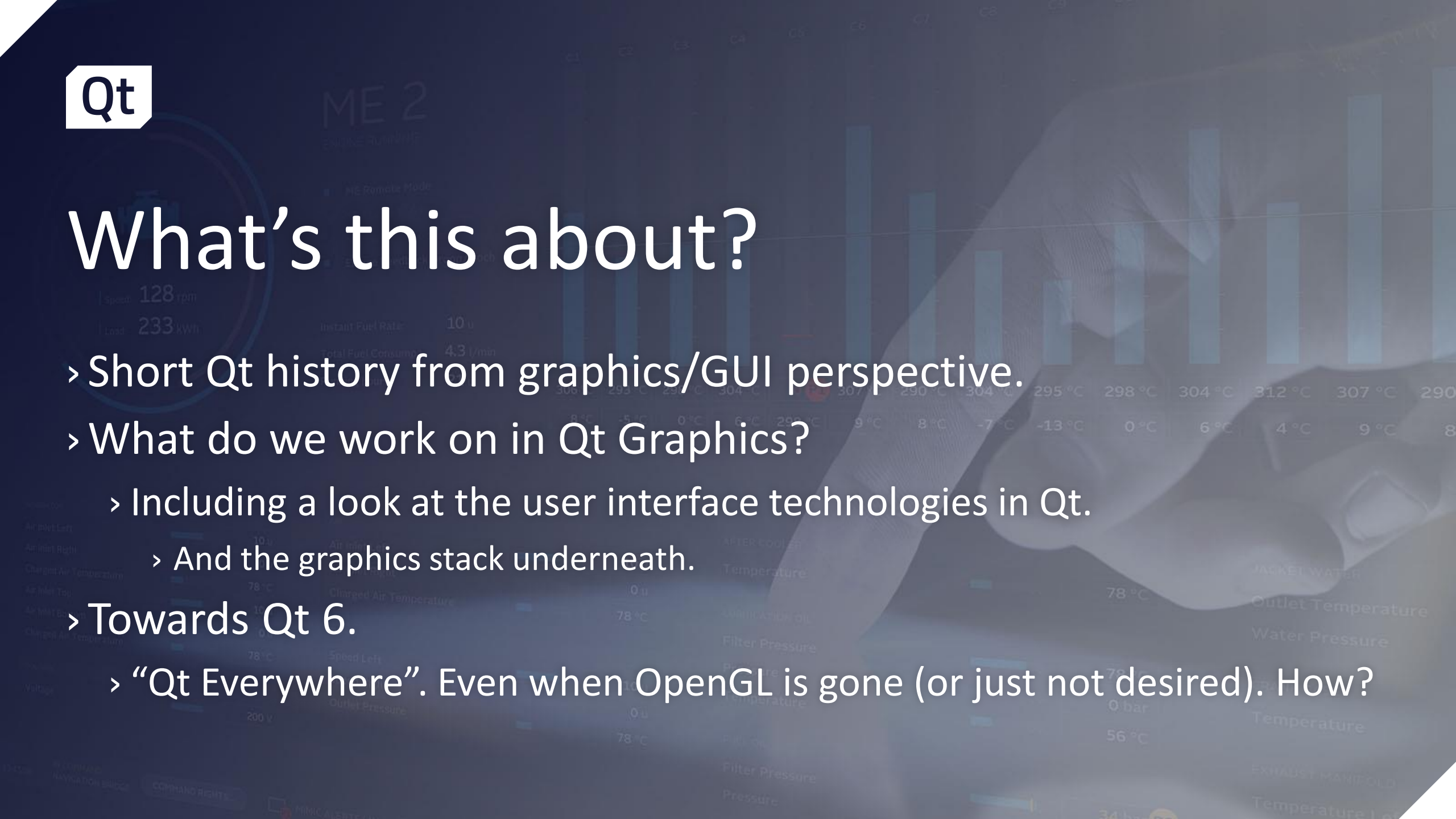
The Evolution of Accelerated 2D and 3D Graphics in Qt

László Agócs
Principal Software Engineer
The Qt Company, Oslo, Norway

05/09/2019 NDC TechTown, Kongsberg, Norway



What’s this about?

- Short Qt history from graphics/GUI perspective.
- What do we work on in Qt Graphics?
  - Including a look at the user interface technologies in Qt.
    - And the graphics stack underneath.
- Towards Qt 6.
  - “Qt Everywhere”. Even when OpenGL is gone (or just not desired). How?

Who am I?

- Working with Qt since ca. 2009.
- In Oslo since 2013.
  - Graphics & Multimedia team in Digia / The Qt Company.
- Previously at Nokia (Symbian. (yay!) UI frameworks. Qt on Symbian.), and ARM (OpenGL ES for Mali).
- Twitter: @alpqr
- IRC: lagocs on #qt-* on Freenode

Qt

Developers Blog English Contact Us Download. Try. Buy.

Qt

Products Professional Services Resources Customers Company

Application UIs & Software

Develop cross-platform UIs & applications.

Embedded Devices

Easily create powerful & connected devices.

Solutions

- Automotive
- Automation
- Medical
- Consumer Electronics
- Internet of Things
- Mobile Apps

Framework

- What is Qt
- Libraries, Tools & IDE
- UI design technologies
- Embedded features
- Reference software stack & SDK
- Supported platforms

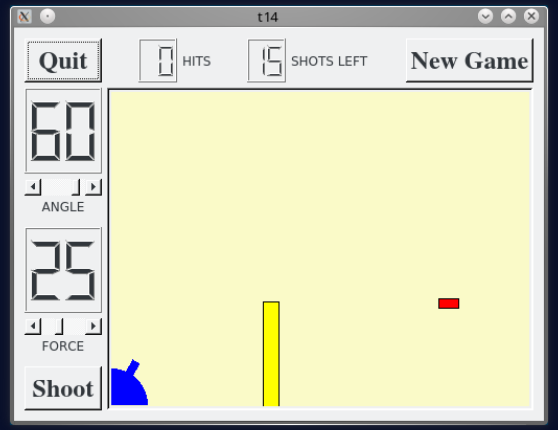
Software development made smarter

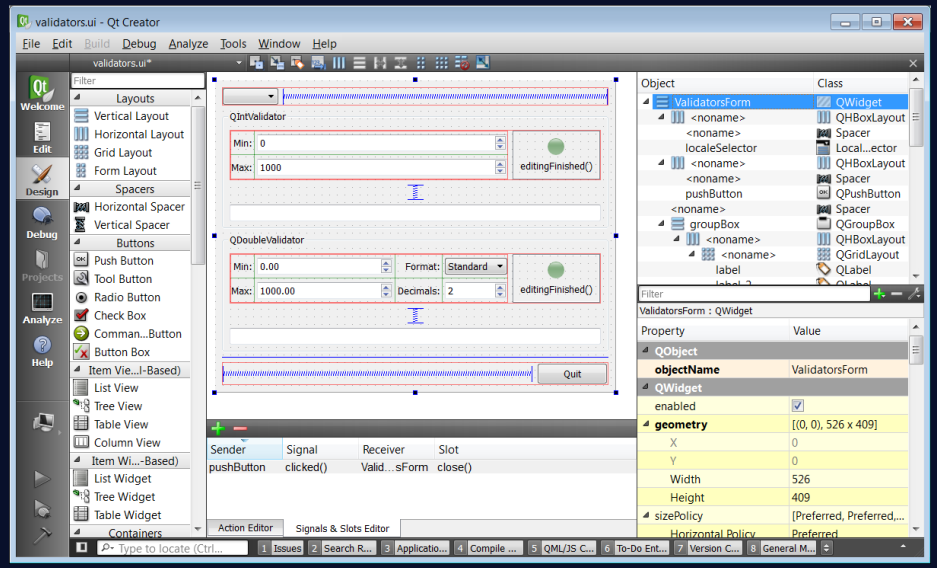
Create fluid, high-performance and intuitive UIs, applications, and embedded devices – with the same code base for all platforms.

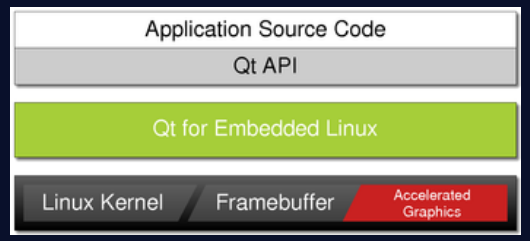

/kjuːt/
DEVELOPER
DAYS2014
EUROPE

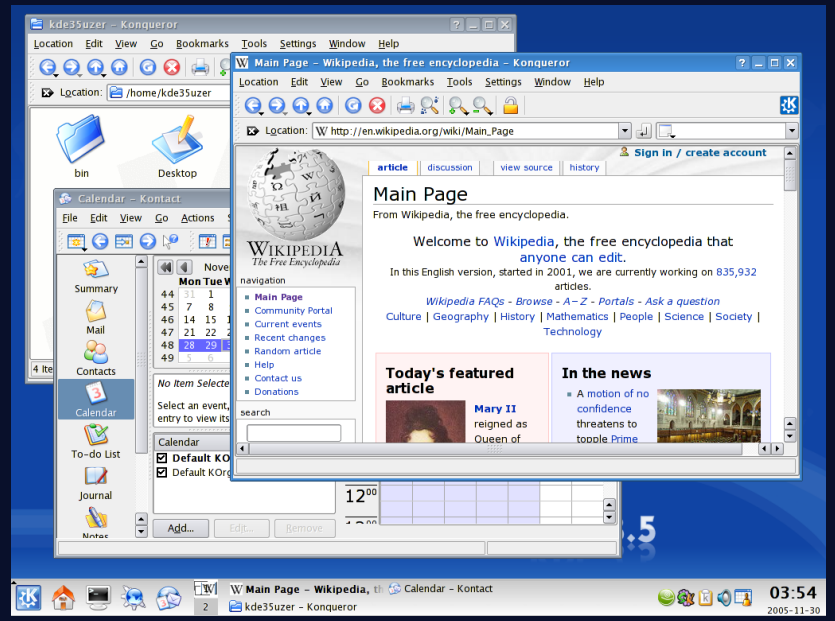
Qt history

- Trolltech (1994 – 2008)
  - Qt
  - Qt for Embedded
  - Qtopia



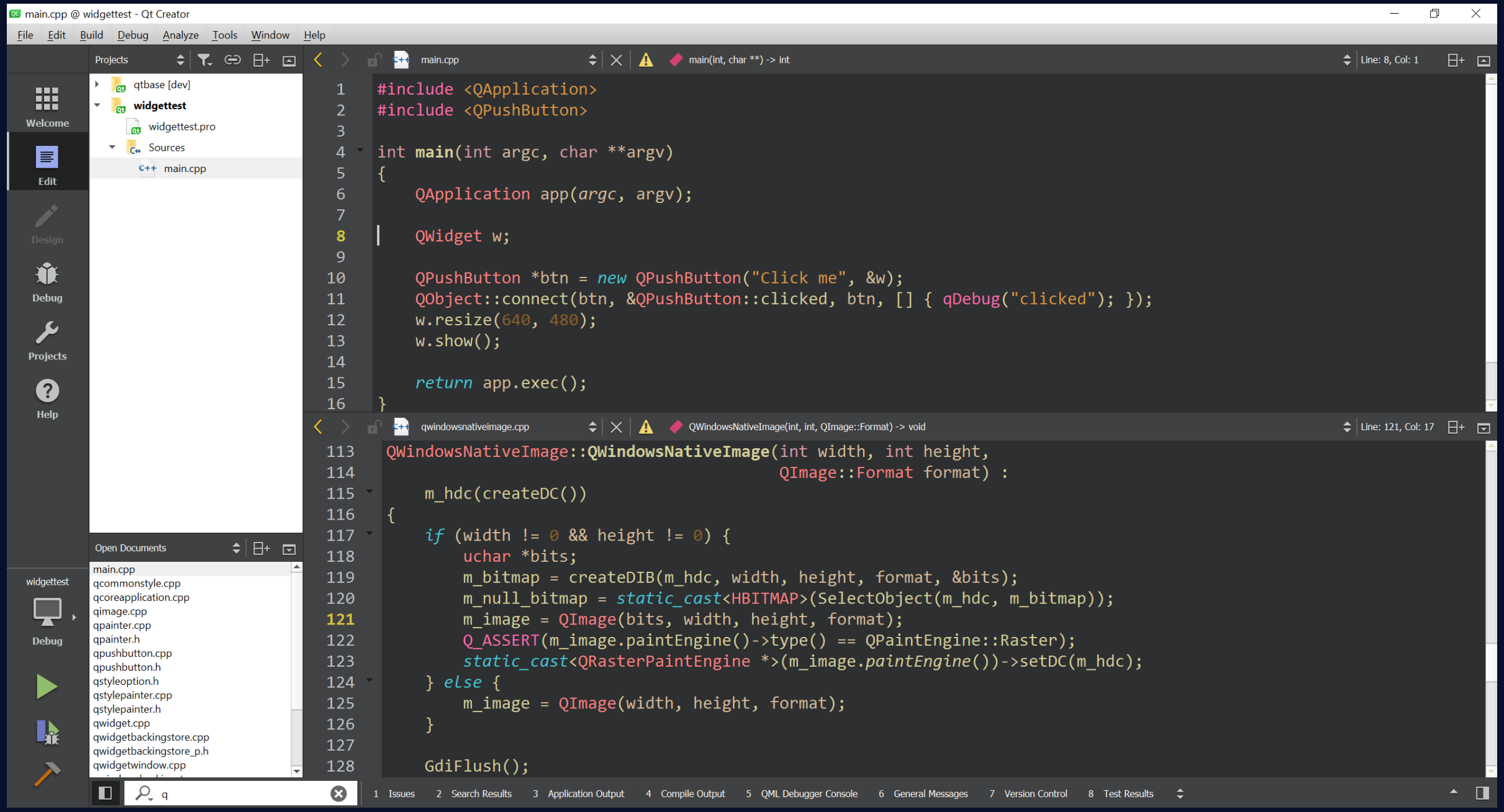






Let’s write a QWidget application

with some debugging to figure out how stuff ends up “on the screen”


main.cpp @ widgettest - Qt Creator
File Edit Build Debug Analyze Tools Window Help
Projects
qtbase [dev]
widgettest
widgettest.pro
Sources
main.cpp
Welcome
Edit
Design
Debug
Projects
Help
main.cpp
main(int, char **) -> int
Line: 8, Col: 1
#include <QApplication>
#include <QPushButton>
int main(int argc, char **argv)
{
QApplication app(argc, argv);
QWidget w;
QPushButton *btn = new QPushButton("Click me", &w);
QObject::connect(btn, &QPushButton::clicked, btn, [] { qDebug("clicked"); });
w.resize(640, 480);
w.show();
return app.exec();
}
qwindowsnativeimage.cpp
QWindowsNativeImage(int, int, QImage::Format) -> void
Line: 121, Col: 17
QWindowsNativeImage::QWindowsNativeImage(int width, int height,
QImage::Format format) :
m_hdc(createDC())
{
if (width != 0 && height != 0) {
uchar *bits;
m_bitmap = createDIB(m_hdc, width, height, format, &bits);
m_null_bitmap = static_cast<HBITMAP>(SelectObject(m_hdc, m_bitmap));
m_image = QImage(bits, width, height, format);
Q_ASSERT(m_image.paintEngine()->type() == QPaintEngine::Raster);
static_cast<QRasterPaintEngine *>(m_image.paintEngine())->setDC(m_hdc);
} else {
m_image = QImage(width, height, format);
}
GdiFlush();
Open Documents
main.cpp
qcommonstyle.cpp
qcoreapplication.cpp
qimage.cpp
qpainter.cpp
qpainter.h
qpushbutton.cpp
qpushbutton.h
qstyleoption.h
qstylepainter.cpp
qstylepainter.h
qwidget.cpp
qwidgetbackingstore.cpp
qwidgetbackingstore_p.h
qwidgetwindow.cpp
widgettest
Debug
1 Issues
2 Search Results
3 Application Output
4 Compile Output
5 QML Debugger Console
6 General Messages
7 Version Control
8 Test Results

Qt

qwindowsstyle.cpp — (anonymous namespace)::QWindowsStyle::drawPrimitive((anonymous namespace)::QStyle::Primitive

```
866             QColor bg_col = fropt->backgroundColor;
867             if (!bg_col.isValid())
868                 bg_col = p->background().color();
869             // Create an "XOR" color.
870             QColor patternCol((bg_col.red() ^ 0xff) & 0xff,
871                               (bg_col.green() ^ 0xff) & 0xff,
872                               (bg_col.blue() ^ 0xff) & 0xff);
873             p->setBrush(QBrush(patternCol, Qt::Dense4Pattern));
874             p->setBrushOrigin(r.topLeft());
875             p->setPen(Qt::NoPen);
876             p->drawRect(r.left(), r.top(), r.width(), 1);    // Top
877             p->drawRect(r.left(), r.bottom(), r.width(), 1); // Bottom
878             p->drawRect(r.left(), r.top(), 1, r.height());   // Left
879             p->drawRect(r.right(), r.top(), 1, r.height());  // Right
880             p->restore();
881         }
882         break;
883     case PE_IndicatorRadioButton:
884         {
885             QRect r = opt->rect;
886             p->save();
887             p->setRenderHint(QPainter::Antialiasing, true);
888
889             QPointF circleCenter = r.center() + QPoint(1, 1);
890             qreal radius = (r.width() + (r.width() + 1) % 2) / 2.0 - 1;
891
892             QPainterPath path1;
893             path1.addEllipse(circleCenter, radius, radius);
894             radius *= 0.85;
895             QPainterPath path2;
896             path2.addEllipse(circleCenter, radius, radius);
897             radius *= 0.85;
898             QPainterPath path3;
899             path3.addEllipse(circleCenter, radius, radius);
```

Widgets -> style -> QPainter -> QImage -> blit

(there are exceptions, as always)

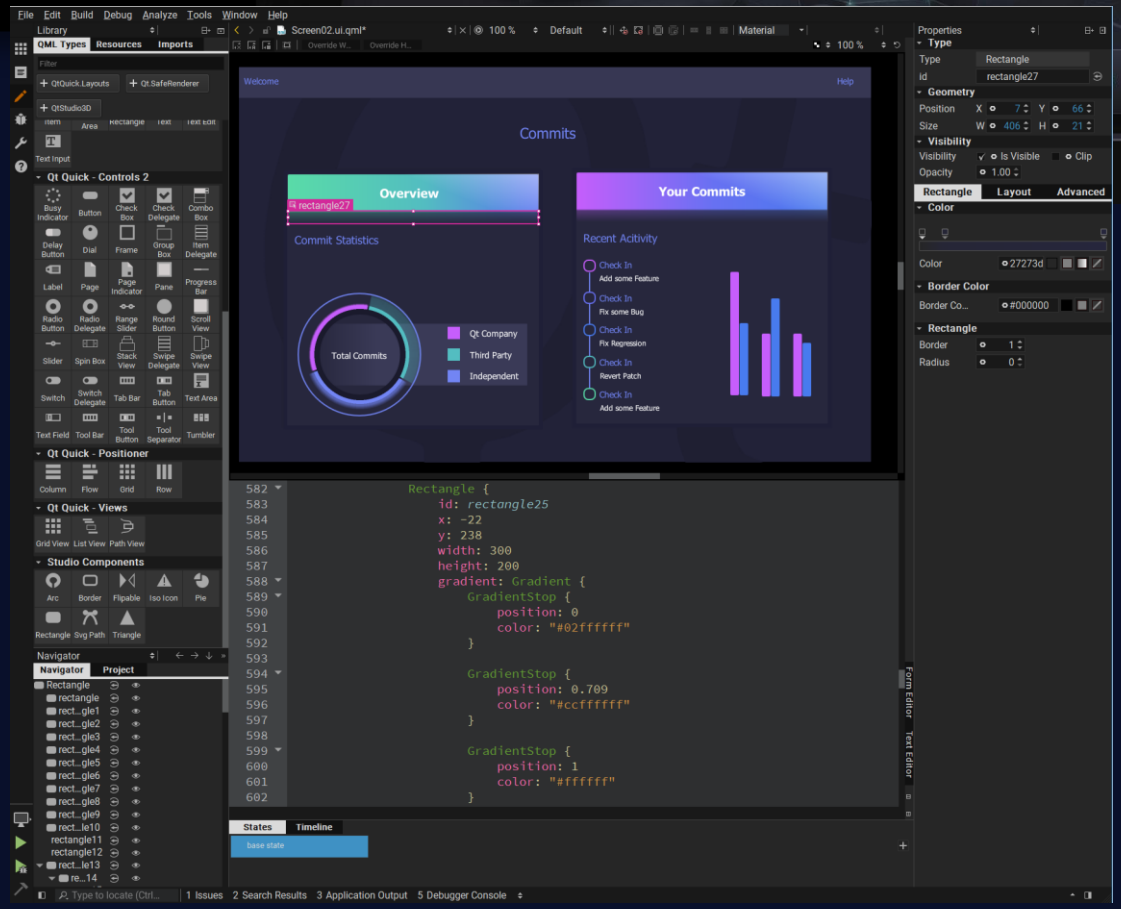
Qt history (2)

- Nokia (2008 – 2012)
  - Desktop
  - Symbian
  - Maemo/Meego
  - Other

A bit of mess with UI stacks... (widgets, GV, QQ1, QQ2)

Increased focus on
- fluid, touch friendly UIs
- HW 2D/3D acceleration
  - OpenVG (Symbian)
    - QPainter backend
  - OpenGL ES 2.0 (Meego, later Symbian)
    - QPainter backend
    - Qt Quick 2 designed for GLES2 from the start

Qt history (3)



› Digia (2012 – 2016)

› The Qt Company (2016 -)

- Android and iOS ports
- Modernizing many aspects
  - C++11/14/17, Wayland, …
- Increased focus on
  - commercial licensing
  - embedded (Linux, QNX, INTEGRITY, …)
  - designer friendly tooling

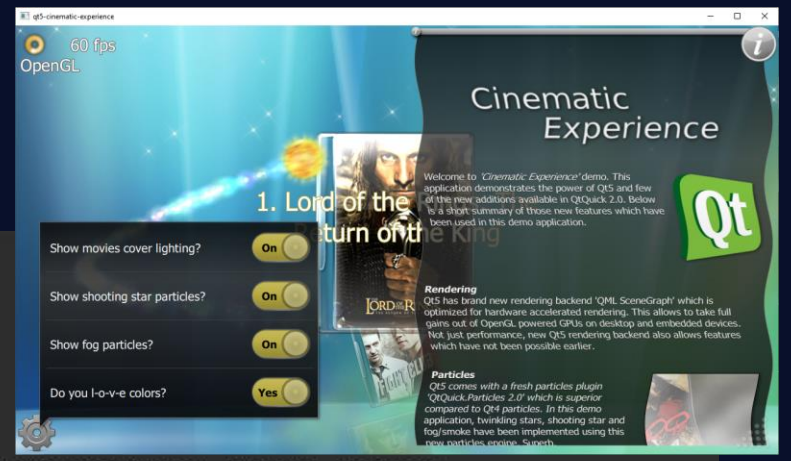
Let’s write a Qt Quick application

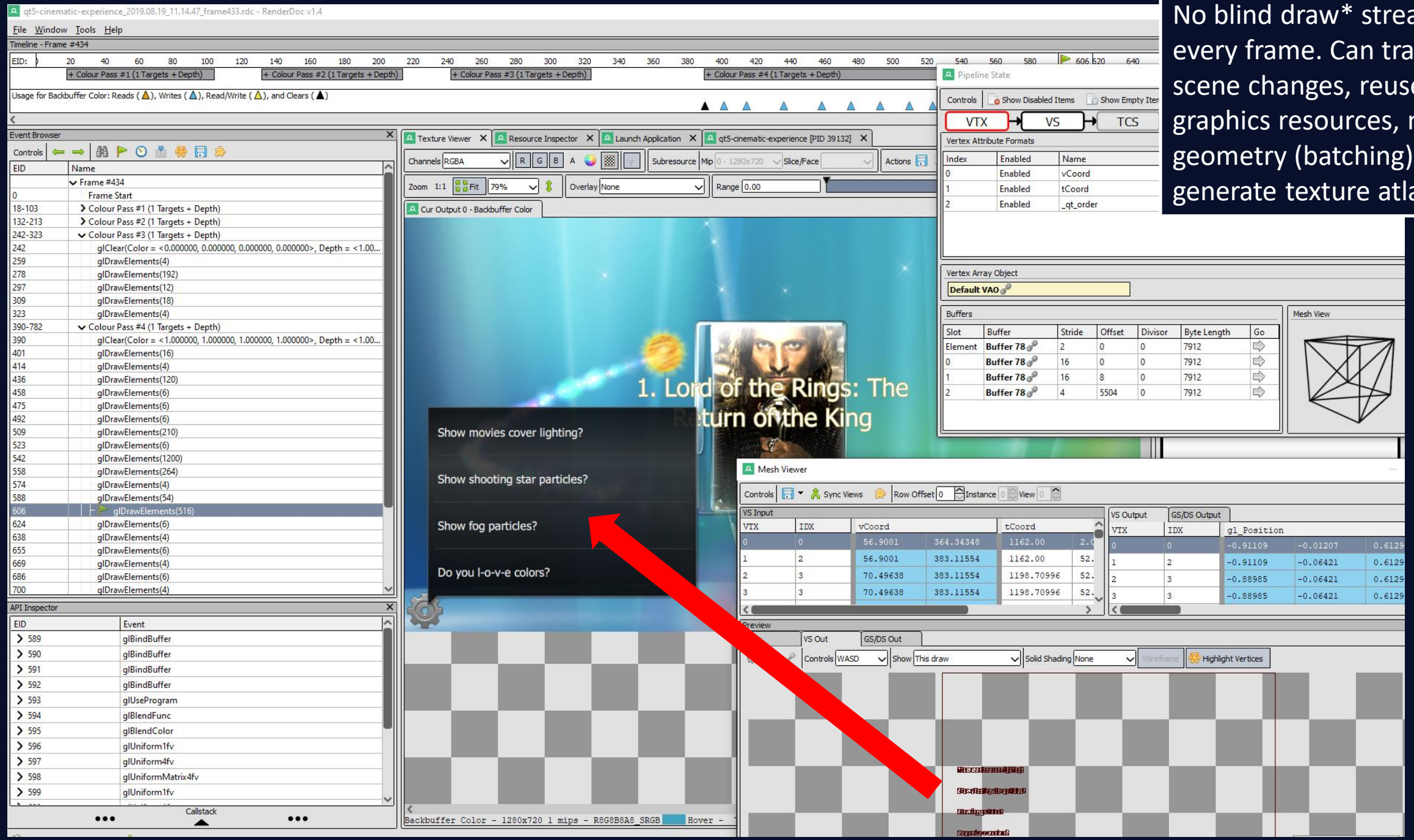
with some debugging to figure out how stuff ends up “on the screen”

Qt

```
[ ] RootNode 0x7fe48800f310  "" ) 1
 [ ] TransformNode( 0x7fe48800f390 identity "QQuickItem(QQuickRootItem:)"  ) 0
  [ ] TransformNode( 0x7fe488015fc0 identity "QQuickItem(QQuickItem:)"  ) 0
   [ ] TransformNode( 0x7fe488016220 identity "QQuickItem(MainView_QMLTYPE_8:)"  ) 0
    [ ] TransformNode( 0x7fe48808f450 identity "QQuickItem(QQuickItem:)"  ) 0
     [ ] TransformNode( 0x7fe48808f620 identity "QQuickItem(QQuickItem:)"  ) 0
      [ ] TransformNode( 0x7fe4881b70e0 identity "QQuickItem(QQuickImage_QML_2:)"  ) 0
       [ ] GeometryNode( 0x7fe4881b7c20 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 1280 y2= 720 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
      [ ] TransformNode( 0x7fe4881b7290 identity "QQuickItem(QQuickParticleSystem:)"  ) 0
       [ ] TransformNode( 0x7fe4881b7480 identity "QQuickItem(QQuickImageParticle:)"  ) 0
       [ ] TransformNode( 0x7fe4881b7620 identity "QQuickItem(QQuickParticleEmitter:)"  ) 0
      [ ] TransformNode( 0x7fe48808f730 identity "QQuickItem(QQuickListView_QML_9:)"  ) 0
       [ ] TransformNode( 0x7fe4881ad830 translate 0 220 0 "QQuickItem(QQuickItem:)"  ) 0
        [ ] TransformNode( 0x7fe4881ad910 det= 0.444444 "QQuickItem(DelegateItem_QMLTYPE_0:)"  ) 0
         [ ] OpacityNode( 0x7fe4881ada10 opacity= 0.5 combined= 1  "" ) 1
          [ ] TransformNode( 0x7fe4881adf20 identity "QQuickItem(QQuickMouseArea:)"  ) 0
          [ ] TransformNode( 0x7fe4881ae0b0 translate 512 0 0 "QQuickItem(QQuickImage:)"  ) 0
           [ ] RootNode 0x7fe4881b0b90  "" ) 1
            [ ] GeometryNode( 0x7fe4881b1de0 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 256 y2= 256 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
[ ] TransformNode( 0x7fe4881adaf0 det= 0.444444 "QQuickItem(DelegateItem_QMLTYPE_0:)"  ) 0
acity= 0.7  combined= 1  "" ) 1
 identity  QQuickItem(QQuickMouseArea:)"  ) 0
 translat  512 0 0 "QQuickItem(QQuickImage:)"  ) 0
) 1
0 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 256 y2= 256 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
entity "QQuickItem(QQuickItem:)"  ) 0
t= 0.0948417 "QQuickItem(DelegateItem_QMLTYPE_0:)"  ) 0
acity= 0.825694 combined= 1  "" ) 1
 identity "QQuickItem(QQuickMouseArea:)"  ) 0
 translate 512 0 0 "QQuickItem(QQuickImage:)"  ) 0
) 1
0 strip #V: 4 #I: 0 x1= 0 y1= 0 x2= 256 y2= 256 materialtype= 0x7fe4c166ad32 ) "internalimage" 1000001 order 0
nslate 390 267 0 "QQuickItem(QQuickText_QML_10:)"  ) 0
entity "text"  ) 1
iangles #V: 140 #I: 210 x1= 20.0244 y1= 8.79277 x2= 479.55 y2= 107.761 materialtype= 0x7fe4c166b051 ) "glyphs" 1000003 order 0
nslate 480 528 0 "QQuickItem(QQuickImage_QML_11:)"  ) 0
```

```
Item {
    id: mainViewArea
    anchors.fill: parent

    Background {
        id: background
    }

    ListView {
        id: listView

        property real globalLightPosX: lightImage.x / root.width
        property real globalLightPosY: lightImage.y / root.height

        // Normal-mapped cover shared among delegates
        ShaderEffectSource {
            id: coverNmapSource
            sourceItem: Image { source: "images/cover_nmap.png" }
            hideSource: true
            visible: false
        }

        anchors.fill: parent
        spacing: -60
        model: moviesModel
        delegate: DelegateItem {
            name: model.name
```



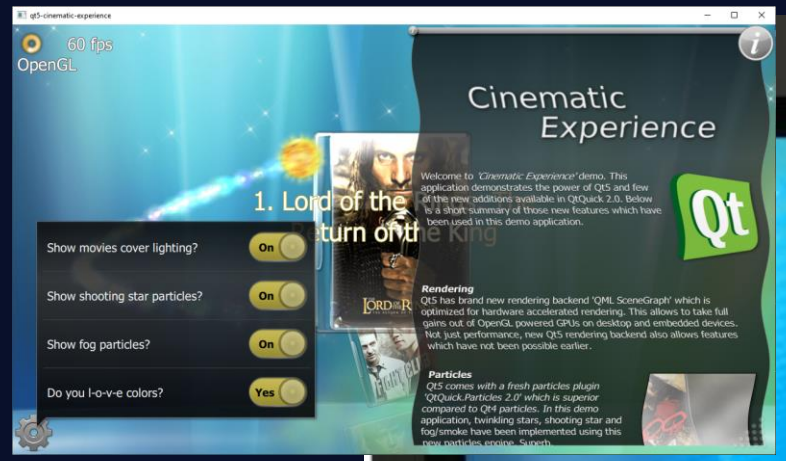


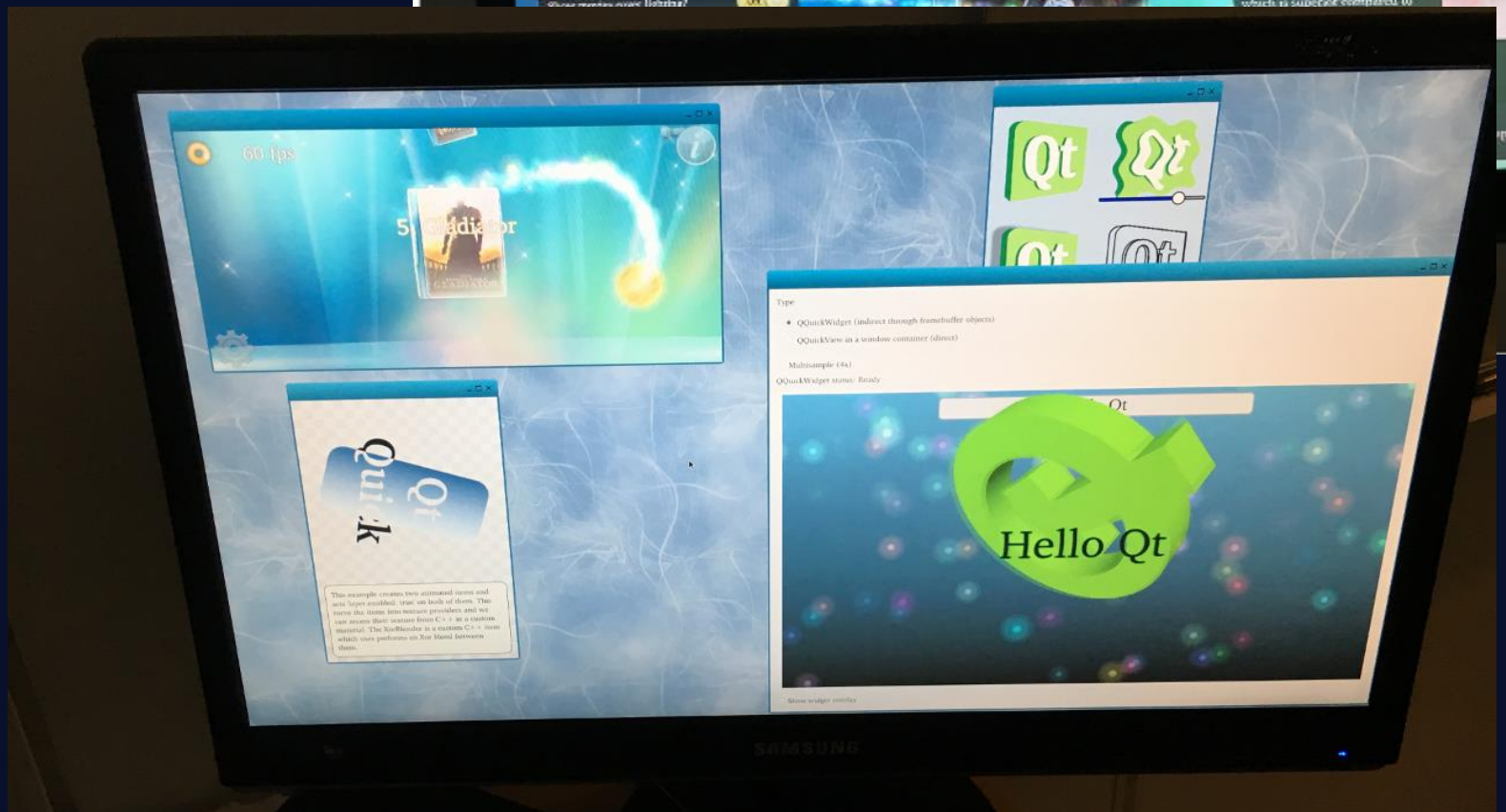
No blind draw* stream on every frame. Can track scene changes, reuse graphics resources, merge geometry (batching), generate texture atlases.

Some things our team works on

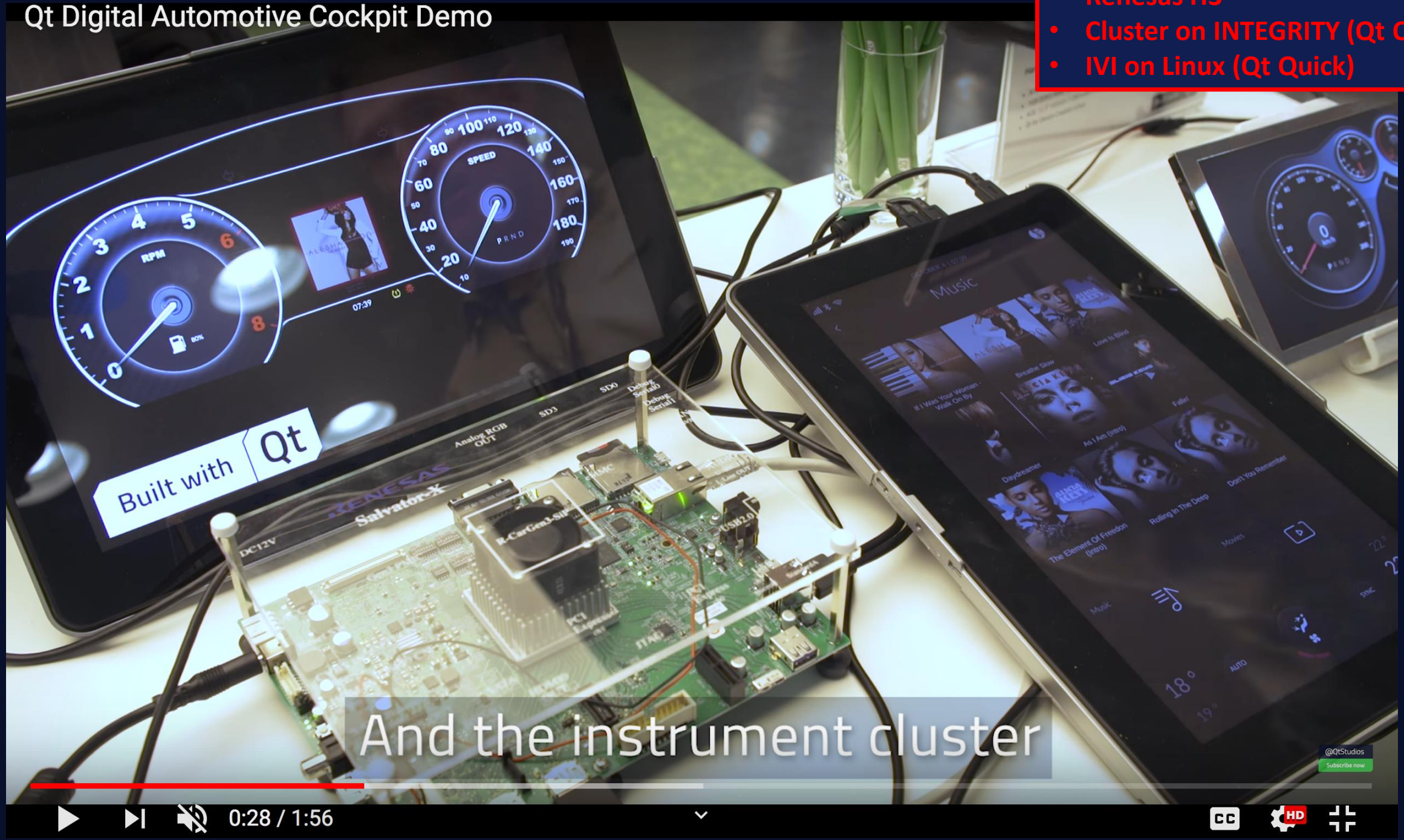
- Windowing system and graphics plumbing.
  - Enable Qt to render everywhere: devices, mobile, desktop, WebAssembly
    - Win32, Xlib/xcb, Wayland, Cocoa, …
    - EGL, GLX, WGL, Vulkan WSI, …
    - fbdev, DRM/KMS (+ gbm, EGLStream), …
    - INTEGRITY, QNX, exotic stuff

Qt Quick
|
OpenGL
|
EGL/GLX/WGL/NS*/…
|
Winsys/DRM+…/EGL fbdev glue/vendor APIs

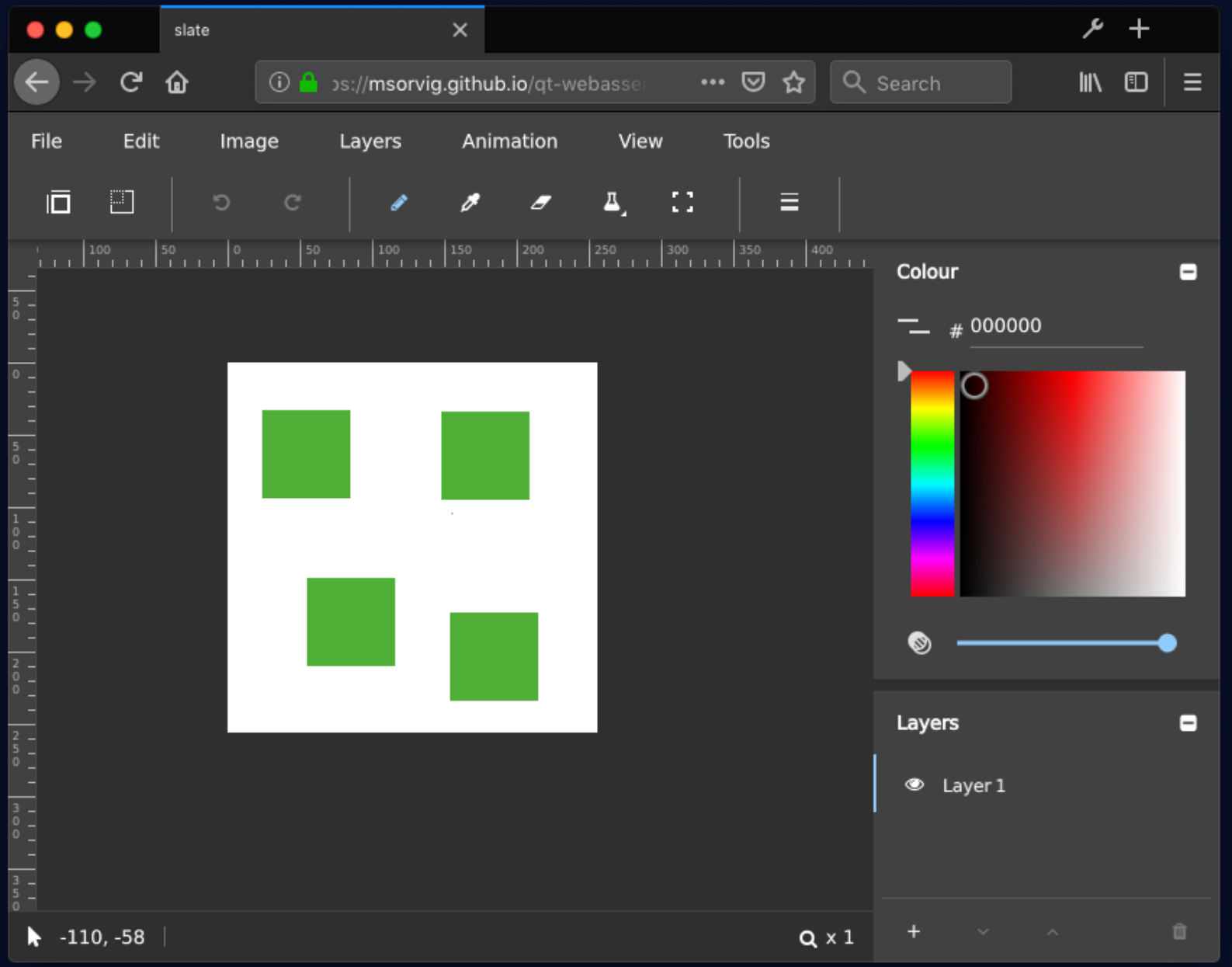




1. A Qt Quick app on Windows
2. Embedded: Linux without a windowing system (here on DRM+EGLStream)
3. Embedded: Qt-based Wayland compositor with Qt apps as Wayland clients



- Renesas H3
- Cluster on INTEGRITY (Qt Quick)
- IVI on Linux (Qt Quick)

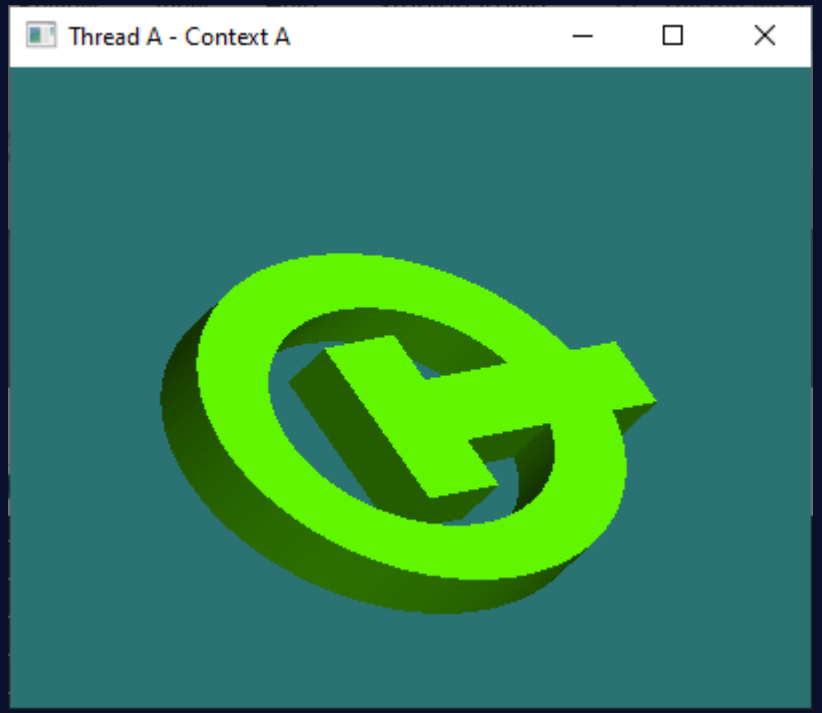


A Qt Quick-based desktop-style application in the browser via WebAssembly

Some things our team works on (2)

- OpenGL enablers.
  - Context management, function resolvers
  - Fight platform & device quirks
    - Spec sheet does not matter. Reality (what's in the BSP) matters.
- Used by applications either directly or indirectly.
  - Because Qt Quick and friends all build on these

```
class MyGLWindow : public QWindow, protected QOpenGLFunctions
{
    Q_OBJECT
public:
    MyGLWindow(QScreen *screen = 0);

protected:
    void initializeGL();
    void paintGL();

    QOpenGLContext *m_context;
};

MyGLWindow(QScreen *screen)
  : QWindow(screen)
{
    setSurfaceType(OpenGLSurface);
    create();

    // Create an OpenGL context
    m_context = new QOpenGLContext;
    m_context->create();

    // Setup scene and render it
    initializeGL();
    paintGL();
}

void MyGLWindow::initializeGL()
{
    m_context->makeCurrent(this);
    initializeOpenGLFunctions();
}
```

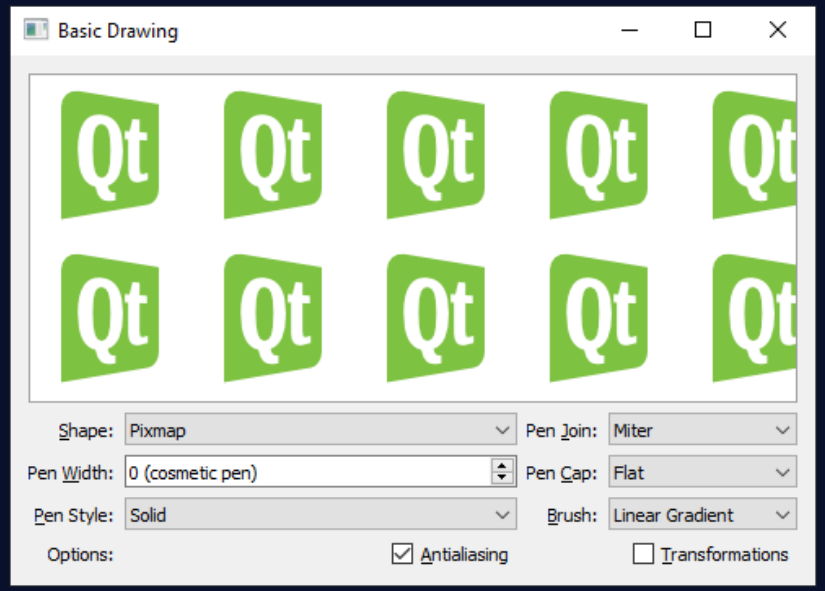


hellowindow example: Just a QWindow, QOpenGLContext, and a few helpers. The actual content is the application's own custom OpenGL rendering.

Some things our team works on (3)

- QPainter and backends
  - Raster paint engine
    - Pure software. Used by QWidget.
  - OpenGL and other paint engines

```
void SimpleExampleWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(Qt::blue);
    painter.setFont(QFont("Arial", 30));
    painter.drawText(rect(), Qt::AlignCenter, "Qt");
}
```

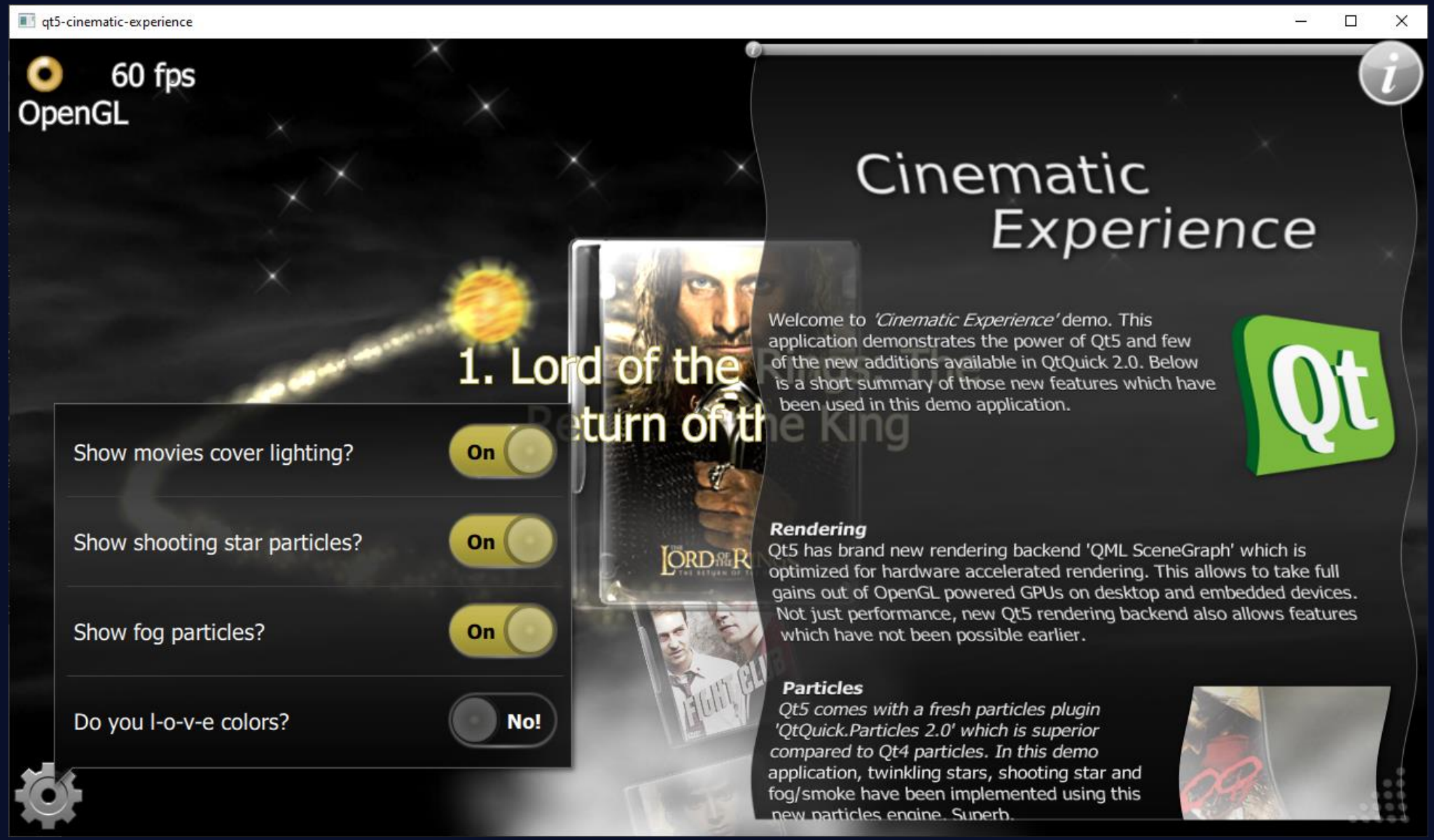


Some things our team works on (4)

- Qt Quick
  - Provides the “standard” types to create UIs with QML.
    - QML is a declarative language, mainly used to describe UIs.
      - Could describe any object tree.
      - Can include Javascript as well.
      - Has nothing to do with graphics on its own. Hence Qt Quick.
  - Item, Rectangle, Image, animations, positioning, input, …

Property bindings. Property animations. Component model. (put this to Stuff.qml then do Stuff { } to instantiate)

```
import QtQuick 2.3

Item {

    width: 320
    height: 480

    Rectangle {
        color: "#272822"
        width: 320
        height: 480
    }

    // This element displays a rectangle with a gradient and a border
    Rectangle {
        x: 160
        y: 20
        width: 100
        height: 100
        radius: 8 // This gives rounded corners to the Rectangle
        gradient: Gradient { // This sets a vertical gradient fill
            GradientStop { position: 0.0; color: "aqua" }
            GradientStop { position: 1.0; color: "teal" }
        }
        border { width: 3; color: "white" } // This sets a 3px wide black border to be drawn
    }

    // This rectangle is a plain color with no border
    Rectangle {
        x: 40
        y: 20
        width: 100
        height: 100
        color: "red"
    }
}
```


qt5-cinematic-experience
60 fps
OpenGL
Cinematic Experience
Welcome to 'Cinematic Experience' demo. This application demonstrates the power of Qt5 and few of the new additions available in QtQuick 2.0. Below is a short summary of those new features which have been used in this demo application.
1. Lord of the
Show movies cover lighting?
On
Show shooting star particles?
On
Show fog particles?
On
Do you l-o-v-e colors?
No!
Rendering
Qt5 has brand new rendering backend 'QML SceneGraph' which is optimized for hardware accelerated rendering. This allows to take full gains out of OpenGL powered GPUs on desktop and embedded devices. Not just performance, new Qt5 rendering backend also allows features which have not been possible earlier.
Particles
Qt5 comes with a fresh particles plugin 'QtQuick.Particles 2.0' which is superior compared to Qt4 particles. In this demo application, twinkling stars, shooting star and fog/smoke have been implemented using this new particles engine. Superb.

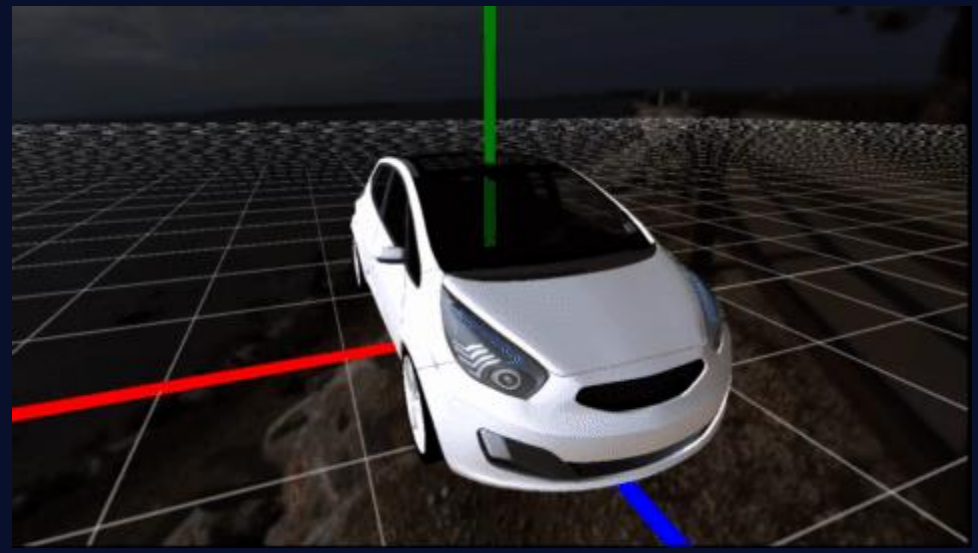
Some things our team works on (5)

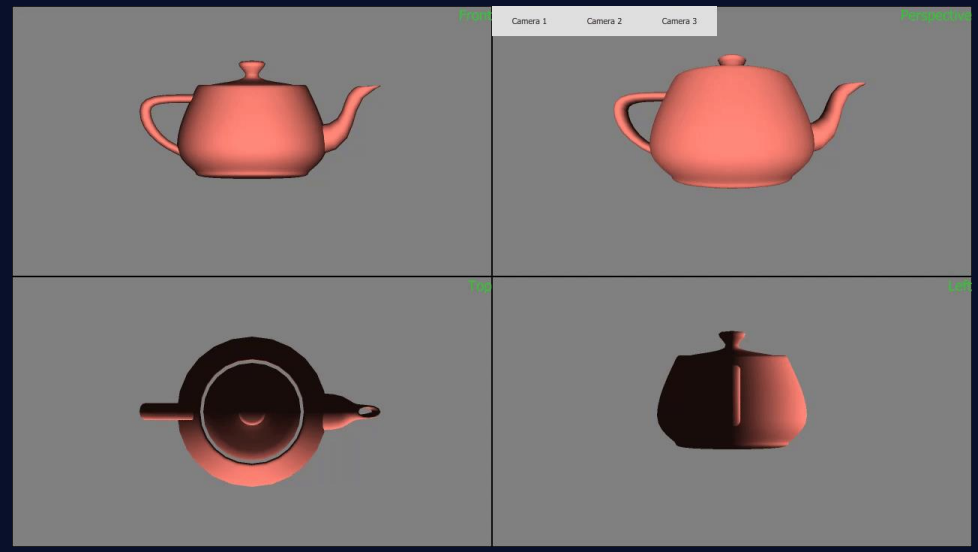
- Qt Quick Scene Graph
  - The rendering engine underneath Qt Quick.
  - Originally targeting OpenGL (ES 2.0) exclusively.
    - Alternative renderers with limited functionality (software, OpenVG).
  - Better suited for accelerated 3D APIs than QPainter.
    - Stateful scene. Batching. Texture atlases.

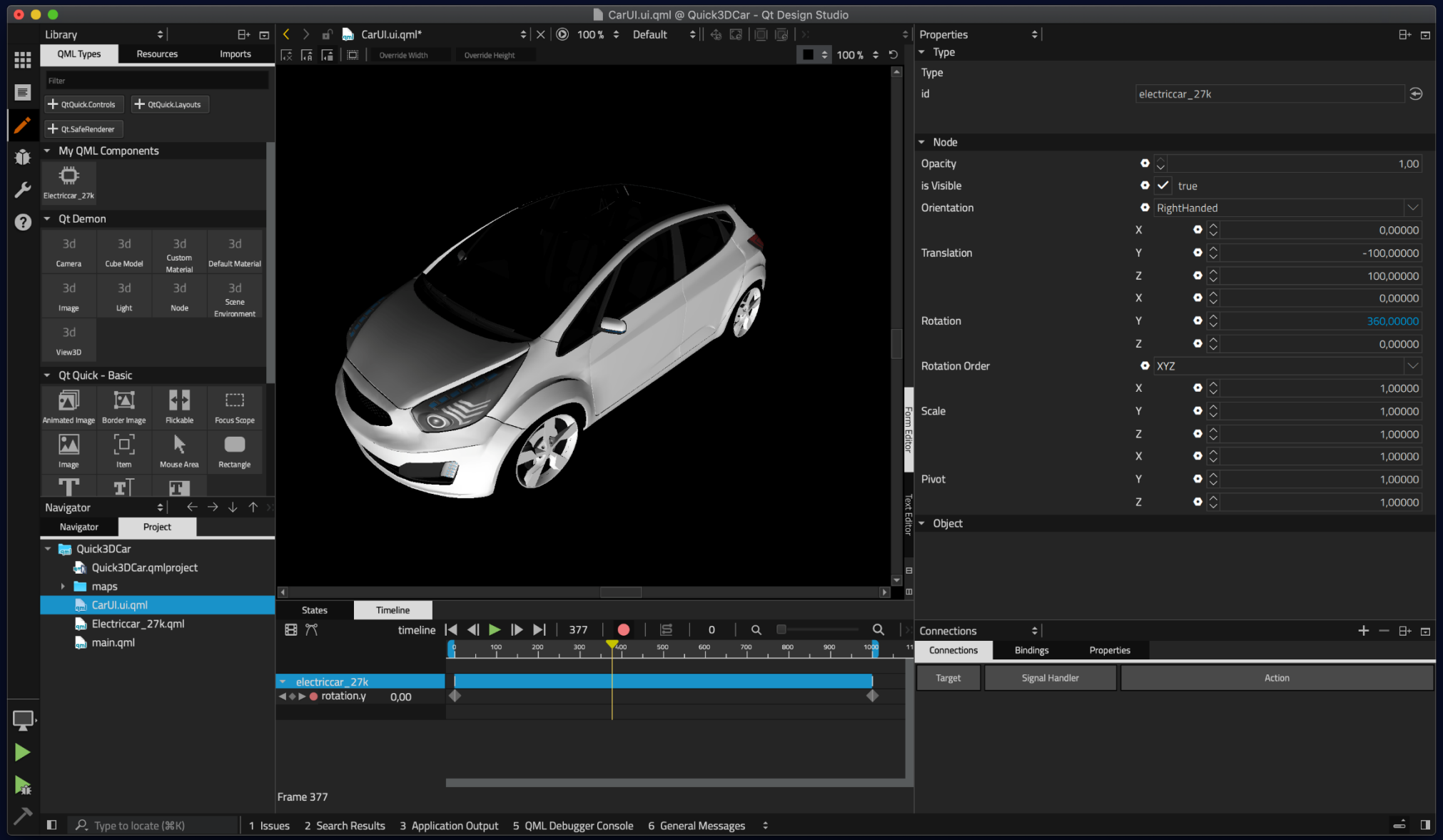
Some things our team works on (6)

- 3D
  - About to introduce Qt Quick 3D.
  - Together with designer-friendly tooling in Qt Design Studio.

```
import QtQuick3D 1.0

Window {
  id: window
  visible: true
  width: 1280
  height: 720

  View3D { // Viewport for 3D content
    id: view
    anchors.fill: parent

    Node {
      id: scene

      Light {
        id: directionalLight
      }

      Camera {
        z: -600
      }

      Model {
        id: cubeModel
        source: "#Cube"
        materials: [
          DefaultMaterial {
            diffuseColor: "red"
          }
        ]
      }
```






CarUI.ui.qml @ Quick3DCar - Qt Design Studio
Library
QML Types
Resources
Imports
Filter
QtQuick.Controls
QtQuick.Layouts
Qt.SafeRenderer
My QML Components
Electriccar_27k
Qt Demon
Camera
Cube Model
Custom Material
Default Material
Image
Light
Node
Scene Environment
View3D
Qt Quick - Basic
Animated Image
Border Image
Flickable
Focus Scope
Image
Item
Mouse Area
Rectangle
Navigator
Navigator
Project
Quick3DCar
Quick3DCar.qmlproject
maps
CarUI.ui.qml
Electriccar_27k.qml
main.qml
CarUI.ui.qml*
100 %
Default
Override Width
Override Height
100 %
Form Editor
Text Editor
States
Timeline
timeline
377
0
electriccar_27k
rotation.y
0,00
Frame 377
Properties
Type
Type
id
electriccar_27k
Node
Opacity
1,00
is Visible
true
Orientation
RightHanded
Translation
X
0,00000
Y
-100,00000
Z
100,00000
Rotation
X
0,00000
Y
360,00000
Z
0,00000
Rotation Order
XYZ
Scale
X
1,00000
Y
1,00000
Z
1,00000
Pivot
X
1,00000
Y
1,00000
Z
1,00000
Object
Connections
Connections
Bindings
Properties
Target
Signal Handler
Action
Type to locate (⌘K)
1 Issues
2 Search Results
3 Application Output
5 QML Debugger Console
6 General Messages

Some things our team does not work on (but is cool stuff still)

- Qt WebEngine
  - Chromium. QWidget & Qt Quick integrations.
- Solutions for the ultra low-end (MCU)

Intermission: Diagram!

QtQuick
+ scenegraph

QtQml

QML (Qt Quick) application

QtWidgets
(QApplication, QWidget, QPushButton, …)

Widget application

QtGui
(QGuiApplication, QWindow, QOpenGLContext, QPainter, …)

Plain QWindow-based application (less common)

QPA
(QPlatformWindow, QPlatformOpenGLContext, QPlatformVulkanInstance, …)

Platform plugin
(implementing QPlatform*)

Plugins for the platform plugin
(if the platform has multiple windowing systems and related APIs)

QtCore
(QCoreApplication, QObject, QString, …)

The OS and platform:
Win32/xcb/Wayland/Cocoa/UIKit/Android, …
EGL/GLX/WGL, OpenGL
DRM+GBM/DRM+EGLStream, Screen, OpenWF,
proprietary stuff, …

Towards Qt 6

How does this change?

Some things our team works on (7)

More stuff? It never ends…

- Keep the stack relevant in the future as well
  - Move away from direct OpenGL usage. Abstract what we can.
    - Graphics API calls. Shading languages.
  - **Use the platform's best supported graphics API at run time.**
    - Be it Vulkan, Metal, Direct3D, or OpenGL.
  - But try not to forget the more exotic use cases.
    - software only? proprietary 2D APIs for this and that SoC? etc.

Detailed Description

Accelerated 2D/3D graphics API abstraction.

The Qt Rendering Hardware Interface is an abstraction for hardware accelerated graphics APIs, such as, **OpenGL**, **OpenGL ES**, **Direct3D**, **Metal**, and **Vulkan**.

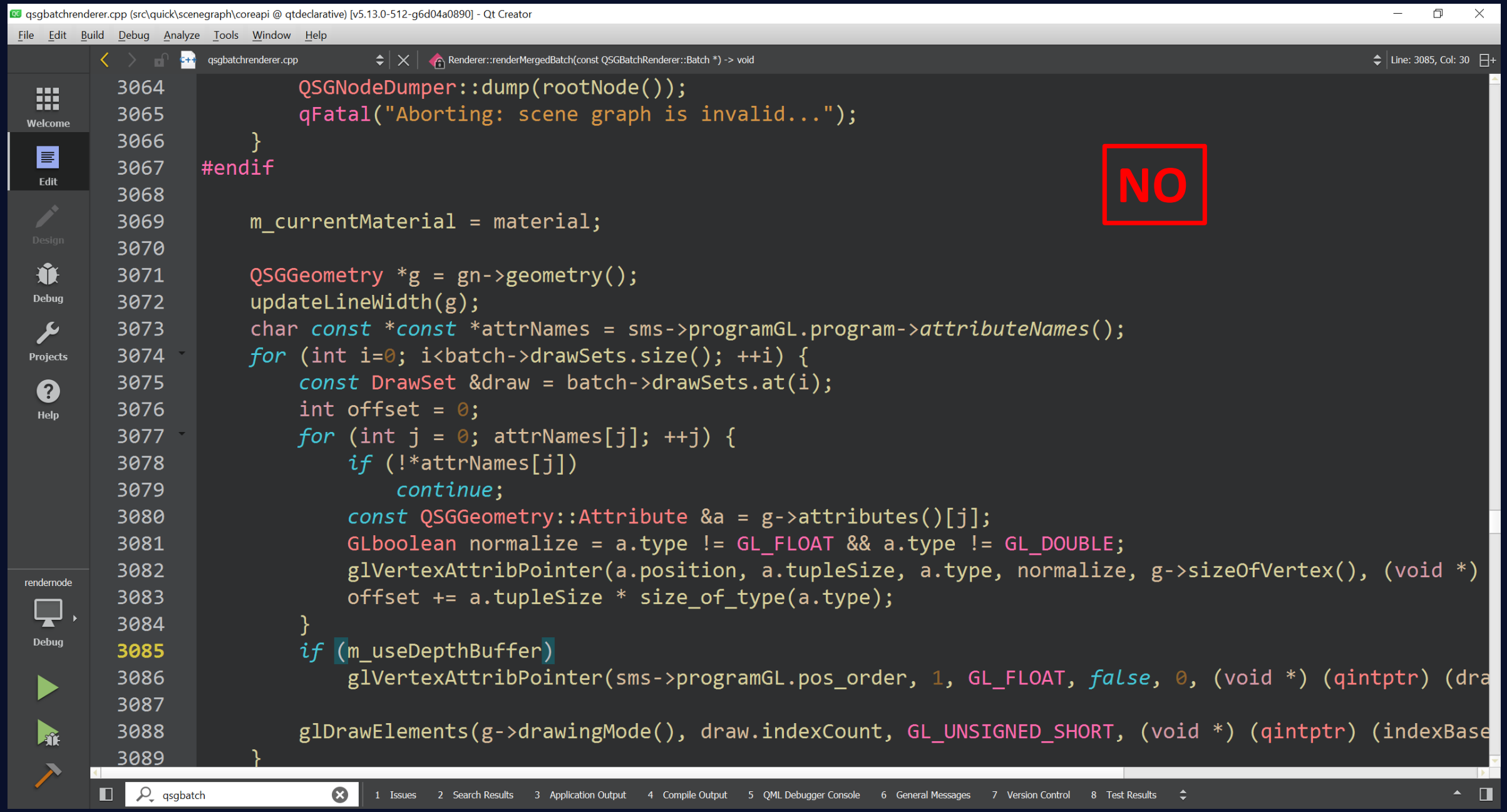
Some of the main design goals are:

- Simple, minimal, understandable, extensible. Follow the proven path of the Qt Quick scenegraph.
- Aim to be a product - and in the bigger picture, part of a product (Qt) - that is usable out of the box both by internal (such as, Qt Quick) and, eventually, external users.
- Not a complete 1:1 wrapper for any of the underlying APIs. The feature set is tuned towards the needs of Qt's 2D and 3D offering (QPainter, Qt Quick, Qt 3D Studio). Iterate and evolve in a sustainable manner.
- Intrinsically cross-platform, without reinventing: abstracting cross-platform aspects of certain APIs (such as, OpenGL context creation and windowing system interfaces, Vulkan instance and surface management) is not in scope here. These are delegated to the existing QtGui facilities (QWindow, QOpenGLContext, QVulkanInstance) and its backing QPA architecture.

Each QRhi instance is backed by a backend for a specific graphics API. The selection of the backend is a run time choice and is up to the application or library that creates the QRhi instance. Some backends are available on multiple platforms (OpenGL, Vulkan, Null), while APIs specific to a given platform are only available when running on the platform in question (Metal on macOS/iOS/tvOS, Direct3D on Windows).

The available backends currently are:

- OpenGL 2.1 or OpenGL ES 2.0 or newer. Some extensions are utilized when present, for example to enable multisample framebuffers.
- Direct3D 11.1
- Metal
- Vulkan 1.0, optionally with some extensions that are part of Vulkan 1.1
- Null - A "dummy" backend that issues no graphics calls at all.


qsgbatchrenderer.cpp (src\quick\scenegraph\coreapi @ qtdeclarative) [v5.13.0-512-g6d04a0890] - Qt Creator
File Edit Build Debug Analyze Tools Window Help
qsgbatchrenderer.cpp
Renderer::renderMergedBatch(const QSGBatchRenderer::Batch *) -> void
Line: 3085, Col: 30
Welcome
Edit
Design
Debug
Projects
Help
rendernode
Debug
3064 QSGNodeDumper::dump(rootNode());
3065 qFatal("Aborting: scene graph is invalid...");
3066 }
3067 #endif
3068
3069 m_currentMaterial = material;
3070
3071 QSGGeometry *g = gn->geometry();
3072 updateLineWidth(g);
3073 char const *const *attrNames = sms->programGL.program->attributeNames();
3074 for (int i=0; i<batch->drawSets.size(); ++i) {
3075 const DrawSet &draw = batch->drawSets.at(i);
3076 int offset = 0;
3077 for (int j = 0; attrNames[j]; ++j) {
3078 if (!*attrNames[j])
3079 continue;
3080 const QSGGeometry::Attribute &a = g->attributes()[j];
3081 GLboolean normalize = a.type != GL_FLOAT && a.type != GL_DOUBLE;
3082 glVertexAttribPointer(a.position, a.tupleSize, a.type, normalize, g->sizeOfVertex(), (void *)
3083 offset += a.tupleSize * size_of_type(a.type);
3084 }
3085 if (m_useDepthBuffer)
3086 glVertexAttribPointer(sms->programGL.pos_order, 1, GL_FLOAT, false, 0, (void *) (qintptr) (dra
3087
3088 glDrawElements(g->drawingMode(), draw.indexCount, GL_UNSIGNED_SHORT, (void *) (qintptr) (indexBase
3089 }
NO
qsgbatch
1 Issues 2 Search Results 3 Application Output 4 Compile Output 5 QML Debugger Console 6 General Messages 7 Version Control 8 Test Results


qsgbatchrenderer.cpp (src\quick\scenegraph\coreapi @ qtdeclarative) [v5.13.0-512-g6d04a0890] - Qt Creator
File Edit Build Debug Analyze Tools Window Help
qsgbatchrenderer.cpp
Renderer::renderMergedBatch(QSGBatchRenderer::Renderer::PreparedRenderBatch *) -> void
Line: 3703, Col: 1
Welcome
Edit
Design
Debug
Projects
Help
rendernode
Debug
3693 void Renderer::renderMergedBatch(PreparedRenderBatch *renderBatch) // split prepare-render (RHI only)
3694 {
3695 const Batch *batch = renderBatch->batch;
3696 Element *e = batch->first;
3697 QSGGeometryNode *gn = e->node;
3698 QSGGeometry *g = gn->geometry();
3699 checkLineWidth(g);
3700
3701 if (batch->clipState.type & ClipState::StencilClip)
3702 enqueueStencilDraw(batch);
3703
3704 QRhiCommandBuffer *cb = commandBuffer();
3705 setGraphicsPipeline(cb, batch, e);
3706
3707 for (int i = 0, ie = batch->drawSets.size(); i != ie; ++i) {
3708 const DrawSet &draw = batch->drawSets.at(i);
3709 const QRhiCommandBuffer::VertexInput vbufBindings[] = {
3710 { batch->vbo.buf, quint32(draw.vertices) },
3711 { batch->vbo.buf, quint32(draw.zorders) }
3712 };
3713 cb->setVertexInput(VERTEX_BUFFER_BINDING, 2, vbufBindings,
3714 batch->ibo.buf, draw.indices,
3715 m_uint32IndexForRhi ? QRhiCommandBuffer::IndexUInt32 : QRhiCommandBuffer::Index
3716 cb->drawIndexed(draw.indexCount);
3717 }
3718 }
YES
qsgbatch
1 Issues 2 Search Results 3 Application Output 4 Compile Output 5 QML Debugger Console 6 General Messages 7 Version Control 8 Test Results

Porting (Qt Quick) to QRhi

- No intermixed
  copy (glBufferData, glTexImage, etc.) operations,
  state changes,
  and draw calls.
- Must collect resource updates before starting to record a (render) pass on the command buffer.
  - because this then maps naturally to Vulkan and Metal under the hood

Porting (Qt Quick) to QRhi (2)

- Must create up front and then reuse
  - pipeline state objects,
  - objects describing the shader resource bindings
    (which uniform buffers, textures, etc. are visible to the shader stages).
- Working with uniform buffers by default, no individual uniforms.

Porting (Qt Quick) to QRhi (3)

- Typically a **render** step becomes **prepare** + **render**.
  - **Prepare**: Gather all data (geometry, pipeline states, etc.) needed for the current frame, enqueue buffer (vertex, index, uniform) and texture resource updates.
  - **Render**: Record starting the pass, record binding a pipeline, record draw call, change bindings if needed, record draw call, …, record end pass.
  - Submit and present.

```
// depth test stays enabled but no need to write out depth from the
// transparent (back-to-front) pass
m_gstate.depthWrite = false;

QVarLengthArray<PreparedRenderBatch, 64> alphaRenderBatches;
if (Q_LIKELY(renderAlpha)) {
    for (int i = 0, ie = m_alphaBatches.size(); i != ie; ++i) {
        Batch *b = m_alphaBatches.at(i);
        PreparedRenderBatch renderBatch;
        bool ok;
        if (b->merged)
            ok = prepareRenderMergedBatch(b, &renderBatch);
        else if (b->isRenderNode)
            ok = prepareRhiRenderNode(b, &renderBatch);
        else
            ok = prepareRenderUnmergedBatch(b, &renderBatch);
        if (ok)
            alphaRenderBatches.append(renderBatch);
    }
}

if (m_visualizer->mode() != Visualizer::VisualizeNothing)
    m_visualizer->prepareVisualize();

QRhiCommandBuffer *cb = commandBuffer();
cb->beginPass(renderTarget(), m_pstate.clearColor, m_pstate.dsClear, m_resourceUpdates);
m_resourceUpdates = nullptr;

for (int i = 0, ie = opaqueRenderBatches.count(); i != ie; ++i) {
    PreparedRenderBatch *renderBatch = &opaqueRenderBatches[i];
    if (renderBatch->batch->merged)
        renderMergedBatch(renderBatch);
    else
        renderUnmergedBatch(renderBatch);
}
```

Prepare uniform buffers and pipeline states

Record starting the render pass (with clearing color/depth/stencil)

Record draw calls

Qt

```
3236  bool Renderer::ensurePipelineState(Element *e, const ShaderManager::Shader *sms) // RHI
3237  {
3238      // In unmerged batches the srbs in the elements are all compatible layout-wise.
3239      const GraphicsPipelineStateKey k { m_gstate, sms, renderPassDescriptor(), e->srb };
3240
3241      // See if there is an existing, matching pipeline state object.
3242      auto it = m_pipelines.constFind(k);
3243      if (it != m_pipelines.constEnd()) {
3244          e->ps = *it;
3245          return true;
3246      }
3247
3248      // Build a new one. This is potentially expensive.
3249      QRhiGraphicsPipeline *ps = m_rhi->newGraphicsPipeline();
3250      ps->setShaderStages(sms->programRhi.shaderStages);
3251      ps->setVertexInputLayout(sms->programRhi.inputLayout);
3252      ps->setShaderResourceBindings(e->srb);
3253      ps->setRenderPassDescriptor(renderPassDescriptor());
3254
3255      QRhiGraphicsPipeline::Flags flags = 0;
3256      if (needsBlendConstant(m_gstate.srcColor) || needsBlendConstant(m_gstate.dstColor))
3257          flags |= QRhiGraphicsPipeline::UsesBlendConstants;
3258      if (m_gstate.usesScissor)
3259          flags |= QRhiGraphicsPipeline::UsesScissor;
3260      if (m_gstate.stencilTest)
3261          flags |= QRhiGraphicsPipeline::UsesStencilRef;
3262
3263      ps->setFlags(flags);
3264      ps->setTopology(qsg_topology(m_gstate.drawMode));
3265      ps->setCullMode(m_gstate.cullMode);
3266
3267      QRhiGraphicsPipeline::TargetBlend blend;
3268      blend.colorWrite = m_gstate.colorWrite;
3269      blend.enable = m_gstate.blending;
3270      blend.srcColor = m_gstate.srcColor;
```

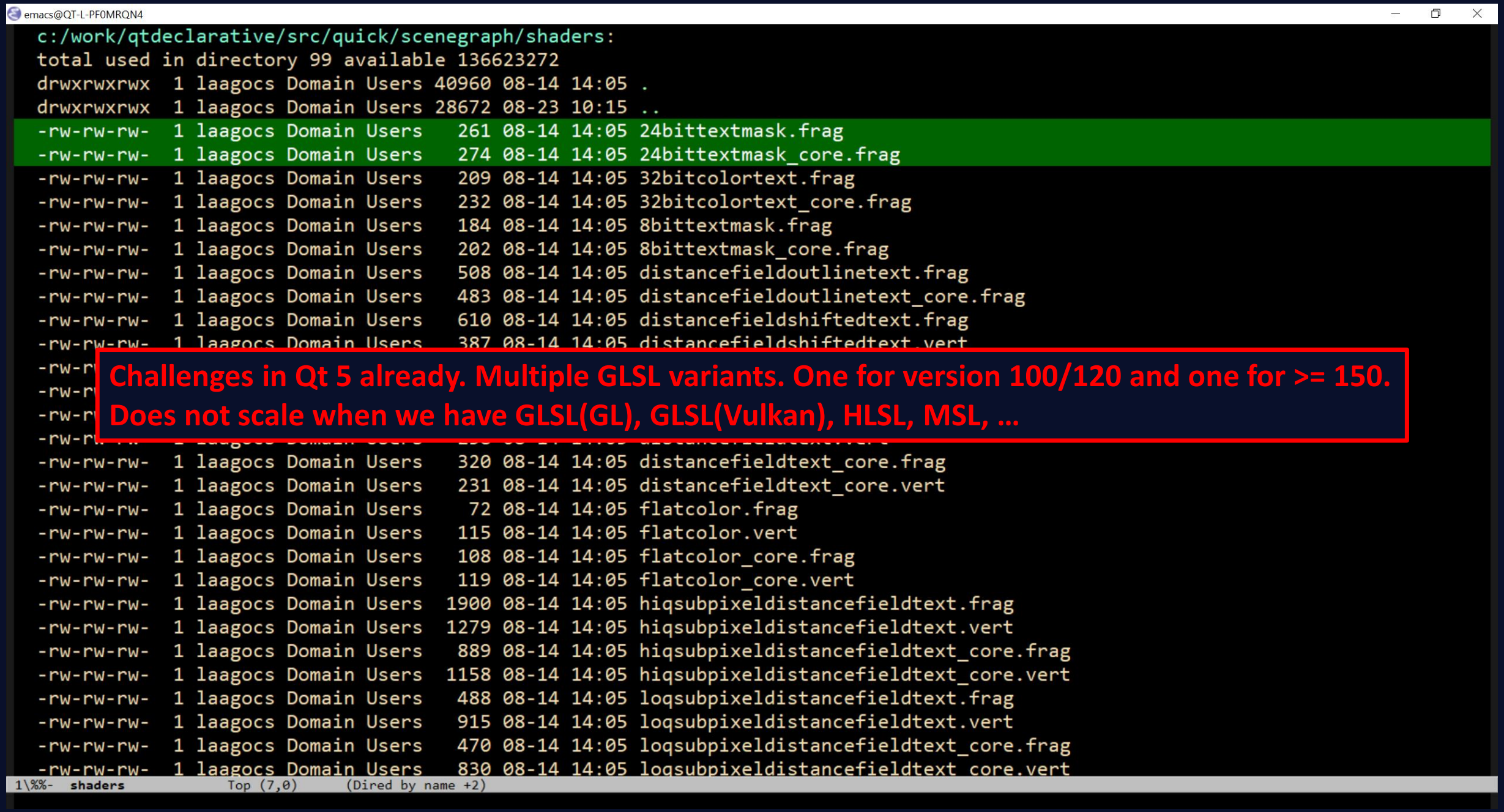
Pipeline state: reuse or create a new one. Have to collect and manage the state which was a blind glWhatever() before.

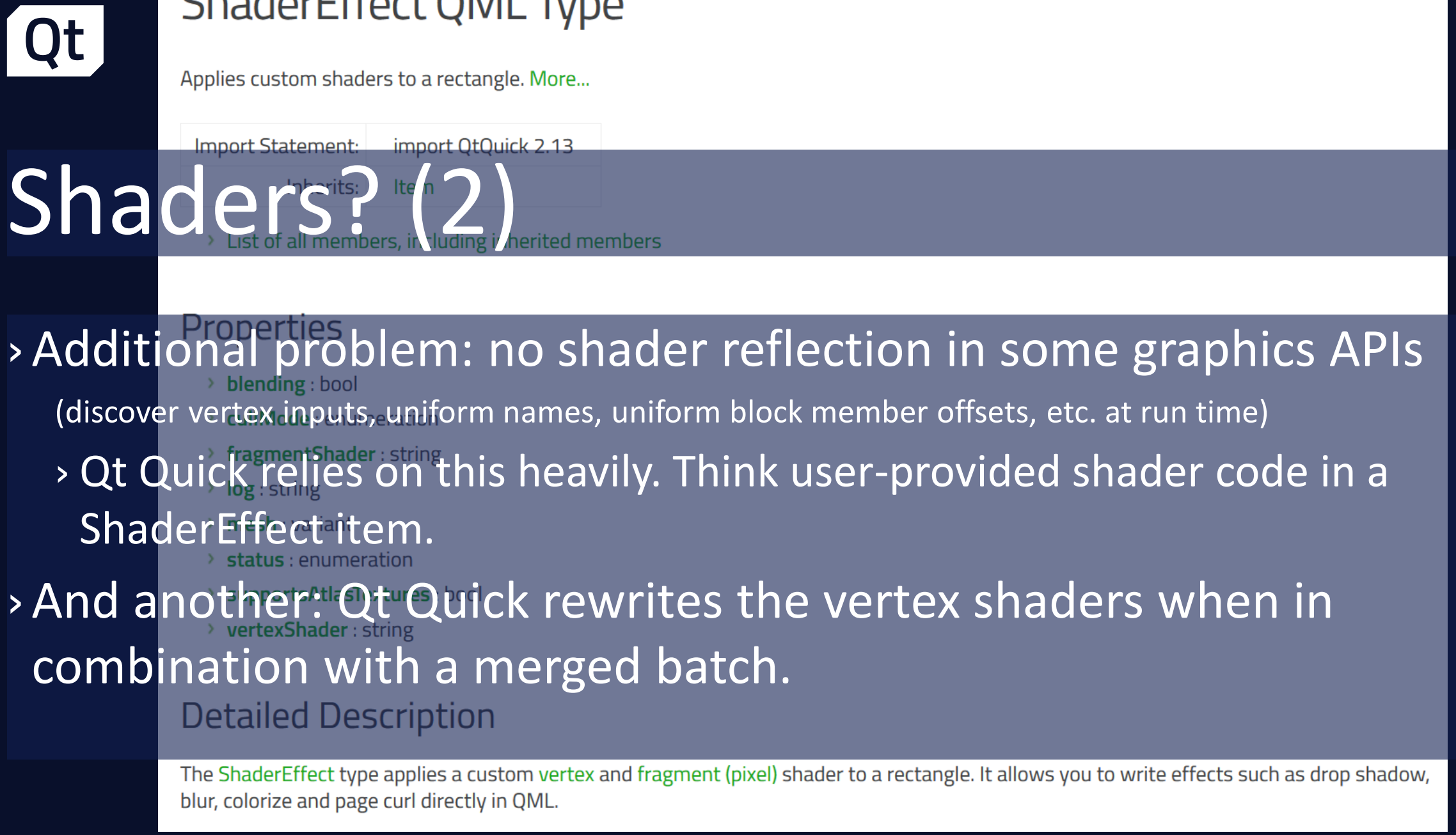
The vertex and fragment shader code is in there somewhere

Shaders?

```
#version 150 core

in vec4 vCoord;
in vec2 tCoord;

out vec2 sampleCoord;

uniform mat4 matrix;
uniform vec2 textureScale;
uniform float dpr;

void main()
{
    sampleCoord = tCoord * textureScale;
    gl_Position = matrix * round(vCoord * dpr) / dpr;
}
```

```
#version 150 core

in vec2 sampleCoord;

out vec4 fragColor;

uniform sampler2D _qt_texture;
uniform vec4 color;

void main()
{
    vec4 glyph = texture(_qt_texture, sampleCoord);
    fragColor = vec4(glyph.rgb * color.a, glyph.a);
}
```


emacs@QT-L-PF0MRQN4
c:/work/qtdeclarative/src/quick/scenegraph/shaders:
total used in directory 99 available 136623272
drwxrwxrwx 1 laagocs Domain Users 40960 08-14 14:05 .
drwxrwxrwx 1 laagocs Domain Users 28672 08-23 10:15 ..
-rw-rw-rw- 1 laagocs Domain Users 261 08-14 14:05 24bittextmask.frag
-rw-rw-rw- 1 laagocs Domain Users 274 08-14 14:05 24bittextmask_core.frag
-rw-rw-rw- 1 laagocs Domain Users 209 08-14 14:05 32bitcolortext.frag
-rw-rw-rw- 1 laagocs Domain Users 232 08-14 14:05 32bitcolortext_core.frag
-rw-rw-rw- 1 laagocs Domain Users 184 08-14 14:05 8bittextmask.frag
-rw-rw-rw- 1 laagocs Domain Users 202 08-14 14:05 8bittextmask_core.frag
-rw-rw-rw- 1 laagocs Domain Users 508 08-14 14:05 distancefieldoutlinetext.frag
-rw-rw-rw- 1 laagocs Domain Users 483 08-14 14:05 distancefieldoutlinetext_core.frag
-rw-rw-rw- 1 laagocs Domain Users 610 08-14 14:05 distancefieldshiftedtext.frag
Challenges in Qt 5 already. Multiple GLSL variants. One for version 100/120 and one for >= 150.
Does not scale when we have GLSL(GL), GLSL(Vulkan), HLSL, MSL, …
-rw-rw-rw- 1 laagocs Domain Users 320 08-14 14:05 distancefieldtext_core.frag
-rw-rw-rw- 1 laagocs Domain Users 231 08-14 14:05 distancefieldtext_core.vert
-rw-rw-rw- 1 laagocs Domain Users 72 08-14 14:05 flatcolor.frag
-rw-rw-rw- 1 laagocs Domain Users 115 08-14 14:05 flatcolor.vert
-rw-rw-rw- 1 laagocs Domain Users 108 08-14 14:05 flatcolor_core.frag
-rw-rw-rw- 1 laagocs Domain Users 119 08-14 14:05 flatcolor_core.vert
-rw-rw-rw- 1 laagocs Domain Users 1900 08-14 14:05 hiqsubpixeldistancefieldtext.frag
-rw-rw-rw- 1 laagocs Domain Users 1279 08-14 14:05 hiqsubpixeldistancefieldtext.vert
-rw-rw-rw- 1 laagocs Domain Users 889 08-14 14:05 hiqsubpixeldistancefieldtext_core.frag
-rw-rw-rw- 1 laagocs Domain Users 1158 08-14 14:05 hiqsubpixeldistancefieldtext_core.vert
-rw-rw-rw- 1 laagocs Domain Users 488 08-14 14:05 loqsubpixeldistancefieldtext.frag
-rw-rw-rw- 1 laagocs Domain Users 915 08-14 14:05 loqsubpixeldistancefieldtext.vert
-rw-rw-rw- 1 laagocs Domain Users 470 08-14 14:05 loqsubpixeldistancefieldtext_core.frag
-rw-rw-rw- 1 laagocs Domain Users 830 08-14 14:05 loqsubpixeldistancefieldtext_core.vert
1\%%- shaders Top (7,0) (Dired by name +2)



Shaders? (2)

- Additional problem: no shader reflection in some graphics APIs

  (discover vertex inputs, uniform names, uniform block member offsets, etc. at run time)

  - Qt Quick relies on this heavily. Think user-provided shader code in a ShaderEffect item.

- And another: Qt Quick rewrites the vertex shaders when in combination with a merged batch.

New pipeline

Vulkan-flavor GLSL
-> [generate batching-friendly variant for vertex shaders]
-> **glslang** -> SPIR-V bytecode
-> **SPIRV-Cross** -> reflection metadata + GLSL/HLSL/MSL source
-> pack it all together -> typically ends up in a .qsb file.

Deserialize into a QShader at run time. (just a simple container)

Qt Quick specifies the variant (normal / batchable). The QRhi backends decide the rest (which language, which version to pick).

```
#version 440

layout(location = 0) in vec2 sampleCoord;
layout(location = 0) out vec4 fragColor;

layout(binding = 1) uniform sampler2D _qt_texture;

layout(std140, binding = 0) uniform buf {
    mat4 matrix;
    vec4 color;
    vec2 textureScale;
    float dpr;
} ubuf;

void main()
{
    vec4 glyph = texture(_qt_texture, sampleCoord);
    fragColor = vec4(glyph.rgb * ubuf.color.a, glyph.a);
}
```

qsb --glsl "150,120,100 es" --hlsl 50 --msl 12
-o textmask.frag.qsb textmask.frag

```
Stage: Fragment

Has 6 shaders: (unordered list)
  Shader 0: HLSL 50 [Standard]
  Shader 1: GLSL 150 [Standard]
  Shader 2: GLSL 100 es [Standard]
  Shader 3: GLSL 120 [Standard]
  Shader 4: SPIR-V 100 [Standard]
  Shader 5: MSL 12 [Standard]

Reflection info: {
    "combinedImageSamplers": [
        {
            "binding": 1,
            "name": "_qt_texture",
            "set": 0,
            "type": "sampler2D"
        }
    ],
    "inputs": [
        {
            "location": 0,
            "name": "sampleCoord",
            "type": "vec2"
        }
```

```
Shader 0: HLSL 50 [Standard]
Entry point: main
Contents:
cbuffer buf : register(b0)
{
    row_major float4x4 ubuf_matrix : packoffset(c0);
    float4 ubuf_color : packoffset(c4);
    float2 ubuf_textureScale : packoffset(c5);
    float ubuf_dpr : packoffset(c5.z);
};

Texture2D<float4> _qt_texture : register(t1);
SamplerState __qt_texture_sampler : register(s1);
```

```
    "uniformBlocks": [
        {
            "binding": 0,
            "blockName": "buf",
            "members": [
                {
                    "matrixStride": 16,
                    "name": "matrix",
                    "offset": 0,
                    "size": 64,
                    "type": "mat4"
                },
                {
                    "name": "color",
                    "offset": 64,
                    "size": 16,
                    "type": "vec4"
                },
                {
                    "name": "textureScale",
                    "offset": 80,
                    "size": 8,
                    "type": "vec2"
                },
                {
                    "name": "dpr",
                    "offset": 88,
                    "size": 4,
                    "type": "float"
                }
            ],
            "set": 0,
            "size": 92,
            "structName": "ubuf"
        }
```

...

qt-labs/qtshadertools

- One catch: all done offline for now.
  - No run time processing, so no dynamic shader strings yet.
  - The APIs are there (in addition to the command line tool), but:
    - it can be heavy,
    - the 3rd party dependencies may be tricky to build/deploy for exotic platforms,
    - potential license problems (Apache 2.0 incompatible with GPLv2).
- Remains to be seen how this evolves for Qt 6.

Rolling all this out

- No big bang changes.
  - *We'll work for > 1 year, redo everything, and it will all be great*.
    – wishful thinking
- So cannot just wait until Qt 6.

Rather:

1. Integrate internal enablers (QRhi and backends)
2. Develop side by side, with Qt Quick as the first client:
   - Behavior does not change by default (direct OpenGL still).
   - Opt-in via environment variable or C++ in main().
3. First preview already in Qt 5.14 (Nov 2019)
   - Included also in many of the official, pre-built Qt packages
     -> easy to try out, more feedback

Rendering via the Qt Rendering Hardware Interface

From Qt 5.14 onwards, the default adaptation gains the option of rendering via a graphics abstraction layer, the Qt Rendering Hardware Interface (RHI), provided by the QtGui module. When enabled, no direct OpenGL calls are made. Rather, the scene graph renders by using the APIs provided by the abstraction layer, which is then translated into OpenGL, Vulkan, Metal, or Direct 3D calls. Shader handling is also unified by writing shader code once, compiling to SPIR-V, and then translating to the language appropriate for the various graphics APIs.

To enable this instead of directly using OpenGL, the following environment variables can be used:

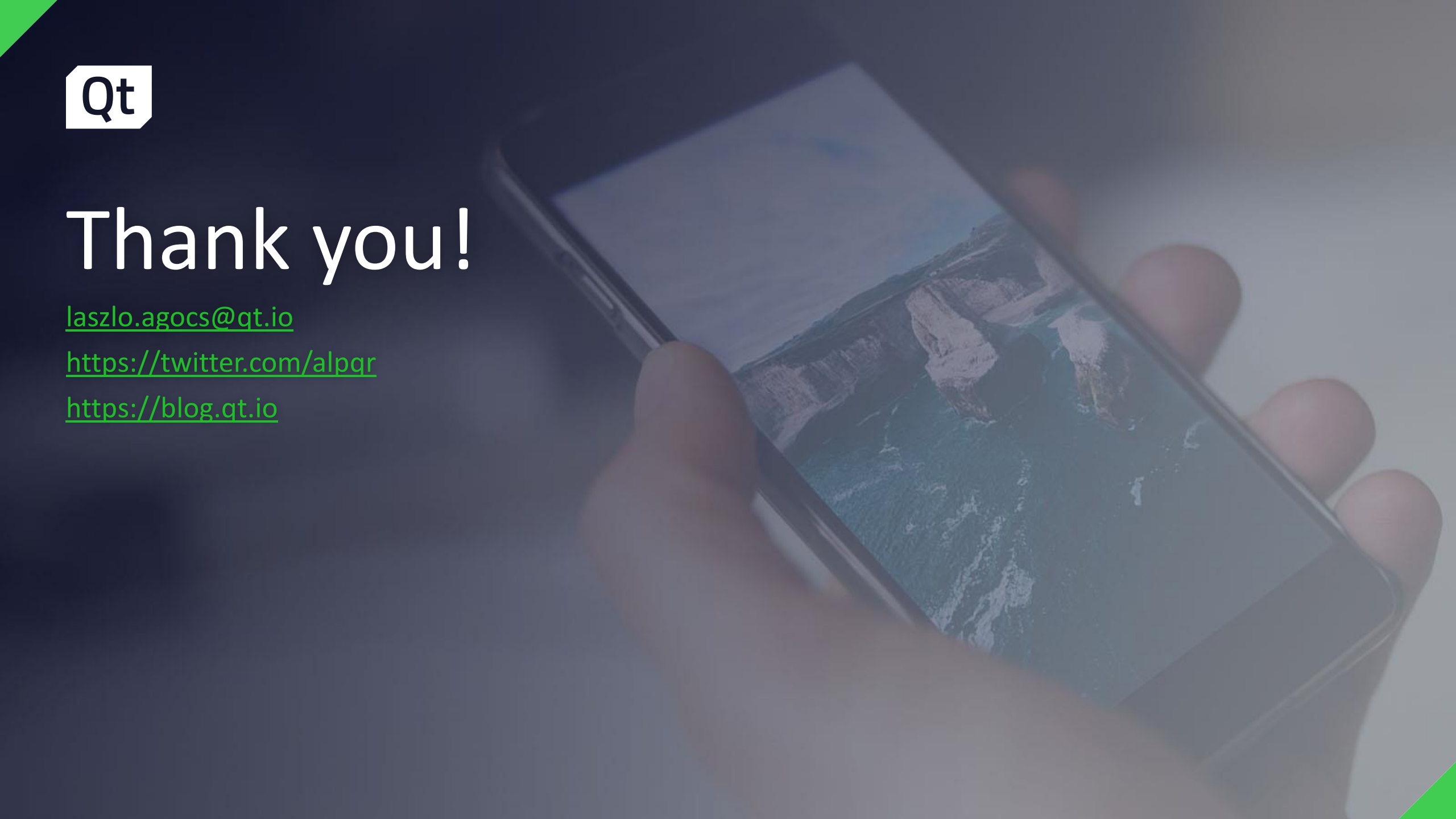
Environment Variable	Possible Values	Description
`QSG_RHI`	1	Enables rendering via the RHI. The targeted graphics API is chosen based on the platform, unless overridden by `QSG_RHI_BACKEND`. The defaults are currently Direct3D 11 for Windows, Metal for macOS, OpenGL elsehwere.
`QSG_RHI_BACKEND`	`vulkan`, `metal`, `opengl`, `d3d11`	Requests the specific RHI backend.
`QSG_INFO`	1	Like with the OpenGL-based rendering path, setting this enables printing system information when initializing the Qt Quick scene graph. This can be very useful for troubleshooting.
`QSG_RHI_DEBUG_LAYER`	1	Where applicable (Vulkan, Direct3D), enables the graphics API implementation's debug and/or validation layers, if available.

Applications wishing to always run with a single given graphics API, can request this via C++ as well. For example, the following call made early in main(), before constructing any QQuickWindow, forces the use of Vulkan (and will fail otherwise);

```
QQuickWindow::setSceneGraphBackend(QSGRendererInterface::VulkanRhi);
```

Is it done?

- Just getting started.
  - The opt-in rendering path in Qt Quick **proves the concept**.
  - **Now** can move towards making it the default in Qt 6.
    - **And then** remove the other code path.
- Then there is Qt Quick 3D, QPainter, and OpenGL-related things all over the place in various Qt modules.
- Target is Qt 6.0 (Nov 2020). We'll see how it goes.



Thank you!

laszlo.agocs@qt.io

https://twitter.com/alpqr

https://blog.qt.io