Integrating OpenGL with Qt Quick 2 applications

Giuseppe D'Angelo, Software Engineer

Qt WORLD SUMMIT 2015

Agenda

KDAB

- Introduction to the Qt Quick 2 renderer
- OpenGL underlays and overlays
- Custom OpenGL-based items
- Controlling the rendering: `QQuickRenderControl`

KDAB

Introduction to the Qt Quick 2 renderer

What is Qt Quick 2?

KDAB

- Framework for modern 2D UIs
  - Scene defined in QML
  - Lots of QML elements out of the box
  - Extensible using C++
- Rendering based on OpenGL
  - Smooth animations
  - Special effects for “free”

The Qt Quick 2 renderer

KDAB

- Renders the contents of a scene graph
  - Data structure containing “visual representation” of the QtQuick elements in a scene
- The scene graph is a tree of nodes, specifying
  - Geometry (i.e. “the shape”)
  - Material (i.e. “how does it look like”)
  - Transformations
  - Clipping
  - etc.

The Qt Quick 2 renderer

KDAB

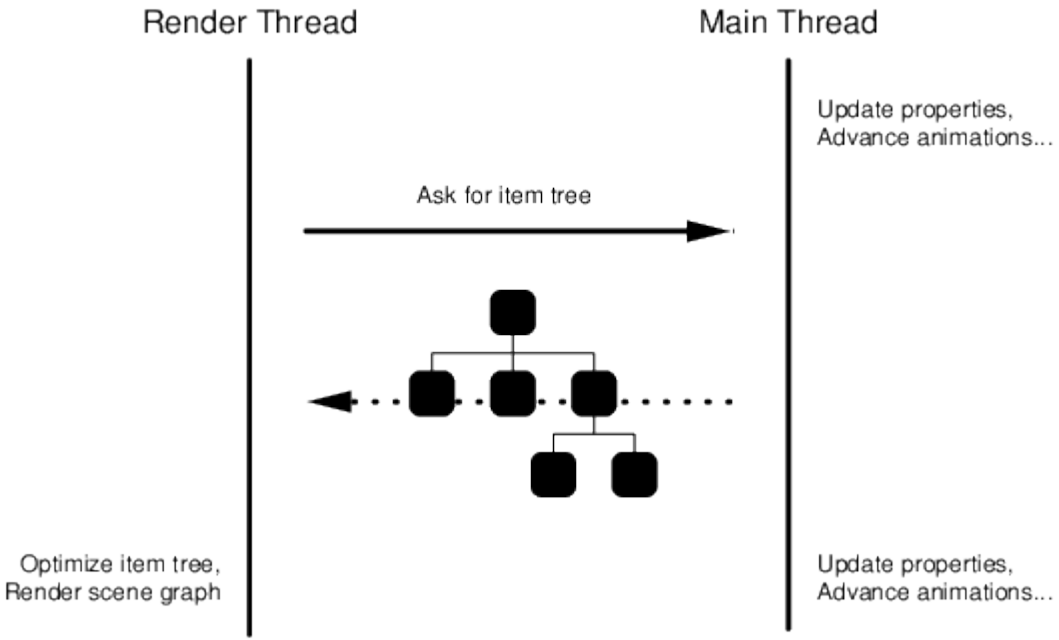
- Rendering is multithreaded on most platforms
  - OpenGL calls issued on a dedicaded render thread != main GUI thread
  - Main thread free to go while render thread submits work to the GPU
  - Render thread free to go in case the GUI thread is stuck
- Explicit main thread / render thread synchronization step
  - During which the scene graph tree for the items in the scene gets created/updated

The synchronization round

KDAB

- Rendering is requested with `QQuickItem/QQuickWindow::update`
- After “some time”, the render thread synchronizes with the GUI thread:
  - GUI thread gets stopped
  - Render thread calls `QQuickItem::updatePaintNode` on all dirty items to retrieve each item's tree of scene graphs nodes
- GUI thread unblocked (free to continue its CPU tasks)
- Render thread analyzes the scene graph + submits work to the GPU

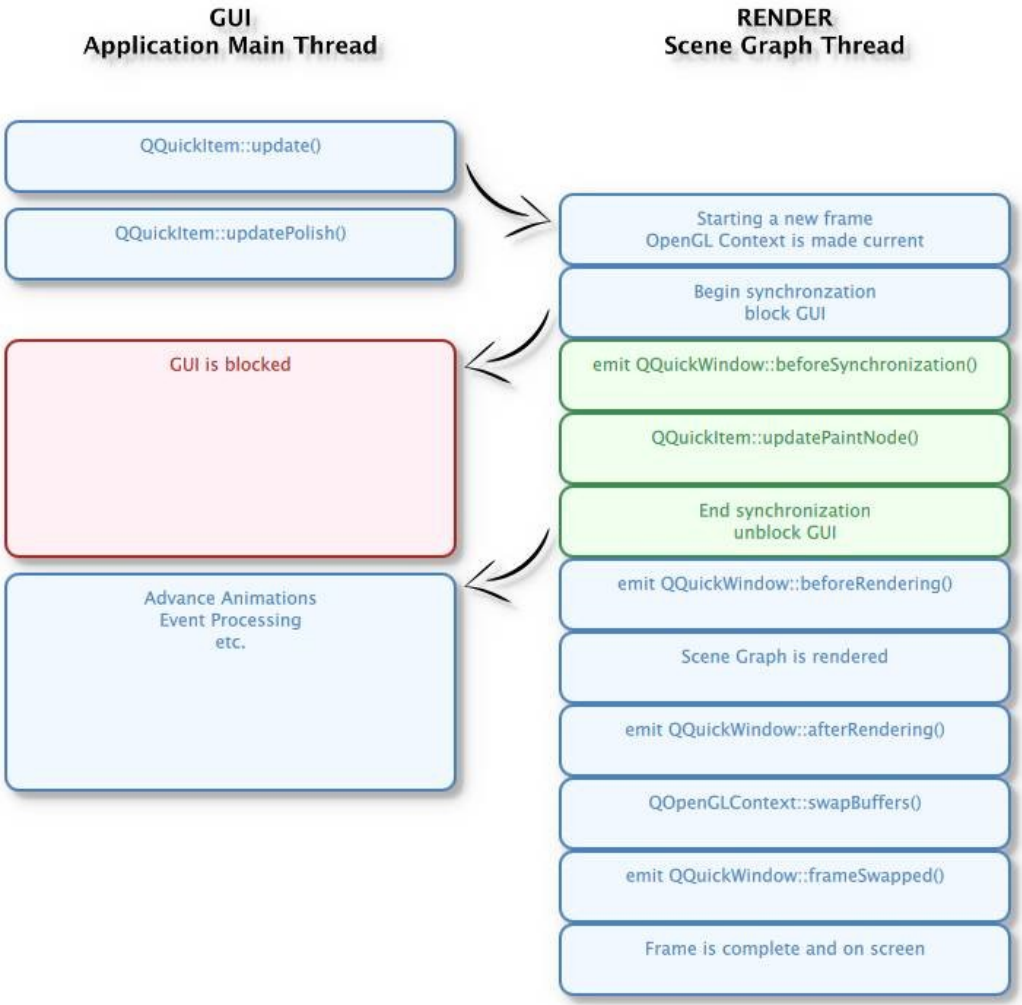
The synchronization round



The *complete* synchronization round

KDAB

- The renderer (through `QQuickWindow`) emits many signals while it proceeds through the synchronization
- We can connect slots to those signals and perform extra drawing using OpenGL


GUI
Application Main Thread
RENDER
Scene Graph Thread
QQuickItem::update()
QQuickItem::updatePolish()
GUI is blocked
Advance Animations
Event Processing
etc.
Starting a new frame
OpenGL Context is made current
Begin synchronzation
block GUI
emit QQuickWindow::beforeSynchronization()
QQuickItem::updatePaintNode()
End synchronization
unblock GUI
emit QQuickWindow::beforeRendering()
Scene Graph is rendered
emit QQuickWindow::afterRendering()
QOpenGLContext::swapBuffers()
emit QQuickWindow::frameSwapped()
Frame is complete and on screen

KDAB

KDAB

OpenGL underlays and overlays

QQuickWindow signals

KDAB

- `QQuickWindow::beforeSynchronizing`
  - Emitted before calling updatePaintNode on the items; GUI thread blocked
- `QQuickWindow::beforeRendering`
  - Emitted after the sync, but before any drawing by the Qt Quick renderer; GUI thread running again
- `QQuickWindow::afterRendering`
  - Emitted after the Qt Quick renderer has done, before the frame is swapped
- `QQuickWindow::frameSwapped`
  - Emitted after the swap buffer call

QQuickWindow signals (2)

KDAB

- `QQuickWindow::sceneGraphInitialized`
  - Emitted when the scene graph is initialized. The OpenGL context will be current
- `QQuickWindow::sceneGraphInvalidated`
  - Emitted when the scene graph has been destroyed; the OpenGL context is going to be destroyed soon after

OpenGL underlays and overlays

KDAB

- Connect to these signals to implement underlays and overlays
  - Cross thread => direct connection required
- In the slots do your custom OpenGL calls
  - The OpenGL context used by the renderer will be available at that point

Demo

Underlays and overlays – Gotchas

KDAB

- By default the renderer clears the color buffer, wiping out underlays
  - Disable the automatic clearing via `QQuickWindow::setClearBeforeRendering(false)`
- The OpenGL context used by the Qt Quick renderer can be destroyed on certain occasions, f.i. when the window is minimized
  - In your rendering code, connect to the destruction signals from the OpenGL context and clear up all OpenGL resources, recreating them when the context gets recreated
  - Or just disable this behavior: `QQuickWindow::setPersistentOpenGLContext(true)`

Underlays and overlays – Gotchas (2)

KDAB

- The Qt Quick renderer tracks OpenGL state and doesn't like changes under its nose
  - Be sure to reset any state that you change in your rendering code to whatever it was before
- In doubt: call `QQuickWindow::resetOpenGLState` to reset all the non-fixed function state before returning from your slots

Underlays and overlays – Gotchas (3)

KDAB

- Beware of accessing state from the main thread without proper synchronization!
- The main thread is unblocked when `QQuickWindow::beforeRendering` or `QQuickWindow::afterRendering` are emitted
  - Copy any render-specific information when `beforeSynchronizing` is emitted
  - And/or protect all accesses to shared state with mutexes

KDAB

Custom OpenGL-based items

Custom OpenGL drawing *into* a Qt Quick scene

KDAB

- `QQuickItem` is the base class of all visible elements in a Qt Quick 2 scene
  - Convenience common properties, event handlers for input, anchor sizing, etc.
- Create a subclass and expose it to the QML engine
  - Using `qmlRegisterType`
  - The renderer will call `QQuickItem::updatePaintNode` to retrieve the subtree of the scene graph for this item
- Create instances in QML as usual

QQuickItem and the scene graph API

KDAB

- Do you really want to know the plumbing of the scene graph?
- ***“Using the Qt Quick Scene Graph API”*** by Jocelyn Turcotte

Custom OpenGL drawing *into* a Qt Quick scene (2)

KDAB

- Convenience subclasses of `QQuickItem` are available, as playing with the scene graph is no easy task:
- `QQuickFramebufferObject` made specifically for integrating custom OpenGL rendering code through a FBO
  - So we don't touch the complexity of the Qt Quick scene graph API

QQuickFramebufferObject

KDAB

- A convenience subclass to wrap custom OpenGL code into a QML element
- Custom OpenGL rendering redirected offscreen into a FBO
- Creates for us the scenegraph nodes needed for rendering the FBO contents
- Subclass `QQuickFramebufferObject` *and* `QQuickFramebufferObject::Renderer`

QQuickFramebufferObject (2)

KDAB

- Subclass `QQuickFramebufferObject::Renderer`
  - This is the class that actually deals with the custom rendering
- Override `render` to draw
  - Called from the render thread
  - FBO already set up when called; customize FBO creation by overriding `createFramebufferObject()`
- Override `synchronize(QQuickFramebufferObject *)` to synchronize the rendering state with the properties of the QML element
  - Called during synchronization, GUI thread stopped

QQuickFramebufferObject (3)

KDAB

- Subclass `QQuickFramebufferObject`
  - This is the class that we expose to QML
- Override `createRenderer()` to create our custom renderer
  - Called from the render thread during synchronization
- Expose the `QQuickFrameBufferObject` subclass to QML
  - `qmlRegisterType`
- Use it from QML

Demo

KDAB

Controlling the rendering: QQuickRenderControl

Getting in control

KDAB

- In some scenarios we don't want Qt Quick to be in charge of the rendering
- We may want to
  - Use a custom/already existing OpenGL context
  - Decide when to synchronize the scene graph
  - Decide when to redraw the Qt Quick contents
- `QQuickRenderControl` to the rescue

QQuickRenderControl

KDAB

- Use `QQuickRenderControl` to manually drive Qt Quick rendering
- Total control over
  - Scene graph & OpenGL initialization
  - Synchronization
  - Rendering
  - Threading
  - Event handling

Using QQuickRenderControl

KDAB

- Create a `QQuickWindow` and a `QQuickRenderControl`
  - Needs an invisible `QQuickWindow`
  - Historical reasons, don't actually `show()` or `create()` the window surface
- Connect to `QQuickRenderControl's` signals
  - See next slide
- Initialize it with `initialize(QOpenGLContext *)`
  - OpenGL context created by us
  - Or possibly adopted using `QOpenGLContext::setNativeHandle`, etc.

Using QQuickRenderControl (2)

KDAB

- When `QQuickRenderControl::sceneUpdated()` is emitted
  - Call `polish()` from the GUI thread
  - Block the GUI thread and call `sync()` from the render thread
  - … in a single-threaded scenario: just call `sync()`
- When `QQuickRenderControl::renderRequested()` is emitted
  - Call `render()` from the render thread (or the GUI thread if single-threaded)

Event handling

KDAB

- To let Qt Quick handle input events (mouse, keyboard, …) simply forward them to the `QQuickWindow`
  - `QCoreApplication::sendEvent(window, event)`

Demo

Questions?

KDAB

Thank you!

www.kdab.com

giuseppe.dangelo@kdab.com

The Scene Graph API

KDAB

- A series of classes holding visual data
  - Mere “containers”, they don't draw themselves
- Renderer analyzes them and submits work to the GPU
  - Many possibilities for optimizations
  - Batching, maybe instancing in the future, etc.

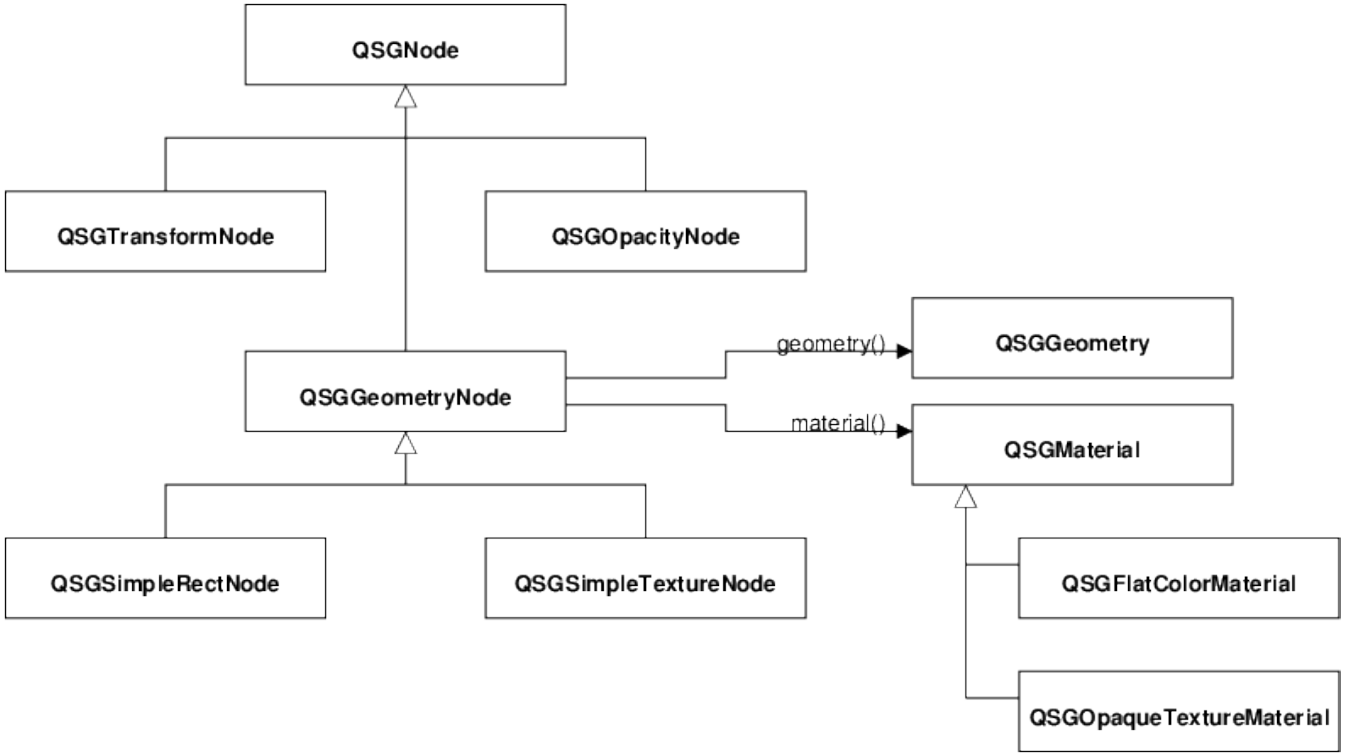
The Scene Graph API

KDAB

- `QQuickItem::updatePaintNode()` returns a tree of `QSGNodes` containing the visual representation for that item
- `QSGNode` base class for actual containers
  - `QSGGeometryNode`
  - `QSGTransformNode`
  - `QSGOpacityNode`
  - etc.
- `QSGNode` is not a `QObject`

The Scene Graph API

KDAB



The Scene Graph API

KDAB

- Although public API, many bits and bolts undocumented or underdocumented
- Check the source code of built-in elements to figure out their scene graph implementation
- Use GammaRay on built-in elements

GammaRay showing the scene graph