Qt World Summit 2016

October 18-20 | San Francisco, USA

Integrating OpenGL with Qt Quick 2 Applications

Jim Albamont, Senior Software Engineer at KDAB

KDAB

Agenda

- Introduction to the Qt Quick 2 renderer (page 3)
- OpenGL underlays and overlays (page 11)
- Custom OpenGL-based items (page 19)
- Controlling the rendering: QQuickRenderControl (page 27)
- The Scene Graph API (page 34)

Introduction to the Qt Quick 2 renderer

- **Introduction to the Qt Quick 2 renderer**
- OpenGL underlays and overlays
- Custom OpenGL-based items
- Controlling the rendering: QQuickRenderControl
- The Scene Graph API

What is Qt Quick 2?

- 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

- Renders the contents of a *scene graph*
  - Data structure containing the "visual representation" of the Qt Quick 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.

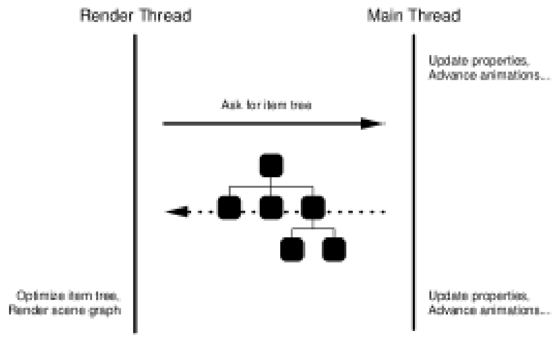
The Qt Quick 2 renderer

- Rendering is multithreaded on most platforms
  - OpenGL calls issued on a dedicated render thread != main GUI thread
  - Main thread free to go while render thread submit works 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

- 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 graph nodes
- GUI thread unblocked (free to continue its CPU tasks)
- Render thread analyzes the scene graph + submits work to the GPU

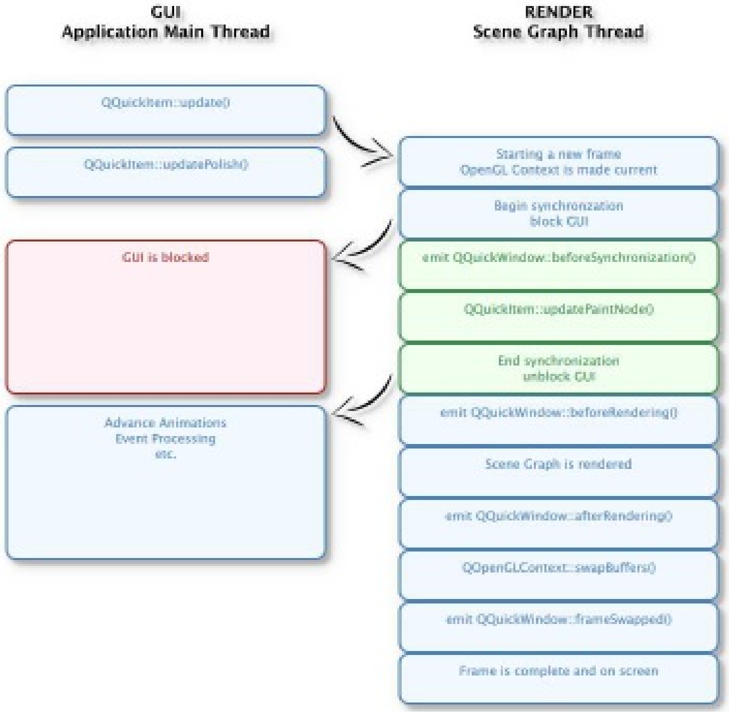
The synchronization round



The complete synchronization round

- 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

The complete synchronization round



GUI
Application Main Thread

RENDER
Scene Graph Thread

QQuickItem::update()

QQuickItem::updatePolish()

Starting a new frame
OpenGL Context is made current

Begin synchronzation
block GUI

GUI is blocked

emit QQuickWindow::beforeSynchronization()

QQuickItem::updatePaintNode()

End synchronization
unblock GUI

Advance Animations
Event Processing
etc.

emit QQuickWindow::beforeRendering()

Scene Graph is rendered

emit QQuickWindow::afterRendering()

QOpenGLContext::swapBuffers()

emit QQuickWindow::frameSwapped()

Frame is complete and on screen

OpenGL underlays and overlays

- Introduction to the Qt Quick 2 renderer
- **OpenGL underlays and overlays**
- Custom OpenGL-based items
- Controlling the rendering: QQuickRenderControl
- The Scene Graph API

QQuickWindow signals

- `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)

- `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

OpenGL underlays and overlays

- 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

- 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 might be destroyed in 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 al OpenGL resources, recreating them when the context gets recreated
  - Or just disable this behavior: `QQuickWindow::setPersistentOpenGLContext(true)`

Underlays and overlays: gotchas (2)

- The Qt Quick renderer tracks OpenGL state and does not like changes under its nose
  - Be sure to reset any state that you change in your rendering code to whatever it was before
- Or: call `QQuickWindow::resetOpenGLState()` to reset the OpenGL state before returning from your custom slots

Underlays and overlays: gotchas (3)

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

Custom OpenGL-based items

- Introduction to the Qt Quick 2 renderer
- OpenGL underlays and overlays
- **Custom OpenGL-based items**
- Controlling the rendering: QQuickRenderControl
- The Scene Graph API

Custom Opengl drawing into a Qt Quick …

- `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

- Will come back to this at the end

Custom Opengl drawing into a Qt Quick ...

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

QQuickFramebufferObject

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

QQuickFramebufferObject

- 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

- Subclass `QQuickFramebufferObject`
  - This is the class that we expose to QML
  - Add properties, signals, etc.
- 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

Controlling the rendering: QQuickRende…

- Introduction to the Qt Quick 2 renderer
- OpenGL underlays and overlays
- Custom OpenGL-based items
- **Controlling the rendering: QQuickRenderControl**
- The Scene Graph API

Getting in control

- 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

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

Using QQuickRenderControl

- Create a `QQuickWindow` and a `QQuickRenderControl`
  - Needs an invisible `QQuickWindow` for historical reasons
  - Do not actually `show()` nor `create()` the window
- Connect to `QQuickRenderControl` signals
  - See next slides
- Initialize the control with `initialize(QOpenGLContext *)`
  - OpenGL context created by us
  - Or possibly adopted using `QOpenGLContext::setNativeHandle()`, etc.

Using QQuickRenderControl (2)

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

Using QQuickRenderControl (3)

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

Demo

The Scene Graph API

- Introduction to the Qt Quick 2 renderer
- OpenGL underlays and overlays
- Custom OpenGL-based items
- Controlling the rendering: QQuickRenderControl
- **The Scene Graph API**

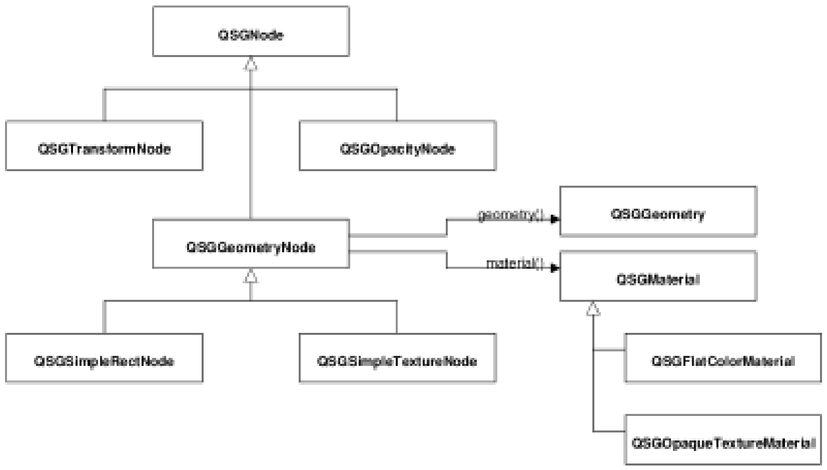
The Scene Graph API

- A series of classes holding visual data
  - Merely "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, ...

The Scene Graph API

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

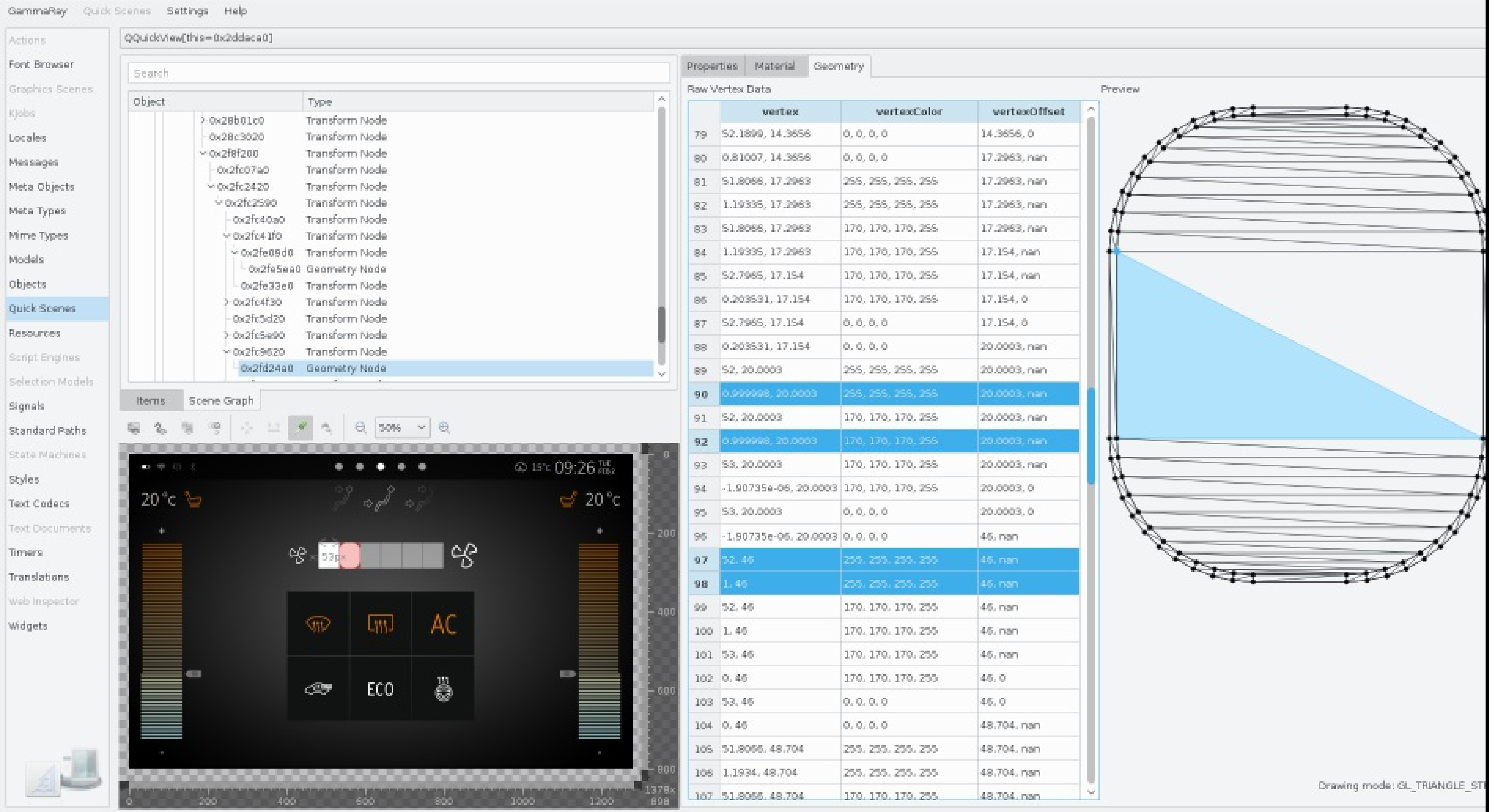
The Scene Graph API



The Scene Graph API

- 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



QQuickView[this=0x2ddaca0]

Object	Type
0x28b01c0	Transform Node
0x28c3020	Transform Node
0x2f8f200	Transform Node
0x2fc07a0	Transform Node
0x2fc2420	Transform Node
0x2fc2590	Transform Node
0x2fc40a0	Transform Node
0x2fc41f0	Transform Node
0x2fe09d0	Transform Node
0x2fe5ea0	Geometry Node
0x2fe33e0	Transform Node
0x2fc4f30	Transform Node
0x2fc5d20	Transform Node
0x2fc5e90	Transform Node
0x2fc9620	Transform Node
0x2fd24a0	Geometry Node

Raw Vertex Data

	vertex	vertexColor	vertexOffset
79	52.1899, 14.3656	0, 0, 0, 0	14.3656, 0
80	0.81007, 14.3656	0, 0, 0, 0	17.2963, nan
81	51.8066, 17.2963	255, 255, 255, 255	17.2963, nan
82	1.19335, 17.2963	255, 255, 255, 255	17.2963, nan
83	51.8066, 17.2963	170, 170, 170, 255	17.2963, nan
84	1.19335, 17.2963	170, 170, 170, 255	17.154, nan
85	52.7965, 17.154	170, 170, 170, 255	17.154, nan
86	0.203531, 17.154	170, 170, 170, 255	17.154, 0
87	52.7965, 17.154	0, 0, 0, 0	17.154, 0
88	0.203531, 17.154	0, 0, 0, 0	20.0003, nan
89	52, 20.0003	255, 255, 255, 255	20.0003, nan
90	0.999998, 20.0003	255, 255, 255, 255	20.0003, nan
91	52, 20.0003	170, 170, 170, 255	20.0003, nan
92	0.999998, 20.0003	170, 170, 170, 255	20.0003, nan
93	53, 20.0003	170, 170, 170, 255	20.0003, nan
94	-1.90735e-06, 20.0003	170, 170, 170, 255	20.0003, 0
95	53, 20.0003	0, 0, 0, 0	20.0003, 0
96	-1.90735e-06, 20.0003	0, 0, 0, 0	46, nan
97	52, 46	255, 255, 255, 255	46, nan
98	1, 46	255, 255, 255, 255	46, nan
99	52, 46	170, 170, 170, 255	46, nan
100	1, 46	170, 170, 170, 255	46, nan
101	53, 46	170, 170, 170, 255	46, nan
102	0, 46	170, 170, 170, 255	46, 0
103	53, 46	0, 0, 0, 0	46, 0
104	0, 46	0, 0, 0, 0	48.704, nan
105	51.8066, 48.704	255, 255, 255, 255	48.704, nan
106	1.1934, 48.704	255, 255, 255, 255	48.704, nan
107	51.8066, 48.704	170, 170, 170, 255	48.704, nan

Questions?

Thank you!

jim.albamont@kdab.com - www.kdab.com

KDAB