Qt Developer Days 2013

Integrating Qt Quick and 3D renderers

James Turner
james@kdab.com

Overview

- Qt Quick 2 rendering model
- Different scene rendering models
- Qt in control
- Qt not in control
- Power-sharing

Qt Quick 2

- QML builds a tree of QQuickItems
- *scenegraph* layer builds parallel tree of `QSGNodes`
  - Created via `QQuickItem::updatePaintNode`
  - Threading synchronization boundary

scene-graph features

- Transforms
- Materials (shader programs)
- Geometry with textures
- Clipping / masking
- Custom nodes
  - with some limitations

rendering model

- In non-threaded OpenGL, simple
  - `QQuickWindow` requests OpenGL support
    - Context with stencil support
  - Update paint nodes from `QQuickItem` state
  - Traverse paint node hierarchy, build batches, submit

Threaded rendering

- Render thread does all OpenGL calls
  - Never waits on main (GUI) thread
  - GUI thread sends custom events to request a sync
- State synchronization is fast
  - No OpenGL calls
- Rendering code is otherwise unchanged

Basic integration

- QQuickWindow provides `beforeRendering` and `afterRendering` hooks
- Use these to make arbitrary OpenGL calls before or after the QSGNode tree is rendered
- Rendering occurs as scheduled by QQuickWindow
- Context is from QQuickWindow

Integration issues

- clear before rendering
- Persistent OpenGL content
- Qt::DirectConnection
- Restore OpenGL state
- OpenGL profile & version

Well, that was easy, wasn't it?

Beer, anyone?

Issues

- Custom OpenGL in a QQ2 item, how?
- What if you can’t control context creation?
- What if you can’t control threading?
- What if you can’t control drawing?

Layering

- Do you actually need to integrate the renderers?
- Instead
  - Overlays
  - Child windows
- Highly dependent on hardware and driver nuances

Adopt-a-context

- `QOpenGLContext` is a nice platform abstraction around `NSGLContext`, `WGLContext`, `GLXContext`, `EGLContext`
  - Unfortunately some crazy people don't use Qt
- They write their own abstraction
- On $platform, how to go from $fooContext to `QOpenGLContext`?

Platform abstractions

- Most renderers have an abstraction layer
  - GraphicsWindow, RenderWindow
  - Target native API / Qt / GLUT
- Abstraction layer covers event-delivery in addition to rendering
  - resizing, closing, mouse & keyboard
- Simplistic compared to Qt

VTK

- Provides a QGLWidget based window
  - And a Qt5 native variant is doable
- Provides a top-level `vtkRenderer::render()` method
  - Compatible with beforeRendering signal
- Straightforward because VTK is not threaded for rendering

VTK integration

- Run VTK rendering under QML
- Ensure VTK only touches OpenGL state when QQ2 renderer permits it
  - from the correct thread
- Forward events to VTK

OpenSceneGraph

- Phased rendering
  - Update
  - Cull
  - Draw
- Configurable threading modes
  - SingleThreaded, DrawThreadPerContext,
  - CullThreadCameraThreadPerContext

OSG continued

- Ideally overlapping each of these traversals of the scene hierarchy
- Internally, OSG is building batches (RenderBins) and other transformation before ultimately calling GL commands
- Render and cull threads, if they exist, are largely hidden from the public API
- All of this strongly parallels QQ2

Let's Battle!

- Only one engine can be in charge of rendering
  - This is a lie, see later
- Engines hide internal threading
- Engines need to share an OpenGL context
  - But not trample each other's state

QQ2 in control

- For libraries which provide a well-defined render entry-point, put QQ2 in charge
  - Providing its timing guarantees work
  - Other rendering code must be thread-safe
- Qt 5.2 adds helper to restore OpenGL state after calling into other code

Privates

I found these interesting private APIs!

Interleaving

- I want my custom OpenGL rendering in an arbitrary scene location via a custom `QQuickItem`
  - I shall use QSGRenderNode, override `render()`, and call my code.
- This OpenGL stuff is easy!

Gunnar's sad face

- Qt 5.2 features a revised scene-graph renderer
- Combines and re-orders all geometry into large batches
- Up to 100 times faster
- Will not tolerate arbitrary OpenGL calls from custom `QSGNodes`
  - Only compose existing node types

Putting QQ2 in a box

- QQ2 contains a (private) plugin API
- We can supply our own renderer, subclassing `QSGRenderer`
  - `render()` method called when QQuick schedules it
  - But who cares about that?!
- External renderer can be in control
- Control context creation

Custom renderer

- We can make our `render()` method do whatever we chose, *or nothing*
- Assuming some entry-point from another renderer, we can use that to setup and run the QQ2 render pass
  - Using the context supplied
- Complex approach

In OSG

- Use a Qt `osg::GraphicsWindow`
  - Updated to use Qt5
- Custom `osg::Drawable`
  - Peer class of our custom QSGRenderer
- Override `drawImplementation()`
  - Run the QSG render code

I don’t want to share!

- Sometimes your render is irreconcilable with QQ2
- But probably supports dynamic textures and similar, as QQ2 itself does
- We can use multiple contexts to keep the renderers apart
  - Use FBOs and texture to move data across

Into QQ2

- `QOpenGLFrameBufferObject`
- Render FBO contents at your leisure
  - Potentially threaded, but beware
- Pass to `QSGGeometry` texture
- Helper for this in Qt 5.2
- Correct, robust approach for custom rendered scene-graph nodes

From QQ2

- QQuickWindow::setRenderTarget
  - to an FBO again
    - careful about threading
    - careful about context sharing
- Full decoupled approach, very appealing

Context-sharing

- QQ2 lacks entry point to pass a share context
  - We have to share from the QQ2 context
- Context-sharing has overhead in the driver
  - Synchronisation of shared objects
- Blitting the data between unshared contexts *might* be faster
  - Or the only choice

Core compatibility

- Legacy renderers often use compatibility profile
- QQ2 currently uses compatibility
  - Soon have an option for core profile mode
- Newer renderers might require, or heavily benefit, from working in Core mode
  - Use 3.x and 4.x features
- No Compatibility profile on Mac

Event-handling

- Easier if QQ2 is in control
  - translate and forward events to your renderer / window abstraction
- Inverse is possible - map from your rendering library to Qt events
  - Simplistic abstractions make this brittle
- Ideally arrange windows such that native routing works

Conclusions

- Many combinations can be made to work
- Potential compromises on one or both sides
  - But often negligible in the real world
- Can impact whole-program architecture
- Check multi-threaded behaviour of your OpenGL code - single thread making all calls is always fastest