Qt Developer Days 2013

QML tricks and treats

Vladimir Moolle

Integrated Computer Solutions, Inc.

ICS

Introduction

This fast-paced talk touches on some of the most basic and yet rarely covered topics a novice-to-intermediate level QML developer may face.

Firstly are covered the issues of:

- structuring mixed QML/C++ applications (properly importing QML from resources, etc.)
- QML language scope / name lookup (knowing which is required for doing anything at least slightly complex)

Building on the above, more practical aspects are addressed:

- creating custom views (what if ListView is not enough for your purposes?)
- keyboard and mouse handling issues
- QML application styling

1. A generalized setup for a moderately complex mixed C++/QML application

goal:

- having multi-project / subdirs (in qmake terms) projects which require zero additional steps upon clone / checkout, but only:
  1) clone
  2) qmake
  3) build

- being able to mix C++ only, pure QML and hybrid C++ / QML projects within the project tree
- being as much platform agnostic as possible in the .pro files while avoiding the worst parts of qmake syntax (“/” vs “\” sometimes do matter, as do “.lib” vs “.a”)
- having dependencies between shared and static libraries and headers correctly tracked during incremental builds
- possibility to access the QML modules both from the C++ applications and qmlscene (during UI prototyping)
- shipping proper QML modules in Qt resources

1. A generalized setup for a moderately complex mixed C++/QML application (continued)

problem A:

- organizing QML sources in proper ("installed", in Qt / QML terms) modules is easy when those are imported from within qmlscene only, but what if the very same code should be shipped in the executable's resources?
- where should QML plugin libraries go, so that they are accessible both from the pure-QML (visual) tests you may have, and also other QML modules and the compiled application as well?
- also, how should images, audio and video assets, logically pertaining to a QML module, be stored and accessed?

tricks:

- importing "../MyModule" (or even "qrc:/MyModule") in QML
- pointing to assets in resources directly:

  ```
  Image { source: "qrc:/images/myImage.png" }
  ```

- copying (either manually or with a qmake script) QML plugin .dll / .so files to where the application will find them

1. A generalized setup for a moderately complex mixed C++/QML application (continued)

treats (QML modules vs plugins and resources):

- access all assets with plain relative paths:

  Image { source: "./images/myImage.png" }
- import all modules by name: import MyModule 1.0
- to access modules from QML files being run with qmlscene, set up .qmlproject's "importPaths" correctly (or add appropriate "-I" arguments to the qmlscene invocation line; add “-P” for plugin paths)
- have all the plugin libraries:

  1) referenced with "plugin xxx_plugin_name" in qmldir

  2) copied to a specific folder upon being built (set up DESTDIR / DLLDESTDIR in your .pri files appropriately)

- put all the QML files, assets and qmldir file itself into application resources, and call:

  1) QQmlEngine::addImportPath(":/resource-path-for-your-qml-modules");

  2) QQmlEngine::addPluginPath("."); //or any other path, but be sure to set dll search paths correctly on Windows then

1. A generalized setup for a moderately complex mixed C++/QML application (continued)

problem B:

- QML objects simultaneously accessed via the class-specific C++ interface on the C++ side, and also instantiated with the usual declarative means on the QML side (what if the same object code gets linked twice -- both into the executable and the QML plugin?)

trick:

- the usual "delete your objects only where you instantiate them (and also watch out for dynamic_cast-s, probably" (bad) advice, and the headaches it brings

treats:

- put the shared code in a shared library (just like Qt does it with its QML items), link both the regular C++ code and the QML plugin to it, and forget about any memory allocation (or related) problems
- (bonus) you can even unit test your QML from C++ in the natural way

1. A generalized setup for a moderately complex mixed C++/QML application (almost finished)

problem C -- the amount of "wrist dance" needed to convince qmake to track:

- INCLUDES (compiler include search dirs)
- PRE_TARGET_DEPS (static libraries)
- DEPEND_PATHS (header dirs watched for changes)

trick:

- properly write down all the slashes and dots and library file extensions for every subproject dependency, test, debug (and hope QBS matures enough soon)

treats:

- generalize and write the tricky parts once (relying on qmake function and variable mechanism), and hide in a set of .pri files
- use a project tree structure rigid enough for the amount of magic and $PWD "Where am I?" error-prone trickery to be kept at minimum

1. A generalized setup for a moderately complex mixed C++/QML application (finished)

an example of a subdirs project following the suggested structure:

```
common.pri
generic.pro

./bin:
        app1
        libpluginmodule1_plugin.so
        libshared1.so
        libshared1.so.1
        libshared1.so.1.0
        libshared1.so.1.0.0
        ...

./app:
        app.pri
        /app1:
                app1.pro
                main.cpp
                ...

./lib:
        libstatic1.a
        libstatic2.a
```

```
./module:
        shared.pri
        static.pri
        /shared1:
                shared1.pro
                shared1.cpp
                shared1.h
                ...
        /shared2:
                shared2.pro
                ...
        /static1:
                static1.pro
                static1.cpp
                static1.h
                ...
        /static2:
                static2.pro
                ...
```

```
./qml:
        qml_plugin.pri
        /PluginModule1:
                PluginModule1.pro
                plugin.cpp
                plugin.h
                PluginModule1.qrc
                Qmldir
                ...
        /tst_pluginmodule1:
                tst_pluginmodule1.qmlproject
                main.qml
                ...
```

2. QML language scope (a short but mostly complete walkthrough)

goal:

- being able to confidently judge on the QML scope / name lookup mechanics even when things get complicated
- using the scoping means to your advantage when designing pure and mixed QML / C++ components

problem:

- the QML scoping rules are not really documented well (yet, but the topic has actually been touched couple of times at DevDays over the years)
- trying to comprehend them from the QQmlXXX / QJsXXX sources will sure be not easy (putting it mildly)

tricks:

- write extremely simplified QML (what if you inherited some?)
- impose a coding standard requiring explicit parameter injection everywhere (no scope chain lookup, id-s usage minimized)
- test and hope for the best

2. QML language scope (a short but mostly complete walkthrough, continued)

treats:

- read all the relevant docs carefully
- stalk Digia employees in charge of QML on StackOverflow (or IRC)
- study the sources of standard QML items (Repeater, ListView, etc.)
- make experiments

good news:

- we've done it for you!
- the results of the investigation were presented to the general audience recently in an episode of an ongoing "Effective QML" webinar series, hosted by ICS (1 hour long session, will be available online soon)
- the most important insights are below

2. QML language scope (a short but mostly complete walkthrough, continued)

an informal C++ / QML taxonomy:

- QtObject - a QML side name for the plain old QObject, not necessarily visual
- Item - a QQuickItem on the C++ side, which IS-A QObject (and thus a QtObject), root of all visual items
- component - a QQuickComponent on the C++ side, an inline Component item or an imported item in QML source
- document - a string (most often coming from a .qml file) defining a component

the various object-like entities' runtime ownership -- there are:

- C++ QObject parent / child trees (important: completely different and separate from QML "visual" parent / child trees)
- QML side objects (QtObject and Item instances), garbage collected by the QML engine
- JS objects managed by the JS runtime
- (the strangest thing) "var" properties which are "not QML objects (and certainly not JavaScript object either)", quoting the docs

note: whether a QObject's lifetime is managed by the QML engine, can also be set on a per-object basis manually (see docs on QqmlEngine::ObjectOwnership())

2. QML language scope (a short but mostly complete walkthrough, continued)

scopes, contexts and id-s -- the most important aspects of QML name lookup:

- within a document, id-s have precedence over everything else (note: if a Component is defined inline, it has a separate set of id-s, but the same rules apply)
- if no id is found for a given name, the enclosing item's properties are searched
- then, the document's root item's ones
- if unsuccessful, the search continues within the scope of the component / document, which instantiates this one
- note: neither QML "visual" parenting, nor the QObject one play role here -- the scope search chain is completely static and defined by the document structure
- scope boundaries are introduced by:

  a) QML document root items
  b) inline Components
  c) QQmlContext instances created for view and Repeater delegates, etc.

2. QML language scope (a short but mostly complete walkthrough, finished)

overloading and some aspects of standard items' behavior:

- QML properties can be overloaded, but the ancestor bindings can not be redefined by this
- JavaScript methods of QML items behave as virtual
- both methods and properties can be redefined on per-instance basis
- Repeater, Loader and standard views instantiate delegate components within specially created contexts (QQmlContext instances), where additional "context" properties like "index", "modelData" (and usually others, named after the model role names) are set
- additionally, Loader sets itself as a "context object" for the loaded instance, exposing all of the own properties to it

2. QML language scope (links)

QML modules:

- http://qt-project.org/doc/qt-5.1/qtqml-modules-topic.html
- http://qt-project.org/doc/qt-5.1/qtqml-modules-qmldir.html

JavaScript files, libraries and imports:

- http://qt-project.org/doc/qt-5.1/qtqml-javascript-hostenvironment.html
- http://qt-project.org/doc/qt-5.1/qtqml-javascript-resources.html
- http://qt-project.org/doc/qt-5.1/qtqml-javascript-imports.html

scope and name lookup:

- http://qt-project.org/doc/qt-5.1/qtqml-documents-scope.html

contexts and context properties:

- http://qt-project.org/doc/qt-5.1/qqmlcontext.html
- http://qt-project.org/doc/qt-5.1/qtqml-cppintegration-contextproperties.html
- http://qt-project.org/doc/qt-5.1/qtqml-cppintegration-exposecppattributes.html

3. Approaches to writing custom QML views (now that you know about QML scope rules and contexts)

goal:

- sometimes it is necessary to provide view behavior which is substantially different from that offered by the standard ones (think "circular" list view, for example)

problem:

- there are natural limits to what you can do with the standard views
- for example, ListView is known to be almost impossible to marry with a scroll bar (in defense of the trolls: the topic of ListView showing age and a need for a new view architecture has already been raised)

tricks (and pitfalls):

- combining Repeater (or Instantiator in Qt 5.1+) with positioners can sometimes give desired results (but only if the number of items in the view is small)
- JavaScript logic for dynamically managing delegate instances (as they go in and out of the visible view area) can soon become overly complicated (as stock JavasScript lacks most of the algorithmic power of STL and / or Qt containers)
- also note: pure QML offers no means to set context properties for the delegates (unless you somehow combine dynamic code generation with Loader abuse)

3. Approaches to writing custom QML views (continued)

treats:

- switch to the C++ side!
- track / cache all the necessary data (like desired delegate instance heights when you would like to provide the scrollbars with an "overall" / virtual view size), retrieve as necessary
- use binary search and the other usual algorithmic means to do this fast
- use QQmlContext: create contexts per delegate, and set the necessary properties ("index", data from the model, etc.) on it
- when a delegate instance is no longer needed, either store it in a cache of some sort, or make invisible, and call deleteLater()
- warning: beware of deleting an item from within an onClick() handler in the item itself -- it will cause a crash
- Flickable is your friend (even the standard views use it!), if necessary, put your view into a Flickable's contentItem, and watch for contentX / contentY changes
- auxiliary components like scrollbars can be usually written in pure QML

4. Keyboard and mouse input: using little known features to your advantage

goal:

- being able to confidently rely on QML keyboard focus / focus scope mechanism
- not being limited by MouseArea's approach to hover and drag-and-drop handling

problem:

- QML keyboard focus mechanism is complicated enough to cause some unexpected surprises (like a suddenly unresponsive app)

tricks:

- go bizarre with manual testing (or be smart and utilize Squish!)
- add timer-driven checks for focus staying where it should, etc.

4. Keyboard and mouse input: using little known features to your advantage (continued)

treat -- keep the following in mind (again, a result of a somewhat lengthy investigation):

- the focus property is false by default on all built-in items
- when the scene loads, an item with statically set focus, that should also have the active focus, actually acquires it after the onCompleted() handlers run. Quoting Michael Brasser from the discussion under QTBUG-16313:

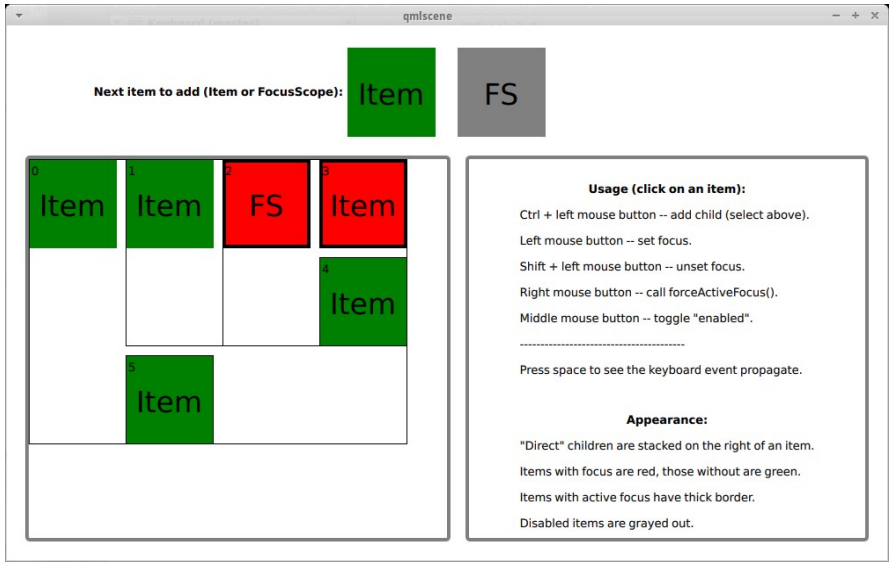
  "In Qt Quick 2.0, [activeFocus changes to true once, at startup]. This is different from most properties (which only report changes after startup). This is due to the fact that the component is created, and then added to the scene, and it is not until it is added to the scene that activeFocus is set (and all property changes after component completion trigger signals handlers)."

- invisible items can have active focus (and focus), but are skipped during keyboard-driven navigation
- disabled items (i.e. those with "enabled" set to false) are skipped during keyboard-driven navigation as well, but can not have active focus
- when user switches to another window, activeFocus value changes to "false"
- probably the best way to debug focus issues is visually

4. Keyboard and mouse input: using little known features to your advantage (continued)

treat -- understanding focus scopes:

- from the application's point of view, the state of focus and activeFocus properties of all items in a QML scene is always consistent
- even when something changes, the QML runtime first makes all the necessary updates, and only then emits the notification signals
- the above means you will never have two items with conflicting focus (not even within an "onFocusChanged" handler)
- a simple visual tool allowing to test various focus-related can be very handy (you can find one in the accompanying sources, a screenshot is below)



ICS

4. Keyboard and mouse input: using little known features to your advantage (almost finished)

problem:

- it is sometimes necessary to simulate keyboard input (imagine writing a virtual keyboard)

tricks:

- emit a signal per key, and properly modify the state of the input-consuming items (track all deletes and backspaces, and arrows by yourself, only items with well-known behavior can be supported)
- use QML TestCase item (http://qt-project.org/doc/qt-5.1/qtquick/qml-testcase.html, part of QtTest module)
- alternatively, on the C++ side QTest namespace (http://qt-project.org/doc/qt-5.1/qttestlib/qtest.html) gives similar possibilities

treats:

- find the item to send events to with QguiApplication::focusObject()
- note: above can be zero, if no item has focus (or the user Alt-Tabbed to another window)
- use QQuickWindow::sendEvent(QQuickItem * item, QEvent * e) or QguiApplication::sendEvent(...)

4. Keyboard and mouse input: using little known features to your advantage (finished)

problem:

- tracking hover over a stack of MouseArea items is difficult
- sometimes the standard drag-and-drop introduces more problems than it solves
- (think dragging from a delegate which gets deleted before a drop occurs, for example)

tricks:

- use fancy mouse event propagation techniques (and some assistance from the C++ side)
- rely on the newer Drag set of attached properties and DropArea item (may be not actually a trick, just not widely used so far)

treats:

- create an item that tracks the mouse, and a custom area item
- hover tracking will become trivial
- drags can be now only initiated by mouse presses, and further handled by dedicated items (which will never go away in the process)

5. Styling QML applications (one curious approach): is clean separation of design and logic achievable with QML?

goal -- what has been promised long ago:

- coders only care about algorithms (and don't care about design issues)
- designers don't have to deal with any (and especially imperative) code, and only apply the UI polish on top of existing logic

problem:

- with usual workflow, designers and coders may end up working on the same QML files
- logic and styling get mixed
- designers get overwhelmed with the complexity of the code and afraid to break something inadvertently
- programmers become stuck in a loop of "move that item N pixels right; then (a week after) -- no, sorry, left!" (no, that's not you, A. B.! :))

5. Styling QML applications (one curious approach, continued)

treats:

- separate logic and styling / UI polish into separate QML files
- further, put those files into separate modules
- asset URLs, item pixel offsets and sizes, and also animations get into "styling" modules
- logic ends up in "core" modules
- designers reside to a purely declarative JSON-like QML syntax (and no JS, of course)
- developers do their job without worrying about styling at all, wiring items and setting up their interactions properly

bonus:

- due to the way QML import mechanism works, a whole set of styling modules may be trivially swapped when a QML scene is created
- (this will not be enough for something like changing styles on the fly, though)
- different OS support becomes much easier (if it is necessary to specify appropriate fonts for each one, etc.)

That's all

The slides and sources for this presentation will be available online* after the US DevDays'13.

Thank you!

* code is coming to https://gitorious.org/~mlvljr soon