23-Jan-2012 dkelly@qnx.com

Basic Qt Speedometer Example

The accompanying archive contains a Qt-Declarative example implementing gauges with coaxial needle indicators. Specifically, the example implements a mocked up automotive speedometer/tachometer cluster sized to match the LCD panel of an i.mx53 QSB (800x480).

You can find similar examples at

- http://doc.qt.nokia.com/4.8-snapshot/declarative-ui-components-dialcontrol.html
- http://doc.qt.nokia.com/4.8-snapshot/declarative-toys-clocks.html

However, the above examples are not driven by C++ code – they are entirely scripted! The example presented here uses scripting only for the UI, with the simulation being written in C++. This hybrid approach means that the UI can readily use data obtained from the RTOS via device drivers and/or other system services.

Since this example is only a simulation, no device drivers are utilized. In fact, the simulation is very simplistic, utilizing a linear power curve implemented within the Qt executable. However, the C++ code could readily be extended to obtain the data from an external source or independent process without impacting the UI.

UI Implementation

The speedometer needle graphic is created with a transparent background and is declared in a transparent rectangle which overlays the base graphic. Note you cannot simply rotate the image since the axis of rotation is not coordinate (0,0). The rotation "origin" coordinates must be correctly specified.

```
transform: Rotation { origin.x:22; origin.y:26; angle: speedMtr }
```

Note that "angle: speedMtr" specifies a variable declared as a signal (from C++) under "Connections", linking the angle of rotation to the C++ simulation.

With just this code, the needle will rotate correctly, but I found the visual results very disappointing. Your eye can see a "digital" effect to the rotation. Fortunately, Qt provides a way to "smooth" the motion by simply adding the clause "behavior on angle{} " with some parameters.

```
Behavior on angle { SpringAnimation { spring: 2; damping: 0.3; modulus: 360 }}
```

This “damping” effect simulates the physical inertia of a mechanical needle producing the desired visual experience. You can easily view results without the “physics” by commenting it out in the script – it is not necessary to rebuild, simply restart. You can also experiment with the behavior parameters in the same way.

Originally, I wanted to implement the tachometer as a rotating “arc” of color. This was attempted by creating a transparent graphic containing the arc and rotating it. However, since the circle containing the arc is a relatively large graphic, the CPU required for the rotation was much too high. Lesson learned – keep the graphic for your indicator as small as possible!

Instead, the tachometer indicator graphic consists of a tiny red-filled circle. The “dot” is placed on a transparent rectangle and the “mostly empty” rectangle is rotated in response to the C++ “signal”. (I found I preferred adding some physics to this element as well.)

[Of course there is an efficient way of implementing an “arc” for the tachometer – simply “draw” the current arc periodically in Qt/C++ code, rather than depending on QML script.]

As provided, the cpu load for the i.mx53 is typically 80-95% with the needles in motion while maintaining a frame count of about 90. With appropriate Qt visual “damping”, it is likely that the frame rate could be lowered and while still maintaining a “smooth” display.

C++ Simulation

The C++ simulation (speedData.cpp) is driven by a Qt thread object containing a while loop. Within the loop, the code alternately runs and sleeps. Each time the thread wakes, it assumes time has advanced by ten milliseconds. Based on that assumption, new angular values for each of the two indicators are generated based on a crude model of the relationship of an engine to the motion of a car.

C++ “signals” the QML script by calling method valuesChanged() with the two angle values and the loop count. The script responds by resetting the position of the two indicators. Additionally, the script computes how many times it has been signaled in one second of real time. This “frame count” is displayed in text at the bottom of the screen. The way this is implemented, the frame count decreases as the time required for graphical processing increases.

The C++ object includes one QML “slot” method which could be called from the script, but no user button is provided to exercise it.

The Archive

Unpack the .zip file to the directory of your choice. A subdirectory "speedo" will be created.

The .zip archive provided contains a Makefile which will build from command line on Windows. This file assumes that QNX Momentics has been installed and the following environment variables are set in the command window prior to starting "make":

```
set MAKEFLAGS=-IC:/QNX650/target/qnx6/usr/include
set PATH=%PATH%;C:\QNX650\host\win32\x86\usr\bin;C:\Program Files (x86)\QNX Software Systems\bin;
set QNX_HOST=C:/QNX650/host/win32/x86
set QNX_TARGET=C:/QNX650/target/qnx6
```

These lines may be put into a .bat (or .cmd) file you run before the "make". The default install to C:\QNX650 is assumed. On some installations, you may have to adjust the "Program Files" directory name.

In addition to the standard QNX 6.5 installation, it is assumed you have installed the Qt host and target files from the "qt_qnx_2011-02-24b.zip" archive at:

http://community.qnx.com/sf/frs/do/viewRelease/projects.qt/frs.binpkg.targets

The Makefile included will only build for one target architecture. To select the target build, uncomment exactly one set of variables, i.e.

```
#x86...
#_QNX_TARGET   = x86
#_QNX_ARCH     = x86
#_QT_TARGET    = i386

#armv7...
_QNX_TARGET   = armle-v7
_QNX_ARCH     = armv7
_QT_TARGET    = armv7
```

Once the environment is set and the Makefile has been adjusted, to build change directory to the "speedo" directory created by the unzip, and enter "make". On a successful build, the "speedo" binary will appear in the "_QNX_TARGET" directory.