Lab – MeeGo

Aim: This lab will take you through the basics of developing and debugging a mobile application for MeeGo.

Duration: 2 h

Introduction

The goal of this lab is to create an application displaying an artificial horizon together with data about your current location. The Qt Mobility API will be used to access the sensor values.

The development process will rely the Qt Simulator platform for debugging and testing. The final result will then be deployed to an actual target device or an emulated environment.

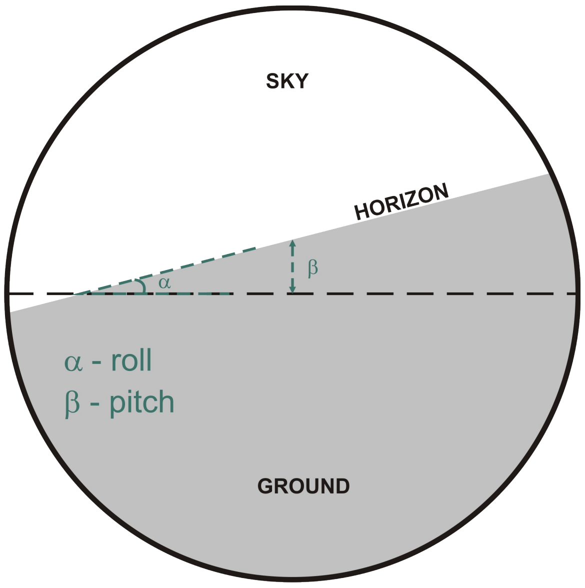
It is assumed you feel comfortable with using Qt/C++ or Qt Quick and no instructions regarding issues related to the programming language will be given. The instructions are based on a C++ based implementation, but you are free to use Qt Quick or a mix of both approaches.

The project consists of two parts. First part is about creating a widget for showing the artificial horizon. The second part will add geographic location (“geolocation”) data to the application.

Artificial Horizon Widget

Start by creating a new mobile application project in Qt Creator. Remember to choose both the simulator and MeeGo as target platforms.

Create a QWidget subclass for displaying the horizon. A schematic image of an artificial horizon is shown in the figure below.



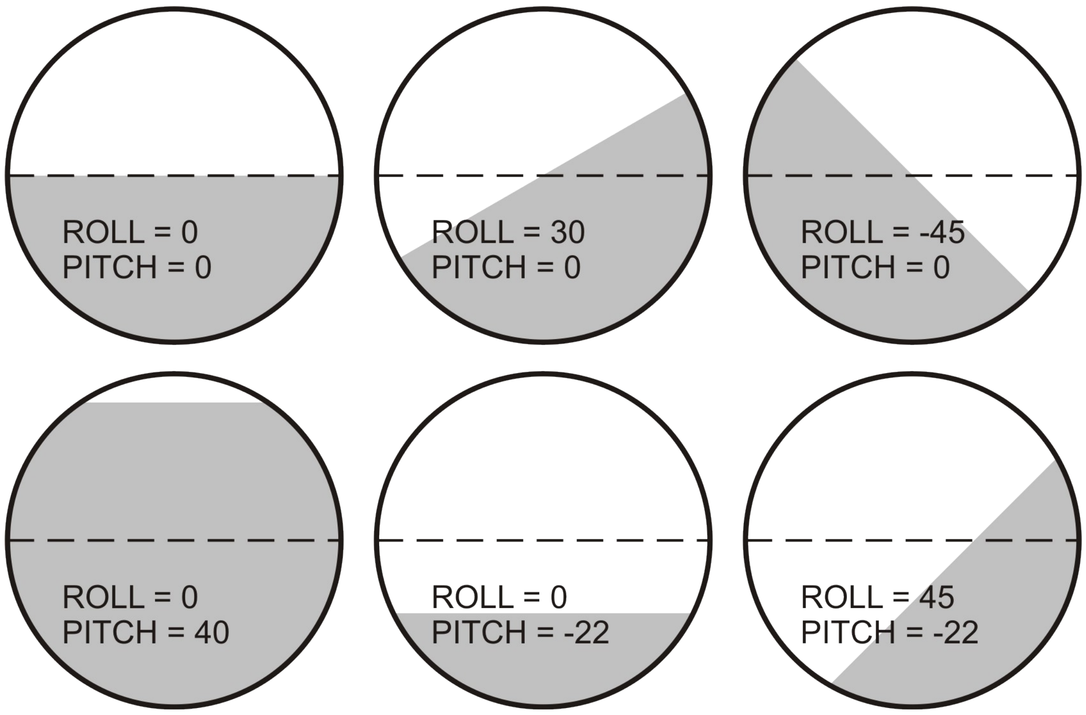
A horizon device is a mandatory indicator in all aircrafts. It is used to show orientation of the aircraft relative to the ground. If the aircraft is positioned perfectly horizontally, the indicator will show a horizontal line crossing the center of the display. The device makes use of two parameters – the pitch and the roll of the aircraft. The first one corresponds to lowering or raising the nose of the airplane, the other one changes when the airplane raises one of its wings. The artificial horizon shows the pilot how the aircraft perceives the true horizon. When the pitch changes, more or less of the ground is shown. When the roll changes the horizontal line rotates in the opposite direction to the craft.

Add two integer properties to your widget – `pitch` and `roll` and implement getters and setters for them. Implement the `sizeHint()` method so that it indicates that the widget likes to be square.

Finally write code for the `paintEvent` handler. Have the widget display a circle and draw the ground indicator based on values of pitch and roll.

The easiest way to draw the ground indicator is to use composition modes. If you use `Source_In` composition mode, painting will only occur on non-transparent areas of what is already drawn (in other words it is a mode where transparency of the painter is "blocked"). This way you can draw the circular background of the indicator, enable the composition mode and draw any shape you like (e.g. a rectangle) and it will automatically be clipped to the background circle. To make it work on every platform, you need to render the indicator to a pixmap (so that you can use composition modes) and then render the pixmap to the widget.

Pitch and roll are given in degrees. Pitch of 90 indicates that the aircraft is flying vertically towards the ground. Roll of 90 indicates that the left wing of the aircraft is pointing towards the sky and the other wing is pointing towards the ground. The image below shows a couple of examples of what the artificial horizon should look like with the corresponding values of pitch and roll.



Test your implementation on the simulator or desktop platform by setting up a GUI consisting of your widget and two sliders that control the pitch and roll of the widget.

Self Check

- Make sure the indicator behaves correctly for easy-checkable values of pitch and roll such as 0, 30, 45, 90;
- Make sure the indicator behaves correctly for combinations of pitch and roll;
- Make sure the indicator scales correctly when the widget is resized

Providing sensor information

Modern mobile devices contain a number of sensors that can be used to determine the state of the device. Two commonly available sensors are the rotation sensor (providing information about the skew of the device around the x, y and possibly also z axis) and the geolocation sensor (usually implementing the A-GPS protocol based on satellite and GSM network information). You can read the sensor values using the QtMobility API; specifically the “sensors” and “location” modules.

To make use of Mobility, add the following entry to your project file:

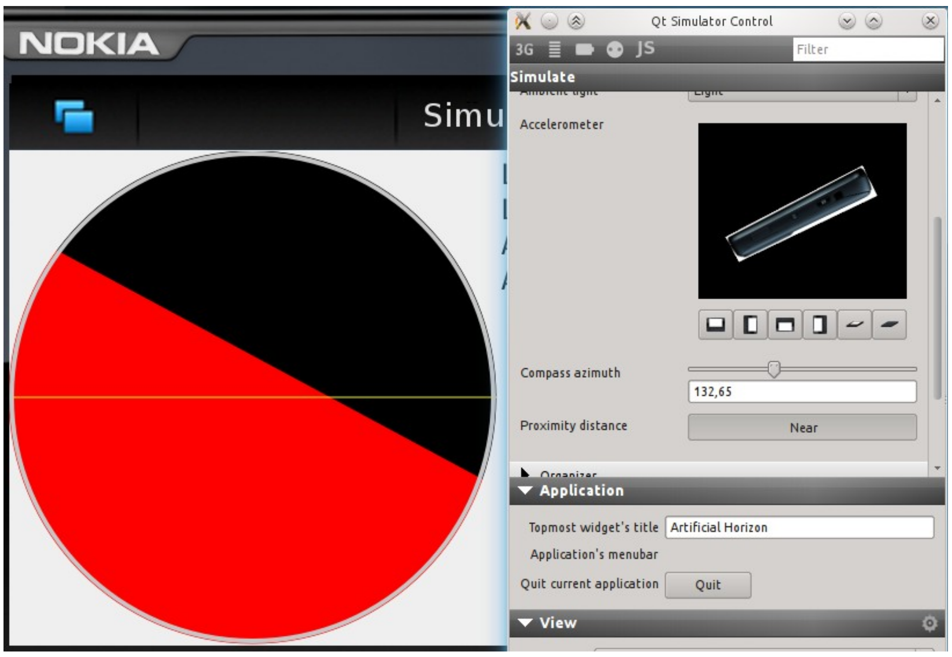
```
CONFIG += mobility
MOBILITY += sensors location
```

All the classes will be contained in the QtMobility namespace. To simplify things, add the `QTM_USE_NAMESPACE` macro to each of the files using QtMobility. This will import the QtMobility namespace into the main namespace of your application.

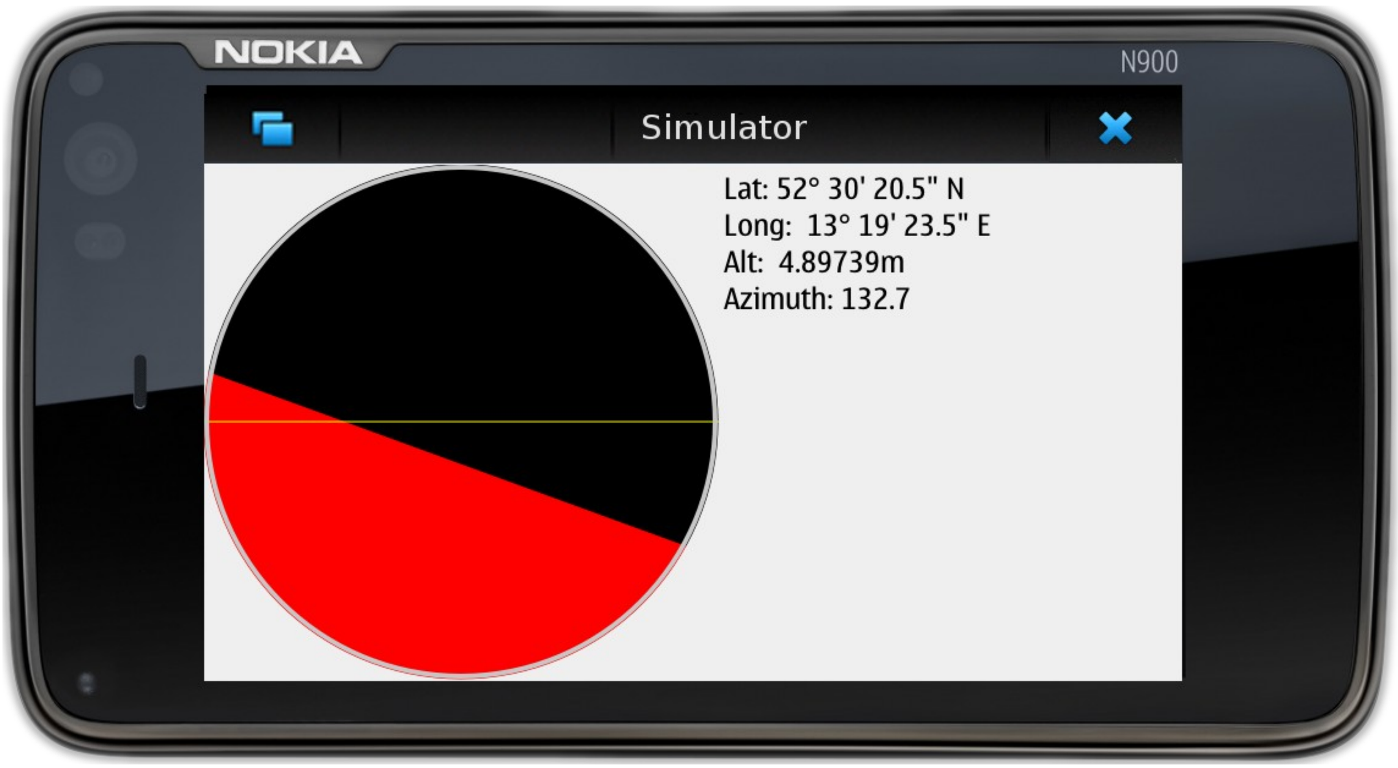
To feed your artificial horizon widget, you will need to use the `QRotationSensor` class. Setup the sensor and connect its `readingChanged()` signal to a custom slot so that you can read the current values of the rotation sensor. In the slot, use the values read from `QRotationReading` to drive the horizon widget with values of pitch and roll.

The sensor may assume that the device is aligned to the ground when it is not laying flat on the table but rather when it is positioned upright as if you held the device in front of your head looking at the device when keeping your head straight. You will have to use a different set of axis than those mentioned in Qt docs if you want the horizon to indicate neutral position while the device is laying on the table.

Build and execute the program for Qt Simulator. Now use the simulator panel to alter the values reported by the rotation sensor to see if your widget behaves correctly. Rotation is controlled by the “Accelerometer” settings in the “Sensors” panel of the simulator control window.



Add a couple of labels to your application that will be showing the direction and location of the device like in the image below.



Lat: 52° 30' 20.5" N
Long: 13° 19' 23.5" E
Alt: 4.89739m
Azimuth: 132.7

Use `QCompass` class to attach compass readings to the program.

Not every device has a compass sensor. Make sure `QCompass::reading()` doesn't return null before trying to dereference the pointer. The same situation applies to the rotation sensor.

The only missing piece is the geographic location of the device. That information is provided by `QGeoPositionInfoSource`. Use its `createDefaultSource()` to get access to the best available source of data for a particular device. Use the `positionUpdated()` signal to get readings from the location source.

You can use `QGeoCoordinate::toString()` to convert numerical data to degrees, minutes and seconds. You will receive a string containing latitude, longitude and possibly altitude readings separated by commas. You can use `QString::split(“,”)` to separate this string into individual readings.

The location source may also report a direction of movement of the device through attributes. You can use this information if available if the device is missing a compass sensor.

Build the application for the simulator and ensure that changing the location values in the simulator control panel correctly updates the location displayed in the application.

When you are done, build the program for MeeGo and test it either in the virtual environment or using a real device.

Self-Check

- Check that you start the sensors by calling `start()` on them.
- Check that your application doesn't crash if a sensor is missing.
- Check that the device behaves sanely in case a sensor is missing; it is a good idea to report “N/A” or “not available” for every data piece that can't be retrieved.