# QML horizon component for camera apps

This article explains how to use the custom QML horizon element to add a horizon line over the camera image. It also explains the element implementation.

## Introduction

This QML horizon element draws a horizon line over the top of its parent item. It's not a simple inclinometer, the horizon line will follow the real Earth horizon according to the device orientation. For example, being at a beach and pointing the device to take a photo at the sea, this line will match the line between sea and sky. Not useful at the sea, but on mountain or in closed spaces, could be helpful to take the right picture. I taken these two photos at a high floor of a palace to have an idea of this.

horizon in portrait

horizon in landscape

(Apologies for poor image quality - the screencapture app I used doesn't work with the camera)

The element uses the QAccelerometer sensor to determine the angle of device rotation. The sensor must be calibrated by the user - the code below shows how - note however that in this case the calibration information is not stored and will need to be done every time the element is used. This is discussed further in the implementation section.

## Usage

Copy **horizon.cpp** and **horizon.h**into your project source directory. In the **main.cpp** file include the **.h**:

```
#include "horizon.h"
```

Then register the component:

```
qmlRegisterType<Horizon>("Horizon", 1, 0, "Horizon");
```

In your qml file you can now declare a `Horizon` component:

```
import QtQuick 1.1
import Horizon 1.0

Rectangle {
        id: mainRect
        Horizon {
            id: horizon
            anchors.fill: parent
            clip: true
            active: true
            penWidth: 3
            color: "black"
        } // Horizon
}
```

Setting the `active` property to true activates the accelerometer sensor and displays a line on top of `mainRect` with a black `color` and a `penWidth` of 3 pixels.

`Horizon` component has also six properties (`calibrate[Min]`) that are needed to calculate the position of the line. By default, min values are set to zero, max values are set to 9.8. All devices need sensor calibration, which involves requesting user input. The following code asks the user to put the device in vertical portrait mode, then in landscape and finally horizontal with display facing upward:


(C) Copyright Nokia 2013. All rights reserved.

```
Dialog {
    id: calibratePortrait
    title: Text {
        anchors.horizontalCenter: parent.horizontalCenter
        font.pixelSize: 28
        color: "white"
        text: "Rotation sensor calibration"
    }

    content: Item {
        height: 50
        width: parent.width
        Text {
            font.pixelSize: 22
            anchors.centerIn: parent
            color: "white"
            text: "Put your phone in portrait and press Ok"
        }
    }

    buttons: ButtonRow {
        anchors.horizontalCenter: parent.horizontalCenter
        spacing: 30
        Button {
            width: 100
            text: "OK"
            onClicked: {
                horizonMenu.open=false;
                horizon.calibrateMinX=horizon.averageRotX;
                horizon.calibrateMaxY=horizon.averageRotY;
                horizon.calibrateMinZ=horizon.averageRotZ;
                calibratePortrait.accept();
                calibrateLandscape.open();
            }
        }
        Button {width: 100; text: "Cancel"; onClicked: calibratePortrait.reject() }
    }
} // calibratePortrait


Dialog {
    id: calibrateLandscape
    visualParent: mainPage
    title: Text {
        anchors.horizontalCenter: parent.horizontalCenter
        font.pixelSize: 28
        color: "white"
        text: "Rotation sensor calibration"
    }

    content: Item {
        height: 50
        width: parent.width
        Text {
            font.pixelSize: 22
            anchors.centerIn: parent
            color: "white"
            text: "Put your phone in landscape and press Ok"
        }
    }

    buttons: ButtonRow {
        anchors.horizontalCenter: parent.horizontalCenter
        spacing: 30
        Button {
            width: 100
            text: "OK"
            onClicked: {
                horizon.calibrateMaxX=horizon.averageRotX;
                horizon.calibrateMinY=horizon.averageRotY;
                calibrateLandscape.accept();
                calibratePlane.open();
            }
        }
        Button {width: 100; text: "Cancel"; onClicked: calibrateLandscape.reject()}
    }
} // calibrateLandscape

Dialog {
    id: calibratePlane
    visualParent: mainPage
    title: Text {
        anchors.horizontalCenter: parent.horizontalCenter
        font.pixelSize: 28
        color: "white"
        text: "Rotation sensor calibration"
    }

    content: Item {
        height: 50
        width: parent.width
        Text {
            font.pixelSize: 22
            anchors.centerIn: parent
            color: "white"
            wrapMode: Text.WrapAnywhere
            text: "Put your phone on a horizontal plane (face up) and press Ok"
        }
    }

    buttons: ButtonRow {
        spacing: 30
        anchors.horizontalCenter: parent.horizontalCenter
        Button {
            width: 100
            text: "OK"
            onClicked: {
                horizon.calibrateMaxZ=horizon.averageRotZ;
                calibratePlane.accept();
            }
        }
        Button {width: 100;text: "Cancel"; onClicked: calibratePlane.reject()}
    }
} // calibrateLandscape
```

You can bind the open event of the first dialog ( `calibratePortrait` ) to a button, when accepted the second dialog is displayed and so for the third.


(C) Copyright Nokia 2013. All rights reserved.

As you can see, we now have another 3 properties: `averageRotX` `averageRotY` `averageRotZ`. To avoid flickering of the horizon line, the component implements an average calculation of a defined number of samples of the accelerometer sensor. In these properties is stored the actual average of the last n sensor readings.

# Implementation

This section explains the `Horizon` implementation. You can bypass this section and skip straight to the source download if you're only interested in using the component.

The element uses QAccelerometer sensor instead QRotationSensor. The reason is that some devices don't support Z-axis and, for example, my Nokia C7 has a snap of 15° deg of values. That said, it would be easy to replace the current accelerometer with the rotation sensor.

One of the main implementation issues is that the sensors must be calibrated for each individual device. To resolve this problem, I added 6 calibration properties (min and max values for each axis). These must be provided by the user - in the example I record these by requesting the user put the device in 3 different orientations. This example does not store the values (which would make sense)! A solution could be store them using QSettings as done in the OMCcam project and in the article Using QSettings in QML with also json and XML support.

## Readings management

In the header file is defined

```
#define ROT_SAMPLES 5
```

this defines the number of rotation readings to be averaged in order to avoid shaking of the horizon line (without this the readings values are very wobbly). I found that 5 samples are enough, but it's possible to increase it to 10 or 20 and see which best value fit a specific device. Less computation occurs if this value is reduced, since the computation is done every time a new reading arrives.

I opted to use a circular list to store the last `ROT_SAMPLES`. This list is composed by this struct and declarations:

```
typedef struct listRot_ {
    qreal value;
    struct listRot_ *curr,*prev,*next;
} listRot;

listRot *rotX_samples[ROT_SAMPLES];
listRot *rotY_samples[ROT_SAMPLES];
listRot *rotZ_samples[ROT_SAMPLES];
listRot *currRotX;
listRot *currRotY;
listRot *currRotZ;
```

These 3 arrays are initialized in constructor:

```
m_rotation = new QAccelerometer();

for (int i = 0; i < ROT_SAMPLES; i++) {
    rotX_samples[i] = new listRot;
    rotY_samples[i] = new listRot;
    rotZ_samples[i] = new listRot;
}
for (int i = 0; i < ROT_SAMPLES; i++) {
    rotX_samples[i]->prev = (i>0 ? rotX_samples[i-1] : rotX_samples[ROT_SAMPLES]);
    rotX_samples[i]->next = (i<ROT_SAMPLES-1 ? rotX_samples[i+1] : rotX_samples[0]);
    rotX_samples[i]->value = i;

    rotY_samples[i]->prev = (i>0 ? rotY_samples[i-1] : rotY_samples[ROT_SAMPLES]);
    rotY_samples[i]->next = (i<ROT_SAMPLES-1 ? rotY_samples[i+1] : rotY_samples[0]);
    rotY_samples[i]->value = i;

    rotZ_samples[i]->prev = (i>0 ? rotZ_samples[i-1] : rotZ_samples[ROT_SAMPLES]);
    rotZ_samples[i]->next = (i<ROT_SAMPLES-1 ? rotZ_samples[i+1] : rotZ_samples[0]);
    rotZ_samples[i]->value = i;
}
currRotX = rotX_samples[0];
currRotY = rotY_samples[0];
currRotZ = rotZ_samples[0];

connect(m_rotation, SIGNAL(readingChanged()), this, SLOT(onReadingChanged()));
```

the slot is connected to `onReadingChanged()`. This function is connected only for the time needed to fill the list. Then the connection is 'redirected' to `onReadingChanged2()` where new readings will be stored sequentially (circulary?) in the list, averages are calculated and the reimplemented QGraphicsItem::paint() function called:

```
// Slot called when the circular list isn't filled yet. When it is filled, change the connected slot to onReadingChanged2()
void Horizon::onReadingChanged()
{
    // fill the circular list of samples
    if (!m_reading) return;
    if (!m_rotation->isActive() || m_rotation->isBusy()) return;
    static int count = 0;

    rotX_samples[count]->value = m_reading->x();
    rotY_samples[count]->value = m_reading->y();
    rotZ_samples[count]->value = m_reading->z();

    // first ROT_SAMPLES samples are achieved, so onReadingChanged2() will be used next time
    if (count>ROT_SAMPLES) {
        disconnect(m_rotation, SIGNAL(readingChanged()), this, SLOT(onReadingChanged()));
        connect(m_rotation, SIGNAL(readingChanged()), this, SLOT(onReadingChanged2()));
    }

    count++;
}
```


(C) Copyright Nokia 2013. All rights reserved.

```
// Slot called when the circular list is filled
void Horizon::onReadingChanged2()
{
    // calculate an average of the last ROT_SAMPLES samples. This prevent horizon line to shake
    if (!m_reading) return;
    if (!m_rotation->isActive() || m_rotation->isBusy()) return;

    currRotX->value = m_reading->x();
    currRotY->value = m_reading->y();
    currRotZ->value = m_reading->z();

    for (int i=0; i<ROT_SAMPLES; i++, currRotX=currRotX->next, currRotY=currRotY->next, currRotZ=currRotZ->next) {
        m_average_rotX += currRotX->value;
        m_average_rotY += currRotY->value;
        m_average_rotZ += currRotZ->value;
    }

    m_average_rotX /= ROT_SAMPLES;
    m_average_rotY /= ROT_SAMPLES;
    m_average_rotZ /= ROT_SAMPLES;

    currRotX=currRotX->next->next;
    currRotY=currRotY->next->next;
    currRotZ=currRotZ->next->next;

    update();

    emit averageRotXChanged();
    emit averageRotYChanged();
    emit averageRotZChanged();
}
```

## Horizon drawing

To draw the horizon line QGraphicsItem::paint() is reimplemented and `update()` is called only when the circular list is filled:

```
void Horizon::paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    Q_UNUSED(option)
    Q_UNUSED(widget)
    qreal rotX, rotY, rotZ;

    // If the device is horizontal +- 45° then exit
    if ( qAbs(m_average_rotZ) > (m_calibrateMaxZ-m_calibrateMinZ)/2 ) return;
```

When the device has an inclination which point below the real horizon, don't draw the line and exit.

```
    painter->setPen(pen);

    if(smooth() == true)
        painter->setRenderHint(QPainter::Antialiasing, true);

    // Normalize rotation values to PI/2
    rotX = (m_average_rotX-m_calibrateMinX)*M_PI_2/(m_calibrateMaxX-m_calibrateMinX);
    rotY = (m_average_rotY-m_calibrateMinY)*M_PI_2/(m_calibrateMaxY-m_calibrateMinY);
    rotZ = (m_average_rotZ-m_calibrateMinZ)*M_PI_2/(m_calibrateMaxZ-m_calibrateMinZ);
```

The acceleration readings, have a range of +/- 12~9.8, rotX, rotY and rotZ have the task to trasform these values in radians regardles the calibration values.

```
    qreal teta;
    teta = -atan(rotY/rotX);

    // Calculate different average position for portrait and landscape
    qreal w,h;
    if ( qAbs(m_average_rotY) < (m_calibrateMaxY-m_calibrateMinY)/2 ) {
        w = boundingRect().width();
        h = boundingRect().height();
        if ( m_average_rotX<0 ) h *= -1;
    }
    else {
        w = boundingRect().height();
        h = boundingRect().width();
        if ( m_average_rotY<0 ) { w *= -1; h *= -1; }
    }
    // Calculate 2 point of the horizon slope
    // At 45° deg the horizon should be at top of the screen (rotZ*2)
    m_x1 = boundingRect().width()/2  + cos( (rotZ*2+M_PI_2) )*w/2;
    m_y1 = boundingRect().height()/2 + cos( (rotZ*2+M_PI_2) )*h/2;

    m_x2 = m_x1 + cos(teta);
    m_y2 = m_y1 + sin(teta);
```

This code calculate the 2 points coordinate of the slope taking care of all the four device orientations.

```
    // With the calculated slope ( (y2-y1)/(x2-x1) ) get the intesction with the (2 edges) bounding rectangle
    qreal x,y;
    qreal x1,y1,x2,y2;
    qreal *px, *py;

    px=&x1;
    py=&y1;

    // Find intersection with upper edge. When a pair is found px and py will point to the x2 and y2 values to set them
    if ( findIntersection( m_x1,m_y1, m_x2,m_y2,  0,0, 1,0,  x,y ) &&
         ( x <= boundingRect().width() && x>=0 ) ) {
        *px=x;
        *py=0;
        px=&x2;
        py=&y2;
    }

    // Find intersection with bottom edge
    if ( findIntersection( m_x1,m_y1, m_x2,m_y2,  0,boundingRect().height(), 1,boundingRect().height(),  x,y ) &&
```


(C) Copyright Nokia 2013. All rights reserved.

```
        ( x <= boundingRect().width() && x>=0 ) ) {
        *px=x;
        *py=boundingRect().height();
        px=&x2;
        py=&y2;
    }

    // Find intersection with left edge (add a little bit to x2 to avoid nan)
    if ( findIntersection( m_x1,m_y1, m_x2,m_y2,  0,0, 0.001,10,  x,y ) &&
         ( y <= boundingRect().height() && y>=0 ) ) {
        *px=0;
        *py=y;
        px=&x2;
        py=&y2;
    }

    // Find intersection with right edge (add a little bit to x2 to avoid nan)
    if ( findIntersection( m_x1,m_y1, m_x2,m_y2,  boundingRect().width(),0, boundingRect().width()+0.001,10,  x,y ) &&
         ( y <= boundingRect().height() && y>=0) ) {
        *px=boundingRect().width();
        *py=y;
    }
```

Here we calculate the intersection points of the slope to the bounding rectangle of this item. `findIntersection()` function will return true if there is an intersection and we store these values in `x` and `y`.

Inside the `if` statement we also check if this point touches the Item edge. If so, store the point coordinate. Since a rectangle can be intersected at most 2 times by a line, when an intersection is found, *px and *py pointers are switched to the 2nd point variables being sure that will be valuated in the next if statement.

```
    painter->drawLine(x1, y1,   x2, y2);
}
```

... and finally draw the horizon !

## Summary

This component is part of the OMGcam Nokia Developer Project which uses Using QSettings in QML with also json and XML support to store the calibration values since they are needed to compute the horizon line.

A lack for camera use, is that at this stage, the component doesn't take care of the camera zoom. I must evaluate which are the changes to make when the zoom is more the 1.

## Download

File:Horizon-v1.0.zip Download OMGcam for Nokia Belle to see how it works.


(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.