Making a facing direction aware camera in Qt

This article explains how to implement a camera that is aware of the direction it is facing. Developers can use this data to enhance the user experience. This can be applied for augmented reality applications, camera based games or some interesting UI - for example, we include a demonstration app that let's you throw baseballs at your surroundings and punch holes that stick to their positions in real world.

Note: This is an entry in the PureView Imaging Competition 2012Q2

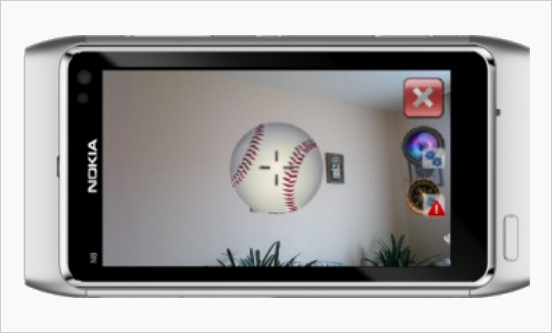
Introduction

This article shows how to implement a camera that has a way of determining the direction it is facing and build upon this data to enhance the user experience. We use various sensors (namely accelerometer, magnetometer\compass and gyroscope) to determine current positioning of the device. Elements can then be added to camera frames thus adding Augmented Reality dimension to the app.

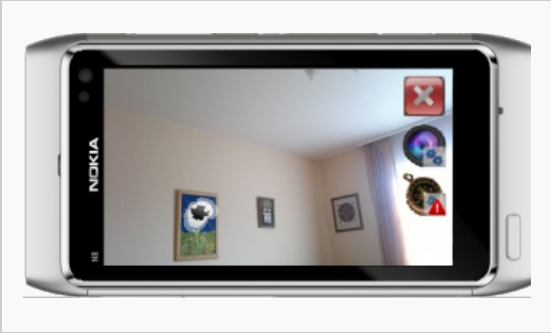
Please note, that generally the gyroscope is needed to fully realise the idea. Since there is no live Symbian^3 device in the market (at the time of writing) that has a gyroscope, we use compass sensor instead. The article will also touch the concept of how the gyroscope should be used in this instance. The media player is loading...

Screenshots

A couple of screenshots of the application in action:



Throwing the baseball



After a couple of seconds and moving the device a bit

The Idea

To make this work we:

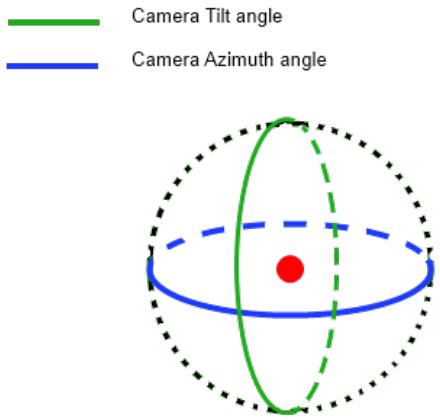
1. create a data model that allows the device to process the positioning. With it we should be able to determine current facing, save positions and approximate field of vision.
2. use the sensors to add real life data to our data model;
3. build on top of this base with our example application to add some Augmented Reality fun.

Space model

We can imagine the data model as a surface of a sphere. Any point on the surface can be determined by two coordinates, in this case we'll call them *azimuth* and *tilt*. GPS uses the same idea to pinpoint location.

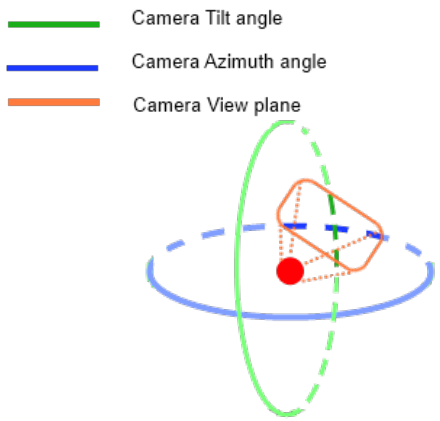
The picture below illustrates this model. If we regard the red dot inside the sphere as the camera we can see that it's possible to tell the direction the camera would be facing with those coordinates:

(C) Copyright Nokia 2013. All rights reserved.



To complicate things a little bit, the camera actually sees a certain ranges of azimuth and tilt and the cross is just the center position of the frame.

We can calculate that though, provided we know the angle of view of the camera, or in our terms: the vertical and horizontal angles the camera can span. We can also easily save positions with our model and check if those are inside our angle of view.



Use of sensors

We use accelerometer for the most part, as it can determine the positioning of the device in all world axes *but one*. Accelerometer calculates the accelerating force when the device is moving. So if the device was to move sharply in a certain direction the accelerometer would provide us with that info.

When the device is not moving though, there is one force that is still affecting the sensor: gravity. The accelerometer always detects it so we can determine the direction of the ground in regards to the device. That is how we determine the vertical facing.

One shortcoming of the accelerometer is that we cannot calculate the horizontal facing from gravity alone. That's where the magnetometer comes in. The magnetometer sensor can detect Earth's magnetic field, so in essence, it can act as a compass and provide us with current azimuth. We can use `QCompass` class that wraps magnetometer nicely for us.

We use the compass mainly because most of the devices have that capability these days as opposed to gyroscope. Magnetometer, however is easily tempered with: it requires calibration upon every use, it can be disturbed by nearby magnetic objects, wires, etc. Gyroscope, on the other hand, calculates the speed that the device is turning at, so in essence it's the accelerometer of rotation. We will have the implementation for it in our example too as the gyroscopes start flooding to the mobile devices.

Implementation

Let's discuss the most important parts of the code.

Moving average

The first thing we need to have is the means to clean the noise that comes from the sensors. One of the easiest ways to do that is with a moving average.

We take a certain number of readings and calculate the average. This tends to eliminate all the sudden spikes and trembles so instead of jumpy behavior we get a more flowing change in values. This, however, might delay the change in values, as it requires a few changed values to come in first before the average starts changing too.

This is implemented in `CMovingAverage` class since we will have quite a few sensors to deal with. It implements the behaviour mentioned above and adds a simple to use interface. Please note, that the example uses different count of values stored for different sensors. The rate in which sensors report changes varies greatly. For example for each of accelerometer's axes 10 values are taken. Compass, on the other hand, uses only one and the rate is still noticeably laggy.

(C) Copyright Nokia 2013. All rights reserved.

Positioning

We use `CMovingAverage` to store information inside `CCamPositioning` class, which is a handy class that has all the sensors and their interworkings combined to get the data we need:

```
class CCamPositioning : public QObject
{
 ...
public slots:
    void AccelChange();
    void CompassChange();
    void RotationChange();

public://getters
    float getAzimuth();//in radians
    float getTilt();//in radians
    float getRotation();//in radians
    float getCompassCalibration(){return iCalibrationLevel;}

private:
    qreal getAccelerationValue(int axis);

private:
    float iCalibrationLevel;
    bool iHasGyro;

    CMovingAverage iAccelerometerVal[3];
    CMovingAverage iRotationVal[3];
    CMovingAverage iCompassVal;

    QAccelerometer iAccelerometer;
    QCompass iCompass;
    QRotationSensor iRotationSensor;
};
```

one thing to note here is the `iHasGyro` property which changes if the `QRotationSensor` detects a change in *z axis* rotation. That basically shows that gyroscope is present on the device and can be used for our benefit. Below we can see the parts where the data differentiates between `QCompass` and `QRotationSensor` based on the value of `iHasGyro`.

```
void CCamPositioning::CompassChange()
{
 iCompassVal.addValue(iCompass.reading()->azimuth());
    iCalibrationLevel = iCompass.reading()->calibrationLevel();
    if (iHasGyro)
        iCalibrationLevel = 1.0;//gyroscopes don't neeed to be calibrated, so we imply it's calibrated if it is used
}

void CCamPositioning::RotationChange()
{
 //add values for moving average calculation on change signal
    QRotationReading *aReading = iRotationSensor.reading();
    iRotationVal[0].addValue( aReading->x() );
    iRotationVal[1].addValue( aReading->y() );
    iRotationVal[2].addValue( aReading->z() );
    if (!iHasGyro)
        if (aReading->z() != 0)//if the horizontal rotation changes, we have a gyroscope on the device
            iHasGyro = true;
}

float CCamPositioning::getAzimuth()
{
    if (iHasGyro)
  //if gyro is present, we use it's z value and turn that to radians
        return iRotationVal[2].getValue() / 180 * M_PI;
    else
 //no gyro - we use compass. BTW compass shows fidderent values when he phone is pitched, yawned etc. so we add the rotation to the
    return iCompassVal.getValue() / 180.0 * M_PI + getRotation();
}
```

Data Model

We take all this and use it inside `CCameraViewField` class which represents our data model and here it all comes together. This class stores the camera view angles and comes with a struct `SViewData` that both helps us transfer positioning data between components and has methods that help calculating point visibility and positions on the view plane.

Two of them require a special mention. First - `isPointWithin(float aAz, float aTilt)`. It uses `normalizeAngle(float)` which returns angle nomralized for comparison operations, and `rotatePt(QPoint, QPoint, float)` which rotates a point around another point a given amount of radians.

```
bool SViewData::isPointWithin(float aAz, float aTilt)
{
    //initializing unrotated values
    float n_iAzimuth = normalizeAngle(iAzimuth);
    float n_aAz = normalizeAngle(aAz);

    //create a viewplane rectangle using current center point and camera angle of view
    QPointF rectangle[4];
    rectangle[0] = QPointF(n_iAzimuth-iHorizontalAngle/2,iTilt-iVerticalAngle/2);
    rectangle[1] = QPointF(n_iAzimuth-iHorizontalAngle/2 + iHorizontalAngle, iTilt-iVerticalAngle/2);
    rectangle[2] = QPointF(n_iAzimuth-iHorizontalAngle/2 + iHorizontalAngle, iTilt-iVerticalAngle/2+iVerticalAngle);
    rectangle[3] = QPointF(n_iAzimuth-iHorizontalAngle/2, iTilt-iVerticalAngle/2+iVerticalAngle);

    //rotate all four corners of the rectangle around the center (if camera was not held perfectly flat)
    for (int i =0; i < 4; i++)
        rectangle[i] = rotatePt(rectangle[i], QPointF(n_iAzimuth, iTilt), iRotation);

    //check if the given point is within the viewplane rectangle
    bool isIn = false;
    int sign = 0;
    for (int i = 0; i < 4; i++)
    {
        int i0 = i;
        int i1 = (i+1) % 4;
```


(C) Copyright Nokia 2013. All rights reserved.

```
        float A = 0.5 * (rectangle[i1].x() * aTilt - rectangle[i1].y()*n_aAz - rectangle[i0].x()*aTilt +
                         rectangle[i0].y() * n_aAz + rectangle[i0].x()*rectangle[i1].y() - rectangle[i0].y()*rectangle[i1].x());
        if (i == 0)
        {
            sign = A;
        }
        if ( (A >= 0 && sign >= 0) || (A < 0 && sign < 0) )
        {
            if (i == 3)
                isIn = true;
        }
        else
        {
            break;
        }
    }

    return isIn;
}
```

The check itself divides the view rectangle into 4 triangles, with a shared vertex - the point that is being checked. The area of one triangle is determined by this formula (where vertical bars represent the determinant):

```
'   1 | x0 y0 1 |
A = - | x1 y1 1 |,
    2 | x2 y2 1 |
```

And the quered point is within the view plane only when all 4 triangles have the area that is eihter negative or positive in all of them. You can check various other approaches here Formula for Point in Rectangle.

The second method is a small but esssential one:

```
QPointF SViewData::getDistFromCenter(float aAz, float aTilt)
{
    QPointF pt = QPointF((aAz - iAzimuth)/(iHorizontalAngle/2), (aTilt-iTilt)/(iVerticalAngle/2));
    pt = rotatePt(pt, QPointF(0,0), iRotation);
    return pt;
}
```

This one returns where is the point, that is checked, in screen coordinates (provided it is withing the view plane in the first place). The returned X & Y values are within range [-1;1]; so this can be used on any sized screen\image: when multiplied by the dimensions of the target, we get exact X and Y points where the final point sits on the screen.

Camera access

Next up is the camera module - `CCamera` class, which is basically a simplified adaptation of Qt Camera Demo. We wrap the `QCamera` in it to provide ourselves with as simple to use camera, as we can get out with. It uses `CMyVideoSurface` that is derived from `QAbstractVideoSurface` to provide us with the frames from the viewfinder to work with.

Once we start the camera with `enableCamera()` method it starts emiting signals whenever a new frame arrives to the video surface. By connecting to this signal we are ready to go with our baseball shooter.

```
class CCamera : public QObject
{
 ...
signals:
    void frameArrived();

private:
    void enableCamera();
 ...
};
```

Throwing baseballs example

We build the augmented reality game on top of the base we've just covered. To be honest it's very straightforward compared to the bottom layer.

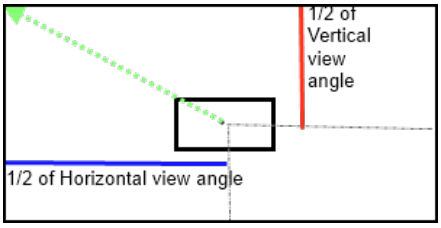
There is the `MainWidget` class that is a derived `QWidget` which controls all the *Views*. Speaking of them there are three: `CMainView`, `CCameraCalibView` and `CCompassCalibView`.

Calibration screens

Compass calibration view displays a graphical progress bar indicating the magnetometer calibration level and instructions to calibrate it.

Camera calibration view displays instructions for calibrating the camera - or to be specific determining the camera view angle. This is done by moving the center of the viewfinder to any of the screen corners, so the spot that was in the corner ends up in the very center. Noting the positions of those two points we can calculate the distance and approximate angle it has traveled. That would be half of the view angle. The picture below illustrates that:

(C) Copyright Nokia 2013. All rights reserved.



Please note, however that this is just an approximation and given noisiness of the sensors, especially magnetometer, it might take a few tries to get it right. The default values in the code are the ones that worked best with N8.

Main View

The third View is the `CMainView`. Here all the magic happens. When the screen is clicked the ball is thrown to where the camera is currently pointing. After a certain amount of time the ball is shown to crack a hole onto the image which keeps it's positioning when the camera is moved around.

Whenever a new frame arrival signal is emitted the redraw of the widget is forced, thus checking the current screen position, checking the list of thrown balls and the holes they made and their position in regards to the view plane. If a point is within a view plane, it's position in the model is transposed to screen coordinates using `SViewData` struct and it's drawn onto the frame. The flying ball animation is done using a `QTimer` that is constantly emiting update events which allows for the ball image to shrink and giving an illusion it's flying into the depth. The code below illustrates this process

```
void CMainView::drawBaseballs(QPainter *aPainter, QSize aPhotoSize, SViewData aData)
{
    for (int i =0; i < iBaseballs.count(); i++)
    {
  //check if object No 'i' is withinb the view plane
        SOverlayObject obj = iBaseballs.at(i);
        if (aData.isPointWithin(obj.iPosition.x(), obj.iPosition.y()))
        {
            //get ball position in relative screen coordinates [-1;1]
            QPointF dist =aData.getDistFromCenter(obj.iPosition.x(), obj.iPosition.y());
            int posX = (aPhotoSize.width() /2) * dist.x() + (aPhotoSize.width() /2);
            int posY = (aPhotoSize.height() /2) * dist.y() + (aPhotoSize.height() /2);

            //calculate the size of the drawn image in accordance to how much time passed since the launch of the balll
            int iSize = int (  200.0 * ((2-obj.iAge)/2.0)  );

            //draw the image using the calculated screen coords and the animated ball size
            aPainter->drawImage(QRect(posX-iSize/2,posY-iSize/2,iSize,iSize), iBallImg, iBallImg.rect());
    }
}
```

For full source please check Media:FacingAware code.zip

Considerations

These are the few considerations to take notice of and maybe improve in regard to your needs.

- Coincidently, since this article was created for `PureView Imaging Competition`, the device that this competition is dedicated to is the first of Symbian devices to be able to handle the article's idea at it's fullest. The Nokia 808 seems to be the first (and hopefully that'll start to be a standard equipment) Nokia's device to sport a gyroscope sensor, which opens a plethora of new opportunities for app developers. Sadly, since, at the time of writing, there was no way to test the working of the gyroscopic implementation. There is no physical device and because of that simulator doesn't simulate the gyroscope values. I'll update the article if/when opportunity to do that arises.
- As mentioned in the article, the bottleneck here is the compass sensor. It's great for applications requiring relatively less accuracy and speed. In this case, however, it updates too slow and in too big of steps sometimes and that throws off the whole augmented reality. One other thing, whilst filming the presentation video, I've made myself a stabilising contraption for this purpose. Since it had metal parts touching around the device, the magnetometer went haywire. Please note your surroundings if your application uses the compass side of implementation.
- The moving average is a great and easy way of evening out spikes in noisy sensors. However it still has a bit of "trembling" if too few values are used and is quite inefficient when using many values. If application requires some precision movement, it might be a good idea to start your research here, hunting for more accuracy in interpolation and speed of operation.
- This article was first thought to be implemented using OpenGL for drawing, since it allows for faster drawing, easier and more efficient effect implementation through shaders and faster image manipulation (image rotation with `QPainter` can overthrow the processor quite quickly.). Though since the size of the idea got a bit out of hand and it seemed there were some problems with accessing frame data from VideoSurface directly as Textures (Please check camera guide 4.5.4.1 Handles) the idea was scrapped. Though if this was to be used inside a game OpenGL should be the way to go.

Summary

This article introduced the camera that can detect its direction and a way to use it for developers benefit. With the advent of gyroscopes in mobile devices, camera apps that have directional awareness can greatly expand the field to which the mobile devices can be applied to. This might be used in navigation, casual games (even requiring as precise movement as First person shooters), photo sharing networks etc; so it is really beneficial to invest some time to research the idea of direction aware cameras.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.