Augmented Reality Applications with NyARToolkit and Qt3D

This article explains how to create an Augmented Reality (AR) application using Qt. The code example adds a "virtual" 3D person to the image displayed in the camera viewfinder.

03 Jun 2012

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

Introduction

This article shows how to create a **Qt Augmented Reality** application using the NyARToolkitCPP project for setting up the camera frames and tracking pattern markers. Note that this is a Japanese webpage, but, it has google translator support. QtQuick3D project is used to display the 3D content on the screen.

The idea is based on the Augmented Reality with Qt. Part 3 tutorial by **dmitry-rizshkov**. The problem is that, in this tutorial, the OpenCV library and native OpenGL API are used. Since there's no easy OpenCV port to both Meego Harmattan and Symbian^3 and many OpenGL native functions don't work on these embedded systems, the Qt Camera API and QtQuick3D project provide a suitable and portable alternative solution.

The steps to create the AR application and have it up and running are:

1. Setup the QtQuick3D
2. Integrate the NyARToolkit source code
3. Create a Custom Camera QML item
4. Create an AR class to process the Camera Frames
5. Create a Custom 3D QML item
6. Display Camera Item + 3D Item contents on the screen

Note: After STEP 1 is complete, it's possible to skip ahead and build the source code for this article (File:Qtar-source-code.tar.gz), if you just want to see the application running right away. The STEPS 2 ~ 6 provide an explanation of the program structure.

The following section provides a video demo of the component in action. See also the screenshots below showing the app running on a Nokia N9 and Nokia C7-00.

Video Demo

The media player is loading...

Warning: On some Symbian^3 devices the application might not display the camera contents immediately (it just shows a black screen in the middle). That's because the camera module initially requires a lot of memory to display on a OpenGL viewport. For now, the only **workaround** for this is to slide the screenlock switch to close and open again (this is usually located on the side of the device), therefore freeing up some memory for the application. A fix for this is currently in the works.

STEPS

Setup the QtQuick3D

Qt3D is a set of C++ APIs for 3D programming built on top of QtOpenGL. QtQuick3D are the QML bindings for Qt3D. The project simplifies creating OpenGL applications by abstracting initially required configurations, such as vextex definition, lighting normal calculations, viewport volume and others.

First, you'll need to create and install the **QtQuick3D** on the device(s). The instructions on this link (http://doc.qt.nokia.com/qt-quick3d-snapshot/qt3d-building.html) can be used for the package creation.

A pre-built version for each platform (Meego Harmattan and Symbian^3) is available here: File:Libqt4-3d 1.0-rc armel.deb.tar.gz and File:Qt Quick 3D.sis, if you want to skip this package creation part.

Getting the source code is still required for integration with the Qt SDK and to be able to build the application. The Qt3D version used on this article (*Qt 4* branch) is available here: http://qt.gitorious.org/qt/qt3d/commits/qt4

Now, after downloading it, go to Qt Creator and open *qt3d.pro* file. If you're building for the Meego Harmattan, first it will ask you to add the *qtc_packaging* folder to the project, just say YES to this. Next, make sure to **Uncheck All** options on the dialog where it asks you to add a bunch of **.pro** files to the build system. If you're building for Symbian^3, just opening the **qt3d.pro** file is enough.

NOTE: If it's necessary to speed up the build time, a couple of lines from the **qt3d.pro** file can be commented, as such:

```
...
## Speed UP build time by skipping the examples, demos, tutorials and utils project dirs.
##
```

```
#!gcov: {
#    SUBDIRS += util examples demos
#}

## DO NOT COMMENT THIS LINE
include(pkg.pri)

#!package: SUBDIRS += tutorials

#SUBDIRS += tests
...
```

After that, you just need to hit the **Build Project** button on Qt Creator. It will create the libraries and copy the **.prf** files to the proper Qt SDK directories. This way, your SDK is ready to build QtQuick3D applications.

Integrate the NyARToolkit source code

This is what the core of the application uses to generate the **Augmented Reality** information. For that, the **NyArToolkitCPP** version is used: http://nyatla.jp/nyartoolkit/wp/?page_id=294 .

In the source code available for this article, the *NyARToolkit* folder already contains the *NyARToolkitCPP* version. Integration is done by adding the *NyARToolkitCPP.pri* file to the project:

```
## NyARToolkitCPP.pri file

NYARTOOLKIT_SRC_DIR = $$PWD/NyARToolkitCPP

INCLUDEPATH += \
    $$NYARTOOLKIT_SRC_DIR/inc \
    $$NYARTOOLKIT_SRC_DIR/inc/core \
    $$NYARTOOLKIT_SRC_DIR/inc/core2 \
    $$NYARTOOLKIT_SRC_DIR/inc/nyidmarker \
    $$NYARTOOLKIT_SRC_DIR/inc/processor \
    $$NYARTOOLKIT_SRC_DIR/inc/utils

HEADERS += \
    $$NYARTOOLKIT_SRC_DIR/inc/NyAR_core.h \
    $$NYARTOOLKIT_SRC_DIR/inc/NyAR_utils.h \
    ...
    $$NYARTOOLKIT_SRC_DIR/inc/utils/NyAR_utils.h \
    $$NYARTOOLKIT_SRC_DIR/inc/utils/NyStdLib.h

SOURCES += \
    $$NYARTOOLKIT_SRC_DIR/src/core/INyARTransportVectorSolver.cpp \
    ...
    $$NYARTOOLKIT_SRC_DIR/src/processor/SingleARMarkerProcessor.cpp \
    $$NYARTOOLKIT_SRC_DIR/src/utils/NyStdLib.cpp
```

Create a Custom Camera QML item

This is the QML element which will display the contents of the Camera.

```
#include <QVideoFrame>

class FrameObserver
{
public:
    virtual bool updateItem(const QVideoFrame &frame) = 0;
};

class CustomCamera : public QDeclarativeItem, public FrameObserver
{
    Q_OBJECT
public:
    ...
    virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                       QWidget *widget);

    virtual bool updateItem(const QVideoFrame &frame);
    ...
};
```

This class inherits from `FrameObserver` to make available the method `updateItem(QVideoFrame)`. The `MyVideoSurface` class uses this method to acquire the camera frames and pass it to the `CustomCamera` item.

```
class MyVideoSurface: public QAbstractVideoSurface
{
    Q_OBJECT
public:
    MyVideoSurface(FrameObserver *target, QObject *parent = 0);
    ...
    virtual bool present(const QVideoFrame &frame);
    ...
};

bool MyVideoSurface::present(const QVideoFrame &frame)
{
    return m_target->updateItem(frame);
}
```

As shown on the code above, `MyVideoSurface` repasses the `QVideoFrame` object to the `FrameObserver` which is the `CustomCamera` item.

The `QVideoFrame` is turned into a `QImage` to be displayed on the `CustomCamera::paint()` method.

One thing to note: on Meego Harmattan the frames come in a different color space (**UYVY**) and need to be converted tp (**RGB32**) for later processing, as done below.

```
bool CustomCamera::updateItem(const QVideoFrame &frame)
{
    m_frame = frame;

    if (m_frame.map(QAbstractVideoBuffer::ReadOnly)) {
        if (m_frame.pixelFormat() == QVideoFrame::Format_UYVY) {
            // Converting from UYVY colorspace to RGB32 colorspace
            m_yuv2rgb.convert(m_frame.bits(), m_frame.width(), m_frame.height());

            m_targetImage = QImage(m_yuv2rgb.bits(), m_yuv2rgb.width(),
                                   m_yuv2rgb.height(), m_yuv2rgb.width() * 4,
                                   QImage::Format_RGB32);
        } else if (m_frame.pixelFormat() == QVideoFrame::Format_RGB32) {
            m_targetImage = QImage(m_frame.bits(), m_frame.width(),
                                   m_frame.height(), m_frame.bytesPerLine(),
                                   QImage::Format_RGB32);
        } else {
            qDebug("Unhandled format: ARGB32");
        }

        update();

        m_imageRect = m_targetImage.rect();

        m_frame.unmap();
    }

    return true;
}
```

This way, the `CustomCamera` provides a "video display" item to show the camera contents on the screen, while retaining each frame (`m_targetImage`) to later generate AR information. That's where the `Ar` class comes in.

Create an AR class to process the camera frames

The **AR class** contains all the code needed from the **NyARToolkitCPP** library. It's used to process the frames. Each frame from the `CustomCamera` item has an array of `chars` (`char *`) which is used by the **AR class** to generate a 3D transformation matrix. This matrix will later be used by the 3D viewport to adjust the 3D world.

```
...
#include <NyAR_core.h>

using namespace NyARToolkitCPP;

class Ar : public QObject
{
    Q_OBJECT
public:
    Ar(QObject *parent = 0);
    ~Ar();

    void initialize(int width, int height);
    bool detectMarker(char *buffer, int bytesPerLine);
    void getTransformationMatrix(qreal *matrix);

private:
    void destroy();

    NyARParam m_parameters;
    NyARCode *m_code;
    NyARRgbRaster_BGRA  *m_aRRgbRaster;
    NyARSingleDetectMarker  *m_detectMarker;
    NyARTransMatResult  m_matrix;
    qreal m_markerWidth;
};
```

There are three methods in this class that will do most of the work:

- `initialize(int width, int height)`

  Setup the size of the frame to be processed and loads two files that specify the camera parameters (**camera.dat**) and the AR pattern (**patt.hiro**). This method is only called once, when the `MyVideoSurface` class starts receiving frames from the camera itself.
- `detectMarker(char *buffer, int bytesPerLine)`

  The frame from `CustomCamera` is passed to this method as an *array of chars* (along with its line size) and this is where the transformation matrix is created. The pattern specification previously loaded (**patt.hiro**) is used here.
- `getTransformationMatrix(qreal *matrix)`

  The matrix now can be retrieved as an `array of float` using this method.

Below are the methods definition:

```
void Ar::initialize(int width, int height)
{
    ...
    // Camera parameters file : "camera.dat"
    QString cameraFile = CAMERAFILE;
    // Pattern file : "patt.hiro"
    QString codeFile = CODEFILE;
    ...
    m_parameters.loadARParamFromFile(cameraFile.toLatin1().constData());
    ...
    m_code->loadARPattFromFile(codeFile.toLatin1().constData());

    m_aRRgbRaster = new NyARRgbRaster_BGRA(width, height);
    m_parameters.changeScreenSize(width, height);
    m_detectMarker = new NyARSingleDetectMarker(&m_parameters, m_code,
                                                m_markerWidth,
                                                m_aRRgbRaster->getBufferType());
    ...
}

bool Ar::detectMarker(char *buffer, int bytesPerLine)
{
    bool markerExists = false;

    ...

    markerExists = m_detectMarker->detectMarkerLite(*m_aRRgbRaster, 100);
```

```
    if (markerExists) {
        if (m_detectMarker->getConfidence() < 0.5) {
            markerExists = false;
        } else {
            m_detectMarker->getTransmationMatrix(m_matrix);
        }
    }

    return markerExists;
}

void Ar::getTransformationMatrix(qreal *matrix)
{
    qreal worldScale = 0.025;

    matrix[0 + 0 * 4] = m_matrix.m00;
    matrix[0 + 1 * 4] = m_matrix.m01;
    matrix[0 + 2 * 4] = m_matrix.m02;
    matrix[0 + 3 * 4] = m_matrix.m03;
    matrix[1 + 0 * 4] = -m_matrix.m10;
    matrix[1 + 1 * 4] = -m_matrix.m11;
    matrix[1 + 2 * 4] = -m_matrix.m12;
    matrix[1 + 3 * 4] = -m_matrix.m13;
    matrix[2 + 0 * 4] = -m_matrix.m20;
    matrix[2 + 1 * 4] = -m_matrix.m21;
    matrix[2 + 2 * 4] = -m_matrix.m22;
    matrix[2 + 3 * 4] = -m_matrix.m23;
    matrix[3 + 0 * 4] = 0.0;
    matrix[3 + 1 * 4] = 0.0;
    matrix[3 + 2 * 4] = 0.0;
    matrix[3 + 3 * 4] = 1.0;

    if (worldScale != 0.0) {
        matrix[12] *= worldScale;
        matrix[13] *= worldScale;
        matrix[14] *= worldScale;
    }
}
```

These methods are used on the `CustomCamera` item as follows:

```
bool CustomCamera::updateItem(const QVideoFrame &frame)
{
    ...
    if (m_frame.map(QAbstractVideoBuffer::ReadOnly)) {
        ...

        m_imageRect = m_targetImage.rect();

        // Get the new AR transformation matrix
        calculateArMatrix();

        m_frame.unmap();
    }

    return true;
}
...
void CustomCamera::calculateArMatrix()
{
    QMutexLocker mutex(&m_mutex);

    bool isMarkerDetected = false;

    isMarkerDetected = m_ar->detectMarker((char *) m_targetImage.bits(),
                                          m_targetImage.bytesPerLine());

    if (isMarkerDetected) {
        // Stores the new AR transformation matrix
        m_ar->getTransformationMatrix(m_arMatrix.data());
    } else {
        m_arMatrix.setToIdentity();
    }

    emit arMatrixChanged();
}
```

Every new frame from the camera is passed to the `Ar` class and on the `calculateMatrix()` method a marker (pattern) detection occurs. If a marker is found on the frame, then the new transformation matrix is stored on a `QMatrix4x4` object (`m_arMatrix`). If not, then the matrix is cleared (set to an "Identity Matrix").

Create a Custom 3D QML item

Now it's time to use QtQuick3D code. In order to create a 3D item and use it on QML, the QDeclarativeItem3D class is used. It's the same as a `QDeclarativeItem`, except that it draws OpenGL objects. Therefore the drawing routines are different:

```
...
class CustomItem3D : public QDeclarativeItem3D
{
    Q_OBJECT
    Q_PROPERTY(QMatrix4x4 arMatrix READ arMatrix WRITE setArMatrix)
public:
    CustomItem3D(QObject *parent = 0);
    ~CustomItem3D();

    void initialize(QGLPainter *painter);
    void draw(QGLPainter *painter);

    void setArMatrix(const QMatrix4x4 &matrix);
    QMatrix4x4 arMatrix() const;

private:
    void updateModel();

    QGLAbstractScene *m_scene;
    QGLSceneNode *m_node;
    QMatrix4x4 m_arMatrix;
};
```

The method for drawing here is the `draw(QGLPainter *)`. The `initialize(QGLPainter *)` method is called before the object is first drawn on the screen. The 3D object can be of any desired shape (a cube, a sphere, the Utah teapot, etc), but, for this example a 3D mesh model is used (**simplemodel.obj**).

```
void CustomItem3D::initialize(QGLPainter *painter)
{
    Q_UNUSED(painter)

    updateModel();
}

// TODO: Move to a QThread
void CustomItem3D::updateModel()
{
    QString filename;

    if (m_scene)
        delete m_scene;

#if defined(Q_WS_S60)
    filename = "c:\\data\\qtar\\3dmodel\\simplemodel.obj";
#elif defined(Q_OS_HARMATTAN)
    filename = RESOURCES_DIR "/3dmodel/simplemodel.obj";
#else
    filename = "./3dmodel/simplemodel.obj";
#endif

    m_scene = QGLAbstractScene::loadScene(filename);

    m_node = m_scene->mainNode();
}

void CustomItem3D::draw(QGLPainter *painter)
{
    // Apply AR transformation matrix
    painter->modelViewMatrix() = m_arMatrix;

    // Rotate the 3D Model UP (Facing the camera)
    painter->modelViewMatrix().rotate(90.0, 1.0, 0.0, 0.0);

    m_node->draw(painter);
}

// Updates the AR transformation matrix
void CustomItem3D::setArMatrix(const QMatrix4x4 &matrix)
{
    if (m_arMatrix != matrix) {
        m_arMatrix = matrix;

        update();
    }
}

QMatrix4x4 CustomItem3D::arMatrix() const
{
    return m_arMatrix;
}
```

The 3D model (stored on `m_node`) is drawn on the `draw()` method only after the "AR matrix" is applied to the 3D world. Every time a new camera frame is processed, the 3D model appears on a different position and with a different size. Therefore, giving the impression of a 3D object appearing on a real world.

Display Camera Item + 3D Item contents on the screen

Now that both items (`CustomCamera` and `CustomItem3D`) are created, we export them to the QML context:

```
qmlRegisterType<CustomCamera>("Widgets", 1, 0, "CustomCamera");
qmlRegisterType<CustomItem3D>("Widgets", 1, 0, "CustomItem3D");
```

And after that, they are ready to be displayed:

```
## Mainview.qml

import QtQuick 1.1
import Qt3D 1.0
import Widgets 1.0

Item {
    id: main

    Rectangle {
        id: background
        ...
        Component.onCompleted: {
            viewportLoader.sourceComponent = viewportComponent;
        }
    }

    Loader {
        id: cameraLoader
        ...
    }

    Component {
        id: cameraComponent

        // Camera : Drawn on the Background
        CustomCamera {
            id: customCamera

            onFrameSizeChanged: {
                viewportLoader.item.width = customCamera.frameSize.width;
                viewportLoader.item.height = customCamera.frameSize.height;
            }

            Component.onCompleted: {
```

```
                customCamera.start();
            }
        }
    }

    Loader {
        id: viewportLoader
        ...
        anchors.centerIn: cameraLoader
    }

    Component {
        id: viewportComponent

        // 3D Viewport: Drawn on the Foreground
        Viewport {
            id: viewport

            navigation: false

            camera: Camera {
                fieldOfView: 45
                nearPlane: 1
                farPlane: 500
            }

            // Added the AR 3D Item
            // It changes according to the AR matrix parameter from the camera
            CustomItem3D {
                id: item3D

                arMatrix: cameraLoader.item.arMatrix

                Component.onCompleted: {
                    cameraLoader.sourceComponent = cameraComponent;
                }
            }
        }
    }
    ...
}
```

The `CustomCamera` item is drawn *before* the 3D Viewport which contains the `CustomItem3D`. This way, the frames appear as a video on background. while the 3D item is shown on the foreground. The `Loaders` are just for delaying the creation of each item and try to allocate more memory for the application.

Notice that there's a binding of the `arMatrix` property from the `CustomItem3D` with `CustomCamera::arMatrix` property. This causes the 3D item to be updated (scaled and positioned properly on the scene) every time a marker is detected.

Screenshots

Application running on Nokia N9

Application running on Nokia N9

Application running on Nokia C7-00

Files

- Application source code: File:Qtar-source-code.tar.gz
- QtQuick3D package files: File:Libqt4-3d 1.0-rc armel.deb.tar.gz (Meego Harmattan) and File:Qt Quick 3D.sis (Symbian^3)
- Application packages: File:Qtar 0.0.1 armel.deb.tar.gz (Meego Harmattan) and File:Qtar.sis (Symbian^3)
- The "Pattern File" used for this application can be found inside the NyARToolkit dir: **qtar/NyARToolkit/NyARToolkitCPP/data/pattHiro.pdf**