camera in practice

Meetup #4 : Caméra

Développement mobile avec Qt/QML
La Cantine Numérique le 29th of February 2016

Guillaume Charbonnier
gcharbonnier@a-team.fr

Workshop objectives

Create a simple QML2 application :

- basic management of QML Camera element

  (take a picture, flash, zoom)
- create custom type in QML
- create custom type in C++
- example : QrCode reader with QZxing

demo project can be found on gitHub : https://github.com/a-team-fr/MeetupMobileQtQml/tree/master/160229/DemoProject

Steps

1. Create a project - (time box : 10mn)
2. Add an overlay panel - (time box : 10mn)
3. Take a photo - (time box : 10mn)
4. Manage zoom and flash - (time box : 15mn)
5. Add a C++ controler - (time box : 30mn)

Step 1

Project creation

- Create a new project
- Multimedia module activation
- Add a window, a camera and a viewfinder

Multimedia module activation

1. Create a new application project (Qt Quick Application)
2. Modify .pro file to activate module(s)

```
QT += qml quick multimedia
```

3. Modify main.qml file to load module(s)

```
import QtMultimedia 5.5
import QtQuick.Controls 1.4
```

Once this steps are done, the following QML types are available :
Audio, **MediaPlayer**, **Radio**, **Video** and of course... **Camera** !

The multimedia types we are interested in...

- **Camera** : get a frame, take a photo or a video
- **VideoOutput** : display the camera frames (viewfinder)
- **QtMultimedia** : a global object storing useful informations

- some additional types will be used to tune the camera :
  - **CameraCapture** : take a picture
  - **CameraRecorder** : take a movie
  - **CameraExposure** : tune exposure
  - **CameraFlash** : handle flash
  - **CameraFocus** : focus management
  - **CameraImageProcessing** : capture settings

Ajoutez une caméra et un viewfinder

Note :

The **Camera** is not vshowing anything. We need **VideoOutput** to display the Camera frames

The **VideoOutput** geometry can be defined as any Item based object (in this example, it is of the main window size).

and we define the camera to be the source of the viewfinder

```
import QtQuick 2.3
import QtQuick.Window 2.2
import QtMultimedia 5.5

Window {
  visible: true
  width:1024
  height:768

  Camera{
      id:camera
  }
  VideoOutput{
      id:viewfinder
      source:camera
      anchors.fill: parent
  }

}
```

That’s all...thanks and see you at the next meetup!

some typical usecases

- select a camera
- viewfinder transformation (rotate, scale)

Before going any further let’s review some useful tips to manage these typical cases...

```
...
Camera{
  id:camera
}

ListView {
    anchors.fill: parent

    model: QtMultimedia.availableCameras
    delegate: Text {
      text: modelData.displayName

      MouseArea {
        anchors.fill: parent
        onClicked: camera.deviceId = modelData.deviceId
      }
    }
  }
```

on mobile plate-form, one can simply select between back or front camera :

```
Camera{
    id:camera
    position: Camera.BackFace
    //position:Camera.FrontFace
  }
```

List the available camera

```
VideoOutput{
    id:viewfinder
    source:camera
    anchors.fill: parent
    orientation:90
    autoOrientation:false
    fillMode: VideoOutput.PreserveAspectFit
//autres mode : Stretch, PreserveAspectCrop
    scale: height/width
    rotation : 12
    transformOrigin: Item.Center
 }
```

Orientation : 90 deg step - one can also use the device orientation : *camera*.orientation.
Use autoOrientation to synchronize the orientiation with UI (portrait vs landscape).

Tips : use *Qt*.platform.os if you need to perform OS specifics things

As the **VideoOutput** type is a visibletype, it is inheritate the **Item** properties :
scale, rotation (in deg)... use transformOrigin

One can also use a sensor (don't forget to activate "sensors" module) :

```
OrientationSensor{
    active:true
    if (reading.orientation === OrientationReading.LeftUp)
        {...}
}
```

Viewfinder transformation (rotate, scale)

Step 2

Add a remote panel

...this will also be the opprtunity to see how QML can be extended with custom types ;-)

- Add an overlay panel
- Add a button to take a picture
- Add a flash mode selector

if you want to create your own QML type (recommanded for organizing your code and build your own library), you need to:

- create a new QML document **starting with an UPPER case letter**
- to ease deploying, add the QML document to your ressource file (QRC)

It is then available into another QML document as long as :

- the new QML document is with the same directory than your new QML document custom type
- use **import** if your custom type QML document is within another directory

/Main.qml

```
import "./myLibs"

MonSuperNouveauType{
    isSomethingImportant : true
    anchors.fill : parent
}
```

/myLibs/MonSuperNouveauType.qml

```
import QtQuick 2.0

Item {
  property bool isSomethingImportant : true
  property alias text : label.text
  Text{
    id:label
    color: parent.isSomethingImportant ? "red" : "black"
  }
}
```

Overlay.qml

```
import QtQuick 2.0
import QtMultimedia 5.5

Item {
  id:root
  property color backgroundColor : "darkblue"
  signal takePhoto()
  property int flashMode : Camera.FlashAuto


  Rectangle{
    id:background
    anchors.fill: parent
    color : parent.backgroundColor
    opacity : 0.4; radius:5
  }
  Row{}
}
```

```
Row{
    Image{
      source:"qrc:/photo.png"
      MouseArea{  anchors.fill: parent;  onClicked: root.takePhoto(); }
    }
    Image{
      id:flashMode
      source: root.flashMode === Camera.FlashAuto ?
              "qrc:/camera_flash_auto.png" : "qrc:/camera_flash_off.png"
      MouseArea{
        anchors.fill: parent; onClicked: {
          if (root.flashMode === Camera.FlashAuto)
              root.flashMode = Camera.FlashOff;
          else root.flashMode = Camera.FlashAuto;
        }
      }
    }
  }
```

exercice : create an overlay panel

Main.qml

```
Overlay{
    //our control panel as custom type
    anchors.bottom: parent.bottom
    anchors.horizontalCenter: parent.horizontalCenter
    width : 100; height:50
    onTakePhoto: console.log("takePhoto")
    onFlashModeChanged: console.log("FlashMode:"+flashMode)
 }
```

Notes :

Any QML property change are notified with a signal in this form : <NomPropriété>Changed()

Questions :

- *The content of the overly panel (inside **Row**) is not a child of Background (**Rectangle**)...what was the reason ?*

- *what are the difference between :*

```
Item{
    //my great stuff
  }
```

```
Rectangle{
        //my great stuff
        color : "transparent"
  }
```

- *what are the difference between* visible:false *and* *opacity:0*

Comments

Answers :

- *The background is transparent but we want the Row content to be opaque !*
- *Any Rectangle pixel gets rendered in OpenGL, this could lead to poor performance on some plate-forms*
- *A not visible item won't be part of OpenGL scenegraph and thus won't be active at all whereas a fully transparent item behaves normally even it cannot be seen..*

Step 3

take a photo

- Add an Image to preview the photo
- take a picture

```
Image{
    id:previewImage
    anchors.fill: parent
    fillMode: Image.PreserveAspectFit
    visible: false
    MouseArea{
        anchors.fill: parent; onClicked:parent.visible = false
    }
}
```

Add preview

```
Overlay{
    onTakePhoto: camera.imageCapture.capture();
 }
```

```
Camera{
    id:camera
    imageCapture {
        onImageCaptured: {
            previewImage.source = preview;
            previewImage.visible = true;
        }
        onImageSaved: console.log("picture saved to :"+path)
    }
 }
```

To take a photo, we modify our existing overlay and camera elements

using ***imageCapture*** (of the type **CameraCapture**), one can :

- know the path of the last image captured using ***capturedImagePath*** (i.o using ***onImageSaved***() slot)
- define capture resolution using...***resolution***
- test if the camera is ready using the property : ***ready***
- save metadata (some usefull data such as GPS coordinate, portrait/landscape ...are available in the **Camera metaData** property group) using **setMetaDataKey** method.
- modify the capture path using **captureToLocation**( ) i.o **capture**()

In a similar manner, one can record a movie using the camera *videoRecorder (of the type* **CameraRecorder**) calling **record**() and **stop**() methods.

Don’t forget to set the camera to the relevant mode using camera.captureMode = Camera.CaptureVideo

To display the captured movie, you can use a **CameraPlayer** (i.o **Image** for a still image) filled through a **MediaPlayer** which source would be set to the camera **videoRecorder** using **camera.videoRecorder.actualLocation**.

Step 4

Manage Zoom and flash

- manage flash
- manage zoom

we only need to modify our Camera to use the desired flash mode we already provide through the overlay panel :

```
Camera{
    //our Camera to play with
    id:camera
    flash.mode: overlay.flashMode
..
  }
```

```
PinchArea{
        anchors.fill: parent
        enabled: true
        pinch.minimumScale: 1
        pinch.maximumScale: camera.maximumDigitalZoom
        scale: camera.digitalZoom
        onPinchStarted: {
            scale = camera.digitalZoom;
            zoom.visible = true;
        }
        onPinchFinished: zoom.visible = false;
        onPinchUpdated: {
            camera.digitalZoom = pinch.scale;
        }
    }
```

gérer le zoom

Step 5

Add a C++ controler

...this will be the opportunity to see how to extend QML with C++

- create a “QML compliant” C++ class
- make a C++ object available as a new QML element (or a C++ class as a new QML type)
- define properties, signals...

```
#ifndef CAMERACONTROLER_H
#define CAMERACONTROLER_H
#include <QObject>

class CameraControler : public QObject
{
  Q_OBJECT
public:
  explicit CameraControler(QObject *parent = 0);

signals:

public slots:
};
#endif // CAMERACONTROLER_H
```

QObject inheritance

use Q_OBJECT macro

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "cameracontroler.h"
#include <QQmlContext>
int main(int argc, char *argv[])
{
  QGuiApplication app(argc, argv);

  QQmlApplicationEngine engine;
  CameraControler theOneAndOnlyCC;
  engine.rootContext()->setContextProperty("controler", &theOneAndOnlyCC);
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

  return app.exec();
}
```

we create our CameraControler object “theOneAndOnlyCC” and add it to the QML context as “controler”

Add to the QML context : modify main.cpp

If we would have needed several objects of that kind, we could have registered the CameraControler class as a new QML type. This would enable us to create new QML element of the CameraControler type directly in QML.

This is how to proceed for registering a new type :

```
qmlRegisterType<CameraControler>("Controler", 1, 0, "Controler");
```

cameracontroler.h

```
class CameraControler : public QObject
{
  Q_OBJECT
  Q_PROPERTY(QString greeting READ getGreeting NOTIFY greetingChanged)

public:
  explicit CameraControler(QObject *parent = 0);
  QString getGreeting(){
          return "Hi, I am "+name;
  }
  Q_INVOKABLE void setName(QString newName){
           name = newName; emit greetingChanged();
  }

signals:
  void greetingChanged();

private:
  QString name="the Big controler";
};
```

main.qml

```
Text{
    anchors.centerIn: parent
    width:200; height:50
    text : controler.greeting
    color:"red"
    MouseArea{
      anchors.fill: parent
      onClicked: controler.setName("the TINY
controler")
    }
  }
```

Extend the QML interface : define properties, signals and invokable methods

Guillaume Charbonnier
gcharbonnier@a-team.fr