QmlPaint - how to make paint application with QML

This article explains how to create paint application with QML

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

Introduction

While QML offers easy way to show images on screen and put those to everywhere, it still lacks of possibility to edit image itself. All imaging is only about to display, not any chance to modify or even save. But when mixing Qt, it is possible to edit image. Also from Qt we can draw on to the display.

Derive new component from QDeclarative to make it accessible from QML. Why would not do all with Qt, answer is simple; QML offer much better and nicer ways to show data to user. Let's imagine paint application, device screen itself is small and eventually drawing small pictures is not so fun. If we create item to handle drawing, put it to Flickable component and we get more space. And more, afterall image to be show'd is QPixmap. No need to be blank, that can be anything from Gallery. Saving is also possible, Qt images already provide functions for that.

Paint application also needs undo. Idea is quite simple, save drawed points so those can be popped out from image.

This item is not derived from QDeclarativeImage or any other image class, it need to implement some of the functions by itself. Drawing and scaling are the most important. Reason for this is the QDeclarativeImage does not offer way to modify or access directly to image data itself.

Whole example can be founded from projects: http://projects.developer.nokia.com/QmlPaintExample

Creating canvas and loading image

Canvas item with undo functionality must be created with two images, one for display and one for backup. Without scaling of displayed image, backup image is not necessary. But when using scaling, like using canvas in Flickable, it is too painful to figure out where is finger pointing and what could be image and display scaling.

Creating empty image and loading from image differs slightly. As creating image in the example, is creating first the displayed image and then backup which is also image used to saving. But in loading method, image is first loaded and then put on to display image. Reason here were to keep displayed image as long as we can, if in image reading occurs error the application won't erase ongoing work.

Loading image

Canvas item images are plain images. Load image to backup, create visible pixmap from backup image and display it. When implementing file handling methods which are visible to QML, Qt uses file paths but QML uses URL. Convenient way to change URL to file path and vice versa is to use QUrl class toLocalFile/fromLocalFile methods.

```
void ImageCanvas::setUnderneathImage(QUrl filename)
{
    m_underneathImageFilename=filename;
    QFile file(m_underneathImageFilename.toLocalFile());
    if(file.open(QIODevice::ReadOnly)){
        QScopedPointer<QImageReader> reader(new QImageReader(&file));
        m_sourceImage = reader->read();

        m_penPoints.clear();
        m_penPointsScreen.clear();

        if(reader->error()|| m_sourceImage.isNull()){
            m_error = LoadError;
            m_errorString = reader->errorString();
            qDebug() << "error: " << reader->error() << " str: " << reader->errorString();
            emit error();
        }else{
            m_editImage = QPixmap::fromImage(m_sourceImage);
            m_sourceSize = m_editImage.size();
            m_originalSourceSize = m_sourceImage.size();
            this->setImplicitHeight(m_editImage.height());
            this->setImplicitWidth(m_editImage.width());
            this->update();
            m_error = NoError;
            emit sourceSizeChanged();
            emit originalSourceSizeChanged();
        }
    }else{
        m_error = LoadError;
        m_errorString = file.errorString();
    }
    emit underneathImageChanged();
}
```

Create empty image

Creating empty image can be assigned instantly to image to be shown. Backup image is created from this image. This is faster way to show image in UI, also creating plain empty image is faster than loading image what makes this preferred approach. And like in loading image, backup image is created.

```
void ImageCanvas::createEmpty(int width,int height)
{
    if(width>0&&height>0){
        m_editImage = QPixmap(width,height);
        if(!m_editImage.isNull()){
            m_editImage.fill();
            m_sourceSize = m_editImage.size();
            m_originalSourceSize = m_sourceSize;
            this->setImplicitHeight(m_editImage.height());
            this->setImplicitWidth(m_editImage.width());
            this->update();
            m_error = NoError;
            m_sourceImage = m_editImage.toImage();
            emit sourceSizeChanged();
            emit originalSourceSizeChanged();
        }else{
            m_error = CreateError;
            emit error();
        }
    }
}
```

Scaling

Creating bigger image than screen resolution is, scaling will be important part. While this generates also difficulties, when mouse event reports X/Y values, those are mapped to screen resolution itself. Therefore it is needed to calculate new mouse points, scale visible graphic surface and of course update drawings.

First, using properties for scaling makes it easy access from QML. As QML is mostly informing new scaling resolutions to image. Keep also original sizes, it comes handy when showing image in original size.

```
Q_PROPERTY(QSize sourceSize READ sourceSize WRITE setSourceSize NOTIFY sourceSizeChanged)
Q_PROPERTY(QSize originalSourceSize READ originalSourceSize WRITE setOriginalSourceSize NOTIFY originalSourceSizeChanged)
```

As these properties are only variables, setters for those does not do any magic actions yet. Now look for visible image updating which is **paint(QPainter* painter, const QStyleOptionGraphicsItem* styleOption, QWidget* widget)** method from derived QDeclarativeItem class. Making things easy, canvas item also knows it's width and height and these are coming from QDeclarativeItem. QML is handling width and height reporting. In updating, check is width and height matching with image size. When image wanted resolution is same as screen resolution, don't do scaling, otherwise do scaling. QPainter::drawPixmap will handle drawing, only important in scaled drawing is to calculate image's rectangle right and this is done already in QML side with PinchArea.

```
onPinchFinished: {
                var setHeight = flickable.contentHeight;
                var setWidth = flickable.contentWidth;
                if(flickable.contentHeight>canvas.originalSourceSize.height)
                    setHeight = canvas.originalSourceSize.height;
                if(flickable.contentWidth>canvas.originalSourceSize.width)
                    setWidth = canvas.originalSourceSize.width;

                flickable.resizeContent(setWidth,setHeight,pinch.center);
                canvas.sourceSize = Qt.size(setWidth,setHeight);
                flickable.returnToBounds()
            }
```

And then updating UI, here will be determined do need draw scaled or not scaled:

```
void ImageCanvas::paint(QPainter* painter, const QStyleOptionGraphicsItem* styleOption, QWidget* widget)
{
    qDebug() << __PRETTY_FUNCTION__ << " image: " << m_editImage.isNull() << " save: " << m_saveInProgress;
    if(m_editImage.isNull())
        return;

    if(m_saveInProgress)
        return;

    bool aa = painter->testRenderHint(QPainter::Antialiasing);

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

    if(width()!=m_editImage.width() || height()!=m_editImage.height()){
        painter->drawPixmap(QRectF(this->x(),this->y(),this->width(),this->height()),
                            m_editImage,
                            m_editImage.rect());
    }else{
        painter->drawPixmap(QPoint(0,0),m_editImage);
    }

    if(smooth()){
        painter->setRenderHint(QPainter::Antialiasing,aa);
    }
}
```

Note that you don't need to take care of drawn points or anything what have been drawn on canvas. Drawn points are stored in list containers but points itself are actually drawn directly to image data. So points become static with image.

Drawing

Example takes care of two different drawing styles; single points and lines. All draw point data is stored in two QList's, one for displayed image and and with original image resolution. Original image resolution is not changed but other list keeps inside scaled image points which resolution can be changed.

Undo function is also dependant of history of drawed points. If keep only one storage for drawn points it would be possible to take those out from screen image but not from image to be saved. Also, editable image meaning the one to be displayed could be scaled and so are the points also. Both lists are following own image resolutions and cannot be mixed together.

Single point drawing is little bit easier, take mouse event point and put coordinates to point. But when drawing line, there is one and few other issues. First is line need starting point. This could be single point on screen and then appending new point to list with this single point as starting and other screen tap to ending point. Or do not draw anything before second tap, line ending coordinates is there. Now if want line to follow last line ending, look for last point from list. Currently last ones ending is new starting.

Drawing needs helper class to keep on track of points or anykind of shapes what to draw, if points would be single points the helper class is not needed. For other shapes, lines for example, helper class is there to help.

Prepare point

```
class CanvasPoint
{
public:
    CanvasPoint(){}

    QPen pen;
    QPointF start;
    QPointF end;
    int type;
};
```

Then we need get point from QML. The point where finger is pointing. Get only when position is changed, if following separately X/Y axis changing leads to situation where QML is informing all the time new points and points get to drawn on places where not wanted.

```
MouseArea{
  anchors.fill: parent
  onMousePositionChanged: {
    if(canvas.penEnabled){
      canvas.penPosition = Qt.point(mouse.x,mouse.y);
      mouse.accepted = true;
    }
  }
}
```

Set point to Qt side, call update image:

```
Q_PROPERTY(QPoint penPosition READ penPosition WRITE setPenPosition NOTIFY penPositionChanged)

QPoint penPosition(){return QPoint(m_penX,m_penY);}
void setPenPosition(QPoint point){m_penX=point.x();m_penY=point.y(); emit penPositionChanged(); updateImage();}
```

Updating image

Mouse event reports resolution points where it is. This means; if screen width is 360 pixels, mouse event never report over that. Lets imagine flickable image has resolution of 1024 by 768 and screen resolution is 360 by 480. Now if move flickable to show image from 500 by 500, mouse event still report X and Y based on screen resolution and not to image resolution. If image is same size as screen resolution, no problem at all. But when using bigger sized images, mouse event points need to be calculated to match into image. Map mouse event point to image, calculate X and Y scale ratios from shown image divided by declarative item. Then same for backup image, note that same ratios cannot be used as backup is not scaled in any point but shown image can be.

```
void ImageCanvas::updateImage()
{
    qDebug() << __PRETTY_FUNCTION__;
    if(m_penEnabled){
        QPen pen(m_color,m_penWidth,Qt::SolidLine,m_penCapStyle,m_penJoinStyle);

        QScopedPointer<QPainter> painter(new QPainter());

        painter->begin(&m_editImage);
        if(smooth()){
            painter->setRenderHint(QPainter::Antialiasing,true);
        }

        qreal scaleX = m_editImage.width()/this->width();
        qreal scaleY = m_editImage.height()/this->height();

        // store point also with original size,
        qreal storeScaleX = m_originalSourceSize.width()/this->width();
        qreal storeScaleY = m_originalSourceSize.height()/this->height();

        QPointF point;
        if(m_editImage.size()!=m_sourceSize){
            point = QPointF(m_penX*scaleX,m_penY*scaleY);
        }else{
            point = QPointF(m_penX,m_penY);
        }

        CanvasPoint cpoint;
        cpoint.type = m_drawType;
        painter->setPen(pen);


        switch(m_drawType){
        case DrawPen:{
            cpoint.pen  = pen;
            cpoint.start = point;
            m_penPointsScreen.append(cpoint);
            painter->drawPoint(point);

            CanvasPoint cstore;
            cstore.pen = pen;
            cstore.start = QPointF(m_penX*storeScaleX,m_penY*storeScaleY);
            cstore.type = m_drawType;
            m_penPoints.append(cstore);
        }break;
        case DrawLine:{
            if(m_penPointsScreen.isEmpty()){
                cpoint.pen  = pen;
                cpoint.start = point;
                m_penPointsScreen.append(cpoint);

                CanvasPoint cstore;
                cstore.pen = pen;
                cstore.start = QPointF(m_penX*storeScaleX,m_penY*storeScaleY);
                cstore.type = m_drawType;
                m_penPoints.append(cstore);
            }else{
                if(m_penPointsScreen.last().type==DrawLine){
                    if(m_penPointsScreen.last().end.isNull()){
                        m_penPointsScreen.last().end = point;
                        painter->drawLine(m_penPointsScreen.last().start,m_penPointsScreen.last().end);
                        m_penPoints.last().end = QPointF(m_penX*storeScaleX,m_penY*storeScaleY);
                    }else{
                        cpoint.start = m_penPointsScreen.last().end;
                        cpoint.end = point;
                        cpoint.pen = pen;
                        m_penPointsScreen.append(cpoint);
                        painter->drawLine(m_penPointsScreen.last().start,m_penPointsScreen.last().end);

                        CanvasPoint cstore;
                        cstore.pen = pen;
                        cstore.start = m_penPoints.last().end;
                        cstore.end = QPointF(m_penX*storeScaleX,m_penY*storeScaleY);
                        cstore.type = m_drawType;
                        m_penPoints.append(cstore);
                    }
                }
                if(m_penPointsScreen.last().type==DrawPen){
                    cpoint.start = m_penPointsScreen.last().start;
                    cpoint.end = point;
                    cpoint.pen = pen;
                    m_penPointsScreen.append(cpoint);
                    painter->drawLine(m_penPointsScreen.last().start,m_penPointsScreen.last().end);

                    CanvasPoint cstore;
                    cstore.pen = pen;
                    cstore.start = m_penPoints.last().start;
                    cstore.end = QPointF(m_penX*storeScaleX,m_penY*storeScaleY);
                    cstore.type = m_drawType;
                    m_penPoints.append(cstore);
                }
            }
        }break;
        default:break;
        }

        painter->end();
    }
    update();
}
```

Undo

Undo is like reversed image updating. Drawn points which are stored to QList containers, makes it easy to remove drawing from image. Note that points are removed from list containers, new visual image is created from backup without scaling. Then draw points, scale image and update to screen.

```
void ImageCanvas::undo()
{
    if(!m_sourceImage.isNull() && !m_penPoints.isEmpty()){
        m_saveInProgress = true;

        QSize size = m_editImage.size();
        m_editImage = QPixmap::fromImage(m_sourceImage);
        m_penPoints.takeLast();
        m_penPointsScreen.takeLast();
        QScopedPointer<QPainter> p(new QPainter());
        p->begin(&m_editImage);
        for(int i=0;i<m_penPoints.count();i++){
            CanvasPoint cpoint = m_penPoints.at(i);
            p->setPen(cpoint.pen);
            switch(cpoint.type){
            case DrawPen:{
                p->drawPoint(cpoint.start);
            }break;
            case DrawLine:{
                p->drawLine(cpoint.start,cpoint.end);
            }break;
            default:break;
            }
        }
        p->end();
        if(size!=m_editImage.size())
            m_editImage = m_editImage.scaled(size);

        m_saveInProgress = false;
        update();
    }
}
```

Saving

Saving is close to methods like undo and update. Looping thru drawn point container, using painter to draw those to backup image. As saving original image with drawing, use backup and it's associated point container.

```
void ImageCanvas::save(QString filename)
{
    qDebug() << __PRETTY_FUNCTION__ << " pen: " << m_penPoints.isEmpty() << " image: " << m_sourceImage.isNull() << " filename: " <<
    m_error = SaveError;
    m_saveInProgress = true;
    m_penEnabled = false;
    if(!filename.isEmpty()){
        QString path(QDesktopServices::storageLocation(QDesktopServices::PicturesLocation));
        qDebug() << "path: " << path;
        QString fn("e:/");
        fn.append( path.split('/',QString::SkipEmptyParts).last());
        fn.append("/");
        fn.append(filename);
        if(!fn.endsWith(".jpg"))
            fn.append(".jpg");

        QFile file(fn);
        file.remove();
        QImage tmp(m_sourceImage);

        QScopedPointer<QPainter> p(new QPainter());
        p->begin(&tmp);
        for(int i=0;i<m_penPoints.count();i++){
            CanvasPoint cpoint = m_penPoints.at(i);
            p->setPen(cpoint.pen);
            switch(cpoint.type){
            case DrawPen:{
                p->drawPoint(cpoint.start);
            }break;
            case DrawLine:{
                p->drawLine(cpoint.start,cpoint.end);
            }break;
            default:break;
            }
        }
        p->end();

        QScopedPointer<QImageWriter> w(new QImageWriter(fn));
        w->write(tmp);
        if(!w->error()){
            m_error = NoError;
        }
    }
    m_penEnabled = true;
    m_saveInProgress = false;
    emit saveChanged();
}
```

Demo video

The media player is loading...