Painting in Qt

Overview

This article demonstrates the **PaintingQt** touch screen painting application. It discusses how to reimplement some of QWidget's event handlers to receive the events generated by application's widgets, including the paint events used to draw the image and the resize event for optimising the application's appearance.

This application consists of two classes:

- `DrawingBoard` : A custom widget that displays a `QImage` and allows to the user to draw on it.
- `MainWindow` : Provides a menu to clear the screen

We will start with the `DrawingBoard` class and then with the `MainWindow` class.

DrawingBoard Class Definition

```
#ifndef DRAWINGBOARD_H
#define DRAWINGBOARD_H

#include <QWidget>
#include <QColor>
#include <QImage>
#include <QPoint>
#include <QtGui>
class DrawingBoard : public QWidget
{
    Q_OBJECT
public:
    explicit DrawingBoard(QWidget *parent = 0);
signals:
public slots:
    void clearImage();
protected:
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
    bool event(QEvent *event);
private:
    void resizeImage(QImage *image, const QSize &newSize);
    bool modified;
    QImage image;
};

#endif // DRAWINGBOARD_H
```

The `DrawingBoard` class inherits from `QWidget`. We reimplement the `event()` functions to implement the drawing. We reimplement the `paintEvent()` function to update the drawing area, and the `resizeEvent()` function to ensure that the `QImage` on which we draw is at least as large as the widget at any time and `resizeImage()` to change the size of a `QImage`. The `clearImage()` slot to clear the drawing area.

DrawingBoard Class Implementation

```
DrawingBoard::DrawingBoard(QWidget *parent) :
    QWidget(parent)
{
    setAttribute(Qt::WA_AcceptTouchEvents);
    setAttribute(Qt::WA_StaticContents);
    modified = false;
}
```

In the constructor, we set the widget attribute to `Qt::WA_StaticContents` to ensure that the widget contents are rooted to the top-left corner and don't change when the widget is resized. Qt uses this attribute to optimize paint events on resizes.

(C) Copyright Nokia 2013. All rights reserved.

```
void DrawingBoard::clearImage()
{
    image.fill(qRgb(255, 255, 255));
    modified = true;
    update();
}
```

The public `clearImage()` slot clears the image displayed in the drawing area. In this case we simply fill the entire image with white with the corresponding RGB value (255, 255, 255). And when we modify the image, we set modified to true and schedule a repaint.

```
void DrawingBoard::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    const QRect rect = event->rect();
    painter.drawImage(rect.topLeft(), image, rect);
}
```

In the reimplementation of the `paintEvent()` function, we simply create a `QPainter` for the drawing area, and then draw the image.

```
void DrawingBoard::resizeEvent(QResizeEvent *event)
{
   if (width() > image.width() || height() > image.height()) {
        int newWidth = qMax(width() + 128, image.width());
        int newHeight = qMax(height() + 128, image.height());
        resizeImage(&image, QSize(newWidth, newHeight));
        update();
    }
    QWidget::resizeEvent(event);
}
```

When the user starts the application, a resize event is generated and an image is created and displayed in the drawing area. We make this initial image sightly larger than the main window and drawing area, to avoid always resizing the image when the user resizes the main window .But when the main window becomes larger than this initial size, the image needs to be resized.

```
void DrawingBoard::resizeImage(QImage *image, const QSize &newSize)
{
   if (image->size() == newSize)
        return;
    QImage newImage(newSize, QImage::Format_RGB32);
    newImage.fill(qRgb(255, 255, 255));
    QPainter painter(&newImage);
    painter.drawImage(QPoint(0, 0), *image);
    *image = newImage;
}
```

There is no such nice API for resizing an image. So we created a new QImage one with the right size, fill it with white, and draw the old image onto it using `QPainter`. The new image is given `QImage::Format_RGB32` format, which means that each pixel is stored as `0xffRRGGBB` (where RR, GG, and BB are the red, green and blue color channels, ff is the hexadecimal value 255).

```
bool DrawingBoard::event(QEvent *event)
{
    switch (event->type())
    {
    case QEvent::TouchBegin:
    case QEvent::TouchUpdate:
    case QEvent::TouchEnd:
    {
        QList<QTouchEvent::TouchPoint> touchPoints = static_cast<QTouchEvent *>(event)->touchPoints();
        foreach (const QTouchEvent::TouchPoint &touchPoint, touchPoints)
        {
...
 modified = true;
```

In `event()` function, first we check what type of event is that, and then we draw a line from the point where the touch was located when the last touch was press or touch move occurred, we set `modified` to true, we generate a repaint event, and we **update** `lastPoint` so that next time `event()` is called, we continue drawing from where we left.

```
...
  QPainter painter(&image);
                    QColor color(255, 0, 0);
                    painter.setBrush(color);
                    painter.setPen(color);
                    painter.drawEllipse(rect);

...                  painter.end();
```

... using `QPainter` we set painting properties and paint the image

```
 modified = true;
  int rad = 2;
  update(rect.toRect().adjusted(-rad,-rad, +rad, +rad));
...
```

We call the `update()}` function and pass parameter `QRect` to specifies that the rectangle inside the drawing area are needs updating, to avoid a complete repaint of the widget

MainWindow Class Definition

(C) Copyright Nokia 2013. All rights reserved.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QList>
class DrawingBoard;

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow *ui;
 private:
     void createMyMenu();
     DrawingBoard *drawingboard;
     QAction *clearScreen;
};

#endif // MAINWINDOW_H
```

The MainWindow class inherits from `QMainWindow`. The `createMyMenu()` function creates the menu in the application.

MainWindow Class Implementation

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{

    drawingboard = new DrawingBoard;
    setCentralWidget(drawingboard);
    createMyMenu();
    setWindowTitle(tr("Drawing Board"));
    resize(360,640 );
}
```

In the constructor, we create a drawing area which we make the central widget of the `MainWindow` widget. Then we create the associated menus in `createMyMenu()` function.

```
void MainWindow::createMyMenu()
{
       clearScreen = new QAction(tr("&Clear Screen"), this);
        menuBar()->addAction(clearScreen);
        connect(clearScreen, SIGNAL(triggered()),  drawingboard, SLOT(clearImage()));
}
```

The `createMyMenu()` function creates the menu for the application.The `QMenuBar` class is used to create horizontal menu bar.

Source Code

The full source code presented in this article is available here File:PaintingQt.zip

--somnathbanik

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.