Archived:Pie Chart Using QPainter

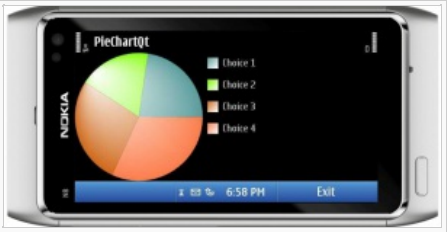
Archived: This article is archived because it is not considered relevant for third-party developers creating commercial solutions today. If you think this article is still relevant, let us know by adding the template {{ReviewForRemovalFromArchive|user=~~~~|*write your reason here*}}.

Qt Quick should be used for all UI development on mobile devices. The approach described in this article (using C++ for the Qt app UI) is deprecated.

This article demonstrates how to use the Qt QPainter class to paint on widget.

Introduction

We will write a Pie Chart class, which paints a pie chart together with its legend on a QWidget. We implement the actual drawing process in `paintEvent()`, and the widget obtains the data necessary for this from a QHash.



Pie Chart

Class Definition

```
#ifndef PIECHARTQT_H
#define PIECHARTQT_H
#include <QWidget>
#include <QWidget>
#include <QHash>
namespace Ui {
    class PiechartQt;
}
class PiechartQt : public QWidget
{
    Q_OBJECT
public:
    explicit PiechartQt(QWidget *parent = 0);
    ~PiechartQt();
    QSize sizeHint() const;
  //  QSize minimumSizeHint () const;
    void addEntry(const QString& key, int val);
protected:
void paintEvent(QPaintEvent *ev);
private:
QHash<QString, int> values;
private:
    Ui::PiechartQt *ui;
};
#endif // PIECHARTQT_H
```

Class Implementation

In this case we will populate the values hash table with data. The associated values are the number of people who made the choice. In the constructor we only perform the initializations for the parent class. The `addEntry()` method allows new values to be entered into the hash table.

```
PiechartQt::PiechartQt(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::PiechartQt)
{
  //  ui->setupUi(this);
}
PiechartQt::~PiechartQt()
{
    delete ui;
}
void PiechartQt::addEntry(const QString& key, int val) {
values.insert(key, val);
}
```

Before we take a look at the `paintEvent()`, we need to see how to divide up the widget. . The pie charts must always be round, so here the height should be the same as the width. The legend part should always be as wide as the longest text in the hash. The minimum height is calculated from the number of legend entries and their vertical spacing. Before we can start painting, we first calculate the sum of all the values.

```
void PiechartQt::paintEvent(QPaintEvent * /*ev*/)
{
// calculate total
QHash<QString, int>::const_iterator it;
int total = 0;
for(it = values.begin(); it != values.end(); ++it)
total += it.value();
```

```
// prepare painter
QPainter p(this);
p.setRenderHint(QPainter::Antialiasing, true);
```

We now instantiate the Painter and assign it the current widget (this) as the paint device. We also enable anti-aliasing. We also need to have a series of colors for the different pie segments in the diagram. To do this we access the `colorNames()` method, which gives us all the colors predefined in QColor. We also introduce the *colorPos* variable, which will later be used to select an element from the list. We initialize it with *13*, because from this point onward there are several pleasant pastel colors in succession .

```
QStringList colorNames = QColor::colorNames();
int colorPos = 13; // pastel colors
int height = rect().height();
QRect pieRect(0, 0, height, height);
```

Then we define the dimensions of the chart. These should exactly match the height of the widget. We obtain this height value from the current size of the widget, `rect()` returns the size in the form of a QRect, and we can extract the height from this with `height()`. *pieRect* is now initialized to contain the rectangle in which we will later draw our pie chart. We reserve the space remaining in the width for the key. We obtain the corresponding rectangle by first copying the measurements of the widget, with `rect()`, and then subtracting the width of pieRect from this square on the left side,with `setLeft()`

```
QRect legendRect = rect();
legendRect.setLeft(pieRect.width());
legendRect.adjust(10,10,-10,-10);
```

With the `adjust()` call we move ten pixels further inward from all sides, so the rectangle becomes smaller. This has the effect that we obtain ten pixels of space from the outer edges and from the right side of the pie graphics.

This causes the geometries for both parts of the widget to be dependent on the current widget size, and we proceed to draw the segments of the pie and the legend entries belonging to it, entry for entry. We need two help variables to do this. `lastAngleOffset` specifies the angle in the circle where we previously stopped drawing. We also require `currentPos`' later to draw the key entry at the correct position:

```
int lastAngleOffset = 0;
int currentPos = 0;
// create an entry for every piece of the pie
for(it = values.begin(); it != values.end(); ++it)
{
int value = it.value();
QString text = it.key();
int angle = (int)(16*360*(value/(double)total));
QColor col(colorNames.at(colorPos%colorNames.count()));
colorPos++;
// gradient for the pie pieces
QRadialGradient rg(pieRect.center(), pieRect.width()/2,
pieRect.topLeft());
rg.setColorAt(0, Qt::white);
rg.setColorAt(1, col);
p.setBrush(rg);
QPen pen = p.pen();
p.setPen(Qt::NoPen);
p.drawPie(pieRect, lastAngleOffset, angle);
lastAngleOffset += angle;
```

We again iterate through the hash and remember the keys and values. For each entry in the hash table we can determine how many degrees of the circle are to be apportioned to the current segment of pie. The value stored in the current key, divided by the total sum, results in the fraction that this value represents.

Multiplied by 360, this reveals how many degrees the segment of pie to be drawn takes up. It only remains to be explained from where the additional factor of 16 comes. This is due to a peculiarity of the `drawPie()` method.

We then select the current color using the `colorPos` variable from the `colorNames` list. With a modulo calculation (%), we ensure that under no circumstances do we overwrite the end of the list, which means that if we were to run out of colors, we would just start assigning the current color from the beginning of the list again.

```
int fh = fontMetrics().height();
QRect legendEntryRect(0,(fh*2)*currentPos,fh,fh);
currentPos++;
legendEntryRect.translate(legendRect.topLeft());
// define gradient for the legend squares
QLinearGradient lg(legendEntryRect.topLeft(),
legendEntryRect.bottomRight());
lg.setColorAt(0, col);
lg.setColorAt(1, Qt::white);
p.setBrush(QBrush(lg));
p.drawRect(legendEntryRect);
```

It is now time to draw the legend text next to the square. So that the distance from the square to the text is adjusted to the size of the font used, we select the width of the letter x in the respective font as the spacing. For this purpose we add the corresponding width to the x component of the `textStart` point. This variable now contains the top left point of our text. The lower left point is determined by combining the right edge of `legendRect` and the lower side of the current entry rectangle *legendEntryRect* into a new point. Together with `textStart`, this now opens up the `textEntryRect` in which space should be made for our text:

```
QPoint textStart = legendEntryRect.topRight();
textStart = textStart + QPoint(fontMetrics().width('x'), 0);
QPoint textEnd(legendRect.right(), legendEntryRect.bottom());
QRect textEntryRect(textStart, textEnd);
p.setPen(pen);
p.drawText(textEntryRect, Qt::AlignVCenter, text);
 }
}
```

After restoring our paintbrush, we insert the legend text precisely into the rectangle specified, using the `drawText()` method. The `AlignVCenter` option

ensures that the text is centered in its vertical alignment. We repeat this procedure for each entry in the list until the circle is completely filled. Finally, we use the class, we instantiate the widget, add a few entries with values, and display it.

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    PiechartQt w;
    w.addEntry("Choice 1", 50);
    w.addEntry("Choice 2", 40);
    w.addEntry("Choice 3", 60);
    w.addEntry("Choice 4", 70);
    w.showMaximized();
    return a.exec();
}
```

Source Code

The full source code for this article is available here: File:PieChartQt.zip

Reference

- Qt Reference Doc