065-qtprog.qxd 28.03.2001 8:48 Uhr Seite 65

Integrating graphics into Qt programs

GRAPHICAL REVOLUTION

MATTHIAS ETTRICH

Now that programming with Gtk/Gnome has been comprehensively examined, it is time to take a look at Qt as well. In this article we show how graphics can be built into a program quickly and easily with QCanvas.

Qt is an application development framework, written from the ground up as object-oriented in C++. Apart from Linux, it is also available on various other Unix platforms and also – in the professional version – under MS-Windows 95/98/NT/2000.

As a true multiplatform framework, Qt guarantees single-source compatibility: A Linux application written with pure Qt compiles and runs without any code changes under MS-Windows, too. It looks exactly the same there and also behaves just like any other MS-Windows application – it may even be a bit faster and more stable.

To make this possible, Qt encases not only graphical elements in objects, but also the other operating system functionalities such as files, printer or even network connections.

Basis and superstructure

Qt is the basis of KDE, so if you have installed KDE 2, there must automatically be a current version of Qt on your computer. If not, download the Qt source code direct from the manufacturer Troll Tech.

Before we pounce on the graphics, first comes the obligatory Hello World (see Listing 1). We use a QLabel to display the string *Hello world* in its own window. In lines 1 and 2, the necessary header files are integrated. QApplication is indispensable at this point, as this class is responsible for the event handling of the window system.

By command

So that the application can be controlled by command line parameters, the application object *a* is initialised in line 6 with the arguments *argc* and *argv*. Lastly, in line 8 we create the label and set the desired text string. The first parameter of each class constructor in this case is always a pointer to the parent object, usually a groupbox, a main window or a dialog.

Since the label in this example is to be used as its own window, without a parent, we simply assign a null pointer here. In line 11 we set the label of the application as main widget. The result of this is that under X11 common command line parameters such as *-geometry* are applied to the label and when the window is closed the application is ended.

Finally, the label in line 12 is made visible and control is handed over using *a.exec()* to the event loop of the application. If you would like to try out the example, save it as *hello.cpp* and compile it with:

```
g++ -O2 -o hello hello.cpp \-I$QTDIR/includ➘
e -L$QTDIR/lib -lqt
```

Listing 1: hello1.cpp

```
 1:#include <qapplication.h>
 2:#include <qlabel.h>
 3:
 4:intmain(intargc,char*argv[])
 5:{
 6:QApplicationa(argc,argv);
 7:
 8:QLabell(0);
 9:l.setText("HelloWorld");
10:
11:a.setMainWidget(&l);
12:l.show();
13:returna.exec();
14:}
```

065-qtprog.qxd 28.03.2001 8:48 Uhr Seite 66

OK, that was an easy text – now for the graphics. Copy your favourite image into the same directory as *hello.cpp*, in the example we are using *qtlogo.png*. Line 9 of the program flies out and in its place we create a pixmap with

```
QPixmap pm( "qtlogo.png" );
```

and set this on the label with:

```
l.setPixmap( pm );
```

So that this code will also compile, it will also need an *#include </qpixmap.h>* at the start of the program. As a little gimmick we have also set the property *ScaledContents* to *TRUE*. This means that the image will always be adapted to the size of the window.

Listing 2 shows the completed program. Naturally, the pixmap object does not load the image itself, it merely encases this functionality. Image formats are implemented in Qt's dynamic image-IO-system. This makes it possible to add drivers for additional image formats.

Listing 2: hello2.cpp

```
 1:#include <qapplication.h>
 2:#include <qlabel.h>
 3:#include <qpixmap.h>
 4:
 5:intmain(intargc,char*argv[])
 6:{
 7:QApplicationa(argc,argv);
 8:
 9:QLabell(0);
10:QPixmappm("qtlogo.png");
11:l.setPixmap(pm);
12:l.setScaledContents(TRUE);
13:
14:a.setMainWidget(&l);
15:l.show();
16:returna.exec();
17:}
```

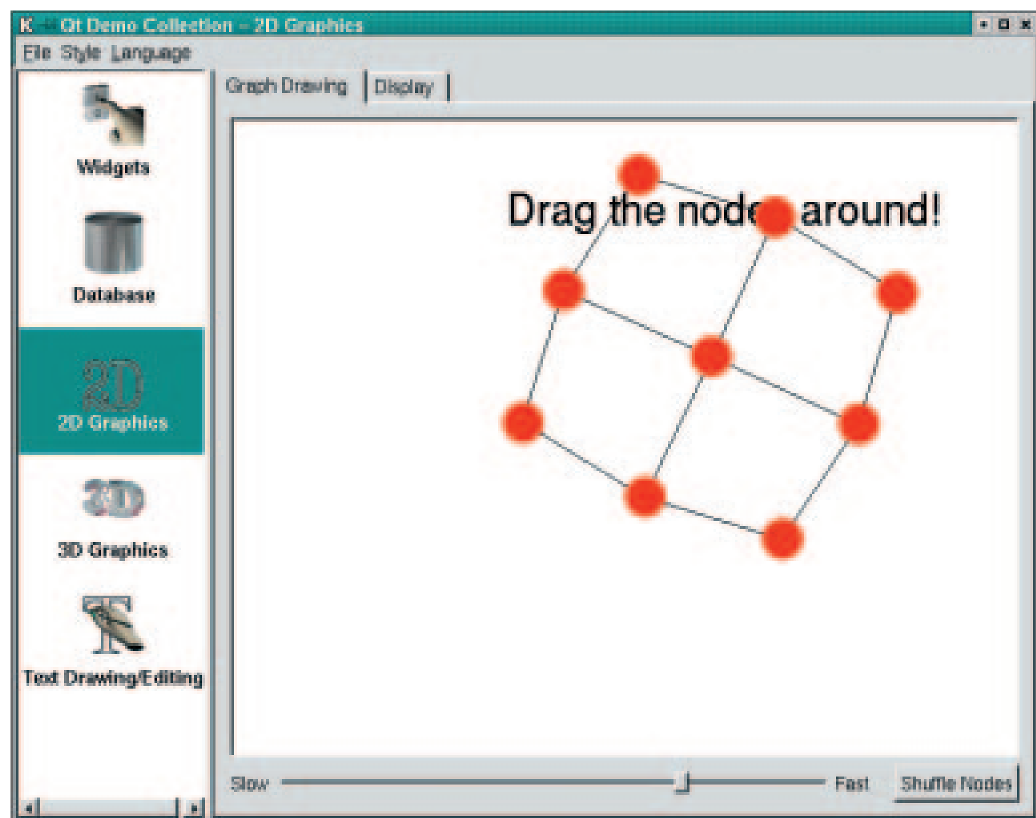
Driving force of development

PNG, BMP, XBM, XPM and PNM are supported as standard, and drivers for JPEG, MNG and GIF can, if required, be compiled into the Qt library. At *http://www.trolltech.com/qtprogs* you will also find a link to the free library Free Image from Floris van der Berg. These expand the image-IO-system by adding drivers for example for TARGA and TIFF.

If, in the program, you replace every mention of *Pixmap* with *Movie*, you can also load and display animated GIFs and MNGs (if Qt was compiled with the appropriate options).

Even more so than with images, one can also start with a canvas. A canvas is an optimised 2D graphics area, on which any so-called items can exist. These can be more or less anything, texts, polygons, circles or again, animated images – so-called Sprites.

QCanvas is one such graphic area, and what's more it is fairly powerful. Qt comes with a neat example called Canvas, which clarifies the options of the class. Examples of applications for QCanvas are of course games, but also vector-oriented drawing programs, diagram or graph editors, presentation programs and practically everything that requires dynamic 2D graphics.



A typical application for QCanvas: An animated graph editor. The implementation is just some 100 lines in length.

065-qtprog.qxd 28.03.2001 8:48 Uhr Seite 67

A broad canvas

In view of this flexibility of QCanvas it is interesting to know that the class was originally called QwSpriteField and was developed by Warwick Allison, who at that time was still outside TrollTech. Target applications were primarily 2D games, for example Qt Net Hack or the Kasteroids game from KDE. Warwick has now become head developer at TrollTech Australia and is in charge of Qt/Embedded development. QCanvas as generalised Spritefield has been an official component of Qt since Version 2.2.0.

The C++ class in Listing 3 defines a *Logo* sprite. The sprite moves freely about the canvas and turns automatically when it reaches the edge or comes across another sprite. The implementation of *Logo* is not complicated (around 65 lines), but does not make much of a contribution to understanding at this point. For reasons of space we have therefore omitted it, but the complete listing is on this month's coverdisc.

Here is just a brief description of the individual methods: *initPos()* selects an initial position at random, *initSpeed()* a random speed. Both functions use the Linux random number generator *rand(3)* for this.

Spritely animation

It starts to get really interesting in *advance()*. This method works out the animation of the sprite. It is triggered by the canvas itself, so you don't need to call it up manually. Canvas supports animations as a standard feature. To make sprites fly around, all you need to do, using *setVelocity(x,y)*, is to define a horizontal and a vertical speed. This is certainly enough for simple objects such as for example the bullets in an arcade game, but otherwise it is fairly boring.

This is where *advance()* comes into play. The aim of this method is to adapt movements of an item to the current situation on the canvas. *advance()* takes a parameter *stage* for this, which is either 0 or 1. The canvas firstly calls up all animated items *advance()* with *0* (Phase 0) and then with *1* (Phase 1).

The clash

In Phase 0 the items should, if at all possible, not move their position but start calculations for further motion, for example in collision with other items. QCanvasItem offers a function for this, *collisions()*, which sends back a list of all items which would collide with the item being called up if their speed remained constant. In Phase 1 the items then execute the previously calculated movement.

This two-phase approach makes it possible to maintain a certain fairness. Otherwise sprites added first to Canvas might take precedence in collisions, which does not, however, necessarily correspond to the physical conditions of a game world.

Listing 3: logos.cpp

```
01: #include <qapplication.h>
02: #include <qcanvas.h>
03: #include <stdlib.h>
04:
05: classLogo:publicQCanvasSprite{
06: voidinitPos();
07: voidinitSpeed();
08: public:
09: Logo(QCanvasPixmapArray*pm,QCanvas*c);
10: voidadvance(intstage);
11: };
12:
13: intmain(intargc,char*argv[])
14: {
15: QApplicationa(argc,argv);
16: QCanvascanvas(800,600);
17: QCanvasPixmapArraypm("qtlogo.png");
18:
19: for(inti=0;i<6;i++)
20: (newLogo(&pm,&canvas))->show();
21:
22: canvas.setAdvancePeriod(30);
23:
24: QCanvasViewcview(&canvas);
25: a.setMainWidget(&cview);
26: cview.show();
27: returna.exec();
28: }
```

In *main()* six logo sprites are created and made visible in a simple loop. The speed of animation is set using *canvas.setAdvancePeriod()*, in the example this is 30 milliseconds.

A sprite does not only have to consist of an image, as in the example. You can specify a whole array of images (hence *QCanvasPixmapArray*) and switch between these during run time with *setFrame()*. This makes it easy to implement running men, rotating boulders, exploding rockets and similar things.

You can find further information on QCanvas in the reference documentation supplied with Qt or online. One tutorial of special interest for games programmers has appeared at *zez.org*. ■

Info

Qt source code: http://www.trolltech.com/products/download/freelicense/qtfree-dl.html
Source code for the examples: logo.png: Animated, wandering and rebounding Qt logos
Reference material on QCanvas: http://doc.trolltech.com/qcanvas.html
Tutorial for games programmers: http://zez.org/article/articleview/2/