Graphics with Qt

Qt 3.0

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

The Coordinate System

A paint device in Qt is a drawable 2D surface. QWidget, QPixmap, QPicture and QPrinter are all paint devices. A QPainter is an object which can draw on such devices.

The default coordinate system of a paint device has its origin at the top left corner. X increases to the right and Y increases downwards. The unit is one pixel on pixel-based devices and one point on printers.

An Example

The illustration below shows a highly magnified portion of the top left corner of a paint device.
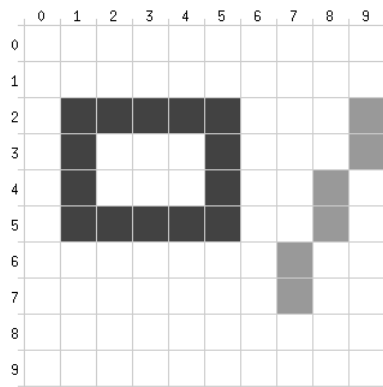
The rectangle and the line were drawn by this code (with the grid added and colors touched up in the illustration):

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    p.setPen( darkGray );
    p.drawRect( 1,2, 5,4 );
    p.setPen( lightGray );
    p.drawLine( 9,2, 7,7 );
}
```

Note that all of the pixels drawn by drawRect() are inside the size specified (5*4 pixels). This is different from some toolkits; in Qt the size you specify exactly encompasses the pixels drawn. This applies to all the relevant functions in QPainter.

Similarly, the drawLine() call draws both endpoints of the line, not just one.

Here are the classes that relate most closely to the coordinate system:

- QPoint is a single 2D point in the coordinate system. Most functions in Qt that deal with points can accept either a QPoint argument or two ints, for example QPainter::drawPoint().

4

- QSize is a single 2D vector. Internally, QPoint and QSize are the same, but a point is not the same as a size, so both classes exist. Again, most functions accept either a QSize or two ints, for example QWidget::resize().

- QRect is a 2D rectangle. Most functions accept either a QRect or four ints, for example QWidget::setGeometry().

- QRegion is an arbitrary set of points, including all the normal set operations, e.g. QRegion::intersect(), and also a less usual function to return a list of rectangles whose union is equal to the region. QRegion is used e.g. by QPainter::setClipRegion(), QWidget::repaint() and QPaintEvent::region().

- QPainter is the class that paints. It can paint on any device with the same code. There are differences between devices, QPrinter::newPage() is a good example, but QPainter works the same way on all devices.

- QPaintDevice is a device on which QPainter can paint. There are two internal devices, both pixel-based, and two external devices, QPrinter and QPicture (which records QPainter commands to a file or other QIODevice, and plays them back). Other devices can be defined.

Transformations

Although Qt's default coordinate system works as described above, QPainter also supports arbitrary transformations.

This transformation engine is a three-step pipeline, closely following the model outlined in books such as Foley & Van Dam and the OpenGL Programming Guide. Refer to those for in-depth coverage; here we give just a brief overview and an example.

The first step uses the world transformation matrix. Use this matrix to orient and position your objects in your model. Qt provides methods such as QPainter::rotate(), QPainter::scale(), QPainter::translate() and so on to operate on this matrix.

QPainter::save() and QPainter::restore() save and restore this matrix. You can also use QWMatrix objects, QPainter::worldMatrix() and QPainter::setWorldMatrix() to store and use named matrices.

The second step uses the window. The window describes the view boundaries in model coordinates. The matrix positions the *objects* and QPainter::setWindow() positions the *window*, deciding what coordinates will be visible. (If you have 3D experience, the window is what's usually called projection in 3D.)

The third step uses the viewport. The viewport too, describes the view boundaries, but in device coordinates. The viewport and the windows describe the same rectangle, but in different coordinate systems.

On-screen, the default is the entire QWidget or QPixmap where you are drawing, which is usually appropriate. For printing this function is vital, since very few printers can print over the entire physical page.

So each object to be drawn is transformed into model coordinates using QPainter::worldMatrix(), then clipped by QPainter::window(), and finally positioned on the drawing device using QPainter::viewport().

It is perfectly possible to do without one or two of the stages. If, for example, your goal is to draw something scaled, then using just QPainter::scale() makes perfect sense. If your goal is to use a fixed-size coordinate system, QPainter::setWindow() is perfect. And so on.

Here is a short example that uses all three mechanisms: the function that draws the clock face in the aclock/aclock.cpp example. We recommend compiling and running the example before you read any further. In particular, try resizing the window to different shapes.

```
void AnalogClock::drawClock( QPainter *paint )
{
    paint->save();
```

Firstly, we save the painter's state, so that the calling function is guaranteed not to be disturbed by the transformations we're going to use.

```
    paint->setWindow( -500,-500, 1000,1000 );
```

We set the model coordinate system we want a 1000*1000 window where 0,0 is in the middle.

```
QRect v = paint->viewport();
int d = QMIN( v.width(), v.height() );
```

The device may not be square and we want the clock to be, so we find its current viewport and compute its shortest side.

```
paint->setViewport( v.left() + (v.width()-d)/2,
                    v.top() + (v.height()-d)/2, d, d );
```

Then we set a new square viewport, centered in the old one.

We're now done with our view. From this point on, when we draw in a 1000*1000 area around 0,0, what we draw will show up in the largest possible square that'll fit in the output device.

Time to start drawing.

```
// time = QTime::currentTime();
QPointArray pts;
```

Since we'll draw a clock, we'll need to know the time. *pts* is just a utility variable to hold some points.

Next come three drawing blocks, one for the hour hand, one for the minute hand and finally one for the clock face itself. First we draw the hour hand:

```
paint->save();
paint->rotate( 30*(time.hour()%12-3) + time.minute()/2 );
```

We save the painter and then rotate it so that one axis points along the hour hand.

```
pts.setPoints( 4, -20,0,  0,-20, 300,0, 0,20 );
paint->drawConvexPolygon( pts );
```

We set *pts* to a four-point polygon that looks like the hour hand at three o'clock, and draw it. Because of the rotation, it's drawn pointed in the right direction.

```
paint->restore();
```

We restore the saved painter, undoing the rotation. We could also call rotate(-30) but that might introduce rounding errors, so it's better to use save() and restore(). Next, the minute hand, drawn almost the same way:

```
paint->save();
paint->rotate( (time.minute()-15)*6 );
pts.setPoints( 4, -10,0, 0,-10, 400,0, 0,10 );
paint->drawConvexPolygon( pts );
paint->restore();
```

The only differences are how the rotation angle is computed and the shape of the polygon.

The last part to be drawn is the clock face itself.

```
for ( int i=0; i<12; i++ ) {
    paint->drawLine( 440,0, 460,0 );
    paint->rotate( 30 );
}
```

Twelve short hour lines at thirty-degree intervals. At the end of that, the painter is rotated in a way which isn't very useful, but we're done with painting so that doesn't matter.

```
    paint->restore();
}
```

The final line of the function restores the painter, so that the caller won't be affected by all the transformations we've done.

Canvas Module

This module is part of the Qt Enterprise Edition.

The canvas module provides a highly optimized 2D graphic area called QCanvas. The canvas can contain an arbitrary number of QCanvasItems. Canvas items can have an arbitrary shape, size and content, can be freely moved around in the canvas, and can be checked for collisions. Canvas items can be set to move across the canvas automatically and animated canvas items are supported with QCanvasSprite. (If you require 3D graphics see Qt's OpenGL module.)

The canvas module uses a document/view model. The QCanvasView class is used to show a particular view of a canvas. Multiple views can operate on the same canvas at the same time. Every view can use an arbitrary transformation matrix on the canvas which makes it easy to implement features such as zooming.
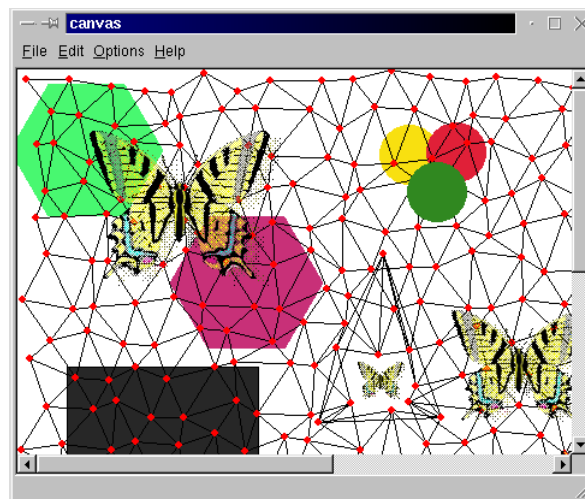
Qt provides a number of predefined QCanvas items as listed below.

- QCanvasItem — An abstract base class for all canvas items.
- QCanvasEllipse — An ellipse or "pie segment".
- QCanvasLine — A line segment.
- QCanvasPolygon — A polygon.
- QCanvasPolygonalItem — A base class for items that have a non-rectangular shape. Most canvas items derive from this class.
- QCanvasRectangle — A rectangle. The rectangle cannot be tilted or rotated. Rotated rectangles can be drawn using QCanvasPolygon.
- QCanvasSpline — A multi-bezier spline.
- QCanvasSprite — An animated pixmap.
- QCanvasText — A text string.

The two classes QCanvasPixmap and QCanvasPixmapArray are used by QCanvasSprite to show animated and moving pixmaps on the canvas.

More specialized items can be created by inheriting from one of the canvas item classes. It is easiest to inherit from one of QCanvasItem's derived classes (usually QCanvasPolygonalItem) rather than inherit QCanvasItem directly.

Window Geometry

Overview

QWidget provides several functions that deal with a widget's geometry. Some of these functions operate on the pure client area (i.e. the window excluding the window frame), others include the window frame. The differentiation is done in a way that covers the most common usage transparently.

Including the window frame: x(), y(), frameGeometry(), pos() and move() Excluding the window frame: geometry(), width(), height(), rect() and size()

Note that the distinction only matters for decorated top-level widgets. For all child widgets, the frame geometry is equal to the widget's client geometry.

This diagram shows most of the functions in use:
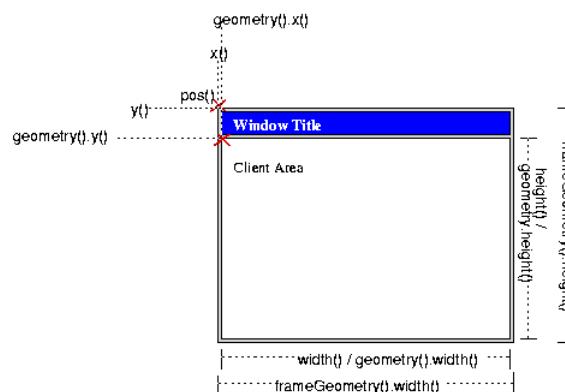
Unix/X11 peculiarities

On Unix/X11, a window does not have a frame until the window manager decorates it. This happens asynchronously at some point in time after calling show() and the first paint event the window receives - or it does not happen at all. Bear in mind that X11 is policy-free (others call it flexible). Thus you cannot make any safe assumption about the decoration frame your window will get. Basic rule: there's always one user who uses a window manager that breaks your assumption, and who will complain to you.

Furthermore, a toolkit cannot simply place windows on the screen. All Qt can do is to send certain hints to the window manager. The window manager, a separate process, may either obey, ignore or misunderstand them. Due to the partially unclear Inter-Client Communication Conventions Manual (ICCCM), window placement is handled quite differently in existing window managers.

X11 provides no standard or easy way to get the frame geometry once the window is decorated. Qt solves this problem with nifty heuristics and clever code that works on a wide range of window managers that exist today. Don't be surprised if you find one where frameGeometry() returns bogus results though.

X11 also does not provide a way to maximize a window. The showMaximized() function in Qt therefore has to emulate the feature. Its result depends totally on the result of frameGeometry() and the capability of the window manager to do proper window placement, both of which cannot be guaranteed.

Restoring a Window's Geometry

A common task in modern applications is to restore a window's geometry in a later session. On Windows, this is basically storing the result of geometry() and calling setGeometry() in the next session before doing show(). On X11, this won't work because an invisible window doesn't have a frame yet. The window manager would decorate the window later. When this happens, the window shifts towards the bottom/right corner of the screen depending on the size of the decoration frame. X theoretically provides a way to avoid this shift. Our tests have shown, though, that almost all window managers fail to implement this feature.

A workaround is to call setGeometry() after show(). This has the two disadvantages that the widget appears at a wrong place for a millisecond (results in flashing) and that currently only every second window manager gets it right. A safer solution is to store both pos() and size() and to restore the geometry using resize() and move() before calling show(), as demonstrated in the following example:

```
MyWidget* widget = new MyWidget
...
QPoint p = widget->pos();   // store position
QSize  s = widget->size();  // store size
...
widget = new MyWidget;
widget->resize( s );        // restore size
widget->move( p );          // restore position
widget->show();             // show widget
```

This method works on both MS-Windows and most existing X11 window managers.

Qt OpenGL 3D Graphics

This module is part of the Qt Enterprise Edition.

Introduction

OpenGL is a standard API for rendering 3D graphics.

OpenGL only deals with 3D rendering and provides little or no support for GUI programming issues. The user interface for an OpenGL* application must be created with another toolkit, such as Motif on the X platform, Microsoft Foundation Classes (MFC) under Windows - or Qt on *both* platforms.

The Qt OpenGL module makes it easy to use OpenGL in Qt applications. It provides an OpenGL widget class that can be used just like any other Qt widget, except that it opens an OpenGL display buffer where you can use the OpenGL API to render the contents.

The Qt OpenGL module is implemented as a platform-independent Qt/C++ wrapper around the platform-dependent GLX and WGL C APIs. The functionality provided is very similar to Mark Kilgard's GLUT library, but with much more non-OpenGL-specific GUI functionality, i.e. the whole Qt API.

Installation

When you install Qt for X11, the configure script will autodetect if OpenGL headers and libraries are installed on your system, and if so, it will include the Qt OpenGL module in the Qt library. (If your OpenGL headers or libraries are placed in a non-standard directory, you may need to change the SYSCONF_CXXFLAGS_OPENGL and/or SYSCONF_LFLAGS_OPENGL in the config file for your system).

When you install Qt for Windows, the Qt OpenGL module is always included.

The Qt OpenGL module is not licensed for use with the Qt Professional Edition. Consider upgrading to the Qt Enterprise Edition if you require OpenGL support.

Note about using Mesa on X11: Mesa versions earlier than 3.1 would use the name "MesaGL" and "MesaGLU" for the libraries, instead of "GL" and "GLU". If you want to use a pre-3.1 version of Mesa, you must change the Makefiles to use these library names instead. The easiest way to do this edit the SYSCONF_LIBS_OPENGL line in the config file you are using, changing "-lGL -lGLU" to "-lMesaGL -lMesaGLU"; then run "configure" again.

The QGL Classes

The OpenGL support classes in Qt are:

- QGLWidget: An easy-to-use Qt widget for rendering OpenGL scenes.
- QGLContext: Encapsulates an OpenGL rendering context.
- QGLFormat: Specifies the display format of a rendering context.

- QGLColormap: Handles indexed colormaps in GL-index mode.

Many applications only need the high-level QGLWidget class. The other QGL classes provide advanced features.

The QGL documentation assumes that you are familiar with OpenGL programming. If you're new to the subject a good starting point is http://www.opengl.org/.

* OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

QBitmap Class Reference

The QBitmap class provides monochrome (1-bit depth) pixmaps.

```
#include <qbitmap.h>
```

Inherits QPixmap [p. 225].

Public Members

- **QBitmap** ()
- **QBitmap** (int w, int h, bool clear = FALSE, QPixmap::Optimization optimization = QPixmap::DefaultOptim)
- **QBitmap** (const QSize & size, bool clear = FALSE, QPixmap::Optimization optimization = QPixmap::DefaultOptim)
- **QBitmap** (int w, int h, const uchar * bits, bool isXbitmap = FALSE)
- **QBitmap** (const QSize & size, const uchar * bits, bool isXbitmap = FALSE)
- **QBitmap** (const QBitmap & bitmap)
- **QBitmap** (const QString & fileName, const char * format = 0)
- QBitmap & **operator=** (const QBitmap & bitmap)
- QBitmap & **operator=** (const QPixmap & pixmap)
- QBitmap & **operator=** (const QImage & image)
- QBitmap **xForm** (const QWMatrix & matrix) const

Detailed Description

The QBitmap class provides monochrome (1-bit depth) pixmaps.

The QBitmap class is a monochrome off-screen paint device used mainly for creating custom QCursor and QBrush objects, in QPixmap::setMask() and for QRegion.

A QBitmap is a QPixmap with a QPixmap::depth() of 1. If a pixmap with a depth greater than 1 is assigned to a bitmap, the bitmap will be dithered automatically. A QBitmap is guaranteed to always have the depth 1, unless it is QPixmap::isNull() which has depth 0.

When drawing in a QBitmap (or QPixmap with depth 1), we recommend using the QColor objects `Qt::color0` and `Qt::color1`. Painting with `color0` sets the bitmap bits to 0, and painting with `color1` sets the bits to 1. For a bitmap, 0-bits indicate background (or white) and 1-bits indicate foreground (or black). Using the `black` and `white` QColor objects make no sense because the QColor::pixel() value is not necessarily 0 for black and 1 for white.

The QBitmap can be transformed (translated, scaled, sheared or rotated) using xForm().

Just like the QPixmap class, QBitmap is optimized by the use of implicit sharing, so it is very efficient to pass QBitmap objects as arguments.

See also QPixmap [p. 225], QPainter::drawPixmap() [p. 190], bitBlt() [p. 175], Shared Classes [Programming with Qt], Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QBitmap::QBitmap ()

Constructs a null bitmap.

See also QPixmap::isNull() [p. 233].

QBitmap::QBitmap (int w, int h, bool clear = FALSE, QPixmap::Optimization optimization = QPixmap::DefaultOptim)

Constructs a bitmap with width *w* and height *h*.

The contents of the bitmap is uninitialized if *clear* is FALSE; otherwise it is filled with pixel value 0 (the QColor `Qt::color0`).

The optional *optimization* argument specifies the optimization setting for the bitmap. The default optimization should be used in most cases. Games and other pixmap-intensive applications may benefit from setting this argument.

See also QPixmap::setOptimization() [p. 237] and QPixmap::setDefaultOptimization() [p. 236].

QBitmap::QBitmap (const QSize & size, bool clear = FALSE, QPixmap::Optimization optimization = QPixmap::DefaultOptim)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Constructs a bitmap with the size *size*.

The contents of the bitmap is uninitialized if *clear* is FALSE; otherwise it is filled with pixel value 0 (the QColor `Qt::color0`).

The optional *optimization* argument specifies the optimization setting for the bitmap. The default optimization should be used in most cases. Games and other pixmap-intensive applications may benefit from setting this argument.

QBitmap::QBitmap (int w, int h, const uchar * bits, bool isXbitmap = FALSE)

Constructs a bitmap with width *w* and height *h* and sets the contents to *bits*.

The *isXbitmap* should be TRUE if *bits* was generated by the X11 bitmap program. The X bitmap bit order is little endian. The QImage documentation discusses bit order of monochrome images.

Example (creates an arrow bitmap):

```
uchar arrow_bits[] = { 0x3f, 0x1f, 0x0f, 0x1f, 0x3b, 0x71, 0xe0, 0xc0 };
QBitmap bm( 8, 8, arrow_bits, TRUE );
```

QBitmap::QBitmap (const QSize & size, const uchar * bits, bool isXbitmap = FALSE)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Constructs a bitmap with the size *size* and sets the contents to *bits*.

The *isXbitmap* should be TRUE if *bits* was generated by the X11 bitmap program. The X bitmap bit order is little endian. The QImage documentation discusses bit order of monochrome images.

QBitmap::QBitmap (const QBitmap & bitmap)

Constructs a bitmap that is a copy of *bitmap*.

QBitmap::QBitmap (const QString & fileName, const char * format = 0)

Constructs a pixmap from the file *fileName*. If the file does not exist or is of an unknown format, the pixmap becomes a null pixmap.

The parameters *fileName* and *format* are passed on to QPixmap::load(). Dithering will be performed if the file format uses more than 1 bit per pixel.

See also QPixmap::isNull() [p. 233], QPixmap::load() [p. 233], QPixmap::loadFromData() [p. 234], QPixmap::save() [p. 236] and QPixmap::imageFormat() [p. 233].

QBitmap & QBitmap::operator= (const QBitmap & bitmap)

Assigns the bitmap *bitmap* to this bitmap and returns a reference to this bitmap.

QBitmap & QBitmap::operator= (const QPixmap & pixmap)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns the pixmap *pixmap* to this bitmap and returns a reference to this bitmap.

Dithering will be performed if the pixmap has a QPixmap::depth() greater than 1.

QBitmap & QBitmap::operator= (const QImage & image)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Converts the image *image* to a bitmap and assigns the result to this bitmap. Returns a reference to the bitmap.

Dithering will be performed if the image has a QImage::depth() greater than 1.

QBitmap QBitmap::xForm (const QWMatrix & matrix) const

Returns a transformed copy of this bitmap by using *matrix*.

This function does exactly the same as QPixmap::xForm(), except that it returns a QBitmap instead of a QPixmap.

See also QPixmap::xForm() [p. 237].

QBrush Class Reference

The QBrush class defines the fill pattern of shapes drawn by a QPainter.

`#include <qbrush.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QBrush** ( )
- **QBrush** (BrushStyle style)
- **QBrush** (const QColor & color, BrushStyle style = SolidPattern)
- **QBrush** (const QColor & color, const QPixmap & pixmap)
- **QBrush** (const QBrush & b)
- **~QBrush** ( )
- QBrush & **operator=** (const QBrush & b)
- BrushStyle **style** ( ) const
- void **setStyle** (BrushStyle s)
- const QColor & **color** ( ) const
- void **setColor** (const QColor & c)
- QPixmap * **pixmap** ( ) const
- void **setPixmap** (const QPixmap & pixmap)
- bool **operator==** (const QBrush & b) const
- bool **operator!=** (const QBrush & b) const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QBrush & b)
- QDataStream & **operator>>** (QDataStream & s, QBrush & b)

Detailed Description

The QBrush class defines the fill pattern of shapes drawn by a QPainter.

A brush has a style and a color. One of the brush styles is a custom pattern, which is defined by a QPixmap.

The brush style defines the fill pattern. The default brush style is NoBrush (depends on how you construct a brush). This style tells the painter to not fill shapes. The standard style for filling is called SolidPattern.

The brush color defines the color of the fill pattern. The QColor documentation lists the predefined colors.

Use the QPen class for specifying line/outline styles.

Example:

```
QPainter painter;
QBrush   brush( yellow );            // yellow solid pattern
painter.begin( &anyPaintDevice );   // paint something
painter.setBrush( brush );          // set the yellow brush
painter.setPen( NoPen );            // do not draw outline
painter.drawRect( 40,30, 200,100 ); // draw filled rectangle
painter.setBrush( NoBrush );        // do not fill
painter.setPen( black );            // set black pen, 0 pixel width
painter.drawRect( 10,10, 30,20 );   // draw rectangle outline
painter.end();                      // painting done
```

See the setStyle() [p. 19] function for a complete list of brush styles.

See also QPainter [p. 178], QPainter::setBrush() [p. 199], QPainter::setBrushOrigin() [p. 200], Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QBrush::QBrush ()

Constructs a default black brush with the style NoBrush (will not fill shapes).

QBrush::QBrush (BrushStyle style)

Constructs a black brush with the style *style*.

See also setStyle() [p. 19].

QBrush::QBrush (const QColor & color, BrushStyle style = SolidPattern)

Constructs a brush with the color *color* and the style *style*.

See also setColor() [p. 19] and setStyle() [p. 19].

QBrush::QBrush (const QColor & color, const QPixmap & pixmap)

Constructs a brush with the color *color* and a custom pattern stored in *pixmap*.

The color will only have an effect for monochrome pixmaps, i.e., QPixmap::depth() == 1.

See also setColor() [p. 19] and setPixmap() [p. 19].

QBrush::QBrush (const QBrush & b)

Constructs a brush that is a shallow copy of *b*.

QBrush::~QBrush ()

Destroys the brush.

const QColor & QBrush::color () const

Returns the brush color.

See also setColor() [p. 19].

bool QBrush::operator!= (const QBrush & b) const

Returns TRUE if the brush is different from *b* or FALSE if the brushes are equal.

Two brushes are different if they have different styles, colors or pixmaps.

See also operator==() [p. 19].

QBrush & QBrush::operator= (const QBrush & b)

Assigns *b* to this brush and returns a reference to this brush.

bool QBrush::operator== (const QBrush & b) const

Returns TRUE if the brush is equal to *b*, or FALSE if the brushes are different.

Two brushes are equal if they have equal styles, colors and pixmaps.

See also operator!=() [p. 19].

QPixmap * QBrush::pixmap () const

Returns a pointer to the custom brush pattern.

A null pointer is returned if no custom brush pattern has been set.

See also setPixmap() [p. 19].

void QBrush::setColor (const QColor & c)

Sets the brush color to *c*.

See also color() [p. 19] and setStyle() [p. 19].

Example: picture/picture.cpp.

void QBrush::setPixmap (const QPixmap & pixmap)

Sets the brush pixmap to *pixmap*. The style is set to CustomPattern.

The current brush color will only have an effect for monochrome pixmaps, i.e. QPixmap::depth() == 1.

See also pixmap() [p. 19] and color() [p. 19].

Example: richtext/richtext.cpp.

void QBrush::setStyle (BrushStyle s)

Sets the brush style to *s*.

The brush styles are:

- NoBrush will not fill shapes (default).
- SolidPattern solid (100%) fill pattern.
- Dense1Pattern 94% fill pattern.
- Dense2Pattern 88% fill pattern.
- Dense3Pattern 63% fill pattern.
- Dense4Pattern 50% fill pattern.
- Dense5Pattern 37% fill pattern.
- Dense6Pattern 12% fill pattern.
- Dense7Pattern 6% fill pattern.
- HorPattern horizontal lines pattern.
- VerPattern vertical lines pattern.
- CrossPattern crossing lines pattern.
- BDiagPattern diagonal lines (directed /) pattern.
- FDiagPattern diagonal lines (directed \ ) pattern.
- DiagCrossPattern diagonal crossing lines pattern.
- CustomPattern set when a pixmap pattern is being used.

See also style() [p. 20].

BrushStyle QBrush::style () const

Returns the brush style.

See also setStyle() [p. 19].

Related Functions

QDataStream & operator<< (QDataStream & s, const QBrush & b)

Writes the brush *b* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QBrush & b)

Reads the brush *b* from the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QBuffer Class Reference

The QBuffer class is an I/O device that operates on a QByteArray.

`#include <qbuffer.h>`

Inherits QIODevice [Input/Output and Networking with Qt].

Public Members

- **QBuffer** ()
- **QBuffer** (QByteArray buf)
- **~QBuffer** ()
- QByteArray **buffer** () const
- bool **setBuffer** (QByteArray buf)
- virtual Q_LONG **writeBlock** (const char * p, Q_ULONG len)
- Q_LONG **writeBlock** (const QByteArray & data)

Detailed Description

The QBuffer class is an I/O device that operates on a QByteArray.

QBuffer is used to read and write to a memory buffer. It is normally used with a QTextStream or a QDataStream. QBuffer has an associated QByteArray which holds the buffer data. The size() of the buffer is automatically adjusted as data is written.

The constructor `QBuffer(QByteArray)` creates a QBuffer with an existing byte array. The byte array can also be set with setBuffer(). Writing to the QBuffer will modify the original byte array because QByteArray is explicitly shared.

Use open() to open the buffer before use and to set the mode (read-only, write-only, etc.). close() closes the buffer. The buffer must be closed before reopening or calling setBuffer().

A common way to use QBuffer is through QDataStream or QTextStream, which have constructors that take a QBuffer parameter. For convenience, there are also QDataStream and QTextStream constructors that take a QByteArray parameter. These constructors create and open an internal QBuffer.

Note that QTextStream can also operate on a QString (a Unicode string); a QBuffer cannot.

You can also use QBuffer directly through the standard QIODevice functions readBlock(), writeBlock() readLine(), at(), getch(), putch() and ungetch().

See also QFile [Input/Output and Networking with Qt], QDataStream [Input/Output and Networking with Qt], QTextStream [Input/Output and Networking with Qt], QByteArray [Datastructures and String Handling with Qt], Shared Classes [Programming with Qt], Collection Classes [Datastructures and String Handling with Qt] and Input/Output and Networking.

Member Function Documentation

QBuffer::QBuffer ()

Constructs an empty buffer.

QBuffer::QBuffer (QByteArray buf)

Constructs a buffer that operates on *buf*. If you open the buffer in write mode (IO_WriteOnly or IO_ReadWrite) and write something into the buffer, *buf* will be modified.

Example:

```
QCString str = "abc";
QBuffer b( str );
b.open( IO_WriteOnly );
b.at( 3 ); // position at the 4th character (the terminating \0)
b.writeBlock( "def", 4 ); // write "def" including the terminating \0
b.close();
// Now, str == "abcdef" with a terminating \0
```

See also setBuffer() [p. 22].

QBuffer::~QBuffer ()

Destroys the buffer.

QByteArray QBuffer::buffer () const

Returns this buffer's byte array.

See also setBuffer() [p. 22].

bool QBuffer::setBuffer (QByteArray buf)

Replaces the buffer's contents with *buf*.

This may not be done when isOpen() is TRUE.

Note that if you open the buffer in write mode (IO_WriteOnly or IO_ReadWrite) and write something into the buffer, *buf* is also modified because QByteArray is an explicitly shared class.

See also buffer() [p. 22], open() [Input/Output and Networking with Qt] and close() [Input/Output and Networking with Qt].

Q_LONG QBuffer::writeBlock (const char * p, Q_ULONG len) [virtual]

Writes *len* bytes from *p* into the buffer at the current index, overwriting any characters there and extending the buffer if necessary. Returns the number of bytes actually written.

Returns -1 if an error occurred.

See also readBlock() [Input/Output and Networking with Qt].

Reimplemented from QIODevice [Input/Output and Networking with Qt].

Q_LONG QBuffer::writeBlock (const QByteArray & data)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This convenience function takes *data* and is the same as calling `writeBlock( data.data(), data.size() )`.

QCanvas Class Reference

The QCanvas class provides a 2D area that can contain QCanvasItem objects.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QCanvas** (QObject * parent = 0, const char * name = 0)
- **QCanvas** (int w, int h)
- **QCanvas** (QPixmap p, int h, int v, int tilewidth, int tileheight)
- virtual **~QCanvas** ()
- virtual void **setTiles** (QPixmap p, int h, int v, int tilewidth, int tileheight)
- virtual void **setBackgroundPixmap** (const QPixmap & p)
- QPixmap **backgroundPixmap** () const
- virtual void **setBackgroundColor** (const QColor & c)
- QColor **backgroundColor** () const
- virtual void **setTile** (int x, int y, int tilenum)
- int **tile** (int x, int y) const
- int **tilesHorizontally** () const
- int **tilesVertically** () const
- int **tileWidth** () const
- int **tileHeight** () const
- virtual void **resize** (int w, int h)
- int **width** () const
- int **height** () const
- QSize **size** () const
- QRect **rect** () const
- bool **onCanvas** (int x, int y) const
- bool **onCanvas** (const QPoint & p) const
- bool **validChunk** (int x, int y) const
- bool **validChunk** (const QPoint & p) const
- int **chunkSize** () const
- virtual void **retune** (int chunksze, int mxclusters = 100)
- virtual void **setAllChanged** ()
- virtual void **setChanged** (const QRect & area)
- virtual void **setUnchanged** (const QRect & area)
- QCanvasItemList **allItems** ()

- QCanvasItemList **collisions** (const QPoint & p) const
- QCanvasItemList **collisions** (const QRect & r) const
- QCanvasItemList **collisions** (const QPointArray & chunklist, const QCanvasItem * item, bool exact) const
- void **drawArea** (const QRect & clip, QPainter * painter, bool dbuf = FALSE)
- virtual void **setAdvancePeriod** (int ms)
- virtual void **setUpdatePeriod** (int ms)
- virtual void **setDoubleBuffering** (bool y)

Public Slots

- virtual void **advance** ()
- virtual void **update** ()

Signals

- void **resized** ()

Protected Members

- virtual void **drawBackground** (QPainter & painter, const QRect & clip)
- virtual void **drawForeground** (QPainter & painter, const QRect & clip)

Detailed Description

The QCanvas class provides a 2D area that can contain QCanvasItem objects.

The QCanvas class manages its 2D graphic area and all the canvas items the area contains. The canvas is displayed on screen with a QCanvasView widget. Multiple QCanvasView widgets may be associated with a canvas to provide multiple views of the same canvas.

The canvas is optimized for large numbers of items. Qt provides a rich set of canvas item classes, e.g. QCanvasEllipse, QCanvasLine, QCanvasPolygon, QCanvasPolygonalItem, QCanvasRectangle, QCanvasSpline, QCanvasSprite and QCanvasText. You can subclass to create your own canvas items; QCanvasPolygonalItem is the most common base class used for this purpose.

Although a canvas may appear to be similar to a widget with child widgets, there are several notable differences:

- Canvas items are usually far faster to manipulate and redraw than child widgets, with the speed advantage becoming especially great when there are *many* canvas items and non-rectangular items. In most situations canvas items are also a lot more memory efficient than child widgets.
- It's easy to detect overlapping items (collision detection).
- The canvas can be larger than a widget. A million-by-million canvas is perfectly possible. Although a widget might be very inefficient at this size and some window systems might not support it at all, QCanvas scales well. Even with a billion pixels and a million items finding a particular canvas item, detecting collisions, etc. is still fast.
- Two or more QCanvasView objects can view the same canvas.
- An arbitrary transformation matrix can be set on each QCanvasView which makes it easy to zoom, rotate or sheer the viewed canvas.

- Widgets provide a lot more functionality, such as input (QKeyEvent, QMouseEvent etc.) and layout management (QGridLayout etc.).

A canvas consists of a background, a number of canvas items organized by x, y and z coordinates, and a foreground. A canvas item's z coordinate may be treated as a layer number — canvas items with higher z coordinate will appear in front of canvas items with a lower z coordinate.

The background is white by default, but can be set to a different color using setBackgroundColor(), or to a repeated pixmap using setBackgroundPixmap() or to a mosaic of smaller pixmaps using setTiles(). Individual tiles can be set with setTile(). As usual, there are corresponding get functions like backgroundColor().

Note that QCanvas does not inherit from QWidget, even though it has some functions which provide the same functionality as those in QWidget. One of these is setBackgroundPixmap(); some others are resize(), size(), width() and height(). QCanvasView is the widget used to display a canvas on the screen.

Canvas items are added to a canvas by constructing them and passing the canvas to the canvas item's constructor. An item can be moved to a different canvas using QCanvasItem::setCanvas().

Canvas items are movable (and in the case of QCanvasSprites, animated) objects that inherit QCanvasItem. Each canvas item has a position on the canvas (x, y coordinates) and a height (z coordinate), all of which are held as floating-point numbers. Moving canvas items also have x and y velocities. It's possible for a canvas item to be outside the canvas (for example QCanvasItem::x() is greater than width()). When a canvas item is off the canvas, onCanvas() returns FALSE and the canvas disregards the item. (Canvas items off the canvas do not slow down any common opererations on the canvas.)

Canvas items can be moved with QCanvasItem::move(). The advance() function moves all QCanvasItem::animated() canvas items and setAdvancePeriod() makes QCanvas move them by itself on a periodic basis. In the context of the QCanvas classes to 'animate' a canvas item is to set it in motion, i.e. using QCanvasItem::setVelocity(). Animation of a canvas item itself, i.e. items which change over time, is enabled by calling QCanvasSprite::setFrameAnimation(), or more generally by subclassing and reimplementing QCanvasItem::advance(). To detect collisions use one of the QCanvasItem::collisions() functions.

The changed parts of the canvas are redrawn (if they are visible in a canvas view) whenever update() is called. You can either call update() manually after having changed the contents of the canvas, or force periodic updates using setUpdatePeriod(). If you have moving objects on the canvas, you need to call advance() every time the objects should move one step further. Periodic calls to advance() can be forced using setAdvancePeriod(). The advance() function will call QCanvasItem::advance() on every item that is QCanvasItem::animated() and trigger an update of the affected areas afterwards. (A canvas item that is 'animated' is simply a canvas item that is in motion.)

QCanvas organizes its canvas items into *chunks* - areas on the canvas that are used to speed up most operations. Many operations start by eliminating most chunks (i.e. those which haven't changed) and then process only the canvas items that are in the few interesting (i.e. changed) chunks. A valid chunk, validChunk(), is one which is on the canvas.

The chunk size is a key factor to QCanvas's speed: if there are too many chunks, the speed benefit of grouping canvas items into chunks is reduced. If the chunks are too large, it takes too long to process each one. The QCanvas constructor picks a hopefully suitable size, but you can call retune() to change it at any time. The chunkSize() function returns the current chunk size.

The canvas items always make sure they're in the right chunks; all you need to make sure of is that the canvas uses the right chunk size. A good rule of thumb is that the size should be a bit smaller than the average canvas item size. If you have moving objects, the chunk size should be a bit smaller than the average size of the moving items.

The foreground is normally nothing, but if you reimplement drawForeground(), you can draw things in front of all canvas items.

Areas can be set as changed with setChanged() and set unchanged with setUnchanged(). The entire canvas can be set as changed with setAllChanged(). A list of all the items on the canvas is returned by allItems().

An area can be copied (painted) to a QPainter with drawArea().

If the canvas is resized it emits the resized() signal.

The examples/canvas application and the 2D graphics page of the examples/demo application demonstrate many

of QCanvas's facilities.

See also QCanvasView [p. 71], QCanvasItem [p. 36], Abstract Widget Classes, Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvas::QCanvas (QObject * parent = 0, const char * name = 0)

Create a QCanvas with no size. *parent* and *name* have the usual QObject meaning.

You *must* call resize() at some time after creation to be able to use the canvas.

QCanvas::QCanvas (int w, int h)

Constructs a QCanvas that is *w* pixels wide and *h* pixels high.

QCanvas::QCanvas (QPixmap p, int h, int v, int tilewidth, int tileheight)

Constructs a QCanvas which will be composed of *h* tiles horizontally and *v* tiles vertically. Each tile will be an image *tilewidth* by *tileheight* pixels taken from pixmap *p*.

The pixmap *p* is a list of tiles, arranged left to right, (and in the case of pixmaps that have multiple rows of tiles, top to bottom), with tile 0 in the top-left corner, tile 1 next to the right, and so on, e.g.

| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |

The QCanvas is initially sized to show exactly the given number of tiles horizontally and vertically. If it is resized to be larger, the entire matrix of tiles will be repeated as much as necessary to cover the area. If it is smaller, tiles to the right and bottom will not be visible.

See also setTiles() [p. 31].

QCanvas::~QCanvas () [virtual]

Destroys the canvas and all the canvas's canvas items.

void QCanvas::advance () [virtual slot]

Moves all QCanvasItem::animated() canvas items on the canvas and refreshes all changes to all views of the canvas. (An 'animated' item is an item that is in motion, see setVelocity().)

The advance is done in two phases. In phase 0, the QCanvasItem::advance() function of each QCanvasItem::animated() canvas item is called with paramater 0. Then all these canvas items are called again, with parameter 1. In phase 0, the canvas items should not change position, merely examine other items on the canvas for which special processing is required, such as collisions between items. In phase 1, all canvas items should change positions, ignoring any other items on the canvas. This two-phase approach allows for considerations of "fairness", though no QCanvasItem subclasses supplied with Qt do anything interesting in phase 0.

The canvas can be configured to call this function periodically with setAdvancePeriod().

See also update() [p. 32].

QCanvasItemList QCanvas::allItems ()

Returns a list of all items in the canvas.

QColor QCanvas::backgroundColor () const

Returns the color set by setBackgroundColor(). By default, this is white.

Note that this function is not a reimplementation of QWidget::backgroundColor() (QCanvas is not a subclass of QWidget), but all QCanvasViews that are viewing the canvas will set their backgrounds to this color.

See also setBackgroundColor() [p. 30] and backgroundPixmap() [p. 28].

QPixmap QCanvas::backgroundPixmap () const

Returns the pixmap set by setBackgroundPixmap(). By default, this is a null pixmap.

See also setBackgroundPixmap() [p. 30] and backgroundColor() [p. 28].

int QCanvas::chunkSize () const

Returns the chunk size of the canvas.

See also retune() [p. 30].

QCanvasItemList QCanvas::collisions (const QPoint & p) const

Returns a list of canvas items that intersect with the point *p*. The list is ordered by z coordinates, from highest z coordinate (front-most item) to lowest z coordinate (rear-most item).

QCanvasItemList QCanvas::collisions (const QRect & r) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a list of items which intersect with the rectangle *r*. The list is ordered by z coordinates, from highest z coordinate (front-most item) to lowest z coordinate (rear-most item).

QCanvasItemList QCanvas::collisions (const QPointArray & chunklist, const QCanvasItem * item, bool exact) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a list of canvas items which intersect with the chunks listed in *chunklist*, excluding *item*. If *exact* is TRUE, only only those which actually QCanvasItem::collidesWith() *item* are returned, otherwise canvas items are included just for being in the chunks.

This is a utility function mainly used to implement the simpler QCanvasItem::collisions() function.

void QCanvas::drawArea (const QRect & clip, QPainter * painter, bool dbuf = FALSE)

Paints all canvas items that are in the area *clip* to *painter*, using double-buffering if *dbuf* is TRUE.

eg. to print the canvas to a printer:

```
QPrinter pr;
if ( pr.setup() ) {
  QPainter p(&pr);
  canvas.drawArea( canvas.rect(), &p );
}
```

void QCanvas::drawBackground (QPainter & painter, const QRect & clip) [virtual protected]

This virtual function is called for all updates of the canvas. It renders any background graphics using the painter *painter*, in the area *clip*. If the canvas has a background pixmap or a tiled background, that graphic is used, otherwise the canvas is cleared using the background color.

If the graphics for an area change, you must explicitly call setChanged(const QRect&) for the result to be visible when update() is next called.

See also setBackgroundColor() [p. 30], setBackgroundPixmap() [p. 30] and setTiles() [p. 31].

void QCanvas::drawForeground (QPainter & painter, const QRect & clip) [virtual protected]

This virtual function is called for all updates of the canvas. It renders any foreground graphics using the painter *painter*, in the area *clip*.

If the graphics for an area change, you must explicitly call setChanged(const QRect&) for the result to be visible when update() is next called.

The default is to draw nothing.

int QCanvas::height () const

Returns the height of the canvas, in pixels.

bool QCanvas::onCanvas (int x, int y) const

Returns TRUE if the pixel position (x, y) is on the canvas; otherwise returns FALSE.

bool QCanvas::onCanvas (const QPoint & p) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the pixel position *p* is on the canvas; otherwise returns FALSE.

QRect QCanvas::rect () const

Returns a rectangle the size of the canvas.

void QCanvas::resize (int w, int h) [virtual]

Changes the size of the canvas to have a width of *w* and a height of *h*. This is a slow operation.

void QCanvas::resized () [signal]

This signal is emitted whenever the canvas is resized. Each QCanvasView connects to this signal to keep the scrollview size correct.

void QCanvas::retune (int chunksze, int mxclusters = 100) [virtual]

Change the efficiency tuning parameters to *mxclusters* clusters, each of size *chunksze*. This is a slow operation if there are many objects on the canvas.

The canvas is divided into chunks which are rectangular areas of the canvas *chunksze* wide by *chunksze* high. Use a chunk size which is about the average size of the canvas items. If you choose a chunk size which is too small it will increase the amount of calculation required when drawing since each change will affect many chunks. If you choose a chunk size which is too large the amount of drawing required will increase because for each change a lot of drawing will be required because there will be many (unchanged) canvas items which are in the same chunk as the changed canvas items.

Internally, a canvas uses a low-resolution "chunk matrix" to keep track of all the items in the canvas. A 64x64 chunk matrix is the default for a 1024x1024 pixel canvas, where each chunk collects canvas items in a 16x16 pixel square. This default is also affected by setTiles(). You can tune this default with this function, for example if you have a very large canvas and want to trade off speed for memory then you might set the chunk size to 32 or 64.

The *mxclusters* argument is the number of rectangular groups of chunks that will be separately drawn. If the canvas has a large number of small, dispersed items, this should be about that number. Our testing suggests that a large number of clusters is almost always best.

void QCanvas::setAdvancePeriod (int ms) [virtual]

Sets the canvas to call advance() every *ms* milliseconds. Any previous setting by setAdvancePeriod() or setUpdate-Period() is overridden.

If *ms* is less than 0 advancing will be stopped.

void QCanvas::setAllChanged () [virtual]

Marks the whole canvas as changed. All views of the canvas will be entirely redrawn when update() is next called.

void QCanvas::setBackgroundColor (const QColor & c) [virtual]

Sets the solid background to be the color *c*.

See also backgroundColor() [p. 28], setBackgroundPixmap() [p. 30] and setTiles() [p. 31].

void QCanvas::setBackgroundPixmap (const QPixmap & p) [virtual]

Sets the solid background to be the pixmap *p* repeated as necessary to cover the entire canvas.

See also backgroundPixmap() [p. 28], setBackgroundColor() [p. 30] and setTiles() [p. 31].

void QCanvas::setChanged (const QRect & area) [virtual]

Marks *area* as changed. This area will be redrawn in all views showing it when update() is next called.

void QCanvas::setDoubleBuffering (bool y) [virtual]

If *y* is TRUE (the default) double-buffering is switched on; otherwise double-buffering is switched off.

Turning off double-buffering causes the redrawn areas to flicker a bit also gives a (usually small) performance improvement.

void QCanvas::setTile (int x, int y, int tilenum) [virtual]

Sets the tile at (*x*, *y*) to use tile number *tilenum*, which is an index into the tile pixmaps. The canvas will update appropriately when update() is next called.

The images are taken from the pixmap set by setTiles() and are arranged left to right, (and in the case of pixmaps that have multiple rows of tiles, top to bottom), with tile 0 in the top-left corner, tile 1 next to the right, and so on, e.g.

0	1	2	3
4	5	6	7

See also tile() [p. 32] and setTiles() [p. 31].

void QCanvas::setTiles (QPixmap p, int h, int v, int tilewidth, int tileheight) [virtual]

Sets the QCanvas to be composed of *h* tiles horizontally and *v* tiles vertically. Each tile will be an image *tilewidth* by *tileheight* pixels from pixmap *p*.

The pixmap *p* is a list of tiles, arranged left to right, (and in the case of pixmaps that have multiple rows of tiles, top to bottom), with tile 0 in the top-left corner, tile 1 next to the right, and so on, e.g.

0	1	2	3
4	5	6	7

If the canvas is larger than the matrix of tiles, the entire matrix is repeated as necessary to cover the whole canvas. If it is smaller, tiles to the right and bottom are not visible.

The width and height of *p* must be a multiple of *tilewidth* and *tileheight*. If they are not the function will return without performing any action.

void QCanvas::setUnchanged (const QRect & area) [virtual]

Marks *area* as *unchanged*. The area will *not* be redrawn in the views for the next update(), unless it is marked a changed again before the next call to update().

void QCanvas::setUpdatePeriod (int ms) [virtual]

Sets the canvas to call update() every *ms* milliseconds. Any previous setting by setAdvancePeriod() or setUpdatePeriod() is cancelled.

If *ms* is less than 0 automatic updating will be stopped.

QSize QCanvas::size () const

Returns the size of the canvas, in pixels.

int QCanvas::tile (int x, int y) const

Returns the tile at position (x, y). Initially, all tiles are 0.

The parameters must be within range, i.e. $0 < x <$ tilesHorizontally() and $0 < y <$ tilesVertically().

See also setTile() [p. 31].

int QCanvas::tileHeight () const

Returns the height of each tile.

int QCanvas::tileWidth () const

Returns the width of each tile.

int QCanvas::tilesHorizontally () const

Returns the number of tiles horizontally.

int QCanvas::tilesVertically () const

Returns the number of tiles vertically.

void QCanvas::update () [virtual slot]

Repaints changed areas in all views of the canvas.

See also advance() [p. 27].

bool QCanvas::validChunk (int x, int y) const

Returns TRUE if the chunk position (x, y) is on the canvas; otherwise returns FALSE.

bool QCanvas::validChunk (const QPoint & p) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns whether the chunk position p is on the canvas.

int QCanvas::width () const

Returns the width of the canvas, in pixels.

QCanvasEllipse Class Reference

The QCanvasEllipse class provides an ellipse or ellipse segment on a QCanvas.

This class is part of the **canvas** module.

`#include <qcanvas.h>`

Inherits QCanvasPolygonalItem [p. 54].

Public Members

- **QCanvasEllipse** (QCanvas * canvas)
- **QCanvasEllipse** (int width, int height, QCanvas * canvas)
- **QCanvasEllipse** (int width, int height, int startangle, int angle, QCanvas * canvas)
- **~QCanvasEllipse** ()
- int **width** () const
- int **height** () const
- void **setSize** (int width, int height)
- void **setAngles** (int start, int length)
- int **angleStart** () const
- int **angleLength** () const
- virtual int **rtti** () const

Protected Members

- virtual void **drawShape** (QPainter & p)

Detailed Description

The QCanvasEllipse class provides an ellipse or ellipse segment on a QCanvas.

A canvas item that paints an ellipse or ellipse segment with a QBrush. The ellipse's height, width, start angle and angle length can be set at construction time. The size can be changed at runtime with setSize(), and the angles can be changed (if you're displaying an ellipse segment rather than a whole ellipse) with setAngles().

Note that angles are specified in 16ths of a degree.
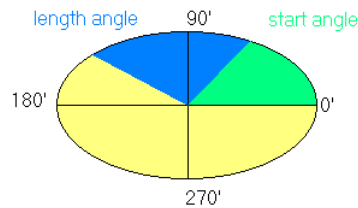
If a start angle and length angle are set then an ellipse segment will be drawn. The start angle is the angle that goes from zero in a counter-clockwise direction (shown in green in the diagram). The length angle is the angle from the start angle in a counter-clockwise direction (shown in blue in the diagram). The blue segment is the segment of the ellipse that would be drawn. If no start angle and length angle are specified the entire ellipse is drawn.

The ellipse can be drawn on a painter with drawShape().

Like any other canvas item ellipses can be moved with move() and moveBy(), or by setting coordinates with setX(), setY() and setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasEllipse::QCanvasEllipse (QCanvas * canvas)

Constructs a 32x32 ellipse, centered at (0, 0) on *canvas*.

QCanvasEllipse::QCanvasEllipse (int width, int height, QCanvas * canvas)

Constructs a *width* by *height* pixel ellipse, centered at (0,0) on *canvas*.

QCanvasEllipse::QCanvasEllipse (int width, int height, int startangle, int angle, QCanvas * canvas)

Constructs a *width* by *height* pixel ellipse, centered at (0,0) on *canvas*. Only a segment of the ellipse is drawn, starting at angle *startangle*, and extending for angle *angle* (the angle length).

Note that angles are specified in 1/16ths of a degree.

QCanvasEllipse::~QCanvasEllipse ()

Destroys the ellipse.

int QCanvasEllipse::angleLength () const

Returns the length angle (the extent of the ellipse segment) in 16ths of a degree. Initially this will be 360 * 16 (a complete ellipse).

See also setAngles() [p. 35] and angleStart() [p. 35].

int QCanvasEllipse::angleStart () const

Returns the start angle in 16ths of a degree. Initially this will be 0.

See also setAngles() [p. 35] and angleLength() [p. 34].

void QCanvasEllipse::drawShape (QPainter & p) [virtual protected]

Draws the ellipse, centered at x(), y() using the painter *p*.

Note that QCanvasEllipse does not support an outline (pen is always NoPen).

Reimplemented from QCanvasPolygonalItem [p. 56].

int QCanvasEllipse::height () const

Returns the height of the ellipse.

int QCanvasEllipse::rtti () const [virtual]

Returns 6 (QCanvasItem::Rtti_Ellipse).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasPolygonalItem [p. 56].

void QCanvasEllipse::setAngles (int start, int length)

Sets the angles for the ellipse. The start angle is *start* and the extent of the segment is *length* (the angle length) from the *start*. The angles are specified in 16ths of a degree. By default the ellipse will start at 0 and have an angle length of 360 * 16 (a complete ellipse).

See also angleStart() [p. 35] and angleLength() [p. 34].

void QCanvasEllipse::setSize (int width, int height)

Sets the *width* and *height* of the ellipse.

int QCanvasEllipse::width () const

Returns the width of the ellipse.

QCanvasItem Class Reference

The QCanvasItem class provides an abstract graphic object on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits Qt [Additional Functionality with Qt].

Inherited by QCanvasSprite [p. 63], QCanvasPolygonalItem [p. 54] and QCanvasText [p. 68].

Public Members

- **QCanvasItem** (QCanvas * canvas)
- virtual **~QCanvasItem** ()
- double **x** () const
- double **y** () const
- double **z** () const
- virtual void **moveBy** (double dx, double dy)
- void **move** (double x, double y)
- void **setX** (double x)
- void **setY** (double y)
- void **setZ** (double z)
- bool **animated** () const
- virtual void **setAnimated** (bool y)
- virtual void **setVelocity** (double vx, double vy)
- void **setXVelocity** (double vx)
- void **setYVelocity** (double vy)
- double **xVelocity** () const
- double **yVelocity** () const
- virtual void **advance** (int phase)
- virtual bool **collidesWith** (const QCanvasItem * other) const
- QCanvasItemList **collisions** (bool exact) const
- virtual void **setCanvas** (QCanvas * c)
- virtual void **draw** (QPainter & painter)
- void **show** ()
- void **hide** ()
- virtual void **setVisible** (bool yes)
- bool **isVisible** () const
- virtual void **setSelected** (bool yes)
- bool **isSelected** () const
- virtual void **setEnabled** (bool yes)

- bool **isEnabled** () const
- virtual void **setActive** (bool yes)
- bool **isActive** () const
- bool visible () const *(obsolete)*
- bool selected () const *(obsolete)*
- bool enabled () const *(obsolete)*
- bool active () const *(obsolete)*
- enum **RttiValues** { Rtti_Item = 0, Rtti_Sprite = 1, Rtti_PolygonalItem = 2, Rtti_Text = 3, Rtti_Polygon = 4, Rtti_Rectangle = 5, Rtti_Ellipse = 6, Rtti_Line = 7, Rtti_Spline = 8 }
- virtual int **rtti** () const
- virtual QRect **boundingRect** () const
- virtual QRect **boundingRectAdvanced** () const
- QCanvas * **canvas** () const

Protected Members

- void **update** ()

Detailed Description

The QCanvasItem class provides an abstract graphic object on a QCanvas.

A variety of subclasses provide immediately usable behaviour; this class is a pure abstract superclass providing the behaviour that is shared among all the concrete canvas item classes. QCanvasItem is not intended for direct subclassing. It is much easier to subclass one of its subclasses, e.g. QCanvasPolygonalItem (the commonest base class), QCanvasRectangle, QCanvasSprite, QCanvasEllipse or QCanvasText.

Canvas items are added to a canvas by constructing them and passing the canvas to the canvas item's constructor. An item can be moved to a different canvas using setCanvas().

A QCanvasItem object can be moved in the x(), y() and z() dimensions using functions such as move(), moveBy(), setX(), setY() and setZ(). A canvas item can be set in motion, 'animated', using setAnimated() and given a velocity in the x and y directions with setXVelocity() and setYVelocity() — the same effect can be achieved by calling setVelocity(). Use the collidesWith() function to see if the canvas item will collide on the *next* advance(1) and use collisions() to see what collisions have occurred.

Use QCanvasSprite or your own subclass of QCanvasSprite to create canvas items which are animated, i.e. which change over time.

The size of a canvas item is given by boundingRect(). Use boundingRectAdvanced() to see what the size of the canvas item will be *after* the next advance(1) call.

The rtti() function is used for identifying subclasses of QCanvasItem. The canvas() function returns a pointer to the canvas which contains the canvas item.

QCanvasItem provides the show() and isVisible() functions like those in QWidget.

QCanvasItem also provides the setEnabled(), setActive() and setSelected() functions; these functions set the relevant boolean and cause a repaint but the boolean values they set are not used in QCanvasItem itself. You can make use of these booleans in your subclasses.

By default canvas items have no velocity, no size and are not in motion. The subclasses provided in Qt do not change these defaults except where noted.

See also Graphics Classes and Image Processing Classes.

Member Type Documentation

QCanvasItem::RttiValues

This enum is used to name the different types of canvas item.

- `QCanvasItem::Rtti_Item` - Canvas item abstract base class
- `QCanvasItem::Rtti_Ellipse`
- `QCanvasItem::Rtti_Line`
- `QCanvasItem::Rtti_Polygon`
- `QCanvasItem::Rtti_PolygonalItem`
- `QCanvasItem::Rtti_Rectangle`
- `QCanvasItem::Rtti_Spline`
- `QCanvasItem::Rtti_Sprite`
- `QCanvasItem::Rtti_Text`

Member Function Documentation

QCanvasItem::QCanvasItem (QCanvas * canvas)

Constructs a QCanvasItem on canvas *canvas*.

See also setCanvas() [p. 41].

QCanvasItem::~QCanvasItem () [virtual]

Destroys the QCanvasItem and removes it from its canvas.

bool QCanvasItem::active () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use isActive() instead.

void QCanvasItem::advance (int phase) [virtual]

The default implementation moves the canvas item, if it is animated(), by the preset velocity if *phase* is 1, and does nothing if *phase* is 0.

Note that if you reimplement this function, the reimplementation must not change the canvas in any way, for example it must not add or remove items.

See also QCanvas::advance() [p. 27] and setVelocity() [p. 42].

Reimplemented in QCanvasSprite.

bool QCanvasItem::animated () const

Returns TRUE is the canvas item is in motion; otherwise returns FALSE.

See also setVelocity() [p. 42] and setAnimated() [p. 41].

QRect QCanvasItem::boundingRect () const [virtual]

Returns the bounding rectangle in pixels that the canvas item covers.

See also boundingRectAdvanced() [p. 39].

Reimplemented in QCanvasSprite, QCanvasPolygonalItem and QCanvasText.

QRect QCanvasItem::boundingRectAdvanced () const [virtual]

Returns the bounding rectangle of pixels that the canvas item *will* cover after advance(1) is called.

See also boundingRect() [p. 39].

QCanvas * QCanvasItem::canvas () const

Returns the canvas containing the canvas item.

bool QCanvasItem::collidesWith (const QCanvasItem * other) const [virtual]

Returns TRUE if the canvas item will collide with the *other* item *after* they have moved by their current velocities; otherwise returns FALSE.

See also collisions() [p. 39].

QCanvasItemList QCanvasItem::collisions (bool exact) const

Returns the list of canvas items that this canvas item has collided with.

A collision is generally defined as pixels of one item drawing on the pixels of another item, but not all subclasses are so precise. Also, since pixel-wise collision detection can be slow, this function works in either exact or inexact mode, according to the *exact* parameter.

If *exact* is TRUE, the canvas items returned have been accurately tested for collision with the canvas item.

If *exact* is FALSE, the canvas items returned are *near* the canvas item. You can test the canvas items returned using collidesWith() if any are interesting collision candidates. By using this approach, you can ignore some canvas items for which collisions are not relevant.

The returned list is a list of QCanvasItems, but often you will need to cast the items to their subclass types. The safe way to do this is to use rtti() before casting. This provides some of the functionality of the standard C++ dynamic cast operation even on compilers where dynamic casts are not available.

Note that a canvas item may be 'on' a canvas, e.g. it was created with the canvas as parameter, even though its coordinates place it beyond the edge of the canvas's area. Collision detection only works for canvas items which are wholly or partly within the canvas's area.

void QCanvasItem::draw (QPainter & painter) [virtual]

This abstract virtual function draws the canvas item using *painter*.

Reimplemented in QCanvasSprite, QCanvasPolygonalItem and QCanvasText.

bool QCanvasItem::enabled () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use isEnabled() instead.

void QCanvasItem::hide ()

Shorthand for setVisible(FALSE).

bool QCanvasItem::isActive () const

Returns TRUE if the QCanvasItem is active; otherwise returns FALSE.

bool QCanvasItem::isEnabled () const

Returns TRUE if the QCanvasItem is enabled; otherwise returns FALSE.

bool QCanvasItem::isSelected () const

Returns TRUE if the canvas item is selected; otherwise returns FALSE.

bool QCanvasItem::isVisible () const

Returns TRUE if the canvas item is visible otherwise returns FALSE.

Note that in this context TRUE does *not* mean that the canvas item is currently in a view, merely that if a view is showing the area where the canvas item is positioned, and the item is not obscured by items with higher z values, and the view is not obscured by overlaying windows, it would be visible.

See also setVisible() [p. 42] and z() [p. 43].

void QCanvasItem::move (double x, double y)

Moves the canvas item to the absolute position (x, y).

void QCanvasItem::moveBy (double dx, double dy) [virtual]

Moves the canvas item relative to its current position by (dx, dy).

int QCanvasItem::rtti () const [virtual]

Returns 0 (QCanvasItem::Rtti_Item).

Although often frowned upon by purists, Run Time Type Identification is very useful in these classes as it allows a QCanvas to be an efficient indexed storage mechanism.

Make your derived classes return their own values for rtti(), so that you can distinguish between objects returned by QCanvas::at(). You should use values greater than 1000 to allow extensions to this class.

Overuse of this functionality can damage it's extensibility. For example, once you have identified a base class of a QCanvasItem found by QCanvas::at(), cast it to that type and call meaningful methods rather than acting upon the object based on its rtti value.

For example:

```
QCanvasItem* item;
// Find an item, eg. with QCanvasItem::collisions().
...
if (item->rtti() == MySprite::RTTI ) {
    MySprite* s = (MySprite*)item;
    if (s->isDamagable()) s->loseHitPoints(1000);
    if (s->isHot()) myself->loseHitPoints(1000);
    ...
}
```

Reimplemented in QCanvasSprite, QCanvasPolygonalItem and QCanvasText.

bool QCanvasItem::selected () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use isSelected() instead.

void QCanvasItem::setActive (bool yes) [virtual]

Sets the active flag of the item to *yes* and causes it to be redrawn when QCanvas::update() is next called.

The QCanvas, QCanvasItem and the Qt-supplied QCanvasItem subclasses do not make use of this value. The setActive() function is supplied because many applications need it, but it is up to you how you use the isActive() value.

void QCanvasItem::setAnimated (bool y) [virtual]

Sets the canvas item to be in motion if *y* is TRUE, or not if *y* is FALSE. The speed and direction of the motion is set with setVelocity(), or setXVelocity() and setYVelocity().

See also advance() [p. 38] and QCanvas::advance() [p. 27].

void QCanvasItem::setCanvas (QCanvas * c) [virtual]

Sets the QCanvas upon which the canvas item is to be drawn to *c*.

See also canvas() [p. 39].

void QCanvasItem::setEnabled (bool yes) [virtual]

Sets the enabled flag of the item to *yes* and causes it to be redrawn when QCanvas::update() is next called.

The QCanvas, QCanvasItem and the Qt-supplied QCanvasItem subclasses do not make use of this value. The setEnabled() function is supplied because many applications need it, but it is up to you how you use the isEnabled() value.

void QCanvasItem::setSelected (bool yes) [virtual]

Sets the selected flag of the item to *yes* and causes it to be redrawn when QCanvas::update() is next called.

The QCanvas, QCanvasItem and the Qt-supplied QCanvasItem subclasses do not make use of this value. The setSelected() function is supplied because many applications need it, but it is up to you how you use the isSelected() value.

void QCanvasItem::setVelocity (double vx, double vy) [virtual]

Sets the canvas item to be in motion, moving by *vx* and *vy* pixels in the horizontal and vertical directions respectively.

void QCanvasItem::setVisible (bool yes) [virtual]

Makes the canvas item visible if *yes* is TRUE, or invisible if *yes* is FALSE. The change takes effect when QCanvas::update() is next called.

void QCanvasItem::setX (double x)

Moves the canvas item so that its x-position is *x*.

void QCanvasItem::setXVelocity (double vx)

Sets the horizontal component of the canvas item's velocity to *vx*.

void QCanvasItem::setY (double y)

Moves the canvas item so that its y-position is *y*.

void QCanvasItem::setYVelocity (double vy)

Sets the vertical component of the canvas item's velocity to *vy*.

void QCanvasItem::setZ (double z)

Sets the z index of the canvas item to *z*. Higher-z items obscure (are in front of) lower-z items.

See also z() [p. 43] and move() [p. 40].

void QCanvasItem::show ()

Shorthand for setVisible(TRUE).

void QCanvasItem::update () [protected]

Call this function to repaint the canvas's changed chunks.

bool QCanvasItem::visible () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use isVisible() instead.

double QCanvasItem::x () const

Returns the horizontal position of the canvas item. Note that subclasses often have an origin other than the top-left corner.

double QCanvasItem::xVelocity () const

Returns the horizontal velocity component of the canvas item.

double QCanvasItem::y () const

Returns the vertical position of the canvas item. Note that subclasses often have an origin other than the top-left corner.

double QCanvasItem::yVelocity () const

Returns the vertical velocity component of the canvas item.

double QCanvasItem::z () const

Returns the z index of the canvas item, which is used for visual order: higher-z items obscure (are in front of) lower-z items.

QCanvasItemList Class Reference

The QCanvasItemList class is a list of QCanvasItems.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QValueList [Datastructures and String Handling with Qt]<QCanvasItem * >.

Detailed Description

The QCanvasItemList class is a list of QCanvasItems.

QCanvasItemList is a QValueList of pointers to QCanvasItems. This class is used by some methods in QCanvas that need to return a list of canvas items.

The QValueList documentation describes how to use this list.

See also Graphics Classes and Image Processing Classes.

QCanvasLine Class Reference

The QCanvasLine class provides a line on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasPolygonalItem [p. 54].

Public Members

- **QCanvasLine** (QCanvas * canvas)
- **~QCanvasLine** ()
- void **setPoints** (int xa, int ya, int xb, int yb)
- QPoint **startPoint** () const
- QPoint **endPoint** () const
- virtual int **rtti** () const

Detailed Description

The QCanvasLine class provides a line on a QCanvas.

The line inherits functionality from QCanvasPolygonalItem, for example the setPen() function. The start and end points of the line are set with setPoints().

Like any other canvas item lines can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasLine::QCanvasLine (QCanvas * canvas)

Constructs a line from (0,0) to (0,0) on *canvas*.

See also setPoints() [p. 46].

QCanvasLine::~QCanvasLine ()

Destroys the line.

QPoint QCanvasLine::endPoint () const

Returns the end point of the line.

See also setPoints() [p. 46] and startPoint() [p. 46].

int QCanvasLine::rtti () const [virtual]

Returns 7 (QCanvasItem::Rtti_Line).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasPolygonalItem [p. 56].

void QCanvasLine::setPoints (int xa, int ya, int xb, int yb)

Sets the line's start point to (*xa*, *ya*) and its end point to (*xb*, *yb*).

QPoint QCanvasLine::startPoint () const

Returns the start point of the line.

See also setPoints() [p. 46] and endPoint() [p. 46].

QCanvasPixmap Class Reference

The QCanvasPixmap class provides a pixmap in a QCanvas.

This class is part of the **canvas** module.

`#include <qcanvas.h>`

Inherits QPixmap [p. 225].

Public Members

- **QCanvasPixmap** (const QString & datafilename)
- **QCanvasPixmap** (const QImage & image)
- **QCanvasPixmap** (const QPixmap & pm, const QPoint & offset)
- **~QCanvasPixmap** ()
- int **offsetX** () const
- int **offsetY** () const
- void **setOffset** (int x, int y)

Detailed Description

The QCanvasPixmap class provides a pixmap in a QCanvas.

The pixmap is a QPixmap and can only be set in the constructor. There are three different constructors, one taking a QPixmap, one a QImage and one a file name that refers to a file in any supported file format (see QImageIO).

QCanvasPixmap can have a hotspot which is defined in terms of an (x, y) offset. When you create a QCanvasPixmap from a PNG file or from a QImage that has a QImage::offset(), the offset() is initialized appropriately, otherwise the constructor leaves it at (0, 0). You can set it later using setOffset(). When the QCanvasPixmap is used in a QCanvasSprite, the offset position is the point at QCanvasItem::x() and QCanvasItem::y(), not the top-left corner of the pixmap.

Note that for QCanvasPixmap objects created by a QCanvasSprite, the position of each QCanvasPixmap object is set so that the hotspot stays in the same position.

Like any other canvas item canvas pixmaps can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also QCanvasPixmapArray [p. 49], QCanvasItem [p. 36], QCanvasSprite [p. 63], Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasPixmap::QCanvasPixmap (const QString & datafilename)

Constructs a QCanvasPixmap that uses the image stored in *datafilename*.

QCanvasPixmap::QCanvasPixmap (const QImage & image)

Constructs a QCanvasPixmap from the image *image*.

QCanvasPixmap::QCanvasPixmap (const QPixmap & pm, const QPoint & offset)

Constructs a QCanvasPixmap from the pixmap *pm* using the offset *offset*.

QCanvasPixmap::~QCanvasPixmap ()

Destroys the pixmap.

int QCanvasPixmap::offsetX () const

Returns the X-offset of the pixmap's hotspot.
See also setOffset() [p. 48].

int QCanvasPixmap::offsetY () const

Returns the Y-offset of the pixmap's hotspot.
See also setOffset() [p. 48].

void QCanvasPixmap::setOffset (int x, int y)

Sets the offset of the pixmap's hotspot to (x, y).

Note that you must not call this function if any QCanvasSprites are currently showing this pixmap.

QCanvasPixmapArray Class Reference

The QCanvasPixmapArray class provides an array of QCanvasPixmaps.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Public Members

- **QCanvasPixmapArray** ()
- **QCanvasPixmapArray** (const QString & datafilenamepattern, int fc = 0)
- QCanvasPixmapArray (QPtrList<QPixmap> list, QPtrList<QPoint> hotspots) *(obsolete)*
- **QCanvasPixmapArray** (QValueList<QPixmap> list, QPointArray hotspots = QPointArray ( ))
- **~QCanvasPixmapArray** ()
- bool **readPixmaps** (const QString & filenamepattern, int fc = 0)
- bool **readCollisionMasks** (const QString & filename)
- bool operator! () *(obsolete)*
- bool **isValid** () const
- QCanvasPixmap * **image** (int i) const
- void **setImage** (int i, QCanvasPixmap * p)
- uint **count** () const

Detailed Description

The QCanvasPixmapArray class provides an array of QCanvasPixmaps.

This class is used by QCanvasSprite to hold an array of pixmaps. It is used to implement animated sprites, i.e. images that change over time, with each pixmap in the array holding one frame.

Depending on the constructor you use you can load multiple pixmaps into the array, either from a directory (specifying a wildcard pattern for the files), or from a list of QPixmaps. You can also read in a set of pixmaps after construction using readPixmaps().

Individual pixmaps can be set with setImage() and retrieved with image(). The number of pixmaps in the array is returned by count().

QCanvasSprite uses an image's mask for collision detection. You can change this by reading in a separate set of image masks using readCollisionMasks().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasPixmapArray::QCanvasPixmapArray ()

Constructs an invalid array (i.e. isValid() will return FALSE). You will need to call readPixmaps() before being able to use it further.

QCanvasPixmapArray::QCanvasPixmapArray (const QString & datafilenamepattern, int fc = 0)

Constructs a QCanvasPixmapArray from files.

The *fc* parameter sets the number of frames to be loaded for this image.

If *fc* is not 0, *datafilenamepattern* should contain "%1", e.g. "foo%1.png". The actual filenames are formed by replacing the %1 with four-digit integers from 0 to (fc - 1), e.g. foo0000.png, foo0001.png, foo0002.png, etc.

If *fc* is 0, *datafilenamepattern* is asssumed to be a filename, and the image contained in this file will be loaded as the first (and only) frame.

If *datafilenamepattern* does not exist, is not readable, isn't an image, or some other error occurs, the array ends up empty and isValid() returns FALSE.

QCanvasPixmapArray::QCanvasPixmapArray (QPtrList<QPixmap> list, QPtrList<QPoint> hotspots)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use QCanvasPixmapArray::QCanvasPixmapArray(QValueList, QPointArray) instead.

Constructs a QCanvasPixmapArray from the list of QPixmaps *list*. The *hotspots* list has to be of the same size as *list*.

QCanvasPixmapArray::QCanvasPixmapArray (QValueList<QPixmap> list, QPointArray hotspots = QPointArray ( ))

Constructs a QCanvasPixmapArray from the list of QPixmaps in the *list*. Each pixmap will get a hotspot according to the *hotspots* array. If no hotspots are specified, each one is set to be at position (0, 0).

If an error occurs, isValid() will return FALSE.

QCanvasPixmapArray::~QCanvasPixmapArray ()

Destroys the pixmap array and all the pixmaps it contains.

uint QCanvasPixmapArray::count () const

Returns the number of pixmaps in the array.

QCanvasPixmap * QCanvasPixmapArray::image (int i) const

Returns pixmap *i* in the array, if *i* is nonnegative and smaller than count(), and returns an unspecified value otherwise.

bool QCanvasPixmapArray::isValid () const

returns TRUE if the pixmap array is valid; otherwise returns FALSE.

bool QCanvasPixmapArray::operator! ()

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use isValid() instead.

This returns FALSE if the array is valid, and TRUE if it is not.

bool QCanvasPixmapArray::readCollisionMasks (const QString & filename)

Reads new collision masks for the array.

By default, QCanvasSprite uses the image mask of a sprite to detect collisions. Use this function to set your own collision image masks.

If count() is 1 *filename* must specify a real filename to read the mask from. If count() is greater than 1, the *filename* must contain a "%1" that will get replaced by the number of the mask to be loaded, similar to QCanvasPixmapArray::readPixmaps().

All collision masks must be 1-bit images or this function call will fail.

If the file isn't readable, contains the wrong number of images, or there is some other error, this function will return FALSE, and the array will be flagged as invalid.

See also isValid() [p. 51].

bool QCanvasPixmapArray::readPixmaps (const QString & filenamepattern, int fc = 0)

Reads one or more pixmaps into the pixmap array.

If *fc* is not 0, *filenamepattern* should contain "%1", e.g. "foo%1.png". The actual filenames are formed by replacing the %1 with four-digit integers from 0 to (fc - 1), e.g. foo0000.png, foo0001.png, foo0002.png, etc.

If *fc* is 0, *filenamepattern* is asssumed to be a filename, and the image contained in this file will be loaded as the first (and only) frame.

If *filenamepattern* does not exist, is not readable, isn't an image, or some other error occurs, this function will return FALSE, and isValid() will return FALSE.

See also isValid() [p. 51].

void QCanvasPixmapArray::setImage (int i, QCanvasPixmap * p)

Replaces the pixmap at index *i* with pixmap *p*.

The array takes ownership of *p* and will delete *p* when the array itself is deleted.

If *i* is beyond the end of the array the array is extended to at least i+1 elements, with elements count() to i-1 being initialized to 0.

QCanvasPolygon Class Reference

The QCanvasPolygon class provides a polygon on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasPolygonalItem [p. 54].

Inherited by QCanvasSpline [p. 61].

Public Members

- **QCanvasPolygon** (QCanvas * canvas)
- **~QCanvasPolygon** ()
- void **setPoints** (QPointArray pa)
- QPointArray **points** () const
- virtual QPointArray **areaPoints** () const
- virtual int **rtti** () const

Protected Members

- virtual void **drawShape** (QPainter & p)

Detailed Description

The QCanvasPolygon class provides a polygon on a QCanvas.

Paints a polygon with a QBrush. The polygon's points can be set in the constructor or set or changed later using setPoints(). Use points() to retrieve the points, or areaPoints() to retrieve the points relative to the canvas's origin.

The polygon can be drawn on a painter with drawShape().

Like any other canvas item polygons can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasPolygon::QCanvasPolygon (QCanvas * canvas)

Constructs a point-less polygon on the canvas *canvas*. You should call setPoints() before using it further.

QCanvasPolygon::~QCanvasPolygon ()

Destroys the polygon.

QPointArray QCanvasPolygon::areaPoints () const [virtual]

Returns the vertices of the polygon translated by the polygon's current x(), y() position, i.e. relative to the canvas's origin.

See also setPoints() [p. 53] and points() [p. 53].

Reimplemented from QCanvasPolygonalItem [p. 55].

void QCanvasPolygon::drawShape (QPainter & p) [virtual protected]

Draws the polygon using the painter *p*.

Note that QCanvasPolygon does not support an outline (pen is always NoPen).

Reimplemented from QCanvasPolygonalItem [p. 56].

QPointArray QCanvasPolygon::points () const

Returns the vertices of the polygon, not translated by the position.

See also setPoints() [p. 53] and areaPoints() [p. 53].

int QCanvasPolygon::rtti () const [virtual]

Returns 4 (QCanvasItem::Rtti_Polygon).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasPolygonalItem [p. 56].

Reimplemented in QCanvasSpline.

void QCanvasPolygon::setPoints (QPointArray pa)

Sets the points of the polygon to be *pa*. These points will have their x and y coordinates automatically translated by x(), y() as the polygon is moved.

QCanvasPolygonalItem Class Reference

The QCanvasPolygonalItem class provides a polygonal canvas item on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasItem [p. 36].

Inherited by QCanvasRectangle [p. 58], QCanvasPolygon [p. 52], QCanvasLine [p. 45] and QCanvasEllipse [p. 33].

Public Members

- **QCanvasPolygonalItem** (QCanvas * canvas)
- virtual **~QCanvasPolygonalItem** ()
- virtual void **setPen** (QPen p)
- virtual void **setBrush** (QBrush b)
- QPen **pen** () const
- QBrush **brush** () const
- virtual QPointArray **areaPoints** () const
- virtual QPointArray **areaPointsAdvanced** () const
- virtual QRect **boundingRect** () const
- virtual int **rtti** () const

Protected Members

- virtual void **draw** (QPainter & p)
- virtual void **drawShape** (QPainter & p)
- bool **winding** () const
- void **setWinding** (bool enable)

Detailed Description

The QCanvasPolygonalItem class provides a polygonal canvas item on a QCanvas.

The mostly rectangular classes, such as QCanvasSprite and QCanvasText, use the object's bounding rectangle for movement, repainting and collision calculation. However, for most other items, the bounding rectangle can be far too large — a diagonal line being the worst case, but there are many other cases that are very bad. QCanvasPolygonalItem provides polygon-based bounding rectangle handling, etc., much speeding up such cases.

Derived classes should try to define as small an area as possible to maximize efficiency, but the polygon must *definitely* be contained completely within the polygonal area. Calculating the exact requirements is usually difficult, but if you allow a small overestimate it can be easy and quick, while still getting almost all of QCanvasPolygonalItem's speed.

Note that all subclasses *must* call hide() in their destructor since hide() needs to be able to access areaPoints().

Normally, QCanvasPolygonalItem uses the odd-even algorithm for determining whether an object intersects this object. You can change this to the winding algorithm using setWinding().

The bounding rectangle is available using boundingRect(). The points bounding the polygonal item are retrieved with areaPoints(). Use areaPointsAdvanced() to retrieve the bounding points the polygonal item *will* have after QCanvasItem::advance(1) has been called.

By default, QCanvasPolygonalItem objects have a black pen and no brush (the default QPen and QBrush constructors). You can change this with setPen() and setBrush(), but note that some QCanvasPolygonalItem subclasses only use the brush, ignoring the pen setting.

The polygonal item can be drawn on a painter with draw(). Subclasses must reimplement drawShape() to draw themselves.

Like any other canvas item polygonal items can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasPolygonalItem::QCanvasPolygonalItem (QCanvas * canvas)

Constructs a QCanvasPolygonalItem on the canvas *canvas*.

QCanvasPolygonalItem::~QCanvasPolygonalItem () [virtual]

Note that all subclasses *must* call hide() in their destructor since hide() needs to be able to access areaPoints().

QPointArray QCanvasPolygonalItem::areaPoints () const [virtual]

Returns the points bounding the shape. Note that the returned points are *outside* the object, not touching it.

Reimplemented in QCanvasPolygon.

QPointArray QCanvasPolygonalItem::areaPointsAdvanced () const [virtual]

Returns the points the polygonal item *will* have after QCanvasItem::advance(1) is called, i.e. what the points are when advanced by the current xVelocity() and yVelocity().

QRect QCanvasPolygonalItem::boundingRect () const [virtual]

Returns the bounding rectangle of the polygonal item, based on areaPoints().

Reimplemented from QCanvasItem [p. 39].

QBrush QCanvasPolygonalItem::brush () const

Returns the QBrush used to fill the item, if filled.

See also setBrush() [p. 56].

void QCanvasPolygonalItem::draw (QPainter & p) [virtual protected]

Reimplemented from QCanvasItem [p. 36], this draws the polygonal item by setting the pen and brush for the item on the painter *p* and calling drawShape() [p. 56].

Reimplemented from QCanvasItem [p. 40].

void QCanvasPolygonalItem::drawShape (QPainter & p) [virtual protected]

Subclasses must reimplement this function to draw their shape. The pen and brush of *p* are already set to pen() and brush() prior to calling this function.

See also draw() [p. 56].

Reimplemented in QCanvasRectangle, QCanvasPolygon and QCanvasEllipse.

QPen QCanvasPolygonalItem::pen () const

Returns the QPen used to draw the outline of the item, if any.

See also setPen() [p. 56].

int QCanvasPolygonalItem::rtti () const [virtual]

Returns 2 (QCanvasItem::Rtti_PolygonalItem).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasItem [p. 41].

Reimplemented in QCanvasRectangle, QCanvasPolygon, QCanvasLine and QCanvasEllipse.

void QCanvasPolygonalItem::setBrush (QBrush b) [virtual]

Sets the QBrush used when drawing the polygonal item to the brush *b*.

See also setPen() [p. 56], brush() [p. 56] and drawShape() [p. 56].

void QCanvasPolygonalItem::setPen (QPen p) [virtual]

Sets the QPen used when drawing the item to the pen *p*. Note that many QCanvasPolygonalItems do not use the pen value.

See also setBrush() [p. 56], pen() [p. 56] and drawShape() [p. 56].

void QCanvasPolygonalItem::setWinding (bool enable) [protected]

If *enable* is TRUE, the polygonal item will use the winding algorithm to determine the "inside" of the polygon; otherwise the odd-even algorithm will be used.

The default is to use the odd-even algorithm.

See also winding() [p. 57].

bool QCanvasPolygonalItem::winding () const [protected]

Returns TRUE if the polygonal item uses the winding algorithm to determine the "inside" of the polygon. Returns FALSE if it uses the odd-even algorithm.

The default is to use the odd-even algorithm.

See also setWinding() [p. 57].

QCanvasRectangle Class Reference

The QCanvasRectangle class provides a rectangle on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasPolygonalItem [p. 54].

Public Members

- **QCanvasRectangle** (QCanvas * canvas)
- **QCanvasRectangle** (const QRect & r, QCanvas * canvas)
- **QCanvasRectangle** (int x, int y, int width, int height, QCanvas * canvas)
- **~QCanvasRectangle** ()
- int **width** () const
- int **height** () const
- void **setSize** (int width, int height)
- QSize **size** () const
- QRect **rect** () const
- virtual int **rtti** () const

Protected Members

- virtual void **drawShape** (QPainter & p)
- virtual QPointArray **chunks** () const

Detailed Description

The QCanvasRectangle class provides a rectangle on a QCanvas.

This item paints a single rectangle which may have any pen() and brush(), but may not be tilted/rotated. For rotated rectangles, use QCanvasPolygon.

The rectangle's size and initial position can be set in the constructor. The size can set or changed later using setSize(). Use height() and width() to retrieve the rectangle's dimensions.

The rectangle can be drawn on a painter with drawShape().

Like any other canvas item rectangles can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasRectangle::QCanvasRectangle (QCanvas * canvas)

Constructs a rectangle at position (0,0) with both width and height set to 32 pixels on *canvas*.

QCanvasRectangle::QCanvasRectangle (const QRect & r, QCanvas * canvas)

Constructs a rectangle positioned and sized by *r* on *canvas*.

QCanvasRectangle::QCanvasRectangle (int x, int y, int width, int height, QCanvas * canvas)

Constructs a rectangle at position (x, y) and size *width* by *height*, on *canvas*.

QCanvasRectangle::~QCanvasRectangle ()

Destroys the rectangle.

QPointArray QCanvasRectangle::chunks () const [virtual protected]

Simply calls QCanvasItem::chunks().

void QCanvasRectangle::drawShape (QPainter & p) [virtual protected]

Draws the rectangle on painter *p*.

Reimplemented from QCanvasPolygonalItem [p. 56].

int QCanvasRectangle::height () const

Returns the height of the rectangle.

QRect QCanvasRectangle::rect () const

Returns the integer-converted x(), y() position and size() of the rectangle as a QRect.

int QCanvasRectangle::rtti () const [virtual]

Returns 5 (QCanvasItem::Rtti_Rectangle).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasPolygonalItem [p. 56].

void QCanvasRectangle::setSize (int width, int height)

Sets the *width* and *height* of the rectangle.

QSize QCanvasRectangle::size () const

Returns the width() and height() of the rectangle.

See also rect() [p. 59] and setSize() [p. 59].

int QCanvasRectangle::width () const

Returns the width of the rectangle.

QCanvasSpline Class Reference

The QCanvasSpline class provides multi-bezier splines on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasPolygon [p. 52].

Public Members

- **QCanvasSpline** (QCanvas * canvas)
- **~QCanvasSpline** ()
- void **setControlPoints** (QPointArray ctrl, bool close = TRUE)
- QPointArray **controlPoints** () const
- bool **closed** () const
- virtual int **rtti** () const

Detailed Description

The QCanvasSpline class provides multi-bezier splines on a QCanvas.

A QCanvasSpline is a sequence of 4-point bezier curves joined together to make a curved shape.

You set the control points of the spline with setControlPoints().

If the bezier is closed(), then the first control point will be re-used as the last control point. Therefore, a closed bezier must have a multiple of 3 control points and an open bezier must have one extra point.

The beziers are not necessarily joined "smoothly". To ensure this, set control points appropriately (general references about beziers will explain this in detail).

Like any other canvas item splines can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasSpline::QCanvasSpline (QCanvas * canvas)

Create a spline with no control points on the canvas *canvas*.

See also setControlPoints() [p. 62].

QCanvasSpline::~QCanvasSpline ()

Destroy the spline.

bool QCanvasSpline::closed () const

Returns whether the control points are considered a closed set.

QPointArray QCanvasSpline::controlPoints () const

Returns the current set of control points.

See also setControlPoints() [p. 62] and closed() [p. 62].

int QCanvasSpline::rtti () const [virtual]

Returns 8 (QCanvasItem::Rtti_Spline).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasPolygon [p. 53].

void QCanvasSpline::setControlPoints (QPointArray ctrl, bool close = TRUE)

Set the spline control points to *ctrl*.

If *close* is TRUE, then the first point in *ctrl* will be re-used as the last point, and the number of control points must be a multiple of 3. If *close* is FALSE, one additional control point is required, and the number of control points must be one of (4, 7, 11, …).

If the number of control points doesn't meet the above conditions, the number of points will be truncated to the largest number of points that do meet the requirement.

QCanvasSprite Class Reference

The QCanvasSprite class provides an animated canvas item on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasItem [p. 36].

Public Members

- **QCanvasSprite** (QCanvasPixmapArray * a, QCanvas * canvas)
- void **setSequence** (QCanvasPixmapArray * a)
- virtual **~QCanvasSprite** ()
- virtual void **move** (double nx, double ny, int nf)
- void **setFrame** (int f)
- enum **FrameAnimationType** { Cycle, Oscillate }
- virtual void **setFrameAnimation** (FrameAnimationType type = Cycle, int step = 1, int state = 0)
- int **frame** () const
- int **frameCount** () const
- virtual int **rtti** () const
- virtual QRect **boundingRect** () const
- int **width** () const
- int **height** () const
- int **leftEdge** () const
- int **topEdge** () const
- int **rightEdge** () const
- int **bottomEdge** () const
- int **leftEdge** (int nx) const
- int **topEdge** (int ny) const
- int **rightEdge** (int nx) const
- int **bottomEdge** (int ny) const
- QCanvasPixmap * **image** () const
- virtual QCanvasPixmap * **imageAdvanced** () const
- QCanvasPixmap * **image** (int f) const
- virtual void **advance** (int phase)
- virtual void **draw** (QPainter & painter)

Detailed Description

The QCanvasSprite class provides an animated canvas item on a QCanvas.

A canvas sprite is an object which contains any number of images (referred to as frames), only one of which is current, e.g. displayed, at any one time. The images can be passed in the constructor or set or changed later with setSequence(). If you subclass QCanvasSprite you can change the frame that is displayed periodically, e.g. whenever QCanvasItem::advance(1) is called to create the effect of animation.

The current frame can be set with setFrame() or with move(). The number of frames available is given by frame-Count(). The bounding rectangle of the current frame is returned by boundingRect().

The current frame's image can be retrieved with image(); use imageAdvanced() to retrieve the image for the frame that will be shown after advance(1) is called. Use the image() overload passing it an integer index to retrieve a particular image from the list of frames.

Use width() and height() to retrieve the dimensions of the current frame.

Use leftEdge() and rightEdge() to retrieve the current frame's left-hand and right-hand x coordinates respectively. Use bottomEdge() and topEdge() to retrieve the current frame's bottom and top y coordinates respectively. These functions have an overload which will accept an integer frame number to retrieve the coordinates of a particular frame.

QCanvasSprite draws very quickly, at the cost of some memory.

The current frame's image can be drawn on a painter with draw().

Like any other canvas item canvas sprites can be moved with move() which sets the x and y coordinates and the frame number, as well as with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Type Documentation

QCanvasSprite::FrameAnimationType

This enum is used to identify the different types of frame animation of QCanvasSprite.

- `QCanvasSprite::Cycle` - at each advance the frame number will be incremented by 1 (modulo the frame count).
- `QCanvasSprite::Oscillate` - at each advance the frame number will be incremented by 1 up to the frame count then decremented to by 1 to 0, repeating this sequence forever.

Member Function Documentation

QCanvasSprite::QCanvasSprite (QCanvasPixmapArray * a, QCanvas * canvas)

Constructs a QCanvasSprite which uses images from the QCanvasPixmapArray *a*.

The sprite in initially positioned at (0,0) on *canvas*, using frame 0.

QCanvasSprite::~QCanvasSprite () [virtual]

Destroys the sprite and removes it from the canvas. Does *not* delete the images.

void QCanvasSprite::advance (int phase) [virtual]

Extends the default QCanvasItem implementation to provide the functionality of setFrameAnimation().

The *phase* is 0 or 1: see QCanvas::animate() for details.

See also QCanvasItem::advance() [p. 38] and setVelocity() [p. 42].

Reimplemented from QCanvasItem [p. 38].

int QCanvasSprite::bottomEdge () const

Returns the y coordinate of the current bottom edge of the sprite. (This may change as the sprite animates since different frames may have different bottom edges.)

See also leftEdge() [p. 66], rightEdge() [p. 66] and topEdge() [p. 67].

int QCanvasSprite::bottomEdge (int ny) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns what the y coordinate of the top edge of the sprite would be if the sprite (actually its hotspot) were moved to y-position *ny*.

See also leftEdge() [p. 66], rightEdge() [p. 66] and topEdge() [p. 67].

QRect QCanvasSprite::boundingRect () const [virtual]

Returns the bounding rectangle for the image in sprite's current frame. This assumes that the images are tightly cropped (ie. do not have transparent pixels all along a side).

Reimplemented from QCanvasItem [p. 39].

void QCanvasSprite::draw (QPainter & painter) [virtual]

Draws the current frame's image at the sprite's current position on painter *painter*.

Reimplemented from QCanvasItem [p. 40].

int QCanvasSprite::frame () const

Returns the index of the current animation frame in the QCanvasSprite's QCanvasPixmapArray.

See also setFrame() [p. 67] and move() [p. 66].

int QCanvasSprite::frameCount () const

Returns the number of frames in the QCanvasSprite's QCanvasPixmapArray.

int QCanvasSprite::height () const

The height of the sprite for the current frame's image.

See also frame() [p. 65].

QCanvasPixmap * QCanvasSprite::image () const

Returns the current frame's image.

See also frame() [p. 65] and setFrame() [p. 67].

QCanvasPixmap * QCanvasSprite::image (int f) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the image for frame *f*. Does not do any bounds checking on *f*.

QCanvasPixmap * QCanvasSprite::imageAdvanced () const [virtual]

Returns the image the sprite *will* have after advance(1) is called. By default this is the same as image().

int QCanvasSprite::leftEdge () const

Returns the x coordinate of the current left edge of the sprite. (This may change as the sprite animates since different frames may have different left edges.)

See also rightEdge() [p. 66], bottomEdge() [p. 65] and topEdge() [p. 67].

int QCanvasSprite::leftEdge (int nx) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns what the x coordinate of the left edge of the sprite would be if the sprite (actually its hotspot) were moved to x-position *nx*.

See also rightEdge() [p. 66], bottomEdge() [p. 65] and topEdge() [p. 67].

void QCanvasSprite::move (double nx, double ny, int nf) [virtual]

Set the position of the sprite to *nx*, *ny* and the current frame to *nf*. *nf* will be ignored if it is larger than frameCount() or smaller than 0.

int QCanvasSprite::rightEdge () const

Returns the x coordinate of the current right edge of the sprite. (This may change as the sprite animates since different frames may have different right edges.)

See also leftEdge() [p. 66], bottomEdge() [p. 65] and topEdge() [p. 67].

int QCanvasSprite::rightEdge (int nx) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns what the x coordinate of the right edge of the sprite would be if the sprite (actually its hotspot) were moved to x-position *nx*.

See also leftEdge() [p. 66], bottomEdge() [p. 65] and topEdge() [p. 67].

int QCanvasSprite::rtti () const [virtual]

Returns 1 (QCanvasItem::Rtti_Sprite).

See also QCanvasItem::rtti() [p. 41].

Reimplemented from QCanvasItem [p. 41].

void QCanvasSprite::setFrame (int f)

Sets the animation frame used for displaying the sprite to *f*, an index into the QCanvasSprite's QCanvasPixmapArray. The call will be ignored if *f* is larger than frameCount() or smaller than 0.

See also frame() [p. 65] and move() [p. 66].

void QCanvasSprite::setFrameAnimation (FrameAnimationType type = Cycle, int step = 1, int state = 0) [virtual]

Sets the animation characteristics for the sprite.

For *type* == Cycle, the frames will increase by *step* at each advance, modulo the frameCount().

For *type* == Oscillate, the frames will increase by *step* at each advance, up to the frameCount(), then decrease by *step* back to 0, etc.

The *state* parameter is for internal use.

void QCanvasSprite::setSequence (QCanvasPixmapArray * a)

Set the array of images used for displaying the sprite to the QCanvasPixmapArray *a*.

If the current frame() is larger than the number of images in *a*, the current frame will be reset to 0.

int QCanvasSprite::topEdge () const

Returns the y coordinate of the top edge of the sprite. (This may change as the sprite animates since different frames may have different top edges.)

See also leftEdge() [p. 66], rightEdge() [p. 66] and bottomEdge() [p. 65].

int QCanvasSprite::topEdge (int ny) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns what the y coordinate of the top edge of the sprite would be if the sprite (actually its hotspot) were moved to y-position *ny*.

See also leftEdge() [p. 66], rightEdge() [p. 66] and bottomEdge() [p. 65].

int QCanvasSprite::width () const

The width of the sprite for the current frame's image.

See also frame() [p. 65].

QCanvasText Class Reference

The QCanvasText class provides a text object on a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QCanvasItem [p. 36].

Public Members

- **QCanvasText** (QCanvas * canvas)
- **QCanvasText** (const QString & t, QCanvas * canvas)
- **QCanvasText** (const QString & t, QFont f, QCanvas * canvas)
- virtual **~QCanvasText** ()
- void **setText** (const QString & t)
- void **setFont** (const QFont & f)
- void **setColor** (const QColor & c)
- QString **text** () const
- QFont **font** () const
- QColor **color** () const
- int **textFlags** () const
- void **setTextFlags** (int f)
- virtual QRect **boundingRect** () const
- virtual int **rtti** () const

Protected Members

- virtual void **draw** (QPainter & painter)

Detailed Description

The QCanvasText class provides a text object on a QCanvas.

A canvas text item has text with font, color and alignment attributes. The text and font can be set in the constructor or set or changed later with setText() and setFont(). The color is set with setColor() and the alignment with setTextFlags(). The text item's bounding rectangle is retrieved with boundingRect().

The text can be drawn on a painter with draw().

Like any other canvas item text items can be moved with QCanvasItem::move() and QCanvasItem::moveBy(), or by setting coordinates with QCanvasItem::setX(), QCanvasItem::setY() and QCanvasItem::setZ().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasText::QCanvasText (QCanvas * canvas)

Constructs a QCanvasText with the text "\", on *canvas*.

QCanvasText::QCanvasText (const QString & t, QCanvas * canvas)

Constructs a QCanvasText with the text *t*, on canvas *canvas*.

QCanvasText::QCanvasText (const QString & t, QFont f, QCanvas * canvas)

Constructs a QCanvasText with the text *t* and font *f*, on the canvas *canvas*.

QCanvasText::~QCanvasText () [virtual]

Destroys the canvas text.

QRect QCanvasText::boundingRect () const [virtual]

Returns the bounding rectangle of the text.
Reimplemented from QCanvasItem [p. 39].

QColor QCanvasText::color () const

Returns the color of the text.
See also setColor() [p. 70].

void QCanvasText::draw (QPainter & painter) [virtual protected]

Draws the text using the painter *painter*.
Reimplemented from QCanvasItem [p. 40].

QFont QCanvasText::font () const

Returns the font in which the text is drawn.
See also setFont() [p. 70].

int QCanvasText::rtti () const [virtual]

Returns 3 (QCanvasItem::Rtti_Text).
See also QCanvasItem::rtti() [p. 41].
Reimplemented from QCanvasItem [p. 41].

void QCanvasText::setColor (const QColor & c)

Sets the color of the text to the color *c*.

See also color() [p. 69] and setFont() [p. 70].

void QCanvasText::setFont (const QFont & f)

Sets the font in which the text is drawn to font *f*.

See also font() [p. 69].

void QCanvasText::setText (const QString & t)

Sets the text item's text to *t*. The text may contain newlines.

See also text() [p. 70], setFont() [p. 70], setColor() [p. 70] and setTextFlags() [p. 70].

void QCanvasText::setTextFlags (int f)

Sets the alignment flags to *f*. These are a bitwise OR of the flags available to QPainter::drawText() — see Qt::AlignmentFlags.

See also setFont() [p. 70] and setColor() [p. 70].

QString QCanvasText::text () const

Returns the text item's text.

See also setText() [p. 70].

int QCanvasText::textFlags () const

Returns the currently set alignment flags.

See also setTextFlags() [p. 70] and Qt::AlignmentFlags [Additional Functionality with Qt].

QCanvasView Class Reference

The QCanvasView class provides an on-screen view of a QCanvas.

This class is part of the **canvas** module.

```
#include <qcanvas.h>
```

Inherits QScrollView [Widgets with Qt].

Public Members

- **QCanvasView** (QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **QCanvasView** (QCanvas * canvas, QWidget * parent = 0, const char * name = 0, WFlags f = 0)
- **~QCanvasView** ()
- QCanvas * **canvas** () const
- void **setCanvas** (QCanvas * canvas)
- const QWMatrix & **worldMatrix** () const
- const QWMatrix & **inverseWorldMatrix** () const
- bool **setWorldMatrix** (const QWMatrix & wm)

Protected Members

- virtual void **drawContents** (QPainter * p, int cx, int cy, int cw, int ch)

Detailed Description

The QCanvasView class provides an on-screen view of a QCanvas.

A QCanvasView is widget which provides a view of a QCanvas.

If you want users to be able to interact with a canvas view, subclass QCanvasView. You might then reimplement QScrollView::contentsMousePressEvent() for example:

```
void MyCanvasView::contentsMousePressEvent( QMouseEvent* e )
{
    QCanvasItemList l = canvas()->collisions(e->pos());
    for (QCanvasItemList::Iterator it=l.begin(); it!=l.end(); ++it) {
        if ( (*it)->rtti() == QCanvasRectangle::RTTI )
            qDebug("A QCanvasRectangle lies somewhere at this point");
    }
}
```

Set the canvas that the view shows with setCanvas() and retrieve the canvas which the view is showing with canvas().

A transformation matrix can be used to transform the view of the canvas in various ways, for example, zooming in or out or rotating. For example:

```
QWMatrix wm;
wm.scale( 2, 2 );   // Zooms in by 2 times
wm.rotate( 90 );    // Rotates 90 degrees clockwise
myCanvasView->setWorldMatrix( wm );
```

Use setWorldMatrix() to set the canvas view's world matrix: you must ensure that the world matrix is invertible. The current world matrix is retrievable with worldMatrix(), and its inversion is retrievable with inverseWorldMatrix().

See also QWMatrix [p. 291], QPainter::setWorldMatrix() [p. 203], Graphics Classes and Image Processing Classes.

Member Function Documentation

QCanvasView::QCanvasView (QWidget * parent = 0, const char * name = 0, WFlags f = 0)

Constructs a QCanvasView with parent *parent*, and name *name*, using the widget flags *f*. The canvas view is not associated with a canvas, so you will need to call setCanvas() to display a canvas.

QCanvasView::QCanvasView (QCanvas * canvas, QWidget * parent = 0, const char * name = 0, WFlags f = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Constructs a QCanvasView which views canvas *canvas*, with parent *parent*, and name *name*, using the widget flags *f*.

QCanvasView::~QCanvasView ()

Destroys the canvas view. The associated canvas is *not* deleted.

QCanvas * QCanvasView::canvas () const

Returns a pointer to the canvas which the QCanvasView is currently showing.

void QCanvasView::drawContents (QPainter * p, int cx, int cy, int cw, int ch) [virtual protected]

Repaints part of the QCanvas that the canvas view is showing starting at *cx* by *cy*, with a width of *cw* and a height of *ch* using the painter *p*.

Reimplemented from QScrollView [Widgets with Qt].

const QWMatrix & QCanvasView::inverseWorldMatrix () const

Returns a reference to the inverse of the canvas view's current transformation matrix.

See also setWorldMatrix() [p. 73] and worldMatrix() [p. 73].

void QCanvasView::setCanvas (QCanvas * canvas)

Sets the canvas that the QCanvasView is showing to the canvas *canvas*.

bool QCanvasView::setWorldMatrix (const QWMatrix & wm)

Sets the transformation matrix of the QCanvasView to *wm*. The matrix must be invertible (i.e. if you create a world matrix that zooms out by 2 times, then the inverse of this matrix is one that will zoom in by 2 times).

When you use this, you should note that the performance of the QCanvasView will decrease considerably.

Returns FALSE if *wm* is not invertable; otherwise returns TRUE.

See also worldMatrix() [p. 73], inverseWorldMatrix() [p. 73] and QWMatrix::isInvertible() [p. 294].

const QWMatrix & QCanvasView::worldMatrix () const

Returns a reference to the canvas view's current transformation matrix.

See also setWorldMatrix() [p. 73] and inverseWorldMatrix() [p. 73].

QColor Class Reference

The QColor class provides colors based on RGB.

```
#include <qcolor.h>
```

Public Members

- enum **Spec** { Rgb, Hsv }
- **QColor** ()
- **QColor** (int r, int g, int b)
- **QColor** (int x, int y, int z, Spec colorSpec)
- **QColor** (QRgb rgb, uint pixel = 0xffffffff)
- **QColor** (const QString & name)
- **QColor** (const char * name)
- **QColor** (const QColor & c)
- QColor & **operator=** (const QColor & c)
- bool **isValid** () const
- QString **name** () const
- void **setNamedColor** (const QString & name)
- void **rgb** (int * r, int * g, int * b) const
- QRgb **rgb** () const
- void **setRgb** (int r, int g, int b)
- void **setRgb** (QRgb rgb)
- int **red** () const
- int **green** () const
- int **blue** () const
- void **hsv** (int * h, int * s, int * v) const
- void getHsv (int & h, int & s, int & v) const *(obsolete)*
- void **setHsv** (int h, int s, int v)
- QColor **light** (int factor = 150) const
- QColor **dark** (int factor = 200) const
- bool **operator==** (const QColor & c) const
- bool **operator!=** (const QColor & c) const
- uint **alloc** ()
- uint **pixel** () const

Static Public Members

- int **maxColors** ()
- int **numBitPlanes** ()
- int **enterAllocContext** ()
- void **leaveAllocContext** ()
- int **currentAllocContext** ()
- void **destroyAllocContext** (int context)
- void **initialize** ()
- void **cleanup** ()

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QColor & c)
- QDataStream & **operator>>** (QDataStream & s, QColor & c)
- int **qRed** (QRgb rgb)
- int **qGreen** (QRgb rgb)
- int **qBlue** (QRgb rgb)
- int **qAlpha** (QRgb rgba)
- QRgb **qRgb** (int r, int g, int b)
- QRgb **qRgba** (int r, int g, int b, int a)
- int **qGray** (int r, int g, int b)
- int **qGray** (qRgb rgb)

Detailed Description

The QColor class provides colors based on RGB.

A color is normally specified in terms of RGB (red, green and blue) components, but it is also possible to specify HSV (hue, saturation and value) or set a color name (the names are copied from from the X11 color database).

In addition to the RGB value, a QColor also has a pixel value and a validity. The pixel value is used by the underlying window system to refer to a color. It can be thought of as an index into the display hardware's color table.

The validity (isValid()) indicates whether the color is legal at all. For example, a RGB color with RGB values out of range is illegal. For performance reasons, QColor mostly disregards illegal colors. The result of using an invalid color is unspecified and will usually be surprising.

There are 19 predefined QColor objects: `black`, `white`, `darkGray`, `gray`, `lightGray`, `red`, `green`, `blue`, `cyan`, `magenta`, `yellow`, `darkRed`, `darkGreen`, `darkBlue`, `darkCyan`, `darkMagenta`, `darkYellow`, `color0` and `color1`.

The colors `color0` (zero pixel value) and `color1` (non-zero pixel value) are special colors for drawing in bitmaps.

The QColor class has an efficient, dynamic color allocation strategy. A color is normally allocated the first time it is used (lazy allocation), that is, whenever the pixel() function is called:

1. Is the pixel value valid? If it is, just return it; otherwise, allocate a pixel value.
2. Check an internal hash table to see if we allocated an equal RGB value earlier. If we did, set the pixel value and return.
3. Try to allocate the RGB value. If we succeed, we get a pixel value that we save in the internal table with the RGB value. Return the pixel value.
4. The color could not be allocated. Find the closest matching color and save it in the internal table.

Because many people don't know the HSV color model very well, we'll cover it briefly here.

The RGB model is hardware-oriented. Its representation is close to what most monitors show. In contrast, HSV represents color in a way more suited to traditional human perception of color. For example, the relationships "stronger than", "darker than" and "the opposite of" are easily expressed in HSV but are much harder to express in RGB.

HSV, like RGB, has three components:

- H, for hue, is either 0-359 if the color is chromatic (not gray), or meaningless if it is gray. It represents degrees on the color wheel familiar to most people. Red is 0 (degrees), green is 120 and blue is 240.
- S, for saturation, is 0-255, and the bigger it is, the stronger the color is. Grayish colors have saturation near 0; very strong colors have saturation near 255.
- V, for value, is 0-255 and represents lightness or brightness of the color. 0 is black; 255 is as far from black as possible.

Here are some examples: Pure red is H=0, S=255, V=255. A dark red, moving slightly towards the magenta, could be H=350 (equivalent to -10), S=255, V=180. A grayish light red could have H about 0 (say 350-359 or 0-10), S about 50-100, and S=255.

Qt returns a hue value of -1 for achromatic colors. If you pass a too-big hue value, Qt forces it into range. Hue 360 or 720 is treated as 0; hue 540 is treated as 180.

A color can be set by passing setNamedColor() an RGB string like "#112233", or a color name, e.g. "blue". The names are taken from X11's rgb.txt database but can also be used under Windows. To get a lighter or darker color use light() and dark() respectively. Colors can also be set using setRgb() and setHsv(). The color components can be accessed in one go with rgb() and hsv(), or individually with red(), green() and blue().

Use maxColors() and numBitPlanes() to determine the maximum number of colors and the number of bit planes supported by the underlying window system,

If you need to allocate many colors temporarily, for example in an image viewer application, enterAllocContext(), leaveAllocContext() and destroyAllocContext() will prove useful.

See also QPalette [p. 209], QColorGroup [p. 84], QApplication::setColorSpec() [Additional Functionality with Qt], Color FAQ, Widget Appearance and Style, Graphics Classes and Image Processing Classes.

Member Type Documentation

QColor::Spec

The type of color specified, either RGB or HSV, e.g. in the `QColor::QColor( x, y, z, colorSpec)` constructor.

- `QColor::Rgb`
- `QColor::Hsv`

Member Function Documentation

QColor::QColor ()

Constructs an invalid color with the RGB value (0,0,0). An invalid color is a color that is not properly set up for the underlying window system.

The alpha value of an invalid color is unspecified.

See also isValid() [p. 80].

QColor::QColor (int r, int g, int b)

Constructs a color with the RGB value *r, g, b*, in the same way as setRgb().

The color is left invalid if any or the arguments are illegal.

See also setRgb() [p. 82].

QColor::QColor (int x, int y, int z, Spec colorSpec)

Constructs a color with the RGB or HSV value *x, y, z*.

The arguments are an RGB value if *colorSpec* is QColor::Rgb. *x* (red), *y* (green), and *z* (blue). All of them must be in the range 0-255.

The arguments are an HSV value if *colorSpec* is QColor::Hsv. *x* (hue) must be -1 for achromatic colors and 0-359 for chromatic colors; *y* (saturation) and *z* (value) must both be in the range 0-255.

See also setRgb() [p. 82] and setHsv() [p. 82].

QColor::QColor (QRgb rgb, uint pixel = 0xffffffff)

Constructs a color with the RGB value *rgb* and a custom pixel value *pixel*.

If *pixel* == 0xffffffff (the default), then the color uses the RGB value in a standard way. If *pixel* is something else, then the pixel value is set directly to *pixel*, skipping the normal allocation procedure.

QColor::QColor (const QString & name)

Constructs a named color in the same way as setNamedColor() using name *name*.

See also setNamedColor() [p. 82].

QColor::QColor (const char * name)

Constructs a named color in the same way as setNamedColor() using name *name*.

See also setNamedColor() [p. 82].

QColor::QColor (const QColor & c)

Constructs a color that is a copy of *c*.

uint QColor::alloc ()

Allocates the RGB color and returns the pixel value.

Allocating a color means to obtain a pixel value from the RGB specification. The pixel value is an index into the global color table, but should be considered an arbitrary platform-dependent value.

The pixel() function calls alloc() if necessary, so in general you don't need to call this function.

See also enterAllocContext() [p. 78].

int QColor::blue () const

Returns the B (blue) component of the RGB value.

void QColor::cleanup () [static]

Internal clean up required for QColor. This function is called from the QApplication destructor.

See also initialize() [p. 80].

int QColor::currentAllocContext () [static]

Returns the current color allocation context.

The default context is 0.

See also enterAllocContext() [p. 78] and leaveAllocContext() [p. 80].

QColor QColor::dark (int factor = 200) const

Returns a darker (or lighter) color, but does not change this object.

Returns a darker color if *factor* is greater than 100. Setting *factor* to 300 returns a color that has one-third the brightness.

Returns a lighter color if *factor* is less than 100. We recommend using lighter() for this purpose. If *factor* is 0 or negative, the return value is unspecified.

(This function converts the current RGB color to HSV, divides V by *factor* and converts back to RGB.)

See also light() [p. 80].

Examples: desktop/desktop.cpp and themes/wood.cpp.

void QColor::destroyAllocContext (int context) [static]

Destroys a color allocation context, *context*.

This function deallocates all colors that were allocated in the specified *context*. If *context* == -1, it frees up all colors that the application has allocated. If *context* == -2, it frees up all colors that the application has allocated, except those in the default context.

The function does nothing for true color displays.

See also enterAllocContext() [p. 78] and alloc() [p. 77].

Example: showimg/showimg.cpp.

int QColor::enterAllocContext () [static]

Enters a color allocation context and returns a nonzero unique identifier.

Color allocation contexts are useful for programs that need to allocate many colors and throw them away later, like image viewers. The allocation context functions work for true color displays as well as colormap display, except that QColor::destroyAllocContext() does nothing for true color.

Example:

```
QPixmap loadPixmap( QString fileName )
{
    static int alloc_context = 0;
    if ( alloc_context )
        QColor::destroyAllocContext( alloc_context );
    alloc_context = QColor::enterAllocContext();
    QPixmap pm( fileName );
    QColor::leaveAllocContext();
    return pm;
}
```

The example code loads a pixmap from file. It frees up all colors that were allocated the last time loadPixmap() was called.

The initial/default context is 0. Qt keeps a list of colors associated with their allocation contexts. You can call destroyAllocContext() to get rid of all colors that were allocated in a specific context.

Calling enterAllocContext() enters an allocation context. The allocation context lasts until you call leaveAllocContext(). QColor has an internal stack of allocation contexts. Each call to enterAllocContex() must have a corresponding leaveAllocContext().

```
    // context 0 active
int c1 = QColor::enterAllocContext();       // enter context c1
    // context c1 active
int c2 = QColor::enterAllocContext();       // enter context c2
    // context c2 active
QColor::leaveAllocContext();                // leave context c2
    // context c1 active
QColor::leaveAllocContext();                // leave context c1
    // context 0 active
    // Now, free all colors that were allocated in context c2
QColor::destroyAllocContext( c2 );
```

You may also want to set the application's color specification. See QApplication::setColorSpec() for more information.

See also leaveAllocContext() [p. 80], currentAllocContext() [p. 78], destroyAllocContext() [p. 78] and QApplication::setColorSpec() [Additional Functionality with Qt].

Example: showimg/showimg.cpp.

void QColor::getHsv (int & h, int & s, int & v) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

int QColor::green () const

Returns the G (green) component of the RGB value.

void QColor::hsv (int * h, int * s, int * v) const

Returns the current RGB value as HSV. The contents of the *h*, *s* and *v* pointers are set to the HSV values. If any of the three pointers are null, the function does nothing.

The hue (which *h* points to) is set to -1 if the color is achromatic.

See also setHsv() [p. 82] and rgb() [p. 81].

Example: themes/metal.cpp.

void QColor::initialize () [static]

Internal initialization required for QColor. This function is called from the QApplication constructor.

See also cleanup() [p. 78].

bool QColor::isValid () const

Returns FALSE if the color is invalid, i.e., it was constructed using the default constructor.

Use of this function is discouraged, as it is slightly slow on Truecolor displays. If you need a "null" QColor, it may be better to use q QColor* where possible.

Example: scribble/scribble.cpp.

void QColor::leaveAllocContext () [static]

Leaves a color allocation context.

See enterAllocContext() for a detailed explanation.

See also enterAllocContext() [p. 78] and currentAllocContext() [p. 78].

Example: showimg/showimg.cpp.

QColor QColor::light (int factor = 150) const

Returns a lighter (or darker) color, but does not change this object.

Returns a lighter color if *factor* is greater than 100. Setting *factor* to 150 returns a color that is 50% brighter.

Returns a darker color if *factor* is less than 100. We recommend using dark() for this purpose. If *factor* is 0 or negative, the return value is unspecified.

(This function converts the current RGB color to HSV, multiplies V by *factor*, and converts the result back to RGB.)

See also dark() [p. 78].

Examples: desktop/desktop.cpp and themes/wood.cpp.

int QColor::maxColors () [static]

Returns the maximum number of colors supported by the underlying window system.

QString QColor::name () const

Returns the name of the color in the format "#RRGGBB", i.e., a "#" character followed by three two-digit hexadecimal numbers.

See also setNamedColor() [p. 82].

int QColor::numBitPlanes () [static]

Returns the number of color bit planes for the underlying window system.

The returned values is equal to the default pixmap depth;

See also QPixmap::defaultDepth() [p. 231].

bool QColor::operator!= (const QColor & c) const

Returns TRUE if this color has a different RGB value from *c*, or FALSE if they have equal RGB values.

QColor & QColor::operator= (const QColor & c)

Assigns a copy of the color *c* and returns a reference to this color.

bool QColor::operator== (const QColor & c) const

Returns TRUE if this color has the same RGB value as *c*, or FALSE if they have different RGB values.

uint QColor::pixel () const

Returns the pixel value.

This value is used by the underlying window system to refer to a color. It can be thought of as an index into the display hardware's color table, but the value is an arbitrary 32-bit value.

See also alloc() [p. 77].

int QColor::red () const

Returns the R (red) component of the RGB value.

void QColor::rgb (int * r, int * g, int * b) const

Sets the contents pointed to by *r*, *g* and *b* to the red, green and blue components of the RGB value respectively. The value range for a component is 0..255.

See also setRgb() [p. 82] and hsv() [p. 79].

QRgb QColor::rgb () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the RGB value.

The return type *QRgb* is equivalent to `unsigned int`.

For an invalid color, the alpha value of the returned valud is unspecified.

See also setRgb() [p. 82], hsv() [p. 79], qRed() [p. 83], qBlue() [p. 83], qGreen() [p. 83] and isValid() [p. 80].

void QColor::setHsv (int h, int s, int v)

Sets a HSV color value. *h* is the hue, *s* is the saturation and *v* is the value of the HSV color.

If *s* or *v* are not in the range 0-255, or *h* is < -1, the color is not changed.

See also hsv() [p. 79] and setRgb() [p. 82].

Examples: drawdemo/drawdemo.cpp, grapher/grapher.cpp and progress/progress.cpp.

void QColor::setNamedColor (const QString & name)

Sets the RGB value to *name*, which may be in one of these formats:

- #RGB (each of R, G and B is a single hex digit)
- #RRGGBB
- #RRRGGGBBB
- #RRRRGGGGBBBB
- A name from the X color database (rgb.txt) (e.g. "steelblue" or "gainsboro"). These color names also work under Qt for Windows.

The color is left invalid if *name* cannot be parsed.

void QColor::setRgb (int r, int g, int b)

Sets the RGB value to *r*, *g*, *b*. The arguments, *r*, *g* and *b* must all be in the range 0..255. If any of them are outside the legal range, the color is not changed.

See also rgb() [p. 81] and setHsv() [p. 82].

void QColor::setRgb (QRgb rgb)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the RGB value to *rgb*.

The type *QRgb* is equivalent to `unsigned int`.

See also rgb() [p. 81] and setHsv() [p. 82].

Related Functions

QDataStream & operator<< (QDataStream & s, const QColor & c)

Writes a color object, *c* to the stream, *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QColor & c)

Reads a color object, *c*, from the stream, *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

int qAlpha (QRgb rgba)

Returns the alpha component of the RGBA quadruplet *rgba*.

int qBlue (QRgb rgb)

Returns the blue component of the RGB triplet *rgb*.
See also qRgb() [p. 83] and QColor::blue() [p. 78].

int qGray (int r, int g, int b)

Returns a gray value 0..255 from the (*r*, *g*, *b*) triplet.
The gray value is calculated using the formula (r*11 + g*16 + b*5)/32.

int qGray (qRgb rgb)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns a gray value 0..255 from the given *rgb* colour.

int qGreen (QRgb rgb)

Returns the green component of the RGB triplet *rgb*.
See also qRgb() [p. 83] and QColor::green() [p. 79].

int qRed (QRgb rgb)

Returns the red component of the RGB triplet *rgb*.
See also qRgb() [p. 83] and QColor::red() [p. 81].

QRgb qRgb (int r, int g, int b)

Returns the RGB triplet *(r,g,b)*.
The return type QRgb is equivalent to `unsigned int`.
See also qRgba() [p. 83], qRed() [p. 83], qGreen() [p. 83] and qBlue() [p. 83].

QRgb qRgba (int r, int g, int b, int a)

Returns the RGBA quadruplet *(r,g,b,a)*.
The return type QRgba is equivalent to `unsigned int`.
See also qRgb() [p. 83], qRed() [p. 83], qGreen() [p. 83] and qBlue() [p. 83].

QColorGroup Class Reference

The QColorGroup class contains a group of widget colors.

```
#include <qpalette.h>
```

Public Members

- **QColorGroup** ()
- QColorGroup (const QColor & foreground, const QColor & background, const QColor & light, const QColor & dark, const QColor & mid, const QColor & text, const QColor & base) *(obsolete)*
- **QColorGroup** (const QBrush & foreground, const QBrush & button, const QBrush & light, const QBrush & dark, const QBrush & mid, const QBrush & text, const QBrush & bright_text, const QBrush & base, const QBrush & background)
- **QColorGroup** (const QColorGroup & other)
- **~QColorGroup** ()
- QColorGroup & **operator=** (const QColorGroup & other)
- enum **ColorRole** { Foreground, Button, Light, Midlight, Dark, Mid, Text, BrightText, ButtonText, Base, Background, Shadow, Highlight, HighlightedText, Link, LinkVisited, NColorRoles }
- const QColor & **color** (ColorRole r) const
- const QBrush & **brush** (ColorRole r) const
- void **setColor** (ColorRole r, const QColor & c)
- void **setBrush** (ColorRole r, const QBrush & b)
- const QColor & **foreground** () const
- const QColor & **button** () const
- const QColor & **light** () const
- const QColor & **dark** () const
- const QColor & **mid** () const
- const QColor & **text** () const
- const QColor & **base** () const
- const QColor & **background** () const
- const QColor & **midlight** () const
- const QColor & **brightText** () const
- const QColor & **buttonText** () const
- const QColor & **shadow** () const
- const QColor & **highlight** () const
- const QColor & **highlightedText** () const
- const QColor & **link** () const
- const QColor & **linkVisited** () const
- bool **operator==** (const QColorGroup & g) const
- bool **operator!=** (const QColorGroup & g) const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QColorGroup & g)

Detailed Description

The QColorGroup class contains a group of widget colors.

A color group contains a group of colors used by widgets for drawing themselves. We recommend that widgets use color group roles such as "foreground" and "base" rather than literal colors like "red" or "turqoise". The color roles are enumerated and defined in the ColorRole documentation.

The most common use of QColorGroup is like this:

```
QPainter p;
...
p.setPen( colorGroup().foreground() );
p.drawLine( ... )
```

See the ColorRole [p. 85] documentation below for more details on roles.

It is also possible to modify color groups or create new color groups from scratch.

The color group class can be created using three different constructors or by modifying one supplied by the Qt. The default constructor creates an all-black color group, which can then be modified using set functions. There are two functions that take long lists of arguments (slightly different lists - beware!). And there is the copy constructor.

We strongly recommend using a system-supplied color group and modifying that as necessary.

You modify a color group by calling the access functions setColor() and setBrush(), depending on whether you want a pure color or a pixmap pattern.

There are also corresponding color() and brush() getters, and a commonly used convenience function to get each ColorRole: background(), foreground(), base(), etc.

See also QColor [p. 74], QPalette [p. 209], QWidget::colorGroup [Widgets with Qt], Widget Appearance and Style, Graphics Classes and Image Processing Classes.

Member Type Documentation

QColorGroup::ColorRole

The ColorRole enum defines the different symbolic color roles used in current GUIs. The central roles are as follow:

- `QColorGroup::Background` - general background color.
- `QColorGroup::Foreground` - general foreground color.
- `QColorGroup::Base` - used as background color for text entry widgets, for example; usually white or another light color.
- `QColorGroup::Text` - the foreground color used with Base. Usually this is the same as the Foreground, in which case it must provide good contrast with Background and Base.
- `QColorGroup::Button` - general button background color in which buttons need a background different from Background, as in the Macintosh style.
- `QColorGroup::ButtonText` - a foreground color used with the Button color.

There are some color roles used mostly for 3D bevel and shadow effects:

- QColorGroup::Light - lighter than Button color.
- QColorGroup::Midlight - between Button and Light.
- QColorGroup::Dark - darker than Button.
- QColorGroup::Mid - between Button and Dark.
- QColorGroup::Shadow - a very dark color.

All of these are normally derived from Background and used in ways that depend on that relationship. For example, buttons depend on it to make the bevels look attractive, and Motif scroll bars depend on Mid to be slightly different from Background.

Selected (marked) items have two roles:

- QColorGroup::Highlight - a color to indicate a selected or highlighted item.
- QColorGroup::HighlightedText - a text color that contrasts with Highlight.

Finally, there is a special role for text that needs to be drawn where Text or Foreground would give poor contrast, such as on pressed push buttons:

- QColorGroup::BrightText - a text color that is very different from Foreground and contrasts well with e.g. Dark.
- QColorGroup::Link - a text color used for unvisited hyperlinks.
- QColorGroup::LinkVisited - a text color used for already visited hyperlinks.
- QColorGroup::NColorRoles - Internal.

Note that text colors can be used for things other than just words; text colors are *usually* used for text, but it's quite common to use the text color roles for lines, icons, etc.
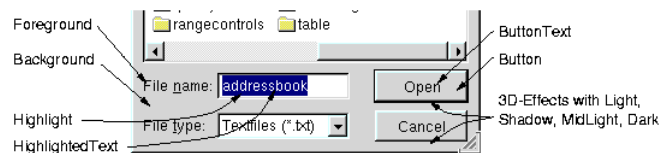
This image shows most of the color roles in use:

Member Function Documentation

QColorGroup::QColorGroup ()

Constructs a color group with all colors set to black.

QColorGroup::QColorGroup (const QColor & foreground, const QColor & background, const QColor & light, const QColor & dark, const QColor & mid, const QColor & text, const QColor & base)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Constructs a color group with the specified colors. The button color will be set to the background color.

QColorGroup::QColorGroup (const QBrush & foreground, const QBrush & button, const QBrush & light, const QBrush & dark, const QBrush & mid, const QBrush & text, const QBrush & bright_text, const QBrush & base, const QBrush & background)

Constructs a color group. You can pass either brushes, pixmaps or plain colors for *foreground, button, light, dark, mid, text, bright_text, base* and *background.*

This constructor can be very handy sometimes, but don't overuse it: such long lists of arguments are rather error-prone.

See also QBrush [p. 17].

QColorGroup::QColorGroup (const QColorGroup & other)

Constructs a color group that is an independent copy of *other.*

QColorGroup::~QColorGroup ()

Destroys the color group.

const QColor & QColorGroup::background () const

Returns the background color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::base () const

Returns the base color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::brightText () const

Returns the bright text foreground color of the color group.

See also ColorRole [p. 85].

Examples: themes/metal.cpp and themes/wood.cpp.

const QBrush & QColorGroup::brush (ColorRole r) const

Returns the brush that has been set for color role *r.*

See also color() [p. 88], setBrush() [p. 89] and ColorRole [p. 85].

Examples: themes/metal.cpp and themes/wood.cpp.

const QColor & QColorGroup::button () const

Returns the button color of the color group.

See also ColorRole [p. 85].

Examples: themes/metal.cpp and themes/wood.cpp.

const QColor & QColorGroup::buttonText () const

Returns the button text foreground color of the color group.

See also ColorRole [p. 85].

Examples: themes/metal.cpp and themes/wood.cpp.

const QColor & QColorGroup::color (ColorRole r) const

Returns the color that has been set for color role *r*.

See also brush() [p. 87] and ColorRole [p. 85].

const QColor & QColorGroup::dark () const

Returns the dark color of the color group.

See also ColorRole [p. 85].

Example: themes/wood.cpp.

const QColor & QColorGroup::foreground () const

Returns the foreground color of the color group.

See also ColorRole [p. 85].

Examples: themes/metal.cpp and themes/wood.cpp.

const QColor & QColorGroup::highlight () const

Returns the highlight color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::highlightedText () const

Returns the highlighted text color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::light () const

Returns the light color of the color group.

See also ColorRole [p. 85].

Example: themes/wood.cpp.

const QColor & QColorGroup::link () const

Returns the unvisited link text color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::linkVisited () const

Returns the visited link text color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::mid () const

Returns the mid color of the color group.

See also ColorRole [p. 85].

Examples: themes/metal.cpp and themes/wood.cpp.

const QColor & QColorGroup::midlight () const

Returns the midlight color of the color group.

See also ColorRole [p. 85].

bool QColorGroup::operator!= (const QColorGroup & g) const

Returns TRUE if this color group is different from *g*; otherwise returns FALSE.

See also

QColorGroup & QColorGroup::operator= (const QColorGroup & other)

Copies the colors of *other* to this color group.

bool QColorGroup::operator== (const QColorGroup & g) const

Returns TRUE if this color group is equal to *g*; otherwise returns FALSE.

See also

void QColorGroup::setBrush (ColorRole r, const QBrush & b)

Sets the brush used for color role *r* to *b*.

See also brush() [p. 87], setColor() [p. 90] and ColorRole [p. 85].

Example: themes/wood.cpp.

void QColorGroup::setColor (ColorRole r, const QColor & c)

Sets the brush used for color role *r* to a solid color *c*.

See also brush() [p. 87] and ColorRole [p. 85].

Examples: listviews/listviews.cpp and themes/metal.cpp.

const QColor & QColorGroup::shadow () const

Returns the shadow color of the color group.

See also ColorRole [p. 85].

const QColor & QColorGroup::text () const

Returns the text foreground color of the color group.

See also ColorRole [p. 85].

Example: listviews/listviews.cpp.

Related Functions

QDataStream & operator<< (QDataStream & s, const QColorGroup & g)

Writes color group, *g* to the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QCursor Class Reference

The QCursor class provides a mouse cursor with an arbitrary shape.

`#include <qcursor.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QCursor** ()
- **QCursor** (int shape)
- **QCursor** (const QBitmap & bitmap, const QBitmap & mask, int hotX = -1, int hotY = -1)
- **QCursor** (const QPixmap & pixmap, int hotX = -1, int hotY = -1)
- **QCursor** (const QCursor & c)
- **~QCursor** ()
- QCursor & **operator=** (const QCursor & c)
- int **shape** () const
- void **setShape** (int shape)
- const QBitmap * **bitmap** () const
- const QBitmap * **mask** () const
- QPoint **hotSpot** () const
- HCURSOR **handle** () const
- **QCursor** (HCURSOR handle)

Static Public Members

- QPoint **pos** ()
- void **setPos** (int x, int y)
- void **setPos** (const QPoint &)
- void **initialize** ()
- void **cleanup** ()

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QCursor & c)
- QDataStream & **operator>>** (QDataStream & s, QCursor & c)

Detailed Description

The QCursor class provides a mouse cursor with an arbitrary shape.

This class is mainly used to create mouse cursors that are associated with particular widgets and to get and set the position of the mouse cursor.

Qt has a number of standard cursor shapes, but you can also make custom cursor shapes based on a QBitmap, a mask and a hotspot.

To associate a cursor with a widget, use QWidget::setCursor(). To associate a cursor with all widgets (normally for a short period of time), use QApplication::setOverrideCursor().

To set a cursor shape use QCursor::setShape() or use the QCursor constructor which takes the shape as argument, or you can use one of the predefined cursors defined in the CursorShape enum.

If you want to create a cursor with your own bitmap, either use the QCursor constructor which takes a bitmap and a mask or the constructor which takes a pixmap as arguments.

To set or get the position of the mouse cursor use the static methods QCursor::pos() and QCursor::setPos().

See also QWidget [Widgets with Qt], GUI Design Handbook: Cursors, Widget Appearance and Style and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QCursor::QCursor ( )

Constructs a cursor with the default arrow shape.

QCursor::QCursor (int shape)

Constructs a cursor with the specified *shape*.

See CursorShape for a list of shapes.

See also setShape() [p. 94].

QCursor::QCursor (const QBitmap & bitmap, const QBitmap & mask, int hotX = -1, int hotY = -1)

Constructs a custom bitmap cursor.

bitmap and *mask* make up the bitmap. *hotX* and *hotY* define the hot spot of this cursor.

If *hotX* is negative, it is set to the `bitmap().width()/2`. If *hotY* is negative, it is set to the `bitmap().height()/2`.

The cursor *bitmap* (B) and *mask* (M) bits are combined this way:

1. B=1 and M=1 gives black.
2. B=0 and M=1 gives white.
3. B=0 and M=0 gives transparency.
4. B=1 and M=0 gives an undefined result.

Use the global color `color0` to draw 0-pixels and `color1` to draw 1-pixels in the bitmaps.

Valid cursor sizes depend on the display hardware (or the underlying window system). We recommend using 32x32 cursors, because this size is supported on all platforms. Some platforms also support 16x16, 48x48 and 64x64 cursors.

See also QBitmap::QBitmap() [p. 15] and QBitmap::setMask() [p. 236].

QCursor::QCursor (const QPixmap & pixmap, int hotX = -1, int hotY = -1)

Constructs a custom pixmap cursor.

pixmap is the image (usually it should have a mask (set using QPixmap::setMask()) *hotX* and *hotY* define the hot spot of this cursor.

If *hotX* is negative, it is set to the `pixmap().width()/2`. If *hotY* is negative, it is set to the `pixmap().height()/2`.

Valid cursor sizes depend on the display hardware (or the underlying window system). We recommend using 32x32 cursors, because this size is supported on all platforms. Some platforms also support 16x16, 48x48 and 64x64 cursors.

Currently, only black-and-white pixmaps can be used.

See also QPixmap::QPixmap() [p. 228] and QPixmap::setMask() [p. 236].

QCursor::QCursor (const QCursor & c)

Constructs a copy of the cursor *c*.

QCursor::QCursor (HCURSOR handle)

Creates a cursor with the specified window system handle *handle*.

Warning: Portable in principle, but if you use it you are probably about to do something non-portable. Be careful.

QCursor::~QCursor ()

Destroys the cursor.

const QBitmap * QCursor::bitmap () const

Returns the cursor bitmap, or 0 if it is one of the standard cursors.

void QCursor::cleanup () [static]

Internal function that deinitializes the predefined cursors. This function is called from the QApplication destructor.

See also initialize() [p. 94].

HCURSOR QCursor::handle () const

Returns the window system cursor handle.

Warning: Portable in principle, but if you use it you are probably about to do something non-portable. Be careful.

QPoint QCursor::hotSpot () const

Returns the cursor hot spot, or (0,0) if it is one of the standard cursors.

void QCursor::initialize () [static]

Internal function that initializes the predefined cursors. This function is called from the QApplication constructor.

See also cleanup() [p. 93].

const QBitmap * QCursor::mask () const

Returns the cursor bitmap mask, or 0 if it is one of the standard cursors.

QCursor & QCursor::operator= (const QCursor & c)

Assigns *c* to this cursor and returns a reference to this cursor.

QPoint QCursor::pos () [static]

Returns the position of the cursor (hot spot) in global screen coordinates.

You can call QWidget::mapFromGlobal() to translate it to widget coordinates.

See also setPos() [p. 94], QWidget::mapFromGlobal() [Widgets with Qt] and QWidget::mapToGlobal() [Widgets with Qt].

Example: fileiconview/qfileiconview.cpp.

void QCursor::setPos (int x, int y) [static]

Moves the cursor (hot spot) to the global screen position *x* and *y*.

You can call QWidget::mapToGlobal() to translate widget coordinates to global screen coordinates.

See also pos() [p. 94], QWidget::mapFromGlobal() [Widgets with Qt] and QWidget::mapToGlobal() [Widgets with Qt].

void QCursor::setPos (const QPoint &) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

void QCursor::setShape (int shape)

Sets the cursor to the shape identified by *shape*.

See CursorShape for a list of shapes.

See also shape() [p. 95].

int QCursor::shape () const

Returns the cursor shape identifier. The return value is one of following values (casted to an int).

See CursorShape for a list of shapes.

See also setShape() [p. 94].

Related Functions

QDataStream & operator<< (QDataStream & s, const QCursor & c)

Writes the cursor *c* to the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QCursor & c)

Reads a cursor from the stream *s* and sets *c* to the read data.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QGL Class Reference

The QGL class is a namespace for miscellaneous identifiers in the Qt OpenGL module.

This class is part of the **OpenGL** module.

`#include <qgl.h>`

Inherited by QGLFormat [p. 106], QGLContext [p. 101] and QGLWidget [p. 114].

Public Members

- enum **FormatOption** { DoubleBuffer = 0x0001, DepthBuffer = 0x0002, Rgba = 0x0004, AlphaChannel = 0x0008, AccumBuffer = 0x0010, StencilBuffer = 0x0020, StereoBuffers = 0x0040, DirectRendering = 0x0080, HasOverlay = 0x0100, SingleBuffer = DoubleBuffer<<16, NoDepthBuffer = DepthBuffer<<16, ColorIndex = Rgba<<16, NoAlphaChannel = AlphaChannel<<16, NoAccumBuffer = AccumBuffer<<16, NoStencilBuffer = StencilBuffer<<16, NoStereoBuffers = StereoBuffers<<16, IndirectRendering = DirectRendering<<16, NoOverlay = HasOverlay<<16 }

Detailed Description

The QGL class is a namespace for miscellaneous identifiers in the Qt OpenGL module.

Normally you can ignore this class. QGLWidget and the other OpenGL* module classes inherit it, so when you make your own QGLWidget subclass you can use the identifiers in the QGL namespace without qualification.

However, you may occasionally find yourself in situations where you need to refer to these identifiers from outside the QGL namespace's scope, e.g. in static functions. In such cases, simply write e.g. QGL::DoubleBuffer instead of just DoubleBuffer.

* OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

See also Graphics Classes and Image Processing Classes.

Member Type Documentation

QGL::FormatOption

This enum specifies the format options.

- `QGL::DoubleBuffer`
- `QGL::DepthBuffer`
- `QGL::Rgba`
- `QGL::AlphaChannel`

- QGL::AccumBuffer
- QGL::StencilBuffer
- QGL::StereoBuffers
- QGL::DirectRendering
- QGL::HasOverlay
- QGL::SingleBuffer
- QGL::NoDepthBuffer
- QGL::ColorIndex
- QGL::NoAlphaChannel
- QGL::NoAccumBuffer
- QGL::NoStencilBuffer
- QGL::NoStereoBuffers
- QGL::IndirectRendering
- QGL::NoOverlay

QGLColormap Class Reference

The QGLColormap class is used for installing custom colormaps into QGLWidgets.

```
#include <qglcolormap.h>
```

Public Members

- **QGLColormap** ()
- **QGLColormap** (const QGLColormap & map)
- **~QGLColormap** ()
- QGLColormap & **operator=** (const QGLColormap & map)
- bool **isEmpty** () const
- int **size** () const
- void **detach** ()
- void **setEntries** (int count, const QRgb * colors, int base = 0)
- void **setEntry** (int idx, QRgb color)
- void **setEntry** (int idx, const QColor & color)
- QRgb **entryRgb** (int idx) const
- QColor **entryColor** (int idx) const
- int **find** (QRgb color) const
- int **findNearest** (QRgb color) const

Detailed Description

The QGLColormap class is used for installing custom colormaps into QGLWidgets.

QGLColormap provides a platform independent way of specifying and installing indexed colormaps into QGLWidgets. QGLColormap is especially useful when using the OpenGL color-index mode.

Under X11 you will have to use an X server that supports either a PseudoColor or DirectColor visual class. If your X server currently only provides a GrayScale, TrueColor, StaticColor or StaticGray visual, you will not be able to allocate colorcells for writing. If this is the case, try setting your X server in 8 bit mode. It should then provide you with at least a PseudoColor visual. Note that you may experience colormap flashing if your X server is running in 8 bit mode.

Under Windows the size of the colormap is always set to 256 colors. Note that under Windows you are allowed to install colormaps into child widgets.

This class uses explicit sharing (see Shared Classes).

Example of use:

```
#include <qapplication.h>
```

```
#include <qglcolormap.h>

int main()
{
    QApplication a( argc, argv );

    MySuperGLWidget widget( 0 ); // A QGLWidget in color-index mode
    QGLColormap colormap;

    // This will fill the colormap with colors ranging from
    // black to white.
    for ( int i = 0; i size(); i++ )
        colormap->setEntry( i, qRgb( i, i, i ) );

    widget.setColormap( colormap );
    widget.show();
    return a.exec();
}
```

See also QGLWidget::setColormap() [p. 121], QGLWidget::colormap() [p. 118], Graphics Classes and Image Processing Classes.

Member Function Documentation

QGLColormap::QGLColormap ()

Construct a QGLColormap.

QGLColormap::QGLColormap (const QGLColormap & map)

Construct a shallow copy of *map*.

QGLColormap::~QGLColormap ()

Dereferences the QGLColormap and deletes it if this was the last reference to it.

void QGLColormap::detach ()

Detaches this QGLColormap from the shared block.

QColor QGLColormap::entryColor (int idx) const

Returns the QRgb value in the colorcell with index *idx*.

QRgb QGLColormap::entryRgb (int idx) const

Returns the QRgb value in the colorcell with index *idx*.

int QGLColormap::find (QRgb color) const

Returns the index of the color *color*. If *color* is not in the map, -1 is returned.

int QGLColormap::findNearest (QRgb color) const

Returns the index of the color that is the closest match to color *color*.

bool QGLColormap::isEmpty () const

Returns TRUE if the colormap is empty; otherwise returns FALSE. A colormap with no color values set is considered to be empty.

QGLColormap & QGLColormap::operator= (const QGLColormap & map)

Assign a shallow copy of *map* to this QGLColormap.

void QGLColormap::setEntries (int count, const QRgb * colors, int base = 0)

Set an array of cells in this colormap. *count* is the number of colors that should be set, *colors* is the array of colors, and *base* is the starting index.

void QGLColormap::setEntry (int idx, QRgb color)

Set cell *idx* in the colormap to color *color*.

void QGLColormap::setEntry (int idx, const QColor & color)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Set cell with index *idx* in the colormap to color *color*.

int QGLColormap::size () const

Returns the number of colorcells in the colormap.

QGLContext Class Reference

The QGLContext class encapsulates an OpenGL rendering context.

This class is part of the **OpenGL** module.

```
#include <qgl.h>
```

Inherits QGL [p. 96].

Public Members

- **QGLContext** (const QGLFormat & format, QPaintDevice * device)
- virtual **~QGLContext** ()
- virtual bool **create** (const QGLContext * shareContext = 0)
- bool **isValid** () const
- bool **isSharing** () const
- virtual void **reset** ()
- QGLFormat **format** () const
- QGLFormat **requestedFormat** () const
- virtual void **setFormat** (const QGLFormat & format)
- virtual void **makeCurrent** ()
- virtual void **swapBuffers** () const
- QPaintDevice * **device** () const
- QColor **overlayTransparentColor** () const

Static Public Members

- const QGLContext * **currentContext** ()

Protected Members

- virtual bool **chooseContext** (const QGLContext * shareContext = 0)
- virtual void **doneCurrent** ()
- virtual int **choosePixelFormat** (void * dummyPfd, HDC pdc)
- bool **deviceIsPixmap** () const
- bool **windowCreated** () const
- void **setWindowCreated** (bool on)
- bool **initialized** () const
- void **setInitialized** (bool on)

Detailed Description

The QGLContext class encapsulates an OpenGL rendering context.

An OpenGL* rendering context is a complete set of OpenGL state variables.

The context's format is set in the constructor or later with setFormat(). The format options that are actually set are returned by format(); the options you asked for are returned by requestedFormat(). The context is created by the create() function which is called from the constructors. The makeCurrent() function makes this context the current rendering context. You can make *no* context current using doneCurrent(). The reset() function will reset the context and make it invalid.

You can examine properties of the context with, e.g. isValid(), isSharing(), initialized(), windowCreated() and overlayTransparentColor().

If you're using double buffering you can swap the screen contents with the off-screen buffer using swapBuffers().

* OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QGLContext::QGLContext (const QGLFormat & format, QPaintDevice * device)

Constructs an OpenGL context for the paint device *device*, which can be a widget or a pixmap. The *format* specifies several display options for the context.

If the underlying OpenGL/Window system cannot satisfy all the features requested in *format*, the nearest subset of features will be used. After creation, the format() method will return the actual format obtained.

The context will be invalid if it was not possible to obtain a GL context at all.

See also format() [p. 103] and isValid() [p. 104].

QGLContext::~QGLContext ( ) [virtual]

Destroys the OpenGL context and frees its resources.

bool QGLContext::chooseContext (const QGLContext * shareContext = 0) [virtual protected]

This semi-internal function is called by create(). It creates a system-dependent OpenGL handle that matches the format() of *shareContext* as closely as possible.

On Windows, it calls the virtual function choosePixelFormat(), which finds a matching pixel format identifier. On X11, it calls the virtual function chooseVisual() which finds an appropriate X visual. On other platforms it may work differently.

int QGLContext::choosePixelFormat (void * dummyPfd, HDC pdc) [virtual protected]

Win32 only This virtual function chooses a pixel format that matches the OpenGL format. Reimplement this function in a subclass if you need a custom context.

Warning: The *dummyPfd* pointer and *pdc* are used as a `PIXELFORMATDESCRIPTOR*`. We use `void` to avoid using Windows-specific types in our header files.

See also chooseContext() [p. 102].

bool QGLContext::create (const QGLContext * shareContext = 0) [virtual]

Creates the GL context. Returns TRUE if it was successful in creating a GL rendering context on the paint device specified in the constructor; otherwise returns FALSE (i.e. the context is invalid).

After successful creation, format() returns the set of features of the created GL rendering context.

If *shareContext* points to a valid QGLContext, this method will try to establish OpenGL display list sharing between this context and the *shareContext*. Note that this may fail if the two contexts have different formats. Use isSharing() to see if sharing succeeded.

Implementation note: initialization of C++ class members usually takes place in the class constructor. QGLContext is an exception because it must be simple to customize. The virtual functions chooseContext() (and chooseVisual() for X11) can be reimplemented in a subclass to select a particular context. The problem is that virtual functions are not properly called during construction (even though this is correct C++) because C++ constructs class hierarchies from the bottom up. For this reason we need a create() function.

See also chooseContext() [p. 102], format() [p. 103] and isValid() [p. 104].

const QGLContext * QGLContext::currentContext () [static]

Returns the current context, i.e. the context to which any OpenGL commands will currently be directed. Returns 0 if no context is current.

See also makeCurrent() [p. 104].

QPaintDevice * QGLContext::device () const

Returns the paint device set for this context.

See also QGLContext::QGLContext() [p. 102].

bool QGLContext::deviceIsPixmap () const [protected]

Returns TRUE if the paint device of this context is a pixmap; otherwise returns FALSE.

void QGLContext::doneCurrent () [virtual protected]

Makes no GL context the current context. Normally, you do not need to call this function; QGLContext calls it as necessary.

QGLFormat QGLContext::format () const

Returns the frame buffer format that was obtained (this may be a subset of what was requested).

See also requestedFormat() [p. 104].

bool QGLContext::initialized () const [protected]

Returns TRUE if this context has been initialized, i.e. if QGLWidget::initializeGL() has been performed on it; otherwise returns FALSE.

See also setInitialized() [p. 105].

bool QGLContext::isSharing () const

Returns TRUE if display list sharing with another context was requested in the create() call and the GL system was able to fulfill this request; otherwise returns FALSE. Note that display list sharing might not be supported between contexts with different formats.

bool QGLContext::isValid () const

Returns TRUE if a GL rendering context has been successfully created; otherwise returns FALSE.

void QGLContext::makeCurrent () [virtual]

Makes this context the current OpenGL rendering context. All GL functions you call operate on this context until another context is made current.

QColor QGLContext::overlayTransparentColor () const

If this context is a valid context in an overlay plane, returns the plane's transparent color. Otherwise returns an invalid color.

The returned color's pixel value is the index of the transparent color in the colormap of the overlay plane. (Naturally, the color's RGB values are meaningless.)

The returned QColor object will generally work as expected only when passed as the argument to QGLWidget::qglColor() or QGLWidget::qglClearColor(). Under certain circumstances it can also be used to draw transparent graphics with a QPainter. See the examples/opengl/overlay_x11 example for details.

QGLFormat QGLContext::requestedFormat () const

Returns the frame buffer format that was originally requested in the constructor or setFormat().

See also format() [p. 103].

void QGLContext::reset () [virtual]

Resets the context and makes it invalid.

See also create() [p. 103] and isValid() [p. 104].

void QGLContext::setFormat (const QGLFormat & format) [virtual]

Sets a *format* for this context. The context is reset.

Call create() to create a new GL context that tries to match the new format.

```
QGLContext *cx;
//  ...
QGLFormat f;
f.setStereo( TRUE );
```

```
    cx->setFormat( f );
    if ( !cx->create() )
        exit(); // no OpenGL support, or cannot render on the specified paintdevice
    if ( !cx->format().stereo() )
        exit(); // could not create stereo context
```

See also format() [p. 103], reset() [p. 104] and create() [p. 103].

void QGLContext::setInitialized (bool on) [protected]

If *on* is TRUE the context has been initialized, i.e. QGLContext::setInitialized() has been called on it. If *on* is FALSE the context has not been initialized.

See also initialized() [p. 103].

void QGLContext::setWindowCreated (bool on) [protected]

If *on* is TRUE the context has had a window created for it. If *on* is FALSE no window has been created for the context.

See also windowCreated() [p. 105].

void QGLContext::swapBuffers () const [virtual]

Swaps the screen contents with an off-screen buffer. Works only if the context is in double buffer mode.

See also QGLFormat::setDoubleBuffer() [p. 112].

bool QGLContext::windowCreated () const [protected]

Returns TRUE if a window has been created for this context; otherwise returns FALSE.

See also setWindowCreated() [p. 105].

QGLFormat Class Reference

The QGLFormat class specifies the display format of an OpenGL rendering context.

This class is part of the **OpenGL** module.

`#include <qgl.h>`

Inherits QGL [p. 96].

Public Members

- **QGLFormat** ()
- **QGLFormat** (int options, int plane = 0)
- bool **doubleBuffer** () const
- void **setDoubleBuffer** (bool enable)
- bool **depth** () const
- void **setDepth** (bool enable)
- bool **rgba** () const
- void **setRgba** (bool enable)
- bool **alpha** () const
- void **setAlpha** (bool enable)
- bool **accum** () const
- void **setAccum** (bool enable)
- bool **stencil** () const
- void **setStencil** (bool enable)
- bool **stereo** () const
- void **setStereo** (bool enable)
- bool **directRendering** () const
- void **setDirectRendering** (bool enable)
- bool **hasOverlay** () const
- void **setOverlay** (bool enable)
- int **plane** () const
- void **setPlane** (int plane)
- void **setOption** (FormatOption opt)
- bool **testOption** (FormatOption opt) const

Static Public Members

- QGLFormat **defaultFormat** ()
- void **setDefaultFormat** (const QGLFormat & f)
- QGLFormat **defaultOverlayFormat** ()

- void **setDefaultOverlayFormat** (const QGLFormat & f)
- bool **hasOpenGL** ()
- bool **hasOpenGLOverlays** ()

Detailed Description

The QGLFormat class specifies the display format of an OpenGL rendering context.

A display format has several characteristics:

- Double or single buffering.
- Depth buffer.
- RGBA or color index mode.
- Alpha channel.
- Accumulation buffer.
- Stencil buffer.
- Stereo buffers.
- Direct rendering.
- Presence of an overlay.
- The plane of an overlay format.

You create and tell a QGLFormat object what rendering options you want from an OpenGL* rendering context.

OpenGL drivers or accelerated hardware may or may not support advanced features such as alpha channel or stereographic viewing. If you request some features that the driver/hardware does not provide when you create a QGLWidget, you will get a rendering context with the nearest subset of features.

There are different ways to define the display characteristics of a rendering context. One is to create a QGLFormat and make it default for the entire application:

```
QGLFormat f;
f.setAlpha( TRUE );
f.setStereo( TRUE );
QGLFormat::setDefaultFormat( f );
```

Or you can specify the desired format when creating an object of your QGLWidget subclass:

```
QGLFormat f;
f.setDoubleBuffer( FALSE );              // single buffer
f.setDirectRendering( FALSE );           // software rendering
MyGLWidget* myWidget = new MyGLWidget( f, ... );
```

After the widget has been created, you can find out which of the requested features the system was able to provide:

```
QGLFormat f;
f.setOverlay( TRUE );
f.setStereo( TRUE );
MyGLWidget* myWidget = new MyGLWidget( f, ... );
if ( !w->format().stereo() ) {
    // ok, goggles off
    if ( !w->format().hasOverlay() ) {
        qFatal( "Cool hardware required" );
    }
}
```

* OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

See also QGLContext [p. 101], QGLWidget [p. 114], Graphics Classes and Image Processing Classes.

Member Function Documentation

QGLFormat::QGLFormat ()

Constructs a QGLFormat object with the factory default settings:

- Double buffer: Enabled.
- Depth buffer: Enabled.
- RGBA: Enabled (i.e., color index disabled).
- Alpha channel: Disabled.
- Accumulator buffer: Disabled.
- Stencil buffer: Disabled.
- Stereo: Disabled.
- Direct rendering: Enabled.
- Overlay: Disabled.
- Plane: 0 (i.e., normal plane).

QGLFormat::QGLFormat (int options, int plane = 0)

Creates a QGLFormat object that is a copy of the current application default format.

If *options* is not 0, this copy is modified by these format options. The *options* parameter should be FormatOption values OR'ed together.

This constructor makes it easy to specify a certain desired format in classes derived from QGLWidget, for example:

```
// The rendering in MyGLWidget depends on using
// stencil buffer and alpha channel
MyGLWidget::MyGLWidget( QWidget* parent, const char* name )
    : QGLWidget( QGLFormat( StencilBuffer | AlphaChannel ), parent, name )
{
  if ( !format().stencil() )
    qWarning( "Could not get stencil buffer; results will be suboptimal" );
  if ( !format().alphaChannel() )
    qWarning( "Could not get alpha channel; results will be suboptimal" );
  ...
}
```

Note that there are FormatOption values to turn format settings both on and off, e.g. DepthBuffer and NoDepthBuffer, DirectRendering and IndirectRendering, etc.

The *plane* parameter defaults to 0 and is the plane which this format should be associated with. Not all OpenGL implmentations supports overlay/underlay rendering planes.

See also defaultFormat() [p. 109] and setOption() [p. 112].

bool QGLFormat::accum () const

Returns TRUE if the accumulation buffer is enabled; otherwise returns FALSE. The accumulation buffer is disabled by default.

See also setAccum() [p. 110].

bool QGLFormat::alpha () const

Returns TRUE if the alpha channel of the framebuffer is enabled; otherwise returns FALSE. The alpha channel is disabled by default.

See also setAlpha() [p. 111].

QGLFormat QGLFormat::defaultFormat () [static]

Returns the default QGLFormat for the application. All QGLWidgets that are created use this format unless another format is specified, e.g. when they are constructed.

If no special default format has been set using setDefaultFormat(), the default format is the same as that created with QGLFormat().

See also setDefaultFormat() [p. 111].

QGLFormat QGLFormat::defaultOverlayFormat () [static]

Returns the default QGLFormat for overlay contexts.

The factory default overlay format is:

- Double buffer: Disabled.
- Depth buffer: Disabled.
- RGBA: Disabled (i.e., color index enabled).
- Alpha channel: Disabled.
- Accumulator buffer: Disabled.
- Stencil buffer: Disabled.
- Stereo: Disabled.
- Direct rendering: Enabled.
- Overlay: Disabled.
- Plane: 1 (i.e., first overlay plane).

See also setDefaultFormat() [p. 111].

bool QGLFormat::depth () const

Returns TRUE if the depth buffer is enabled; otherwise returns FALSE. The depth buffer is enabled by default.

See also setDepth() [p. 112].

bool QGLFormat::directRendering () const

Returns TRUE if direct rendering is enabled; otherwise returns FALSE.

Direct rendering is enabled by default.

See also setDirectRendering() [p. 112].

bool QGLFormat::doubleBuffer () const

Returns TRUE if double buffering is enabled; otherwise returns FALSE. Double buffering is enabled by default.

See also setDoubleBuffer() [p. 112].

bool QGLFormat::hasOpenGL () [static]

Returns TRUE if the window system has any OpenGL support; otherwise returns FALSE.

Note: this function must not be called until the QApplication object has been created.

bool QGLFormat::hasOpenGLOverlays () [static]

Returns TRUE if the window system supports OpenGL overlays; otherwise returns FALSE.

Note: this function must not be called until the QApplication object has been created.

bool QGLFormat::hasOverlay () const

Returns TRUE if overlay plane is enabled; otherwise returns FALSE.

Overlay is disabled by default.

See also setOverlay() [p. 112].

int QGLFormat::plane () const

Returns the plane of this format. The default for normal formats is 0, which means the normal plane. The default for overlay formats is 1, which is the first overlay plane.

See also setPlane() [p. 112].

bool QGLFormat::rgba () const

Returns TRUE if RGBA color mode is set. Returns FALSE if color index mode is set. The default color mode is RGBA.

See also setRgba() [p. 113].

void QGLFormat::setAccum (bool enable)

If *enable* is TRUE enables the accumulation buffer; otherwise disables the accumulation buffer.

The accumulation buffer is disabled by default.

The accumulation buffer is used for create blur effects and multiple exposures.

See also accum() [p. 109].

void QGLFormat::setAlpha (bool enable)

If *enable* is TRUE enables the alpha channel; otherwise disables the alpha channel.

The alpha buffer is disabled by default.

The alpha channel is typically used for implementing transparency or translucency. The A in RGBA specifies the transparency of a pixel.

See also alpha() [p. 109].

void QGLFormat::setDefaultFormat (const QGLFormat & f) [static]

Sets a new default QGLFormat for the application to *f*. For example, to set single buffering as the default instead of double buffering, your main() can contain code like this:

```
QApplication a(argc, argv);
QGLFormat f;
f.setDoubleBuffer( FALSE );
QGLFormat::setDefaultFormat( f );
```

See also defaultFormat() [p. 109].

void QGLFormat::setDefaultOverlayFormat (const QGLFormat & f) [static]

Sets a new default QGLFormat for overlay contexts to *f*. This format is used whenever a QGLWidget is created with a format that hasOverlay() enabled.

For example, to get a double buffered overlay context (if available), use code like this:

```
QGLFormat f = QGLFormat::defaultOverlayFormat();
f.setDoubleBuffer( TRUE );
QGLFormat::setDefaultOverlayFormat( f );
```

As usual, you can find out after widget creation whether the underlying OpenGL system was able to provide the requested specification:

```
// ...continued from above
MyGLWidget* myWidget = new MyGLWidget( QGLFormat( QGL::HasOverlay ), ... );
if ( myWidget->format().hasOverlay() ) {
  // Yes, we got an overlay, let's check _its_ format:
  QGLContext* olContext = myWidget->overlayContext();
  if ( olContext->format().doubleBuffer() )
      ; // yes, we got a double buffered overlay
  else
      ; // no, only single buffered overlays are available
}
```

See also defaultOverlayFormat() [p. 109].

void QGLFormat::setDepth (bool enable)

If *enable* is true enables the depth buffer; otherwise disables the depth buffer.

The depth buffer is enabled by default.

The purpose of a depth buffer (or z-buffering) is to remove hidden surfaces. Pixels are assigned z values based on the distance to the viewer. A pixel with a high z value is closer to the viewer than a pixel with a low z value. This information is used to decide whether to draw a pixel or not.

See also depth() [p. 109].

void QGLFormat::setDirectRendering (bool enable)

If *enable* is TRUE enables direct rendering; otherwise disables direct rendering.

Direct rendering is enabled by default.

Enabling this option will make OpenGL bypass the underlying window system and render directly from hardware to the screen, if this is supported by the system.

See also directRendering() [p. 110].

void QGLFormat::setDoubleBuffer (bool enable)

If *enable* is true sets double buffering; otherwise sets single buffering.

Double buffering is enabled by default.

Double buffering is a technique where graphics are rendered on an off-screen buffer and not directly to the screen. When the drawing has been completed, the program calls a swapBuffers function to exchange the screen contents with the buffer. The result is flicker-free drawing and often better performance.

See also doubleBuffer() [p. 110], QGLContext::swapBuffers() [p. 105] and QGLWidget::swapBuffers() [p. 122].

void QGLFormat::setOption (FormatOption opt)

Sets the format option to *opt*.

See also testOption() [p. 113].

void QGLFormat::setOverlay (bool enable)

If *enable* is TRUE enables an overlay plane; otherwise disables the overlay plane.

Enabling the overlay plane will cause QGLWidget to create an additional context in an overlay plane. See the QGLWidget documentation for further information.

See also hasOverlay() [p. 110].

void QGLFormat::setPlane (int plane)

Sets the requested plane to *plane*. 0 is the normal plane, 1 is the first overlay plane, 2 is the second overlay plane, etc.; -1, -2, etc. are underlay planes.

Note that in contrast to other format specifications, the plane specifications will be matched exactly. This means that if you specify a plane that the underlying OpenGL system cannot provide, an invalidQGLWidget will be created.

See also plane() [p. 110].

void QGLFormat::setRgba (bool enable)

If *enable* is TRUE sets RGBA mode. If *enable* is FALSE sets color index mode.

The default color mode is RGBA.

RGBA is the preferred mode for most OpenGL applications. In RGBA color mode you specify colors as red + green + blue + alpha quadruplets.

In color index mode you specify an index into a color lookup table.

See also rgba() [p. 110].

void QGLFormat::setStencil (bool enable)

If *enable* is TRUE enables the stencil buffer; otherwise disables the stencil buffer.

The stencil buffer is disabled by default.

The stencil buffer masks certain parts of the drawing area so that masked parts are not drawn on.

See also stencil() [p. 113].

void QGLFormat::setStereo (bool enable)

If *enable* is TRUE enables stereo buffering; otherwise disables stereo buffering.

Stereo buffering is disabled by default.

Stereo buffering provides extra color buffers to generate left-eye and right-eye images.

See also stereo() [p. 113].

bool QGLFormat::stencil () const

Returns TRUE if the stencil buffer is enabled; otherwise returns FALSE. The stencil buffer is disabled by default.

See also setStencil() [p. 113].

bool QGLFormat::stereo () const

Returns TRUE if stereo buffering is enabled; otherwise returns FALSE. Stereo buffering is disabled by default.

See also setStereo() [p. 113].

bool QGLFormat::testOption (FormatOption opt) const

Returns TRUE if format option *opt* is set; otherwise returns FALSE.

See also setOption() [p. 112].

QGLWidget Class Reference

The QGLWidget class is a widget for rendering OpenGL graphics.

This class is part of the **OpenGL** module.

```
#include <qgl.h>
```

Inherits QWidget [Widgets with Qt] and QGL [p. 96].

Public Members

- **QGLWidget** (QWidget * parent = 0, const char * name = 0, const QGLWidget * shareWidget = 0, WFlags f = 0)
- **QGLWidget** (const QGLFormat & format, QWidget * parent = 0, const char * name = 0, const QGLWidget * shareWidget = 0, WFlags f = 0)
- **~QGLWidget** ()
- void **qglColor** (const QColor & c) const
- void **qglClearColor** (const QColor & c) const
- bool **isValid** () const
- bool **isSharing** () const
- virtual void **makeCurrent** ()
- bool **doubleBuffer** () const
- virtual void **swapBuffers** ()
- QGLFormat **format** () const
- const QGLContext * **context** () const
- virtual QPixmap **renderPixmap** (int w = 0, int h = 0, bool useContext = FALSE)
- virtual QImage **grabFrameBuffer** (bool withAlpha = FALSE)
- virtual void **makeOverlayCurrent** ()
- const QGLContext * **overlayContext** () const
- const QGLColormap & **colormap** () const
- void **setColormap** (const QGLColormap & cmap)

Public Slots

- virtual void **updateGL** ()
- virtual void **updateOverlayGL** ()

Static Public Members

- QImage **convertToGLFormat** (const QImage & img)

Protected Members

- ■ virtual void **initializeGL** ()
- ■ virtual void **resizeGL** (int width, int height)
- ■ virtual void **paintGL** ()
- ■ virtual void **initializeOverlayGL** ()
- ■ virtual void **resizeOverlayGL** (int width, int height)
- ■ virtual void **paintOverlayGL** ()
- ■ void **setAutoBufferSwap** (bool on)
- ■ bool **autoBufferSwap** () const
- ■ virtual void **paintEvent** (QPaintEvent *)
- ■ virtual void **resizeEvent** (QResizeEvent *)
- ■ virtual void **glInit** ()
- • virtual void **glDraw** ()

Detailed Description

The QGLWidget class is a widget for rendering OpenGL graphics.

QGLWidget provides functionality for displaying OpenGL* graphics integrated into a Qt application. It is very simple to use. You inherit from it and use the subclass like any other QWidget, except that instead of drawing the widget's contents using QPainter etc. you use the standard OpenGL rendering commands.

QGLWidget provides three convenient virtual functions that you can reimplement in your subclass to perform the typical OpenGL tasks:

- • paintGL() - Renders the OpenGL scene. Gets called whenever the widget needs to be updated.
- • resizeGL() - Sets up the OpenGL viewport, projection, etc. Gets called whenever the the widget has been resized (and also when it shown for the first time because all newly created widgets get a resize event automatically).
- • initializeGL() - Sets up the OpenGL rendering context, defines display lists, etc. Gets called once before the first time resizeGL() or paintGL() is called.

Here is a rough outline of how your QGLWidget subclass may look:

```
class MyGLDrawer : public QGLWidget
{
    Q_OBJECT        // must include this if you use Qt signals/slots

public:
    MyGLDrawer( QWidget *parent, const char *name )
        : QGLWidget(parent,name) {}

protected:

    void initializeGL()
    {
      // Set up the rendering context, define display lists etc.:
      ...
      glClearColor( 0.0, 0.0, 0.0, 0.0 );
      glEnable(GL_DEPTH_TEST);
      ...
    }
```

```
        void resizeGL( int w, int h )
        {
          // setup viewport, projection etc.:
          glViewport( 0, 0, (GLint)w, (GLint)h );
          ...
          glFrustum( ... );
          ...
        }

        void paintGL()
        {
          // draw the scene:
          ...
          glRotatef( ... );
          glMaterialfv( ... );
          glBegin( GL_QUADS );
          glVertex3f( ... );
          glVertex3f( ... );
          ...
          glEnd();
          ...
        }

    };
```

If you need to trigger a repaint from places other than paintGL() (a typical example is when using timers to animate scenes), you should call the widget's updateGL() function.

Your widget's OpenGL rendering context is made current when paintGL(), resizeGL(), or initializeGL() is called. If you need to call the standard OpenGL API functions from other places (e.g. in your widget's constructor or in your own paint functions), you must call makeCurrent() first.

QGLWidget provides advanced functions for requesting a new display format and you can even set a new rendering context.

You can achieve sharing of OpenGL display lists between QGLWidgets (see the documentation of the QGLWidget constructors for details).

Overlays

The QGLWidget creates a GL overlay context in addition to the normal context if overlays are supported by the underlying system.

If you want to use overlays, you specify it in the format. (Note: Overlay must be requested in the format passed to the QGLWidget constructor.) Your GL widget should also implement some or all of these virtual methods:

- paintOverlayGL()
- resizeOverlayGL()
- initializeOverlayGL()

These methods work in the same way as the normal paintGL() etc. functions, except that they will be called when the overlay context is made current. You can explicitly make the overlay context current by using makeOverlayCurrent(), and you can access the overlay context directly (e.g. to ask for its transparent color) by calling overlayContext().

QGLWidget overlay support is only currently implemented for the X11 window system. The Windows implementation is experimental.

On X servers in which the default visual is in an overlay plane, non-GL Qt windows can also be used for overlays. See the examples/opengl/overlay_x11 example program for details.

* OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QGLWidget::QGLWidget (QWidget * parent = 0, const char * name = 0, const QGLWidget * shareWidget = 0, WFlags f = 0)

Constructs an OpenGL widget with a *parent* widget and a *name*.

The default format is used. The widget will be invalid if the system has no OpenGL support.

The *parent*, *name* and widget flag, *f*, arguments are passed to the QWidget constructor.

If the *shareWidget* parameter points to a valid QGLWidget, this widget will share OpenGL display lists with *shareWidget*. If this widget and *shareWidget* have different formats, display list sharing may fail. You can check whether display list sharing succeeded by calling isSharing().

The initialization of OpenGL rendering state, etc. should be done by overriding the initializeGL() function, rather than in the constructor of your QGLWidget subclass.

See also QGLFormat::defaultFormat() [p. 109].

QGLWidget::QGLWidget (const QGLFormat & format, QWidget * parent = 0, const char * name = 0, const QGLWidget * shareWidget = 0, WFlags f = 0)

Constructs an OpenGL widget with parent *parent*, called *name*.

The *format* argument specifies the desired rendering options. If the underlying OpenGL/Window system cannot satisfy all the features requested in *format*, the nearest subset of features will be used. After creation, the format() method will return the actual format obtained.

The widget will be invalid if the system has no OpenGL support.

The *parent*, *name* and widget flag, *f*, arguments are passed to the QWidget constructor.

If the *shareWidget* parameter points to a valid QGLWidget, this widget will share OpenGL display lists with *shareWidget*. If this widget and *shareWidget* have different formats, display list sharing may fail. You can check whether display list sharing succeeded by calling isSharing().

The initialization of OpenGL rendering state, etc. should be done by overriding the initializeGL() function, rather than in the constructor of your QGLWidget subclass.

See also QGLFormat::defaultFormat() [p. 109] and isValid() [p. 119].

QGLWidget::~QGLWidget ()

Destroys the widget.

bool QGLWidget::autoBufferSwap () const [protected]

Returns TRUE if the widget is doing automatic GL buffer swapping; otherwise returns FALSE.

See also setAutoBufferSwap() [p. 121].

const QGLColormap & QGLWidget::colormap () const

Returns the colormap for this widget.

Usually it is only top-level widgets that can have different colormaps installed. Asking for the colormap of a child widget will return the colormap for the child's top-level widget.

If no colormap has been set for this widget, the QColormap returned will be empty.

See also setColormap() [p. 121].

const QGLContext * QGLWidget::context () const

Returns the context of this widget.

It is possible that the context is not valid (see isValid()), for example, if the underlying hardware does not support the format attributes that were requested.

QImage QGLWidget::convertToGLFormat (const QImage & img) [static]

Converts the image *img* into the unnamed format expected by OpenGL functions such as glTexImage2D(). The returned image is not usable as a QImage, but QImage::width(), QImage::height() and QImage::bits() may be used with OpenGL. The following few lines are from the texture example. Most of the code is irrelevant, so we just quote the few lines we want:

```
QImage tex1, tex2, buf;
if ( !buf.load( "gllogo.bmp" ) ) {  // Load first image from file
```

We create *tex1* (and another variable) for OpenGL, and load a real image into *buf*.

```
tex1 = QGLWidget::convertToGLFormat( buf );  // flipped 32bit RGBA
```

A few lines later, we convert *buf* into OpenGL format and store it in *tex1*.

```
glTexImage2D( GL_TEXTURE_2D, 0, 3, tex1.width(), tex1.height(), 0,
              GL_RGBA, GL_UNSIGNED_BYTE, tex1.bits() );
```

Another function in the same example uses *tex1* with OpenGL.

Example: opengl/texture/gltexobj.cpp.

bool QGLWidget::doubleBuffer () const

Returns TRUE if the contained GL rendering context has double buffering; otherwise returns FALSE.

See also QGLFormat::doubleBuffer() [p. 110].

QGLFormat QGLWidget::format () const

Returns the format of the contained GL rendering context.

void QGLWidget::glDraw () [virtual protected]

Executes the virtual function paintGL().

The widget's rendering context will become the current context and initializeGL() will be called if it hasn't already been called.

void QGLWidget::glInit () [virtual protected]

Initializes OpenGL for this widget's context. Calls the virtual function initializeGL().

QImage QGLWidget::grabFrameBuffer (bool withAlpha = FALSE) [virtual]

Returns an image of the frame buffer. If *withAlpha* is TRUE the alpha channel is included.

void QGLWidget::initializeGL () [virtual protected]

This virtual function is called once before the first call to paintGL() or resizeGL(), and then once whenever the widget has been assigned a new QGLContext. Reimplement it in a subclass.

This function should set up any required OpenGL context rendering flags, defining display lists, etc.

There is no need to call makeCurrent() because this has already been done when this function is called.

void QGLWidget::initializeOverlayGL () [virtual protected]

This virtual function is used in the same manner as initializeGL() except that it operates on the widget's overlay context instead of the widget's main context. This means that initializeOverlayGL() is called once before the first call to paintOverlayGL() or resizeOverlayGL(). Reimplement it in a subclass.

This function should set up any required OpenGL context rendering flags, defining display lists, etc. for the overlay context.

There is no need to call makeOverlayCurrent() because this has already been done when this function is called.

bool QGLWidget::isSharing () const

Returns TRUE if display list sharing with another QGLWidget was requested in the constructor, and the GL system was able to provide it; otherwise returns FALSE. The GL system may fail to provide display list sharing if the two QGLWidgets use different formats.

See also format() [p. 118].

bool QGLWidget::isValid () const

Returns TRUE if the widget has a valid GL rendering context; otherwise returns FALSE. A widget will be invalid if the system has no OpenGL support.

void QGLWidget::makeCurrent () [virtual]

Makes this widget the current widget for OpenGL operations, i.e. makes the widget's rendering context the current OpenGL rendering context.

void QGLWidget::makeOverlayCurrent () [virtual]

Makes the overlay context of this widget current. Use this if you need to issue OpenGL commands to the overlay context outside of initializeOverlayGL(), resizeOverlayGL(), and paintOverlayGL().

Does nothing if this widget has no overlay.

See also makeCurrent() [p. 119].

const QGLContext * QGLWidget::overlayContext () const

Returns the overlay context of this widget, or 0 if this widget has no overlay.

See also context() [p. 118].

void QGLWidget::paintEvent (QPaintEvent *) [virtual protected]

Handles paint events. Will cause the virtual paintGL() function to be called.

The widget's rendering context will become the current context and initializeGL() will be called if it hasn't already been called.

Reimplemented from QWidget [Widgets with Qt].

void QGLWidget::paintGL () [virtual protected]

This virtual function is called whenever the widget needs to be painted. Reimplement it in a subclass.

There is no need to call makeCurrent() because this has already been done when this function is called.

void QGLWidget::paintOverlayGL () [virtual protected]

This virtual function is used in the same manner as paintGL() except that it operates on the widget's overlay context instead of the widget's main context. This means that paintOverlayGL() is called whenever the widget's overlay needs to be painted. Reimplement it in a subclass.

There is no need to call makeOverlayCurrent() because this has already been done when this function is called.

void QGLWidget::qglClearColor (const QColor & c) const

Convenience function for specifying the clearing color to OpenGL. Calls glClearColor (in RGBA mode) or glClearIndex (in color-index mode) with the color c. Applies to the current GL context.

See also qglColor() [p. 120], QGLContext::currentContext() [p. 103] and QColor [p. 74].

void QGLWidget::qglColor (const QColor & c) const

Convenience function for specifying a drawing color to OpenGL. Calls glColor3 (in RGBA mode) or glIndex (in color-index mode) with the color c. Applies to the current GL context.

See also qglClearColor() [p. 120], QGLContext::currentContext() [p. 103] and QColor [p. 74].

QPixmap QGLWidget::renderPixmap (int w = 0, int h = 0, bool useContext = FALSE) [virtual]

Renders the current scene on a pixmap and returns the pixmap.

You may use this method on both visible and invisible QGLWidgets.

This method will create a pixmap and a temporary QGLContext to render on the pixmap. It will then call initializeGL(), resizeGL(), and paintGL() on this context. Finally, the widget's original GL context is restored.

The size of the pixmap will be *w* pixels wide and *h* pixels high unless one of these parameters is 0 (the default), in which case the pixmap will have the same size as the widget.

If *useContext* is TRUE, this method will try to be more efficient by using the existing GL context to render the pixmap. The default is FALSE. Only use TRUE if you understand the risks.

Overlays are not rendered onto the pixmap.

If the GL rendering context and the desktop have different bit depths, the result will most likely look surprising.

void QGLWidget::resizeEvent (QResizeEvent *) [virtual protected]

Handles resize events. Calls the virtual function resizeGL().

Reimplemented from QWidget [Widgets with Qt].

void QGLWidget::resizeGL (int width, int height) [virtual protected]

This virtual function is called whenever the widget has been resized. The new size is passed in *width* and *height*. Reimplement it in a subclass.

There is no need to call makeCurrent() because this has already been done when this function is called.

void QGLWidget::resizeOverlayGL (int width, int height) [virtual protected]

This virtual function is used in the same manner as paintGL() except that it operates on the widget's overlay context instead of the widget's main context. This means that resizeOverlayGL() is called whenever the widget has been resized. The new size is passed in *width* and *height*. Reimplement it in a subclass.

There is no need to call makeOverlayCurrent() because this has already been done when this function is called.

void QGLWidget::setAutoBufferSwap (bool on) [protected]

If *on* is TRUE automatic GL buffer swapping is switched on; otherwise it is switched off.

If *on* is TRUE and the widget is using a double-buffered format, the background and foreground GL buffers will automatically be swapped after each time the paintGL() function has been called.

The buffer auto-swapping is on by default.

See also autoBufferSwap() [p. 117], doubleBuffer() [p. 118] and swapBuffers() [p. 122].

void QGLWidget::setColormap (const QGLColormap & cmap)

Set the colormap for this widget to *cmap*. Usually it is only top-level widgets that can have colormaps installed.

See also colormap() [p. 118].

void QGLWidget::swapBuffers () [virtual]

Swaps the screen contents with an off-screen buffer. This only works if the widget's format specifies double buffer mode.

Normally, there is no need to explicitly call this function because it is done automatically after each widget repaint, i.e. each time after paintGL() has been executed.

See also doubleBuffer() [p. 118], setAutoBufferSwap() [p. 121] and QGLFormat::setDoubleBuffer() [p. 112].

void QGLWidget::updateGL () [virtual slot]

Updates the widget by calling glDraw().

void QGLWidget::updateOverlayGL () [virtual slot]

Updates the widget's overlay (if any). Will cause the virtual function paintOverlayGL() to be executed.

The widget's rendering context will become the current context and initializeGL() will be called if it hasn't already been called.

QIconSet Class Reference

The QIconSet class provides a set of icons with different styles and sizes.

```
#include <qiconset.h>
```

Public Members

- enum **Size** { Automatic, Small, Large }
- enum **Mode** { Normal, Disabled, Active }
- enum **State** { On, Off }
- **QIconSet** ()
- **QIconSet** (const QPixmap & pixmap, Size size = Automatic)
- **QIconSet** (const QPixmap & smallPix, const QPixmap & largePix)
- **QIconSet** (const QIconSet & other)
- virtual **~QIconSet** ()
- void **reset** (const QPixmap & pm, Size size)
- virtual void **setPixmap** (const QPixmap & pm, Size size, Mode mode = Normal, State state = Off)
- virtual void **setPixmap** (const QString & fileName, Size size, Mode mode = Normal, State state = Off)
- QPixmap **pixmap** (Size size, Mode mode, State state = Off) const
- QPixmap **pixmap** (Size size, bool enabled, State state = Off) const
- QPixmap **pixmap** () const
- bool **isGenerated** (Size size, Mode mode, State state = Off) const
- void **clearGenerated** ()
- bool **isNull** () const
- void **detach** ()
- QIconSet & **operator=** (const QIconSet & other)

Static Public Members

- void **setIconSize** (Size s, const QSize & size)
- const QSize & **iconSize** (Size s)

Detailed Description

The QIconSet class provides a set of icons with different styles and sizes.

A QIconSet can generate smaller, larger, active, and disabled pixmaps from the set of icons it is given. Such pixmaps are used by QToolButton, QHeader, QPopupMenu, etc. to show an icon representing a particular action.

The simplest use of QIconSet is to create one from a QPixmap and then use it, allowing Qt to work out all the required icon styles and sizes. For example:

```
QToolButton *tb = new QToolButton( QIconSet( QPixmap("open.xpm") ), ... );
```

Using whichever pixmap(s) you specify as a base, QIconSet provides a set of six icons, each with a Size and a Mode:

- *Small Normal* - can only be calculated from Large Normal.
- *Small Disabled* - calculated from Large Disabled or Small Normal.
- *Small Active* - same as Small Normal unless you set it.
- *Large Normal* - can only be calculated from Small Normal.
- *Large Disabled* - calculated from Small Disabled or Large Normal.
- *Large Active* - same as Large Normal unless you set it.

An additional set of six icons can be provided for widgets that have an "On" or "Off" state, like checkable menu items or toggleable toolbuttons. If you provide pixmaps for the "On" state, but not for the "Off" state, the QIconSet will provide the "Off" pixmaps. You may specify icons for both states in you wish.

You can set any of the icons using setPixmap().

When you retrieve a pixmap using pixmap(Size,Mode,State), QIconSet will return the icon that has been set or previously generated for that size, mode and state combination. If no pixmap has been set or previously generated for the combination QIconSet will generate a pixmap based on the pixmap(s) it has been given, cache the generated pixmap for later use, and return it. The isGenerated() function returns TRUE if an icon was generated by QIconSet.

The Disabled appearance is computed using a "shadow" algorithm that produces results very similar to those used in Microsoft Windows 95.

The Active appearance is identical to the Normal appearance unless you use setPixmap() to set it to something special.

When scaling icons, QIconSet uses smooth scaling, which can partially blend the color component of pixmaps. If the results look poor, the best solution is to supply pixmaps in both large and small sizes.

You can use the static function setIconSize() to set the preferred size of the generated large/small icons. The default small size is 22x22 (compatible with Qt 2.x), while the default large size is 32x32. Please note that these sizes only affect generated icons.

QIconSet provides a function, isGenerated(), that indicates whether an icon was set by the application programmer or computed by QIconSet itself.

Making Classes that use QIconSet

If you write your own widgets that have an option to set a small pixmap, consider allowing a QIconSet to be set for that pixmap. The Qt class QToolButton is an example of such a widget.

Provide a method to set a QIconSet, and when you draw the icon, choose whichever icon is appropriate for the current state of your widget. For example:

```
void MyWidget::drawIcon( QPainter* p, QPoint pos )
{
    p->drawPixmap( pos, icons->pixmap(QIconSet::Small, isEnabled()) );
}
```

You might also make use of the Active mode, perhaps making your widget Active when the mouse is over the widget (see QWidget::enterEvent()), while the mouse is pressed pending the release that will activate the function, or when it is the currently selected item. If the widget can be toggled, the "On" mode might be used to draw a different icon.

See also QPixmap [p. 225], QLabel [Widgets with Qt], QToolButton [Dialogs and Windows with Qt], QPopupMenu [Dialogs and Windows with Qt], QMainWindow::usesBigPixmaps [Dialogs and Windows with Qt], GUI Design Handbook: Iconic Label, Microsoft Icon Gallery, Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Type Documentation

QIconSet::Mode

This enum type describes the mode for which a pixmap is intended to be used. The currently defined modes are:

- `QIconSet::Normal` - Display the pixmap when the user is not interacting with the icon, but the functionality represented by the icon is available.
- `QIconSet::Disabled` - Display the pixmap when the functionality represented by the icon is not available.
- `QIconSet::Active` - Display the pixmap when the functionality represented by the icon is available and the user is interacting with the icon, for example, moving the mouse over it or clicking it.

QIconSet::Size

This enum type describes the size at which a pixmap is intended to be used. The currently defined sizes are:

- `QIconSet::Automatic` - The size of the pixmap is determined from its pixel size. This is a useful default.
- `QIconSet::Small` - The pixmap is the smaller of two.
- `QIconSet::Large` - The pixmap is the larger of two.

If a Small pixmap is not set by QIconSet::setPixmap(), the Large pixmap will be automatically scaled down to the size of a small pixmap to generate the Small pixmap when required. Similarly, a Small pixmap will be automatically scaled up to generate a Large pixmap. The preferred sizes for large/small generated icons can be set using setIconSize().

See also setIconSize() [p. 127], iconSize() [p. 126], setPixmap() [p. 127], pixmap() [p. 127] and QMainWindow::usesBigPixmaps [Dialogs and Windows with Qt].

QIconSet::State

This enum describes the state for which a pixmap is intended to be used. The *state* can be:

- `QIconSet::Off` - Display the pixmap when the widget is in an "off" state
- `QIconSet::On` - Display the pixmap when the widget is in an "on" state

See also setPixmap() [p. 127] and pixmap() [p. 127].

Member Function Documentation

QIconSet::QIconSet ()

Constructs a null icon set. Use setPixmap(), reset(), or operator=() to set some pixmaps.

See also reset() [p. 127].

QIconSet::QIconSet (const QPixmap & pixmap, Size size = Automatic)

Constructs an icon set for which the Normal pixmap is *pixmap*, which is assumed to be of size *size*.

The default for *size* is Automatic, which means that QIconSet will determine whether the pixmap is Small or Large from its pixel size. Pixmaps less than the width of a small generated icon are considered to be Small. You can use setIconSize() to set the preferred size of a generated icon.

See also setIconSize() [p. 127] and reset() [p. 127].

QIconSet::QIconSet (const QPixmap & smallPix, const QPixmap & largePix)

Creates an iconset which uses the pixmap *smallPix* for for displaying a small icon, and the pixmap *largePix* for displaying a large icon.

QIconSet::QIconSet (const QIconSet & other)

Constructs a copy of *other*. This is very fast.

QIconSet::~QIconSet () [virtual]

Destroys the icon set and frees any allocated resources.

void QIconSet::clearGenerated ()

Clears all generated pixmaps.

void QIconSet::detach ()

Detaches this icon set from others with which it may share data.

You will never need to call this function; other QIconSet functions call it as necessary.

const QSize & QIconSet::iconSize (Size s) [static]

If *s* is Small, returns the preferred size of a small generated icon; if *s* is Large, returns the preferred size of a large generated icon.

See also setIconSize() [p. 127].

bool QIconSet::isGenerated (Size size, Mode mode, State state = Off) const

Returns TRUE if the pixmap with size *size*, mode *mode* and state *state* has been generated; otherwise returns FALSE.

bool QIconSet::isNull () const

Returns TRUE if the icon set is empty; otherwise returns FALSE.

QIconSet & QIconSet::operator= (const QIconSet & other)

Assigns *other* to this icon set and returns a reference to this icon set.

This is very fast.

See also detach() [p. 126].

QPixmap QIconSet::pixmap (Size size, Mode mode, State state = Off) const

Returns a pixmap with size *size*, mode *mode* and state *state*, generating one if necessary. Generated pixmaps are cached.

QPixmap QIconSet::pixmap (Size size, bool enabled, State state = Off) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a pixmap with size *size*, state *state* and a Mode which is Normal if *enabled* is TRUE, or Disabled if *enabled* is FALSE.

QPixmap QIconSet::pixmap () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the pixmap originally provided to the constructor or to reset(). This is the Normal pixmap of unspecified Size.

See also reset() [p. 127].

void QIconSet::reset (const QPixmap & pm, Size size)

Sets this icon set to use pixmap *pm* for the Normal pixmap, assuming it to be of size *size*.

This is equivalent to assigning QIconSet(*pm*, *size*) to this icon set.

This function does nothing if *pm* is a null pixmap.

void QIconSet::setIconSize (Size s, const QSize & size) [static]

Set the preferred size for all small or large icons that are generated after this call. If *s* is Small, sets the preferred size of small generated icons to *size*. Similarly, if *s* is Large, sets the preferred size of large generated icons to *size*.

Note that cached icons will not be regenerated, so it is recommended that you set the preferred icon sizes before generating any icon sets.

See also iconSize() [p. 126].

void QIconSet::setPixmap (const QPixmap & pm, Size size, Mode mode = Normal, State state = Off) [virtual]

Sets this icon set to provide pixmap *pm* for size *size*, mode *mode* and state *state*. The icon set may also use *pm* for generating other pixmaps if they are not explicitly set.

The *size* can be one of Automatic, Large or Small. If Automatic is used, QIconSet will determine if the pixmap is Small or Large from its pixel size.

Pixmaps less than the width of a small generated icon are considered to be Small. You can use setIconSize() to set the preferred size of a generated icon.

This function does nothing if *pm* is a null pixmap.

See also reset() [p. 127].

void QIconSet::setPixmap (const QString & fileName, Size size, Mode mode = Normal, State state = Off) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets this icon set to load the file called *fileName* as a pixmap and use it for size *size*, mode *mode* and state *state*. The icon set may also use this pixmap for generating other pixmaps if they are not explicitly set.

The *size* can be one of Automatic, Large or Small. If Automatic is used, QIconSet will determine if the pixmap is Small or Large from its pixel size. Pixmaps less than the width of a small generated icon are considered to be Small. You can use setIconSize() to set the preferred size of a generated icon.

QImage Class Reference

The QImage class provides a hardware-independent pixmap representation with direct access to the pixel data.

`#include <qimage.h>`

Public Members

- enum **Endian** { IgnoreEndian, BigEndian, LittleEndian }
- **QImage** ()
- **QImage** (int w, int h, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)
- **QImage** (const QSize & size, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)
- **QImage** (const QString & fileName, const char * format = 0)
- **QImage** (const char * const xpm[])
- **QImage** (const QByteArray & array)
- **QImage** (uchar * yourdata, int w, int h, int depth, QRgb * colortable, int numColors, Endian bitOrder)
- **QImage** (const QImage & image)
- **~QImage** ()
- QImage & **operator=** (const QImage & image)
- QImage & **operator=** (const QPixmap & pixmap)
- bool **operator==** (const QImage & i) const
- bool **operator!=** (const QImage & i) const
- void **detach** ()
- QImage **copy** () const
- QImage **copy** (int x, int y, int w, int h, int conversion_flags = 0) const
- QImage **copy** (const QRect & r) const
- bool **isNull** () const
- int **width** () const
- int **height** () const
- QSize **size** () const
- QRect **rect** () const
- int **depth** () const
- int **numColors** () const
- Endian **bitOrder** () const
- QRgb **color** (int i) const
- void **setColor** (int i, QRgb c)
- void **setNumColors** (int numColors)
- bool **hasAlphaBuffer** () const
- void **setAlphaBuffer** (bool enable)
- bool **allGray** () const
- bool **isGrayscale** () const

- uchar * **bits** ( ) const
- uchar * **scanLine** (int i) const
- uchar ** **jumpTable** ( ) const
- QRgb * **colorTable** ( ) const
- int **numBytes** ( ) const
- int **bytesPerLine** ( ) const
- bool **create** (int width, int height, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)
- bool **create** (const QSize &, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)
- void **reset** ( )
- void **fill** (uint pixel)
- void **invertPixels** (bool invertAlpha = TRUE)
- QImage **convertDepth** (int depth) const
- QImage **convertDepthWithPalette** (int d, QRgb * palette, int palette_count, int conversion_flags = 0) const
- QImage **convertDepth** (int depth, int conversion_flags) const
- QImage **convertBitOrder** (Endian bitOrder) const
- enum **ScaleMode** { ScaleFree, ScaleMin, ScaleMax }
- QImage **smoothScale** (int w, int h, ScaleMode mode = ScaleFree) const
- QImage **smoothScale** (const QSize & s, ScaleMode mode = ScaleFree) const
- QImage **scale** (int w, int h, ScaleMode mode = ScaleFree) const
- QImage **scale** (const QSize & s, ScaleMode mode = ScaleFree) const
- QImage **scaleWidth** (int w) const
- QImage **scaleHeight** (int h) const
- QImage **xForm** (const QWMatrix & matrix) const
- QImage **createAlphaMask** (int conversion_flags = 0) const
- QImage **createHeuristicMask** (bool clipTight = TRUE) const
- QImage **mirror** ( ) const
- QImage **mirror** (bool horizontal, bool vertical) const
- QImage **swapRGB** ( ) const
- bool **load** (const QString & fileName, const char * format = 0)
- bool **loadFromData** (const uchar * buf, uint len, const char * format = 0)
- bool **loadFromData** (QByteArray buf, const char * format = 0)
- bool **save** (const QString & fileName, const char * format, int quality = -1) const
- bool **valid** (int x, int y) const
- int **pixelIndex** (int x, int y) const
- QRgb **pixel** (int x, int y) const
- void **setPixel** (int x, int y, uint index_or_rgb)
- int **dotsPerMeterX** ( ) const
- int **dotsPerMeterY** ( ) const
- void **setDotsPerMeterX** (int x)
- void **setDotsPerMeterY** (int y)
- QPoint **offset** ( ) const
- void **setOffset** (const QPoint & p)
- QValueList<QImageTextKeyLang> **textList** ( ) const
- QStringList **textLanguages** ( ) const
- QStringList **textKeys** ( ) const
- QString **text** (const char * key, const char * lang = 0) const
- QString **text** (const QImageTextKeyLang & kl) const
- void **setText** (const char * key, const char * lang, const QString & s)

Static Public Members

- Endian **systemBitOrder** ()
- Endian **systemByteOrder** ()
- const char * **imageFormat** (const QString & fileName)
- QStrList **inputFormats** ()
- QStrList **outputFormats** ()
- QStringList **inputFormatList** ()
- QStringList **outputFormatList** ()

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QImage & image)
- QDataStream & **operator>>** (QDataStream & s, QImage & image)

Detailed Description

The QImage class provides a hardware-independent pixmap representation with direct access to the pixel data.

It is one of the two classes Qt provides for dealing with images, the other being QPixmap. QImage is designed and optimized for I/O and for direct pixel access/manipulation. QPixmap is designed and optimized for drawing. There are (slow) functions to convert between QImage and QPixmap: QPixmap::convertToImage() and QPixmap::convertFromImage().

An image has the parameters width, height and depth (bits per pixel, bpp), a color table and the actual pixels. QImage supports 1-bpp, 8-bpp and 32-bpp image data. 1-bpp and 8-bpp images use a color lookup table; the pixel value is a color table index.

32-bpp images encode an RGB value in 24 bits and ignore the color table. The most significant byte is used for the alpha buffer.

An entry in the color table is an RGB triplet encoded as `uint`. Use the qRed, qGreen and qBlue functions (qcolor.h) to access the components, and qRgb to make an RGB triplet (see the QColor class documentation).

1-bpp (monochrome) images have a color table with maximum two colors. There are two different formats: big endian (MSB first) or little endian (LSB first) bit order. To access a single bit you will have to do some bit shifts:

```
QImage image;
// sets bit at (x,y) to 1
if ( image.bitOrder() == QImage::LittleEndian )
    *(image.scanLine(y) + (x >> 3)) |= 1 << (x & 7);
else
    *(image.scanLine(y) + (x >> 3)) |= 1 << (7 -(x & 7));
```

If this looks complicated, it might be a good idea to convert the 1-bpp image to an 8-bpp image using convertDepth().

8-bpp images are much easier to work with than 1-bpp images because they have a single byte per pixel:

```
QImage image;
// set entry 19 in the color table to yellow
image.setColor( 19, qRgb(255,255,0) );
// set 8 bit pixel at (x,y) to value yellow (in color table)
*(image.scanLine(y) + x) = 19;
```

32-bpp images ignore the color table; instead, each pixel contains the RGB triplet. 24 bits contain the RGB value; the most significant byte is reserved for the alpha buffer.

```
QImage image;
// sets 32 bit pixel at (x,y) to yellow.
uint *p = (uint *)image.scanLine(y) + x;
*p = qRgb(255,255,0);
```

On Qt/Embedded, scanlines are aligned to the pixel depth and may be padded to any degree, while on all other platforms, the scanlines are 32-bit aligned for all depths. The constructor taking a

```
 uchar*
```

argument always expects 32-bit aligned data. On Qt/Embedded, an additional constructor allows the number of byte-per-line to be specified.

QImage supports a variety of methods for getting information about the image, for example, colorTable(), allGray(), isGrayscale(), bitOrder(), bytesPerLine(), depth(), dotsPerMeterX() and dotsPerMeterY(), hasAlphaBuffer(), num-Bytes(), numColors(), and width() and height().

Pixel colors are retrieved with pixel() and set with setPixel().

QImage also supports a number of functions for creating a new image that is a transformed version of the original. For example, copy(), convertBitOrder(), convertDepth(), createAlphaMask(), createHeuristicMask(), mirror(), scale(), smoothScale(), swapRGB() and xForm(). There are also functions for changing attributes of an image in-place, for example, setAlphaBuffer(), setColor(), setDotsPerMeterX() and setDotsPerMeterY() and setNumColors().

Images can be loaded and saved in the supported formats. Images are saved to a file with save(). Images are loaded from a file with load() (or in the constructor) or from an array of data with loadFromData(). The lists of supported formats are available from inputFormatList() and outputFormatList().

Strings of text may be added to images using setText().

The QImage class uses explicit sharing, similar to that used by QMemArray.

New image formats can be added as plugins.

See also QImageIO [p. 154], QPixmap [p. 225], Shared Classes [Programming with Qt], Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Type Documentation

QImage::Endian

This enum type is used to describe the endianness of the CPU and graphics hardware.

The current values are:

- `QImage::IgnoreEndian` - Endianness does not matter. Useful for some operations that are independent of endianness.
- `QImage::BigEndian` - Network byte order, as on SPARC and Motorola CPUs.
- `QImage::LittleEndian` - PC/Alpha byte order.

QImage::ScaleMode

The functions scale() and smoothScale() use different modes for scaling the image. The purpose of these modes is to retain the ratio of the image if this is required.

- QImage::ScaleFree - The image is scaled freely: the resulting image fits exactly into the specified size; the ratio will not necessarily be preserved.
- QImage::ScaleMin - The ratio of the image is preserved and the resulting image is guaranteed to fit into the specified size (it is as large as possible within these constraints) - the image might be smaller than the requested size.
- QImage::ScaleMax - The ratio of the image is preserved and the resulting image fills the whole specified rectangle (it is as small as possible within these constraints) - the image might be larger than the requested size.

Member Function Documentation

QImage::QImage ()

Constructs a null image.

See also isNull() [p. 139].

QImage::QImage (int w, int h, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)

Constructs an image with *w* width, *h* height, *depth* bits per pixel, *numColors* colors and bit order *bitOrder*.

Using this constructor is the same as first constructing a null image and then calling the create() function.

See also create() [p. 136].

QImage::QImage (const QSize & size, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)

Constructs an image with size *size* pixels, depth *depth* bits, *numColors* and *bitOrder* endianness.

Using this constructor is the same as first constructing a null image and then calling the create() function.

See also create() [p. 136].

QImage::QImage (const QString & fileName, const char * format = 0)

Constructs an image and tries to load it image from the file *fileName*.

If *format* is specified, the loader attempts to read the image using the specified format. If *format* is not specified (which is the default), the loader reads a few bytes from the header to guess the file format.

If the loading of the image failed, this object is a null image.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also load() [p. 139], isNull() [p. 139] and QImageIO [p. 154].

QImage::QImage (const char * const xpm[])

Constructs an image from *xpm*, which must be a valid XPM image.

Errors are silently ignored.

Note that it's possible to squeeze the XPM variable a little bit by using an unusual declaration:

```
static const char * const start_xpm[]={
    "16 15 8 1",
    "a c #cec6bd",
....
```

The extra `const` makes the entire definition read-only, which is slightly more efficient (e.g., when the code is in a shared library) and ROMable when the application is to be stored in ROM.

QImage::QImage (const QByteArray & array)

Constructs an image from the binary data *array*. It tries to guess the file format.

If the loading of the image failed, this object is a null image.

See also loadFromData() [p. 140], isNull() [p. 139] and imageFormat() [p. 138].

QImage::QImage (uchar * yourdata, int w, int h, int depth, QRgb * colortable, int numColors, Endian bitOrder)

Constructs an image *w* pixels wide, *h* pixels high with a color depth of *depth*, that uses an existing memory buffer, *yourdata*. The buffer must remain valid throughout the life of the QImage. The image does not delete the buffer at destruction.

If *colortable* is 0, a color table sufficient for *numColors* will be allocated (and destructed later).

Note that *yourdata* must be 32-bit aligned.

The endianness is given in *bitOrder*.

QImage::QImage (const QImage & image)

Constructs a shallow copy of *image*.

QImage::~QImage ()

Destroys the image and cleans up.

bool QImage::allGray () const

Returns TRUE if all the colors in the image are shades of gray (i.e., their red, green and blue components are equal).

This function is slow for large 16-bit and 32-bit images.

See also isGrayscale() [p. 139].

Endian QImage::bitOrder () const

Returns the bit order for the image.

If it is a 1-bpp image, this function returns either QImage::BigEndian or QImage::LittleEndian.

If it is not a 1-bpp image, this function returns QImage::IgnoreEndian.

See also depth() [p. 137].

uchar * QImage::bits () const

Returns a pointer to the first pixel data. This is equivalent to scanLine(0).

See also numBytes() [p. 140], scanLine() [p. 143] and jumpTable() [p. 139].

Example: opengl/texture/gltexobj.cpp.

int QImage::bytesPerLine () const

Returns the number of bytes per image scanline. This is equivalent to numBytes()/height().

See also numBytes() [p. 140] and scanLine() [p. 143].

QRgb QImage::color (int i) const

Returns the color in the color table at index *i*. The first color is at index 0.

A color value is an RGB triplet. Use the qRed(), qGreen() and qBlue() functions (defined in qcolor.h) to get the color value components.

See also setColor() [p. 143], numColors() [p. 140] and QColor [p. 74].

Example: themes/wood.cpp.

QRgb * QImage::colorTable () const

Returns a pointer to the color table.

See also numColors() [p. 140].

QImage QImage::convertBitOrder (Endian bitOrder) const

Converts the bit order of the image to *bitOrder* and returns the converted image. The original image is not changed.

Returns `*this` if the *bitOrder* is equal to the image bit order, or a null image if this image cannot be converted.

See also bitOrder() [p. 134], systemBitOrder() [p. 145] and isNull() [p. 139].

QImage QImage::convertDepth (int depth, int conversion_flags) const

Converts the depth (bpp) of the image to *depth* and returns the converted image. The original image is not changed.

The *depth* argument must be 1, 8, 16 or 32.

Returns `*this` if *depth* is equal to the image depth, or a null image if this image cannot be converted.

If the image needs to be modified to fit in a lower-resolution result (eg. converting from 32-bit to 8-bit), use the *conversion_flags* to specify how you'd prefer this to happen.

See also Qt::ImageConversionFlags [Additional Functionality with Qt], depth() [p. 137] and isNull() [p. 139].

QImage QImage::convertDepth (int depth) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QImage QImage::convertDepthWithPalette (int d, QRgb * palette, int palette_count, int conversion_flags = 0) const

Note: currently no closest-color search is made. If colors are found that are not in the palette, the palette may not be used at all. This result should not be considered valid because it may change in future implementations.

Currently inefficient for non-32-bit images.

Returns an image with depth *d*, using the *palette_count* colors pointed to by *palette*. If *d* is 1 or 8, the returned image will have its color table ordered the same as *palette*.

If the image needs to be modified to fit in a lower-resolution result (eg. converting from 32-bit to 8-bit), use the *conversion_flags* to specify how you'd prefer this to happen.

See also Qt::ImageConversionFlags [Additional Functionality with Qt].

QImage QImage::copy () const

Returns a deep copy of the image.

See also detach() [p. 137].

QImage QImage::copy (int x, int y, int w, int h, int conversion_flags = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a deep copy of a sub-area of the image.

The returned image is always *w* by *h* pixels in size, and is copied from position *x, y* in this image. In areas beyond this image pixels are filled with pixel 0.

If the image needs to be modified to fit in a lower-resolution result (eg. converting from 32-bit to 8-bit), use the *conversion_flags* to specify how you'd prefer this to happen.

See also bitBlt() [p. 175] and Qt::ImageConversionFlags [Additional Functionality with Qt].

QImage QImage::copy (const QRect & r) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a deep copy of a sub-area of the image.

The returned image has always the size of the rectangle *r*. In areas beyond this image pixels are filled with pixel 0.

bool QImage::create (int width, int height, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)

Sets the image *width*, *height*, *depth*, its number of colors (in *numColors*), and bit order. Returns TRUE if successful, or FALSE if the parameters are incorrect or if memory cannot be allocated.

The *width* and *height* is limited to 32767. *depth* must be 1, 8, or 32. If *depth* is 1, *bitOrder* must be set to either QImage::LittleEndian or QImage::BigEndian. For other depths *bitOrder* must be QImage::IgnoreEndian.

This function allocates a color table and a buffer for the image data. The image data is not initialized.

The image buffer is allocated as a single block that consists of a table of scanline pointers (jumpTable()) and the image data (bits()).

See also fill() [p. 138], width() [p. 146], height() [p. 138], depth() [p. 137], numColors() [p. 140], bitOrder() [p. 134], jumpTable() [p. 139], scanLine() [p. 143], bits() [p. 135], bytesPerLine() [p. 135] and numBytes() [p. 140].

bool QImage::create (const QSize &, int depth, int numColors = 0, Endian bitOrder = IgnoreEndian)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QImage QImage::createAlphaMask (int conversion_flags = 0) const

Builds and returns a 1-bpp mask from the alpha buffer in this image. Returns a null image if alpha buffer mode is disabled.

See QPixmap::convertFromImage() for a description of the *conversion_flags* argument.

The returned image has little-endian bit order, which you can convert to big-endianness using convertBitOrder().

See also createHeuristicMask() [p. 137], hasAlphaBuffer() [p. 138] and setAlphaBuffer() [p. 143].

QImage QImage::createHeuristicMask (bool clipTight = TRUE) const

Creates and returns a 1-bpp heuristic mask for this image. It works by selecting a color from one of the corners, then chipping away pixels of that color starting at all the edges.

The four corners vote for which color is to be masked away. In case of a draw (this generally means that this function is not applicable to the image), the result is arbitrary.

The returned image has little-endian bit order, which you can convert to big-endianness using convertBitOrder().

If *clipTight* is TRUE the mask is just large enough to cover the pixels; otherwise, the mask is larger than the data pixels.

This function disregards the alpha buffer.

See also createAlphaMask() [p. 137].

int QImage::depth () const

Returns the depth of the image.

The image depth is the number of bits used to encode a single pixel, also called bits per pixel (bpp) or bit planes of an image.

The supported depths are 1, 8, 16 and 32.

See also convertDepth() [p. 135].

void QImage::detach ()

Detaches from shared image data and makes sure that this image is the only one referring the data.

If multiple images share common data, this image makes a copy of the data and detaches itself from the sharing mechanism. Nothing is done if there is just a single reference.

See also copy() [p. 136].

Example: themes/wood.cpp.

int QImage::dotsPerMeterX () const

Returns the number of pixels that fit horizontally in a physical meter. This and dotsPerMeterY() define the intended scale and aspect ratio of the image.

See also setDotsPerMeterX() [p. 143].

int QImage::dotsPerMeterY () const

Returns the number of pixels that fit vertically in a physical meter. This and dotsPerMeterX() define the intended scale and aspect ratio of the image.

See also setDotsPerMeterY() [p. 144].

void QImage::fill (uint pixel)

Fills the entire image with the pixel value *pixel*.

If the depth of this image is 1, only the lowest bit is used. If you say fill(0), fill(2), etc., the image is filled with 0s. If you say fill(1), fill(3), etc., the image is filled with 1s. If the depth is 8, the lowest 8 bits are used.

If the depth is 32 and the image has no alpha buffer, the *pixel* value is written to each pixel in the image. If the image has an alpha buffer, only the 24 RGB bits are set and the upper 8 bits (alpha value) are left unchanged.

See also invertPixels() [p. 139], depth() [p. 137], hasAlphaBuffer() [p. 138] and create() [p. 136].

bool QImage::hasAlphaBuffer () const

Returns TRUE if alpha buffer mode is enabled, otherwise FALSE.

See also setAlphaBuffer() [p. 143].

int QImage::height () const

Returns the height of the image.

See also width() [p. 146], size() [p. 144] and rect() [p. 142].

Example: opengl/texture/gltexobj.cpp.

const char * QImage::imageFormat (const QString & fileName) [static]

Returns a string that specifies the image format of the file *fileName*, or null if the file cannot be read or if the format is not recognized.

The QImageIO documentation lists the guaranteed supported image formats, or use QImage::inputFormats() and QImage::outputFormats() to get lists that include the installed formats.

See also load() [p. 139] and save() [p. 142].

QStringList QImage::inputFormatList () [static]

Returns a list of image formats that are supported for image input.

See also outputFormatList() [p. 141], inputFormats() [p. 139] and QImageIO [p. 154].

Example: showimg/showimg.cpp.

QStrList QImage::inputFormats () [static]

Returns a list of image formats that are supported for image input.

See also outputFormats() [p. 141], inputFormatList() [p. 138] and QImageIO [p. 154].

void QImage::invertPixels (bool invertAlpha = TRUE)

Inverts all pixel values in the image.

If the depth is 32: if *invertAlpha* is TRUE, the alpha bits are also inverted, otherwise they are left unchanged.

If the depth is not 32, the argument *invertAlpha* has no meaning.

Note that inverting an 8-bit image means to replace all pixels using color index i with a pixel using color index 255 minus i. Similarly for a 1-bit image. The color table is not changed.

See also fill() [p. 138], depth() [p. 137] and hasAlphaBuffer() [p. 138].

bool QImage::isGrayscale () const

For 16-bit and 32-bit images, this function is equivalent to allGray().

For 8-bpp images, this function returns TRUE if color(i) is QRgb(i,i,i) for all indices of the color table.

See also allGray() [p. 134] and depth() [p. 137].

bool QImage::isNull () const

Returns TRUE if it is a null image, otherwise FALSE.

A null image has all parameters set to zero and no allocated data.

Examples: qtimage/qtimage.cpp and showimg/showimg.cpp.

uchar ** QImage::jumpTable () const

Returns a pointer to the scanline pointer table.

This is the beginning of the data block for the image.

See also bits() [p. 135] and scanLine() [p. 143].

bool QImage::load (const QString & fileName, const char * format = 0)

Loads an image from the file *fileName*. Returns TRUE if the image was successfully loaded; otherwise returns FALSE.

If *format* is specified, the loader attempts to read the image using the specified format. If *format* is not specified (which is the default), the loader reads a few bytes from the header to guess the file format.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also loadFromData() [p. 140], save() [p. 142], imageFormat() [p. 138], QPixmap::load() [p. 233] and QImageIO [p. 154].

bool QImage::loadFromData (const uchar * buf, uint len, const char * format = 0)

Loads an image from the first *len* bytes of binary data in *buf*. Returns TRUE if the image was successfully loaded; otherwise returns FALSE.

If *format* is specified, the loader attempts to read the image using the specified format. If *format* is not specified (which is the default), the loader reads a few bytes from the header to guess the file format.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also load() [p. 139], save() [p. 142], imageFormat() [p. 138], QPixmap::loadFromData() [p. 234] and QImageIO [p. 154].

bool QImage::loadFromData (QByteArray buf, const char * format = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Loads an image from the QByteArray *buf*.

QImage QImage::mirror () const

Returns a QImage which is a vertically mirrored copy of this image. The original QImage is not changed.

QImage QImage::mirror (bool horizontal, bool vertical) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the image mirrored in the horizontal and/or the vertical direction depending on whether *horizontal* and *vertical* are set to TRUE or FALSE. The original image is not changed.

See also smoothScale() [p. 144].

int QImage::numBytes () const

Returns the number of bytes occupied by the image data.

See also bytesPerLine() [p. 135] and bits() [p. 135].

int QImage::numColors () const

Returns the size of the color table for the image.

Notice that numColors() returns 0 for 16-bpp and 32-bpp images because these images do not use color tables, but instead encode pixel values as RGB triplets.

See also setNumColors() [p. 144] and colorTable() [p. 135].

Example: themes/wood.cpp.

QPoint QImage::offset () const

Returns the number of pixels by which the image is intended to be offset by when positioning relative to other images.

bool QImage::operator!= (const QImage & i) const

Returns TRUE if this image and image *i* have different contents; otherwise returns FALSE. The comparison can be slow, unless there is some obvious difference, such as different widths, in which case the function will return quickly.

See also operator=() [p. 141].

QImage & QImage::operator= (const QImage & image)

Assigns a shallow copy of *image* to this image and returns a reference to this image.

See also copy() [p. 136].

QImage & QImage::operator= (const QPixmap & pixmap)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the image bits to the *pixmap* contents and returns a reference to the image.

If the image shares data with other images, it will first dereference the shared data.

Makes a call to QPixmap::convertToImage().

bool QImage::operator== (const QImage & i) const

Returns TRUE if this image and image *i* have the same contents; otherwise returns FALSE. The comparison can be slow, unless there is some obvious difference, such as different widths, in which case the function will return quickly.

See also operator=() [p. 141].

QStringList QImage::outputFormatList () [static]

Returns a list of image formats that are supported for image output.

See also inputFormatList() [p. 138], outputFormats() [p. 141] and QImageIO [p. 154].

QStrList QImage::outputFormats () [static]

Returns a list of image formats that are supported for image output.

See also inputFormats() [p. 139], outputFormatList() [p. 141] and QImageIO [p. 154].

Example: showimg/showimg.cpp.

QRgb QImage::pixel (int x, int y) const

Returns the color of the pixel at the coordinates (x, y).

If (x, y) is not on the image, the results are undefined.

See also setPixel() [p. 144], qRed() [p. 83], qGreen() [p. 83], qBlue() [p. 83] and valid() [p. 146].

Example: qmag/qmag.cpp.

int QImage::pixelIndex (int x, int y) const

Returns the pixel index at the given coordinates.

If (*x*, *y*) is not valid, or if the image is not a paletted image (depth() > 8), the results are undefined.

See also valid() [p. 146] and depth() [p. 137].

QRect QImage::rect () const

Returns the enclosing rectangle (0,0,width(),height()) of the image.

See also width() [p. 146], height() [p. 138] and size() [p. 144].

void QImage::reset ()

Resets all image parameters and deallocates the image data.

Example: qtimage/qtimage.cpp.

bool QImage::save (const QString & fileName, const char * format, int quality = -1) const

Saves the image to the file *fileName*, using the image file format *format* and a quality factor of *quality*. *quality* must be in the range 0..100 or -1. Specify 0 to obtain small compressed files, 100 for large uncompressed files, and -1 (the default) to use the default settings.

Returns TRUE if the image was successfully saved; otherwise returns FALSE.

See also load() [p. 139], loadFromData() [p. 140], imageFormat() [p. 138], QPixmap::save() [p. 236] and QImageIO [p. 154].

QImage QImage::scale (int w, int h, ScaleMode mode = ScaleFree) const

Returns a scaled copy of the image. The returned image has a size of width *w* by height *h* pixels if *mode* is ScaleFree. The modes ScaleMin and ScaleMax may be used to preserve the ratio of the image: if *mode* is ScaleMin, the returned image is guaranteed to fit into the rectangle specified by *w* and *h* (it is as large as possible within the constraints); if *mode* is ScaleMax, the returned image fits at least into the specified rectangle (it is a small as possible within the constraints).

If either the width *w* or the height *h* is 0 or negative, this function returns a null image.

This function uses a rather simple algorithm; if you need a better quality, use smoothScale() instead.

See also scaleWidth() [p. 143], scaleHeight() [p. 142], smoothScale() [p. 144] and xForm() [p. 146].

QImage QImage::scale (const QSize & s, ScaleMode mode = ScaleFree) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The requested size of the image is *s*.

QImage QImage::scaleHeight (int h) const

Returns a scaled copy of the image. The returned image has a height of *h* pixels. This function automatically calculates the width of the image so that the ratio of the image is preserved.

If *h* is 0 or negative a null image is returned.

See also scale() [p. 142], scaleWidth() [p. 143], smoothScale() [p. 144] and xForm() [p. 146].

QImage QImage::scaleWidth (int w) const

Returns a scaled copy of the image. The returned image has a width of *w* pixels. This function automatically calculates the height of the image so that the ratio of the image is preserved.

If *w* is 0 or negative a null image is returned.

See also scale() [p. 142], scaleHeight() [p. 142], smoothScale() [p. 144] and xForm() [p. 146].

uchar * QImage::scanLine (int i) const

Returns a pointer to the pixel data at the scanline with index *i*. The first scanline is at index 0.

The scanline data is aligned on a 32-bit boundary.

Warning: If you are accessing 32-bpp image data, cast the returned pointer to `QRgb*` (QRgb has a 32-bit size) and use it to read/write the pixel value. You cannot use the `uchar*` pointer directly, because the pixel format depends on the byte order on the underlying platform. Hint: use qRed() and friends (qcolor.h) to access the pixels.

Warning: If you are accessing 16-bpp image data, you have to handle endianness yourself for now.

See also bytesPerLine() [p. 135], bits() [p. 135] and jumpTable() [p. 139].

Example: desktop/desktop.cpp.

void QImage::setAlphaBuffer (bool enable)

Enables alpha buffer mode if *enable* is TRUE, otherwise disables it. The default setting is disabled.

An 8-bpp image has 8-bit pixels. A pixel is an index into the color table, which contains 32-bit color values. In a 32-bpp image, the 32-bit pixels are the color values.

This 32-bit value is encoded as follows: The lower 24 bits are used for the red, green, and blue components. The upper 8 bits contain the alpha component.

The alpha component specifies the transparency of a pixel. 0 means completely transparent and 255 means opaque. The alpha component is ignored if you do not enable alpha buffer mode.

The alpha buffer is used to set a mask when a QImage is translated to a QPixmap.

See also hasAlphaBuffer() [p. 138] and createAlphaMask() [p. 137].

void QImage::setColor (int i, QRgb c)

Sets a color in the color table at index *i* to *c*.

A color value is an RGB triplet. Use the qRgb function (defined in qcolor.h) to make RGB triplets.

See also color() [p. 135], setNumColors() [p. 144] and numColors() [p. 140].

Examples: desktop/desktop.cpp and themes/wood.cpp.

void QImage::setDotsPerMeterX (int x)

Sets the value returned by dotsPerMeterX() to *x*.

void QImage::setDotsPerMeterY (int y)

Sets the value returned by dotsPerMeterY() to *y*.

void QImage::setNumColors (int numColors)

Resizes the color table to *numColors* colors.

If the color table is expanded, then all new colors will be set to black (RGB 0,0,0).

See also numColors() [p. 140], color() [p. 135], setColor() [p. 143] and colorTable() [p. 135].

void QImage::setOffset (const QPoint & p)

Sets the value returned by offset() to *p*.

void QImage::setPixel (int x, int y, uint index_or_rgb)

Sets the pixel index or color at the coordinates (*x, y*) to *index_or_rgb*.

If (*x, y*) is not valid, the result is undefined.

If the image is a paletted image (depth() <= 8) and *index_or_rgb* >= numColors(), the result is undefined.

See also pixelIndex() [p. 142], pixel() [p. 141], qRgb() [p. 83], qRgba() [p. 83] and valid() [p. 146].

void QImage::setText (const char * key, const char * lang, const QString & s)

Records string *s* for the keyword *key*. The *key* should be a portable keyword recognizable by other software - some suggested values can be found in the PNG specification. *s* can be any text. *lang* should specify the language code (see RFC 1766) or 0.

QSize QImage::size () const

Returns the size of the image, i.e. its width and height.

See also width() [p. 146], height() [p. 138] and rect() [p. 142].

QImage QImage::smoothScale (int w, int h, ScaleMode mode = ScaleFree) const

Returns a smoothly scaled copy of the image. The returned image has a size of width *w* by height *h* pixels if *mode* is ScaleFree. The modes ScaleMin and ScaleMax may be used to preserve the ratio of the image: if *mode* is ScaleMin, the returned image is guaranteed to fit into the rectangle specified by *w* and *h* (it is as large as possible within the constraints); if *mode* is ScaleMax, the returned image fits at least into the specified rectangle (it is a small as possible within the constraints).

For 32-bpp images and 1-bpp/8-bpp color images the result will be 32-bpp, whereas all-gray images (including black-and-white 1-bpp) will produce 8-bit grayscale images with the palette spanning 256 grays from black to white.

This function uses code based on pnmscale.c by Jef Poskanzer.

pnmscale.c - read a portable anymap and scale it

Copyright (C) 1989, 1991 by Jef Poskanzer.

See also scale() [p. 142] and mirror() [p. 140].

QImage QImage::smoothScale (const QSize & s, ScaleMode mode = ScaleFree) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The requested size of the image is *s*.

QImage QImage::swapRGB () const

Returns a QImage in which the values of the red and blue components of all pixels have been swapped, effectively converting an RGB image to a BGR image. The original QImage is not changed.

Endian QImage::systemBitOrder () [static]

Determines the bit order of the display hardware. Returns QImage::LittleEndian (LSB first) or QImage::BigEndian (MSB first).

See also systemByteOrder() [p. 145].

Endian QImage::systemByteOrder () [static]

Determines the host computer byte order. Returns QImage::LittleEndian (LSB first) or QImage::BigEndian (MSB first).

See also systemBitOrder() [p. 145].

QString QImage::text (const char * key, const char * lang = 0) const

Returns the string recorded for the keyword *key* in language *lang,* or in a default language if *lang* is 0.

QString QImage::text (const QImageTextKeyLang & kl) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the string recorded for the keyword and language *kl.*

QStringList QImage::textKeys () const

Returns the keywords for which some texts are recorded.

See also textList() [p. 146], text() [p. 145], setText() [p. 144] and textLanguages() [p. 145].

QStringList QImage::textLanguages () const

Returns the language identifiers for which some texts are recorded.

See also textList() [p. 146], text() [p. 145], setText() [p. 144] and textKeys() [p. 145].

QValueList<QImageTextKeyLang> QImage::textList () const

Returns a list of QImageTextKeyLang objects that enumerate all the texts key/language pairs set by setText() for this image.

bool QImage::valid (int x, int y) const

Returns TRUE if (*x*, *y*) is a valid coordinate in the image, otherwise it returns FALSE.

See also width() [p. 146], height() [p. 138] and pixelIndex() [p. 142].

Example: qmag/qmag.cpp.

int QImage::width () const

Returns the width of the image.

See also height() [p. 138], size() [p. 144] and rect() [p. 142].

Example: opengl/texture/gltexobj.cpp.

QImage QImage::xForm (const QWMatrix & matrix) const

Returns a copy of the image that is transformed using the transformation matrix, *matrix*.

The transformation *matrix* is internally adjusted to compensate for unwanted translation, i.e. xForm() returns the smallest image that contains all the transformed points of the original image.

See also scale() [p. 142], QPixmap::xForm() [p. 237], QPixmap::trueMatrix() [p. 237] and QWMatrix [p. 291].

Related Functions

QDataStream & operator<< (QDataStream & s, const QImage & image)

Writes the image *image* to the stream *s* as a PNG image.

See also QImage::save() [p. 142] and Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QImage & image)

Reads an image from the stream *s* and stores it in *image*.

See also QImage::load() [p. 139] and Format of the QDataStream operators [Input/Output and Networking with Qt].

QImageConsumer Class Reference

The QImageConsumer class is an abstraction used by QImageDecoder.

```
#include <qasyncimageio.h>
```

Public Members

- virtual void **end** ()
- virtual void **changed** (const QRect &)
- virtual void **frameDone** ()
- virtual void **frameDone** (const QPoint & offset, const QRect & rect)
- virtual void **setLooping** (int n)
- virtual void **setFramePeriod** (int milliseconds)
- virtual void **setSize** (int, int)

Detailed Description

The QImageConsumer class is an abstraction used by QImageDecoder.

The QMovie class, or QLabel::setMovie(), are easy to use and for most situations do what you want with regards animated images.

A QImageConsumer consumes information about changes to the QImage maintained by a QImageDecoder. Think of the QImage as the model or source of the image data, with the QImageConsumer as a view of that data and the QImageDecoder being the controller that orchestrates the relationship between the model and the view.

You'd use the QImageConsumer class, for example, if you were implementing a web browser with your own image loaders.

See also QImageDecoder [p. 149], Graphics Classes, Image Processing Classes and Multimedia Classes.

Member Function Documentation

void QImageConsumer::changed (const QRect &) [virtual]

Called when the given area of the image has changed.

void QImageConsumer::end () [virtual]

Called when all data of all frames has been decoded and revealed as changed().

void QImageConsumer::frameDone () [virtual]

One of the two frameDone() functions will be called when a frame of an animated image has ended and been revealed as changed().

When this function is called, the current image should be displayed.

The decoder will not make any further changes to the image until the next call to QImageFormat::decode().

void QImageConsumer::frameDone (const QPoint & offset, const QRect & rect) [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

One of the two frameDone() functions will be called when a frame of an animated image has ended and been revealed as changed().

When this function is called, the area *rect* in the current image should be moved by *offset* and displayed.

The decoder will not make any further changes to the image until the next call to QImageFormat::decode().

void QImageConsumer::setFramePeriod (int milliseconds) [virtual]

Notes that the frame about to be decoded should not be displayed until the given number of *milliseconds* after the time that this function is called. Of course, the image may not have been decoded by then, in which case the frame should not be displayed until it is complete. A value of -1 (the assumed default) indicates that the image should be displayed even while it is only partially loaded.

void QImageConsumer::setLooping (int n) [virtual]

Called to indicate that the sequence of frames in the image should be repeated *n* times, including the sequence during decoding.

- 0 = Forever
- 1 = Only display frames the first time through
- 2 = Repeat once after first pass through images
- etc.

To make the QImageDecoder do this, just delete it and pass the information to it again for decoding (setLooping() will be called again, of course, but that can be ignored), or keep copies of the changed areas at the ends of frames.

void QImageConsumer::setSize (int, int) [virtual]

This function is called as soon as the size of the image has been determined.

QImageDecoder Class Reference

The QImageDecoder class is an incremental image decoder for all supported image formats.

`#include <qasyncimageio.h>`

Public Members

- **QImageDecoder** (QImageConsumer * c)
- **~QImageDecoder** ()
- const QImage & **image** ()
- int **decode** (const uchar * buffer, int length)

Static Public Members

- const char * **formatName** (const uchar * buffer, int length)
- QImageFormatType * **format** (const char * name)
- QStrList **inputFormats** ()
- void **registerDecoderFactory** (QImageFormatType * f)
- void **unregisterDecoderFactory** (QImageFormatType * f)

Detailed Description

The QImageDecoder class is an incremental image decoder for all supported image formats.

New formats are installed by creating objects of class QImageFormatType; the QMovie class can be used for all installed incremental image formats. QImageDecoder is useful only for creating new ways of feeding data to an QImageConsumer.

A QImageDecoder is a machine that decodes images. It takes encoded image data via its decode() method and expresses its decoding by supplying information to a QImageConsumer. It implements its decoding by using a QImageFormat created by one of the currently-existing QImageFormatType factory-objects.

QImageFormatType and QImageFormat are the classes that you might need to implement support for additional image formats.

Qt supports GIF reading if it is configured that way during installation (see qgif.h). If it is, we are required to state that "The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated."

See also Graphics Classes, Image Processing Classes and Multimedia Classes.

Member Function Documentation

QImageDecoder::QImageDecoder (QImageConsumer * c)

Constructs a QImageDecoder that will send change information to the QImageConsumer *c*.

QImageDecoder::~QImageDecoder ()

Destroys a QImageDecoder. The image it built is destroyed. The decoder built by the factory for the file format is destroyed. The consumer for which it decoded the image is *not* destroyed.

int QImageDecoder::decode (const uchar * buffer, int length)

Call this function to decode some data into image changes. The data in *buffer* will be decoded, sending change information to the QImageConsumer of this QImageDecoder until one of the change functions of the consumer returns FALSE. The length of the data is given in *length*.

Returns the number of bytes consumed: 0 if consumption is complete, and -1 if decoding fails due to invalid data.

QImageFormatType * QImageDecoder::format (const char * name) [static]

Returns a QImageFormatType by name. This might be used when the user needs to force data to be interpreted as being in a certain format. *name* is one of the formats listed by QImageDecoder::inputFormats(). Note that you will still need to supply decodable data to result->decoderFor() before you can begin decoding the data.

const char * QImageDecoder::formatName (const uchar * buffer, int length) [static]

Call this function to find the name of the format of the given header. The returned string is statically allocated. The function will look at the first *length* characters in the *buffer*.

Returns 0 if the format is not recognized.

const QImage & QImageDecoder::image ()

Returns the image currently being decoded.

QStrList QImageDecoder::inputFormats () [static]

Returns a sorted list of formats for which asynchronous loading is supported.

void QImageDecoder::registerDecoderFactory (QImageFormatType * f) [static]

Registers the new QImageFormatType *f*. This is not needed in application code because factories call this themselves.

void QImageDecoder::unregisterDecoderFactory (QImageFormatType * f) [static]

Unregisters the QImageFormatType *f*. This is not needed in application code because factories call this themselves.

QImageFormat Class Reference

The QImageFormat class is an incremental image decoder for a specific image format.

```
#include <qasyncimageio.h>
```

Public Members

- virtual int **decode** (QImage & img, QImageConsumer * consumer, const uchar * buffer, int length)

Detailed Description

The QImageFormat class is an incremental image decoder for a specific image format.

By making a derived class of QImageFormatType, which in turn creates objects that are a subclass of QImageFormat, you can add support for more incremental image formats, allowing such formats to be sources for a QMovie or for the first frame of the image stream to be loaded as a QImage or QPixmap.

Your new subclass must reimplement the decode() function in order to process your new format.

New QImageFormat objects are generated by new QImageFormatType factories.

See also Graphics Classes, Image Processing Classes and Multimedia Classes.

Member Function Documentation

int QImageFormat::decode (QImage & img, QImageConsumer * consumer, const uchar * buffer, int length) [virtual]

New subclasses must reimplement this method.

It should decode some or all of the bytes from *buffer* into *img*, calling the methods of *consumer* as the decoding proceeds to inform that consumer of changes to the image. The length of the data is given in *length*. The consumer may be 0, in which case the function should just process the data into *img* without telling any consumer about the changes. Note that the decoder must store enough state to be able to continue in subsequent calls to this method - this is the essence of the incremental image loading.

The function should return without processing all the data if it reaches the end of a frame in the input.

The function must return the number of bytes it has processed.

QImageFormatType Class Reference

The QImageFormatType class is a factory that makes QImageFormat objects.

`#include <qasyncimageio.h>`

Public Members

- virtual **~QImageFormatType** ()
- virtual QImageFormat * **decoderFor** (const uchar * buffer, int length)
- virtual const char * **formatName** () const

Protected Members

- **QImageFormatType** ()

Detailed Description

The QImageFormatType class is a factory that makes QImageFormat objects.

Whereas the QImageIO class allows for *complete* loading of images, QImageFormatType allows for *incremental* loading of images.

New image file formats are installed by creating objects of derived classes of QImageFormatType. They must implement decoderFor() and formatName().

QImageFormatType is a very simple class. Its only task is to recognize image data in some format and make a new object, subclassed from QImageFormat, which can decode that format.

The factories for formats built into Qt are automatically defined before any other factory is initialized. If two factories would recognize an image format, the factory created last will override the earlier one; you can thus override current and future built-in formats.

See also Graphics Classes, Image Processing Classes and Multimedia Classes.

Member Function Documentation

QImageFormatType::QImageFormatType () [protected]

Constructs a factory. It automatically registers itself with QImageDecoder.

QImageFormatType::~QImageFormatType () [virtual]

Destroys a factory. It automatically unregisters itself from QImageDecoder.

QImageFormat * QImageFormatType::decoderFor (const uchar * buffer, int length) [virtual]

Returns a decoder for decoding an image that starts with the bytes in *buffer*. The length of the data is given in *length*. This function should only return a decoder if it is certain that the decoder applies to data with the given header. Returns 0 if there is insufficient data in the header to make a positive identification or if the data is not recognized.

const char * QImageFormatType::formatName () const [virtual]

Returns the name of the format supported by decoders from this factory. The string is statically allocated.

QImageIO Class Reference

The QImageIO class contains parameters for loading and saving images.

`#include <qimage.h>`

Public Members

- **QImageIO** ()
- **QImageIO** (QIODevice * ioDevice, const char * format)
- **QImageIO** (const QString & fileName, const char * format)
- **~QImageIO** ()
- const QImage & **image** () const
- int **status** () const
- const char * **format** () const
- QIODevice * **ioDevice** () const
- QString **fileName** () const
- int **quality** () const
- QString **description** () const
- const char * **parameters** () const
- float **gamma** () const
- void **setImage** (const QImage & image)
- void **setStatus** (int status)
- void **setFormat** (const char * format)
- void **setIODevice** (QIODevice * ioDevice)
- void **setFileName** (const QString & fileName)
- void **setQuality** (int q)
- void **setDescription** (const QString & description)
- void **setParameters** (const char * parameters)
- void **setGamma** (float gamma)
- bool **read** ()
- bool **write** ()

Static Public Members

- const char * **imageFormat** (const QString & fileName)
- const char * **imageFormat** (QIODevice * d)
- QStrList **inputFormats** ()
- QStrList **outputFormats** ()
- void **defineIOHandler** (const char * format, const char * header, const char * flags, image_io_handler readImage, image_io_handler writeImage)

Detailed Description

The QImageIO class contains parameters for loading and saving images.

QImageIO contains a QIODevice object that is used for image data I/O. The programmer can install new image file formats in addition to those that Qt implements.

Qt currently supports the following image file formats: PNG, BMP, XBM, XPM and PNM. It may also support JPEG, MNG and GIF, if specially configured during compilation. The different PNM formats are: PBM (P1 or P4), PGM (P2 or P5), and PPM (P3 or P6).

You don't normally need to use this class; QPixmap::load(), QPixmap::save(), and QImage contain sufficient functionality.

For image files that contain sequences of images, only the first is read. See the QMovie for loading multiple images.

PBM, PGM, and PPM format *output* is always in the more condensed raw format. PPM and PGM files with more than 256 levels of intensity are scaled down when reading.

Warning: If you are in a country which recognizes software patents and in which Unisys holds a patent on LZW compression and/or decompression and you want to use GIF, Unisys may require you to license the technology. Such countries include Canada, Japan, the USA, France, Germany, Italy and the UK.

GIF support may be removed completely in a future version of Qt. We recommend using the PNG format.

See also QImage [p. 129], QPixmap [p. 225], QFile [Input/Output and Networking with Qt], QMovie [p. 161], Graphics Classes, Image Processing Classes and Input/Output and Networking.

Member Function Documentation

QImageIO::QImageIO ()

Constructs a QImageIO object with all parameters set to zero.

QImageIO::QImageIO (QIODevice * ioDevice, const char * format)

Constructs a QImageIO object with the I/O device *ioDevice* and a *format* tag.

QImageIO::QImageIO (const QString & fileName, const char * format)

Constructs a QImageIO object with the file name *fileName* and a *format* tag.

QImageIO::~QImageIO ()

Destroys the object and all related data.

void QImageIO::defineIOHandler (const char * format, const char * header, const char * flags, image_io_handler readImage, image_io_handler writeImage) [static]

Defines an image I/O handler for the image format called *format*, which is recognized using the regular expression *header*, read using *readImage* and written using *writeImage*.

flags is a string of single-character flags for this format. The only flag defined currently is T (upper case), so the only legal value for *flags* are "T" and the empty string. The "T" flag means that the image file is a text file, and Qt should treat all newline conventions as equivalent. (XPM files and some PPM files are text files for example.)

format is used to select a handler to write a QImage; *header* is used to select a handler to read an image file.

If *readImage* is a null pointer, the QImageIO will not be able to read images in *format*. If *writeImage* is a null pointer, the QImageIO will not be able to write images in *format*. If both are null, the QImageIO object is valid but useless.

Example:

```
void readGIF( QImageIO *image )
{
  // read the image using the image->ioDevice()
}

void writeGIF( QImageIO *image )
{
  // write the image using the image->ioDevice()
}

// add the GIF image handler

QImageIO::defineIOHandler( "GIF",
                           "^GIF[0-9][0-9][a-z]",
                           0,
                           readGIF,
                           writeGIF );
```

Before the regexp test, all the 0 bytes in the file header are converted to 1 bytes. This is done because when Qt was ASCII-based, QRegExp could not handle 0 bytes in strings.

(Note that if one handlerIO supports writing a format and another supports reading it, Qt supports both reading and writing. If two handlers support the same operation, Qt chooses one arbitrarily.)

QString QImageIO::description () const

Returns the image description string.

See also setDescription() [p. 158].

QString QImageIO::fileName () const

Returns the file name currently set.

See also setFileName() [p. 158].

const char * QImageIO::format () const

Returns the image format string or 0 if no format has been explicitly set.

float QImageIO::gamma () const

Returns the gamma value at which the image will be viewed.

See also setGamma() [p. 158].

const QImage & QImageIO::image () const

Returns the image currently set.

See also setImage() [p. 159].

const char * QImageIO::imageFormat (const QString & fileName) [static]

Returns a string that specifies the image format of the file *fileName*, or null if the file cannot be read or if the format is not recognized.

const char * QImageIO::imageFormat (QIODevice * d) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string that specifies the image format of the image read from IO device *d*, or a null pointer if the device cannot be read or if the format is not recognized.

Make sure that *d* is at the right position in the device (for example, at the beginning of the file).

See also QIODevice::at() [Input/Output and Networking with Qt].

QStrList QImageIO::inputFormats () [static]

Returns a sorted list of image formats that are supported for image input.

QIODevice * QImageIO::ioDevice () const

Returns the IO device currently set.

See also setIODevice() [p. 159].

QStrList QImageIO::outputFormats () [static]

Returns a sorted list of image formats that are supported for image output.

Example: scribble/scribble.cpp.

const char * QImageIO::parameters () const

Returns the image's parameters string.

See also setParameters() [p. 159].

int QImageIO::quality () const

Returns the quality of the written image, related to the compression ratio.

See also setQuality() [p. 159] and QImage::save() [p. 142].

bool QImageIO::read ()

Reads an image into memory and returns TRUE if the image was successfully read; otherwise returns FALSE.

Before reading an image you must set an IO device or a file name. If both an IO device and a file name have been set, the IO device will be used.

Setting the image file format string is optional.

Note that this function does *not* set the format used to read the image. If you need that information, use the imageFormat() static functions.

Example:

```
QImageIO iio;
QPixmap  pixmap;
iio.setFileName( "vegeburger.bmp" );
if ( image.read() )        // ok
    pixmap = iio.image();  // convert to pixmap
```

See also setIODevice() [p. 159], setFileName() [p. 158], setFormat() [p. 158], write() [p. 159] and QPixmap::load() [p. 233].

void QImageIO::setDescription (const QString & description)

Sets the image description string for image handlers that support image descriptions to *description*.

Currently, no image format supported by Qt uses the description string.

void QImageIO::setFileName (const QString & fileName)

Sets the name of the file to read or write an image from to *fileName*.

See also setIODevice() [p. 159].

void QImageIO::setFormat (const char * format)

Sets the image format *format* for the image to be read or written.

It is necessary to specify a format before writing an image.

It is not necessary to specify a format before reading an image. If no format has been set, Qt guesses the image format before reading it. If a format is set the image will only be read if it has that format.

See also read() [p. 158], write() [p. 159] and format() [p. 156].

void QImageIO::setGamma (float gamma)

Sets the gamma value at which the image will be viewed to *gamma*. If the image format stores a gamma value for which the image is intended to be used, then this setting will be used to modify the image. Setting to 0.0 will disable gamma correction (ie. any specification in the file will be ignored).

The default value is 0.0.

See also gamma() [p. 156].

void QImageIO::setIODevice (QIODevice * ioDevice)

Sets the IO device to be used for reading or writing an image.

Setting the IO device allows images to be read/written to any block-oriented QIODevice.

If *ioDevice* is not null, this IO device will override file name settings.

See also setFileName() [p. 158].

void QImageIO::setImage (const QImage & image)

Sets the image to *image*.

See also image() [p. 157].

void QImageIO::setParameters (const char * parameters)

Sets the image's parameter string to *parameters*. This is for image handlers that require special parameters.

Although the current image formats supported by Qt ignore the parameters string, it may be used in future extensions or contributions (for example, JPEG).

See also parameters() [p. 157].

void QImageIO::setQuality (int q)

Sets the quality of the written image to *q*, related to the compression ratio.

q must be in the range 0..100. Specify 0 to obtain small compressed files, 100 for large uncompressed files

See also quality() [p. 157] and QImage::save() [p. 142].

void QImageIO::setStatus (int status)

Sets the image IO status to *status*. A non-zero value indicates an error, whereas 0 means that the IO operation was successful.

See also status() [p. 159].

int QImageIO::status () const

Returns the image's IO status. A non-zero value indicates an error, whereas 0 means that the IO operation was successful.

See also setStatus() [p. 159].

bool QImageIO::write ()

Writes an image to an IO device and returns TRUE if the image was successfully written; otherwise returns FALSE.

Before writing an image you must set an IO device or a file name. If both an IO device and a file name have been set, the IO device will be used.

The image will be written using the specified image format.

Example:

```
QImageIO iio;
QImage   im;
im = pixmap; // convert to image
iio.setImage( im );
iio.setFileName( "vegeburger.bmp" );
iio.setFormat( "BMP" );
if ( iio.write() )
    // returned TRUE if written successfully
```

See also setIODevice() [p. 159], setFileName() [p. 158], setFormat() [p. 158], read() [p. 158] and QPixmap::save() [p. 236].

QMovie Class Reference

The QMovie class provides incremental loading of animations or images, signalling as it progresses.

```
#include <qmovie.h>
```

Public Members

- **QMovie** ()
- **QMovie** (int bufsize)
- **QMovie** (QDataSource * src, int bufsize = 1024)
- **QMovie** (const QString & fileName, int bufsize = 1024)
- **QMovie** (QByteArray data, int bufsize = 1024)
- **QMovie** (const QMovie & movie)
- **~QMovie** ()
- QMovie & **operator=** (const QMovie & movie)
- int **pushSpace** () const
- void **pushData** (const uchar * data, int length)
- const QColor & **backgroundColor** () const
- void **setBackgroundColor** (const QColor & c)
- const QRect & **getValidRect** () const
- const QPixmap & **framePixmap** () const
- const QImage & **frameImage** () const
- bool **isNull** () const
- int **frameNumber** () const
- int **steps** () const
- bool **paused** () const
- bool **finished** () const
- bool **running** () const
- void **unpause** ()
- void **pause** ()
- void **step** ()
- void **step** (int steps)
- void **restart** ()
- int **speed** () const
- void **setSpeed** (int percent)
- void **connectResize** (QObject * receiver, const char * member)
- void **disconnectResize** (QObject * receiver, const char * member = 0)
- void **connectUpdate** (QObject * receiver, const char * member)
- void **disconnectUpdate** (QObject * receiver, const char * member = 0)

- ▪ enum **Status** { SourceEmpty = -2, UnrecognizedFormat = -1, Paused = 1, EndOfFrame = 2, EndOfLoop = 3, EndOfMovie = 4, SpeedChanged = 5 }
- ▪ void **connectStatus** (QObject * receiver, const char * member)
- ● void **disconnectStatus** (QObject * receiver, const char * member = 0)

Detailed Description

The QMovie class provides incremental loading of animations or images, signalling as it progresses.

A QMovie provides a QPixmap as the framePixmap(); connections can be made via connectResize() and connectUpdate() to receive notification of size and pixmap changes. All decoding is driven by the normal event-processing mechanisms. The simplest way to display a QMovie is to use a QLabel and QLabel::setMovie().

The movie begins playing as soon as the QMovie is created (actually, once control returns to the event loop). When the last frame in the movie has been played, it may loop back to the start if such looping is defined in the input source.

QMovie objects are explicitly shared. This means that a QMovie copied from another QMovie will be displaying the same frame at all times. If one shared movie pauses, all pause. To make *independent* movies, they must be constructed separately.

The set of data formats supported by QMovie is determined by the decoder factories that have been installed; the format of the input is determined as the input is decoded.

The supported formats are MNG (if Qt is built with MNG support enabled) and GIF (if Qt is built with GIF support enabled). For MNG support, you need to have installed libmng from http://www.libmng.com.

Archives of animated GIFs and tools for building them can be found, for example, at Yahoo.

We are required to state the following: The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

Warning: If you are in a country that recognizes software patents and in which Unisys holds a patent on LZW compression and/or decompression and you want to use GIF, Unisys may require you to license that technology. Such countries include Canada, Japan, the USA, France, Germany, Italy and the UK.

GIF support may be removed completely in a future version of Qt. We recommend using the MNG or PNG format.

See also QLabel::setMovie() [Widgets with Qt], Graphics Classes, Image Processing Classes and Multimedia Classes.

Member Type Documentation

QMovie::Status

- ● `QMovie::SourceEmpty`
- ● `QMovie::UnrecognizedFormat`
- ● `QMovie::Paused`
- ● `QMovie::EndOfFrame`
- ● `QMovie::EndOfLoop`
- ● `QMovie::EndOfMovie`
- ● `QMovie::SpeedChanged`

Member Function Documentation

QMovie::QMovie ()

Constructs a null QMovie. The only interesting thing to do to such a movie is to assign another movie to it.

See also isNull() [p. 165].

QMovie::QMovie (int bufsize)

Constructs a QMovie with an external data source. You should later call pushData() to send incoming animation data to the movie.

The *bufsize* argument sets the maximum amount of data the movie will transfer from the data source per event loop. The lower this value, the better interleaved the movie playback will be with other event processing, but the slower the overall processing will be.

See also pushData() [p. 166].

QMovie::QMovie (QDataSource * src, int bufsize = 1024)

Constructs a QMovie that reads an image sequence from the given data source, *src*. The source must be allocated dynamically, because QMovie will take ownership of it and will destroy it when the movie is destroyed. The movie starts playing as soon as event processing continues.

The *bufsize* argument sets the maximum amount of data the movie will transfer from the data source per event loop. The lower this value, the better interleaved the movie playback will be with other event processing, but the slower the overall processing will be.

QMovie::QMovie (const QString & fileName, int bufsize = 1024)

Constructs a QMovie that reads an image sequence from the file, *fileName*.

The *bufsize* argument sets the maximum amount of data the movie will transfer from the data source per event loop. The lower this value, the better interleaved the movie playback will be with other event processing, but the slower the overall processing will be.

QMovie::QMovie (QByteArray data, int bufsize = 1024)

Constructs a QMovie that reads an image sequence from the byte array, *data*.

The *bufsize* argument sets the maximum amount of data the movie will transfer from the data source per event loop. The lower this value, the better interleaved the movie playback will be with other event processing, but the slower the overall processing will be.

QMovie::QMovie (const QMovie & movie)

Constructs a movie that uses the same data as movie *movie*. QMovies use explicit sharing, so operations on the copy will affect both.

QMovie::~QMovie ()

Destroys the QMovie. If this is the last reference to the data of the movie, the data are deallocated.

const QColor & QMovie::backgroundColor () const

Returns the background color of the movie set by setBackgroundColor().

void QMovie::connectResize (QObject * receiver, const char * member)

Connects the *receiver*'s *member* of type `void member(const QSize&)` so that it is signalled when the movie changes size.

Note that due to the explicit sharing of QMovie objects, these connections persist until they are explicitly disconnected with disconnectResize() or until *every* shared copy of the movie is deleted.

Example: movies/main.cpp.

void QMovie::connectStatus (QObject * receiver, const char * member)

Connects the *receiver*'s *member*, of type `void member(int)` so that it is signalled when the movie changes status. The status codes are negative for errors and positive for information, and they are currently:

- QMovie::SourceEmpty signalled if the input cannot be read.
- QMovie::UnrecognizedFormat signalled if the input data is unrecognized.
- QMovie::Paused signalled when the movie is paused by a call to paused() or by after stepping pauses.
- QMovie::EndOfFrame signalled at end-of-frame after any update and Paused signals.
- QMovie::EndOfLoop signalled at end-of-loop, after any update signals, EndOfFrame - but before EndOfMovie.
- QMovie::EndOfMovie signalled when the movie completes and is not about to loop.

More status messages may be added in the future, so a general test for errors would test for negative.

Note that due to the explicit sharing of QMovie objects, these connections persist until they are explicitly disconnected with disconnectStatus() or until *every* shared copy of the movie is deleted.

Example: movies/main.cpp.

void QMovie::connectUpdate (QObject * receiver, const char * member)

Connects the *receiver*'s *member* of type `void member(const QRect&)` so that it is signalled when an area of the framePixmap() has changed since the previous frame.

Note that due to the explicit sharing of QMovie objects, these connections persist until they are explicitly disconnected with disconnectUpdate() or until *every* shared copy of the movie is deleted.

Example: movies/main.cpp.

void QMovie::disconnectResize (QObject * receiver, const char * member = 0)

Disconnects the *receiver*'s *member* (or all members if *member* is zero) that were previously connected by connectResize().

void QMovie::disconnectStatus (QObject * receiver, const char * member = 0)

Disconnects the *receiver*'s *member* (or all members if *member* is zero) that were previously connected by connectStatus().

void QMovie::disconnectUpdate (QObject * receiver, const char * member = 0)

Disconnects the *receiver*'s *member* (or all members if \q member is zero) that were previously connected by connectUpdate().

bool QMovie::finished () const

Returns TRUE if the image is no longer playing - this happens when all loops of all frames are complete; otherwise returns FALSE.

Example: movies/main.cpp.

const QImage & QMovie::frameImage () const

Returns the current frame of the movie, as a QImage. It is not generally useful to keep a copy of this image. Also note that you must not call this function if the movie is finished(), as the image is not them available.

See also framePixmap() [p. 165].

int QMovie::frameNumber () const

Returns the number of times EndOfFrame has been emitted since the start of the current loop of the movie. Thus, before any EndOfFrame has been emitted the value will be 0; within slots processing the first signal, frameNumber() will be 1, and so on.

const QPixmap & QMovie::framePixmap () const

Returns the current frame of the movie, as a QPixmap. It is not generally useful to keep a copy of this pixmap. It is better to keep a copy of the QMovie and get the framePixmap() only when needed for drawing.

See also frameImage() [p. 165].

Example: movies/main.cpp.

const QRect & QMovie::getValidRect () const

Returns the area of the pixmap for which pixels have been generated.

bool QMovie::isNull () const

Returns TRUE if the movie is null; otherwise returns FALSE.

QMovie & QMovie::operator= (const QMovie & movie)

Makes this movie use the same data as movie *movie*. QMovies use explicit sharing.

void QMovie::pause ()

Pauses the progress of the animation.

See also unpause() [p. 167].

Example: movies/main.cpp.

bool QMovie::paused () const

Returns TRUE if the image is paused; otherwise returns FALSE.

Example: movies/main.cpp.

void QMovie::pushData (const uchar * data, int length)

Pushes *length* bytes from *data* into the movie. *length* must be no more than the amount returned by pushSpace() since the previous call to pushData().

int QMovie::pushSpace () const

Returns the maximum amount of data that can currently be pushed into the movie by a call to pushData(). This is affected by the initial buffer size, but varies as the movie plays and data is consumed.

void QMovie::restart ()

Rewinds the movie to the beginning. If the movie has not been paused, it begins playing again.

Example: movies/main.cpp.

bool QMovie::running () const

Returns TRUE if the image is not single-stepping, not paused, and not finished; otherwise returns FALSE.

void QMovie::setBackgroundColor (const QColor & c)

Sets the background color of the pixmap to *c*. If the background color isValid(), the pixmap will never have a mask because the background color will be used in transparent regions of the image.

See also backgroundColor() [p. 164].

Example: movies/main.cpp.

void QMovie::setSpeed (int percent)

Sets the speed-up factor of the movie to *percent*. This is a percentage of the speed dictated by the input data format. The default is 100 percent.

int QMovie::speed () const

Returns the speed-up factor of the movie. The default is 100 percent.

See also setSpeed() [p. 166].

void QMovie::step ()

Steps forward 1 frame and then pauses.

Example: movies/main.cpp.

void QMovie::step (int steps)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Steps forward, showing *steps* frames, and then pauses.

int QMovie::steps () const

Returns the number of steps remaining after a call to step(). If the movie is paised, steps() returns 0. If it's running normally or is finished, steps() returns a negative number.

Example: movies/main.cpp.

void QMovie::unpause ()

Unpauses the progress of the animation.

See also pause() [p. 165].

Example: movies/main.cpp.

QPNGImagePacker Class Reference

The QPNGImagePacker class creates well-compressed PNG animations.

```
#include <qpngio.h>
```

Public Members

- **QPNGImagePacker** (QIODevice * iod, int storage_depth, int conversionflags)
- void **setPixelAlignment** (int x)
- bool **packImage** (const QImage & img)

Detailed Description

The QPNGImagePacker class creates well-compressed PNG animations.

By using transparency, QPNGImagePacker allows you to build a PNG image from a sequence of QImages.

Images are added using packImage().

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QPNGImagePacker::QPNGImagePacker (QIODevice * iod, int storage_depth, int conversionflags)

Creates an image packer that writes PNG data to IO device *iod* using a *storage_depth* bit encoding (use 8 or 32, depending on the desired quality and compression requirements).

If the image needs to be modified to fit in a lower-resolution result (eg. converting from 32-bit to 8-bit), use the *conversionflags* to specify how you'd prefer this to happen.

See also Qt::ImageConversionFlags [Additional Functionality with Qt].

bool QPNGImagePacker::packImage (const QImage & img)

Adds the image *img* to the PNG animation, analyzing the differences between this and the previous image to improve compression.

void QPNGImagePacker::setPixelAlignment (int x)

Aligns pixel differences to *x* pixels. For example, using 8 can improve playback on certain hardware. Normally the default of 1-pixel alignment (i.e. no alignment) gives better compression and performance.

QPaintDevice Class Reference

The QPaintDevice class is the base class of objects that can be painted.

`#include <qpaintdevice.h>`

Inherited by QPixmap [p. 225], QWidget [Widgets with Qt], QPicture [p. 220] and QPrinter [p. 254].

Public Members

- virtual **~QPaintDevice** ()
- int **devType** () const
- bool **isExtDev** () const
- bool **paintingActive** () const
- virtual HDC **handle** () const
- Display * **x11Display** () const
- int **x11Screen** () const
- int **x11Depth** () const
- int **x11Cells** () const
- Qt::HANDLE **x11Colormap** () const
- bool **x11DefaultColormap** () const
- void * **x11Visual** () const
- bool **x11DefaultVisual** () const

Static Public Members

- Display * **x11AppDisplay** ()
- int **x11AppScreen** ()
- int **x11AppDepth** ()
- int **x11AppCells** ()
- int **x11AppDpiX** ()
- int **x11AppDpiY** ()
- Qt::HANDLE **x11AppColormap** ()
- bool **x11AppDefaultColormap** ()
- void * **x11AppVisual** ()
- bool **x11AppDefaultVisual** ()
- void **x11SetAppDpiX** (int dpi)
- void **x11SetAppDpiY** (int dpi)

Protected Members

- **QPaintDevice** (uint devflags)
- virtual bool **cmd** (int, QPainter *, QPDevCmdParam *)
- virtual int **metric** (int) const

Related Functions

- void **bitBlt** (QPaintDevice * dst, int dx, int dy, const QPaintDevice * src, int sx, int sy, int sw, int sh, Qt::RasterOp rop, bool ignoreMask)
- void **bitBlt** (QPaintDevice * dst, const QPoint & dp, const QPaintDevice * src, const QRect & sr, RasterOp rop)

Detailed Description

The QPaintDevice class is the base class of objects that can be painted.

A paint device is an abstraction of a two-dimensional space that can be drawn using a QPainter. The drawing capabilities are implemented by the subclasses QWidget, QPixmap, QPicture and QPrinter.

The default coordinate system of a paint device has its origin located at the top-left position. X increases to the right and Y increases downward. The unit is one pixel. There are several ways to set up a user-defined coordinate system using the painter - for example, by QPainter::setWorldMatrix().

Example (draw on a paint device):

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p;                        // our painter
    p.begin( this );                   // start painting the widget
    p.setPen( red );                   // red outline
    p.setBrush( yellow );              // yellow fill
    p.drawEllipse( 10, 20, 100,100 );  // 100x100 ellipse at positin (10, 20)
    p.end();                           // painting done
}
```

The bit block transfer is an extremely useful operation for copying pixels from one paint device to another (or to itself). It is implemented as the global function bitBlt().

Example (scroll widget contents 10 pixels to the right):

```
    bitBlt( myWidget, 10, 0, myWidget );
```

Warning: Qt requires that a QApplication object must exist before any paint devices can be created. Paint devices access window system resources, and these resources are not initialized before an application object is created.

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QPaintDevice::QPaintDevice (uint devflags) [protected]

Constructs a paint device with internal flags *devflags*. This constructor can be invoked only from subclasses of QPaintDevice.

QPaintDevice::~QPaintDevice () [virtual]

Destroys the paint device and frees window system resources.

bool QPaintDevice::cmd (int, QPainter *, QPDevCmdParam *) [virtual protected]

Internal virtual function that interprets drawing commands from the painter.

Implemented by subclasses that have no direct support for drawing graphics (external paint devices - for example, QPicture).

int QPaintDevice::devType () const

Returns the device type identifier, which is `QInternal::Widget` if the device is a QWidget, `QInternal::Pixmap` if it's a QPixmap, `QInternal::Printer` if it's a QPrinter, `QInternal::Picture` if it's a QPicture or `QInternal::UndefinedDevice` in other cases (which should never happen).

HDC QPaintDevice::handle () const [virtual]

Returns the window system handle of the paint device, for low-level access. Using this function is not portable.

The HANDLE type varies with platform; see qpaintdevice.h and qwindowdefs.h for details.

See also x11Display() [p. 174].

bool QPaintDevice::isExtDev () const

Returns TRUE if the device is an external paint device; otherwise returns FALSE.

External paint devices cannot be bitBlt()'ed from. QPicture and QPrinter are external paint devices.

int QPaintDevice::metric (int) const [virtual protected]

Internal virtual function that returns paint device metrics.

Please use the QPaintDeviceMetrics class instead.

Reimplemented in QPixmap, QWidget and QPicture.

bool QPaintDevice::paintingActive () const

Returns TRUE if the device is being painted, i.e. someone has called QPainter::begin() but not yet QPainter::end() for this device; otherwise returns FALSE.

See also QPainter::isActive() [p. 197].

int QPaintDevice::x11AppCells () [static]

Returns the number of entries in the colormap of the X display global to the application (X11 only). Using this function is not portable.

See also x11Colormap() [p. 174].

Qt::HANDLE QPaintDevice::x11AppColormap () [static]

Returns the colormap of the X display global to the application (X11 only). Using this function is not portable.

See also x11Cells() [p. 174].

bool QPaintDevice::x11AppDefaultColormap () [static]

Returns the default colormap of the X display global to the application (X11 only). Using this function is not portable.

See also x11Cells() [p. 174].

bool QPaintDevice::x11AppDefaultVisual () [static]

Returns the default Visual of the X display global to the application (X11 only). Using this function is not portable.

int QPaintDevice::x11AppDepth () [static]

Returns the depth of the X display global to the application (X11 only). Using this function is not portable.

See also QPixmap::defaultDepth() [p. 231].

Display * QPaintDevice::x11AppDisplay () [static]

Returns a pointer to the X display global to the application (X11 only). Using this function is not portable.

See also handle() [p. 172].

int QPaintDevice::x11AppDpiX () [static]

Returns the horizontal DPI of the X display (X11 only). Using this function is not portable. See QPaintDeviceMetrics for portable access to related information. Using this function is not portable.

See also x11AppDpiY() [p. 173], x11SetAppDpiX() [p. 174] and QPaintDeviceMetrics::logicalDpiX() [p. 177].

int QPaintDevice::x11AppDpiY () [static]

Returns the vertical DPI of the X11 display (X11 only). Using this function is not portable. See QPaintDeviceMetrics for portable access to related information. Using this function is not portable.

See also x11AppDpiX() [p. 173], x11SetAppDpiY() [p. 174] and QPaintDeviceMetrics::logicalDpiY() [p. 177].

int QPaintDevice::x11AppScreen () [static]

Returns the screen number on the X display global to the application (X11 only). Using this function is not portable.

void * QPaintDevice::x11AppVisual () [static]

Returns the Visual of the X display global to the application (X11 only). Using this function is not portable.

int QPaintDevice::x11Cells () const

Returns the number of entries in the colormap of the X display for the paint device (X11 only). Using this function is not portable.

See also x11Colormap() [p. 174].

Qt::HANDLE QPaintDevice::x11Colormap () const

Returns the colormap of the X display for the paint device (X11 only). Using this function is not portable.

See also x11Cells() [p. 174].

bool QPaintDevice::x11DefaultColormap () const

Returns the default colormap of the X display for the paint device (X11 only). Using this function is not portable.

See also x11Cells() [p. 174].

bool QPaintDevice::x11DefaultVisual () const

Returns the default Visual of the X display for the paint device (X11 only). Using this function is not portable.

int QPaintDevice::x11Depth () const

Returns the depth of the X display for the paint device (X11 only). Using this function is not portable.

See also QPixmap::defaultDepth() [p. 231].

Display * QPaintDevice::x11Display () const

Returns a pointer to the X display for the paint device (X11 only). Using this function is not portable.

See also handle() [p. 172].

int QPaintDevice::x11Screen () const

Returns the screen number on the X display for the paint device (X11 only). Using this function is not portable.

void QPaintDevice::x11SetAppDpiX (int dpi) [static]

Sets the value returned by x11AppDpiX() to *dpi*. The default is determined by the display configuration. Changing this value will alter the scaling of fonts and many other metrics and is not recommended. Using this function is not portable.

See also x11SetAppDpiY() [p. 174].

void QPaintDevice::x11SetAppDpiY (int dpi) [static]

Sets the value returned by x11AppDpiY() to *dpi*. The default is determined by the display configuration. Changing this value will alter the scaling of fonts and many other metrics and is not recommended. Using this function is not

portable.

See also x11SetAppDpiX() [p. 174].

void * QPaintDevice::x11Visual () const

Returns the Visual of the X display for the paint device (X11 only). Using this function is not portable.

Related Functions

void bitBlt (QPaintDevice * dst, int dx, int dy, const QPaintDevice * src, int sx, int sy, int sw, int sh, Qt::RasterOp rop, bool ignoreMask)

Copies a block of pixels from *src* to *dst*, perhaps merging each pixel according to the raster operation *rop*. *sx*, *sy* is the top-left pixel in *src* (0, 0) by default, *dx*, *dy* is the top-left position in *dst* and *sw*, *sh* is the size of the copied block (all of *src* by default).

The most common values for *rop* are CopyROP and XorROP; the Qt::RasterOp documentation defines all the possible values.

If *ignoreMask* is TRUE (the default is FALSE) and *src* is a masked QPixmap, the entire blit is masked by *src*->mask().

If *src*, *dst*, *sw* or *sh* is 0, bitBlt() does nothing. If *sw* or *sh* is negative bitBlt() copies starting at *sx* (and resp. *sy*) and ending at the right end (resp. bottom) of *src*.

src must be a QWidget or QPixmap. You cannot blit from a QPrinter, for example. bitBlt() does nothing if you attempt to blit from an unsupported device.

bitBlt() does nothing if *src* has a greater depth than *dst*. If you need to, for example, draw a 24-bit pixmap on an 8-bit widget, you must use drawPixmap().

void bitBlt (QPaintDevice * dst, const QPoint & dp, const QPaintDevice * src, const QRect & sr, RasterOp rop)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Overloaded bitBlt() with the destination point *dp* and source rectangle *sr*.

QPaintDeviceMetrics Class Reference

The QPaintDeviceMetrics class provides information about a paint device.

`#include <qpaintdevicemetrics.h>`

Public Members

- **QPaintDeviceMetrics** (const QPaintDevice * pd)
- int **width** () const
- int **height** () const
- int **widthMM** () const
- int **heightMM** () const
- int **logicalDpiX** () const
- int **logicalDpiY** () const
- int **numColors** () const
- int **depth** () const

Detailed Description

The QPaintDeviceMetrics class provides information about a paint device.

Sometimes when drawing graphics it is necessary to obtain information about the physical characteristics of a paint device. This class provides this information. For example, to compute the aspect ratio of a paint device:

```
QPaintDeviceMetrics pdm( myWidget );
double aspect = (double)pdm.widthMM() / (double)pdm.heightMM();
```

QPaintDeviceMetrics contains methods to provide the width and height of a device in both pixels (width() and height()) and millimeters (widthMM() and heightMM()), the number of colors the device supports (numColors()), the number of bit planes (depth()), and the resolution of the device (logicalDpiX() and logicalDpiY()).

It is not always possible for QPaintDeviceMetrics to compute the values you ask for, particularly for external devices. The ultimate example is asking for the resolution of of a QPrinter that is set to "print to file": who knows what printer that file will end up on?

See also Graphics Classes and Image Processing Classes.

Member Function Documentation

QPaintDeviceMetrics::QPaintDeviceMetrics (const QPaintDevice * pd)

Constructs a metric for the paint device *pd*.

int QPaintDeviceMetrics::depth () const

Returns the bit depth (number of bit planes) of the paint device.

int QPaintDeviceMetrics::height () const

Returns the height of the paint device in default coordinate system units (e.g. pixels for QPixmap and QWidget).

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp and qwerty/qwerty.cpp.

int QPaintDeviceMetrics::heightMM () const

Returns the height of the paint device, measured in millimeters.

int QPaintDeviceMetrics::logicalDpiX () const

Returns the horizontal resolution of the device in dots per inch, which is used when computing font sizes. For X, this is usually the same as could be computed from widthMM(), but it varies on Windows.

Examples: helpviewer/helpwindow.cpp and qwerty/qwerty.cpp.

int QPaintDeviceMetrics::logicalDpiY () const

Returns the vertical resolution of the device in dots per inch, which is used when computing font sizes. For X, this is usually the same as could be computed from heightMM(), but it varies on Windows.

Example: helpviewer/helpwindow.cpp.

int QPaintDeviceMetrics::numColors () const

Returns the number of different colors available for the paint device.

int QPaintDeviceMetrics::width () const

Returns the width of the paint device in default coordinate system units (e.g. pixels for QPixmap and QWidget).

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp and qwerty/qwerty.cpp.

int QPaintDeviceMetrics::widthMM () const

Returns the width of the paint device, measured in millimeters.

QPainter Class Reference

The QPainter class does low-level painting e.g. on widgets.

`#include <qpainter.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- enum **CoordinateMode** { CoordDevice, CoordPainter }
- **QPainter** ()
- **QPainter** (const QPaintDevice * pd, bool unclipped = FALSE)
- **QPainter** (const QPaintDevice * pd, const QWidget * copyAttributes, bool unclipped = FALSE)
- **~QPainter** ()
- bool **begin** (const QPaintDevice * pd, bool unclipped = FALSE)
- bool **begin** (const QPaintDevice * pd, const QWidget * copyAttributes, bool unclipped = FALSE)
- bool **end** ()
- QPaintDevice * **device** () const
- bool **isActive** () const
- void **flush** (const QRegion & region, CoordinateMode cm = CoordDevice)
- void **flush** ()
- void **save** ()
- void **restore** ()
- QFontMetrics **fontMetrics** () const
- QFontInfo **fontInfo** () const
- const QFont & **font** () const
- void **setFont** (const QFont & font)
- const QPen & **pen** () const
- void **setPen** (const QPen & pen)
- void **setPen** (PenStyle style)
- void **setPen** (const QColor & color)
- const QBrush & **brush** () const
- void **setBrush** (const QBrush & brush)
- void **setBrush** (BrushStyle style)
- void **setBrush** (const QColor & color)
- QPoint **pos** () const
- const QColor & **backgroundColor** () const
- void **setBackgroundColor** (const QColor & c)
- BGMode **backgroundMode** () const
- void **setBackgroundMode** (BGMode m)
- RasterOp **rasterOp** () const

- void **setRasterOp** (RasterOp r)
- const QPoint & **brushOrigin** () const
- void **setBrushOrigin** (int x, int y)
- void **setBrushOrigin** (const QPoint & p)
- bool **hasViewXForm** () const
- bool **hasWorldXForm** () const
- void **setViewXForm** (bool enable)
- QRect **window** () const
- void **setWindow** (const QRect & r)
- void **setWindow** (int x, int y, int w, int h)
- QRect **viewport** () const
- void **setViewport** (const QRect & r)
- void **setViewport** (int x, int y, int w, int h)
- void **setWorldXForm** (bool enable)
- const QWMatrix & **worldMatrix** () const
- void **setWorldMatrix** (const QWMatrix & m, bool combine = FALSE)
- void saveWorldMatrix () *(obsolete)*
- void restoreWorldMatrix () *(obsolete)*
- void **scale** (double sx, double sy)
- void **shear** (double sh, double sv)
- void **rotate** (double a)
- void **translate** (double dx, double dy)
- void **resetXForm** ()
- QPoint **xForm** (const QPoint & pv) const
- QRect **xForm** (const QRect & rv) const
- QPointArray **xForm** (const QPointArray & av) const
- QPointArray **xForm** (const QPointArray & av, int index, int npoints) const
- QPoint **xFormDev** (const QPoint & pd) const
- QRect **xFormDev** (const QRect & rd) const
- QPointArray **xFormDev** (const QPointArray & ad) const
- QPointArray **xFormDev** (const QPointArray & ad, int index, int npoints) const
- void **setClipping** (bool enable)
- bool **hasClipping** () const
- QRegion **clipRegion** (CoordinateMode m = CoordDevice) const
- void **setClipRect** (const QRect & r, CoordinateMode m = CoordDevice)
- void **setClipRect** (int x, int y, int w, int h, CoordinateMode m = CoordDevice)
- void **setClipRegion** (const QRegion & rgn, CoordinateMode m = CoordDevice)
- void **drawPoint** (int x, int y)
- void **drawPoint** (const QPoint & p)
- void **drawPoints** (const QPointArray & a, int index = 0, int npoints = -1)
- void **moveTo** (int x, int y)
- void **moveTo** (const QPoint & p)
- void **lineTo** (int x, int y)
- void **lineTo** (const QPoint & p)
- void **drawLine** (int x1, int y1, int x2, int y2)
- void **drawLine** (const QPoint & p1, const QPoint & p2)
- void **drawRect** (int x, int y, int w, int h)
- void **drawRect** (const QRect & r)
- void **drawWinFocusRect** (int x, int y, int w, int h)
- void **drawWinFocusRect** (int x, int y, int w, int h, const QColor & bgColor)

- void **drawWinFocusRect** (const QRect & r)
- void **drawWinFocusRect** (const QRect & r, const QColor & bgColor)
- void **drawRoundRect** (int x, int y, int w, int h, int xRnd = 25, int yRnd = 25)
- void **drawRoundRect** (const QRect & r, int xRnd = 25, int yRnd = 25)
- void **drawEllipse** (int x, int y, int w, int h)
- void **drawEllipse** (const QRect & r)
- void **drawArc** (int x, int y, int w, int h, int a, int alen)
- void **drawArc** (const QRect & r, int a, int alen)
- void **drawPie** (int x, int y, int w, int h, int a, int alen)
- void **drawPie** (const QRect & r, int a, int alen)
- void **drawChord** (int x, int y, int w, int h, int a, int alen)
- void **drawChord** (const QRect & r, int a, int alen)
- void **drawLineSegments** (const QPointArray & a, int index = 0, int nlines = -1)
- void **drawPolyline** (const QPointArray & a, int index = 0, int npoints = -1)
- void **drawPolygon** (const QPointArray & a, bool winding = FALSE, int index = 0, int npoints = -1)
- void **drawConvexPolygon** (const QPointArray & pa, int index = 0, int npoints = -1)
- void **drawCubicBezier** (const QPointArray & a, int index = 0)
- void **drawPixmap** (int x, int y, const QPixmap & pixmap, int sx = 0, int sy = 0, int sw = -1, int sh = -1)
- void **drawPixmap** (const QPoint & p, const QPixmap & pm, const QRect & sr)
- void **drawPixmap** (const QPoint & p, const QPixmap & pm)
- void **drawPixmap** (const QRect & r, const QPixmap & pm)
- void **drawImage** (int x, int y, const QImage & image, int sx = 0, int sy = 0, int sw = -1, int sh = -1, int conversionFlags = 0)
- void **drawImage** (const QPoint &, const QImage &, const QRect & sr, int conversionFlags = 0)
- void **drawImage** (const QPoint & p, const QImage & i, int conversion_flags = 0)
- void **drawImage** (const QRect & r, const QImage & i)
- void **drawTiledPixmap** (int x, int y, int w, int h, const QPixmap & pixmap, int sx = 0, int sy = 0)
- void **drawTiledPixmap** (const QRect & r, const QPixmap & pm, const QPoint & sp)
- void **drawTiledPixmap** (const QRect & r, const QPixmap & pm)
- void drawPicture (const QPicture & pic) *(obsolete)*
- void **drawPicture** (int x, int y, const QPicture & pic)
- void **drawPicture** (const QPoint & p, const QPicture & pic)
- void **fillRect** (int x, int y, int w, int h, const QBrush & brush)
- void **fillRect** (const QRect & r, const QBrush & brush)
- void **eraseRect** (int x, int y, int w, int h)
- void **eraseRect** (const QRect & r)
- enum **TextDirection** { Auto, RTL, LTR }
- void **drawText** (int x, int y, const QString &, int len = -1, TextDirection dir = Auto)
- void **drawText** (const QPoint &, const QString &, int len = -1, TextDirection dir = Auto)
- void **drawText** (int x, int y, const QString &, int pos, int len, TextDirection dir = Auto)
- void **drawText** (const QPoint & p, const QString &, int pos, int len, TextDirection dir = Auto)
- void **drawText** (int x, int y, int w, int h, int flags, const QString &, int len = -1, QRect * br = 0, QTextParag ** internal = 0)
- void **drawText** (const QRect & r, int tf, const QString & str, int len = -1, QRect * brect = 0, QTextParag ** internal = 0)
- QRect **boundingRect** (int x, int y, int w, int h, int flags, const QString &, int len = -1, QTextParag ** intern = 0)
- QRect **boundingRect** (const QRect & r, int flags, const QString & str, int len = -1, QTextParag ** internal = 0)
- int **tabStops** () const
- void **setTabStops** (int ts)

- int * **tabArray** () const
- void **setTabArray** (int * ta)
- HDC **handle** () const

Static Public Members

- void **redirect** (QPaintDevice * pdev, QPaintDevice * replacement)
- void **initialize** ()
- void **cleanup** ()

Related Functions

- void **qDrawShadeLine** (QPainter * p, int x1, int y1, int x2, int y2, const QColorGroup & g, bool sunken, int lineWidth, int midLineWidth)
- void **qDrawShadeRect** (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, int lineWidth, int midLineWidth, const QBrush * fill)
- void **qDrawShadePanel** (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, int lineWidth, const QBrush * fill)
- void **qDrawWinButton** (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, const QBrush * fill)
- void **qDrawWinPanel** (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, const QBrush * fill)
- void **qDrawPlainRect** (QPainter * p, int x, int y, int w, int h, const QColor & c, int lineWidth, const QBrush * fill)

Detailed Description

The QPainter class does low-level painting e.g. on widgets.

The painter provides highly optimized functions to do most of the drawing GUI programs require. QPainter can draw everything from simple lines to complex shapes like pies and chords. It can also draw aligned text and pixmaps. Normally, it draws in a "natural" coordinate system, but it can also do view and world transformation.

The typical use of a painter is:

- Construct a painter.
- Set a pen, a brush etc.
- Draw.
- Destroy the painter.

Mostly, all this is done inside a paint event. (In fact, 99% of all QPainter use is in a reimplementation of QWidget::paintEvent(), and the painter is heavily optimized for such use.) Here's one very simple example:

```
void SimpleExampleWidget::paintEvent()
{
    QPainter paint( this );
    paint.setPen( Qt::blue );
    paint.drawText( rect(), AlignCenter, "The Text" );
}
```

Simple. However, there are many settings you may use:

- font() is the currently set font. If you set a font that isn't available, Qt finds a close match. In fact font() returns what you set using setFont() and fontInfo() returns the font actually being used (which may be the same).
- brush() is the currently set brush; the color or pattern that's used for filling e.g. circles.
- pen() is the currently set pen; the color or stipple that's used for drawing lines or boundaries.
- backgroundMode() is `Opaque` or `Transparent`, i.e. whether backgroundColor() is used or not.
- backgroundColor() only applies when backgroundMode() is Opaque and pen() is a stipple. In that case, it describes the color of the background pixels in the stipple.
- rasterOp() is how pixels drawn interact with the pixels already there.
- brushOrigin() is the origin of the tiled brushes, normally the origin of the window.
- viewport(), window(), worldMatrix() and many more make up the painter's coordinate transformation system. See The Coordinate System for an explanation of this, or see below for a very brief overview of the functions.
- clipping() is whether the painter clips at all. (The paint device clips, too.) If the painter clips, it clips to clipRegion().
- pos() is the current position, set by moveTo() and used by lineTo().

Note that some of these settings mirror settings in some paint devices, e.g. QWidget::font(). QPainter::begin() (or the QPainter constructor) copies these attributes from the paint device. Calling, for example, QWidget::setFont() doesn't take effect until the next time a painter begins painting on it.

save() saves all of these settings on an internal stack, restore() pops them back.

The core functionality of QPainter is drawing, and there are functions to draw most primitives: drawPoint(), drawPoints(), drawLine(), drawRect(), drawWinFocusRect(), drawRoundRect(), drawEllipse(), drawArc(), drawPie(), drawChord(), drawLineSegments(), drawPolyline(), drawPolygon(), drawConvexPolygon() and drawCubicBezier(). All of these functions take integer coordinates; there are no floating-point versions. Floatint-point operations are outside the scope of QPainter (providing *fast* drawing of the things GUI programs draw).

There are functions to draw pixmaps/images, namely drawPixmap(), drawImage() and drawTiledPixmap(). drawPixmap() and drawImage() produce the same result, except that drawPixmap() is faster on-screen and drawImage() faster and sometimes better on QPrinter and QPicture.

Text drawing is done using drawText(), and when you need fine-grained positioning, boundingRect() tells you where a given drawText() command would draw.

There is a drawPicture() that draws the contents of an entire QPicture using this painter. drawPicture() is the only function that disregards all the painter's settings: the QPicture has its own settings.

Normally, the QPainter operates on the device's own coordinate system (usually pixels), but QPainter has good support for coordinate transformation. See The Coordinate System for a more general overview and a simple example.

The most common functions used are scale(), rotate(), translate() and shear(), all of which operate on the worldMatrix(). setWorldMatrix() can replace or add to the currently set matrix().

setViewport() sets the rectangle on which QPainter operates. The default is the entire device, which is usually fine, except on printers. setWindow() sets the coordinate system, that is, the rectangle that maps to viewport(). What's drawn inside the window() ends up being inside the viewport(). The window's default is the same as the viewport, and if you don't use the transformations, they are optimized away, gaining another little bit of speed.

After all the coordinate transformation is done, QPainter can clip the drawing to an arbitrary rectangle or region. hasClipping() is TRUE if QPainter clips, and clipRegion() returns the clip region. You can set it using either setClipRegion() or setClipRect(). Note that the clipping can be slow. It's all system-dependent, but as a rule of thumb, you can assume that drawing speed is inversely proportional to the number of rectangles in the clip region.

After QPainter's clipping, the paint device may also clip. For example, most widgets clip away the pixels used by child widgets, and most printers clip away an area near the edges of the paper. This additional clipping is not reflected by the return value of clipRegion() or hasClipping().

QPainter also includes some less-used functions that are very useful the few times you need them.

isActive() indicates whether the painter is active. begin() (and the most usual constructor) makes it active. end() (and the destructor) deactivates it. If the painter is active, device() returns the paint device on which the painter paints.

Sometimes it is desirable to make someone else paint on an unusual QPaintDevice. QPainter supports a static function to do this, redirect(). We recommend not using it, but for some hacks it's perfect.

setTabStops() and setTabArray() can change where the tab stops are, but these are very seldomly used.

Warning: Note that QPainter does not attempt to work around coordinate limitations in the underlying window system. Some platforms may behave incorrectly with coordinates as small as +/-4000.

See also QPaintDevice [p. 170], QWidget [Widgets with Qt], QPixmap [p. 225], QPrinter [p. 254], QPicture [p. 220], Application Walkthrough, Coordinate System Overview [p. 4], Graphics Classes and Image Processing Classes.

Member Type Documentation

QPainter::CoordinateMode

- `QPainter::CoordDevice`
- `QPainter::CoordPainter`

See also clipRegion() [p. 187].

QPainter::TextDirection

- `QPainter::Auto`
- `QPainter::RTL` - right to left
- `QPainter::LTR` - left to right

See also drawText() [p. 192].

Member Function Documentation

QPainter::QPainter ()

Constructs a painter.

Notice that all painter settings (setPen, setBrush etc.) are reset to default values when begin() is called.

See also begin() [p. 184] and end() [p. 194].

QPainter::QPainter (const QPaintDevice * pd, bool unclipped = FALSE)

Constructs a painter that begins painting the paint device *pd* immediately. Depending on the underlying graphic system the painter will paint over children of the paintdevice if *unclipped* is TRUE.

This constructor is convenient for short-lived painters, e.g. in a paint event and should be used only once. The constructor calls begin() for you and the QPainter destructor automatically calls end().

Here's an example using begin() and end():

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p;
    p.begin( this );
    p.drawLine( ... );      // drawing code
    p.end();
}
```

The same example using this constructor:

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    p.drawLine( ... );      // drawing code
}
```

See also begin() [p. 184] and end() [p. 194].

QPainter::QPainter (const QPaintDevice * pd, const QWidget * copyAttributes, bool unclipped = FALSE)

Constructs a painter that begins painting the paint device *pd* immediately, with the default arguments taken from *copyAttributes*. The painter will paint over children of the paint device if *unclipped* is TRUE (although this is not supported on all platforms).

See also begin() [p. 184].

QPainter::~QPainter ()

Destroys the painter.

const QColor & QPainter::backgroundColor () const

Returns the current background color.

See also setBackgroundColor() [p. 199] and QColor [p. 74].

BGMode QPainter::backgroundMode () const

Returns the current background mode.

See also setBackgroundMode() [p. 199] and BGMode [Additional Functionality with Qt].

bool QPainter::begin (const QPaintDevice * pd, bool unclipped = FALSE)

Begins painting the paint device *pd* and returns TRUE if successful, or FALSE if an error occurs. If *unclipped* is TRUE, the painting will not be clipped at the paint device's boundaries, yet note that this is not supported by all platforms.

The errors that can occur are serious problems, such as these:

```
p->begin( 0 ); // impossible - paint device cannot be 0

QPixmap pm( 0, 0 );
p->begin( pm ); // impossible - pm.isNull();

p->begin( myWidget );
p2->begin( myWidget ); // impossible - only one painter at a time
```

Note that most of the time, you can use one of the constructors instead of begin(), and that end() is automatically done at destruction.

Warning: A paint device can only be painted by one painter at a time.

See also end() [p. 194] and flush() [p. 195].

Examples: aclock/aclock.cpp, application/application.cpp, desktop/desktop.cpp, hello/hello.cpp, picture/picture.cpp, t10/cannon.cpp and xform/xform.cpp.

bool QPainter::begin (const QPaintDevice * pd, const QWidget * copyAttributes, bool unclipped = FALSE)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This version opens the painter on a paint device *pd* and sets the initial pen, background color and font from *copyAttributes*, painting over the paint devices' children when *unclipped* is TRUE. This is equivalent to:

```
QPainter p;
p.begin( pd );
p.setPen( copyAttributes->foregroundColor() );
p.setBackgroundColor( copyAttributes->backgroundColor() );
p.setFont( copyAttributes->font() );
```

This begin function is convenient for double buffering. When you draw in a pixmap instead of directly in a widget (to later bitBlt the pixmap into the widget) you will need to set the widgets's font etc. This function does exactly that.

Example:

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPixmap pm(size());
    QPainter p;
    p.begin(&pm, this);
    // ... potentially flickering paint operation ...
    p.end();
    bitBlt(this, 0, 0, &pm);
}
```

See also end() [p. 194].

QRect QPainter::boundingRect (int x, int y, int w, int h, int flags, const QString &, int len = -1, QTextParag ** intern = 0)

Returns the bounding rectangle of the aligned text that would be printed with the corresponding drawText() function using the first *len* characters of the string if *len* is > -1, or the whole of the string if *len* is -1. The drawing,

and hence the bounding rectangle, is constrained to the rectangle that begins at point *(x, y)* with width *w* and hight *h*.

The *flags* argument is the bitwise OR of the following flags:

- AlignAuto aligns according to the language, usually left.
- AlignLeft aligns to the left border.
- AlignRight aligns to the right border.
- AlignHCenter aligns horizontally centered.
- AlignTop aligns to the top border.
- AlignBottom aligns to the bottom border.
- AlignVCenter aligns vertically centered
- AlignCenter (== AlignHCenter | AlignVCenter)
- SingleLine ignores newline characters in the text.
- ExpandTabs expands tabulators.
- ShowPrefix interprets "&x" as "x" underlined.
- WordBreak breaks the text to fit the rectangle.

Horizontal alignment defaults to AlignLeft and vertical alignment defaults to AlignTop.

If several of the horizontal or several of the vertical alignment flags are set, the resulting alignment is undefined.

The *intern* parameter should not be used.

See also Qt::TextFlags [Additional Functionality with Qt].

QRect QPainter::boundingRect (const QRect & r, int flags, const QString & str, int len = -1, QTextParag ** internal = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the bounding rectangle of the aligned text that would be printed with the corresponding drawText() function using the first *len* characters from *str* if *len* is > -1, or the whole of *str* if *len* is -1. The drawing, and hence the bounding rectangle, is constrained to the rectangle *r*.

The *internal* parameter should not be used.

See also drawText() [p. 192], fontMetrics() [p. 196], QFontMetrics::boundingRect() [Additional Functionality with Qt] and Qt::TextFlags [Additional Functionality with Qt].

const QBrush & QPainter::brush () const

Returns the current painter brush.

See also QPainter::setBrush() [p. 199].

Examples: themes/metal.cpp and themes/wood.cpp.

const QPoint & QPainter::brushOrigin () const

Returns the brush origin currently set.

See also setBrushOrigin() [p. 200].

void QPainter::cleanup () [static]

Internal function that cleans up the painter.

QRegion QPainter::clipRegion (CoordinateMode m = CoordDevice) const

Returns the currently set clip region. Note that the clip region is given in physical device coordinates and *not* subject to any coordinate transformation if *m* is equal to CoordDevice (the default). If *m* equals CoordPainter the returned region is in model coordinates.

See also setClipRegion() [p. 200], setClipRect() [p. 200], setClipping() [p. 200] and QPainter::CoordinateMode [p. 183].

Example: themes/wood.cpp.

QPaintDevice * QPainter::device () const

Returns the paint device on which this painter is currently painting, or null if the painter is not active.

See also QPaintDevice::paintingActive() [p. 172].

Examples: helpviewer/helpwindow.cpp and listboxcombo/listboxcombo.cpp.

void QPainter::drawArc (int x, int y, int w, int h, int a, int alen)

Draws an arc defined by the rectangle *(x, y, w, h)*, the start angle *a* and the arc length *alen*.

The angles *a* and *alen* are 1/16th of a degree, i.e. a full circle equals 5760 (16*360). Positive values of *a* and *alen* mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

Example:

```
QPainter p( myWidget );
p.drawArc( 10,10, 70,100, 100*16, 160*16 ); // draws a "(" arc
```

See also drawPie() [p. 190] and drawChord() [p. 187].

void QPainter::drawArc (const QRect & r, int a, int alen)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the arc that fits inside the rectangle *r* with start angle *a* and arc length *alen*.

void QPainter::drawChord (int x, int y, int w, int h, int a, int alen)

Draws a chord defined by the rectangle *(x, y, w, h)*, the start angle *a* and the arc length *alen*.

The chord is filled with the current brush().

The angles *a* and *alen* are 1/16th of a degree, i.e. a full circle equals 5760 (16*360). Positive values of *a* and *alen* mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

See also drawArc() [p. 187] and drawPie() [p. 190].

void QPainter::drawChord (const QRect & r, int a, int alen)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a chord that fits inside the rectangle *r* with start angle *a* and arc length *alen*.

void QPainter::drawConvexPolygon (const QPointArray & pa, int index = 0, int npoints = -1)

Draws the convex polygon defined by the *npoints* points in *pa* starting at *pa[index]* (*index* defaults to 0).

If the supplied polygon is not convex, the results are undefined.

On some platforms (e.g., X Window), this is faster than drawPolygon().

Example: aclock/aclock.cpp.

void QPainter::drawCubicBezier (const QPointArray & a, int index = 0)

Draws a cubic Bezier curve defined by the control points in *a*, starting at *a[index]* (*index* defaults to 0).

Control points after *a[index + 3]* are ignored. Nothing happens if there aren't enough control points.

void QPainter::drawEllipse (int x, int y, int w, int h)

Draws an ellipse with center at *(x + w/2, y + h/2)* and size *(w, h)*.

Examples: drawdemo/drawdemo.cpp, picture/picture.cpp and tictac/tictac.cpp.

void QPainter::drawEllipse (const QRect & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the ellipse that fits inside rectangle *r*.

void QPainter::drawImage (int x, int y, const QImage & image, int sx = 0, int sy = 0, int sw = -1, int sh = -1, int conversionFlags = 0)

Draws at *(x, y)* the *sw* by *sh* area of pixels from *(sx, sy)* in *image*, using *conversionFlags* if the image needs to be converted to a pixmap. The default value for *conversionFlags* is 0; see convertFromImage() for information about what other values do.

This function may convert *image* to a pixmap and then draw it, if device() is a QPixmap or a QWidget, or else draw it directly, if device() is a QPrinter or QPicture.

See also drawPixmap() [p. 190] and QPixmap::convertFromImage() [p. 230].

void QPainter::drawImage (const QPoint &, const QImage &, const QRect & sr, int conversionFlags = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the rectangle *sr* from the image at the given point.

void QPainter::drawImage (const QPoint & p, const QImage & i, int conversion_flags = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the image *i* at point *p*.

If the image needs to be modified to fit in a lower-resolution result (eg. converting from 32-bit to 8-bit), use the *conversion_flags* to specify how you'd prefer this to happen.

See also Qt::ImageConversionFlags [Additional Functionality with Qt].

void QPainter::drawImage (const QRect & r, const QImage & i)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the image *i* into the rectangle *r*. The image will be scaled to fit the rectangle if image and rectangle dimensions differ.

void QPainter::drawLine (int x1, int y1, int x2, int y2)

Draws a line from (*x1, y1*) to (*x2, y2*) and sets the current pen position to (*x2, y2*).

See also pen() [p. 197].

Examples: aclock/aclock.cpp, drawlines/connect.cpp, progress/progress.cpp, splitter/splitter.cpp, themes/metal.cpp and themes/wood.cpp.

void QPainter::drawLine (const QPoint & p1, const QPoint & p2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a line from point *p1* to point *p2*.

void QPainter::drawLineSegments (const QPointArray & a, int index = 0, int nlines = -1)

Draws *nlines* separate lines from points defined in *a*, starting at a[*index*] (*index* defaults to 0). If *nlines* is -1 (the defauls) all points until the end of the array are used (i.e. (a.size()-index)/2 lines are drawn).

Draws the 1st line from *a[index]* to *a[index+1]*. Draws the 2nd line from *a[index+2]* to *a[index+3]* etc.

See also drawPolyline() [p. 191], drawPolygon() [p. 191] and QPen [p. 215].

void QPainter::drawPicture (int x, int y, const QPicture & pic)

Replays the picture *pic* translated by (*x, y*).

This function does exactly the same as QPicture::play() when called with (*x, y*) = (0, 0).

Examples: picture/picture.cpp and xform/xform.cpp.

void QPainter::drawPicture (const QPicture & pic)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Use one of the other QPainter::drawPicture() functions with a (0, 0) offset instead.

void QPainter::drawPicture (const QPoint & p, const QPicture & pic)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws picture *pic* at point *p*.

void QPainter::drawPie (int x, int y, int w, int h, int a, int alen)

Draws a pie defined by the rectangle *(x, y, w, h)*, the start angle *a* and the arc length *alen*.

The pie is filled with the current brush().

The angles *a* and *alen* are 1/16th of a degree, i.e. a full circle equals 5760 (16*360). Positive values of *a* and *alen* mean counter-clockwise while negative values mean clockwise direction. Zero degrees is at the 3'o clock position.

See also drawArc() [p. 187] and drawChord() [p. 187].

Examples: drawdemo/drawdemo.cpp, grapher/grapher.cpp, t10/cannon.cpp and t9/cannon.cpp.

void QPainter::drawPie (const QRect & r, int a, int alen)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a pie segment that fits inside the rectangle *r* with start angle *a* and arc length *alen*.

void QPainter::drawPixmap (int x, int y, const QPixmap & pixmap, int sx = 0, int sy = 0, int sw = -1, int sh = -1)

Draws a pixmap at *(x, y)* by copying a part of *pixmap* into the paint device.

(x, y) specify the top-left point in the paint device that is to be drawn onto. *(sx, sy)* specify the top-left point in *pixmap* that is to be drawn. The default is (0, 0).

(sw, sh) specify the size of the pixmap that is to be drawn. The default, (-1, -1), means all the way to the bottom right of the pixmap.

See also bitBlt() [p. 175] and QPixmap::setMask() [p. 236].

Examples: grapher/grapher.cpp, picture/picture.cpp, qdir/qdir.cpp, qtimage/qtimage.cpp, showimg/showimg.cpp, t10/cannon.cpp and xform/xform.cpp.

void QPainter::drawPixmap (const QPoint & p, const QPixmap & pm, const QRect & sr)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the rectangle *sr* of pixmap *pm* with its origin at point *p*.

void QPainter::drawPixmap (const QPoint & p, const QPixmap & pm)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the pixmap *pm* with its origin at point *p*.

void QPainter::drawPixmap (const QRect & r, const QPixmap & pm)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the pixmap *pm* into the rectangle *r*. The pixmap is scaled to fit the rectangle, if image and rectangle size disagree.

void QPainter::drawPoint (int x, int y)

Draws/plots a single point at *(x, y)* using the current pen.

See also QPen [p. 215].

Examples: desktop/desktop.cpp and drawlines/connect.cpp.

void QPainter::drawPoint (const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the point *p*.

void QPainter::drawPoints (const QPointArray & a, int index = 0, int npoints = -1)

Draws/plots an array of points, *a*, using the current pen.

If *index* is non-zero (the default is zero) only points from *index* are drawn. If *npoints* is negative (the default) the rest of the points from *index* are drawn. If is is zero or greater, *npoints* points are drawn.

void QPainter::drawPolygon (const QPointArray & a, bool winding = FALSE, int index = 0, int npoints = -1)

Draws the polygon defined by the *npoints* points in *a* starting at *a[index]*. (*index* defaults to 0.)

If *npoints* is -1 (the default) all points until the end of the array are used (i.e. a.size()-index line segments define the polygon).

The first point is always connected to the last point.

The polygon is filled with the current brush(). If *winding* is TRUE, the polygon is filled using the winding fill algorithm. If *winding* is FALSE, the polygon is filled using the even-odd (alternative) fill algorithm.

See also drawLineSegments() [p. 189], drawPolyline() [p. 191] and QPen [p. 215].

Examples: desktop/desktop.cpp and picture/picture.cpp.

void QPainter::drawPolyline (const QPointArray & a, int index = 0, int npoints = -1)

Draws the polyline defined by the *npoints* points in *a* starting at *a[index]*. (*index* defaults to 0.)

If *npoints* is -1 (the default) all points until the end of the array are used (i.e. a.size()-index-1 line segments are drawn).

See also drawLineSegments() [p. 189], drawPolygon() [p. 191] and QPen [p. 215].

Examples: scribble/scribble.cpp and themes/metal.cpp.

void QPainter::drawRect (int x, int y, int w, int h)

Draws a rectangle with upper left corner at *(x, y)* and with width *w* and height *h*.

See also QPen [p. 215] and drawRoundRect() [p. 192].

Examples: drawdemo/drawdemo.cpp, picture/picture.cpp, t10/cannon.cpp, t11/cannon.cpp, t9/cannon.cpp, tooltip/tooltip.cpp and trivial/trivial.cpp.

void QPainter::drawRect (const QRect & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the rectange *r*.

void QPainter::drawRoundRect (int x, int y, int w, int h, int xRnd = 25, int yRnd = 25)

Draws a rectangle with round corners at *(x, y)*, with width *w* and height *h*.

The *xRnd* and *yRnd* arguments specify how rounded the corners should be. 0 is angled corners, 99 is maximum roundedness.

The width and height include all of the drawn lines.

See also drawRect() [p. 192] and QPen [p. 215].

Examples: drawdemo/drawdemo.cpp and themes/wood.cpp.

void QPainter::drawRoundRect (const QRect & r, int xRnd = 25, int yRnd = 25)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a rounded rectange *r*, rounding to the x position *xRnd* and the y position *yRnd* on each corner.

void QPainter::drawText (const QPoint & p, const QString &, int pos, int len, TextDirection dir = Auto)

Draws the text from position *pos*, at point *p* If *len* is -1 the entire string is drawn, otherwise just the first *len* characters. The text's direction is specified by *dir*.

See also QPainter::TextDirection [p. 183].

Examples: desktop/desktop.cpp, drawdemo/drawdemo.cpp, grapher/grapher.cpp, picture/picture.cpp, progress/progress.cpp, t8/cannon.cpp and trivial/trivial.cpp.

void QPainter::drawText (int x, int y, const QString &, int len = -1, TextDirection dir = Auto)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the given text at position *x, y*. If *len* is -1 (the default) all the text is drawn, otherwise the first *len* characters are drawn. The text's direction is given by *dir*.

See also QPainter::TextDirection [p. 183].

void QPainter::drawText (const QPoint &, const QString &, int len = -1, TextDirection dir = Auto)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the text at the given point.

See also QPainter::TextDirection [p. 183].

void QPainter::drawText (int x, int y, const QString &, int pos, int len, TextDirection dir = Auto)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the text from position *pos*, at point *(x, y)*. If len is -1 the entire string is drawn, otherwise just the first len characters. The text's direction is specified by dir.

void QPainter::drawText (int x, int y, int w, int h, int flags, const QString &, int len = -1, QRect * br = 0, QTextParag ** internal = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws the given text within the rectangle starting at *x, y*, with width *w* and height *h*. If *len* is -1 (the default) all the text is drawn, otherwise the first *len* characters are drawn. The text's alignment is given in the *flags* parameter (see Qt::AlignmentFlags). *br* (if not null) is set to the actual bounding rectangle of the output. The *internal* parameter is for internal use only.

void QPainter::drawText (const QRect & r, int tf, const QString & str, int len = -1, QRect * brect = 0, QTextParag ** internal = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws at most *len* characters from *str* in the rectangle *r*.

Note that the meaning of *r*.y() is not the same for the two drawText() varieties.

This function draws formatted text. The *tf* text format is really of type Qt::AlignmentFlags.

Horizontal alignment defaults to AlignAuto and vertical alignment defaults to AlignTop.

brect (if not null) is set to the actual bounding rectangle of the output. *internal* is, yes, internal.

See also boundingRect() [p. 185].

void QPainter::drawTiledPixmap (int x, int y, int w, int h, const QPixmap & pixmap, int sx = 0, int sy = 0)

Draws a tiled *pixmap* in the specified rectangle.

(x, y) specifies the top-left point in the paint device that is to be drawn onto; with the width and height given by *w* and *h*. *(sx, sy)* specify the top-left point in *pixmap* that is to be drawn. The default is (0, 0).

Calling drawTiledPixmap() is similar to calling drawPixmap() several times to fill (tile) an area with a pixmap, but is potentially much more efficient depending on the underlying window system.

See also drawPixmap() [p. 190].

void QPainter::drawTiledPixmap (const QRect & r, const QPixmap & pm, const QPoint & sp)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a tiled pixmap, *pm*, inside rectangle *r* with its origin at point *sp*.

void QPainter::drawTiledPixmap (const QRect & r, const QPixmap & pm)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a tiled pixmap, *pm*, inside rectangle *r*.

void QPainter::drawWinFocusRect (int x, int y, int w, int h, const QColor & bgColor)

Draws a Windows focus rectangle with upper left corner at (x, y) and with width *w* and height *h* using a pen color that contrasts with *bgColor*.

This function draws a stippled rectangle (XOR is not used) that is used to indicate keyboard focus (when the QApplication::style() is `WindowStyle`).

The pen color used to draw the rectangle is either white or black depending on the color of *bgColor* (see QColor::gray()).

Warning: This function draws nothing if the coordinate system has been rotated or sheared.

See also drawRect() [p. 192] and QApplication::style() [Additional Functionality with Qt].

void QPainter::drawWinFocusRect (int x, int y, int w, int h)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a Windows focus rectangle with upper left corner at (x, y) and with width *w* and height *h*.

This function draws a stippled XOR rectangle that is used to indicate keyboard focus (when QApplication::style() is `WindowStyle`).

Warning: This function draws nothing if the coordinate system has been rotated or sheared.

See also drawRect() [p. 192] and QApplication::style() [Additional Functionality with Qt].

void QPainter::drawWinFocusRect (const QRect & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws rectangle *r* as a window focus rectangle.

void QPainter::drawWinFocusRect (const QRect & r, const QColor & bgColor)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws rectangle *r* as a window focus rectangle using background color *bgColor*.

bool QPainter::end ()

Ends painting. Any resources used while painting are released.

Note that while you mostly don't need to call end(), the destructor will do it, there is at least one common case, namely double buffering.

```
QPainter p( myPixmap, this )
// ...
p.end(); // stops drawing on myPixmap
p.begin( this );
p.drawPixmap( myPixmap );
```

Since you can't draw a QPixmap while it is being painted, it is necessary to close the active painter.

See also begin() [p. 184] and isActive() [p. 197].

Examples: aclock/aclock.cpp, application/application.cpp, desktop/desktop.cpp, hello/hello.cpp, picture/picture.cpp, t10/cannon.cpp and xform/xform.cpp.

void QPainter::eraseRect (int x, int y, int w, int h)

Erases the area inside *x*, *y*, *w*, *h*. Equivalent to `fillRect( x, y, w, h, backgroundColor() )`.

Examples: listboxcombo/listboxcombo.cpp and showimg/showimg.cpp.

void QPainter::eraseRect (const QRect & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Erases the area inside the rectangle *r*.

void QPainter::fillRect (int x, int y, int w, int h, const QBrush & brush)

Fills the rectangle *(x, y, w, h)* with the *brush*.

You can specify a QColor as *brush*, since there is a QBrush constructor that takes a QColor argument and creates a solid pattern brush.

See also drawRect() [p. 192].

Examples: listboxcombo/listboxcombo.cpp, progress/progress.cpp, qdir/qdir.cpp, qfd/fontdisplayer.cpp, themes/metal.cpp and themes/wood.cpp.

void QPainter::fillRect (const QRect & r, const QBrush & brush)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Fills the rectangle *r* using brush *brush*.

void QPainter::flush (const QRegion & region, CoordinateMode cm = CoordDevice)

Flushes any buffered drawing operations inside the region *region* using clipping mode *cm*.

The flush may update the whole device if the platform does not support flushing to a specified region.

See also CoordinateMode [p. 183].

void QPainter::flush ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Flushes any buffered drawing operations.

const QFont & QPainter::font () const

Returns the currently set painter font.

See also setFont() [p. 201] and QFont [Additional Functionality with Qt].

Example: fileiconview/qfileiconview.cpp.

QFontInfo QPainter::fontInfo () const

Returns the font info for the painter, if the painter is active. It is not possible to obtain font information for an inactive painter, so the return value is undefined if the painter is not active.

See also fontMetrics() [p. 196] and isActive() [p. 197].

QFontMetrics QPainter::fontMetrics () const

Returns the font metrics for the painter, if the painter is active. It is not possible to obtain metrics for an inactive painter, so the return value is undefined if the painter is not active.

See also fontInfo() [p. 196] and isActive() [p. 197].

Examples: action/application.cpp, application/application.cpp, desktop/desktop.cpp, drawdemo/drawdemo.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp and qwerty/qwerty.cpp.

HDC QPainter::handle () const

Returns the platform-dependent handle used for drawing.

bool QPainter::hasClipping () const

Returns TRUE if clipping has been set; otherwise returns FALSE.

See also setClipping() [p. 200].

Example: themes/wood.cpp.

bool QPainter::hasViewXForm () const

Returns TRUE if view transformation is enabled; otherwise returns FALSE.

See also setViewXForm() [p. 202] and xForm() [p. 205].

bool QPainter::hasWorldXForm () const

Returns TRUE if world transformation is enabled; otherwise returns FALSE.

See also setWorldXForm() [p. 203].

void QPainter::initialize () [static]

Internal function that initializes the painter.

bool QPainter::isActive () const

Returns TRUE if the painter is active painting, i.e. begin() has been called and end() has not yet been called; otherwise returns FALSE.

See also QPaintDevice::paintingActive() [p. 172].

Example: desktop/desktop.cpp.

void QPainter::lineTo (int x, int y)

Draws a line from the current pen position to *(x, y)* and sets *(x, y)* to be the new current pen position.

See also QPen [p. 215], moveTo() [p. 197], drawLine() [p. 189] and pos() [p. 197].

void QPainter::lineTo (const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Draws a line to the point *p*.

void QPainter::moveTo (int x, int y)

Sets the current pen position to *(x, y)*

See also lineTo() [p. 197] and pos() [p. 197].

void QPainter::moveTo (const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Moves to the point *p*.

const QPen & QPainter::pen () const

Returns the current pen for the painter.

See also setPen() [p. 201].

Examples: progress/progress.cpp and themes/wood.cpp.

QPoint QPainter::pos () const

Returns the current position of the pen.

See also moveTo() [p. 197].

RasterOp QPainter::rasterOp () const

Returns the current raster operation.

See also setRasterOp() [p. 201] and RasterOp [Additional Functionality with Qt].

void QPainter::redirect (QPaintDevice * pdev, QPaintDevice * replacement) [static]

Redirects all paint command for a paint device *pdev* to another paint device *replacement*, unless *replacement* is 0. If *replacement* is 0, the redirection for *pdev* is removed.

Mostly, you can get better results with less work by calling QPixmap::grabWidget() or QPixmap::grapWindow().

void QPainter::resetXForm ()

Resets any transformations that were made using translate(), scale(), shear(), rotate(), setWorldMatrix(), setViewport() and setWindow()

See also worldMatrix() [p. 204], viewport() [p. 204] and window() [p. 204].

void QPainter::restore ()

Restores the current painter state (pops a saved state off the stack).

See also save() [p. 198].

Example: aclock/aclock.cpp.

void QPainter::restoreWorldMatrix ()

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

We recommend using restore() instead.

void QPainter::rotate (double a)

Rotates the coordinate system *a* degrees counterclockwise.

See also translate() [p. 204], scale() [p. 199], shear() [p. 203], resetXForm() [p. 198], setWorldMatrix() [p. 203] and xForm() [p. 205].

Examples: aclock/aclock.cpp, t10/cannon.cpp and t9/cannon.cpp.

void QPainter::save ()

Saves the current painter state (pushes the state onto a stack). A save() must be followed by a corresponding restore(). end() unwinds the stack.

See also restore() [p. 198].

Example: aclock/aclock.cpp.

void QPainter::saveWorldMatrix ()

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

We recommend using save() instead.

void QPainter::scale (double sx, double sy)

Scales the coordinate system by *(sx, sy)*.

See also translate() [p. 204], shear() [p. 203], rotate() [p. 198], resetXForm() [p. 198], setWorldMatrix() [p. 203] and xForm() [p. 205].

Example: xform/xform.cpp.

void QPainter::setBackgroundColor (const QColor & c)

Sets the background color of the painter to *c*.

The background color is the color that is filled in when drawing opaque text, stippled lines and bitmaps. The background color has no effect in transparent background mode (which is the default).

See also backgroundColor() [p. 184], setBackgroundMode() [p. 199] and BackgroundMode [Additional Functionality with Qt].

void QPainter::setBackgroundMode (BGMode m)

Sets the background mode of the painter to *m*, which must be one of TransparentMode (the default) and Opaque-Mode.

Transparent mode draws stippled lines and text without setting the background pixels. Opaque mode fills these space with the current background color.

Note that in order to draw a bitmap or pixmap transparently, you must use QPixmap::setMask().

See also backgroundMode() [p. 184] and setBackgroundColor() [p. 199].

Example: picture/picture.cpp.

void QPainter::setBrush (BrushStyle style)

Sets a new painter brush with black color and the specified *style*.

See also brush() [p. 186] and QBrush [p. 17].

Examples: aclock/aclock.cpp, drawdemo/drawdemo.cpp, picture/picture.cpp, t10/cannon.cpp, t9/cannon.cpp, themes/wood.cpp and tooltip/tooltip.cpp.

void QPainter::setBrush (const QBrush & brush)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets a new painter brush.

The *brush* defines how to fill shapes.

See also brush() [p. 186].

void QPainter::setBrush (const QColor & color)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets a new painter brush with the style SolidPattern and the specified *color*.

See also brush() [p. 186] and QBrush [p. 17].

void QPainter::setBrushOrigin (int x, int y)

Sets the brush origin to *(x, y)*.

The brush origin specifies the (0, 0) coordinate of the painter's brush. This setting only applies to pattern brushes and pixmap brushes.

See also brushOrigin() [p. 186].

void QPainter::setBrushOrigin (const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the brush origin to point *p*.

void QPainter::setClipRect (int x, int y, int w, int h, CoordinateMode m = CoordDevice)

Sets the clip region to the rectangle *x, y, w, h* and enables clipping. The clip mode is set to *m*.

Note that the clip region is given in physical device coordinates and *not* subject to any coordinate transformation if *m* is equal to CoordDevice (the default). If *m* equals CoordPainter the returned region is in model coordinates.

See also setClipRegion() [p. 200], clipRegion() [p. 187], setClipping() [p. 200] and QPainter::CoordinateMode [p. 183].

Examples: grapher/grapher.cpp, progress/progress.cpp, qtimage/qtimage.cpp, showimg/showimg.cpp, splitter/splitter.cpp and trivial/trivial.cpp.

void QPainter::setClipRect (const QRect & r, CoordinateMode m = CoordDevice)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the clip region to the rectangle *r* and enables clipping. The clip mode is set to *m*.

void QPainter::setClipRegion (const QRegion & rgn, CoordinateMode m = CoordDevice)

Sets the clip region to *rgn* and enables clipping. The clip mode is set to *m*.

Note that the clip region is given in physical device coordinates and *not* subject to any coordinate transformation.

See also setClipRect() [p. 200], clipRegion() [p. 187] and setClipping() [p. 200].

Examples: qfd/fontdisplayer.cpp and themes/wood.cpp.

void QPainter::setClipping (bool enable)

Enables clipping if *enable* is TRUE, or disables clipping if *enable* is FALSE.

See also hasClipping() [p. 196], setClipRect() [p. 200] and setClipRegion() [p. 200].

Example: themes/wood.cpp.

void QPainter::setFont (const QFont & font)

Sets a new painter font to *font*.

This font is used by subsequent drawText() functions. The text color is the same as the pen color.

See also font() [p. 196] and drawText() [p. 192].

Examples: application/application.cpp, drawdemo/drawdemo.cpp, grapher/grapher.cpp, hello/hello.cpp, picture/picture.cpp, t13/cannon.cpp and xform/xform.cpp.

void QPainter::setPen (const QPen & pen)

Sets a new painter pen.

The *pen* defines how to draw lines and outlines, and it also defines the text color.

See also pen() [p. 197].

Examples: desktop/desktop.cpp, drawdemo/drawdemo.cpp, progress/progress.cpp, t10/cannon.cpp, t9/cannon.cpp, themes/metal.cpp and themes/wood.cpp.

void QPainter::setPen (PenStyle style)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets a new painter pen to have style *style*, width 0 and black color.

See also pen() [p. 197] and QPen [p. 215].

void QPainter::setPen (const QColor & color)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets a new painter pen with style SolidLine, width 0 and the specified *color*.

See also pen() [p. 197] and QPen [p. 215].

void QPainter::setRasterOp (RasterOp r)

Sets the raster operation to *r*. The default is CopyROP.

See also rasterOp() [p. 198].

void QPainter::setTabArray (int * ta)

Sets the tab stop array to *ta*. This puts tab stops at *ta[0]*, *ta[1]* and so on. The array is null-terminated.

If both a tab array and a tab top size is set, the tab array wins.

See also tabArray() [p. 204], setTabStops() [p. 202], drawText() [p. 192] and fontMetrics() [p. 196].

void QPainter::setTabStops (int ts)

Set the tab stop width to *ts*, i.e. locates tab stops at *ts*, 2*ts, 3*ts and so on.

Tab stops are used when drawing formatted text with ExpandTabs set. This fixed tab stop value is used only if no tab array is set (which is the default case).

See also tabStops() [p. 204], setTabArray() [p. 201], drawText() [p. 192] and fontMetrics() [p. 196].

void QPainter::setViewXForm (bool enable)

Enables view transformations if *enable* is TRUE, or disables view transformations if *enable* is FALSE.

See also hasViewXForm() [p. 196], setWindow() [p. 202], setViewport() [p. 202], setWorldMatrix() [p. 203], setWorldXForm() [p. 203] and xForm() [p. 205].

void QPainter::setViewport (int x, int y, int w, int h)

Sets the viewport rectangle view transformation for the painter and enables view transformation.

The viewport rectangle is part of the view transformation. The viewport specifies the device coordinate system and is specified by the *x, y, w* width and *h* height parameters. Its sister, the window(), specifies the logical coordinate system.

The default viewport rectangle is the same as the device's rectangle. See the Coordinate System Overview [p. 4] for an overview of coordinate transformation.

See also viewport() [p. 204], setWindow() [p. 202], setViewXForm() [p. 202], setWorldMatrix() [p. 203], setWorldXForm() [p. 203] and xForm() [p. 205].

Example: aclock/aclock.cpp.

void QPainter::setViewport (const QRect & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the painter's viewport to rectangle *r*.

void QPainter::setWindow (int x, int y, int w, int h)

Sets the window rectangle view transformation for the painter and enables view transformation.

The window rectangle is part of the view transformation. The window specifies the logical coordinate system and is specified by the *x, y, w* width and *h* height parameters. Its sister, the viewport(), specifies the device coordinate system.

The default window rectangle is the same as the device's rectangle. See the Coordinate System Overview [p. 4] for an overview of coordinate transformation.

See also window() [p. 204], setViewport() [p. 202], setViewXForm() [p. 202], setWorldMatrix() [p. 203] and setWorldXForm() [p. 203].

Examples: aclock/aclock.cpp and drawdemo/drawdemo.cpp.

void QPainter::setWindow (const QRect & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the painter's window to rectangle *r*.

void QPainter::setWorldMatrix (const QWMatrix & m, bool combine = FALSE)

Sets the world transformation matrix to *m* and enables world transformation.

If *combine* is TRUE, then *m* is combined with the current transformation matrix, otherwise *m* replaces the current transformation matrix.

If *m* is the identity matrix and *combine* is FALSE, this function calls setWorldXForm(FALSE). (The identity matrix is the matrix where QWMatrix::m11() and QWMatrix::m22() are 1.0 and the rest are 0.0.)

World transformations are applied after the view transformations (i.e. window and viewport).

The following functions can transform the coordinate system without using a QWMatrix:

- translate()
- scale()
- shear()
- rotate()

They operate on the painter's worldMatrix() and are implemented like this:

```
void QPainter::rotate( double a )
{
    QWMatrix m;
    m.rotate( a );
    setWorldMatrix( m, TRUE );
}
```

Note that you should always use *combine* when you are drawing into a QPicture. Otherwise it may not be possible to replay the picture with additional transformations. Using translate(), scale(), etc. is safe.

For a brief overview of coordinate transformation, see the Coordinate System Overview.

See also worldMatrix() [p. 204], setWorldXForm() [p. 203], setWindow() [p. 202], setViewport() [p. 202], setViewXForm() [p. 202], xForm() [p. 205] and QWMatrix [p. 291].

Examples: drawdemo/drawdemo.cpp and xform/xform.cpp.

void QPainter::setWorldXForm (bool enable)

Enables world transformations if *enable* is TRUE, or disables world transformations if *enable* is FALSE. The world transformation matrix is not changed.

See also setWorldMatrix() [p. 203], setWindow() [p. 202], setViewport() [p. 202], setViewXForm() [p. 202] and xForm() [p. 205].

void QPainter::shear (double sh, double sv)

Shears the coordinate system by *(sh, sv)*.

See also translate() [p. 204], scale() [p. 199], rotate() [p. 198], resetXForm() [p. 198], setWorldMatrix() [p. 203] and xForm() [p. 205].

int * QPainter::tabArray () const

Returns the currently set tab stop array.

See also setTabArray() [p. 201].

int QPainter::tabStops () const

Returns the tab stop setting.

See also setTabStops() [p. 202].

void QPainter::translate (double dx, double dy)

Translates the coordinate system by *(dx, dy)*. After this call, *(dx, dy)* is added to points.

For example, the following code draws the same point twice:

```
void MyWidget::paintEvent()
{
    QPainter paint( this );

    paint.drawPoint( 0, 0 );

    paint.translate( 100.0, 40.0 );
    paint.drawPoint( -100, -40 );
}
```

See also scale() [p. 199], shear() [p. 203], rotate() [p. 198], resetXForm() [p. 198], setWorldMatrix() [p. 203] and xForm() [p. 205].

Examples: helpviewer/helpwindow.cpp, t10/cannon.cpp, t9/cannon.cpp, themes/metal.cpp, themes/wood.cpp and xform/xform.cpp.

QRect QPainter::viewport () const

Returns the viewport rectangle.

See also setViewport() [p. 202] and setViewXForm() [p. 202].

Example: aclock/aclock.cpp.

QRect QPainter::window () const

Returns the window rectangle.

See also setWindow() [p. 202] and setViewXForm() [p. 202].

const QWMatrix & QPainter::worldMatrix () const

Returns the world transformation matrix.

See also setWorldMatrix() [p. 203].

QPoint QPainter::xForm (const QPoint & pv) const

Returns the point *pv* transformed from model coordinates to device coordinates.

See also xFormDev() [p. 205] and QWMatrix::map() [p. 294].

QRect QPainter::xForm (const QRect & rv) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the rectangle *rv* transformed from model coordinates to device coordinates.

If world transformation is enabled and rotation or shearing has been specified, then the bounding rectangle is returned.

See also xFormDev() [p. 205] and QWMatrix::map() [p. 294].

QPointArray QPainter::xForm (const QPointArray & av) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point array *av* transformed from model coordinates to device coordinates.

See also xFormDev() [p. 205] and QWMatrix::map() [p. 294].

QPointArray QPainter::xForm (const QPointArray & av, int index, int npoints) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point array *av* transformed from model coordinates to device coordinates. The *index* is the first point in the array and *npoints* denotes the number of points to be transformed. If *npoints* is negative, all points from *av[index]* until the last point in the array are transformed.

The returned point array consists of the number of points that were transformed.

Example:

```
QPointArray a(10);
QPointArray b;
b = painter.xForm(a, 2, 4);  // b.size() == 4
b = painter.xForm(a, 2, -1); // b.size() == 8
```

See also xFormDev() [p. 205] and QWMatrix::map() [p. 294].

QRect QPainter::xFormDev (const QRect & rd) const

Returns the rectangle *rd* transformed from device coordinates to model coordinates.

If world transformation is enabled and rotation or shearing is used, then the bounding rectangle is returned.

See also xForm() [p. 205] and QWMatrix::map() [p. 294].

QPoint QPainter::xFormDev (const QPoint & pd) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point *pd* transformed from device coordinates to model coordinates.

See also xForm() [p. 205] and QWMatrix::map() [p. 294].

QPointArray QPainter::xFormDev (const QPointArray & ad) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point array *ad* transformed from device coordinates to model coordinates.

See also xForm() [p. 205] and QWMatrix::map() [p. 294].

QPointArray QPainter::xFormDev (const QPointArray & ad, int index, int npoints) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point array *ad* transformed from device coordinates to model coordinates. The *index* is the first point in the array and *npoints* denotes the number of points to be transformed. If *npoints* is negative, all points from *ad[index]* until the last point in the array are transformed.

The returned point array consists of the number of points that were transformed.

Example:

```
QPointArray a(10);
QPointArray b;
b = painter.xFormDev(a, 1, 3);  // b.size() == 3
b = painter.xFormDev(a, 1, -1); // b.size() == 9
```

See also xForm() [p. 205] and QWMatrix::map() [p. 294].

Related Functions

void qDrawPlainRect (QPainter * p, int x, int y, int w, int h, const QColor & c, int lineWidth, const QBrush * fill)

```
#include <qdrawutil.h>
```

Draws a plain rectangle given by (*x*, *y*, *w*, *h*) using the painter *p*.

The color argument *c* specifies the line color.

The *lineWidth* argument specifies the line width.

The rectangle interior is filled with the *fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a plain rectangle, for example `QFrame::setFrameStyle( QFrame::Box | QFrame::Plain )`.

Warning: This function does not look at QWidget::style() or QApplication::style(). Use the drawing functions in QStyle to make widgets that follow the current GUI style.

See also qDrawShadeRect() [p. 207] and QStyle::drawPrimitive() [Events, Actions, Layouts and Styles with Qt].

void qDrawShadeLine (QPainter * p, int x1, int y1, int x2, int y2, const QColorGroup & g, bool sunken, int lineWidth, int midLineWidth)

\{#include }

Draws a horizontal (*y1* == *y2*) or vertical (*x1* == *x2*) shaded line using the painter *p*.

Nothing is drawn if *y1* != *y2* and *x1* != *x2* (i.e. the line is neither horizontal nor vertical).

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The line appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The *lineWidth* argument specifies the line width for each of the lines. It is not the total line width.

The *midLineWidth* argument specifies the width of a middle line drawn in the QColorGroup::mid() color.

If you want to use a QFrame widget instead, you can make it display a shaded line, for example `QFrame::setFrameStyle( QFrame::HLine | QFrame::Sunken )`.

Warning: This function does not look at QWidget::style() or QApplication::style(). Use the drawing functions in QStyle to make widgets that follow the current GUI style.

See also qDrawShadeRect() [p. 207], qDrawShadePanel() [p. 207] and QStyle::drawPrimitive() [Events, Actions, Layouts and Styles with Qt].

void qDrawShadePanel (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, int lineWidth, const QBrush * fill)

```
#include <qdrawutil.h>
```

Draws a shaded panel given by (*x*, *y*, *w*, *h*) using the painter *p*.

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The panel appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The *lineWidth* argument specifies the line width.

The panel interior is filled with the *fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded panel, for example `QFrame::setFrameStyle( QFrame::Panel | QFrame::Sunken )`.

Warning: This function does not look at QWidget::style() or QApplication::style(). Use the drawing functions in QStyle to make widgets that follow the current GUI style.

See also qDrawWinPanel() [p. 208], qDrawShadeLine() [p. 206], qDrawShadeRect() [p. 207] and QStyle::drawPrimitive() [Events, Actions, Layouts and Styles with Qt].

void qDrawShadeRect (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, int lineWidth, int midLineWidth, const QBrush * fill)

```
#include <qdrawutil.h>
```

Draws a shaded rectangle/box given by (*x*, *y*, *w*, *h*) using the painter *p*.

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The rectangle appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The *lineWidth* argument specifies the line width for each of the lines. It is not the total line width.

The *midLineWidth* argument specifies the width of a middle line drawn in the QColorGroup::mid() color.

The rectangle interior is filled with the *fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded rectangle, for example `QFrame::setFrameStyle( QFrame::Box | QFrame::Raised )`.

Warning: This function does not look at QWidget::style() or QApplication::style(). Use the drawing functions in

QStyle to make widgets that follow the current GUI style.

See also qDrawShadeLine() [p. 206], qDrawShadePanel() [p. 207], qDrawPlainRect() [p. 206], QStyle::drawItem() [Events, Actions, Layouts and Styles with Qt], QStyle::drawControl() [Events, Actions, Layouts and Styles with Qt] and QStyle::drawComplexControl() [Events, Actions, Layouts and Styles with Qt].

void qDrawWinButton (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, const QBrush * fill)

```
#include <qdrawutil.h>
```

Draws a Windows-style button given by (*x*, *y*, *w*, *h*) using the painter *p*.

The color group argument *g* specifies the shading colors (light, dark and middle colors).

The button appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The line width is 2 pixels.

The button interior is filled with the **fill* brush unless *fill* is null.

Warning: This function does not look at QWidget::style() or QApplication::style(). Use the drawing functions in QStyle to make widgets that follow the current GUI style.

See also qDrawWinPanel() [p. 208] and QStyle::drawControl() [Events, Actions, Layouts and Styles with Qt].

void qDrawWinPanel (QPainter * p, int x, int y, int w, int h, const QColorGroup & g, bool sunken, const QBrush * fill)

```
#include <qdrawutil.h>
```

Draws a Windows-style panel given by (*x*, *y*, *w*, *h*) using the painter *p*.

The color group argument *g* specifies the shading colors.

The panel appears sunken if *sunken* is TRUE, or raised if *sunken* is FALSE.

The line width is 2 pixels.

The button interior is filled with the *fill* brush unless *fill* is null.

If you want to use a QFrame widget instead, you can make it display a shaded panel, for example QFrame::setFrameStyle(QFrame::WinPanel | QFrame::Raised).

Warning: This function does not look at QWidget::style() or QApplication::style(). Use the drawing functions in QStyle to make widgets that follow the current GUI style.

See also qDrawShadePanel() [p. 207], qDrawWinButton() [p. 208] and QStyle::drawPrimitive() [Events, Actions, Layouts and Styles with Qt].

QPalette Class Reference

The QPalette class contains color groups for each widget state.

`#include <qpalette.h>`

Public Members

- **QPalette** ()
- QPalette (const QColor & button) *(obsolete)*
- **QPalette** (const QColor & button, const QColor & background)
- **QPalette** (const QColorGroup & active, const QColorGroup & disabled, const QColorGroup & inactive)
- **QPalette** (const QPalette & p)
- **~QPalette** ()
- QPalette & **operator=** (const QPalette & p)
- enum **ColorGroup** { Disabled, Active, Inactive, NColorGroups, Normal = Active }
- const QColor & **color** (ColorGroup gr, QColorGroup::ColorRole r) const
- const QBrush & **brush** (ColorGroup gr, QColorGroup::ColorRole r) const
- void **setColor** (ColorGroup gr, QColorGroup::ColorRole r, const QColor & c)
- void **setBrush** (ColorGroup gr, QColorGroup::ColorRole r, const QBrush & b)
- void **setColor** (QColorGroup::ColorRole r, const QColor & c)
- void **setBrush** (QColorGroup::ColorRole r, const QBrush & b)
- QPalette **copy** () const
- const QColorGroup & **active** () const
- const QColorGroup & **disabled** () const
- const QColorGroup & **inactive** () const
- const QColorGroup & **normal** () const
- void **setActive** (const QColorGroup & g)
- void **setDisabled** (const QColorGroup & g)
- void **setInactive** (const QColorGroup & g)
- void **setNormal** (const QColorGroup & cg)
- bool **operator==** (const QPalette & p) const
- bool **operator!=** (const QPalette & p) const
- bool **isCopyOf** (const QPalette & p)
- int **serialNumber** () const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QPalette & p)
- QDataStream & **operator>>** (QDataStream & s, QPalette & p)

Detailed Description

The QPalette class contains color groups for each widget state.

A palette consists of three color groups: *active, disabled*, and *inactive*. All widgets contain a palette, and all widgets in Qt use their palette to draw themselves. This makes the user interface easily configurable and easier to keep consistent.

If you create a new widget we strongly recommend that you use the colors in the palette rather than hard-coding specific colors.

The color groups:

- The active() group is used for the window that has keyboard focus.
- The inactive() group is used for other windows.
- The disabled() group is used for widgets (not windows) that are disabled for some reason.

Both active and inactive windows can contain disabled widgets. (Disabled widgets are often called *inaccessible* or *grayed out*.)

In Motif style, active() and inactive() look precisely the same. In Windows 2000 style and Macintosh Platinum style, the two styles look slightly different.

There are setActive(), setInactive(), and setDisabled() functions to modify the palette. (Qt also supports a normal() group; this is an obsolete alias for active(), supported for backwards compatibility.)

Colors and brushes can be set for particular roles in any of a palette's color groups with setColor() and setBrush().

You can copy a palette using the copy constructor and test to see if two palettes are *identical* using isCopyOf().

See also QApplication::setPalette() [Additional Functionality with Qt], QWidget::palette [Widgets with Qt], QColorGroup [p. 84], QColor [p. 74], Widget Appearance and Style, Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Type Documentation

QPalette::ColorGroup

- `QPalette::Disabled`
- `QPalette::Active`
- `QPalette::Inactive`
- `QPalette::NColorGroups`
- `QPalette::Normal` - synonym for Active

Member Function Documentation

QPalette::QPalette ()

Constructs a palette that consists of color groups with only black colors.

QPalette::QPalette (const QColor & button)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Constructs a palette from the *button* color. The other colors are automatically calculated, based on this color. Background will be the button color as well.

QPalette::QPalette (const QColor & button, const QColor & background)

Constructs a palette from a *button* color and a *background*. The other colors are automatically calculated, based on these colors.

QPalette::QPalette (const QColorGroup & active, const QColorGroup & disabled, const QColorGroup & inactive)

Constructs a palette that consists of the three color groups *active*, *disabled* and *inactive*. See QPalette for definitions of the color groups and QColorGroup::ColorRole for definitions of each color role in the three groups.

See also QColorGroup [p. 84] and QColorGroup::ColorRole [p. 85].

QPalette::QPalette (const QPalette & p)

Constructs a copy of *p*.

This constructor is fast (it uses copy-on-write).

QPalette::~QPalette ()

Destroys the palette.

const QColorGroup & QPalette::active () const

Returns the active color group of this palette.

See also QColorGroup [p. 84], setActive() [p. 213], inactive() [p. 212] and disabled() [p. 212].

Examples: themes/metal.cpp and themes/wood.cpp.

const QBrush & QPalette::brush (ColorGroup gr, QColorGroup::ColorRole r) const

Returns the brush in color group *gr*, used for color role *r*.

See also color() [p. 211], setBrush() [p. 213] and QColorGroup::ColorRole [p. 85].

const QColor & QPalette::color (ColorGroup gr, QColorGroup::ColorRole r) const

Returns the color in color group *gr*, used for color role *r*.

See also brush() [p. 211], setColor() [p. 213] and QColorGroup::ColorRole [p. 85].

QPalette QPalette::copy () const

Returns a deep copy of this palette. This is slower than the copy constructor and assignment operator and no longer offers any advantages.

const QColorGroup & QPalette::disabled () const

Returns the disabled color group of this palette.

See also QColorGroup [p. 84], setDisabled() [p. 213], active() [p. 211] and inactive() [p. 212].

Examples: themes/metal.cpp and themes/wood.cpp.

const QColorGroup & QPalette::inactive () const

Returns the inactive color group of this palette.

See also QColorGroup [p. 84], setInactive() [p. 213], active() [p. 211] and disabled() [p. 212].

bool QPalette::isCopyOf (const QPalette & p)

Returns TRUE if this palette and *p* are copies of each other, ie. one of them was created as a copy of the other and neither was subsequently modified. This is much stricter than equality.

See also operator=() [p. 212] and operator==() [p. 212].

const QColorGroup & QPalette::normal () const

Returns the active color group.

See also setActive() [p. 213] and active() [p. 211].

bool QPalette::operator!= (const QPalette & p) const

Returns TRUE (slowly) if this palette is different from *p*; otherwise returns FALSE (usually quickly).

QPalette & QPalette::operator= (const QPalette & p)

Assigns *p* to this palette and returns a reference to this palette.

This is fast (it uses copy-on-write).

See also copy() [p. 211].

bool QPalette::operator== (const QPalette & p) const

Returns TRUE (usually quickly) if this palette is equal to *p*; otherwise returns FALSE (slowly).

int QPalette::serialNumber () const

Returns a number that uniquely identifies this QPalette object. The serial number is intended for caching. Its value may not be used for anything other than equality testing.

Note that QPalette uses copy-on-write, and the serial number changes during the lazy copy operation (detach()), not during a shallow copy (copy constructor or assignment).

See also QPixmap [p. 225], QPixmapCache [p. 239] and QCache [Datastructures and String Handling with Qt].

void QPalette::setActive (const QColorGroup & g)

Sets the Active color group to *g*.

See also active() [p. 211], setDisabled() [p. 213], setInactive() [p. 213] and QColorGroup [p. 84].

void QPalette::setBrush (ColorGroup gr, QColorGroup::ColorRole r, const QBrush & b)

Sets the brush in color group *gr*, used for color role *r*, to *b*.

See also brush() [p. 211], setColor() [p. 213] and QColorGroup::ColorRole [p. 85].

void QPalette::setBrush (QColorGroup::ColorRole r, const QBrush & b)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the brush in for color role *r* in all three color groups to *b*.

See also brush() [p. 211], setColor() [p. 213], QColorGroup::ColorRole [p. 85], active() [p. 211], inactive() [p. 212] and disabled() [p. 212].

void QPalette::setColor (ColorGroup gr, QColorGroup::ColorRole r, const QColor & c)

Sets the brush in color group *gr*, used for color role *r*, to the solid color *c*.

See also setBrush() [p. 213], color() [p. 211] and QColorGroup::ColorRole [p. 85].

Example: themes/themes.cpp.

void QPalette::setColor (QColorGroup::ColorRole r, const QColor & c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the brush color used for color role *r* to color *c* in all three color groups.

See also color() [p. 211], setBrush() [p. 213] and QColorGroup::ColorRole [p. 85].

void QPalette::setDisabled (const QColorGroup & g)

Sets the Disabled color group to *g*.

See also disabled() [p. 212], setActive() [p. 213] and setInactive() [p. 213].

void QPalette::setInactive (const QColorGroup & g)

Sets the Inactive color group to *g*.

See also active() [p. 211], setDisabled() [p. 213], setActive() [p. 213] and QColorGroup [p. 84].

void QPalette::setNormal (const QColorGroup & cg)

Sets the active color group to *cg*.

See also setActive() [p. 213] and active() [p. 211].

Related Functions

QDataStream & operator<< (QDataStream & s, const QPalette & p)

Writes the palette, *p* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QPalette & p)

Reads a palette from the stream, *s* into the palette *p*, and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QPen Class Reference

The QPen class defines how a QPainter should draw lines and outlines of shapes.

`#include <qpen.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QPen** ()
- **QPen** (PenStyle style)
- **QPen** (const QColor & color, uint width = 0, PenStyle style = SolidLine)
- **QPen** (const QColor & cl, uint w, PenStyle s, PenCapStyle c, PenJoinStyle j)
- **QPen** (const QPen & p)
- **~QPen** ()
- QPen & **operator=** (const QPen & p)
- PenStyle **style** () const
- void **setStyle** (PenStyle s)
- uint **width** () const
- void **setWidth** (uint w)
- const QColor & **color** () const
- void **setColor** (const QColor & c)
- PenCapStyle **capStyle** () const
- void **setCapStyle** (PenCapStyle c)
- PenJoinStyle **joinStyle** () const
- void **setJoinStyle** (PenJoinStyle j)
- bool **operator==** (const QPen & p) const
- bool **operator!=** (const QPen & p) const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QPen & p)
- QDataStream & **operator>>** (QDataStream & s, QPen & p)

Detailed Description

The QPen class defines how a QPainter should draw lines and outlines of shapes.

A pen has a style, width, color, cap style and join style.

The pen style defines the line type. The default pen style is Qt::SolidLine. Setting the style to NoPen tells the painter to not draw lines or outlines.

When drawing 1 pixel wide diagonal lines you can either use a very fast algorithm (specified by a line width of 0, which is the default), or a slower but more accurate algorithm (specified by a line width of 1). For horizontal and vertical lines a line width of 0 is the same as a line width of 1. The cap and join style have no effect on 0-width lines.

The pen color defines the color of lines and text. The default line color is black. The QColor documentation lists predefined colors.

The cap style defines how the end points of lines are drawn. The join style defines how the joins between two lines are drawn when multiple connected lines are drawn (QPainter::drawPolyLine() etc.). The cap and join styles only apply to wide lines, i.e. when the width is 1 or greater.

Use the QBrush class to specify fill styles.

Example:

```
QPainter painter;
QPen     pen( red, 2 );            // red solid line, 2 pixels wide
painter.begin( &anyPaintDevice );  // paint something
painter.setPen( pen );             // set the red, wide pen
painter.drawRect( 40,30, 200,100 ); // draw a rectangle
painter.setPen( blue );            // set blue pen, 0 pixel width
painter.drawLine( 40,30, 240,130 ); // draw a diagonal in rectangle
painter.end();                     // painting done
```

See the Qt::PenStyle [Additional Functionality with Qt] enum type for a complete list of pen styles.

With reference to the end points of lines, for wide (non-0-width) pens it depends on the cap style whether the end point is drawn or not. QPainter will try to make sure that the end point is drawn for 0-width pens, but this cannot be absolutely guaranteed because the underlying drawing engine is free to use any (typically accelerated) algorithm for drawing 0-width lines. On all tested systems, however, the end point of at least all non-diagonal lines are drawn.

A pen's color(), width(), style(), capStyle() and joinStyle() can be set in the constructor or later with setColor(), setWidth(), setStyle(), setCapStyle() and setJoinStyle(). Pens may also be compared and streamed.
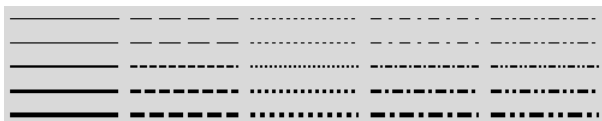
See also QPainter [p. 178], QPainter::setPen() [p. 201], Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QPen::QPen ()

Constructs a default black solid line pen with 0 width, which renders lines 1 pixel wide (fast diagonals).

QPen::QPen (PenStyle style)

Constructs a black pen with 0 width (fast diagonals) and style *style*.

See also setStyle() [p. 218].

QPen::QPen (const QColor & color, uint width = 0, PenStyle style = SolidLine)

Constructs a pen with the specified *color*, *width* and *style*.

See also setWidth() [p. 219], setStyle() [p. 218] and setColor() [p. 218].

QPen::QPen (const QColor & cl, uint w, PenStyle s, PenCapStyle c, PenJoinStyle j)

Constructs a pen with the specified color *cl* and width *w*. The pen style is set to *s*, the pen cap style to *c* and the pen join style to *j*.

A line width of 0 will produce a 1 pixel wide line using a fast algorithm for diagonals. A line width of 1 will also produce a 1 pixel wide line, but uses a slower more accurate algorithm for diagonals. For horizontal and vertical lines a line width of 0 is the same as a line width of 1. The cap and join style have no effect on 0-width lines.

See also setWidth() [p. 219], setStyle() [p. 218] and setColor() [p. 218].

QPen::QPen (const QPen & p)

Constructs a pen that is a copy of *p*.

QPen::~QPen ()

Destroys the pen.

PenCapStyle QPen::capStyle () const

Returns the pen's cap style.
See also setCapStyle() [p. 218].

const QColor & QPen::color () const

Returns the pen color.
See also setColor() [p. 218].
Example: scribble/scribble.h.

PenJoinStyle QPen::joinStyle () const

Returns the pen's join style.
See also setJoinStyle() [p. 218].

bool QPen::operator!= (const QPen & p) const

Returns TRUE if the pen is different from *p*; otherwise returns FALSE

Two pens are different if they have different styles, widths or colors.

See also operator==() [p. 218].

QPen & QPen::operator= (const QPen & p)

Assigns *p* to this pen and returns a reference to this pen.

bool QPen::operator== (const QPen & p) const

Returns TRUE if the pen is equal to *p*; otherwise returns FALSE

Two pens are equal if they have equal styles, widths and colors.

See also operator!=() [p. 217].

void QPen::setCapStyle (PenCapStyle c)

Sets the pen's cap style to *c*.

The default value is FlatCap. The cap style has no effect on 0-width pens.

Warning: On Windows 95/98 and Macintosh, the cap style setting has no effect. Wide lines are rendered as if the cap style was SquareCap.

See also capStyle() [p. 217].

Example: themes/wood.cpp.

void QPen::setColor (const QColor & c)

Sets the pen color to *c*.

See also color() [p. 217].

Examples: progress/progress.cpp and scribble/scribble.h.

void QPen::setJoinStyle (PenJoinStyle j)

Sets the pen's join style to *j*.

The default value is MiterJoin. The join style has no effect on 0-width pens.

Warning: On Windows 95/98 and Macintosh, the join style setting has no effect. Wide lines are rendered as if the join style was BevelJoin.

See also joinStyle() [p. 217].

Example: themes/wood.cpp.

void QPen::setStyle (PenStyle s)

Sets the pen style to *s*.

See the Qt::PenStyle [Additional Functionality with Qt] documentation for a list of all the styles.

Warning: On Windows 95/98 and Macintosh, the style setting (other than NoPen and SolidLine) has no effect for lines with width greater than 1.

See also style() [p. 219].

void QPen::setWidth (uint w)

Sets the pen width to *w*.

A line width of 0 will produce a 1 pixel wide line using a fast algorithm for diagonals. A line width of 1 will also produce a 1 pixel wide line, but uses a slower more accurate algorithm for diagonals. For horizontal and vertical lines a line width of 0 is the same as a line width of 1. The cap and join style have no effect on 0-width lines.

See also width() [p. 219].

Examples: progress/progress.cpp and scribble/scribble.h.

PenStyle QPen::style () const

Returns the pen style.

See also setStyle() [p. 218].

uint QPen::width () const

Returns the pen width.

See also setWidth() [p. 219].

Example: scribble/scribble.h.

Related Functions

QDataStream & operator<< (QDataStream & s, const QPen & p)

Writes the pen *p* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QPen & p)

Reads a pen from the stream *s* into *p* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QPicture Class Reference

The QPicture class is a paint device that records and replays QPainter commands.

`#include <qpicture.h>`

Inherits QPaintDevice [p. 170].

Public Members

- **QPicture** (int formatVersion = -1)
- **QPicture** (const QPicture & pic)
- **~QPicture** ()
- bool **isNull** () const
- uint **size** () const
- const char * **data** () const
- virtual void **setData** (const char * data, uint size)
- bool **play** (QPainter * painter)
- bool **load** (QIODevice * dev, const char * format = 0)
- bool **load** (const QString & fileName, const char * format = 0)
- bool **save** (QIODevice * dev, const char * format = 0)
- bool **save** (const QString & fileName, const char * format = 0)
- QRect **boundingRect** () const
- QPicture & **operator=** (const QPicture & p)

Protected Members

- virtual int **metric** (int m) const
- void **detach** ()
- QPicture **copy** () const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QPicture & r)
- QDataStream & **operator>>** (QDataStream & s, QPicture & r)

Detailed Description

The QPicture class is a paint device that records and replays QPainter commands.

A picture serializes painter commands to an IO device in a platform-independent format. A picture created under Windows can be read on a Sun SPARC.

Pictures are called meta-files on some platforms.

Qt pictures use a proprietary binary format. Unlike native picture (meta-file) formats on many window systems, Qt pictures have no limitations regarding their contents. Everything that can be painted can also be stored in a picture, e.g. fonts, pixmaps, regions, transformed graphics, etc.

QPicture is an implicitely shared class.

Example of how to record a picture:

```
QPicture  pic;
QPainter  p;
p.begin( &pic );              // paint in picture
p.drawEllipse( 10,20, 80,70 ); // draw an ellipse
p.end();                      // painting done
pic.save( "drawing.pic" );    // save picture
```

Example of how to replay a picture:

```
QPicture  pic;
pic.load( "drawing.pic" );    // load picture
QPainter  p;
p.begin( &myWidget );         // paint in myWidget
p.drawPicture( pic );         // draw the picture
p.end();                      // painting done
```

Pictures can also be drawn using play(). Some basic data about a picture is available, for example, size(), isNull() and boundingRect().

See also Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QPicture::QPicture (int formatVersion = -1)

Constructs an empty picture.

The *formatVersion* parameter may be used to *create* a QPicture that can be read by applications that are compiled with earlier versions of Qt.

- *formatVersion* == 1 is binary compatible with Qt 1.x and later.
- *formatVersion* == 2 is binary compatible with Qt 2.0.x and later.
- *formatVersion* == 3 is binary compatible with Qt 2.1.x and later.
- *formatVersion* == 4 is binary compatible with Qt 3.x.

Note that the default formatVersion is -1 which signifies the current release, i.e. for Qt 3.0 a formatVersion of 4 is the same as the default formatVersion of -1.

Reading pictures generated by earlier versions of Qt is supported and needs no special coding; the format is automatically detected.

QPicture::QPicture (const QPicture & pic)

Constructs a shallow copy of *pic*.

QPicture::~QPicture ()

Destroys the picture.

QRect QPicture::boundingRect () const

Returns the picture's bounding rectangle or an invalid rectangle if the picture contains no data.

QPicture QPicture::copy () const [protected]

Returns a deep copy of the picture.

const char * QPicture::data () const

Returns a pointer to the picture data. The pointer is only valid until the next non-const function is called on this picture. The returned pointer is null if the picture contains no data.

See also size() [p. 223] and isNull() [p. 222].

void QPicture::detach () [protected]

Detaches from shared picture data and makes sure that this picture is the only one referring to the data.

If multiple pictures share common data, this picture makes a copy of the data and detaches itself from the sharing mechanism. Nothing is done if there is just a single reference.

bool QPicture::isNull () const

Returns TRUE if the picture contains no data; otherwise returns FALSE.

bool QPicture::load (const QString & fileName, const char * format = 0)

Loads a picture from the file specified by *fileName* and returns TRUE if successful; otherwise returns FALSE.

By default, the file will be interpreted as being in the native QPicture format. Specifying the *format* string is optional and is only needed for importing picture data stored in a different format.

Currently, the only external format supported is the W3C SVG format which requires the Qt XML module. The corresponding *format* string is "svg".

See also save() [p. 223].

Examples: picture/picture.cpp and xform/xform.cpp.

bool QPicture::load (QIODevice * dev, const char * format = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

dev is the device to use for loading.

int QPicture::metric (int m) const [virtual protected]

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

A picture has the following hard-coded values: dpi=72, numcolors=16777216 and depth=24.

m is the metric to get.

Reimplemented from QPaintDevice [p. 172].

QPicture & QPicture::operator= (const QPicture & p)

Assigns a shallow copy of *p* to this picture and returns a reference to this picture.

bool QPicture::play (QPainter * painter)

Replays the picture using *painter*, and returns TRUE if successful or FALSE if the internal picture data is inconsistent.

This function does exactly the same as QPainter::drawPicture() with (x, y) = (0, 0).

bool QPicture::save (const QString & fileName, const char * format = 0)

Saves a picture to the file specified by *fileName* and returns TRUE if successful; otherwise returns FALSE.

Specifying the file *format* string is optional. It's not recommended unless you intend to export the picture data for the use in a 3rd party reader. By default the data will be saved in the native QPicture file format.

Currently, the only external format supported is the W3C SVG format which requires the Qt XML module. The corresponding *format* string is "svg".

See also load() [p. 222].

Example: picture/picture.cpp.

bool QPicture::save (QIODevice * dev, const char * format = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

dev is the device to use for loading.

void QPicture::setData (const char * data, uint size) [virtual]

Sets the picture data directly from *data* and *size*. This function copies the input data.

See also data() [p. 222] and size() [p. 223].

uint QPicture::size () const

Returns the size of the picture data.

See also data() [p. 222].

Related Functions

QDataStream & operator<< (QDataStream & s, const QPicture & r)

Writes picture, *r* to the stream *s* and returns a reference to the stream.

QDataStream & operator>> (QDataStream & s, QPicture & r)

Reads a picture from the stream *s* into picture *r* and returns a reference to the stream.

QPixmap Class Reference

The QPixmap class is an off-screen, pixel-based paint device.

`#include <qpixmap.h>`

Inherits QPaintDevice [p. 170] and Qt [Additional Functionality with Qt].

Inherited by QBitmap [p. 14] and QCanvasPixmap [p. 47].

Public Members

- enum **ColorMode** { Auto, Color, Mono }
- enum **Optimization** { DefaultOptim, NoOptim, MemoryOptim = NoOptim, NormalOptim, BestOptim }
- **QPixmap** ()
- **QPixmap** (const QImage & image)
- **QPixmap** (int w, int h, int depth = -1, Optimization optimization = DefaultOptim)
- **QPixmap** (const QSize & size, int depth = -1, Optimization optimization = DefaultOptim)
- **QPixmap** (const QString & fileName, const char * format = 0, ColorMode mode = Auto)
- **QPixmap** (const QString & fileName, const char * format, int conversion_flags)
- **QPixmap** (const char * xpm[])
- **QPixmap** (const QByteArray & img_data)
- **QPixmap** (const QPixmap & pixmap)
- **~QPixmap** ()
- QPixmap & **operator=** (const QPixmap & pixmap)
- QPixmap & **operator=** (const QImage & image)
- bool **isNull** () const
- int **width** () const
- int **height** () const
- QSize **size** () const
- QRect **rect** () const
- int **depth** () const
- void **fill** (const QColor & fillColor = Qt::white)
- void **fill** (const QWidget * widget, int xofs, int yofs)
- void **fill** (const QWidget * widget, const QPoint & ofs)
- void **resize** (int w, int h)
- void **resize** (const QSize & size)
- const QBitmap * **mask** () const
- void **setMask** (const QBitmap & newmask)
- bool **selfMask** () const
- QBitmap **createHeuristicMask** (bool clipTight = TRUE) const
- QPixmap **xForm** (const QWMatrix & matrix) const

- QImage **convertToImage** () const
- bool **convertFromImage** (const QImage & image, ColorMode mode = Auto)
- bool **convertFromImage** (const QImage & img, int conversion_flags)
- bool **load** (const QString & fileName, const char * format = 0, ColorMode mode = Auto)
- bool **load** (const QString & fileName, const char * format, int conversion_flags)
- bool **loadFromData** (const uchar * buf, uint len, const char * format = 0, ColorMode mode = Auto)
- bool **loadFromData** (const uchar * buf, uint len, const char * format, int conversion_flags)
- bool **loadFromData** (const QByteArray & buf, const char * format = 0, int conversion_flags = 0)
- bool **save** (const QString & fileName, const char * format, int quality = -1) const
- int **serialNumber** () const
- Optimization **optimization** () const
- void **setOptimization** (Optimization optimization)
- virtual void **detach** ()
- bool **isQBitmap** () const

Static Public Members

- int **defaultDepth** ()
- QPixmap **grabWindow** (WId window, int x = 0, int y = 0, int w = -1, int h = -1)
- QPixmap **grabWidget** (QWidget * widget, int x = 0, int y = 0, int w = -1, int h = -1)
- QWMatrix **trueMatrix** (const QWMatrix & matrix, int w, int h)
- const char * **imageFormat** (const QString & fileName)
- Optimization **defaultOptimization** ()
- void **setDefaultOptimization** (Optimization optimization)

Protected Members

- **QPixmap** (int w, int h, const uchar * bits, bool isXbitmap)
- virtual int **metric** (int m) const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QPixmap & pixmap)
- QDataStream & **operator>>** (QDataStream & s, QPixmap & pixmap)

Detailed Description

The QPixmap class is an off-screen, pixel-based paint device.

QPixmap is one of the two classes Qt provides for dealing with images; the other is QImage. QPixmap is designed and optimized for drawing; QImage is designed and optimized for I/O and for direct pixel access/manipulation. There are (slow) functions to convert between QImage and QPixmap: convertToImage() and convertFromImage().

One common use of the QPixmap class is to enable smooth updating of widgets. Whenever something complex needs to be drawn, you can use a pixmap to obtain flicker-free drawing, like this:

1. Create a pixmap with the same size as the widget.
2. Fill the pixmap with the widget background color.

3. Paint the pixmap.

4. bitBlt() the pixmap contents onto the widget.

Pixel data in a pixmap is internal and is managed by the underlying window system. Pixels can be accessed only through QPainter functions, through bitBlt(), and by converting the QPixmap to a QImage.

You can easily display a QPixmap on the screen using QLabel::setPixmap(), for example, all the QButton subclasses support pixmap use.

The QPixmap class uses lazy copying, so it is practical to pass QPixmap objects as arguments.

You can retrieve the width(), height(), depth() and size() of a pixmap. The enclosing rectangle is given by rect(). Pixmaps can be filled with fill() and resized with resize(). You can create and set a mask with createHeuristicMask() and setMask(). Use selfMask() to see if the pixmap is identical to its mask.

In addition to loading a pixmap from file using load() you can also loadFromData(). You can control optimization with setOptimization() and obtain a transformed version of the pixmap using xForm()

Note regarding Windows 95 and 98: on Windows 9x the system crashes if you create more than about 1000 pixmaps, independent of the size of the pixmaps or installed RAM. Windows NT and 2000 do not have this limitation.

Qt tries to work around the resource limitation. If you set the pixmap optimization to QPixmap::MemoryOptim and the width of your pixmap is less than or equal to 128 pixels, Qt stores the pixmap in a way that is very memory-efficient when there are many pixmaps.

If your application uses dozens or hundreds of pixmaps (for example on tool bar buttons and in popup menus), and you plan to run it on Windows 95 or Windows 98, we recommend using code like this:

```
QPixmap::setDefaultOptimization( QPixmap::MemoryOptim );
while ( ... ) {
  // load tool bar pixmaps etc.
  QPixmap *pixmap = new QPixmap(fileName);
}
QPixmap::setDefaultOptimization( QPixmap::NormalOptim );
```

See also QBitmap [p. 14], QImage [p. 129], QImageIO [p. 154], Shared Classes [Programming with Qt], Graphics Classes and Implicitly and Explicitly Shared Classes.

Member Type Documentation

QPixmap::ColorMode

This enum type defines the color modes that exist for converting QImage objects to QPixmap. The current values are:

- `QPixmap::Auto` - Select Color or Mono on a case-by-case basis.
- `QPixmap::Color` - Always create colored pixmaps.
- `QPixmap::Mono` - Always create bitmaps.

QPixmap::Optimization

QPixmap has the choice of optimizing for speed or memory in a few places; the best choice varies from pixmap to pixmap but can generally be derived heuristically. This enum type defines a number of optimization modes that you can set for any pixmap to tweak the speed/memory tradeoffs:

- `QPixmap::DefaultOptim` - Whatever QPixmap::defaultOptimization() returns. A pixmap with this optimization will have whatever the current default optimization is. If the default optimization is changed using setDefaultOptimization, then this will not effect any pixmaps that have already been created.
- `QPixmap::NoOptim` - No optimization (currently the same as MemoryOptim).
- `QPixmap::MemoryOptim` - Optimize for minimal memory use.
- `QPixmap::NormalOptim` - Optimize for typical usage. Often uses more memory than MemoryOptim, and is often faster.
- `QPixmap::BestOptim` - Optimize for pixmaps that are drawn very often and where performance is critical. Generally uses more memory than NormalOptim and may provide a little better speed.

We recommend using DefaultOptim.

Member Function Documentation

QPixmap::QPixmap ()

Constructs a null pixmap.

See also isNull() [p. 233].

QPixmap::QPixmap (const QImage & image)

Constructs a pixmap from the QImage *image*.

See also convertFromImage() [p. 230].

QPixmap::QPixmap (int w, int h, int depth = -1, Optimization optimization = DefaultOptim)

Constructs a pixmap with *w* width, *h* height and *depth* bits per pixel. The pixmap is optimized in accordance with the *optimization* value.

The contents of the pixmap is uninitialized.

The *depth* can be either 1 (monochrome) or the depth of the current video mode. If *depth* is negative, then the hardware depth of the current video mode will be used.

If either *w* or *h* is zero, a null pixmap is constructed.

See also isNull() [p. 233] and QPixmap::Optimization [p. 227].

QPixmap::QPixmap (const QSize & size, int depth = -1, Optimization optimization = DefaultOptim)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Constructs a pixmap of size *size*, *depth* bits per pixel, optimized in accordance with the *optimization* value.

QPixmap::QPixmap (const QString & fileName, const char * format = 0, ColorMode mode = Auto)

Constructs a pixmap from the file *fileName*. If the file does not exist or is of an unknown format, the pixmap becomes a null pixmap.

The *fileName*, *format* and *mode* parameters are passed on to load(). This means that the data in *fileName* is not compiled into the binary. If *fileName* contains a relative path (e.g. the filename only) the relevant file must be found relative to the runtime working directory.

See also QPixmap::ColorMode [p. 227], isNull() [p. 233], load() [p. 233], loadFromData() [p. 234], save() [p. 236] and imageFormat() [p. 233].

QPixmap::QPixmap (const QString & fileName, const char * format, int conversion_flags)

Constructs a pixmap from the file *fileName*. If the file does not exist or is of an unknown format, the pixmap becomes a null pixmap.

The *fileName*, *format* and *conversion_flags* parameters are passed on to load(). This means that the data in *fileName* is not compiled into the binary. If *fileName* contains a relative path (e.g. the filename only) the relevant file must be found relative to the runtime working directory.

If the image needs to be modified to fit in a lower-resolution result (e.g. converting from 32-bit to 8-bit), use the *conversion_flags* to specify how you'd prefer this to happen.

See also Qt::ImageConversionFlags [Additional Functionality with Qt], isNull() [p. 233], load() [p. 233], loadFromData() [p. 234], save() [p. 236] and imageFormat() [p. 233].

QPixmap::QPixmap (const char * xpm[])

Constructs a pixmap from *xpm*, which must be a valid XPM image.

Errors are silently ignored.

Note that it's possible to squeeze the XPM variable a little bit by using an unusual declaration:

```
static const char * const start_xpm[]={
    "16 15 8 1",
    "a c #cec6bd",
....
```

The extra `const` makes the entire definition read-only, which is slightly more efficient (for example, when the code is in a shared library) and ROMable when the application is to be stored in ROM.

In order to use that sort of declaration you must cast the variable back to `const char **` when you create the QPixmap.

QPixmap::QPixmap (const QByteArray & img_data)

Constructs a pixmaps by loading from *img_data*. The data can be in any image format supported by Qt.

See also loadFromData() [p. 234].

QPixmap::QPixmap (const QPixmap & pixmap)

Constructs a pixmap that is a copy of *pixmap*.

QPixmap::QPixmap (int w, int h, const uchar * bits, bool isXbitmap) [protected]

Constructs a monochrome pixmap, with width *w* and height *h*, that is initialized with the data in *bits*. The *isXbitmap* indicates whether the data is an X bitmap and defaults to FALSE. This constructor is protected and used by the QBitmap class.

QPixmap::~QPixmap ()

Destroys the pixmap.

bool QPixmap::convertFromImage (const QImage & img, int conversion_flags)

Converts image *img* and sets this pixmap. Returns TRUE if successful; otherwise returns FALSE.

The *conversion_flags* argument is a bitwise-OR of the Qt::ImageConversionFlags. Passing 0 for *conversion_flags* gives all the default options.

Note that even though a QPixmap with depth 1 behaves much like a QBitmap, isQBitmap() returns FALSE.

If a pixmap with depth 1 is painted with color0 and color1 and converted to an image, the pixels painted with color0 will produce pixel index 0 in the image and those painted with color1 will produce pixel index 1.

See also convertToImage() [p. 230], isQBitmap() [p. 233], QImage::convertDepth() [p. 135], defaultDepth() [p. 231] and QImage::hasAlphaBuffer() [p. 138].

Examples: qtimage/qtimage.cpp and themes/wood.cpp.

bool QPixmap::convertFromImage (const QImage & image, ColorMode mode = Auto)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Converts *image* and sets this pixmap using color mode *mode*. Returns TRUE if successful; otherwise returns FALSE.

See also QPixmap::ColorMode [p. 227].

QImage QPixmap::convertToImage () const

Converts the pixmap to a QImage. Returns a null image if the operation failed.

If the pixmap has 1-bit depth, the returned image will also be 1 bit deep. If the pixmap has 2- to 8-bit depth, the returned image has 8-bit depth. If the pixmap has greater than 8-bit depth, the returned image has 32-bit depth.

Note that for the moment, alpha masks on monochrome images are ignored.

See also convertFromImage() [p. 230].

Example: qmag/qmag.cpp.

QBitmap QPixmap::createHeuristicMask (bool clipTight = TRUE) const

Creates and returns a heuristic mask for this pixmap. It works by selecting a color from one of the corners and then chipping away pixels of that color, starting at all the edges.

The mask may not be perfect but it should be reasonable, so you can do things such as the following:

```
pm->setMask( pm->createHeuristicMask() );
```

This function is slow because it involves transformation to a QImage, non-trivial computations and a transformation back to a QBitmap.

If *clipTight* is TRUE the mask is just large enough to cover the pixels; otherwise, the mask is larger than the data pixels.

See also QImage::createHeuristicMask() [p. 137].

int QPixmap::defaultDepth () [static]

Returns the default pixmap depth, i.e., the depth a pixmap gets if -1 is specified.

See also depth() [p. 231].

Optimization QPixmap::defaultOptimization () [static]

Returns the default pixmap optimization setting.

See also setDefaultOptimization() [p. 236], setOptimization() [p. 237] and optimization() [p. 235].

int QPixmap::depth () const

Returns the depth of the image.

The pixmap depth is also called bits per pixel (bpp) or bit planes of a pixmap. A null pixmap has depth 0.

See also defaultDepth() [p. 231], isNull() [p. 233] and QImage::convertDepth() [p. 135].

void QPixmap::detach () [virtual]

This is a special-purpose function that detaches the pixmap from shared pixmap data.

A pixmap is automatically detached by Qt whenever its contents is about to change. This is done in all QPixmap member functions that modify the pixmap (fill(), resize(), convertFromImage(), load(), etc.), in bitBlt() for the destination pixmap and in QPainter::begin() on a pixmap.

It is possible to modify a pixmap without letting Qt know. You can first obtain the system-dependent handle() and then call system-specific functions (for instance, BitBlt under Windows) that modify the pixmap contents. In this case, you can call detach() to cut the pixmap loose from other pixmaps that share data with this one.

detach() returns immediately if there is just a single reference or if the pixmap has not been initialized yet.

void QPixmap::fill (const QColor & fillColor = Qt::white)

Fills the pixmap with the color *fillColor*.

Examples: aclock/aclock.cpp, desktop/desktop.cpp, grapher/grapher.cpp, hello/hello.cpp, t10/cannon.cpp, themes/metal.cpp and xform/xform.cpp.

void QPixmap::fill (const QWidget * widget, int xofs, int yofs)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Fills the pixmap with the *widget*'s background color or pixmap. If the background is empty, nothing is done. *xofs*, *yofs* is an offset in the widget.

void QPixmap::fill (const QWidget * widget, const QPoint & ofs)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Fills the pixmap with the *widget*'s background color or pixmap. If the background is empty, nothing is done.

The *ofs* point is an offset in the widget.

The point *ofs* is a point in the widget's coordinate system. The pixmap's top-left pixel will be mapped to the point *ofs* in the widget. This is significant if the widget has a background pixmap; otherwise the pixmap will simply be filled with the background color of the widget.

Example:

```
void CuteWidget::paintEvent( QPaintEvent *e )
{
QRect ur = e->rect();              // rectangle to update

QPixmap  pix( ur.size() );         // Pixmap for double-buffering

pix.fill( this, ur.topLeft() );  // fill with widget background

QPainter p( &pix );
p.translate( -ur.x(), -ur.y() ); // use widget coordinate system
                                    // when drawing on pixmap
//    ... draw on pixmap ...

p.end();

bitBlt( this, ur.topLeft(), &pix );
}
```

QPixmap QPixmap::grabWidget (QWidget * widget, int x = 0, int y = 0, int w = -1, int h = -1) [static]

Creates a pixmap and paints *widget* in it.

If the *widget* has any children, then they are also painted in the appropriate positions.

If you specify *x, y, w* or *h*, only the rectangle you specify is painted. The defaults are 0, 0 (top-left corner) and -1,-1 (which means the entire widget).

(If *w* is negative, the function copies everything to the right border of the window. If *h* is negative, the function copies everything to the bottom of the window.)

If *widget* is 0, or if the rectangle defined by *x, y,* the modified *w* and the modified *h* does not overlap the *widget->rect()*, this function will return a null QPixmap.

This function actually asks *widget* to paint itself (and its children to paint themselves). QPixmap::grabWindow() grabs pixels off the screen, which is a bit faster and picks up *exactly* what's on-screen. This function works by calling paintEvent() with painter redirection turned on. If there are overlaying windows, grabWindow() will see them, but not this function.

If there is overlap, it returns a pixmap of the size you want, containing a rendering of *widget*. If the rectangle you ask for is a superset of *widget*, the areas outside *widget* are covered with the widget's background.

See also grabWindow() [p. 232], QPainter::redirect() [p. 198] and QWidget::paintEvent() [Widgets with Qt].

QPixmap QPixmap::grabWindow (WId window, int x = 0, int y = 0, int w = -1, int h = -1) [static]

Grabs the contents of the window *window* and makes a pixmap out of it. Returns the pixmap.

The arguments *(x, y)* specify the offset in the window, whereas *(w, h)* specify the width and height of the area to be copied.

If *w* is negative, the function copies everything to the right border of the window. If *h* is negative, the function copies everything to the bottom of the window.

Note that grabWindows() grabs pixels from the screen, not from the window. If there is another window partially or entirely over the one you grab, you get pixels from the overlying window, too.

Note also that the mouse cursor is generally not grabbed.

The reason we use a window identifier and not a QWidget is to enable grabbing of windows that are not part of the application, window system frames, and so on.

Warning: Grabbing an area outside the screen is not safe in general. This depends on the underlying window system.

See also grabWidget() [p. 232].

Example: qmag/qmag.cpp.

int QPixmap::height () const

Returns the height of the pixmap.

See also width() [p. 237], size() [p. 237] and rect() [p. 235].

Examples: desktop/desktop.cpp, movies/main.cpp, qtimage/qtimage.cpp, scribble/scribble.cpp, scrollview/scrollview.cpp, t10/cannon.cpp and xform/xform.cpp.

const char * QPixmap::imageFormat (const QString & fileName) [static]

Returns a string that specifies the image format of the file *fileName*, or null if the file cannot be read or if the format cannot be recognized.

The QImageIO documentation lists the supported image formats.

See also load() [p. 233] and save() [p. 236].

bool QPixmap::isNull () const

Returns TRUE if this is a null pixmap; otherwise returns FALSE.

A null pixmap has zero width, zero height and no contents. You cannot draw in a null pixmap or bitBlt() anything to it.

Resizing an existing pixmap to (0, 0) makes a pixmap into a null pixmap.

See also resize() [p. 235].

Examples: qdir/qdir.cpp, qmag/qmag.cpp and scrollview/scrollview.cpp.

bool QPixmap::isQBitmap () const

Returns TRUE if this is a QBitmap; otherwise returns FALSE.

bool QPixmap::load (const QString & fileName, const char * format, int conversion_flags)

Loads a pixmap from the file *fileName* at runtime. Returns TRUE if successful, or FALSE if the pixmap could not be loaded.

If *format* is specified, the loader attempts to read the pixmap using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file's format.

See the convertFromImage() [p. 230] documentation for a description of the *conversion_flags* argument.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also loadFromData() [p. 234], save() [p. 236], imageFormat() [p. 233], QImage::load() [p. 139] and QImageIO [p. 154].

Examples: picture/picture.cpp, scrollview/scrollview.cpp and xform/xform.cpp.

bool QPixmap::load (const QString & fileName, const char * format = 0, ColorMode mode = Auto)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Loads a pixmap from the file *fileName* at runtime.

If *format* is specified, the loader attempts to read the pixmap using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file's format.

The *mode* is used to specify the color mode of the pixmap.

See also QPixmap::ColorMode [p. 227].

bool QPixmap::loadFromData (const uchar * buf, uint len, const char * format, int conversion_flags)

Loads a pixmap from the binary data in *buf* (*len* bytes). Returns TRUE if successful, or FALSE if the pixmap could not be loaded.

If *format* is specified, the loader attempts to read the pixmap using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file's format.

See the convertFromImage() [p. 230] documentation for a description of the *conversion_flags* argument.

The QImageIO documentation lists the supported image formats and explains how to add extra formats.

See also load() [p. 233], save() [p. 236], imageFormat() [p. 233], QImage::loadFromData() [p. 140] and QImageIO [p. 154].

bool QPixmap::loadFromData (const uchar * buf, uint len, const char * format = 0, ColorMode mode = Auto)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Loads a pixmap from the binary data in *buf* (*len* bytes) using color mode *mode*. Returns TRUE if successful, or FALSE if the pixmap could not be loaded.

If *format* is specified, the loader attempts to read the pixmap using the specified format. If *format* is not specified (default), the loader reads a few bytes from the header to guess the file's format.

See also QPixmap::ColorMode [p. 227].

bool QPixmap::loadFromData (const QByteArray & buf, const char * format = 0, int conversion_flags = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

const QBitmap * QPixmap::mask () const

Returns the mask bitmap, or null if no mask has been set.

See also setMask() [p. 236] and QBitmap [p. 14].

int QPixmap::metric (int m) const [virtual protected]

Internal implementation of the virtual QPaintDevice::metric() function.

Use the QPaintDeviceMetrics class instead.

m is the metric to get.

Reimplemented from QPaintDevice [p. 172].

QPixmap & QPixmap::operator= (const QPixmap & pixmap)

Assigns the pixmap *pixmap* to this pixmap and returns a reference to this pixmap.

QPixmap & QPixmap::operator= (const QImage & image)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Converts the image *image* to a pixmap that is assigned to this pixmap. Returns a reference to the pixmap.

See also convertFromImage() [p. 230].

Optimization QPixmap::optimization () const

Returns the optimization setting for this pixmap.

The default optimization setting is QPixmap::NormalOptim. You may change this settings in two ways:

- Call setDefaultOptimization() to set the default optimization for all new pixmaps.
- Call setOptimization() to set a the optimization for individual pixmaps.

See also setOptimization() [p. 237], setDefaultOptimization() [p. 236] and defaultOptimization() [p. 231].

QRect QPixmap::rect () const

Returns the enclosing rectangle (0,0,width(),height()) of the pixmap.

See also width() [p. 237], height() [p. 233] and size() [p. 237].

Example: xform/xform.cpp.

void QPixmap::resize (int w, int h)

Resizes the pixmap to *w* width and *h* height. If either *w* or *h* is 0, the pixmap becomes a null pixmap.

If both *w* and *h* are greater than 0, a valid pixmap is created. New pixels will be uninitialized (random) if the pixmap is expanded.

Examples: desktop/desktop.cpp and grapher/grapher.cpp.

void QPixmap::resize (const QSize & size)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Resizes the pixmap to size *size*.

bool QPixmap::save (const QString & fileName, const char * format, int quality = -1) const

Saves the pixmap to the file *fileName* using the image file format *format* and a quality factor *quality*. *quality* must be in the range [0,100] or -1. Specify 0 to obtain small compressed files, 100 for large uncompressed files, and -1 to use the default settings. Returns TRUE if successful, or FALSE if the pixmap could not be saved.

See also load() [p. 233], loadFromData() [p. 234], imageFormat() [p. 233], QImage::save() [p. 142] and QImageIO [p. 154].

Example: qmag/qmag.cpp.

bool QPixmap::selfMask () const

Returns TRUE if the pixmap's mask is identical to the pixmap itself; otherwise returns FALSE.

See also mask() [p. 235].

int QPixmap::serialNumber () const

Returns a number that uniquely identifies the contents of this QPixmap object. This means that multiple QPixmaps objects can have the same serial number as long as they refer to the same contents.

An example of where this is useful is for caching QPixmaps.

See also QPixmapCache [p. 239].

void QPixmap::setDefaultOptimization (Optimization optimization) [static]

Sets the default pixmap optimization.

All *new* pixmaps that are created will use this default optimization. You may also set optimization for individual pixmaps using the setOptimization() function.

The initial default *optimization* setting is `QPixmap::Normal`.

See also defaultOptimization() [p. 231], setOptimization() [p. 237] and optimization() [p. 235].

void QPixmap::setMask (const QBitmap & newmask)

Sets a mask bitmap.

The *newmask* bitmap defines the clip mask for this pixmap. Every pixel in *newmask* corresponds to a pixel in this pixmap. Pixel value 1 means opaque and pixel value 0 means transparent. The mask must have the same size as this pixmap.

Setting a null mask resets the mask.

See also mask() [p. 235], createHeuristicMask() [p. 230] and QBitmap [p. 14].

void QPixmap::setOptimization (Optimization optimization)

Sets pixmap drawing optimization for this pixmap.

The *optimization* setting affects pixmap operations, in particular drawing of transparent pixmaps (bitBlt() a pixmap with a mask set) and pixmap transformations (the xForm() function).

Pixmap optimization involves keeping intermediate results in a cache buffer and use the data in the cache to speed up bitBlt() and xForm(). The cost is more memory consumption, up to twice as much as an unoptimized pixmap.

Use the setDefaultOptimization() to change the default optimization for all new pixmaps.

See also optimization() [p. 235], setDefaultOptimization() [p. 236] and defaultOptimization() [p. 231].

Example: desktop/desktop.cpp.

QSize QPixmap::size () const

Returns the size of the pixmap.

See also width() [p. 237], height() [p. 233] and rect() [p. 235].

Examples: movies/main.cpp and qtimage/qtimage.cpp.

QWMatrix QPixmap::trueMatrix (const QWMatrix & matrix, int w, int h) [static]

Returns the actual matrix used for transforming a pixmap with *w* width and *h* height and matrix *matrix*.

When transforming a pixmap with xForm(), the transformation matrix is internally adjusted to compensate for unwanted translation, i.e. xForm() returns the smallest pixmap containing all transformed points of the original pixmap.

This function returns the modified matrix, which maps points correctly from the original pixmap into the new pixmap.

See also xForm() [p. 237] and QWMatrix [p. 291].

int QPixmap::width () const

Returns the width of the pixmap.

See also height() [p. 233], size() [p. 237] and rect() [p. 235].

Examples: desktop/desktop.cpp, movies/main.cpp, qtimage/qtimage.cpp, scribble/scribble.cpp, scrollview/scrollview.cpp and xform/xform.cpp.

QPixmap QPixmap::xForm (const QWMatrix & matrix) const

Returns a copy of the pixmap that is transformed using *matrix*. The original pixmap is not changed.

The transformation *matrix* is internally adjusted to compensate for unwanted translation, i.e. xForm() returns the smallest image that contains all the transformed points of the original image.

See also trueMatrix() [p. 237], QWMatrix [p. 291], QPainter::setWorldMatrix() [p. 203] and QImage::xForm() [p. 146].

Examples: desktop/desktop.cpp, fileiconview/qfileiconview.cpp, movies/main.cpp, qmag/qmag.cpp, qtimage/qtimage.cpp and xform/xform.cpp.

Related Functions

QDataStream & operator<< (QDataStream & s, const QPixmap & pixmap)

Writes the pixmap *pixmap* to the stream *s* as a PNG image.

See also QPixmap::save() [p. 236] and Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QPixmap & pixmap)

Reads a pixmap from the stream *s* into the pixmap *pixmap*.

See also QPixmap::load() [p. 233] and Format of the QDataStream operators [Input/Output and Networking with Qt].

QPixmapCache Class Reference

The QPixmapCache class provides an application-global cache for pixmaps.

```
#include <qpixmapcache.h>
```

Static Public Members

- int **cacheLimit** ()
- void **setCacheLimit** (int n)
- QPixmap * **find** (const QString & key)
- bool **find** (const QString & key, QPixmap & pm)
- bool insert (const QString & key, QPixmap * pm) *(obsolete)*
- bool **insert** (const QString & key, const QPixmap & pm)
- void **clear** ()

Detailed Description

The QPixmapCache class provides an application-global cache for pixmaps.

This class is a tool for optimized drawing with QPixmap. You can use it to store temporary pixmaps that are expensive to generate without using more storage space than cacheLimit(). Use insert() to insert pixmaps, find() to find them and clear() to empty the cache.

For example, QRadioButton has a non-trivial visual representation so we don't want to regenerate a pixmap whenever a radio button is displayed or changes state. In the function QRadioButton::drawButton(), we do not draw the radio button directly. Instead, we first check the global pixmap cache for a pixmap with the key "$qt_radio_nnn_", where nnn is a numerical value that specifies the the radio button state. If a pixmap is found, we bitBlt() it onto the widget and return. Otherwise, we create a new pixmap, draw the radio button in the pixmap, and finally insert the pixmap in the global pixmap cache, using the key above. The bitBlt() is 10 times faster than drawing the radio button. All radio buttons in the program share the cached pixmap since QPixmapCache is application-global.

QPixmapCache contains no member data, only static functions to access the global pixmap cache. It creates an internal QCache for caching the pixmaps.

The cache associates a pixmap with a string (key). If two pixmaps are inserted into the cache using equal keys, then the last pixmap will hide the first pixmap. The QDict and QCache classes do exactly the same.

The cache becomes full when the total size of all pixmaps in the cache exceeds cacheLimit(). The initial cache limit is 1024 KByte (1 MByte); it is changed with setCacheLimit(). A pixmap takes roughly width*height*depth/8 bytes of memory.

See the QCache [Datastructures and String Handling with Qt] documentation for more details about the cache mechanism.

See also Environment Classes, Graphics Classes and Image Processing Classes.

Member Function Documentation

int QPixmapCache::cacheLimit () [static]

Returns the cache limit (in kilobytes).

The default setting is 1024 kilobytes.

See also setCacheLimit() [p. 241].

void QPixmapCache::clear () [static]

Removes all pixmaps from the cache.

QPixmap * QPixmapCache::find (const QString & key) [static]

Returns the pixmap associated with the *key* in the cache, or null if there is no such pixmap.

Note: if valid, you should copy the pixmap immediately (this is quick). Subsequent insertions into the cache could cause the pointer to become invalid. For this reason, we recommend you use find(const QString&, QPixmap&) instead.

Example:

```
QPixmap* pp;
QPixmap p;
if ( (pp=QPixmapCache::find("my_previous_copy", pm)) ) {
    p = *pp;
} else {
    p.load("bigimage.png");
    QPixmapCache::insert("my_previous_copy", new QPixmap(p));
}
painter->drawPixmap(0, 0, p);
```

bool QPixmapCache::find (const QString & key, QPixmap & pm) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Looks for a cached pixmap associated with the *key* in the cache. If a pixmap is found, the function sets *pm* to that pixmap and returns TRUE. Otherwise, the function returns FALSE and does not change *pm*.

Example:

```
QPixmap p;
if ( !QPixmapCache::find("my_previous_copy", pm) ) {
    pm.load("bigimage.png");
    QPixmapCache::insert("my_previous_copy", pm);
}
painter->drawPixmap(0, 0, p);
```

bool QPixmapCache::insert (const QString & key, const QPixmap & pm) [static]

Inserts a copy of the pixmap *pm* associated with the *key* into the cache.

All pixmaps inserted by the Qt library have a key starting with "$qt". Use something else for your own pixmaps.

When a pixmap is inserted and the cache is about to exceed its limit, it removes pixmaps until there is enough room for the pixmap to be inserted.

The oldest pixmaps (least recently accessed in the cache) are deleted when more space is needed.

See also setCacheLimit() [p. 241].

bool QPixmapCache::insert (const QString & key, QPixmap * pm) [static]

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Inserts the pixmap *pm* associated with *key* into the cache. Returns TRUE if successful, or FALSE if the pixmap is too big for the cache.

Note: *pm* must be allocated on the heap (using `new`).

If this function returns FALSE, you must delete *pm* yourself.

If this function returns TRUE, do not use *pm* afterwards or keep references to it because any other insertions into the cache, whether from anywhere in the application or within Qt itself, could cause the pixmap to be discarded from the cache and the pointer to become invalid.

Due to these dangers, we strongly recommend that you use insert(const QString&, const QPixmap&) instead.

void QPixmapCache::setCacheLimit (int n) [static]

Sets the cache limit to *n* kilobytes.

The default setting is 1024 kilobytes.

See also cacheLimit() [p. 240].

QPoint Class Reference

The QPoint class defines a point in the plane.

```
#include <qpoint.h>
```

Public Members

- **QPoint** ()
- **QPoint** (int xpos, int ypos)
- bool **isNull** () const
- int **x** () const
- int **y** () const
- void **setX** (int x)
- void **setY** (int y)
- int **manhattanLength** () const
- QCOORD & **rx** ()
- QCOORD & **ry** ()
- QPoint & **operator+=** (const QPoint & p)
- QPoint & **operator-=** (const QPoint & p)
- QPoint & **operator*=** (int c)
- QPoint & **operator*=** (double c)
- QPoint & **operator/=** (int c)
- QPoint & **operator/=** (double c)

Related Functions

- bool **operator==** (const QPoint & p1, const QPoint & p2)
- bool **operator!=** (const QPoint & p1, const QPoint & p2)
- const QPoint **operator+** (const QPoint & p1, const QPoint & p2)
- const QPoint **operator-** (const QPoint & p1, const QPoint & p2)
- const QPoint **operator*** (const QPoint & p, int c)
- const QPoint **operator*** (int c, const QPoint & p)
- const QPoint **operator*** (const QPoint & p, double c)
- const QPoint **operator*** (double c, const QPoint & p)
- const QPoint **operator-** (const QPoint & p)
- const QPoint **operator/** (const QPoint & p, int c)
- const QPoint **operator/** (const QPoint & p, double c)
- QDataStream & **operator<<** (QDataStream & s, const QPoint & p)
- QDataStream & **operator>>** (QDataStream & s, QPoint & p)

Detailed Description

The QPoint class defines a point in the plane.

A point is specified by an x coordinate and a y coordinate.

The coordinate type is `QCOORD` (a 32-bit integer). The minimum value of `QCOORD` is `QCOORD_MIN` (-2147483648) and the maximum value is `QCOORD_MAX` (2147483647).

The coordinates are accessed by the functions x() and y(); they can be set by setX() and setY() or by the reference functions rx() and ry().

Given a point *p*, the following statements are all equivalent:

```
p.setX( p.x() + 1 );
p += QPoint( 1, 0 );
p.rx()++;
```

A QPoint can also be used as a vector. Addition and subtraction of QPoints are defined as for vectors (each component is added separately). You can divide or multiply a QPoint by an `int` or a `double`. The function manhattanLength() gives an inexpensive approximation of the length of the QPoint interpreted as a vector.

Example:

```
//QPoint oldPos is defined somewhere else
MyWidget::mouseMoveEvent( QMouseEvent *e )
{
    QPoint vector = e->pos() - oldPos;
    if ( vector.manhattanLength() > 3 )
        ... //mouse has moved more than 3 pixels since oldPos
}
```

QPoints can be compared for equality or inequality, and they can be written to and read from a QStream.

See also QSize [p. 285], QRect [p. 266], Graphics Classes and Image Processing Classes.

Member Function Documentation

QPoint::QPoint ()

Constructs a point with coordinates (0,0) (isNull() returns TRUE).

QPoint::QPoint (int xpos, int ypos)

Constructs a point with the x value *xpos* and y value *ypos*.

bool QPoint::isNull () const

Returns TRUE if both the x value and the y value are 0; otherwise returns FALSE.

int QPoint::manhattanLength () const

Returns the sum of the absolute values of x() and y(), traditionally known as the "Manhattan length" of the vector from the origin to the point. The tradition arises because such distances apply to travelers who can only travel on a rectangular grid, like the streets of Manhattan.

This is a useful, and quick to calculate, approximation to the true length: sqrt(pow(x(),2)+pow(y(),2)).

QPoint & QPoint::operator*= (int c)

Multiplies both x and y with *c*, and returns a reference to this point.

Example:

```
QPoint p( -1, 4 );
p *= 2;                     // p becomes (-2,8)
```

QPoint & QPoint::operator*= (double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Multiplies both x and y with *c*, and returns a reference to this point.

Example:

```
QPoint p( -1, 4 );
p *= 2.5;                   // p becomes (-3,10)
```

Note that the result is truncated.

QPoint & QPoint::operator+= (const QPoint & p)

Adds *p* to the point and returns a reference to this point.

Example:

```
QPoint p(  3, 7 );
QPoint q( -1, 4 );
p += q;                     // p becomes (2,11)
```

QPoint & QPoint::operator-= (const QPoint & p)

Subtracts *p* from the point and returns a reference to this point.

Example:

```
QPoint p(  3, 7 );
QPoint q( -1, 4 );
p -= q;                     // p becomes (4,3)
```

QPoint & QPoint::operator/= (int c)

Divides both x and y by *c*, and returns a reference to this point.

Example:

```
QPoint p( -2, 8 );
p /= 2;                     // p becomes (-1,4)
```

QPoint & QPoint::operator/= (double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Divides both x and y by *c*, and returns a reference to this point.

Example:

```
QPoint p( -3, 10 );
p /= 2.5;                    // p becomes (-1,4)
```

Note that the result is truncated because points are held as integers.

QCOORD & QPoint::rx ()

Returns a reference to the x coordinate of the point.

Using a reference makes it possible to directly manipulate x.

Example:

```
QPoint p( 1, 2 );
p.rx()--;                    // p becomes (0,2)
```

See also ry() [p. 245].

QCOORD & QPoint::ry ()

Returns a reference to the y coordinate of the point.

Using a reference makes it possible to directly manipulate y.

Example:

```
QPoint p( 1, 2 );
p.ry()++;                    // p becomes (1,3)
```

See also rx() [p. 245].

void QPoint::setX (int x)

Sets the x coordinate of the point to *x*.

See also x() [p. 246] and setY() [p. 245].

Example: t14/cannon.cpp.

void QPoint::setY (int y)

Sets the y coordinate of the point to *y*.

See also y() [p. 246] and setX() [p. 245].

Example: t14/cannon.cpp.

int QPoint::x () const

Returns the x coordinate of the point.

See also setX() [p. 245] and y() [p. 246].

Examples: dirview/dirview.cpp, fileiconview/qfileiconview.cpp, life/life.cpp, t14/cannon.cpp and themes/wood.cpp.

int QPoint::y () const

Returns the y coordinate of the point.

See also setY() [p. 245] and x() [p. 246].

Examples: fileiconview/qfileiconview.cpp, life/life.cpp, t14/cannon.cpp and themes/wood.cpp.

Related Functions

bool operator!= (const QPoint & p1, const QPoint & p2)

Returns TRUE if *p1* and *p2* are not equal; otherwise returns FALSE.

const QPoint operator* (const QPoint & p, int c)

Returns the QPoint formed by multiplying both components of *p* by *c*.

const QPoint operator* (int c, const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the QPoint formed by multiplying both components of *p* by *c*.

const QPoint operator* (const QPoint & p, double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the QPoint formed by multiplying both components of *p* by *c*.

Note that the result is truncated because points are held as integers.

const QPoint operator* (double c, const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the QPoint formed by multiplying both components of *p* by *c*.

Note that the result is truncated because points are held as integers.

const QPoint operator+ (const QPoint & p1, const QPoint & p2)

Returns the sum of *p1* and *p2*; each component is added separately.

const QPoint operator- (const QPoint & p1, const QPoint & p2)

Returns *p2* subtracted from *p1*; each component is subtracted separately.

const QPoint operator- (const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the QPoint formed by changing the sign of both components of *p*, equivalent to `QPoint(0,0) - p`.

const QPoint operator/ (const QPoint & p, int c)

Returns the QPoint formed by dividing both components of *p* by *c*.

const QPoint operator/ (const QPoint & p, double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the QPoint formed by dividing both components of *p* by *c*.

Note that the result is truncated because points are held as integers.

QDataStream & operator<< (QDataStream & s, const QPoint & p)

Writes point *p* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

bool operator== (const QPoint & p1, const QPoint & p2)

Returns TRUE if *p1* and *p2* are equal; otherwise returns FALSE.

QDataStream & operator>> (QDataStream & s, QPoint & p)

Reads a QPoint from the stream *s* into point *p* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QPointArray Class Reference

The QPointArray class provides an array of points.

```
#include <qpointarray.h>
```

Inherits QMemArray [Datastructures and String Handling with Qt]<QPoint>.

Public Members

- **QPointArray** ()
- **~QPointArray** ()
- **QPointArray** (int size)
- **QPointArray** (const QPointArray & a)
- **QPointArray** (const QRect & r, bool closed = FALSE)
- **QPointArray** (int nPoints, const QCOORD * points)
- QPointArray & **operator=** (const QPointArray & a)
- QPointArray **copy** () const
- void **translate** (int dx, int dy)
- QRect **boundingRect** () const
- void **point** (uint index, int * x, int * y) const
- QPoint **point** (uint index) const
- void **setPoint** (uint index, int x, int y)
- void **setPoint** (uint i, const QPoint & p)
- bool **setPoints** (int nPoints, const QCOORD * points)
- bool **setPoints** (int nPoints, int firstx, int firsty, ...)
- bool **putPoints** (int index, int nPoints, const QCOORD * points)
- bool **putPoints** (int index, int nPoints, int firstx, int firsty, ...)
- bool **putPoints** (int index, int nPoints, const QPointArray & from, int fromIndex = 0)
- void **makeArc** (int x, int y, int w, int h, int a1, int a2)
- void **makeEllipse** (int x, int y, int w, int h)
- void **makeArc** (int x, int y, int w, int h, int a1, int a2, const QWMatrix & xf)
- QPointArray **cubicBezier** () const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QPointArray & a)
- QDataStream & **operator>>** (QDataStream & s, QPointArray & a)

Detailed Description

The QPointArray class provides an array of points.

The QPointArray is an array of QPoint objects. In addition to the functions provided by QMemArray, QPointArray provides some point-specific functions.

For convenient reading and writing of the point data use setPoints(), putPoints(), point(), and setPoint().

For geometry operations: boundingRect() and translate(). There is also a QWMatrix::map() function for more general transformation of QPointArrays. You can also create arcs and ellipses with makeArc() and makeEllipse().

Among others, QPointArray is used by QPainter::drawLineSegments(), QPainter::drawPolyline(), QPainter::drawPolygon() and QPainter::drawCubicBezier().

Note that because this class is a QMemArray, copying an array and modifying the copy modifies the original as well, i.e. a shallow copy. If you need a deep copy use copy() or detach(), for example:

```
void drawGiraffe( const QPointArray & r, QPainter * p )
{
    QPointArray tmp = r;
    tmp.detach();
    // some code that modifies tmp
    p->drawPoints( tmp );
}
```

If you forget the tmp.detach(), the const array will be modified.

See also QPainter [p. 178], QWMatrix [p. 291], QMemArray [Datastructures and String Handling with Qt], Graphics Classes, Image Processing Classes and Implicitly and Explicitly Shared Classes.

Member Function Documentation

QPointArray::QPointArray ()

Constructs a null point array.

See also isNull() [Datastructures and String Handling with Qt].

QPointArray::QPointArray (int size)

Constructs a point array with room for *size* points. Makes a null array if *size* == 0.

See also resize() [Datastructures and String Handling with Qt] and isNull() [Datastructures and String Handling with Qt].

QPointArray::QPointArray (const QPointArray & a)

Constructs a shallow copy of the point array *a*.

See also copy() [p. 250].

QPointArray::QPointArray (const QRect & r, bool closed = FALSE)

Constructs a point array from the rectangle *r*.

If *closed* is FALSE, then the point array just contains the following four points in the listed order: r.topLeft(), r.topRight(), r.bottomRight() and r.bottomLeft().

If *closed* is TRUE, then a fifth point is set to r.topLeft().

QPointArray::QPointArray (int nPoints, const QCOORD * points)

Constructs a point array with *nPoints* points, taken from the *points* array.

Equivalent to setPoints(nPoints, points).

QPointArray::~QPointArray ()

Destroys the point array.

QRect QPointArray::boundingRect () const

Returns the bounding rectangle of the points in the array, or QRect(0,0,0,0) if the array is empty.

QPointArray QPointArray::copy () const

Creates a deep copy of the array.

QPointArray QPointArray::cubicBezier () const

Returns the Bezier points for the four control points in this array.

void QPointArray::makeArc (int x, int y, int w, int h, int a1, int a2)

Sets the points of the array to those describing an arc of an ellipse with size *w* by *h* and position (*x*, *y*), starting from angle *a1* and spanning *a2*. The resulting array has sufficient resolution for pixel accuracy (see the overloaded function which takes an additional QWMatrix parameter).

Angles are specified in 16ths of a degree, i.e. a full circle equals 5760 (16*360). Positive values mean counter-clockwise, whereas negative values mean a clockwise direction. Zero degrees is at the 3 o'clock position.

See the angle diagram.

void QPointArray::makeArc (int x, int y, int w, int h, int a1, int a2, const QWMatrix & xf)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the points of the array to those describing an arc of an ellipse with width *w* and height *h* and position (*x*, *y*), starting from angle *a1*, spanning angle *a2* and transformed by the matrix *xf*. The resulting array has sufficient resolution for pixel accuracy.

Angles are specified in 16ths of a degree, i.e. a full circle equals 5760 (16*360). Positive values mean counter-clockwise, whereas negative values mean a clockwise direction. Zero degrees is at the 3 o'clock position.

See the angle diagram.

void QPointArray::makeEllipse (int x, int y, int w, int h)

Sets the points of the array to those describing an ellipse with size *w* by *h* and position (*x, y*).

The returned array has sufficient resolution for use as pixels.

QPointArray & QPointArray::operator= (const QPointArray & a)

Assigns a shallow copy of *a* to this point array and returns a reference to this point array.

Equivalent to assign(a).

See also copy() [p. 250].

void QPointArray::point (uint index, int * x, int * y) const

Reads the coordinates of the point at position *index* within the array and writes them into **x* and **y*.

QPoint QPointArray::point (uint index) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point at position *index* within the array.

bool QPointArray::putPoints (int index, int nPoints, int firstx, int firsty, ...)

Copies *nPoints* points from the variable argument list into this point array from position *index*, and resizes the point array if `index+nPoints` exceeds the size of the array.

Returns TRUE if successful, or FALSE if the array could not be resized (typically due to lack of memory).

The example code creates an array with three points (1,2), (3,4) and (5,6), by expanding the array from 1 to 3 points:

```
QPointArray a( 1 );
a[0] = QPoint( 1, 2 );
a.putPoints( 1, 2, 3,4, 5,6 ); // index == 1, points == 2
```

This has the same result, but here putPoints overwrites rather than extends:

```
QPointArray a( 3 );
a.putPoints( 0, 3, 1,2, 0,0, 5,6 );
a.putPoints( 1, 1, 3,4 );
```

The points are given as a sequence of integers, starting with *firstx* then *firsty*, and so on.

See also resize() [Datastructures and String Handling with Qt] and setPoints() [p. 252].

bool QPointArray::putPoints (int index, int nPoints, const QCOORD * points)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Copies *nPoints* points from the *points* coord array into this point array, and resizes the point array if `index+nPoints` exceeds the size of the array.

Returns TRUE if successful, or FALSE if the array could not be resized (typically due to lack of memory).

bool QPointArray::putPoints (int index, int nPoints, const QPointArray & from, int fromIndex = 0)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This version of the function copies *nPoints* from *from* into this array, starting at *index* in this array and *fromIndex* in *from*. *fromIndex* is 0 by default.

```
QPointArray a;
a.putPoints( 0, 3, 1,2, 0,0, 5,6 );
// a is now the three-point array ( 1,2, 0,0, 5,6 );
QPointArray b;
b.putPoints( 0, 3, 4,4, 5,5, 6,6 );
// b is now ( 4,4, 5,5, 6,6 );
a.putPoints( 2, 3, b );
// a is now ( 1,2, 0,0, 4,4, 5,5, 6,6 );
```

void QPointArray::setPoint (uint index, int x, int y)

Sets the point at position *index* in the array to *(x, y)*.

Example: themes/wood.cpp.

void QPointArray::setPoint (uint i, const QPoint & p)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the point at array index *i* to *p*.

bool QPointArray::setPoints (int nPoints, const QCOORD * points)

Resizes the array to *nPoints* and sets the points in the array to the values taken from *points*.

Returns TRUE if successful, or FALSE if the array could not be resized (normally due to lack of memory).

The example code creates an array with two points (1,2) and (3,4):

```
static QCOORD points[] = { 1,2, 3,4 };
QPointArray a;
a.setPoints( 2, points );
```

See also resize() [Datastructures and String Handling with Qt] and putPoints() [p. 251].

Examples: aclock/aclock.cpp, picture/picture.cpp, themes/metal.cpp and themes/wood.cpp.

bool QPointArray::setPoints (int nPoints, int firstx, int firsty, ...)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Resizes the array to *nPoints* and sets the points in the array to the values taken from the variable argument list.

Returns TRUE if successful, or FALSE if the array could not be resized (typically due to lack of memory).

The example code creates an array with two points (1,2) and (3,4):

```
QPointArray a;
a.setPoints( 2, 1,2, 3,4 );
```

The points are given as a sequence of integers, starting with *firstx* then *firsty*, and so on.

See also resize() [Datastructures and String Handling with Qt] and putPoints() [p. 251].

void QPointArray::translate (int dx, int dy)

Translates all points in the array *(dx, dy)*.

Related Functions

QDataStream & operator<< (QDataStream & s, const QPointArray & a)

Writes the point array, *a* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QPointArray & a)

Reads a point array, *a* from the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QPrinter Class Reference

The QPrinter class is a paint device that paints on a printer.

`#include <qprinter.h>`

Inherits QPaintDevice [p. 170].

Public Members

- enum **PrinterMode** { ScreenResolution, PrinterResolution, HighResolution, Compatible }
- **QPrinter** (PrinterMode m = ScreenResolution)
- **~QPrinter** ()
- enum **Orientation** { Portrait, Landscape }
- enum **PageSize** { A4, B5, Letter, Legal, Executive, A0, A1, A2, A3, A5, A6, A7, A8, A9, B0, B1, B10, B2, B3, B4, B6, B7, B8, B9, C5E, Comm10E, DLE, Folio, Ledger, Tabloid, Custom, NPageSize = Custom }
- enum **PageOrder** { FirstPageFirst, LastPageFirst }
- enum **ColorMode** { GrayScale, Color }
- enum **PaperSource** { OnlyOne, Lower, Middle, Manual, Envelope, EnvelopeManual, Auto, Tractor, SmallFormat, LargeFormat, LargeCapacity, Cassette, FormSource }
- QString **printerName** () const
- virtual void **setPrinterName** (const QString & name)
- bool **outputToFile** () const
- virtual void **setOutputToFile** (bool enable)
- QString **outputFileName** () const
- virtual void **setOutputFileName** (const QString & fileName)
- QString **printProgram** () const
- virtual void **setPrintProgram** (const QString & printProg)
- QString **printerSelectionOption** () const
- virtual void **setPrinterSelectionOption** (const QString & option)
- QString **docName** () const
- virtual void **setDocName** (const QString & name)
- QString **creator** () const
- virtual void **setCreator** (const QString & creator)
- Orientation **orientation** () const
- virtual void **setOrientation** (Orientation orientation)
- PageSize **pageSize** () const
- virtual void **setPageSize** (PageSize newPageSize)
- short **winPageSize** () const
- virtual void **setPageOrder** (PageOrder newPageOrder)
- PageOrder **pageOrder** () const
- virtual void **setResolution** (int dpi)

- virtual int **resolution** () const
- virtual void **setColorMode** (ColorMode newColorMode)
- ColorMode **colorMode** () const
- virtual void **setFullPage** (bool fp)
- bool **fullPage** () const
- QSize **margins** () const
- int **fromPage** () const
- int **toPage** () const
- virtual void **setFromTo** (int fromPage, int toPage)
- int **minPage** () const
- int **maxPage** () const
- virtual void **setMinMax** (int minPage, int maxPage)
- int **numCopies** () const
- virtual void **setNumCopies** (int numCopies)
- bool **newPage** ()
- bool **abort** ()
- bool **aborted** () const
- bool **setup** (QWidget * parent = 0)
- PaperSource **paperSource** () const
- virtual void **setPaperSource** (PaperSource source)

Detailed Description

The QPrinter class is a paint device that paints on a printer.

On Windows it uses the built-in printer drivers. On X11 it generates postscript and sends that to lpr, lp, or another print command.

QPrinter is used much the same way as QWidget and QPixmap are used. The big difference is that you must keep track of the pages.

QPrinter supports a number of settable parameters, most of which can be changed by the end user when the application calls QPrinter::setup().

The most important parameters are:

- setOrientation() tells QPrinter which page orientation to use (virtual).
- setPageSize() tells QPrinter what page size to expect from the printer.
- setResolution() tells QPrinter what resolution you wish the printer to provide (in dpi).
- setFullPage() tells QPrinter whether you want to deal with the full page (so you can have accurate margins, etc.) or just with the part the printer can draw on. The default is FALSE, so that by default you can probably paint on (0,0) but the document's margins are unknown.
- setNumCopies() tells QPrinter how many copies of the document it should print.
- setMinMax() tells QPrinter and QPrintDialog what the allowed range for fromPage() and toPage() are.

Except where noted, you can only call the set functions before setup(), or between QPainter::end() and setup(). (Some may take effect between setup() and begin(), or between begin() and end(), but that's strictly undocumented and such behaviour may differ depending on platform.)

There are also some settings that the user sets (through the printer dialog) and that applications are expected to obey:

- pageOrder() tells the application program whether to print first-page-first or last-page-first.

- colorMode() tells the application program whether to print in color or grayscale. (If you print in color and the printer does not support color, Qt will try to approximate. The document may take longer to print, but the quality should not be made visibly poorer.)
- fromPage() and toPage() indicate what pages the application program should print.
- paperSource() tells the application progam which paper source to print from.

You can of course call these functions to establish defaults before you ask the user through QPrinter::setup().

Once you start printing, newPage() is essential. You will probably also need to look at the QPaintDeviceMetrics for the printer (see the print function in the Application walk-through). Note that the paint device metrics are valid only after the QPrinter has been set up, i.e. after setup() has returned successfully. If you want high-quality printing with accurate margins, it is essential to call setFullPage(TRUE).

If you want to abort the print job, abort() will try its best to stop printing. It may cancel the entire job or just some of it.

The true type font embedding for Qt's post script driver uses code by David Chappell of Trinity College Computing Center.

Copyright 1995, Trinity College Computing Center. Written by David Chappell.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

TrueType font support. These functions allow PPR to generate PostScript fonts from Microsoft compatible TrueType font files.

The functions in this file do most of the work to convert a TrueType font to a type 3 PostScript font.

Most of the material in this file is derived from a program called "ttf2ps" which L. S. Ng posted to the usenet news group "comp.sources.postscript". The author did not provide a copyright notice or indicate any restrictions on use.

Last revised 11 July 1995.

See also Graphics Classes and Image Processing Classes.

Member Type Documentation

QPrinter::ColorMode

This enum type is used to indicate whether QPrinter should print in color or not. The possible values are:

- `QPrinter::Color` - print in color if available, otherwise in grayscale. This is the default.
- `QPrinter::GrayScale` - print in grayscale, even on color printers. Might be a little faster than Color.

QPrinter::Orientation

This enum type (not to be confused with Qt::Orientation) is used to specify each page's orientation.

- `QPrinter::Portrait` - the page's height is greater than its width (the default).
- `QPrinter::Landscape` - the page's width is greater than its height.

This type interacts with QPrinter::PageSize and QPrinter::setFullPage() to determine the final size of the page available to the application.

QPrinter::PageOrder

This enum type is used by QPrinter to tell the application program how to print. The possible values are

- `QPrinter::FirstPageFirst` - the lowest-numbered page should be printed first.
- `QPrinter::LastPageFirst` - the highest-numbered page should be printed first.

QPrinter::PageSize

This enum type specifies what paper size QPrinter should use. QPrinter does not check that the paper size is available; it just uses this information, together with QPrinter::Orientation and QPrinter::setFullPage(), to determine the printable area (see QPaintDeviceMetrics).

The defined sizes (with setFullPage(TRUE)) are:

- `QPrinter::A0` - 841 x 1189 mm
- `QPrinter::A1` - 594 x 841 mm
- `QPrinter::A2` - 420 x 594 mm
- `QPrinter::A3` - 297 x 420 mm
- `QPrinter::A4` - 210 x 297 mm, 8.26 x 11.7 inches
- `QPrinter::A5` - 148 x 210 mm
- `QPrinter::A6` - 105 x 148 mm
- `QPrinter::A7` - 74 x 105 mm
- `QPrinter::A8` - 52 x 74 mm
- `QPrinter::A9` - 37 x 52 mm
- `QPrinter::B0` - 1030 x 1456 mm
- `QPrinter::B1` - 728 x 1030 mm
- `QPrinter::B10` - 32 x 45 mm
- `QPrinter::B2` - 515 x 728 mm
- `QPrinter::B3` - 364 x 515 mm
- `QPrinter::B4` - 257 x 364 mm
- `QPrinter::B5` - 182 x 257 mm, 7.17 x 10.13 inches
- `QPrinter::B6` - 128 x 182 mm
- `QPrinter::B7` - 91 x 128 mm
- `QPrinter::B8` - 64 x 91 mm
- `QPrinter::B9` - 45 x 64 mm
- `QPrinter::C5E` - 163 x 229 mm
- `QPrinter::Comm10E` - 105 x 241 mm, US Common #10 Envelope
- `QPrinter::DLE` - 110 x 220 mm
- `QPrinter::Executive` - 7.5 x 10 inches, 191 x 254 mm
- `QPrinter::Folio` - 210 x 330 mm
- `QPrinter::Ledger` - 432 x 279 mm
- `QPrinter::Legal` - 8.5 x 14 inches, 216 x 356 mm
- `QPrinter::Letter` - 8.5 x 11 inches, 216 x 279 mm
- `QPrinter::Tabloid` - 279 x 432 mm
- `QPrinter::Custom`
- `QPrinter::NPageSize` - (internal)

With setFullPage(FALSE) (the default), the metrics will be a bit smaller; how much depends on the printer in use.

QPrinter::PaperSource

This enum type specifies what paper source QPrinter is to use. QPrinter does not check that the paper source is available; it just uses this information to try and set the paper source. Whether it will set the paper source depends on whether the printer has that particular source.

Note: this is currently only implemented for Windows.

- `QPrinter::OnlyOne`
- `QPrinter::Lower`
- `QPrinter::Middle`
- `QPrinter::Manual`
- `QPrinter::Envelope`
- `QPrinter::EnvelopeManual`
- `QPrinter::Auto`
- `QPrinter::Tractor`
- `QPrinter::SmallFormat`
- `QPrinter::LargeFormat`
- `QPrinter::LargeCapacity`
- `QPrinter::Cassette`
- `QPrinter::FormSource`

QPrinter::PrinterMode

This enum describes the mode the printer should work in. It basically presets a certain resolution and working mode.

- `QPrinter::ScreenResolution` - Sets the resolution of the print device to the screen resolution. This has the big advantage that the results obtained when painting on the printer will match more or less exactly the visible output on the screen. It is the easiest to use, as font metrics on the screen and on the printer are the same. This is the default value.
- `QPrinter::PrinterResolution` - Use the physical resolution of the printer on Windows. On Unix, set the postscript resolution to 72 dpi.
- `QPrinter::HighResolution` - Use printer resolution on windows, set the resolution of the postscript driver to 600dpi.
- `QPrinter::Compatible` - Almost the same as PrinterResolution, but keeps some peculiarities of the printer dirver of Qt 2.x. This is useful, when porting an application from Qt 2.x to Qt 3.x.

Member Function Documentation

QPrinter::QPrinter (PrinterMode m = ScreenResolution)

Constructs a printer paint device with mode *m*.

See also QPrinter::PrinterMode [p. 258].

QPrinter::~QPrinter ()

Destroys the printer paint device and cleans up.

bool QPrinter::abort ()

Aborts the print job. Returns TRUE if successful, otherwise FALSE.

See also aborted() [p. 259].

bool QPrinter::aborted () const

Returns TRUE is the printer job was aborted, otherwise FALSE.

See also abort() [p. 259].

ColorMode QPrinter::colorMode () const

Returns the current color mode. The default color mode is Color.

See also setColorMode() [p. 262].

QString QPrinter::creator () const

Returns the name of the application that created the document.

See also setCreator() [p. 262].

QString QPrinter::docName () const

Returns the document name.

See also setDocName() [p. 262].

int QPrinter::fromPage () const

Returns the from-page setting. The default value is 0.

If fromPage() and toPage() both return 0 this should signify 'print the whole document'.

The programmer is responsible for reading this setting and printing accordingly.

See also setFromTo() [p. 262] and toPage() [p. 265].

bool QPrinter::fullPage () const

Returns TRUE if the origin of the printer's coordinate system is at the corner of the sheet and FALSE if it is at the edge of the printable area.

See setFullPage() for details and caveats.

See also setFullPage() [p. 262], PageSize [p. 257] and QPaintDeviceMetrics [p. 176].

QSize QPrinter::margins () const

Returns the width of the left/right and top/bottom margins of the printer. This is a best-effort guess, not based on perfect knowledge.

If you have called setFullPage(TRUE) (this is recommended for high-quality printing), margins().width() may be treated as the smallest sane left/right margin you can use, and margins().height() as the smallest sane top/bottom margins you can use.

If you have called setFullPage(FALSE) (this is the default), margins() is automatically subtracted from the pageSize() by QPrinter.

See also setFullPage() [p. 262], QPaintDeviceMetrics [p. 176] and PageSize [p. 257].

int QPrinter::maxPage () const

Returns the max-page setting. A user can't choose a higher page number than maxPage() when they select a print range. The default value is 0.

See also minPage() [p. 260], setMinMax() [p. 262] and setFromTo() [p. 262].

int QPrinter::minPage () const

Returns the min-page setting, i.e. the lowest page number a user is allowed to choose. The default value is 0.

See also maxPage() [p. 260], setMinMax() [p. 262] and setFromTo() [p. 262].

bool QPrinter::newPage ()

Advances to a new page on the printer. Returns TRUE if successful, otherwise FALSE.

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp and mdi/application.cpp.

int QPrinter::numCopies () const

Returns the number of copies to be printed. The default value is 1.

See also setNumCopies() [p. 263].

Orientation QPrinter::orientation () const

Returns the orientation setting. The default value is QPrinter::Portrait.

See also setOrientation() [p. 263].

QString QPrinter::outputFileName () const

Returns the name of the output file. There is no default file name.

See also setOutputFileName() [p. 263] and setOutputToFile() [p. 263].

bool QPrinter::outputToFile () const

Returns TRUE if the output should be written to a file, or FALSE if the output should be sent directly to the printer. The default setting is FALSE.

This function is currently only supported under X11.

See also setOutputToFile() [p. 263] and setOutputFileName() [p. 263].

PageOrder QPrinter::pageOrder () const

Returns the current page order.

The default page order is FirstPageFirst.

PageSize QPrinter::pageSize () const

Returns the printer page size. The default value is system-dependent.

See also setPageSize() [p. 263].

PaperSource QPrinter::paperSource () const

Returns the currently set paper source of the printer.

See also setPaperSource() [p. 264].

QString QPrinter::printProgram () const

Returns the name of the program that sends the print output to the printer.

The default is to return a null string; meaning that QPrinter will try to be smart in a system-dependent way. On X11 only, you can set it to something different to use a specific print program.

On Windows, this function returns the name of the printer device driver.

See also setPrintProgram() [p. 264] and setPrinterSelectionOption() [p. 264].

QString QPrinter::printerName () const

Returns the printer name. This value is initially set to the name of the default printer.

See also setPrinterName() [p. 264].

QString QPrinter::printerSelectionOption () const

Returns the printer options selection string. This is useful only if the print command has been explicitly set.

The default value (a null string) implies that the printer should be selected in a system-dependent manner.

Any other value implies that the given value should be used.

See also setPrinterSelectionOption() [p. 264].

int QPrinter::resolution () const [virtual]

Returns the current assumed resolution of the printer, as set by setResolution() or by the printer subsystem.

See also setResolution() [p. 264].

void QPrinter::setColorMode (ColorMode newColorMode) [virtual]

Sets the printer's color mode to *newColorMode*, which can be one of Color (the default) or GrayScale.

See also colorMode() [p. 259].

void QPrinter::setCreator (const QString & creator) [virtual]

Sets the name of the application that created the document to *creator*.

This function is only applicable to the X11 version of Qt. If no creator name is specified, the creator will be set to "Qt" followed by some version number.

See also creator() [p. 259].

void QPrinter::setDocName (const QString & name) [virtual]

Sets the document name to *name*.

void QPrinter::setFromTo (int fromPage, int toPage) [virtual]

Sets the from-page and to-page settings to *fromPage* and *toPage* respectively.

The from-page and to-page settings specify what pages to print.

If *fromPage* and *toPage* are both 0 this should signify 'print the whole document'.

This function is useful mostly to set a default value that the user can override in the print dialog when you call setup().

See also fromPage() [p. 259], toPage() [p. 265], setMinMax() [p. 262] and setup() [p. 264].

void QPrinter::setFullPage (bool fp) [virtual]

Sets QPrinter to have the origin of the coordinate system at the top-left corner of the paper if *fp* is TRUE, or where it thinks the top-left corner of the printable area is if *fp* is FALSE.

The default is FALSE. You can (probably) print on (0,0), and QPaintDeviceMetrics will report something smaller than the size indicated by PageSize. (Note that QPrinter may be wrong - it does not have perfect knowledge of the physical printer.)

If you set *fp* to TRUE, QPaintDeviceMetrics will report the exact same size as indicated by PageSize, but you cannot print on all of that - you have to take care of the output margins yourself.

See also PageSize [p. 257], setPageSize() [p. 263], QPaintDeviceMetrics [p. 176] and fullPage() [p. 259].

Example: helpviewer/helpwindow.cpp.

void QPrinter::setMinMax (int minPage, int maxPage) [virtual]

Sets the min-page and max-page settings to *minPage* and *maxPage* respectively.

The min-page and max-page restrict the from-page and to-page settings. When the printer setup dialog appears, the user cannot select a from page or a to page that are outside the range specified by min and max pages.

See also minPage() [p. 260], maxPage() [p. 260], setFromTo() [p. 262] and setup() [p. 264].

void QPrinter::setNumCopies (int numCopies) [virtual]

Sets the number of pages to be printed to *numCopies*.

The printer driver reads this setting and prints the specified number of copies.

See also numCopies() [p. 260] and setup() [p. 264].

void QPrinter::setOrientation (Orientation orientation) [virtual]

Sets the print orientation to *orientation*.

The orientation can be either QPrinter::Portrait or QPrinter::Landscape.

The printer driver reads this setting and prints using the specified orientation. On Windows however, this setting won't take effect until the printer dialog is shown (using QPrinter::setup()).

See also orientation() [p. 260].

void QPrinter::setOutputFileName (const QString & fileName) [virtual]

Sets the name of the output file to *fileName*.

Setting a null or empty name (0 or "") disables output to a file, i.e. calls setOutputToFile(FALSE). Setting a non-empty name enables output to a file, i.e. calls setOutputToFile(TRUE).

This function is currently only supported under X11.

See also outputFileName() [p. 260] and setOutputToFile() [p. 263].

void QPrinter::setOutputToFile (bool enable) [virtual]

Specifies whether the output should be written to a file or sent directly to the printer.

Will output to a file if *enable* is TRUE, or will output directly to the printer if *enable* is FALSE.

This function is currently only supported under X11.

See also outputToFile() [p. 260] and setOutputFileName() [p. 263].

void QPrinter::setPageOrder (PageOrder newPageOrder) [virtual]

Sets the page order to *newPageOrder*.

The page order can be QPrinter::FirstPageFirst or QPrinter::LastPageFirst. The application programmer is responsible for reading the page order and printing accordingly.

This function is useful mostly for setting a default value that the user can override in the print dialog when you call setup().

void QPrinter::setPageSize (PageSize newPageSize) [virtual]

Sets the printer page size to *newPageSize* if that size is supported. The result if undefined if *newPageSize* is not supported.

The default page size is system-dependent.

This function is useful mostly for setting a default value that the user can override in the print dialog when you call setup().

See also pageSize() [p. 261], PageSize [p. 257], setFullPage() [p. 262] and setResolution() [p. 264].

void QPrinter::setPaperSource (PaperSource source) [virtual]

Sets the paper source setting to *source*.

See also paperSource() [p. 261].

void QPrinter::setPrintProgram (const QString & printProg) [virtual]

Sets the name of the program that should do the print job to *printProg*.

On X11, this function sets the program to call with the PostScript output. On other platforms, it has no effect.

See also printProgram() [p. 261].

void QPrinter::setPrinterName (const QString & name) [virtual]

Sets the printer name to *name*.

The default printer will be used if no printer name is set.

Under X11, the PRINTER environment variable defines the default printer. Under any other window system, the window system defines the default printer.

See also printerName() [p. 261].

void QPrinter::setPrinterSelectionOption (const QString & option) [virtual]

Sets the printer to use *option* to select the printer. *option* is null by default (which implies that Qt should be smart enough to guess correctly), but it can be set to other values to use a specific printer selection option.

If the printer selection option is changed while the printer is active, the current print job may or may not be affected.

void QPrinter::setResolution (int dpi) [virtual]

Requests that the printer prints at *dpi* or as near to *dpi* as possible.

This setting affects the coordinate system as returned by e.g. QPaintDeviceMetrics and QPainter::viewport().

The value depends on the PrintingMode used in the QPrinter constructor. By default, the dpi value of the screen is used.

This function must be called before setup() to have an effect on all platforms.

See also resolution() [p. 261] and setPageSize() [p. 263].

bool QPrinter::setup (QWidget * parent = 0)

Opens a printer setup dialog, with parent *parent*, and asks the user to specify what printer to use and miscellaneous printer settings.

Returns TRUE if the user pressed "OK" to print, or FALSE if the user cancelled the operation.

Examples: action/application.cpp, application/application.cpp, drawdemo/drawdemo.cpp, helpviewer/helpwindow.cpp and mdi/application.cpp.

int QPrinter::toPage () const

Returns the to-page setting. The default value is 0.

If fromPage() and toPage() both return 0 this should signify 'print the whole document'.

The programmer is responsible for reading this setting and printing accordingly.

See also setFromTo() [p. 262] and fromPage() [p. 259].

short QPrinter::winPageSize () const

Returns the Windows page size value as used by the DEVMODE struct (Windows only). Using this function is not portable.

Use pageSize() to get the PageSize, e.g. 'A4', 'Letter', etc.

QRect Class Reference

The QRect class defines a rectangle in the plane.

```
#include <qrect.h>
```

Public Members

- **QRect** ()
- **QRect** (const QPoint & topLeft, const QPoint & bottomRight)
- **QRect** (const QPoint & topLeft, const QSize & size)
- **QRect** (int left, int top, int width, int height)
- bool **isNull** () const
- bool **isEmpty** () const
- bool **isValid** () const
- QRect **normalize** () const
- int **left** () const
- int **top** () const
- int **right** () const
- int **bottom** () const
- QCOORD & **rLeft** ()
- QCOORD & **rTop** ()
- QCOORD & **rRight** ()
- QCOORD & **rBottom** ()
- int **x** () const
- int **y** () const
- void **setLeft** (int pos)
- void **setTop** (int pos)
- void **setRight** (int pos)
- void **setBottom** (int pos)
- void **setX** (int x)
- void **setY** (int y)
- QPoint **topLeft** () const
- QPoint **bottomRight** () const
- QPoint **topRight** () const
- QPoint **bottomLeft** () const
- QPoint **center** () const
- void **rect** (int * x, int * y, int * w, int * h) const
- void **coords** (int * xp1, int * yp1, int * xp2, int * yp2) const
- void **moveTopLeft** (const QPoint & p)
- void **moveBottomRight** (const QPoint & p)

- void **moveTopRight** (const QPoint & p)
- void **moveBottomLeft** (const QPoint & p)
- void **moveCenter** (const QPoint & p)
- void **moveBy** (int dx, int dy)
- void **setRect** (int x, int y, int w, int h)
- void **setCoords** (int xp1, int yp1, int xp2, int yp2)
- void **addCoords** (int xp1, int yp1, int xp2, int yp2)
- QSize **size** () const
- int **width** () const
- int **height** () const
- void **setWidth** (int w)
- void **setHeight** (int h)
- void **setSize** (const QSize & s)
- QRect **operator|** (const QRect & r) const
- QRect **operator&** (const QRect & r) const
- QRect & **operator|=** (const QRect & r)
- QRect & **operator&=** (const QRect & r)
- bool **contains** (const QPoint & p, bool proper = FALSE) const
- bool **contains** (int x, int y, bool proper = FALSE) const
- bool **contains** (const QRect & r, bool proper = FALSE) const
- QRect **unite** (const QRect & r) const
- QRect **intersect** (const QRect & r) const
- bool **intersects** (const QRect & r) const

Related Functions

- bool **operator==** (const QRect & r1, const QRect & r2)
- bool **operator!=** (const QRect & r1, const QRect & r2)
- QDataStream & **operator<<** (QDataStream & s, const QRect & r)
- QDataStream & **operator>>** (QDataStream & s, QRect & r)

Detailed Description

The QRect class defines a rectangle in the plane.

A rectangle is internally represented as an upper-left corner and a bottom-right corner, but it is normally expressed as an upper-left corner and a size.

The coordinate type is QCOORD (defined in qwindowdefs.h as int). The minimum value of QCOORD is QCOORD_MIN (-2147483648) and the maximum value is QCOORD_MAX (2147483647).

Note that the size (width and height) of a rectangle might be different from what you are used to. If the top-left corner and the bottom-right corner are the same, the height and the width of the rectangle will both be 1.

Generally, *width = right - left + 1* and *height = bottom - top + 1*. We designed it this way to make it correspond to rectangular spaces used by drawing functions in which the width and height denote a number of pixels. For example, drawing a rectangle with width and height 1 draws a single pixel.

The default coordinate system has origin (0, 0) in the top-left corner. The positive direction of the y axis is down, and the positive x axis is from left to right.

A QRect can be constructed with a set of left, top, width and height integers, from two QPoints or from a QPoint and a QSize. After creation the dimensions can be changed, e.g. with setLeft(), setRight(), setTop() and setBottom(),

or by setting sizes, e.g. setWidth(), setHeight() and setSize(). The dimensions can also be changed with the move functions, e.g. moveBy(), moveCenter(), moveBottomRight(), etc. You can also add coordinates to a rectangle with addCoords().

You can test to see if a QRect contains a specific point with contains(). You can also test to see if two QRects intersect with intersects() (see also intersect()). To get the bounding rectangle of two QRects use unite().

See also QPoint [p. 242], QSize [p. 285], Graphics Classes and Image Processing Classes.

Member Function Documentation

QRect::QRect ()

Constructs an invalid rectangle.

QRect::QRect (const QPoint & topLeft, const QPoint & bottomRight)

Constructs a rectangle with *topLeft* as the top-left corner and *bottomRight* as the bottom-right corner.

QRect::QRect (const QPoint & topLeft, const QSize & size)

Constructs a rectangle with *topLeft* as the top-left corner and *size* as the rectangle size.

QRect::QRect (int left, int top, int width, int height)

Constructs a rectangle with the *top*, *left* corner and *width* and *height*.

Example (creates three identical rectangles):

```
QRect r1( QPoint(100,200), QPoint(110,215) );
QRect r2( QPoint(100,200), QSize(11,16) );
QRect r3( 100, 200, 11, 16 );
```

void QRect::addCoords (int xp1, int yp1, int xp2, int yp2)

Adds *xp1*, *yp1*, *xp2* and *yp2* respectively to the existing coordinates of the rectangle.

int QRect::bottom () const

Returns the bottom coordinate of the rectangle.

See also top() [p. 275], setBottom() [p. 273], bottomLeft() [p. 268] and bottomRight() [p. 269].

Examples: desktop/desktop.cpp, helpviewer/helpwindow.cpp, qfd/fontdisplayer.cpp, scribble/scribble.cpp and themes/wood.cpp.

QPoint QRect::bottomLeft () const

Returns the bottom-left position of the rectangle.

See also moveBottomLeft() [p. 271], bottomRight() [p. 269], topLeft() [p. 275], topRight() [p. 275], bottom() [p. 268] and left() [p. 271].

Example: tictac/tictac.cpp.

QPoint QRect::bottomRight () const

Returns the bottom-right position of the rectangle.

See also moveBottomRight() [p. 271], bottomLeft() [p. 268], topLeft() [p. 275], topRight() [p. 275], bottom() [p. 268] and right() [p. 273].

Example: tictac/tictac.cpp.

QPoint QRect::center () const

Returns the center point of the rectangle.

See also moveCenter() [p. 271], topLeft() [p. 275], topRight() [p. 275], bottomLeft() [p. 268] and bottomRight() [p. 269].

Example: tooltip/tooltip.cpp.

bool QRect::contains (const QPoint & p, bool proper = FALSE) const

Returns TRUE if the point *p* is inside or on the edge of the rectangle; otherwise returns FALSE.

If *proper* is TRUE, this function returns TRUE only if *p* is inside (not on the edge).

Example: t14/cannon.cpp.

bool QRect::contains (int x, int y, bool proper = FALSE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the point *x*, *y* is inside this rectangle; otherwise returns FALSE.

If *proper* is TRUE, this function returns TRUE only if the point is entirely inside (not on the edge).

bool QRect::contains (const QRect & r, bool proper = FALSE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the rectangle *r* is inside this rectangle; otherwise returns FALSE.

If *proper* is TRUE, this function returns TRUE only if *r* is entirely inside (not on the edge).

See also unite() [p. 276], intersect() [p. 270] and intersects() [p. 270].

void QRect::coords (int * xp1, int * yp1, int * xp2, int * yp2) const

Extracts the rectangle parameters as the top-left point *xp1, *yp1* and the bottom-right point *xp2, *yp2*.

See also setCoords() [p. 273] and rect() [p. 273].

Examples: themes/metal.cpp and themes/wood.cpp.

int QRect::height () const

Returns the height of the rectangle. The height includes both the top and bottom edges, i.e. height = bottom - top + 1.

See also width() [p. 276], size() [p. 275] and setHeight() [p. 273].

Examples: aclock/aclock.cpp, desktop/desktop.cpp, movies/main.cpp, scribble/scribble.cpp, themes/metal.cpp, themes/wood.cpp and xform/xform.cpp.

QRect QRect::intersect (const QRect & r) const

Returns the intersection of this rectangle and rectangle *r*. `r.intersect(s)` is equivalent to `r&s`.

bool QRect::intersects (const QRect & r) const

Returns TRUE if this rectangle intersects with rectangle *r* (there is at least one pixel that is within both rectangles); otherwise returns FALSE.

See also intersect() [p. 270] and contains() [p. 269].

Examples: t11/cannon.cpp, t12/cannon.cpp, t13/cannon.cpp and t14/cannon.cpp.

bool QRect::isEmpty () const

Returns TRUE if the rectangle is empty; otherwise returns FALSE.

An empty rectangle has a left() > right() or top() > bottom().

An empty rectangle is not valid. `isEmpty() == !isValid()`

See also isNull() [p. 270] and isValid() [p. 270].

bool QRect::isNull () const

Returns TRUE if the rectangle is a null rectangle; otherwise returns FALSE.

A null rectangle has both the width and the height set to 0, that is right() == left() - 1 and bottom() == top() - 1.

Remember that if right() == left() and bottom() == top(), then the rectangle has width 1 and height 1.

A null rectangle is also empty.

A null rectangle is not valid.

See also isEmpty() [p. 270] and isValid() [p. 270].

bool QRect::isValid () const

Returns TRUE if the rectangle is valid or FALSE if it is invalid (empty).

A valid rectangle has a left() <= right() and top() <= bottom().

`isValid() == !isEmpty()`

See also isNull() [p. 270], isEmpty() [p. 270] and normalize() [p. 272].

Examples: themes/metal.cpp and tooltip/tooltip.cpp.

int QRect::left () const

Returns the left coordinate of the rectangle. Identical to x().

See also x() [p. 276], top() [p. 275], right() [p. 273], setLeft() [p. 274], topLeft() [p. 275] and bottomLeft() [p. 268].

Examples: aclock/aclock.cpp, desktop/desktop.cpp, qfd/fontdisplayer.cpp, scribble/scribble.cpp, tictac/tictac.cpp and xform/xform.cpp.

void QRect::moveBottomLeft (const QPoint & p)

Sets the bottom-left position of the rectangle to *p*, leaving the size unchanged.

See also bottomLeft() [p. 268], moveBottomRight() [p. 271], moveTopLeft() [p. 271], moveTopRight() [p. 271], setBottom() [p. 273] and setLeft() [p. 274].

Example: t10/cannon.cpp.

void QRect::moveBottomRight (const QPoint & p)

Sets the bottom-right position of the rectangle to *p*, leaving the size unchanged.

See also bottomRight() [p. 269], moveBottomLeft() [p. 271], moveTopLeft() [p. 271], moveTopRight() [p. 271], setBottom() [p. 273] and setRight() [p. 274].

void QRect::moveBy (int dx, int dy)

Moves the rectangle *dx* along the X axis and *dy* along the Y axis, relative to the current position. (Positive values move the rectangle right and/or down.)

Examples: helpviewer/helpwindow.cpp, themes/wood.cpp and xform/xform.cpp.

void QRect::moveCenter (const QPoint & p)

Sets the center point of the rectangle to *p*, leaving the size unchanged.

See also center() [p. 269], moveTopLeft() [p. 271], moveTopRight() [p. 271], moveBottomLeft() [p. 271] and moveBottomRight() [p. 271].

Examples: t11/cannon.cpp and t12/cannon.cpp.

void QRect::moveTopLeft (const QPoint & p)

Sets the top-left position of the rectangle to *p*, leaving the size unchanged.

See also topLeft() [p. 275], moveTopRight() [p. 271], moveBottomLeft() [p. 271], moveBottomRight() [p. 271], setTop() [p. 274] and setLeft() [p. 274].

Example: xform/xform.cpp.

void QRect::moveTopRight (const QPoint & p)

Sets the top-right position of the rectangle to *p*, leaving the size unchanged.

See also topRight() [p. 275], moveTopLeft() [p. 271], moveBottomLeft() [p. 271], moveBottomRight() [p. 271], setTop() [p. 274] and setRight() [p. 274].

QRect QRect::normalize () const

Returns a normalized rectangle, i.e. a rectangle that has a non-negative width and height.

It swaps left and right if left() > right(), and swaps top and bottom if top() > bottom().

See also isValid() [p. 270].

Example: scribble/scribble.cpp.

QRect QRect::operator& (const QRect & r) const

Returns the intersection of this rectangle and rectangle *r*.

Returns an empty rectangle if there is no intersection.

See also operator&=() [p. 272], operator|() [p. 272], isEmpty() [p. 270], intersects() [p. 270] and contains() [p. 269].

QRect & QRect::operator&= (const QRect & r)

Intersects this rectangle with rectangle *r*.

QRect QRect::operator| (const QRect & r) const

Returns the bounding rectangle of this rectangle and rectangle *r*.

The bounding rectangle of a nonempty rectangle and an empty or invalid rectangle is defined to be the nonempty rectangle.

See also operator|=() [p. 272], operator&() [p. 272], intersects() [p. 270] and contains() [p. 269].

QRect & QRect::operator|= (const QRect & r)

Unites this rectangle with rectangle *r*.

QCOORD & QRect::rBottom ()

Returns a reference to the bottom coordinate of the rectangle.

See also rLeft() [p. 272], rTop() [p. 273] and rRight() [p. 273].

QCOORD & QRect::rLeft ()

Returns a reference to the left coordinate of the rectangle.

See also rTop() [p. 273], rRight() [p. 273] and rBottom() [p. 272].

QCOORD & QRect::rRight ()

Returns a reference to the right coordinate of the rectangle.

See also rLeft() [p. 272], rTop() [p. 273] and rBottom() [p. 272].

QCOORD & QRect::rTop ()

Returns a reference to the top coordinate of the rectangle.

See also rLeft() [p. 272], rRight() [p. 273] and rBottom() [p. 272].

void QRect::rect (int * x, int * y, int * w, int * h) const

Extracts the rectangle parameters as the position *x, *y and width *w and height *h.

See also setRect() [p. 274] and coords() [p. 269].

Examples: themes/metal.cpp and themes/wood.cpp.

int QRect::right () const

Returns the right coordinate of the rectangle.

See also left() [p. 271], setRight() [p. 274], topRight() [p. 275] and bottomRight() [p. 269].

Examples: customlayout/flow.cpp, desktop/desktop.cpp, helpviewer/helpwindow.cpp, qfd/fontdisplayer.cpp, scribble/scribble.cpp, t11/cannon.cpp and themes/wood.cpp.

void QRect::setBottom (int pos)

Sets the bottom edge of the rectangle to *pos*. May change the height, but will never change the top edge of the rectangle.

See also bottom() [p. 268], setTop() [p. 274] and setHeight() [p. 273].

Example: scribble/scribble.cpp.

void QRect::setCoords (int xp1, int yp1, int xp2, int yp2)

Sets the coordinates of the rectangle's top-left corner to *(xp1, yp1)*, and the coordinates of its bottom-right corner to *(xp2, yp2)*.

See also coords() [p. 269] and setRect() [p. 274].

void QRect::setHeight (int h)

Sets the height of the rectangle to *h*. The top edge is not moved, but the bottom edge may be moved.

See also height() [p. 270], setTop() [p. 274], setBottom() [p. 273] and setSize() [p. 274].

Example: desktop/desktop.cpp.

void QRect::setLeft (int pos)

Sets the left edge of the rectangle to *pos*. May change the width, but will never change the right edge of the rectangle.

Identical to setX().

See also left() [p. 271], setTop() [p. 274] and setWidth() [p. 274].

Example: scribble/scribble.cpp.

void QRect::setRect (int x, int y, int w, int h)

Sets the coordinates of the rectangle's top-left corner to *(x, y)*, and its size to *(w, h)*.

See also rect() [p. 273] and setCoords() [p. 273].

Example: themes/wood.cpp.

void QRect::setRight (int pos)

Sets the right edge of the rectangle to *pos*. May change the width, but will never change the left edge of the rectangle.

See also right() [p. 273], setLeft() [p. 274] and setWidth() [p. 274].

Example: scribble/scribble.cpp.

void QRect::setSize (const QSize & s)

Sets the size of the rectangle to *s*. The top-left corner is not moved.

See also size() [p. 275], setWidth() [p. 274] and setHeight() [p. 273].

Example: xform/xform.cpp.

void QRect::setTop (int pos)

Sets the top edge of the rectangle to *pos*. May change the height, but will never change the bottom edge of the rectangle.

Identical to setY().

See also top() [p. 275], setBottom() [p. 273] and setHeight() [p. 273].

Example: scribble/scribble.cpp.

void QRect::setWidth (int w)

Sets the width of the rectangle to *w*. The right edge is changed, but not the left edge.

See also width() [p. 276], setLeft() [p. 274], setRight() [p. 274] and setSize() [p. 274].

Example: desktop/desktop.cpp.

void QRect::setX (int x)

Sets the x position of the rectangle (its left end) to *x*. May change the width, but will never change the right edge of the rectangle.

Identical to setLeft().

See also x() [p. 276] and setY() [p. 275].

void QRect::setY (int y)

Sets the y position of the rectangle (its top) to *y*. May change the height, but will never change the bottom edge of the rectangle.

Identical to setTop().

See also y() [p. 276] and setX() [p. 275].

QSize QRect::size () const

Returns the size of the rectangle.

See also width() [p. 276] and height() [p. 270].

Examples: desktop/desktop.cpp, movies/main.cpp and t10/cannon.cpp.

int QRect::top () const

Returns the top coordinate of the rectangle. Identical to y().

See also y() [p. 276], left() [p. 271], bottom() [p. 268], setTop() [p. 274], topLeft() [p. 275] and topRight() [p. 275].

Examples: aclock/aclock.cpp, desktop/desktop.cpp, helpviewer/helpwindow.cpp, scribble/scribble.cpp, themes/wood.cpp, tictac/tictac.cpp and xform/xform.cpp.

QPoint QRect::topLeft () const

Returns the top-left position of the rectangle.

See also moveTopLeft() [p. 271], topRight() [p. 275], bottomLeft() [p. 268], bottomRight() [p. 269], left() [p. 271] and top() [p. 275].

Examples: t10/cannon.cpp and tictac/tictac.cpp.

QPoint QRect::topRight () const

Returns the top-right position of the rectangle.

See also moveTopRight() [p. 271], topLeft() [p. 275], bottomLeft() [p. 268], bottomRight() [p. 269], top() [p. 275] and right() [p. 273].

Example: tictac/tictac.cpp.

QRect QRect::unite (const QRect & r) const

Returns the bounding rectangle of this rectangle and rectangle *r*. `r.unite(s)` is equivalent to `r|s`.

Examples: t11/cannon.cpp, t12/cannon.cpp and xform/xform.cpp.

int QRect::width () const

Returns the width of the rectangle. The width includes both the left and right edges, i.e. width = right - left + 1.

See also height() [p. 270], size() [p. 275] and setHeight() [p. 273].

Examples: aclock/aclock.cpp, customlayout/border.cpp, desktop/desktop.cpp, movies/main.cpp, themes/metal.cpp, themes/wood.cpp and xform/xform.cpp.

int QRect::x () const

Returns the left coordinate of the rectangle. Identical to left().

See also left() [p. 271], y() [p. 276] and setX() [p. 275].

Examples: customlayout/border.cpp, desktop/desktop.cpp, movies/main.cpp, scribble/scribble.cpp, t12/cannon.cpp, themes/metal.cpp and themes/wood.cpp.

int QRect::y () const

Returns the top coordinate of the rectangle. Identical to top().

See also top() [p. 275], x() [p. 276] and setY() [p. 275].

Examples: desktop/desktop.cpp, movies/main.cpp, scribble/scribble.cpp, t12/cannon.cpp, t14/cannon.cpp, themes/metal.cpp and themes/wood.cpp.

Related Functions

bool operator!= (const QRect & r1, const QRect & r2)

Returns TRUE if *r1* and *r2* are different; otherwise returns FALSE.

QDataStream & operator<< (QDataStream & s, const QRect & r)

Writes the QRect, *r*, to the stream *s*, and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

bool operator== (const QRect & r1, const QRect & r2)

Returns TRUE if *r1* and *r2* are equal; otherwise returns FALSE.

QDataStream & operator>> (QDataStream & s, QRect & r)

Reads a QRect from the stream *s* into rect *r* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QRegion Class Reference

The QRegion class specifies a clip region for a painter.

```
#include <qregion.h>
```

Public Members

- enum **RegionType** { Rectangle, Ellipse }
- **QRegion** ()
- **QRegion** (int x, int y, int w, int h, RegionType t = Rectangle)
- **QRegion** (const QRect & r, RegionType t = Rectangle)
- **QRegion** (const QPointArray & a, bool winding = FALSE)
- **QRegion** (const QRegion & r)
- **QRegion** (const QBitmap & bm)
- **~QRegion** ()
- QRegion & **operator=** (const QRegion & r)
- bool **isNull** () const
- bool **isEmpty** () const
- bool **contains** (const QPoint & p) const
- bool **contains** (const QRect & r) const
- void **translate** (int dx, int dy)
- QRegion **unite** (const QRegion & r) const
- QRegion **intersect** (const QRegion & r) const
- QRegion **subtract** (const QRegion & r) const
- QRegion **eor** (const QRegion & r) const
- QRect **boundingRect** () const
- QMemArray<QRect> **rects** () const
- const QRegion **operator|** (const QRegion & r) const
- const QRegion **operator+** (const QRegion & r) const
- const QRegion **operator&** (const QRegion & r) const
- const QRegion **operator-** (const QRegion & r) const
- const QRegion **operator^** (const QRegion & r) const
- QRegion & **operator|=** (const QRegion & r)
- QRegion & **operator+=** (const QRegion & r)
- QRegion & **operator&=** (const QRegion & r)
- QRegion & **operator-=** (const QRegion & r)
- QRegion & **operator^=** (const QRegion & r)
- bool **operator==** (const QRegion & r) const
- bool **operator!=** (const QRegion & r) const
- HRGN **handle** () const

Related Functions

- QDataStream & **operator<<** (QDataStream & s, const QRegion & r)
- QDataStream & **operator>>** (QDataStream & s, QRegion & r)

Detailed Description

The QRegion class specifies a clip region for a painter.

QRegion is used with QPainter::setClipRegion() to limit the paint area to what needs to be painted. There is also a QWidget::repaint() that takes a QRegion parameter. QRegion is the best tool for reducing flicker.

A region can be created from a rectangle, an ellipse, a polygon or a bitmap. Complex regions may be created by combining simple regions using unite(), intersect(), subtract() or eor() (exclusive or). You can move a region using translate().

You can test whether a region isNull(), isEmpty() or if it contains() a QPoint or QRect. The bounding rectangle is given by boundingRect().

The function rects() gives a decomposition of the region into rectangles.

Example of using complex regions:

```
void MyWidget::paintEvent( QPaintEvent * )
{
QPainter p;                              // our painter
QRegion r1( QRect(100,100,200,80),   // r1 = elliptic region
            QRegion::Ellipse );
QRegion r2( QRect(100,120,90,30) ); // r2 = rectangular region
QRegion r3 = r1.intersect( r2 );     // r3 = intersection
p.begin( this );                         // start painting widget
p.setClipRegion( r3 );                   // set clip region
...                                       // paint clipped graphics
p.end();                                  // painting done
}
```

QRegion is an implicitly shared class.

Due to window system limitations, the width and height of a region is limited to 65535 on Unix/X11.

See also QPainter::setClipRegion() [p. 200], QPainter::setClipRect() [p. 200], Graphics Classes and Image Processing Classes.

Member Type Documentation

QRegion::RegionType

Determines the shape of the region to be created.

- `QRegion::Rectangle` - the region covers the entire rectangle.
- `QRegion::Ellipse` - the region is an ellipse inside the rectangle.

Member Function Documentation

QRegion::QRegion ()

Constructs a null region.

See also isNull() [p. 282].

QRegion::QRegion (int x, int y, int w, int h, RegionType t = Rectangle)

Constructs a rectangular or elliptic region.

If t is Rectangle, the region is the filled rectangle (x, y, w, h). If t is Ellipse, the region is the filled ellipse with center at ($x + w / 2$, $y + h / 2$) and size (w ,h).

QRegion::QRegion (const QRect & r, RegionType t = Rectangle)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Create a region based on the rectange r with region type t.

If the rectangle is invalid a null region will be created.

See also QRegion::RegionType [p. 279].

QRegion::QRegion (const QPointArray & a, bool winding = FALSE)

Constructs a polygon region from the point array a.

If *winding* is TRUE, the polygon region is filled using the winding algorithm, otherwise the default even-odd fill algorithm is used.

This constructor may create complex regions that will slow down painting when used.

QRegion::QRegion (const QRegion & r)

Constructs a new region which is equal to r.

QRegion::QRegion (const QBitmap & bm)

Constructs a region from the bitmap *bm*.

The resulting region consists of the pixels in *bm* that are `color1`, as if each pixel was a 1 by 1 rectangle.

This constructor may create complex regions that will slow down painting when used. Note that drawing masked pixmaps can be done much faster using QPixmap::setMask().

QRegion::~QRegion ()

Destroys the region.

QRect QRegion::boundingRect () const

Returns the bounding rectangle of this region. An empty region gives a rectangle that is QRect::isNull().

bool QRegion::contains (const QPoint & p) const

Returns TRUE if the region contains the point *p*, or FALSE if *p* is outside the region.

bool QRegion::contains (const QRect & r) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the region overlaps the rectangle *r*; or FALSE if *r* is completely outside the region.

QRegion QRegion::eor (const QRegion & r) const

Returns a region which is the exclusive or (XOR) of this region and *r*.

The figure shows the exclusive or of two elliptical regions.

HRGN QRegion::handle () const

Returns the region's handle.

QRegion QRegion::intersect (const QRegion & r) const

Returns a region which is the intersection of this region and *r*.

The figure shows the intersection of two elliptical regions.

bool QRegion::isEmpty () const

Returns TRUE if the region is empty, or FALSE if it is non-empty. An empty region is a region that contains no points.

Example:

```
QRegion r1( 10, 10, 20, 20 );
QRegion r2( 40, 40, 20, 20 );
QRegion r3;
r1.isNull();                // FALSE
r1.isEmpty();               // FALSE
r3.isNull();                // TRUE
r3.isEmpty();               // TRUE
r3 = r1.intersect( r2 );    // r3 = intersection of r1 and r2
r3.isNull();                // FALSE
r3.isEmpty();               // TRUE
```

```
    r3 = r1.unite( r2 );          // r3 = union of r1 and r2
    r3.isNull();                  // FALSE
    r3.isEmpty();                 // FALSE
```

See also isNull() [p. 282].

bool QRegion::isNull () const

Returns TRUE if the region is a null region, otherwise FALSE.

A null region is a region that has not been initialized. A null region is always empty.

See also isEmpty() [p. 281].

bool QRegion::operator!= (const QRegion & r) const

Returns TRUE if the region is different from *r*, or FALSE if the regions are equal.

const QRegion QRegion::operator& (const QRegion & r) const

Applies the intersect() function to this region and *r*. `r1&r2` is equivalent to `r1.intersect(r2)`

See also intersect() [p. 281].

QRegion & QRegion::operator&= (const QRegion & r)

Applies the intersect() function to this region and *r* and assigns the result to this region. `r1&=r2` is equivalent to `r1=r1.intersect(r2)`

See also intersect() [p. 281].

const QRegion QRegion::operator+ (const QRegion & r) const

Applies the unite() function to this region and *r*. `r1+r2` is equivalent to `r1.unite(r2)`

See also unite() [p. 284] and operator|() [p. 283].

QRegion & QRegion::operator+= (const QRegion & r)

Applies the unite() function to this region and *r* and assigns the result to this region. `r1+=r2` is equivalent to `r1=r1.unite(r2)`

See also intersect() [p. 281].

const QRegion QRegion::operator- (const QRegion & r) const

Applies the subtract() function to this region and *r*. `r1-r2` is equivalent to `r1.subtract(r2)`

See also subtract() [p. 283].

QRegion & QRegion::operator-= (const QRegion & r)

Applies the subtract() function to this region and *r* and assigns the result to this region. `r1-=r2` is equivalent to `r1=r1.subtract(r2)`

See also subtract() [p. 283].

QRegion & QRegion::operator= (const QRegion & r)

Assigns *r* to this region and returns a reference to the region.

bool QRegion::operator== (const QRegion & r) const

Returns TRUE if the region is equal to *r*, or FALSE if the regions are different.

const QRegion QRegion::operator^ (const QRegion & r) const

Applies the eor() function to this region and *r*. `r1^r2` is equivalent to `r1.eor(r2)`

See also eor() [p. 281].

QRegion & QRegion::operator^= (const QRegion & r)

Applies the eor() function to this region and *r* and assigns the result to this region. `r1^=r2` is equivalent to `r1=r1.eor(r2)`

See also eor() [p. 281].

const QRegion QRegion::operator| (const QRegion & r) const

Applies the unite() function to this region and *r*. `r1|r2` is equivalent to `r1.unite(r2)`

See also unite() [p. 284] and operator+() [p. 282].

QRegion & QRegion::operator|= (const QRegion & r)

Applies the unite() function to this region and *r* and assigns the result to this region. `r1|=r2` is equivalent to `r1=r1.unite(r2)`

See also unite() [p. 284].

QMemArray<QRect> QRegion::rects () const

Returns an array of non-overlapping rectangles that make up the region.

The union of all the rectangles is equal to the original region.

QRegion QRegion::subtract (const QRegion & r) const

Returns a region which is *r* subtracted from this region.
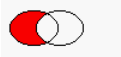
The figure shows the result when the ellipse on the right is subtracted from the ellipse on the left. (`left-right`)

void QRegion::translate (int dx, int dy)

Translates (moves) the region *dx* along the X axis and *dy* along the Y axis.

QRegion QRegion::unite (const QRegion & r) const

Returns a region which is the union of this region and *r*.

The figure shows the union of two elliptical regions.

Related Functions

QDataStream & operator<< (QDataStream & s, const QRegion & r)

Writes the region *r* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QRegion & r)

Reads a region from the stream *s* into *r* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QSize Class Reference

The QSize class defines the size of a two-dimensional object.

```
#include <qsize.h>
```

Public Members

- **QSize** ()
- **QSize** (int w, int h)
- bool **isNull** () const
- bool **isEmpty** () const
- bool **isValid** () const
- int **width** () const
- int **height** () const
- void **setWidth** (int w)
- void **setHeight** (int h)
- void **transpose** ()
- QSize **expandedTo** (const QSize & otherSize) const
- QSize **boundedTo** (const QSize & otherSize) const
- QCOORD & **rwidth** ()
- QCOORD & **rheight** ()
- QSize & **operator+=** (const QSize & s)
- QSize & **operator-=** (const QSize & s)
- QSize & **operator*=** (int c)
- QSize & **operator*=** (double c)
- QSize & **operator/=** (int c)
- QSize & **operator/=** (double c)

Related Functions

- bool **operator==** (const QSize & s1, const QSize & s2)
- bool **operator!=** (const QSize & s1, const QSize & s2)
- const QSize **operator+** (const QSize & s1, const QSize & s2)
- const QSize **operator-** (const QSize & s1, const QSize & s2)
- const QSize **operator*** (const QSize & s, int c)
- const QSize **operator*** (int c, const QSize & s)
- const QSize **operator*** (const QSize & s, double c)
- const QSize **operator*** (double c, const QSize & s)
- const QSize **operator/** (const QSize & s, int c)

- const QSize **operator/** (const QSize & s, double c)
- QDataStream & **operator<<** (QDataStream & s, const QSize & sz)
- QDataStream & **operator>>** (QDataStream & s, QSize & sz)

Detailed Description

The QSize class defines the size of a two-dimensional object.

A size is specified by a width and a height.

The coordinate type is QCOORD (defined in qwindowdefs.h as `int`). The minimum value of QCOORD is QCO-ORD_MIN (-2147483648) and the maximum value is QCOORD_MAX (2147483647).

The size can be set in the constructor and changed with setWidth() and setHeight(), or using operator+=(), operator-=(), operator*=() and operator/=(), etc. You can swap the width and height with transpose(). You can get a size which holds the maximum height and width of two sizes using expandedTo(), and the minimum height and width of two sizes using boundedTo().

See also QPoint [p. 242], QRect [p. 266], Graphics Classes and Image Processing Classes.

Member Function Documentation

QSize::QSize ()

Constructs a size with invalid (negative) width and height.

QSize::QSize (int w, int h)

Constructs a size with width *w* and height *h*.

QSize QSize::boundedTo (const QSize & otherSize) const

Returns a size with the minimum width and height of this size and *otherSize*.

QSize QSize::expandedTo (const QSize & otherSize) const

Returns a size with the maximum width and height of this size and *otherSize*.

Examples: customlayout/card.cpp and customlayout/flow.cpp.

int QSize::height () const

Returns the height.

See also width() [p. 288].

Examples: movies/main.cpp, qfd/fontdisplayer.cpp and qfd/qfd.cpp.

bool QSize::isEmpty () const

Returns TRUE if the width is <= 0 or the height is <= 0, otherwise FALSE.

bool QSize::isNull () const

Returns TRUE if the width is 0 and the height is 0; otherwise returns FALSE.

bool QSize::isValid () const

Returns TRUE if the width is equal to or greater than 0 and the height is equal to or greater than 0; otherwise returns FALSE.

QSize & QSize::operator*= (int c)

Multiplies both the width and height by *c* and returns a reference to the size.

QSize & QSize::operator*= (double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Multiplies both the width and height by *c* and returns a reference to the size.

Note that the result is truncated.

QSize & QSize::operator+= (const QSize & s)

Adds *s* to the size and returns a reference to this size.

Example:

```
QSize s(  3, 7 );
QSize r( -1, 4 );
s += r;                    // s becomes (2,11)
```

QSize & QSize::operator-= (const QSize & s)

Subtracts *s* from the size and returns a reference to this size.

Example:

```
QSize s(  3, 7 );
QSize r( -1, 4 );
s -= r;                    // s becomes (4,3)
```

QSize & QSize::operator/= (int c)

Divides both the width and height by *c* and returns a reference to the size.

QSize & QSize::operator/= (double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Divides both the width and height by *c* and returns a reference to the size.

Note that the result is truncated.

QCOORD & QSize::rheight ()

Returns a reference to the height.

Using a reference makes it possible to directly manipulate the height.

Example:

```
QSize s( 100, 10 );
s.rheight() += 5;           // s becomes (100,15)
```

See also rwidth() [p. 288].

QCOORD & QSize::rwidth ()

Returns a reference to the width.

Using a reference makes it possible to directly manipulate the width.

Example:

```
QSize s( 100, 10 );
s.rwidth() += 20;           // s becomes (120,10)
```

See also rheight() [p. 288].

void QSize::setHeight (int h)

Sets the height to *h*.

See also height() [p. 286] and setWidth() [p. 288].

void QSize::setWidth (int w)

Sets the width to *w*.

See also width() [p. 288] and setHeight() [p. 288].

void QSize::transpose ()

Swaps the values of width and height.

int QSize::width () const

Returns the width.

See also height() [p. 286].

Examples: movies/main.cpp, qfd/fontdisplayer.cpp and qfd/qfd.cpp.

Related Functions

bool operator!= (const QSize & s1, const QSize & s2)

Returns TRUE if *s1* and *s2* are different; otherwise returns FALSE.

const QSize operator* (const QSize & s, int c)

Multiplies *s* by *c* and returns the result.

const QSize operator* (int c, const QSize & s)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. Multiplies *s* by *c* and returns the result.

const QSize operator* (const QSize & s, double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. Multiplies *s* by *c* and returns the result.

const QSize operator* (double c, const QSize & s)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. Multiplies *s* by *c* and returns the result.

const QSize operator+ (const QSize & s1, const QSize & s2)

Returns the sum of *s1* and *s2*; each component is added separately.

const QSize operator- (const QSize & s1, const QSize & s2)

Returns *s2* subtracted from *s1*; each component is subtracted separately.

const QSize operator/ (const QSize & s, int c)

Divides *s* by *c* and returns the result.

const QSize operator/ (const QSize & s, double c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. Divides *s* by *c* and returns the result.

Note that the result is truncated.

QDataStream & operator<< (QDataStream & s, const QSize & sz)

Writes the size *sz* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

bool operator== (const QSize & s1, const QSize & s2)

Returns TRUE if *s1* and *s2* are equal; otherwise returns FALSE.

QDataStream & operator>> (QDataStream & s, QSize & sz)

Reads the size from the stream *s* into size *sz* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QWMatrix Class Reference

The QWMatrix class specifies 2D transformations of a coordinate system.

`#include <qwmatrix.h>`

Public Members

- **QWMatrix** ()
- **QWMatrix** (double m11, double m12, double m21, double m22, double dx, double dy)
- void **setMatrix** (double m11, double m12, double m21, double m22, double dx, double dy)
- double **m11** () const
- double **m12** () const
- double **m21** () const
- double **m22** () const
- double **dx** () const
- double **dy** () const
- void **map** (int x, int y, int * tx, int * ty) const
- void **map** (double x, double y, double * tx, double * ty) const
- QRect **mapRect** (const QRect & rect) const
- QPoint map (const QPoint & p) const *(obsolete)*
- QRect map (const QRect & r) const *(obsolete)*
- QPointArray map (const QPointArray & a) const *(obsolete)*
- void **reset** ()
- bool **isIdentity** () const
- QWMatrix & **translate** (double dx, double dy)
- QWMatrix & **scale** (double sx, double sy)
- QWMatrix & **shear** (double sh, double sv)
- QWMatrix & **rotate** (double a)
- bool **isInvertible** () const
- QWMatrix **invert** (bool * invertible = 0) const
- bool **operator==** (const QWMatrix & m) const
- bool **operator!=** (const QWMatrix & m) const
- QWMatrix & **operator*=** (const QWMatrix & m)
- QPoint **operator*** (const QPoint & p) const
- QRegion **operator*** (const QRect & r) const
- QRegion **operator*** (const QRegion & r) const
- QPointArray **operator*** (const QPointArray & a) const

Related Functions

- QWMatrix **operator*** (const QWMatrix & m1, const QWMatrix & m2)
- QDataStream & **operator<<** (QDataStream & s, const QWMatrix & m)
- QDataStream & **operator>>** (QDataStream & s, QWMatrix & m)

Detailed Description

The QWMatrix class specifies 2D transformations of a coordinate system.

The standard coordinate system of a paint device has the origin located at the top-left position. X values increase to the right; Y values increase downward.

This coordinate system is default for the QPainter, which renders graphics in a paint device. A user-defined coordinate system can be specified by setting a QWMatrix for the painter.

Example:

```
MyWidget::paintEvent( QPaintEvent * )
{
  QPainter p;                        // our painter
  QWMatrix m;                        // our transformation matrix
  m.rotate( 22.5 );                  // rotated coordinate system
  p.begin( this );                   // start painting
  p.setWorldMatrix( m );             // use rotated coordinate system
  p.drawText( 30,20, "detator" );    // draw rotated text at 30,20
  p.end();                           // painting done
}
```

A matrix specifies how to translate, scale, shear or rotate the graphics; the actual transformation is performed by the drawing routines in QPainter and by QPixmap::xForm().

The QWMatrix class contains a 3*3 matrix of the form:

```
m11   m12   0
m21   m22   0
dx    dy    1
```

A matrix transforms a point in the plane to another point:

```
x' = m11*x + m21*y + dx
y' = m22*y + m12*x + dy
```

The point *(x, y)* is the original point, and *(x', y')* is the transformed point. *(x', y')* can be transformed back to *(x, y)* by performing the same operation on the inverted matrix.

The elements *dx* and *dy* specify horizontal and vertical translation. The elements *m11* and *m22* specify horizontal and vertical scaling. The elements *m12* and *m21* specify horizontal and vertical shearing.

The identity matrix has *m11* and *m22* set to 1; all others are set to 0. This matrix maps a point to itself.

Translation is the simplest transformation. Setting *dx* and *dy* will move the coordinate system *dx* units along the X axis and *dy* units along the Y axis.

Scaling can be done by setting *m11* and *m22*. For example, setting *m11* to 2 and *m22* to 1.5 will double the height and increase the width by 50%.

Shearing is controlled by *m12* and *m21*. Setting these elements to values different from zero will twist the coordinate system.

Rotation is achieved by carefully setting both the shearing factors and the scaling factors. The QWMatrix has a function that sets rotation directly.

QWMatrix lets you combine transformations like this:

```
QWMatrix m;              // identity matrix
m.translate(10, -20);    // first translate (10,-20)
m.rotate(25);            // then rotate 25 degrees
m.scale(1.2, 0.7);       // finally scale it
```

Here's the same example using basic matrix operations:

```
double a    = pi/180 * 25;          // convert 25 to radians
double sina = sin(a);
double cosa = cos(a);
QWMatrix m1(0, 0, 0, 0, 10, -20);   // translation matrix
QWMatrix m2( cosa, sina,            // rotation matrix
            -sina, cosa, 0, 0 );
QWMatrix m3(1.2, 0, 0, 0.7, 0, 0);  // scaling matrix
QWMatrix m;
m = m3 * m2 * m1;                   // combine all transformations
```

QPainter has functions to translate, scale, shear and rotate the coordinate system without using a QWMatrix. Although these functions are very convenient, it can be more efficient to build a QWMatrix and call QPainter::setWorldMatrix() if you want to perform more than a single transform operation.

See also QPainter::setWorldMatrix() [p. 203], QPixmap::xForm() [p. 237], Graphics Classes and Image Processing Classes.

Member Function Documentation

QWMatrix::QWMatrix ()

Constructs an identity matrix. All elements are set to zero except *m11* and *m22* (scaling), which are set to 1.

QWMatrix::QWMatrix (double m11, double m12, double m21, double m22, double dx, double dy)

Constructs a matrix with the elements, *m11*, *m12*, *m21*, *m22*, *dx* and *dy*.

double QWMatrix::dx () const

Returns the horizontal translation.

double QWMatrix::dy () const

Returns the vertical translation.

QWMatrix QWMatrix::invert (bool * invertible = 0) const

Returns the inverted matrix.

If the matrix is singular (not invertible), the identity matrix is returned.

If *invertible* is not null, the value of **invertible* is set to TRUE if the matrix is invertible or to FALSE if the matrix is not invertible.

See also isInvertible() [p. 294].

Example: t14/cannon.cpp.

bool QWMatrix::isIdentity () const

Returns TRUE if the matrix is the identity matrix; otherwise returns FALSE.

See also reset() [p. 296].

bool QWMatrix::isInvertible () const

Returns TRUE if the matrix is invertible; otherwise returns FALSE.

See also invert() [p. 293].

double QWMatrix::m11 () const

Returns the X scaling factor.

double QWMatrix::m12 () const

Returns the vertical shearing factor.

double QWMatrix::m21 () const

Returns the horizontal shearing factor.

double QWMatrix::m22 () const

Returns the Y scaling factor.

void QWMatrix::map (int x, int y, int * tx, int * ty) const

Transforms (*x, y*) to (**tx, *ty*) using the formulae:

```
*tx = m11*x + m21*y + dx   (rounded to the nearest integer)
*ty = m22*y + m12*x + dy   (rounded to the nearest integer)
```

Examples: t14/cannon.cpp and xform/xform.cpp.

void QWMatrix::map (double x, double y, double * tx, double * ty) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms (*x, y*) to (**tx, *ty*) using the following formulae:

```
    *tx = m11*x + m21*y + dx
    *ty = m22*y + m12*x + dy
```

QPoint QWMatrix::map (const QPoint & p) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Does the same as operator *(const QPoint &)

QRect QWMatrix::map (const QRect & r) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Please use QWMatrix::mapRect() instead.

Note that this method does return the bounding rectangle of the *r*, when shearing or rotations are used.

QPointArray QWMatrix::map (const QPointArray & a) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Does the same as operator *(const QPointArray &)

QRect QWMatrix::mapRect (const QRect & rect) const

Returns the transformed rectangle *rect*.

The bounding rectangle is returned if rotation or shearing has been specified.

If you need to know the exact region *rect* maps to use operator*().

See also operator*() [p. 295].

bool QWMatrix::operator!= (const QWMatrix & m) const

Returns TRUE if this matrix is not equal to *m*; otherwise returns FALSE.

QPoint QWMatrix::operator* (const QPoint & p) const

Transforms *p* to using the formulae:

```
    retx = m11*px + m21*py + dx  (rounded to the nearest integer)
    rety = m22*py + m12*px + dy  (rounded to the nearest integer)
```

QRegion QWMatrix::operator* (const QRect & r) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms the rectangle *r*.

Rotation and shearing a rectangle results in a more general region, which is returned here.

Calling this method can be rather expensive, if rotations or shearing are used. If you just need to know the bounding rectangle of the returned region, use mapRect() which is a lot faster than this function.

See also QWMatrix::mapRect() [p. 295].

QRegion QWMatrix::operator* (const QRegion & r) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms the region *r*.

Calling this method can be rather expensive, if rotations or shearing are used.

QPointArray QWMatrix::operator* (const QPointArray & a) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the point array *a* transformed by calling map for each point.

QWMatrix & QWMatrix::operator*= (const QWMatrix & m)

Returns the result of multiplying this matrix with matrix *m*.

bool QWMatrix::operator== (const QWMatrix & m) const

Returns TRUE if this matrix is equal to *m*; otherwise returns FALSE.

void QWMatrix::reset ()

Resets the matrix to an identity matrix.

All elements are set to zero, except *m11* and *m22* (scaling) that are set to 1.

See also isIdentity() [p. 294].

QWMatrix & QWMatrix::rotate (double a)

Rotates the coordinate system *a* degrees counterclockwise.

Returns a reference to the matrix.

See also translate() [p. 297], scale() [p. 296] and shear() [p. 297].

Examples: desktop/desktop.cpp, drawdemo/drawdemo.cpp, t14/cannon.cpp and xform/xform.cpp.

QWMatrix & QWMatrix::scale (double sx, double sy)

Scales the coordinate system unit by *sx* horizontally and *sy* vertically.

Returns a reference to the matrix.

See also translate() [p. 297], shear() [p. 297] and rotate() [p. 296].

Examples: fileiconview/qfileiconview.cpp, movies/main.cpp, qmag/qmag.cpp, qtimage/qtimage.cpp, showimg/showimg.cpp and xform/xform.cpp.

void QWMatrix::setMatrix (double m11, double m12, double m21, double m22, double dx, double dy)

Sets the matrix elements to the specified values, *m11, m12, m21, m22, dx* and *dy*.

QWMatrix & QWMatrix::shear (double sh, double sv)

Shears the coordinate system by *sh* horizontally and *sv* vertically.

Returns a reference to the matrix.

See also translate() [p. 297], scale() [p. 296] and rotate() [p. 296].

Examples: drawdemo/drawdemo.cpp and xform/xform.cpp.

QWMatrix & QWMatrix::translate (double dx, double dy)

Moves the coordinate system *dx* along the X-axis and *dy* along the Y-axis.

Returns a reference to the matrix.

See also scale() [p. 296], shear() [p. 297] and rotate() [p. 296].

Examples: drawdemo/drawdemo.cpp, t14/cannon.cpp and xform/xform.cpp.

Related Functions

QWMatrix operator* (const QWMatrix & m1, const QWMatrix & m2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the product of *m1 * m2*.

Note that matrix multiplication is not commutative, i.e. a*b != b*a.

QDataStream & operator<< (QDataStream & s, const QWMatrix & m)

Writes the matrix *m* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QWMatrix & m)

Reads the matrix *m* from the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

Index