

Dessiner avec Qt4

<http://doc.trolltech.com/4.3/paintsystem.html>

Véronique Lefrere

| | |
|---|----|
| Dessiner | 2 |
| 1. Quoi ? | 2 |
| Un (ou plusieurs) point(s) sur un plan donné | 2 |
| Des segments | 2 |
| Un polygone | 3 |
| Un rectangle | 3 |
| Un rectangles (angles arrondis) | 3 |
| Une Ellipse | 3 |
| Un Arc de cercle | 3 |
| Une forme circulaire | 3 |
| Une corde | 4 |
| Un chemin | 4 |
| Une région ou une zone | 4 |
| Du texte | 4 |
| Une pixmap ou une image | 4 |
| 2. Sur quoi? | 5 |
| Le support | 5 |
| La region | 5 |
| 3. Comment ? | 7 |
| Les pré-réglages de dessin | 7 |
| La transparence | 8 |
| L'Anti-crènelage (anti-aliasing) | 8 |
| Les lignes et contours | 9 |
| Le remplissage | 11 |
| Avec du style | 17 |
| Du texte haut en couleur | 17 |
| Le Système de coordonnées | 18 |
| La Conversion des coordonnées logiques en coordonnées physiques (et inversement) .. | 20 |

Les 3 complices



Qpainter : outil de dessin

QpaintDevice : Abstraction d'un Espace 2D dans lequel on peut dessiner(*)

QpaintEngine : interface utilisée par Qpainter pour dessiner sur un QpaintDevice

(*) le QpaintDevice est utilisé par Qpainter et QpaintEngine. Cette classe n'est utilisée par le développeur que dans le cas où celui-ci crée sa propre abstraction.

Ces trois compères forment une seule et même entité prénommée [Arthur](#)

Dessiner

1. Quoi ?

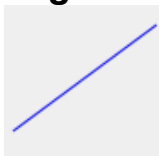
Qpainter vous facilite le dessin de formes simples

(<http://doc.trolltech.com/4.3/html/painting-basicdrawing.html>)

Un (ou plusieurs) point(s) sur un plan donné

QPainter::drawPoint(s)

Une ligne



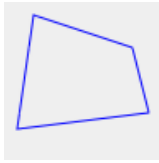
```
QPointF line(10.0, 80.0, 90.0, 20.0);  
QPainter painter(this);  
painter.drawLine(line);
```

Des segments



```
static const QPointF points[3] = {  
    QPointF(10.0, 80.0), QPointF(20.0, 10.0), QPointF(80.0, 30.0), };  
QPainter painter(this);  
painter.drawPolyline(points, 3);
```

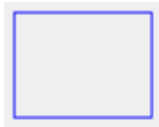
Un polygone



```
static const QPointF points[4] = {
    QPointF(10.0, 80.0),QPointF(20.0, 10.0),
    QPointF(80.0, 30.0),QPointF(90.0,
70.0)};
    QPainter painter(this);
    painter.drawPolygon(points, 4);
```

(ie [QPainter::drawConvexPolygon](#))

Un rectangle



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
QPainter painter(this);
painter.drawRect(rectangle);
```

Un rectangles (angles arrondis)



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
QPainter painter(this);
painter.drawRoundRect(rectangle);
```

Une Ellipse



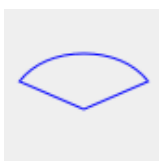
```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
QPainter painter(this);
painter.drawEllipse(rectangle);
```

Un Arc de cercle



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;
QPainter painter(this);
painter.drawArc(rectangle, startAngle, spanAngle);
```

Une forme circulaire



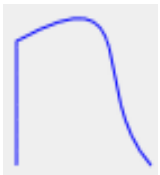
```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;
QPainter painter(this);
painter.drawPie(rectangle, startAngle, spanAngle);
```

Une corde



```
QRectF rect(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;
QPainter painter(this);
painter.drawChord(rect, startAngle, spanAngle);
```

Un chemin



```
QPainterPath path;
path.moveTo(20, 80);
path.lineTo(20, 30);
path.cubicTo(80, 0, 50, 50, 80, 80);
QPainter painter(this);
painter.drawPath(path);
```

<http://doc.trolltech.com/4.3/painting-painterpaths.html>

Une région ou une zone

QPainter::setClipRegion, QPainter::setClipPath



```
QPainterPath starPath;
starPath.moveTo(90, 50);
for (int i = 1; i < 5; ++i) {
    starPath.lineTo(50 + 40 * cos(0.8 * i * Pi),
                    50 + 40 * sin(0.8 * i * Pi));
}
starPath.closeSubpath();
```

<http://doc.trolltech.com/4.3/qpainterpath.html>

Du texte



```
QPainter painter(this);
painter.drawText(rect, Qt::AlignCenter, tr("Qt
by\nTrolltech"));
```

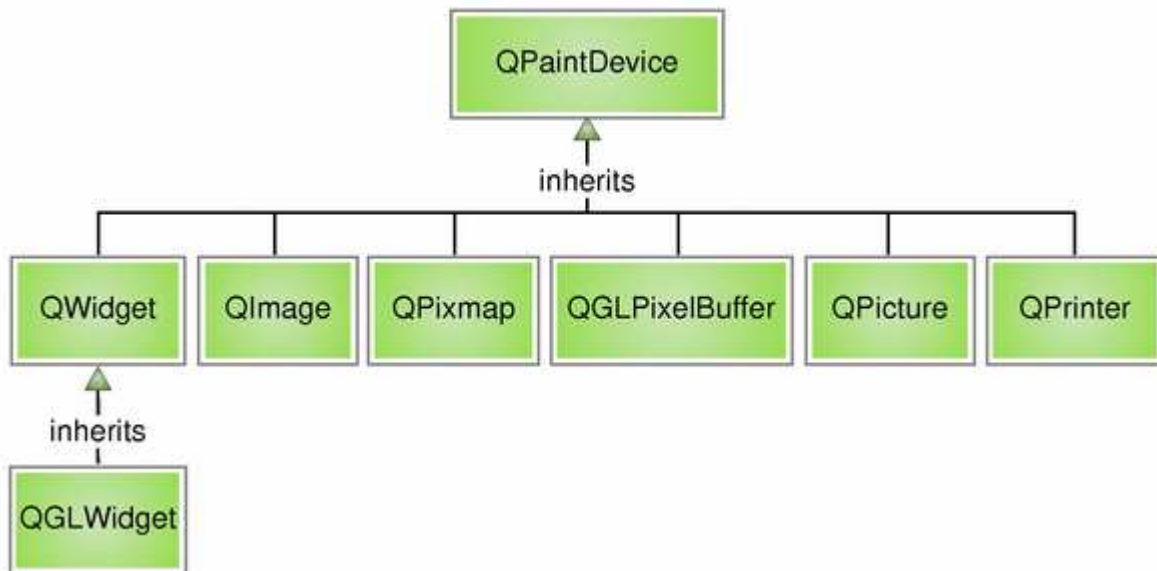
Une pixmap ou une image



```
QPainter painter(this);
QPixmap pixmap(":/qt-logo.png");
painter.drawPixmap(10, 10, pixmap);
ou QImage pimage(":/qt-logo.png");
painter.drawImage(QPoint(0, 0), pimage);
```

2. Sur quoi?

Le support



QpaintDevice est la classe de base de tout support de dessin.

Le schéma ci-dessus donne la liste des classes de base sur lesquelles il est possible de dessiner.

QWidget est LA classe de base des widgets « équipés » pour le dessin, l'affichage dans différents formats d'image 2D ou 3D ([QGLWidget](#)) ou encore l'impression (QPrinter)

La region

QRegion est une classe qui, utilisée avec QPainter (cf [QPainter::setClipRegion\(\)](#)), permet d'optimiser les opérations de dessin en délimitant la zone à (re)-dessiner.

L'utilisation de cette classe contribue à la réduction du phénomène de clignotement (flicker) lors d'opération de rafraîchissement fréquents d'une zone de dessin.

Une région peut être définie en tant que rectangle (QRegion::Polygon), ellipse (QRegion::Ellipse), polygone () ou Bitmap.

On peut contrôler qu'une région contient un dessin ou non (QRegion::isEmpty())

On peut vérifier si cette région contient un point (Qpoint) ou une autre région.

Cette région peut être déplacée (translate()).

Elle peut être la résultante de :

- l'intersection de deux régions (QRegion::intersect)



```
void MyWidget::paintEvent(QPaintEvent *)
{
    QPainter p;           // our painter
    QRegion r1(QRect(100,100,200,80), QRegion::Ellipse); // r1 = elliptic region
    QRegion r2(QRect(100,120,90,30), QRegion::Ellipse); // r2 = elliptic 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
}
```

- une combinaison de deux régions (QRegion::unite)



- leur soustraction (QRegion::subtract)



- leur exclusion (QRegion::xor)



On peut également utiliser cette classe pour déterminer l'appartenance ou non d'un point dans une région. QPainter::contains(const QPoint& p)

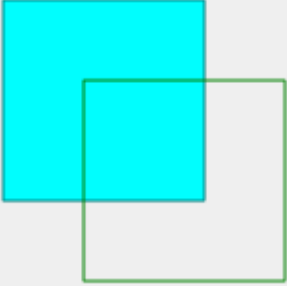
3. Comment ?

Les pré-réglages de dessin

La classe QPainter fournit un ensemble de fonctions destinées au paramétrage des actions de dessin.

- [font\(\)](#)
utilisée pour spécifier la police de caractères utilisée pour dessiner du texte.
- [brush\(\)](#)
couleur ou motif de remplissage de la forme dessinée.
- [pen\(\)](#)
couleur ou moucheté utilisés pour dessiner des lignes ou contours.
- [backgroundMode\(\)](#)
indique si il y a ou non un fond ([Qt::Opaque](#) - [Qt::TransparentMode](#))
- [background\(\)](#)
indique la couleur des pixels de mouchetage
valable en cas de fond [Qt::Opaque](#) moucheté.
- [brushOrigin\(\)](#)
défini l'origine du fond.
valable dans le cas de motifs (brush pattern styled) ou pixmap.
- [viewport\(\)](#), [window\(\)](#), [matrix\(\)](#)
constituent le système de transformation de coordonnées de dessin.
cf [Coordinate Transformations](#) et [The Coordinate System documentation](#).
- [hasClipping\(\)](#)
indique que le QPainter est appliqué sur région déterminée.
- [layoutDirection\(\)](#)
définis la direction de dessin du texte
- [matrixEnabled\(\)](#)
indique que les transformations de coordonnées sont possibles.
cf <http://doc.trolltech.com/4.3/qpainter.html> - coordinate-transformations.
- [viewTransformEnabled\(\)](#)
indique que la conversion de vue est possible.
cf <http://doc.trolltech.com/4.3/coordsys.html> - window-viewport-conversion

La transparence

| | |
|---|--|
|  | <pre>QPainter painter(this); painter.setBrush(Qt::cyan); painter.setPen(Qt::darkCyan); painter.drawRect(0, 0, 100,100); painter.setBrush(Qt::NoBrush); painter.setPen(Qt::darkGreen); painter.drawRect(40, 40, 100, 100);</pre> |
|---|--|

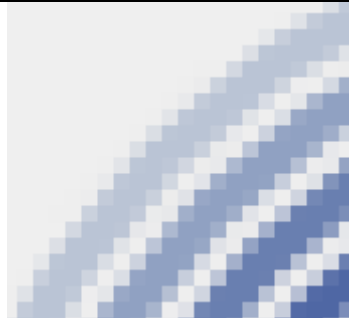
NB : à noter que par défaut le contour de l'objet du painter est dessiné (drawrect) en utilisant la couleur déterminée par setpen.

Dans le cas où on ne veut pas de contour : [painter.setPen\(Qt::NoPen\)](#).

La gestion de la transparence (alpha blended support) peut également s'opérer au niveau de la gestion du remplissage pour combiner deux couleurs de remplissage.

(voir le chapitre « La couleur de remplissage »).

L'Anti-crènelage (*anti-aliasing*)

| | |
|---|---|
|  | <p>La qualité de rendu du pixel lors du dessin peut être contrôlée via l'utilisation de la constante <code>QPainter::Antialiasing</code>.</p> <p>L'appel de la fonction <code>QPainter::setRenderHint(QPainter::Antialiasing, true)</code> va déclencher l'usage de couleurs proches de différente intensité afin d'arrondir les angles des primitives dessinées.</p> <p>(cf http://doc.trolltech.com/4.3/painting-concentriccircles.html)</p> |
|---|---|

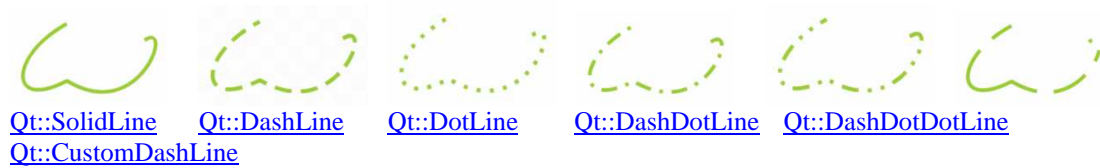
Il est également possible d'améliorer le rendu d'un texte [QPainter::TextAntialiasing](#) ou encore de faire appel à un algorithme de lissage [QPainter::SmoothPixmapTransform](#).

Les lignes et contours

Pour ce qui est des contours arrondis d'un rectangle, Qt nous a mûché le travail en fournissant la fonction `QPainter::drawRoundRectangle`.

Reste que, si l'on souhaite maîtriser style, couleur, épaisseur, jointures ou terminaison des contours et des traits, la classe [QPen](#) est notre amie.

`QPen::setStyle (Qt::PenStyle style)`



Le dernier enum permet de créer son propre style.

Il suffit de fournir un vecteur comprend en &, », (,.... La taille du segment et en 2,4,6,.... La taille de l'espace. Pour obtenir l'illustration ci-dessus on a donc le code suivant :

```
QPen pen;
QVector<qreal> dashes;
qreal space = 4;
dashes << 1 << space << 3 << space << 9 << space << 27 << space << 9;
pen.setDashPattern(dashes);
```

`QPen::setCapStyle((Qt::PenCapStyle style)`



QPen::setJoinStyle ([Qt::PenJoinStyle style](#))



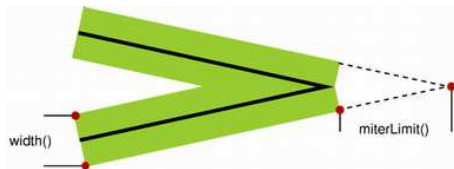
Qt::BevelJoin

Qt::MiterJoin

Qt::RoundJoin

Dans le cas de l'utilisation du style MiterJoin, il est possible de préciser la limite de l'onglet
QPen::setMiterLimit ([qreal limit](#)). Le paramètre *limit* est spécifié en unité d'épaisseur du
QPen.

Par exemple, une limite de 5 pour une épaisseur de 10 représente 50 pixels de long.



La valeur limite de jointure par défaut est de 2 (2 fois l'épaisseur par défaut).

Par défaut, un stylo (QPen) est un segment noir d'épaisseur 0 se finissant de façon carrée ([Qt::SquareCap](#)) et dont la jonction entre deux segments est biseautée ([Qt::BevelJoin](#)).

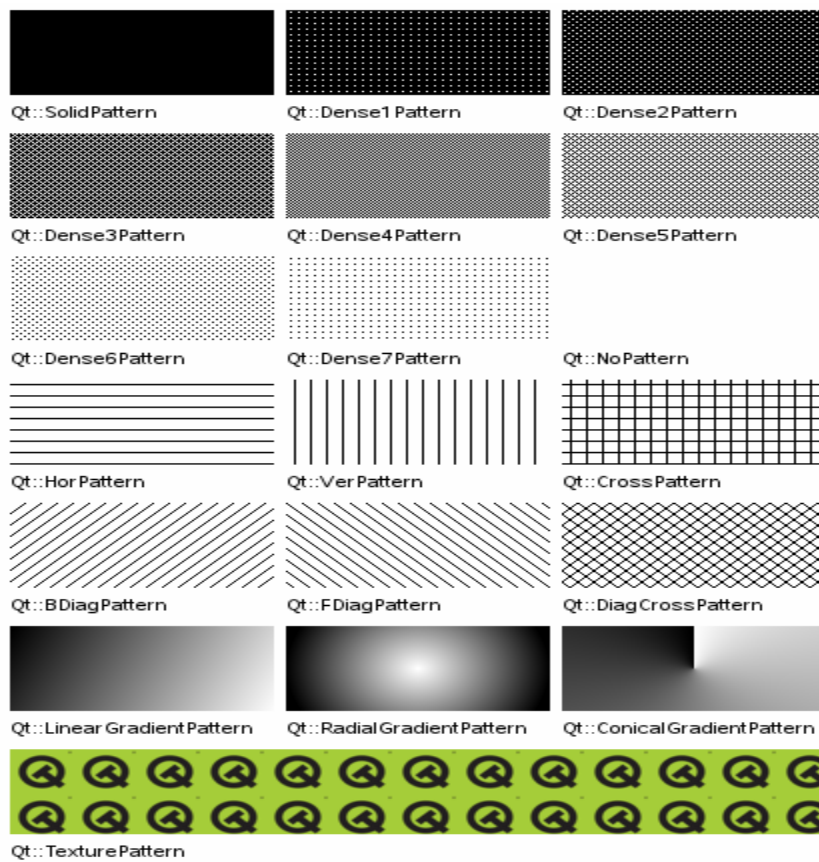
Le remplissage

La définition du remplissage d'une zone dessinée est caractérisée par deux informations majeures : le style de remplissage et la couleur de remplissage.

Le style de remplissage:

QBrush::setStyle ([Qt::BrushStyle](#) style)

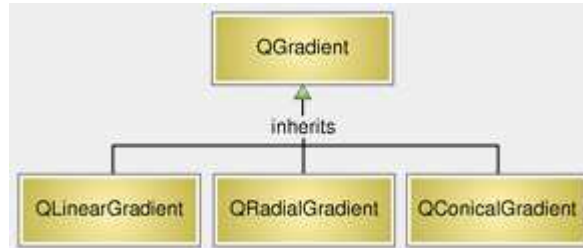
Tableau 1: erratum Qt::NoPattern lire Qt::NoBrush



Les Dégradés

QGradient

Le remplissage de type gradient combiné avec les 3 classes dérivées de QGradient permet d'obtenir un remplissage linéaire, circulaire ou conique d'une couleur donnée



3 types de dégradés sont proposés de base .

La classe de base QGradient est combinée à QBrush pour définir le dégradé.

La (ou les) couleur(s) du dégradé sont définies soit une à une (setColorAt) en indiquant la couleur et la position de début d'utilisation de cette couleur dans le dégradé.

On peut également préciser en un seul appel la liste de couleurs et leur position respective en fournissant une liste de points (couleur et position) dans la structure [QGradientStop](#) à la fonction setStops. Par défaut, en l'absence de point définit le dégradé ira du noir au blanc

- **Linear** - interpolation des couleurs entre les points de départ et d'arrivée.



```

QRect rect(10, 20, 80, 60);
QPainter painter(this);
painter.setPen(pen);
painter.setBrush(brush);
QLinearGradient linearGradient(0, 0, 100, 100);
linearGradient.setColorAt(0.0, Qt::white);
linearGradient.setColorAt(0.2, Qt::green);
linearGradient.setColorAt(1.0, Qt::black);
renderArea->setBrush(linearGradient);
painter.drawEllipse(rect); OU painter.drawRect(rect);
  
```

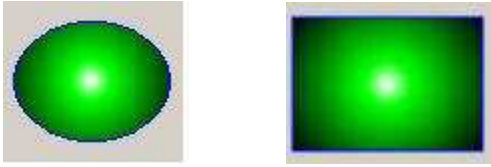
Revenons sur la ligne d'instanciation du dégradé linéaire :

QLinearGradient linearGradient(0, 0, 100, 100);

Les quatre paramètres délimitent un carré de 100x100 pixels positionnée en (0,0).

L'appel à `setColorAt` définit un dégradé allant du blanc vers le noir en passant par du vert. La progression du dégradé étant induite par l'indice associé à chaque couleur. La position (0,0) de l'objet gradient indique que le dégradé débute au coin haut gauche de l'objet graphique dessiné.

- **Radial** - interpolation des couleurs entre un point central et les points situés sur le pourtour d'une ellipse entourant la zone dessinée.



Pour obtenir ce type de dégradé nous remplaçons la ligne d'instanciation du dégradé par :

```
QRadialGradient radialGradient(40,30,40,40,30);
```

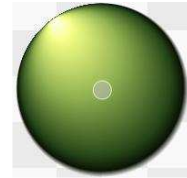
Les deux premiers paramètres définissent les coordonnées x/y du centre.

Le troisième paramètre indique le rayon.

Les deux derniers paramètres définissent la focale qui, ci-dessus, coïncide avec le centre.

L'absence des deux derniers paramètres auraient pour conséquence de déplacer la focale vers 0,0 et donc d'obtenir un dégradé circulaire « attiré » par le point (0,0) et partant en direction du coin bas gauche de la zone dessinée.

```
QRadialGradient radialGradient(40,30,40);
```



- **Conical** – interpolation des couleurs autour d'un point central.



Le dégradé conique ne demande qu'un centre et un angle de départ :

```
QConicalGradient conicalGradient(40,30,40,180);
```

L'angle est exprimé entre 0 et 360 degrés dans le sens trigonométrique (sens inverse des aiguilles d'une montre).

Tips !!

Pour éviter une rupture franche entre la première et la dernière couleur, il suffit de répéter la première couleur en dernière position et de répartir uniformément les autres couleurs.

La diffusion du dégradé

Par défaut le dégradé n'est pas rediffusé (**QGradient ::PadSpread**). Le dégradé rempli l'espace qui lui a été attribué et le reste de la zone dessinée est occupé par la dernière couleur définie

Pour obtenir un *dégradé vertical* (bandes horizontales allant du blanc en haut au noir en bas), il suffit de définir la largeur du dégradé à zéro :

```
QLinearGradient linearGradient(0, 0, 0, 100);
```

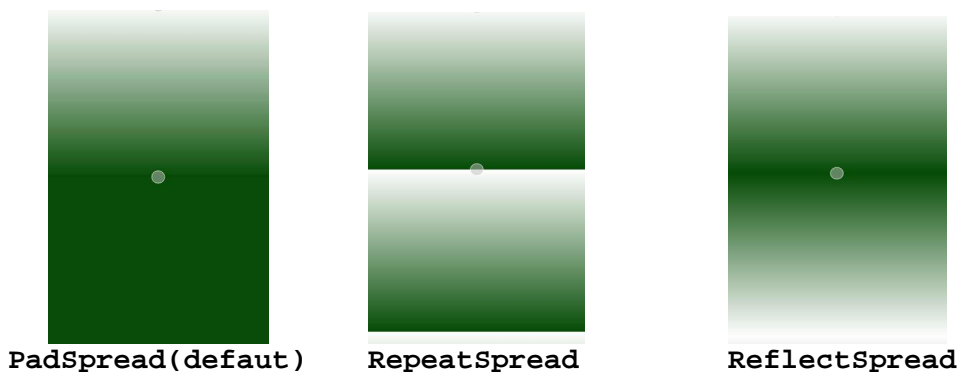
Pour obtenir un *dégradé répétitif* (séquences de bandes obliques allant du blanc au noir du coin haut gauche au coin bas droit), il suffit de réduire la taille du rectangle de dégradé dans les deux directions :

```
QLinearGradient linearGradient(0, 0, 25, 25);  
LinearGradient.setSpread(QGradient ::RepeatSpread) ;
```

La répétition sera obtenue par l'appel à setSpread.

Sans cette ligne seul le coin haut gauche serait dégradé, le reste serait noir.

Si on souhaite obtenir un dégradé inverse on utilisera le même appel mais on utilisera l'énuméré **QGradient ::ReflectSpread**.

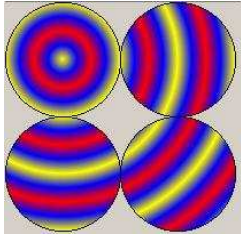


L'origine de la brosse

Les paramètres de positionnement et de dimension donnés au dégradé sont relatifs à l'espace de dessin sur lequel opère le QPainter. Ce qui implique que l'origine du dégradé n'est pas située à l'origine de la forme dessinée. Si, donc, le même QPainter est utilisé pour effectuer plusieurs dessins, l'origine du dégradé n'étant pas modifiée le dégradé partira donc toujours du même point.

```
void paintArea::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    QRadialGradient grad(50.0,50.0,25.0,50.0,50.0);
    grad.setSpread(QGradient::ReflectSpread);
    grad.setColorAt(0.0,Qt::yellow);
    grad.setColorAt(0.5,Qt::blue);
    grad.setColorAt(1.0,Qt::red);
    painter.setBrush(QBrush(grad));
    painter.drawEllipse(0,0,100,100);      //1er cercle en haut à gauche
    painter.drawEllipse(100,0,100,100);   //2eme cercle en haut à droite
    painter.drawEllipse(100,100,100,100); //3eme cercle en bas à droite
    painter.drawEllipse(0,100,100,100);   //4eme cercle en bas à droite
}
```

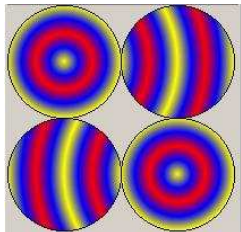
Le code ci-dessus donne en conséquence le résultat suivant :



L'effet visuel donne l'impression que le dégradé remplit tout l'espace de dessin. Les formes dessinées étant comme des fenêtres ouvrant sur cette couverture de dessin.

Cela est en réalité le résultat d'une utilisation d'une brosse (QBrush) qui, par défaut, a comme origine le coin supérieur gauche de la zone de dessin.

Une solution est d'utiliser la fonction `painter.setBrushOrigin(100,100)` afin de déplacer l'origine du dégradé. Si on rajoute cet appel avant de dessiner le 3^{ème} cercle, cela donne le résultat suivant :



à noter que le 4^{ème} cercle voit son origine de dégradée modifiée

Une autre possibilité est de modifier les paramètres du dégradé pour le faire coïncider comme vous le souhaitez avec la forme à dessiner.

La couleur de remplissage

La classe [QColor](#) supporte les formats RGB, HSV et CMYK(*).
Cette classe est utilisée pour la définition de la couleur de remplissage.

(*)

RGB : red, blue, green

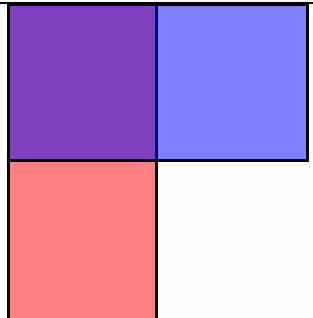
HSV : Hue, hue, saturation, value

CMYK : cyan, magenta, yellow, black

L'effet de [transparence](#) (alpha-blended support) est également paramétrable lors de l'affectation d'une couleur et ce quel que soit le format utilisé.

La transparence est définie dans une fourchette de valeur de 0 à 255.

La valeur 0 correspond à la transparence totale et 255 la colorisation totale.

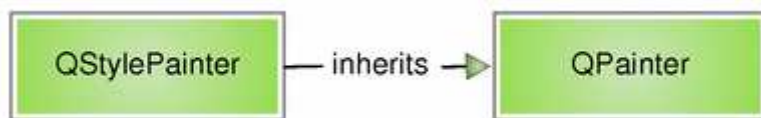
| | |
|--|--|
|  | <pre>//rectangle rouge semi-transparent painter.setBrush(QColor(255, 0, 0, 127)); painter.drawRect(0, 0, width()/2, height()); //rectangle bleu semi-transparent painter.setBrush(QColor(0, 0, 255, 127)); painter.drawRect(0, 0, width(), height()/2); la couleur de la zone commune est alors un mélange de bleu et de rouge</pre> |
|--|--|

Avec du style

QStyle est une classe abstraite qui permet de définir l'apparence d'un élément graphique. Un grand nombre de fonctions permettant le dessin requièrent les 4 arguments suivants :

- Un énuméré pour préciser le type d'élément graphique à dessiner
- [QStyleOption](#) afin de spécifier comment et où dessiner cet élément
- [QPainter](#) l'outil de dessin à utiliser pour dessiner cet élément
- [QWidget](#) l'objet graphique sur lequel dessiner (optionnel)

Par commodité, la classe QPainter permet de combiner un outil de dessin (QPainter), dans un style donné (QPainter::Style()) sur un widget défini (QPainter::QPainter (QWidget * widget)).



Du texte haut en couleur

Nous avons vu l'usage des classes QGradient afin de rendre plus « fun » le remplissage d'une zone dessinée.

Ces classes peuvent également être utilisées afin de coloriser du texte graduellement.

| | |
|--|---|
| <pre>Qt 4 by Trolltech Qt 4 by Trolltech Qt 4 by Trolltech Qt 4 by Trolltech Qt 4 by Trolltech Qt 4 by Trolltech Qt 4 by Trolltech</pre> | <pre>QLinearGradient gradient(0, 0, 100, 100); gradient.setColorAt(0, Qt::blue); gradient.setColorAt(1, Qt::red); painter.setPen(QPen(gradient, 0)); for (int y=fontSize; y<100; y+=fontSize) painter.drawText(0, y, text);</pre> |
|--|---|

Le Système de coordonnées

(<http://doc.trolltech.com/4.3/coordsys.html>)

Le QPainter est un espace à deux dimensions dans lequel nous allons dessiner.

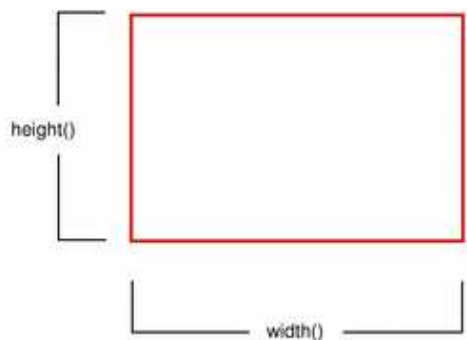
L'origine du système de coordonnées par défaut d'un QPainter est le coin haut gauche. Les coordonnées en x values s'incrémentent vers la droite et les coordonnées y vers le bas. L'unité par défaut est d'un pixel pour les espaces basés sur le pixel et d'un point (1/72 d'un pouce (inch)) pour les imprimantes.

Par défaut, il y a correspondance entre les coordonnées logiques du QPainter et coordonnées physiques du QPainter.

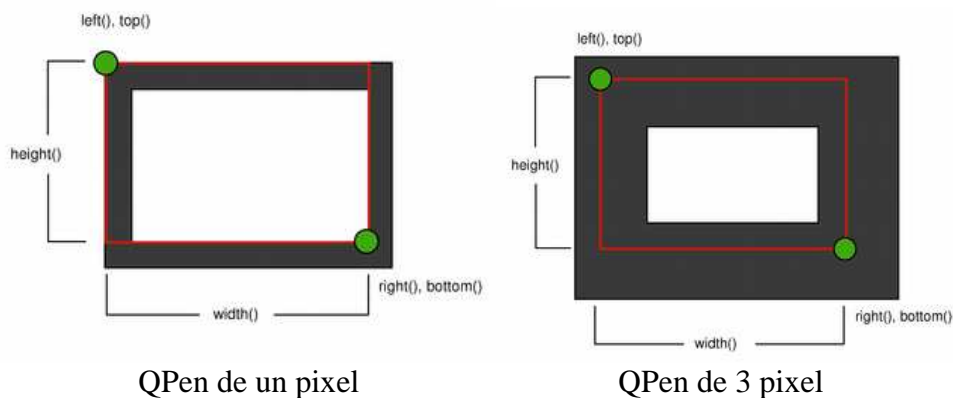
La correspondance entre les coordonnées physiques et logique est assurée par la matrice de transformation du QPainter.

Une [matrice](#) spécifie comment déplacer, décaler, redimensionner, effectuer une rotation du système de coordonnées, et est utilisé spécifiquement lors du rendu du dessin.

La représentation logique d'une primitive correspond à son model mathématique, que que soit l'épaisseur de l'outil de dessin (QPen)



représentation logique (QPainter)



QPen de un pixel

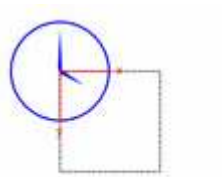
QPen de 3 pixel

L'utilisation de l'**anti-aliasing** (ou anti-crénelage in french) va ajouter des pixels de couleur graduée symétriquement, aux coins du modèle mathématique.

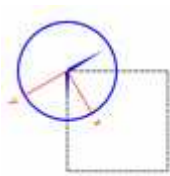
| | |
|--|--|
| | <pre> QPainter painter(this); painter.setRenderHint(Qt::Antialiasing); painter.setPen(Qt::darkGreen); painter.drawRect(1, 2, 6, 4); </pre> |
|--|--|

Par défaut le `QPainter` opère sur le système de coordonnées du périphérique (`QpaintDevice`) associé.

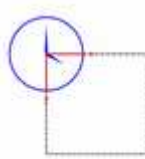
En complément, cette classe offre la possibilité d'affiner la **transformation des coordonnées** en proposant les fonctions suivantes :



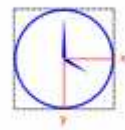
position de départ



[rotate\(\)](#)



[scale\(\)](#)



[translate\(\)](#)

(cf l'exemple `$QTDIR/\examples\painting\transformations\release\transformations`)

Il est possible également de vriller le système de coordonnées autour de son origine ([QPainter::shear\(\)](#)). Ces transformations sont opérées sur la matrice du `QPainter` que l'on peut récupérer via l'appel à [QPainter::matrix\(\)](#) ou définir via [QPainter::setMatrix\(\)](#).

Si on doit modifier un nombre de fois conséquent cette matrice ou revenir en arrière, on peut sauver ([QPainter::save\(\)](#)) et restaurer ([QPainter::restore\(\)](#)) ces [QMatrix](#).

La Conversion des coordonnées logiques en coordonnées physiques (et inversement)

<http://doc.trolltech.com/4.3/coordsys.html> - window-viewport-conversion)

En plus de la matrice, le QPainter utilise les [viewport\(\)](#) et [window\(\)](#) afin d'assurer la correspondance entre coordonnées logiques (QPainter) et coordonnées physiques (QPaintDevice).

Le « viewport » représente les coordonnées physiques qui spécifient un rectangle.

La “window” décrit le même rectangle en coordonnées logiques.

Par default les deux systèmes de coordonnées et sont équivalents au rectangle du périphérique de dessin(QPaintDevice)..

L'utilisation de ces systèmes de coordonnées offrent la possibilité d'être indépendant du périphérique utilisé en gérant les conversions de coordonnées physiques/logiques.