

Graphismes en PyQt

Sylvain Malacria

<http://www.malacria.com/>

<mailto:sylvain.malacria@inria.fr>



Diapositives inspirées de Gilles Bailly

Objectifs

Introduction

- ▶ Signaux et slots
- ▶ Bases de PyQt
- ▶ Les principales classes Qt

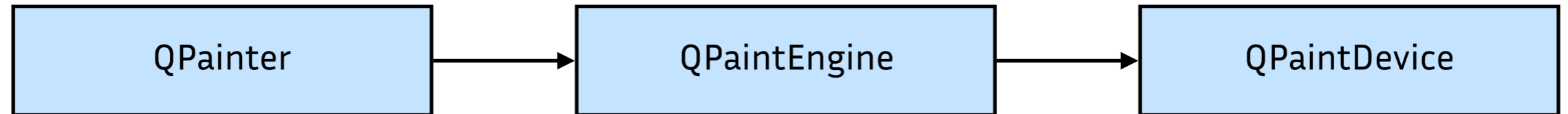
Graphisme avancé

- ▶ Le **dessin** en PyQt
- ▶ Programmation **événementielle**
- ▶ Notion avancée de **graphisme** avec Qt

#1 Le dessin en PyQt

Dessin en PyQt

Paint system



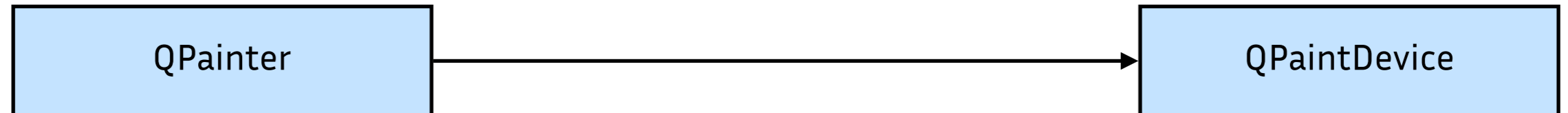
L'API de *peinture* de Qt permet de *peindre* à l'écran, dans un fichier, etc.

3 classes principales :

- ▶ **QPainter** pour effectuer des opérations de dessin
- ▶ **QPaintDevice** abstraction 2D dans laquelle on dessine
- ▶ **QPaintEngine** interface pour relier les deux

Dessin en PyQt

Paint system



L'API de *peinture* de Qt permet de *peindre* à l'écran, dans un fichier, etc.

3 classes principales :

- ▶ **QPainter** pour effectuer des opérations de dessin
- ▶ **QPaintDevice** abstraction 2D dans laquelle on dessine
- ▶ **QPaintEngine** interface pour relier les deux

QPaintEngine utilisée de manière interne (cachée) par QPainter et QPaintDevice

QPainter

QPainter est *l'outil* de dessin

- ▶ lignes simples
- ▶ path
- ▶ formes géométriques (ellipse, rectangle, etc.)
- ▶ texte
- ▶ images
- ▶ etc.

Utilise deux objets principaux

- ▶ QBrush (pour les *fill*)
- ▶ QPen (pour les *stroke*)

Fonction principale est de dessiner, mais dispose d'autres fonctions pour optimiser son rendu

Peut dessiner sur n'importe quel objet qui hérite de la classe *QPaintDevice*

Exemples de QPaintDevice

Classe de base dans laquelle on peut *peindre* avec un QPainter

- ▶ **QWidget**
- ▶ QImage
- ▶ QPixmap
- ▶ QPicture
- ▶ QPainter
- ▶ ...

Dessiner dans un QWidget ?

Le widget se “dessine” lorsqu’il est *repaint*

Le widget est repaint lorsque :

- ▶ une fenêtre passe au dessus
- ▶ on redimensionne la fenêtre
- ▶ on lui demande explicitement
 - Repaint(), force le widget à être redessiné
 - Update(), un évènement de dessin est ajouté en file d’attente

Dessiner dans un QWidget ?

Le widget se “dessine” lorsqu’il est *repaint*

Le widget est repaint lorsque :

- ▶ une fenêtre passe au dessus
- ▶ on redimensionne la fenêtre
- ▶ on lui demande explicitement
 - Repaint(), force le widget à être redessiné
 - Update(), un évènement de dessin est ajouté en file d’attente

Dans tous les cas, c’est la méthode :

```
paintEvent(self, QPaintEvent)
```

qui est appelée (et vous ne devez jamais l’appeler manuellement)





Dessiner dans un QWidget

```
class Dessin(QWidget):
```

```
    #evenement QPaintEvent
```

```
    def paintEvent(self, event):
```

```
        # Blabla de dessin ici
```

```
# event de type QPaintEvent
```

```
def main(args):
```

```
    app = QApplication(args)
```

```
    win = QMainWindow()
```

```
    win.setCentralWidget(Dessin())
```

```
    win.resize(300,200)
```

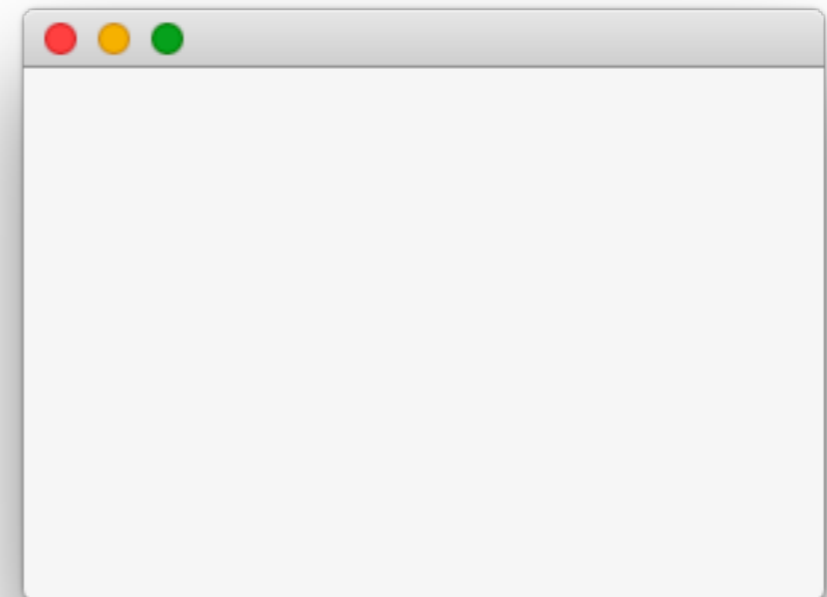
```
    win.show()
```

```
    app.exec_()
```

```
    return
```

```
if __name__ == "__main__":
```

```
    main(sys.argv)
```



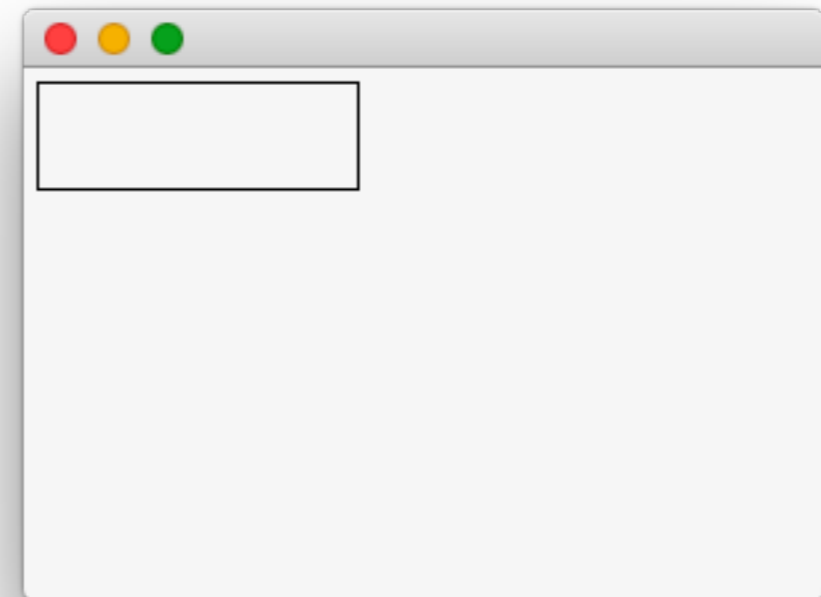
Dessiner dans un QWidget

```
class Dessin(QWidget):  
  
    #evenement QPaintEvent  
    def paintEvent(self, event):  
        painter = QPainter(self)  
        painter.drawRect(5,5,120,40)  
        return
```

```
# event de type QPaintEvent  
# recupere le QPainter du widget  
# dessiner un rectangle noir
```

```
def main(args):  
    app = QApplication(args)  
    win = QMainWindow()  
    win.setCentralWidget(Dessin())  
    win.resize(300,200)  
    win.show()  
    app.exec_()  
    return
```

```
if __name__ == "__main__":  
    main(sys.argv)
```



Dessiner dans un QWidget

Pourquoi cela fonctionne?

```
class Dessin(QWidget):
```

```
    #evenement QPaintEvent
```

```
    def paintEvent(self, event):
```

```
        painter = QPainter(self)
```

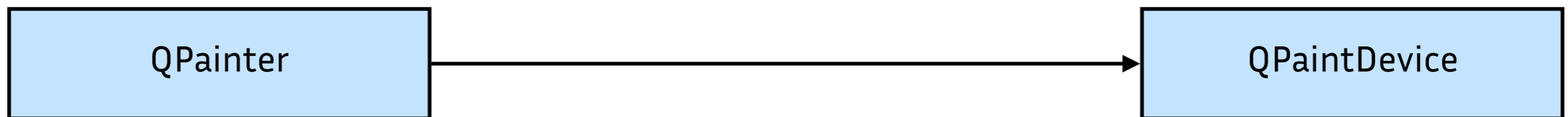
```
        painter.drawRect(5,5,120,40)
```

```
        return
```

```
    # event de type QPaintEvent
```

```
    # recupere le QPainter du widget
```

```
    # dessiner un rectangle noir
```



QPainter comme outil de dessin

QWidget hérite de QPaintDevice

Dessin avancé

QPainter:

- ▶ DrawLine(), drawEllipse(), drawRect(), drawPath(), etc.
- ▶ fillRect(), fillEllipse()
- ▶ drawText()
- ▶ drawPixmap(), drawImage()
- ▶ setPen(), setBrush()



Dessin coloré

```
class Dessin(QWidget):
```

```
    #evenement QPaintEvent
```

```
    def paintEvent(self, event):
```

```
        painter = QPainter(self)
```

```
        painter.setPen(Qt.red)
```

```
        painter.setBrush(Qt.lightGray)
```

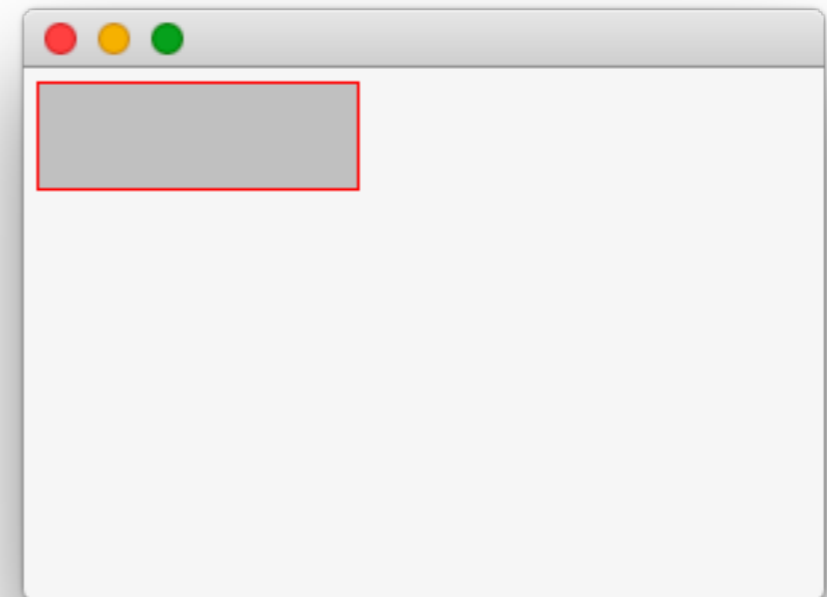
```
        painter.drawRect(5,5,120,40)
```

```
    # recupere le QPainter du widget
```

```
    # add a red pen
```

```
    # set a light gray brush
```

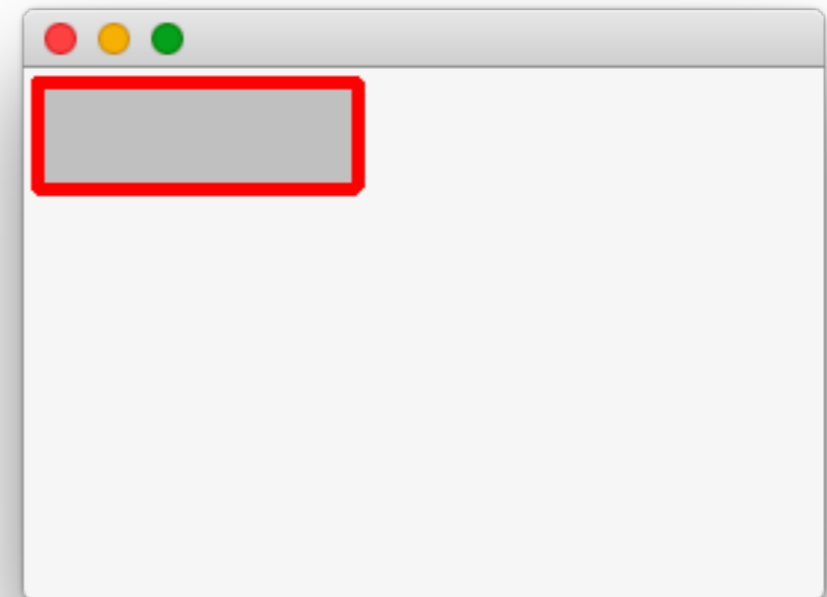
```
    # dessine le rectangle
```



Instancier un Pen

```
class Dessin(QWidget):  
  
    #evenement QPaintEvent  
    def paintEvent(self, event):  
        painter = QPainter(self)  
        pen = QPen(Qt.red)  
        pen.setWidth(5)  
        painter.setPen(pen)  
        painter.setBrush(Qt.lightGray)  
        painter.drawRect(5,5,120,40)
```

```
# recupere le QPainter du widget  
# instancier un pen  
# change l'epaisseur  
# appliquer ce pen au painter  
# set a light gray brush  
# dessine le rectangle
```



```
class Dessin(QWidget):
```

```
    #evenement QPaintEvent
```

```
    def paintEvent(self, event):
```

```
        painter = QPainter(self)
```

```
        pen = QPen(Qt.red)
```

```
        pen.setWidth(5)
```

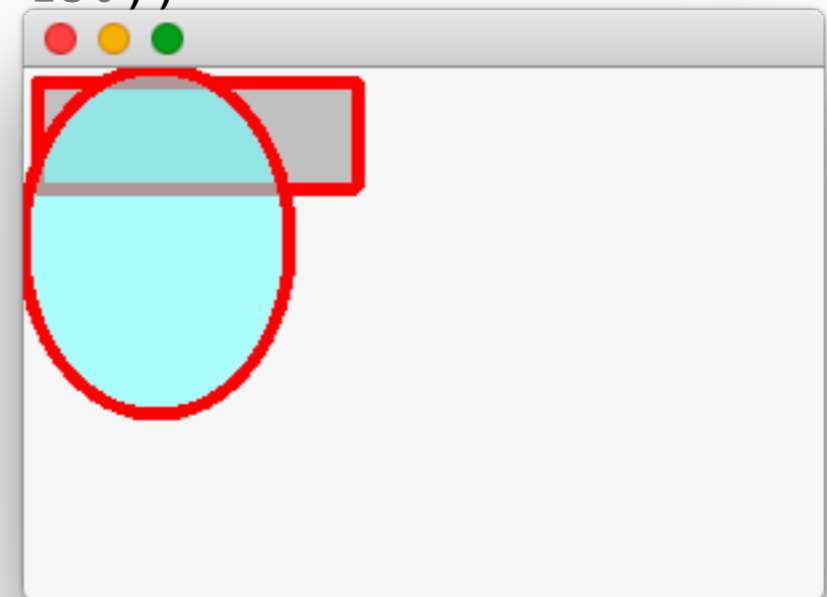
```
        painter.setPen(pen)
```

```
        painter.setBrush(Qt.lightGray)
```

```
        painter.drawRect(5,5,120,40)
```

```
        painter.setBrush(QColor(120, 255, 255, 150))
```

```
        painter.drawEllipse(0,0,100,130)
```



Questions

Où dessiner?

- ▶ Dans la méthode `paintEvent(self, QPaintEvent)`

Comment demander le réaffichage d'un widget

- ▶ `update()`
- ▶ `repaint()`

Quelle est la différence entre `update()` et `repaint()`?

- ▶ `update()` indique qu'une zone est à réafficher (mais l'affichage n'est pas instantané)
- ▶ `repaint()` réaffiche immédiatement (mais peut introduire de la latence)

Comment dessiner dans `paintEvent(QPaintEvent)`

- ▶ instancier un `QPainter`: `p = QPainter(self)`

Dans quoi dessiner?



Tous ce qui hérite de QPaintDevice

- ▶ QWidget
- ▶ QPrinter
- ▶ QPixmap
- ▶ QImage
- ▶ etc.

Possibilité de rendu de "haut niveau" (SVG)

- ▶ QSvgRenderer
- ▶ QSvgwidget

#2 Gestion des évènements

Gestions des évènements souris dans un QWidget

Méthodes qui héritent de QWidget

```
def mousePressEvent(self, event):
```

```
def mouseReleaseEvent(self, event):
```

```
def mouseMoveEvent(self, event):
```

```
def enterEvent(self, event):
```

```
def leaveEvent(self, event):
```

```
class Dessin(QWidget):  
  
    def mousePressEvent(self, event):           # evenement mousePress  
        self.pStart = event.pos()  
        print("press: ", self.pStart)  
  
    def mouseReleaseEvent(self, event):        # evenement mouseRelease  
        self.pStart = event.pos()  
        print("release: ", event.pos())
```

```
193-51-236-93:TPs sylvain$ python3 drawingexample.py  
press: PyQt5.QtCore.QPoint(141, 28)  
release: PyQt5.QtCore.QPoint(141, 28)  
press: PyQt5.QtCore.QPoint(274, 129)  
release: PyQt5.QtCore.QPoint(274, 129)
```

Les méthodes sont appelées automatiquement car la classe hérite de QWidget !!

```
class Dessin(QWidget):  
  
    def __init__(self):  
        super().__init__() #activer le "mouse tracking"  
        self.setMouseTracking(True)  
  
    def mousePressEvent(self, event): # evenement mouseMove  
        self.pStart = event.pos()  
        print("move: ", self.pStart)
```

Par défaut, *mousePressEvent* envoyés seulement si bouton de souris enfoncé (*Drag*)

Possible d'activer/désactiver en permanence en utilisant ***setMouseTracking(bool)***

Exemple

```
class Dessin(QWidget):

    def __init__(self):
        super().__init__()
        self.setMouseTracking(True)           # on active le mouseTracking
        self.cursorPos = None

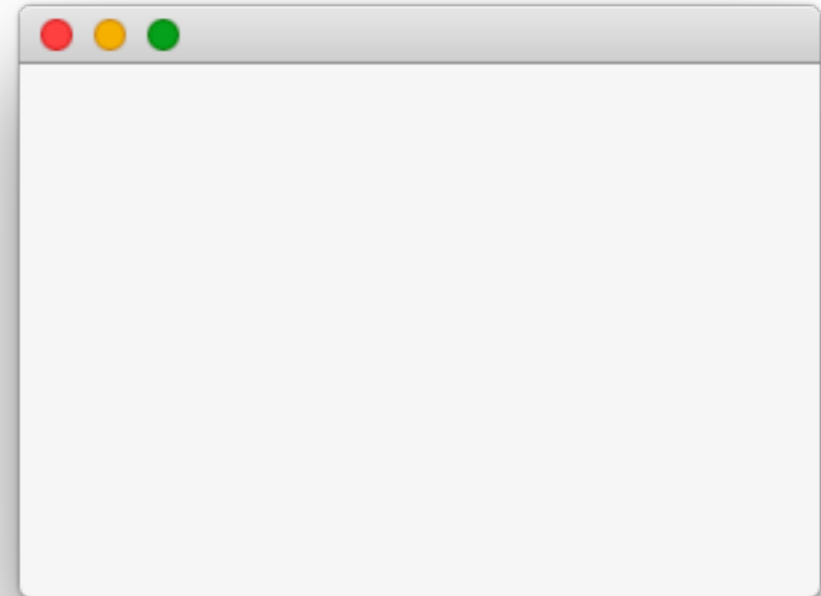
    def mouseMoveEvent(self, event):          # evenement mouseMove
        self.cursorPos = event.pos()         # on stocke la position du curseur
        self.update()                        # on met à jour l'affichage

    #evenement QPaintEvent
    def paintEvent(self, event):
        painter = QPainter(self)

        if self.cursorPos != None:
            painter.drawEllipse(\
                self.cursorPos.x()-5,\
                self.cursorPos.y()-5,10,10)   # On dessine l'ellipse autour du curseur
```

Exemple

```
class Dessin(QWidget):  
  
    def __init__(self):  
        super().__init__()  
        self.setMouseTracking(True)  
        self.cursorPos = None  
  
    def mouseMoveEvent(self, event):  
        self.cursorPos = event.pos()  
        self.update()  
  
    #evenement QPaintEvent  
    def paintEvent(self, event):  
        painter = QPainter(self)  
  
        if self.cursorPos != None:  
            painter.drawEllipse(\  
                self.cursorPos.x()-5,\  
                self.cursorPos.y()-5,10,10)
```



```
# on active le mouseTracking
```

```
# evenement mouseMove
```

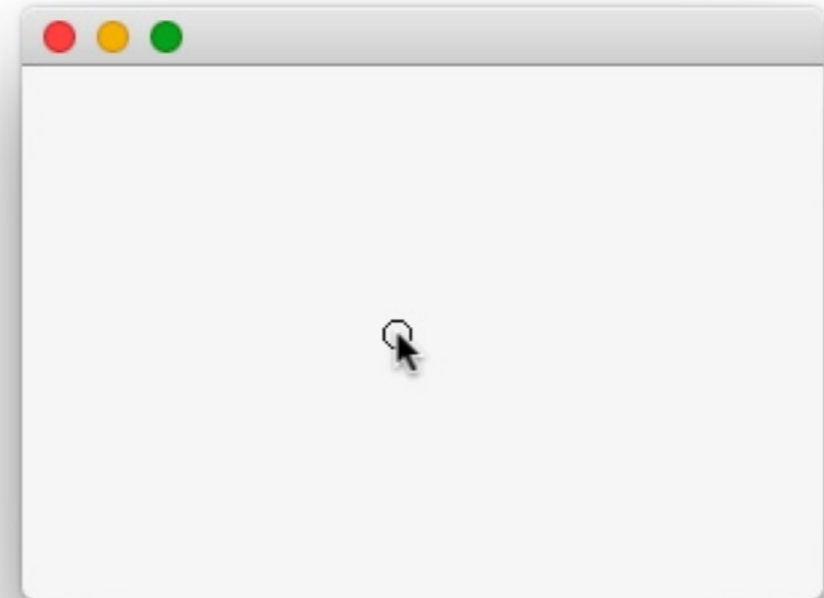
```
# on stocke la position du curseur
```

```
# on met à jour l'affichage
```

```
# On dessine l'ellipse autour du curseur
```

Exemple

```
class Dessin(QWidget):  
  
    def __init__(self):  
        super().__init__()  
        self.setMouseTracking(True)  
        self.cursorPos = None  
  
    def mouseMoveEvent(self, event):  
        self.cursorPos = event.pos()  
        self.update()  
  
    #evenement QPaintEvent  
    def paintEvent(self, event):  
        painter = QPainter(self)  
  
        if self.cursorPos != None:  
            painter.drawEllipse(\  
                self.cursorPos.x()-5,\  
                self.cursorPos.y()-5,10,10)
```



```
# on active le mouseTracking
```

```
# evenement mouseMove
```


```
# on stocke la position du curseur
```

```
# on met à jour l'affichage
```

```
# On dessine l'ellipse autour du curseur
```

QMouseEvent

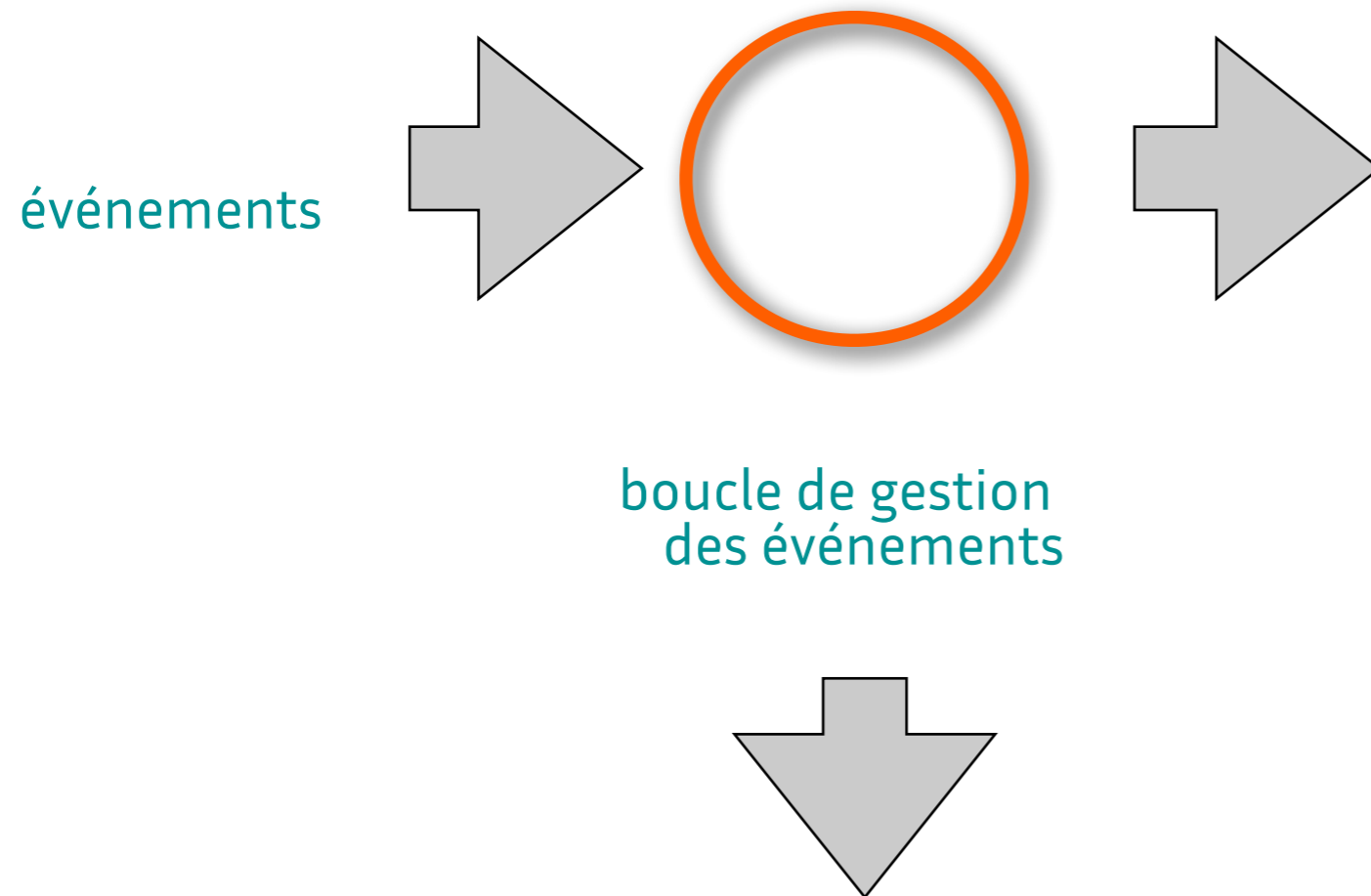
`def mousePressEvent(self, event):` De type QMouseEvent



QMouseEvent permet de récupérer (selon versions) :

- ▶ `button()` : bouton souris qui a déclenché l'événement. ex: `Qt.LeftButton`
- ▶ `buttons()` : état des autres boutons. ex: `Qt.LeftButton` | `Qt.MidButton`
- ▶ `modifiers()` : modificateurs clavier. ex: `Qt.ControlModifier` | `Qt.ShiftModifier`
- ▶ `pos()` : position locale (relative au widget)
- ▶ `globalPos()`, `windowPos()`, `screenPos()` : position globale ou relative à ce référentiel
 - utile si on déplace le widget interactivement !

Synthèse



```
def mousePressEvent(self, QMouseEvent):  
    .....  
    .....  
    update();  
}
```

```
def mouseMoveEvent(self, QMouseEvent):  
    .....  
    .....  
    update();  
}
```

```
def mouseReleaseEvent(QMouseEvent* e):  
    .....  
    .....  
    update();  
}
```

```
def paintEvent(self, QPaintEvent):  
    painter = QPainter(self)  
    .....  
    return
```

#3 Dessin avancé

QPainter

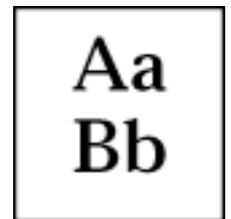
Attributs



- ▶ **setPen()** : lignes et contours



- ▶ **setBrush()** : remplissage



- ▶ **setFont()** : texte



- ▶ **setTransform()**, etc. : transformations affines



- ▶ **setClipRect/Path/Region()** : clipping (découpage)

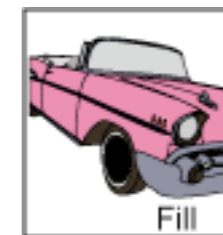


- ▶ **setCompositionMode()** : composition

QPainter

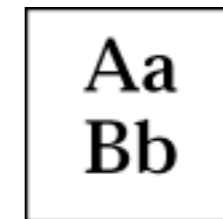
Lignes et contours

- ▶ `drawPoint()`, `drawPoints()`
- ▶ `drawLine()`, `drawLines()`
- ▶ `drawRect()`, `drawRects()`
- ▶ `drawArc()`, `drawEllipse()`
- ▶ `drawPolygon()`, `drawPolyline()`, etc...
- ▶ **`drawPath()`** : chemin complexe



Remplissage

- ▶ `fillRect()`, **`fillPath()`**



Divers

- ▶ `drawText()`
- ▶ `drawPixmap()`, `drawImage()`, `drawPicture()`
- ▶ etc.

QPainter

Classes utiles

- ▶ entiers: **QPoint, QLine, QRect, QPolygon**
- ▶ flottants: **QPointF, QLineF, ...**
- ▶ chemin complexe: **QPainterPath**
- ▶ zone d'affichage: **QRegion**

Pinceau: QPen



Attributs

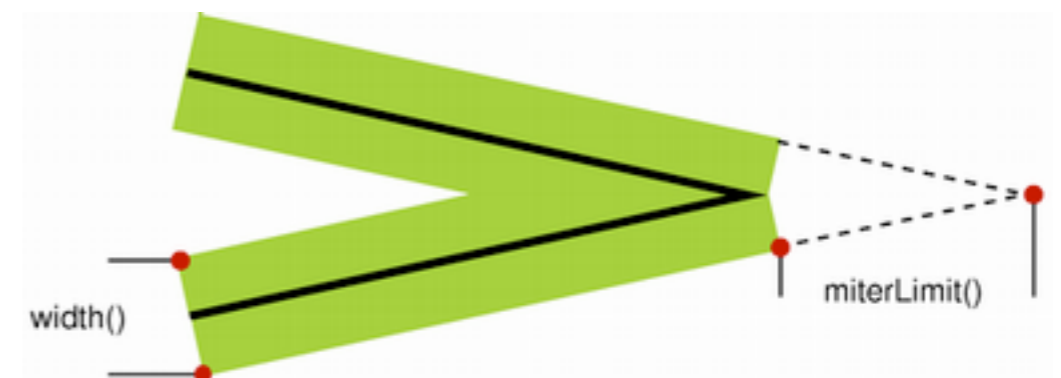
- ▶ **style** : type de ligne
- ▶ **width** : épaisseur
- ▶ **brush** : attributs du pinceau (couleur...)
- ▶ **capStyle** : terminaisons
- ▶ **joinStyle** : jointures



PenStyle



Join Style



Pinceau: QPen



Exemple

```
// dans méthode PaintEvent()  
  
pen = QPen() // default pen  
pen.setStyle(Qt.DashDotLine)  
pen.setWidth(3)  
pen.setBrush(Qt.green)  
pen.setCapStyle(Qt.RoundCap)  
pen.setJoinStyle(Qt.RoundJoin)  
  
painter = QPainter( self )  
painter.setPen( pen )
```

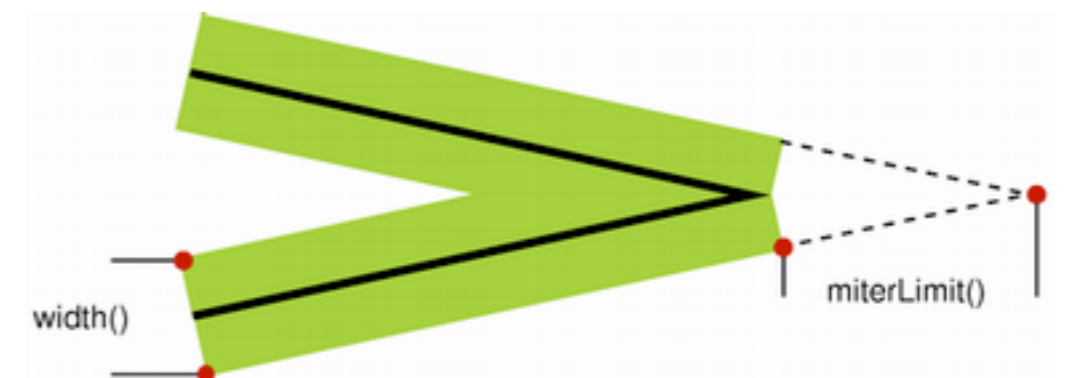


PenStyle



Cap Style

Join Style



Remplissage: QBrush



Attributs

- ▶ style
- ▶ color
- ▶ gradient
- ▶ texture

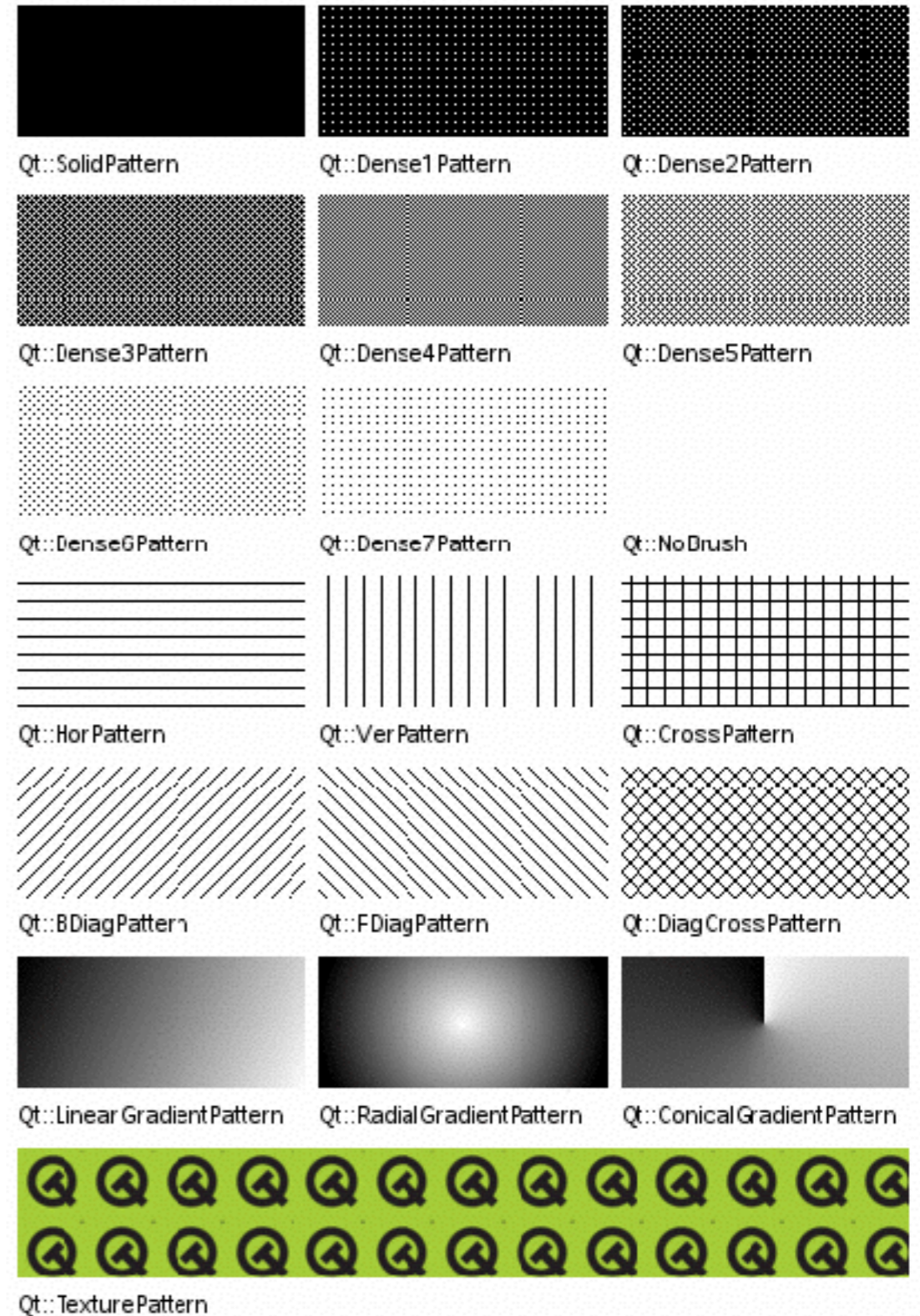
BrushStyle

```
brush = QBrush( ... )
```

.....

```
painter = QPainter(self)
```

```
painter.setBrush(brush)
```



Remplissage: QBrush



Attributs

- ▶ style
- ▶ color
- ▶ gradient
- ▶ texture

| | | | |
|-----------|-----------|---------|-------------|
| white | black | cyan | darkCyan |
| red | darkRed | magenta | darkMagenta |
| green | darkGreen | yellow | darkYellow |
| blue | darkBlue | gray | darkGray |
| lightGray | | | |

QColor

- ▶ **modèles** RGB, HSV or CMYK
- ▶ **composante alpha** (transparence) :
 - alpha blending
- ▶ **couleurs prédéfinies** :
 - Qt.GlobalColor

Qt.GlobalColor

Remplissage: gradients



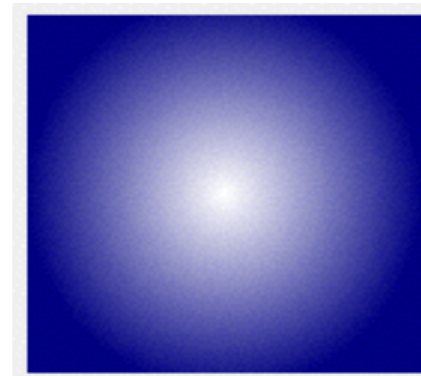
Type de gradients

- ▶ lineaire,
- ▶ radial
- ▶ conique

```
gradient = QLinearGradient(QPointF(0, 0), QPointF(100, 100))  
gradient.setColorAt(0,Qt.white)  
gradient.setColorAt(1,Qt.blue)  
painter.setBrush(gradient)
```



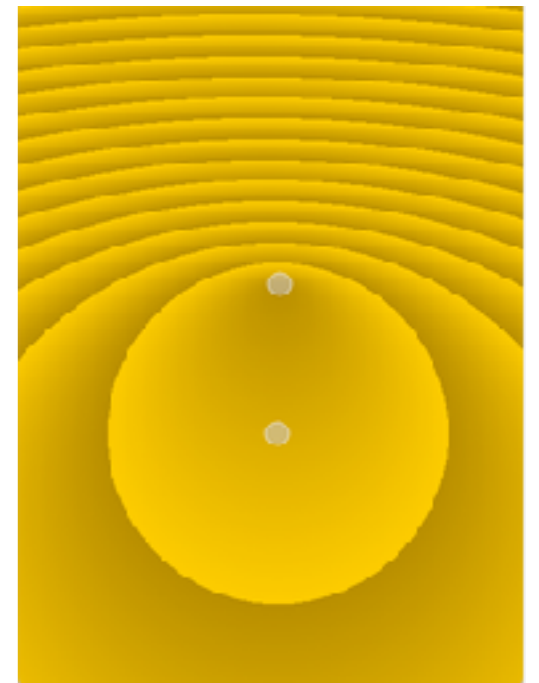
QLinearGradient



QRadialGradient



QConicalGradient

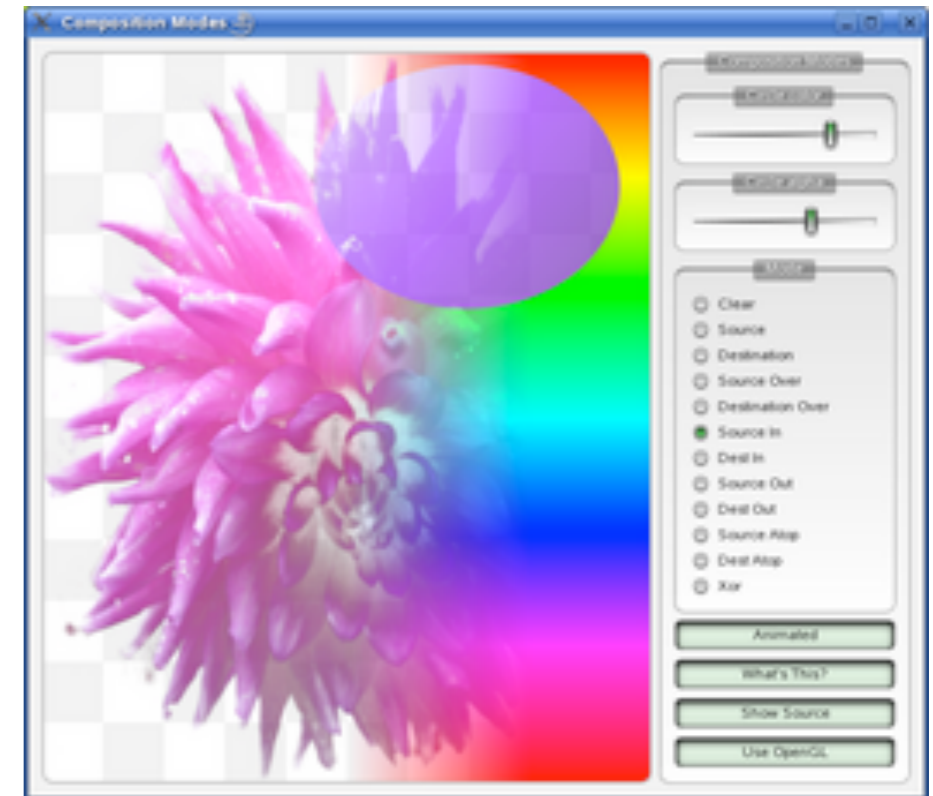


répétition: setSpread()

Composition

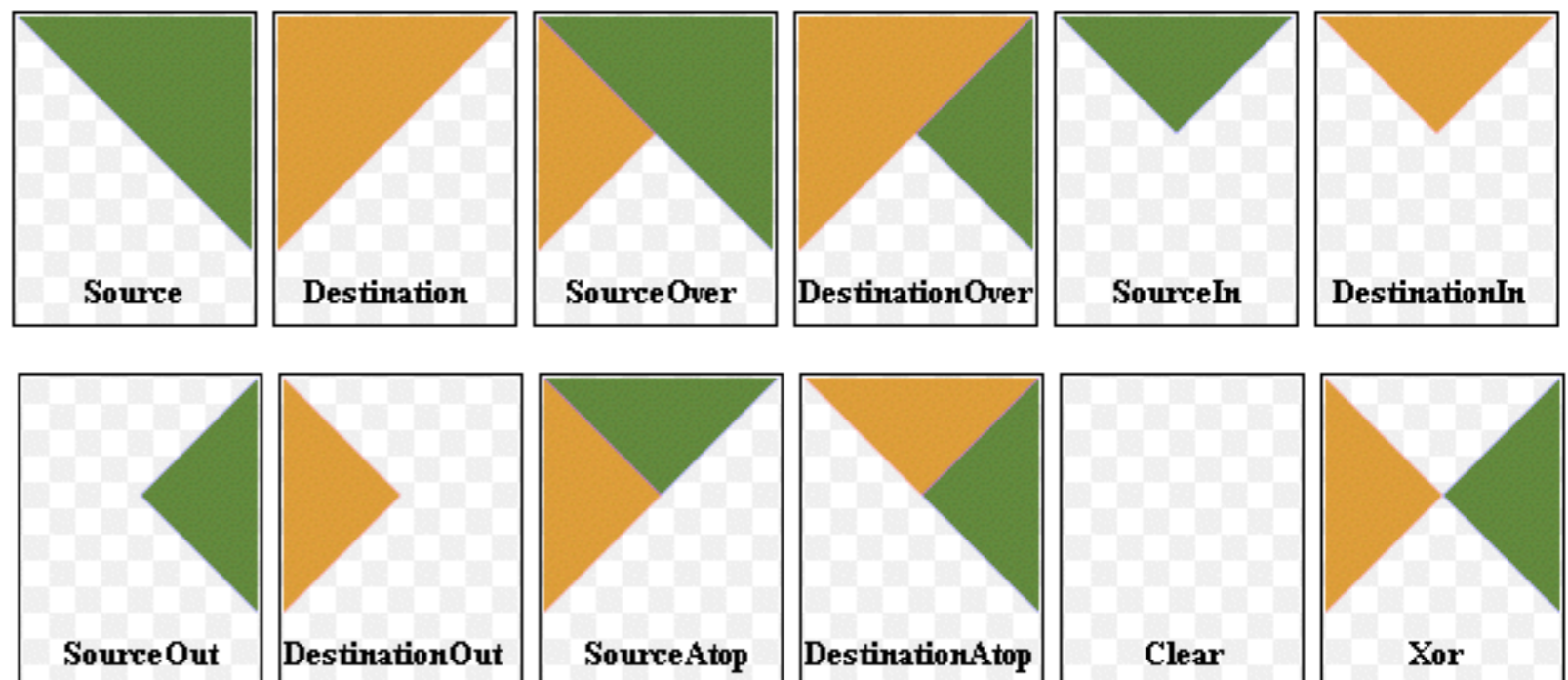
Modes de composition

- ▶ opérateurs de **Porter Duff**:
- ▶ définissent : $F(\text{source}, \text{destination})$
- ▶ défaut : **SourceOver**
 - avec alpha blending
 - $\text{dst} \leq a_{\text{src}} * \text{src} + (1 - a_{\text{src}}) * a_{\text{dst}} * \text{dst}$
- ▶ limitations
 - selon implémentation et Paint Device



Méthode:

`setCompositionMode()`



Découpage (clipping)

Découpage

- ▶ selon un rectangle, une région ou un path
- ▶ méthodes de QPainter
`setClipping()`, `setClipRect()`, `setClipRegion()`, `setClipPath()`

QRegion

- ▶ `united()`, `intersected()`, `subtracted()`, `xored()`



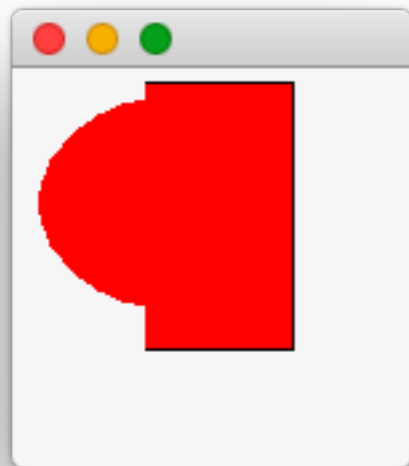
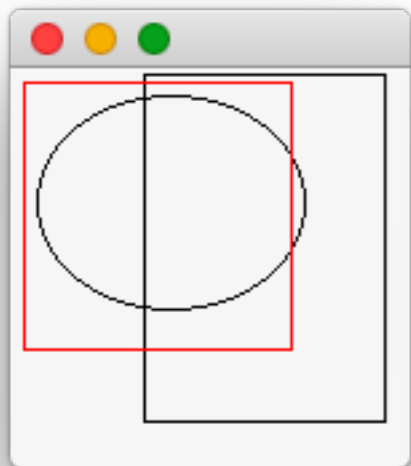
```
r1 = QRegion( QRect(100, 100, 200, 80), QRegion.Ellipse)      # r1: elliptic region
r2 = QRegion( QRect(100, 120, 90, 30) )                       # r2: rectangular region
r3 = r1.intersected(r2);                                       # r3: intersection

painter = QPainter(self);
painter.setClipRegion(r3,Qt.ReplaceClip);

...etc... // paint clipped graphics
```


Example

```
def paintEvent(self, event):  
    painter = QPainter(self)  
    painter.setBrush(QColor(255,0,0))  
    rect1 = QRect(10, 10, 100, 80)  
    rect2 = QRect(50, 2, 90, 130)  
    rect3 = QRect(5, 5, 100,100)  
  
    r1 = QRegion( rect1, QRegion.Ellipse) # definition des regions  
    r2 = QRegion( rect2)  
    rc = r1.subtracted(r2) # combinaison de regions  
  
    painter.setClipRegion(rc) # on attribue la clipregion  
    painter.drawRect(rect3) # on dessine
```



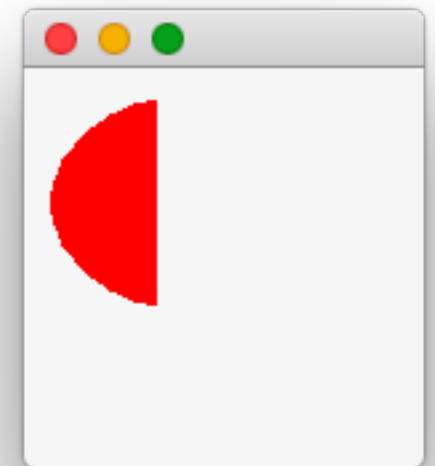
united()



xor()



intersected()



subtracted()

Transformations affines

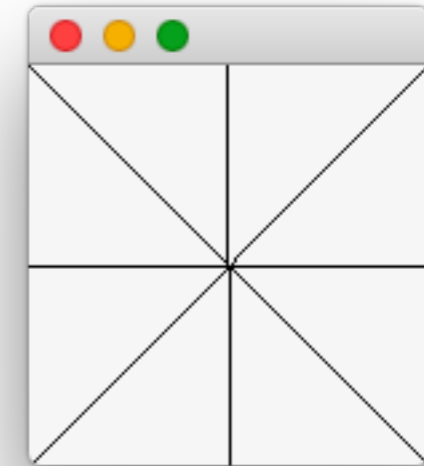
Transformations

- ▶ `translate()`
- ▶ `rotate()`
- ▶ `scale()`
- ▶ `shear()`
- ▶ `setTransform()`

```
#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)
    painter.save()
    painter.translate(self.width()/2,
                     self.height()/2)

    for k in range(0,10):
        painter.drawLine(0, 0, 400, 0)
        painter.rotate(45)

    painter.restore()
```



```
# empile l'état courant

# on "centre" le painter

# dessine une ligne
# rotation de 45°

# dépile l'état courant
```

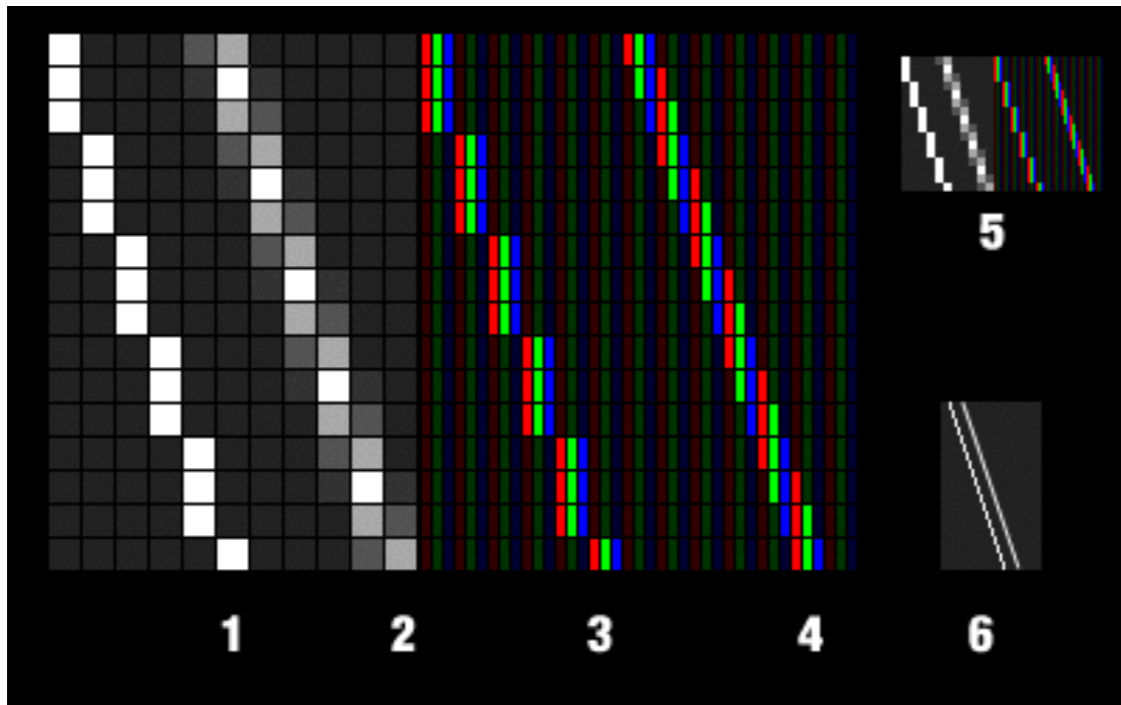
Anti-aliasing

Anti-aliasing

- ▶ éviter l'effet d'escalier
- ▶ particulièrement utile pour les polices de caractères

Subpixel rendering

- ▶ exemples : ClearType, texte sous MacOSX



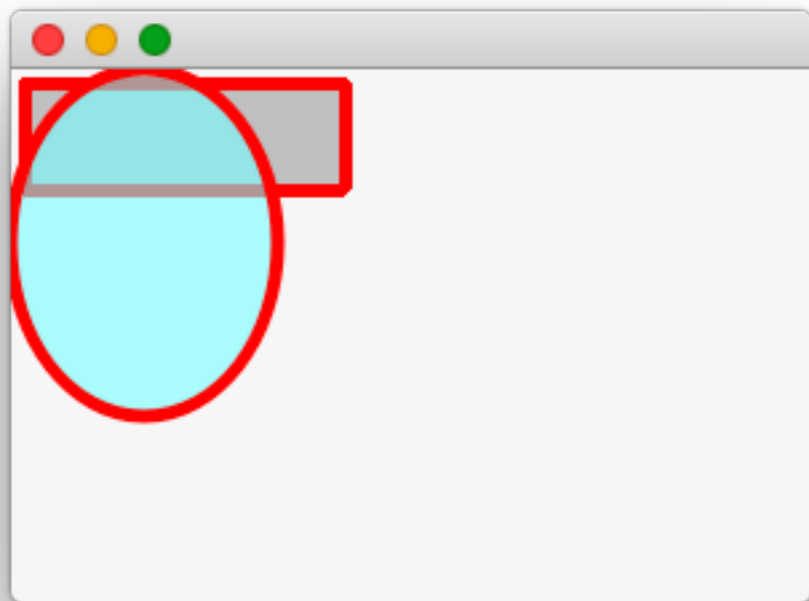
ClearType
(Wikipedia)



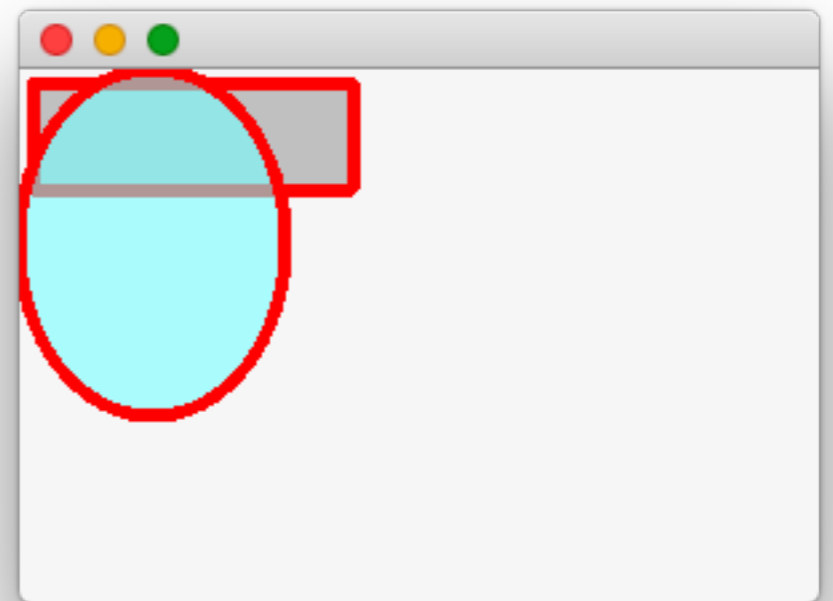
MacOSX

Anti-aliasing

```
painter.setRenderHints(QPainter.Antialiasing)
```



Avec



Sans

Antialiasing et coordonnées

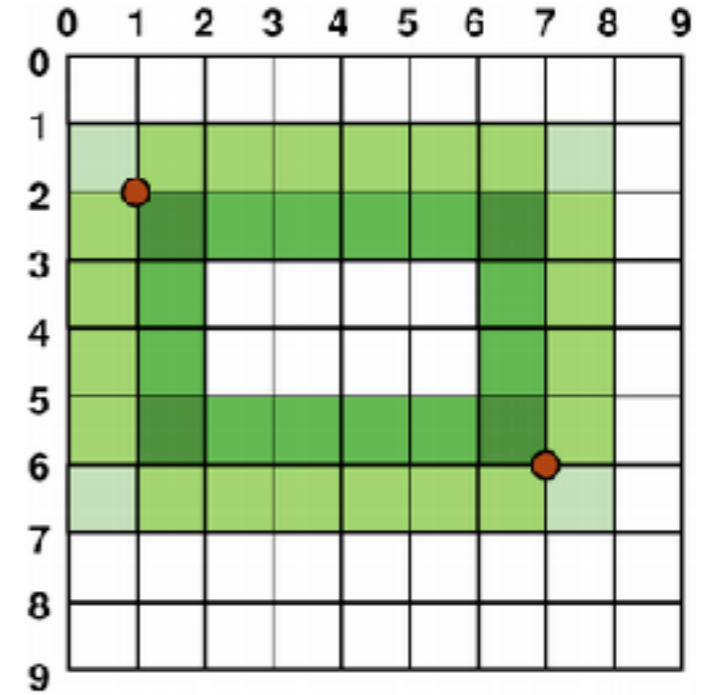
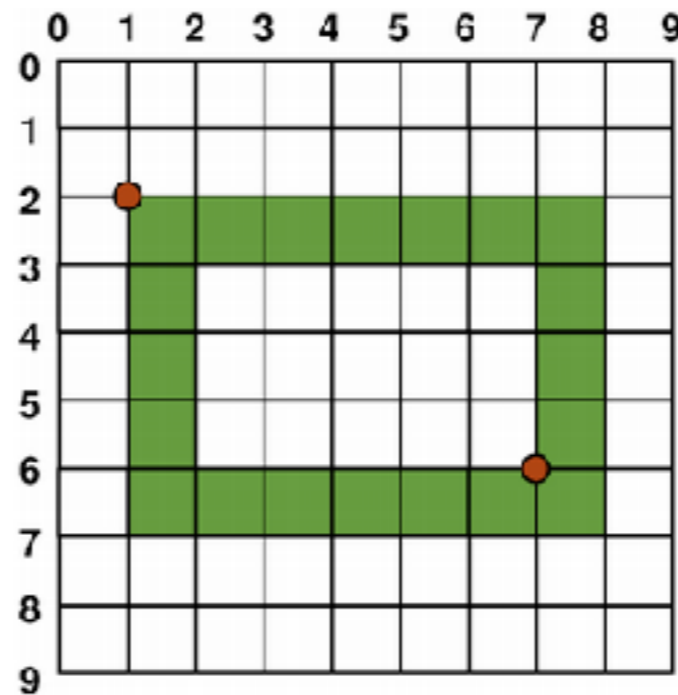
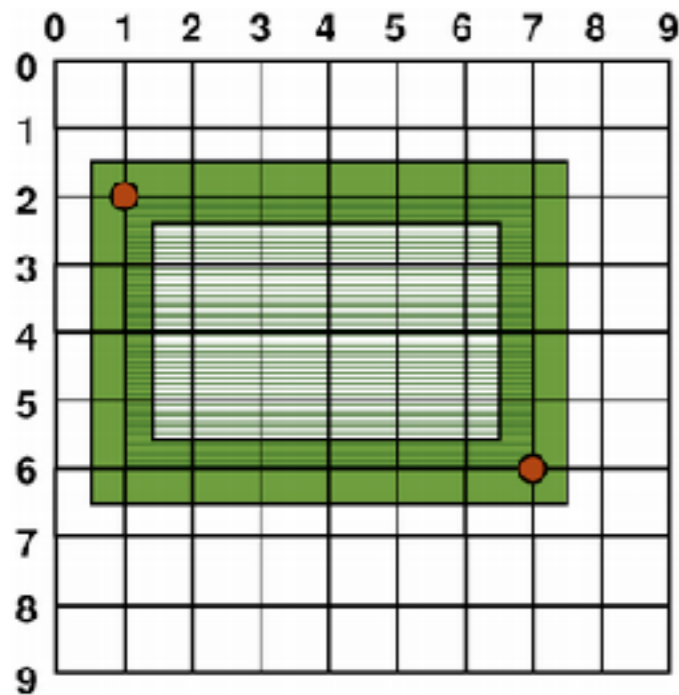
Epaisseurs impaire

- ▶ pixels dessinés à droite et en dessous

Dessin anti-aliasé

- ▶ pixels répartis autour de la ligne idéale

```
QRect.right() = left() + width() - 1  
QRect.bottom() = top() + height() - 1  
Mieux : QRectF (en flottant)
```



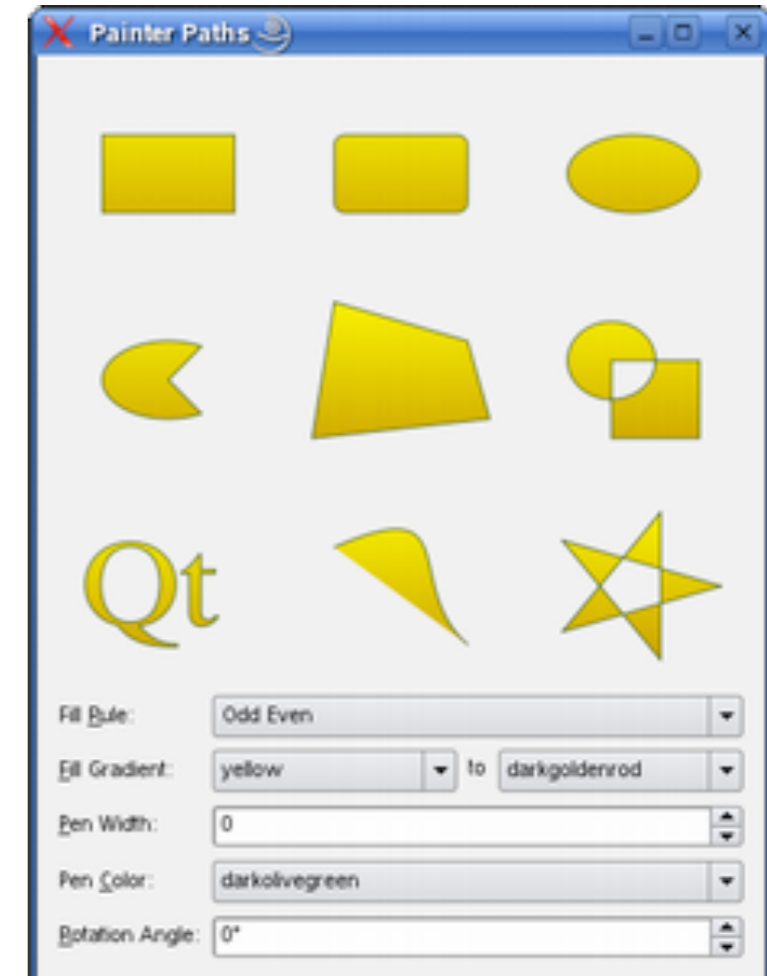
Paths

QPainterPath

- ▶ figure composée d'une suite arbitraire de lignes et courbes
 - affichée par: **QPainter.drawPath()**
 - peut aussi servir pour remplissage, profilage, découpage

Méthodes

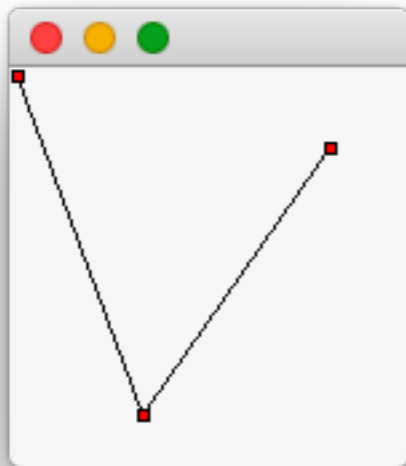
- ▶ **déplacements**: `moveTo()`, `arcMoveTo()`
- ▶ **dessin**: `lineTo()`, `arcTo()`
- ▶ **courbes** de Bezier: `quadTo()`, `cubicTo()`
- ▶ `addRect()`, `addEllipse()`, `addPolygon()`, `addPath()` ...
- ▶ `addText()`
- ▶ `translate()`, union, addition, soustraction...
- ▶ et d'autres encore ...



Path

exemples

```
#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)
    path = QPainterPath()           #instancie le path
    path.moveTo(3, 3)              # position initiale du path
    path.lineTo(50,130)            # Tracé droit
    path.lineTo(120,30)            # tracé droit
    path.cubicTo(120,30,60,50,3,3) # tracé bezier
    painter.drawPath(path)         #dessiner le path
```

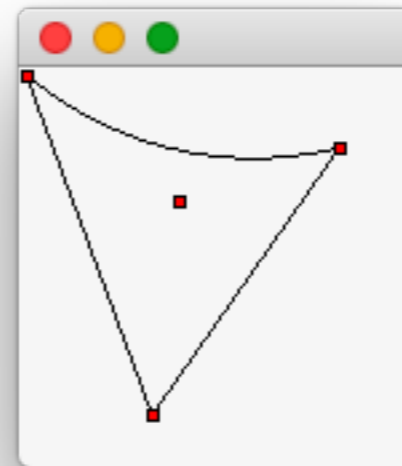
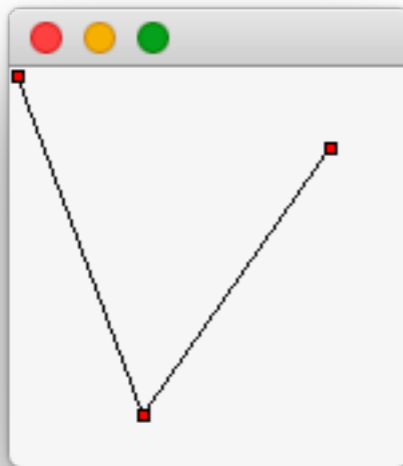


Path

exemples

```
#evenement QPaintEvent  
def paintEvent(self, event):  
    painter = QPainter(self)  
    path = QPainterPath()  
    path.moveTo(3, 3)  
    path.lineTo(50, 130)  
    path.lineTo(120, 30)  
  
    painter.drawPath(path)
```

```
#instancie le path  
# position initiale du path  
    # Tracé droit  
    # tracé droit  
  
#dessiner le path
```

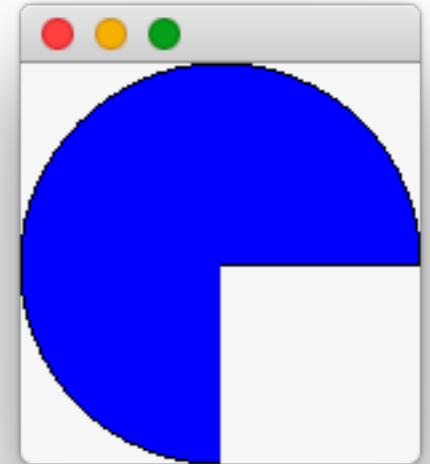


Paths

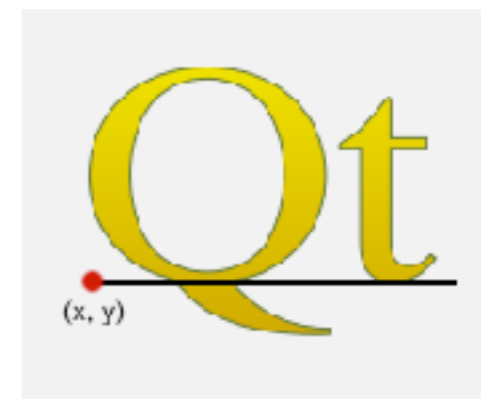
exemples

```
#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)
    center = QPointF(self.width()/2.0, self.height()/2.0)
    myPath = QPainterPath()
    myPath.moveTo( center )
    myPath.arcTo( QRectF(0,0, self.width(), self.height()), 0, 270 )

    painter.setBrush(Qt.blue)
    painter.drawPath( myPath )
```



```
#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)
    myPath = QPainterPath()
    myPath.addText(QPointF(40,60), QFont( 'SansSerif', 50), "Qt" )
    painter.drawPath(myPath)
```



Paths

```
#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)

    path = QPainterPath()
    path.addRect(20, 20, 60, 60)
    path.moveTo(0, 0)
    path.cubicTo(99, 0, 50, 50, 99, 99)
    path.cubicTo(0, 99, 50, 50, 0, 0)

    painter.fillRect(0, 0, 100, 100, Qt.white)
    painter.setPen( QPen(QColor(79, 106, 25), 1,
                        Qt.SolidLine, Qt.FlatCap, Qt.MiterJoin))
    painter.setBrush( QColor(122, 163, 39));
    painter.drawPath(path);
```



Qt.OddEvenFill
(default)

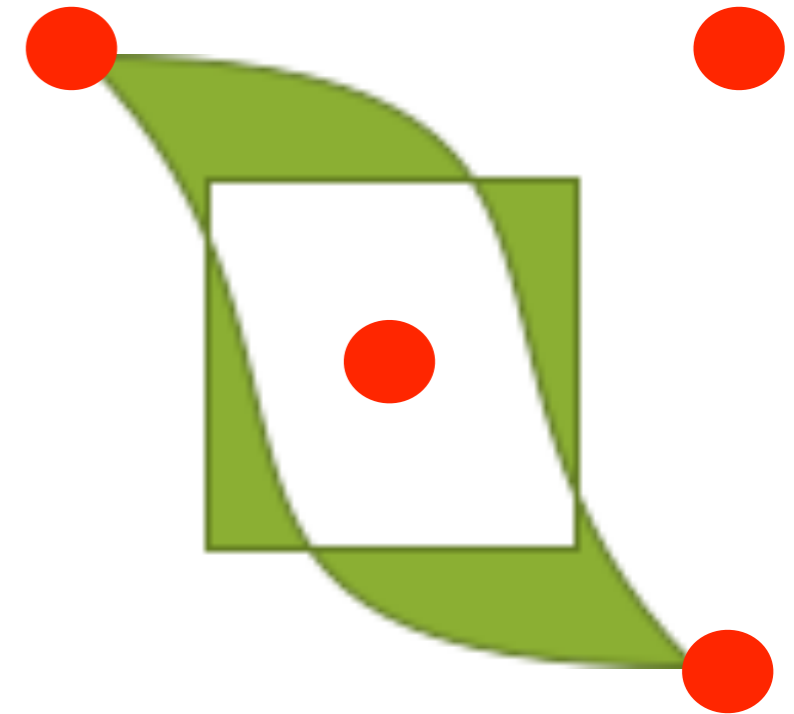


Qt.WindingFill



Paths

```
#evenement QPaintEvent  
def paintEvent(self, event):  
    painter = QPainter(self)  
  
    path = QPainterPath()  
    path.addRect(20, 20, 60, 60)  
    path.moveTo(0, 0)  
    path.cubicTo(99, 0, 50, 50, 99, 99)  
    path.cubicTo(0, 99, 50, 50, 0, 0)  
  
    painter.fillRect(0, 0, 100, 100, Qt.white)  
    painter.setPen( QPen(QColor(79, 106, 25), 1,  
                        Qt.SolidLine, Qt.FlatCap, Qt.MiterJoin))  
    painter.setBrush( QColor(122, 163, 39));  
    painter.drawPath(path);
```



Qt.OddEvenFill
(default)

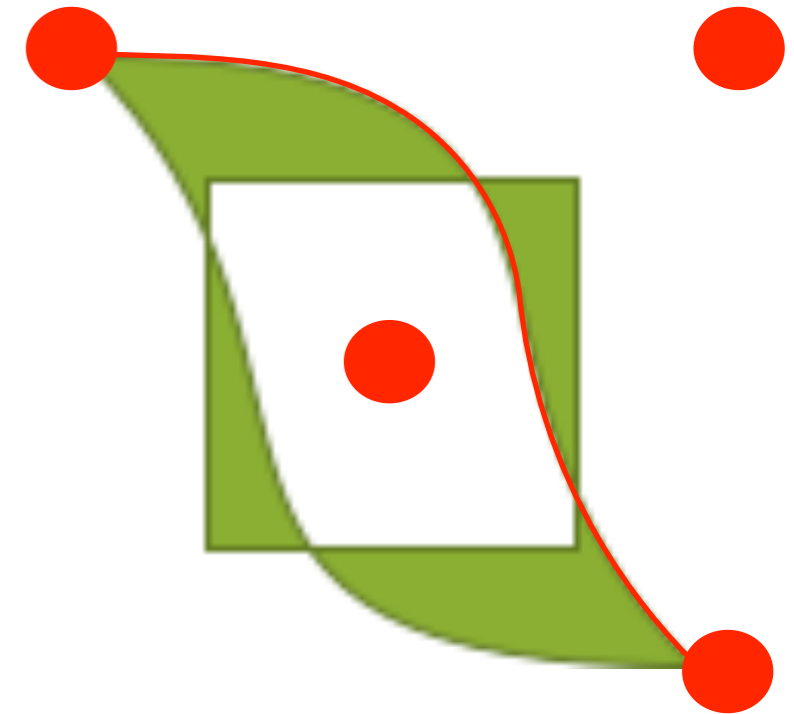


Qt.WindingFill

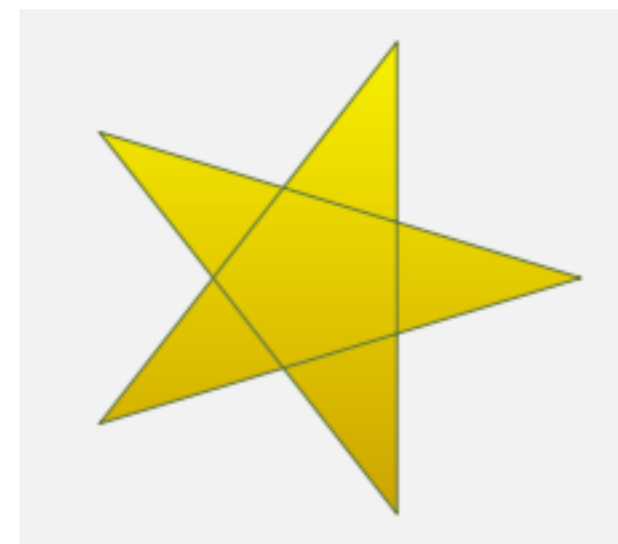


Paths

```
#evenement QPaintEvent  
def paintEvent(self, event):  
    painter = QPainter(self)  
  
    path = QPainterPath()  
    path.addRect(20, 20, 60, 60)  
    path.moveTo(0, 0)  
    path.cubicTo(99, 0, 50, 50, 99, 99)  
    path.cubicTo(0, 99, 50, 50, 0, 0)  
  
    painter.fillRect(0, 0, 100, 100, Qt.white)  
    painter.setPen( QPen(QColor(79, 106, 25), 1,  
                        Qt.SolidLine, Qt.FlatCap, Qt.MiterJoin))  
    painter.setBrush( QColor(122, 163, 39));  
    painter.drawPath(path);
```

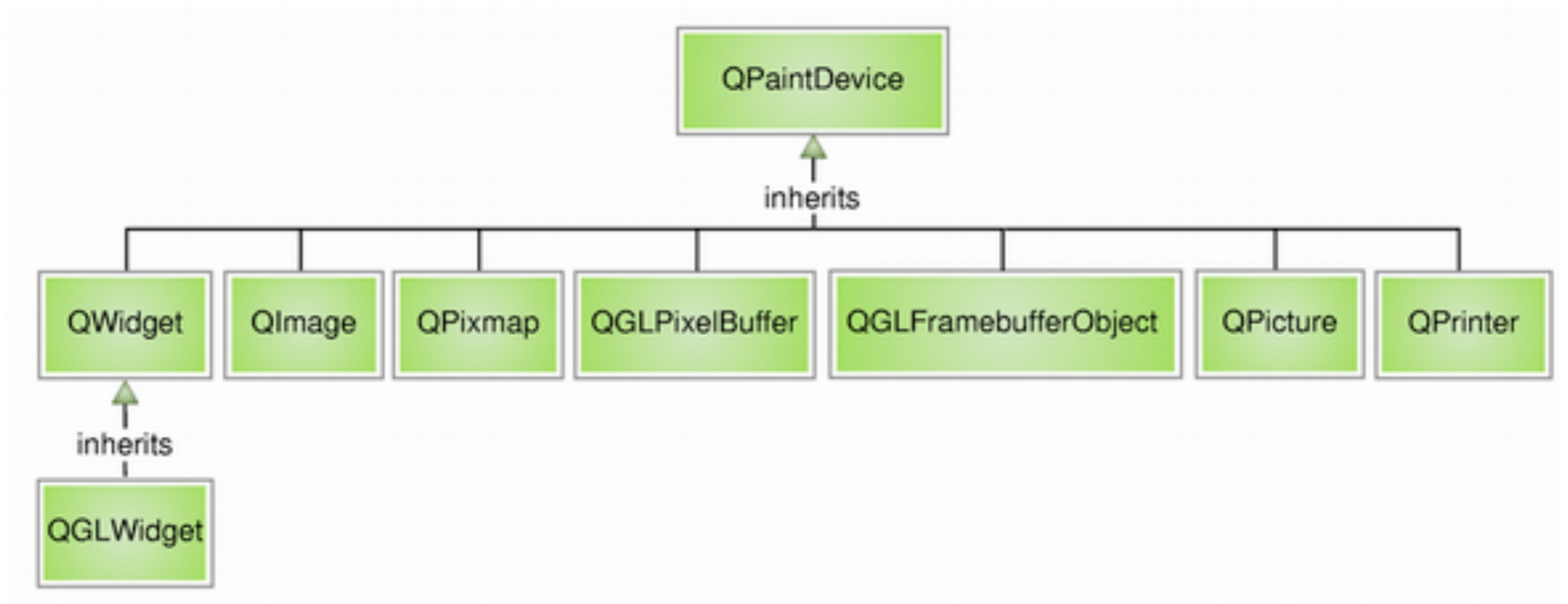


Qt.OddEvenFill
(default)



Qt.WindingFill

Surfaces d'affichage



Images

Types d'images

- ▶ **QImage**: optimisé pour E/S et accès/manipulation des pixels
 - **QPixmap, QBitmap** : optimisés pour affichage l'écran
- ▶ **QPicture**: pour enregistrer et rejouer les commandes d'un **QPainter**
- ▶ dans tous les cas : on peut dessiner dedans avec un **QPainter**

Entrées/sorties

- ▶ **load()** / **save()** : depuis/vers un fichier, principaux formats supportés
- ▶ **loadFromData()** : depuis la mémoire

Accès aux pixels

Format 32 bits : accès direct

```
image = QImage(3, 3, QImage.Format_RGB32)
```

```
value = QRgb(122, 163, 39) // 0xff7aa327  
image.setPixel(0, 1, value)  
image.setPixel(1, 0, value)
```

Format 8 bits : indexé

```
image = QImage(3, 3, QImage::Format_Indexed8)
```

```
value = QRgb(122, 163, 39) // 0xff7aa327  
image.setColor(0, value)
```

```
value = QRgb(237, 187, 51) // 0xffedba31  
image.setColor(1, value)
```

```
image.setPixel(0, 1, 0)  
image.setPixel(1, 0, 1)
```

| | | |
|------------|------------|------------|
| | 0xff7aa327 | |
| 0xff7aa327 | 0xffbd9527 | 0xffedba31 |
| | | |

| | | | | |
|---|---|---|---|------------|
| | 0 | | 0 | 0xff7aa327 |
| | | | 1 | 0xffedba31 |
| | | | 2 | 0xffbd9527 |
| 0 | | | | |
| | 2 | | | |
| | | 1 | | |
| | | | | |
| | | | | |
| | | | | |

Autres surfaces d'affichage

SVG

- ▶ QSvgWidget
 - QSvgRenderer

OpenGL

- ▶ QGLWidget
 - QGLPixelBuffer
 - QGLFramebufferObject

Impression

- ▶ QPrinter



QSvgWidget

#4 Interaction avec formes géométriques

Picking

Picking avec QRect, QRectF

- ▶ **intersects()**
- ▶ **contains()**

Picking avec QPainterPath

- ▶ **intersects**(const QRectF & rectangle)
- ▶ **intersects**(const QPainterPath & path)
- ▶ **contains**(const QPointF & point)
- ▶ **contains**(const QRectF & rectangle)
- ▶ **contains**(const QPainterPath & path)

Retourne l'intersection

- ▶ QPainterPath **intersected**(const QPainterPath & path)

Picking / Interaction

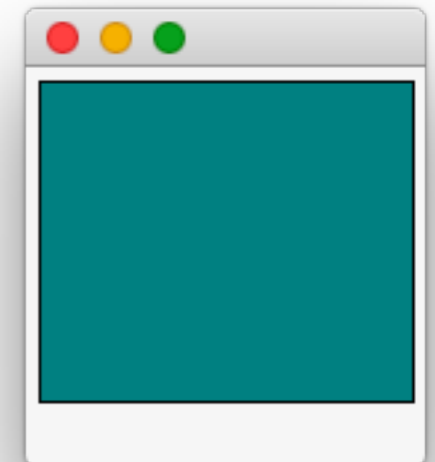
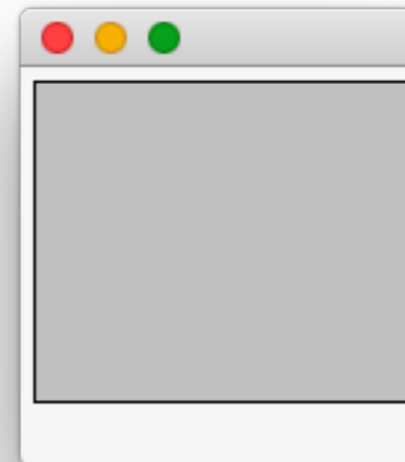
Exemple

- ▶ teste si la souris est dans le rectangle quand on appuie sur le bouton de la souris
- ▶ change la couleur du rectangle à chaque clique

```
def __init__(self):
    super().__init__()
    self.myBool = True # variable d'instance booleen
    self.rect = QRect(5,5,140,120) # variable d'instance rectangle

def mousePressEvent(self, event): # evenement mousePress
    self.pStart = event.pos()
    if self.rect.contains(event.pos()): # test la position
        self.myBool = not self.myBool
    self.update() # demande la MAJ du dessin

#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)
    if self.myBool:
        painter.setBrush(Qt.lightGray)
    else:
        painter.setBrush(QColor(Qt.darkCyan))
    painter.drawRect(self.rect)
```



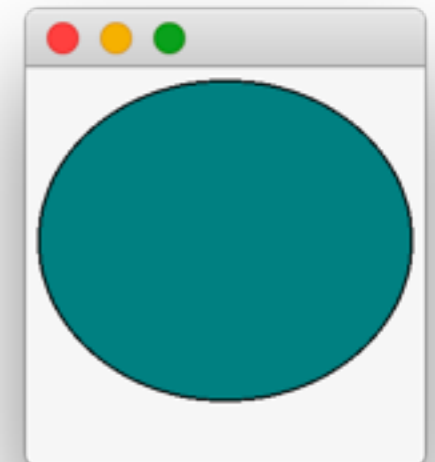
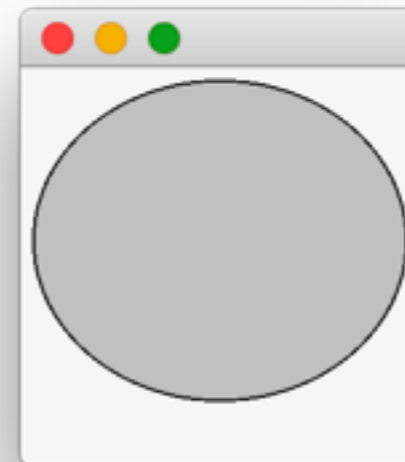
Formes non rectangulaires

Utilisation des **regions**

```
def __init__(self):
    super().__init__()
    self.myBool = True # variable d'instance booleen
    self.rect = QRect(5,5,140,120) # variable d'instance rectangle

def mousePressEvent(self, event): # evenement mousePress
    self.pStart = event.pos()
    ellipse = QRegion(self.rect,QRegion.Ellipse) # defini une region elliptique
    if ellipse.contains(event.pos()): # test la position
        self.myBool = not self.myBool
    self.update() # demande la MAJ du dessin

#evenement QPaintEvent
def paintEvent(self, event):
    painter = QPainter(self)
    if self.myBool:
        painter.setBrush(Qt.lightGray)
    else:
        painter.setBrush(QColor(Qt.darkCyan))
    painter.drawEllipse(self.rect)
```



#5 Performances de l'affichage

Performance de l'affichage

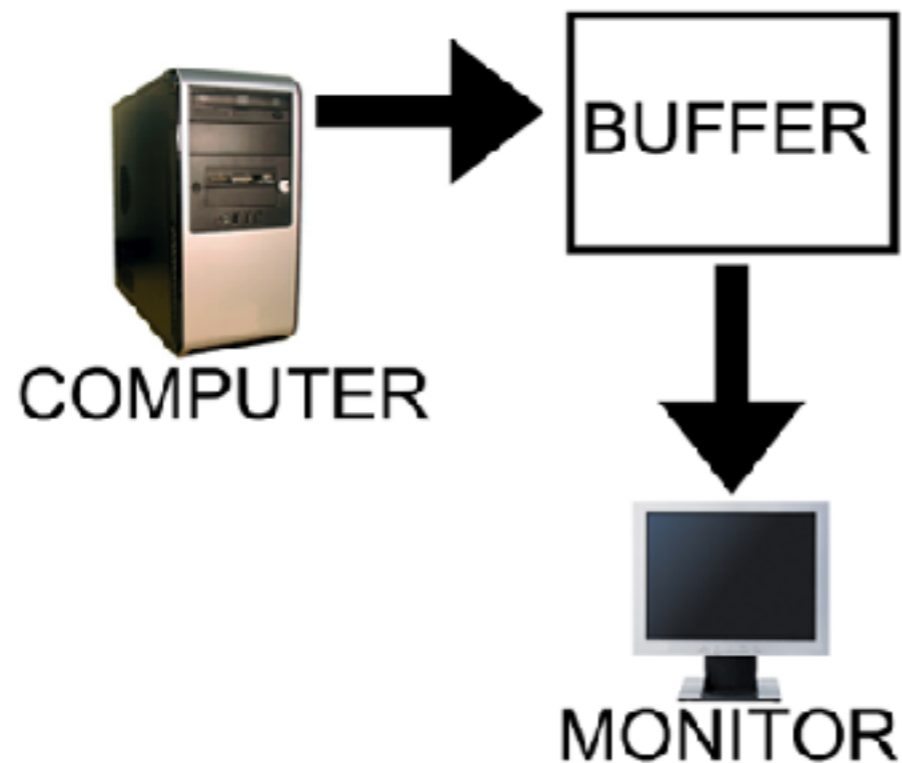
Problèmes classiques :

- ▶ **Flickering** et **tearing** (scintillement et déchirement)
- ▶ **Lag** (latence)

Flickering

Flickering

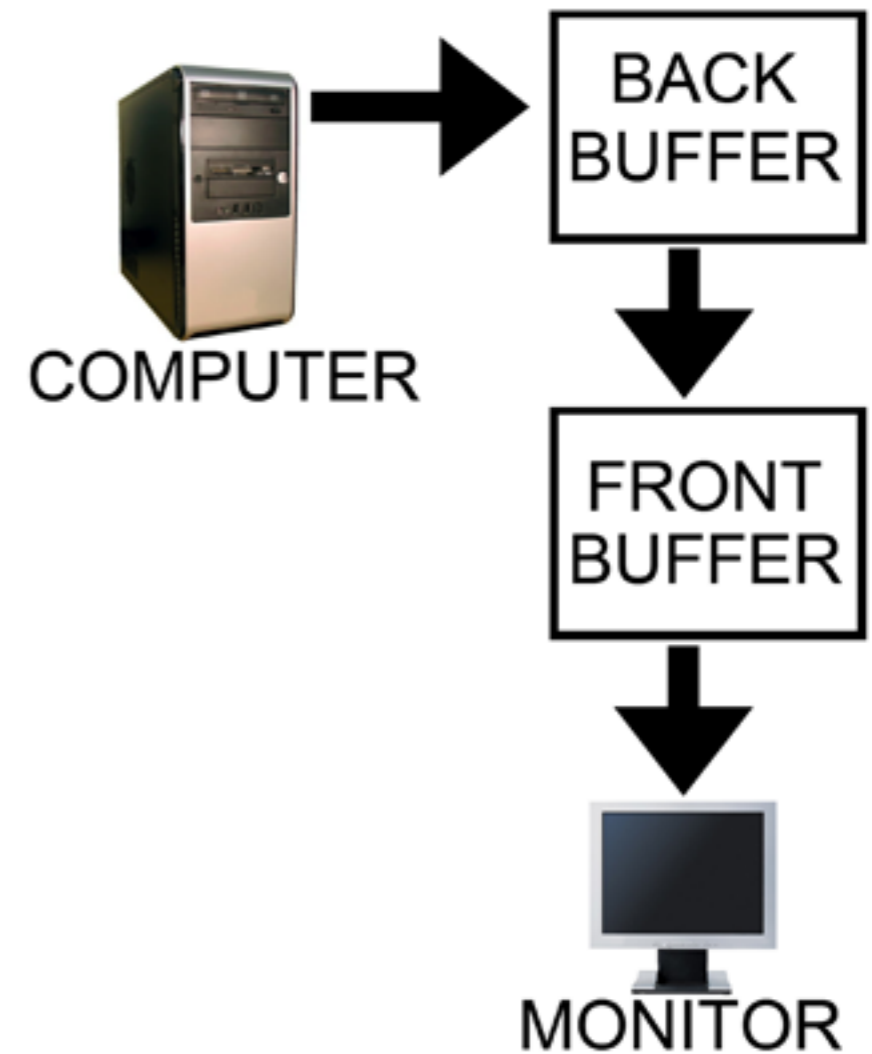
- ▶ scintillement de l'affichage car l'oeil perçoit les images intermédiaires
- ▶ exemple : pour rafraichir cette page il faut :
 - 1) tout « effacer » (repeindre le fond)
 - 2) tout redessiner
 - => scintillement si le fond du transparent est sombre alors que le fond de la fenêtre est blanc



Double buffering

Double buffering

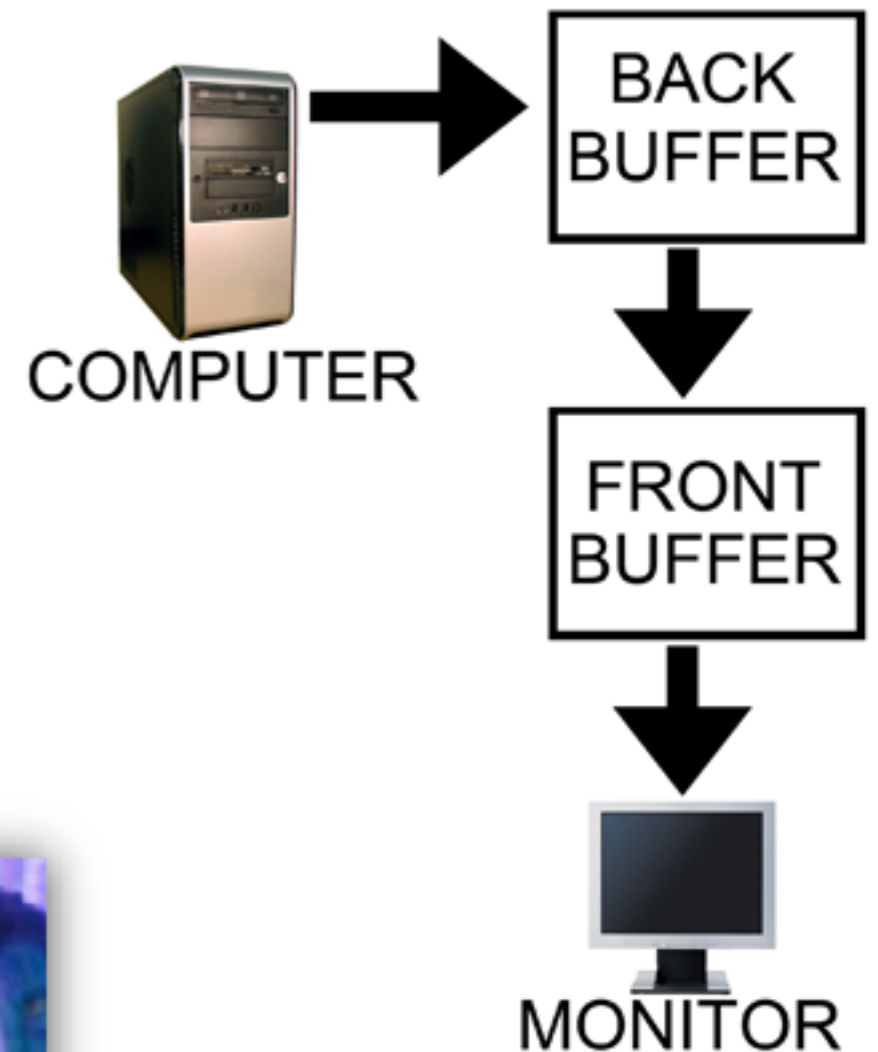
- ▶ solution au flickering :
 - dessin dans le back buffer
 - recopie dans le front buffer (le buffer vidéo qui contrôle ce qui est affiché sur l'écran)
- ▶ par défaut avec Qt4



Tearing

Possible problème : **Tearing**

- ▶ l'image apparait en 2 (ou 3...) parties horizontales
- ▶ problème : recopie du back buffer avant que le dessin soit complet
 - mélange de plusieurs "frames" vidéo
 - en particulier avec jeux vidéo et autres applications graphiquement demandantes



Example:

With Tearing



No Tearing

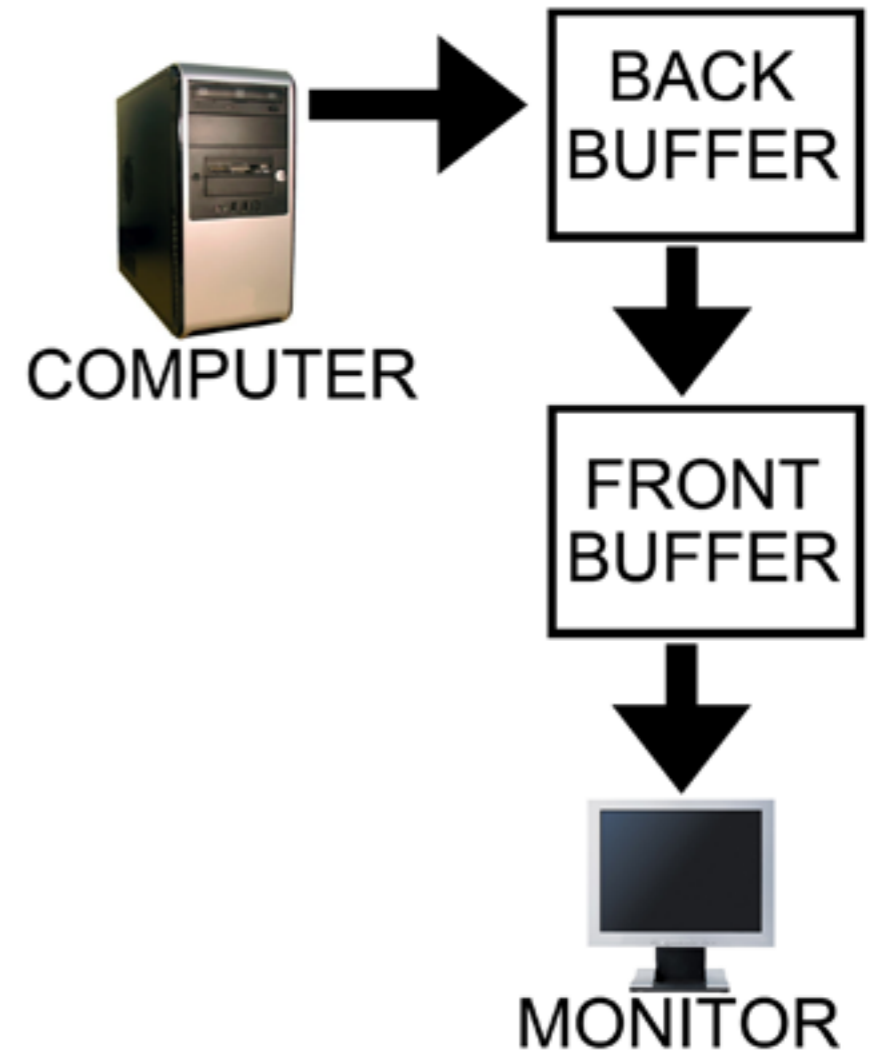


source: AnandTech

Tearing

Solution au **Tearing**

- ▶ **VSync** (Vertical synchronization)
- ▶ rendu synchronisé avec l'affichage
- ▶ inconvénient : ralentit l'affichage



source: AnandTech

Performance de l'affichage

Latence (lag)

- ▶ effet : l'affichage «ne suit pas» l'interaction
- ▶ raison : le rendu n'est pas assez rapide



Performance de l'affichage

Latence (lag)

- ▶ effet : l'affichage *ne suit pas* l'interaction
- ▶ raison : le rendu n'est pas assez rapide

Solutions

- ▶ afficher moins de choses :
 - dans l'espace
 - dans le temps
- ▶ afficher en mode XOR

Performance de l'affichage

Afficher moins de choses dans l'espace

- ▶ **Clipping** : réduire la zone d'affichage
 - méthodes `rect()` et `region()` de `QPaintEvent`
- ▶ "**Backing store**" ou équivalent
 - 1) copier ce qui ne change pas dans une **image**
 - 2) afficher cette image dans la zone de dessin
 - 3) afficher la partie qui change par dessus

Performance de l'affichage

Afficher moins de choses dans le temps

- ▶ sauter les images intermédiaires :
 - réafficher une fois sur deux... ou selon l'heure
- ▶ les timers peuvent être utiles (cf. **QTimer**)

