Graphics in PyQt

Sylvain Malacria

http://www.malacria.com/ mailto:

sylvain.malacria@inria.fr

Slides inspired Gilles Bailly

objectives

Introduction

- Signals and slots
- Bases PyQt
- The main Qt classes

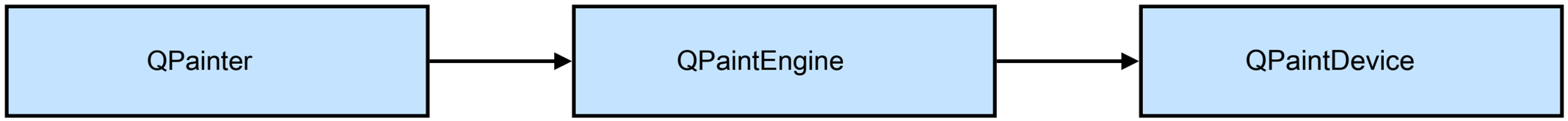
advanced graphics

- The **drawing** in PyQt
- Programming **event**
- Advanced Concept **graphics** Qt

1 The drawing in PyQt

Drawing in PyQt

Paint system

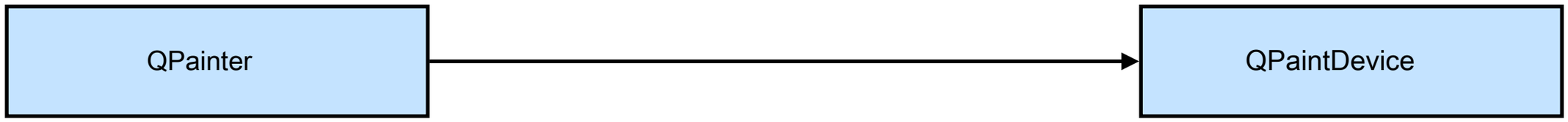


The API *painting* Qt enables *to paint* on the screen, to a file, etc. 3 main classes:

- **QPainter** for e ff ectuer drawing operations
- **QPaintDevice** 2D abstraction in which we draw
- **QPaintEngine** interface for connecting the two

Drawing in PyQt

Paint system



The API *painting* Qt enables *to paint* on the screen, to a file, etc. 3 main classes:

- **QPainter** for e ff ectuer drawing operations
- **QPaintDevice** 2D abstraction in which we draw
- **QPaintEngine** interface for connecting the two

QPaintEngine used internally (hidden) by QPainter and QPaintDevice

QPainter

QPainter is *the tool* of drawing

- simple lines
- path
- geometric shapes (ellipse, rectangle, etc.)
- text
- imagery
- etc.

Uses two main objects

- QBrush (for *fi ll)*
- QPen (for *stroke)*

main function is to draw, but has other functions to optimize rendering

Can draw on any object that inherits from the class *QPaintDevice*

Examples QPaintDevice

Base class in which we can *to paint* with QPainter

- **QWidget**
- QImage
- QPixmap
- QPicture
- QPrinter
- ...

Draw in a QWidget?

The widget "draws" when *repainted*

The widget is repainted when:

- a window passes over
- the window is resized
- asked explicitly
  - Repaint () forces the widget to be redrawn
  - Update (), a drawing event is added finally the wait

Draw in a QWidget?

The widget "draws" when *repainted*

The widget is repainted when:

- a window passes over
- the window is resized
- asked explicitly
  - Repaint () forces the widget to be redrawn
  - Update (), a drawing event is added finally the wait

In all cases, it is the method:

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

which is called (and you never have to manually call)

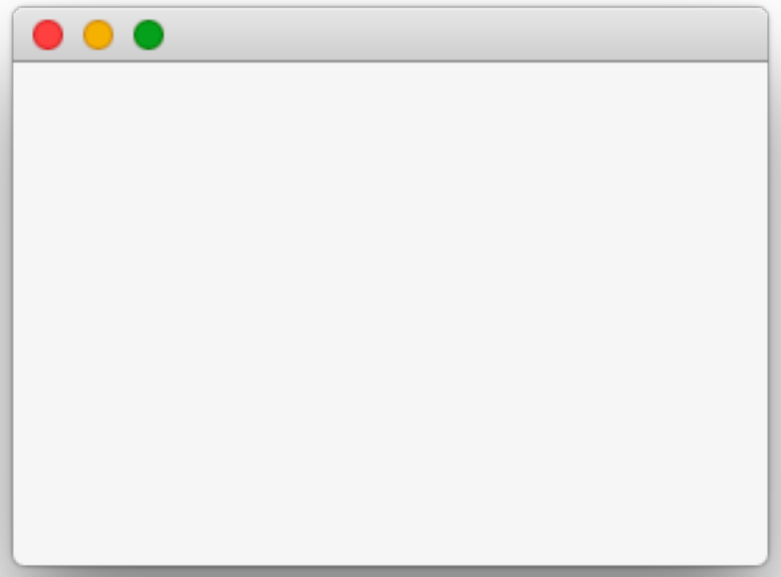

0, 0
X
the e


0, 0
X
1920, 0
the e
1080; 0
1920, 1080
acer

Draw in a QWidget

```
class Drawing ( QWidget):

        #event QPaintEvent
        def paintEvent ( self , Events):                    #   type of event QPaintEvent
                #   drawing blah here
```

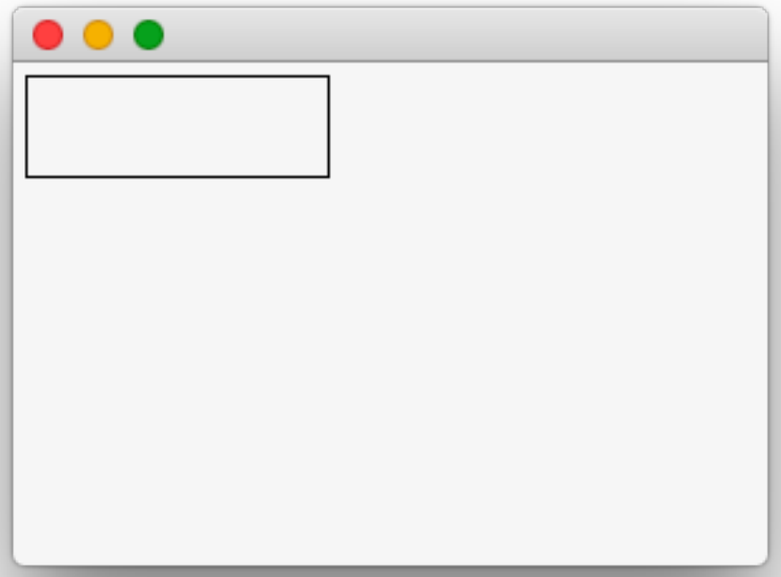
```
def hand (Args):
        app = QApplication (args) win = QMainWindow ()
        win . setCentralWidget (Draw ()) win . resize ( 300 , 200 )
        win . show () app . exec_ ()



        return if __ name__ == "__hand__" :


        hand (sys . argv)
```



Draw in a QWidget

```
class Drawing ( QWidget):

    #event QPaintEvent
    def paintEvent ( self , Events):                    #   type of event QPaintEvent
        painter = QPainter ( self )                     #   recovered the QPainter widget
        painter . drawRect ( 5 , 5 , 120 , 40 )         #   draw a black rectangle
        return
```

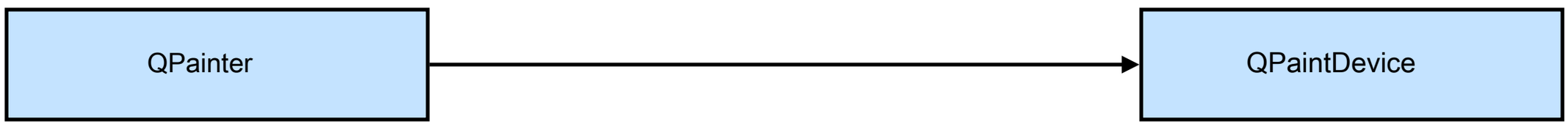
```
def hand (Args):
        app = QApplication (args) win = QMainWindow ()
        win . setCentralWidget (Draw ()) win . resize ( 300 , 200 )
        win . show () app . exec_ ()


        return if __ name__ == "__hand__" :


        hand (sys . argv)
```



Draw in a QWidget

Why it works?

```
class Drawing ( QWidget):

    #event QPaintEvent
    def paintEvent ( self , Events):                    #   type of event QPaintEvent
        painter = QPainter ( self )                     #   recovered the QPainter widget
        painter . drawRect ( 5 , 5 , 120 , 40 )         #   draw a black rectangle
        return
```



QPainter QWidget as drawing tool

inherits QPaintDevice

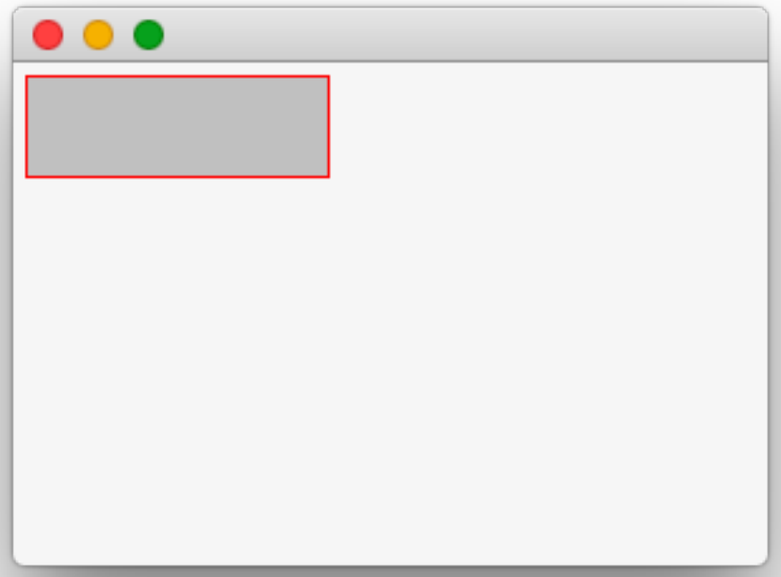
advanced drawing

QPainter:

- DrawLine () drawEllipse (), drawRect (), drawPath (), etc.
- llRect fi (), fi llEllipse ()
- drawText ()
- drawPixmap () drawImage ()
- SetPen () SetBrush ()

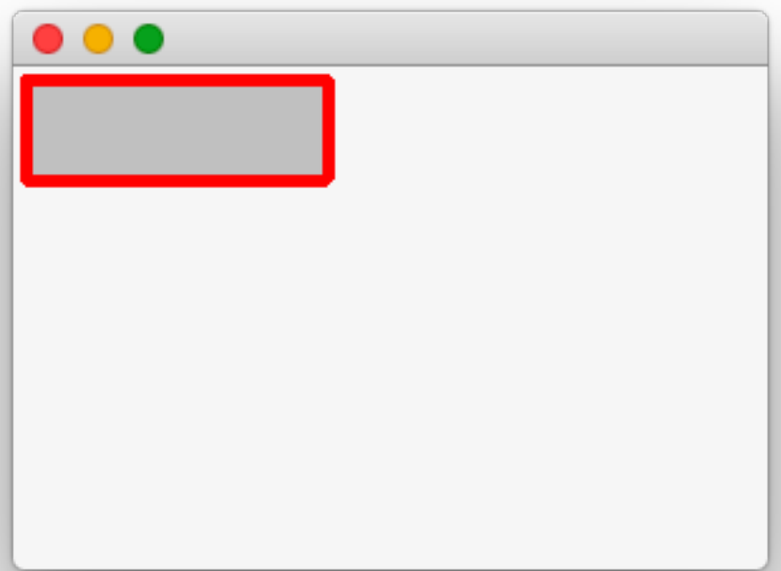
colored drawing

```
class Drawing ( QWidget):

    #event QPaintEvent
    def paintEvent ( self , Events): painter = QPainter ( self
            )                                                    #   recovered the QPainter widget
            painter . SetPen (Qt . red)                          #   add a red pen
            painter . SetBrush (Qt . lightgray)                  #   set a light gray brush
            painter . drawRect ( 5 , 5 , 120 , 40 )              #   draws the rectangle
```

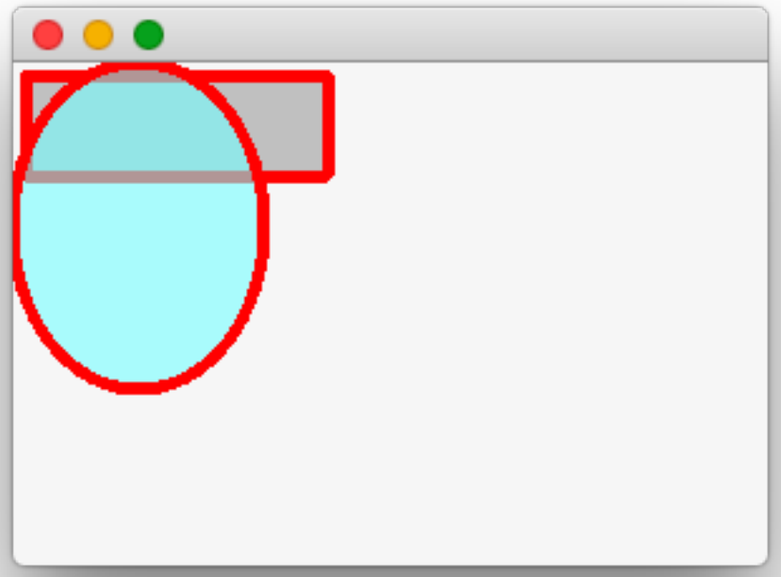


Instantiate a Pen

```
class Drawing ( QWidget):

    #event QPaintEvent
    def paintEvent ( self , Events): painter = QPainter ( self
            )                                                    #   recovered the QPainter widget
            pen = QPen (Qt . red)                                #   instantiate a pen
            pen . setWidth ( 5 )                                 #   changing the thickness
            painter . SetPen (pen)                               #   apply this pen the painter
            painter . SetBrush (Qt . lightgray)                  #   set a light gray brush
            painter . drawRect ( 5 , 5 , 120 , 40 )              #   draws the rectangle
```



```
class Drawing ( QWidget):

    #event QPaintEvent
    def paintEvent ( self , Events): 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

Where to draw?

- In paintEvent method (self, QPaintEvent) How to apply

the sheave ffi chage a widget

- update ()
- repaint ()

What is the di ff erence between update () and repaint ()?

- update () indicates that area is expensive ffi sheave (but a ffi drying is not instant)
- repaint () ffi sheave che immediately (but can introduce latency) How to draw in

paintEvent (QPaintEvent)

- instantiate a QPainter p = QPainter (self)

In what to draw?

All who inherits QPaintDevice

- QWidget
- QPrinter
- QPixmap
- QImage
- etc.

Rendering option to "high" (SVG)

- QSvgRenderer
- QSvgwidget

2 Events Manager

Management of mouse events in a QWidget

Methods that inherit QWidget

```
def mouseMoveEvent ( self , Events):

def mousePressEvent ( self , Events):

def mouseReleaseEvent ( self , Events):

def enterEvent ( self , Events):

def leaveEvent ( self , Events):
```

```
class Drawing ( QWidget):

    def mousePressEvent ( self , Events):                    #   event mousePress
        self . PStart = event . pos ()
        print ( " press " , self . PStart)

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

The methods are automatically called because the class inherits QWidget !!

```
class Drawing ( QWidget):

        def __ init__ ( self ):
                Great () . __init __ ()
                self . setMouseTracking ( true )                    # activate the "mouse tracking"

        def mouseMoveEvent ( self , Events):                        #   mouseMove event
                self . PStart = event . pos ()
                print ( " move " , self . PStart)
```

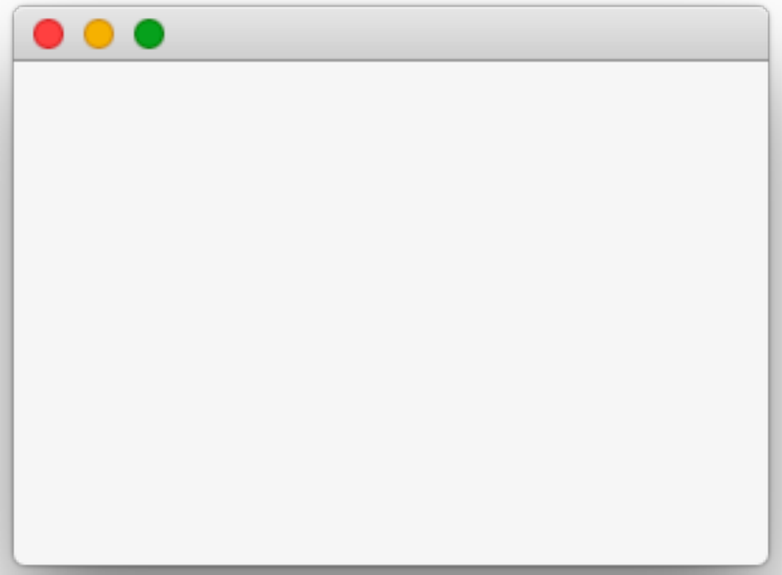
By default, *mouseMoveEvent* sent only if mouse button (*Drag)*

Possible to enable / disable constantly using ***setMouseTracking (bool)***

Example

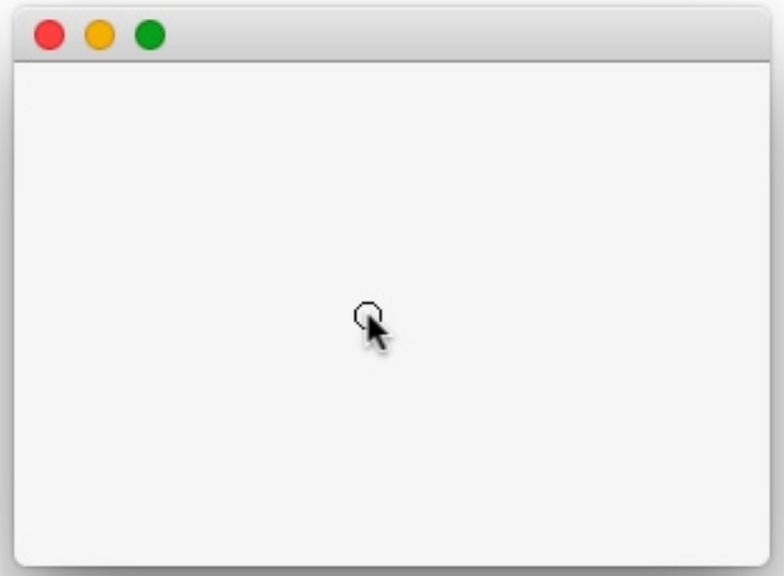
```
class Drawing ( QWidget):

        def __ init__ ( self ):
                Great () . __init __ ()
                self . setMouseTracking ( true )                    #   activates the mouseTracking
                self . cursorPos = None

        def mouseMoveEvent ( self , Events):                        # MouseMove event
                self . cursorPos = event . pos ()                   #   storing the cursor position
                self . update ()                                    #   one updates the display

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

                yew self . cursorPos ! = None :
                        painter . drawEllipse (\
                                self . cursorPos . x () -5 , \
                                self . cursorPos . y () -5 , 10 , 10 )      #   the ellipse is drawn around the cursor
```

Example



```
class Drawing ( QWidget):

        def __ init__ ( self ):
                Great () . __init __ ()
                self . setMouseTracking ( true )                    # activates the mouseTracking
                self . cursorPos = None

        def mouseMoveEvent ( self , Events):                        # MouseMove event
                self . cursorPos = event . pos ()                   # storing the cursor position
                self . update ()                                    # one updates the display

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

                yew self . cursorPos ! = None :
                        painter . drawEllipse (\
                                self . cursorPos . x () -5 , \
                                self . cursorPos . y () -5 , 10 , 10 )      # the ellipse is drawn around the cursor
```

Example



```
class Drawing ( QWidget):

    def __ init__ ( self ):
        Great () . __init __ ()
        self . setMouseTracking ( true )                    #   activates the mouseTracking
        self . cursorPos = None

    def mouseMoveEvent ( self , Events):                    # MouseMove event
        self . cursorPos = event . pos ()                   #   storing the cursor position
        self . update ()                                    #   one updates the display

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

        yew self . cursorPos ! = None :
            painter . drawEllipse (\
                self . cursorPos . x () -5 , \
                self . cursorPos . y () -5 , 10 , 10 )      #   the ellipse is drawn around the cursor
```

QMouseEvent

Of type QMouseEvent

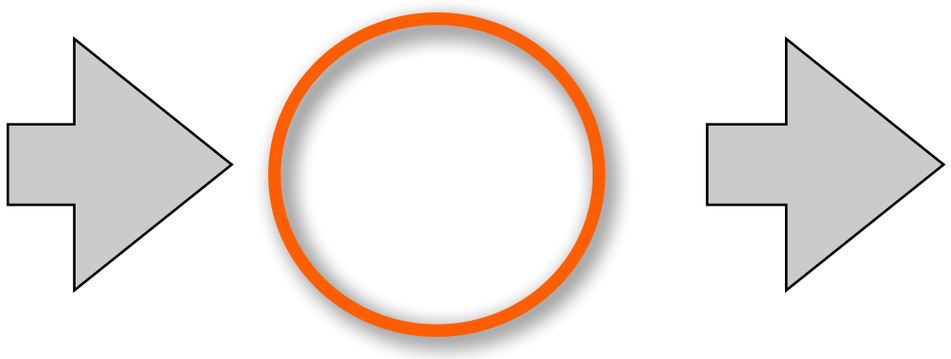
```
def mousePressEvent ( self , Events):
```

QMouseEvent retrieves (depending on version):

- button () : mouse button that triggered the event. ex: Qt.Le # Button
- buttons () : status of other buttons. ex: Qt.Le # Button | Qt.MidButton
- modified ers () : keyboard modifiers. ex: Qt.ControlModi fi ed | Qt.Shi # Altering
- pos () : local position (relative to the widget)
- globalPos () WINDOWPOS () screenPos () : or overall position on this reference
  - useful if you move the widget interactively!

Synthesis

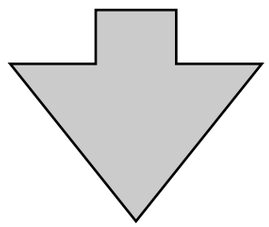
events



event management loop

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

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



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

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

3 advanced drawing

QPainter

attributes

- **SetPen ( ):** lines and contours

- **SetBrush ( ):** filling

- **setFont ( ):** text

- **setTransform ( )** etc. : Transformations ffi nes

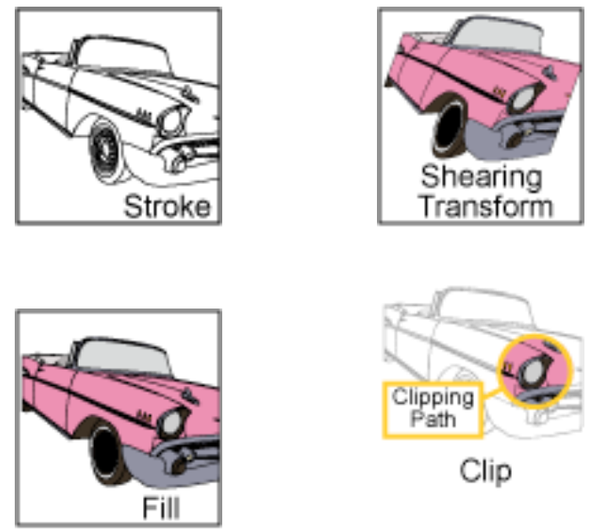
- **setClipRect / Path / Region ( ):** clipping (cutting)

- **setCompositionMode ( ):** composition

QPainter

Lines and contours

- drawPoint (), drawPoints ()
- drawLine (), drawLines ()
- drawRect (), drawRects ()
- drawArc (), drawEllipse ()
- drawPolygon (), drawPolyline (), Etc ...
- **drawPath ():** complex path



Filling

- fi llRect (), **fi llPath ()**

Various

- drawText ()
- drawPixmap (), drawImage (), DrawPicture ()
- etc.

QPainter

useful classes

- integers: **QPoint, QLine, QRect, QPolygon**
- fl oats: **QPointF, QLineF , ...**
- complex way: **QPainterPath**
- ffi drying zone has: **QRegion**

Brush: QPen

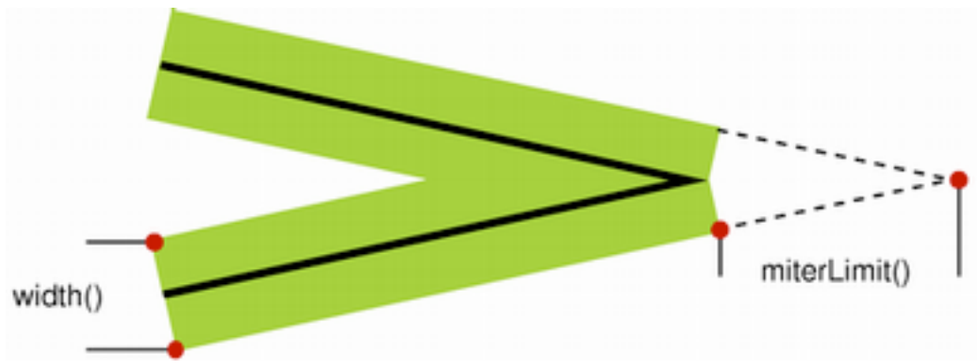
attributes

- **style** : line type
- **width** : thickness
- **brush** : brush attributes (color ...)
- **capStyle** : endings
- **joinStyle** : joints

PenStyle

Cap Style

Join Style



Brush: QPen

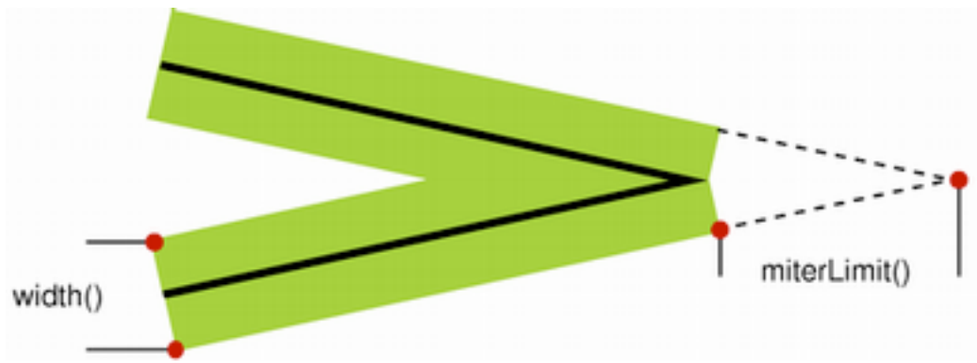
Example

```
// in paintEvent () method

pen = QPen ()                                   // default pen
pen. setStyle (Qt.DashDotLine) pen. setWidth (3)
pen. SetBrush (Qt.green) pen. setCapStyle (Qt.RoundCap)
pen. setJoinStyle (Qt.RoundJoin)

= QPainter painter (self) painter. SetPen (Pen)
```

PenStyle

Cap Style

Join Style

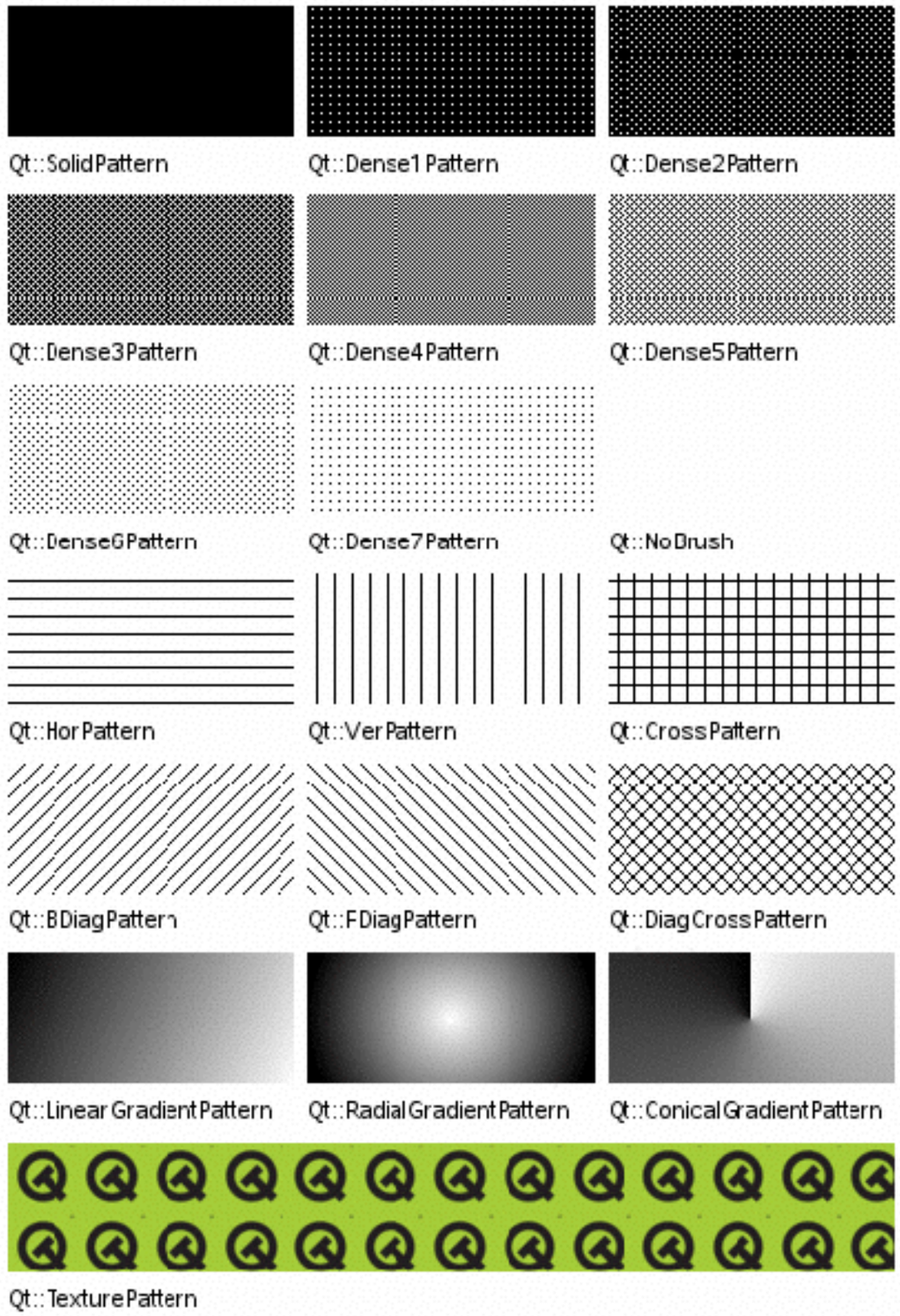


Filling: QBrush

attributes

- style
- color
- gradient
- texture

BrushStyle



brush QBrush = (...)

. . . . . .

= QPainter painter (self) painter. SetBrush

(Brush)

Filling: QBrush

attributes

- style
- color
- gradient
- texture

QColor

Qt.GlobalColor

- **models** RGB, HSV or CMYK
- **alpha component (** transparency):
  - alpha blending
- **predefined colors:**
  - Qt.GlobalColor

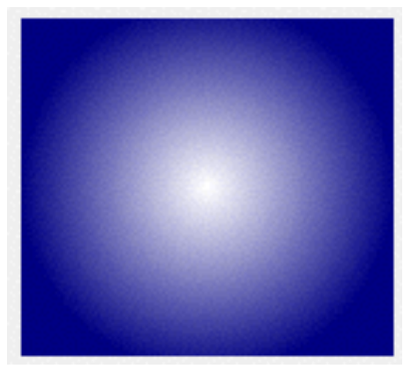
Filling: gradients

Type gradients

- linear,
- radial
- conical

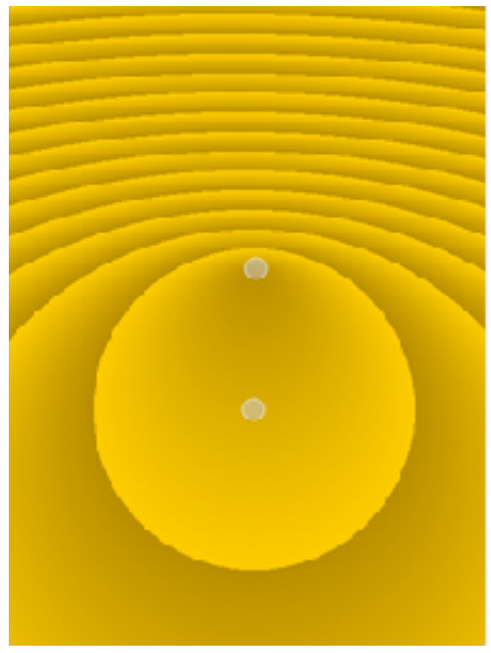
```
gradient = QLinearGradient (QPointF ( 0 , 0 ) QPointF ( 100 , 100 )) Gradient . setColorAt ( 0 Qt . white)
gradient . setColorAt ( 1 Qt . blue) painter . SetBrush (gradient)
```

QLinearGradient



QRadialGradient

QConicalGradient

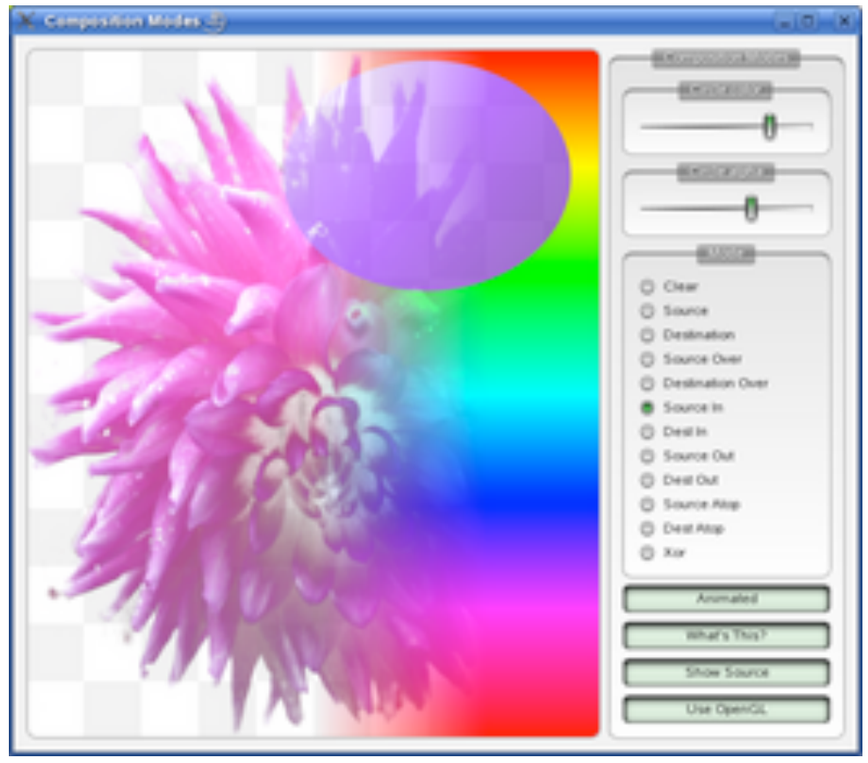


repetition: setSpread ()

Composition

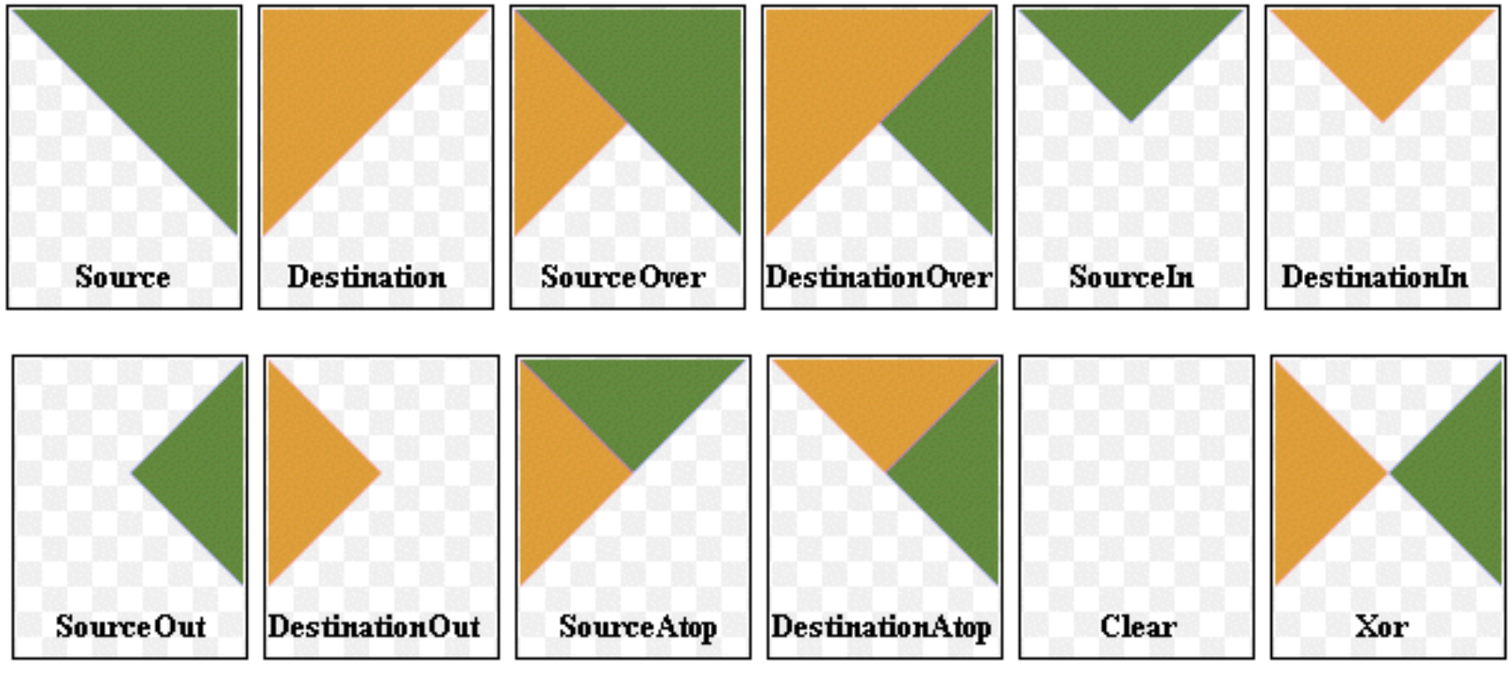
Dialing Modes

- operators **Porter Du ff:**
- define: *F (source, destination)*
- default: **SourceOver**
  - with alpha blending
  - **dst <= a** $_{src}$ * **src + (1-a** $_{src}$**)** * **at** $_{dst}$ * **dst**
- limitations
  - by implementation and Paint Device



Method:

setCompositionMode ()



Cutting (clipping)

Cutting

- as a rectangle, a region or a path
- methods QPainter
  setClipping (), setClipRect (), setClipRegion (), setClipPath ()

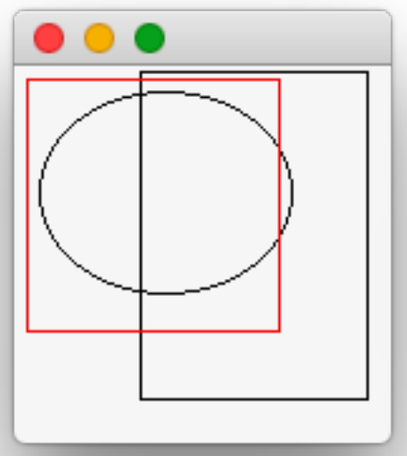
QRegion

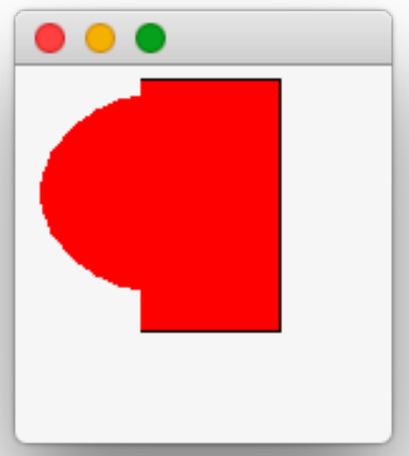
- united (), intersected (), Subtracted (), XORed ()

```
r1 = QRegion (QRect (100, 100, 200, 80) QRegion.Ellipse)        # R1: elliptic area
r2 = QRegion (QRect (100, 120, 90, 30))                         #   r2: rectangular area
r3 = r1. intersected ( r2);                                     # R3: intersection

= QPainter painter (self); painter. setClipRegion (R3, Qt.ReplaceClip);


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

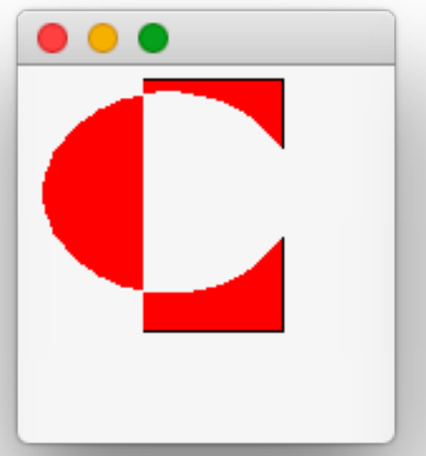
example

```
def paintEvent ( self , Events): 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 of
        regions



        r2 = QRegion (rect2) rc = r1 . Subtracted
        (r2)                                                    #   combination of regions

        painter . setClipRegion (rc)                            #   attributed the clipregion
        painter . drawRect (rect3)                              # We draw
```

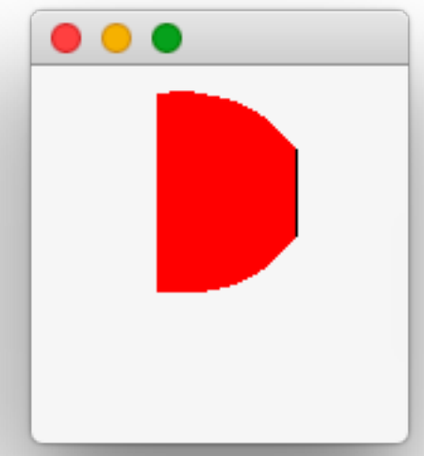




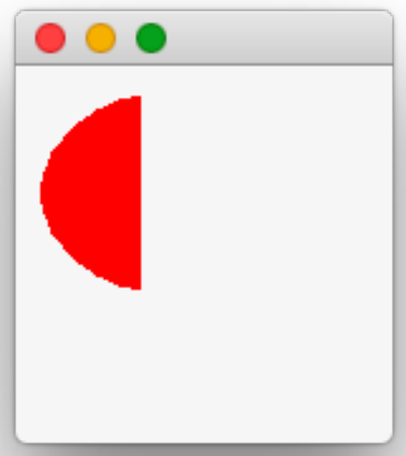
united ()



xor ()



intersected ()

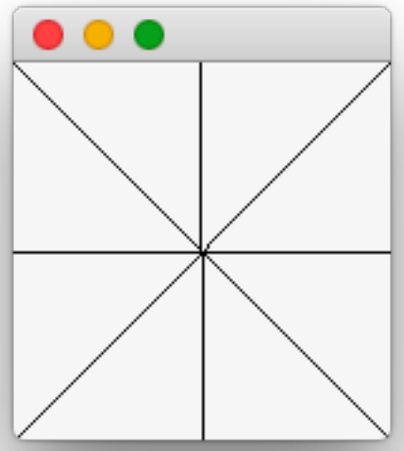


Subtracted ()

Transformations ffi nes

transformations

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



```
# event QPaintEvent
def paintEvent ( self , Events): painter = QPainter ( self
    ) painter . save ()                                          #  stacking the current state

    painter . translate ( self . width () / 2 ,
                                  self . height () / 2 )         #  one "center" the painter

    for k in tidy ( 0 , 10 ): Painter . drawLine ( 0 , 0 , 400 , 0 )

                                                                 #  draws a line
        painter . rotate ( 45 )                                  # 45 ° rotation

    painter . restore ()                                         #  pops the current state
```

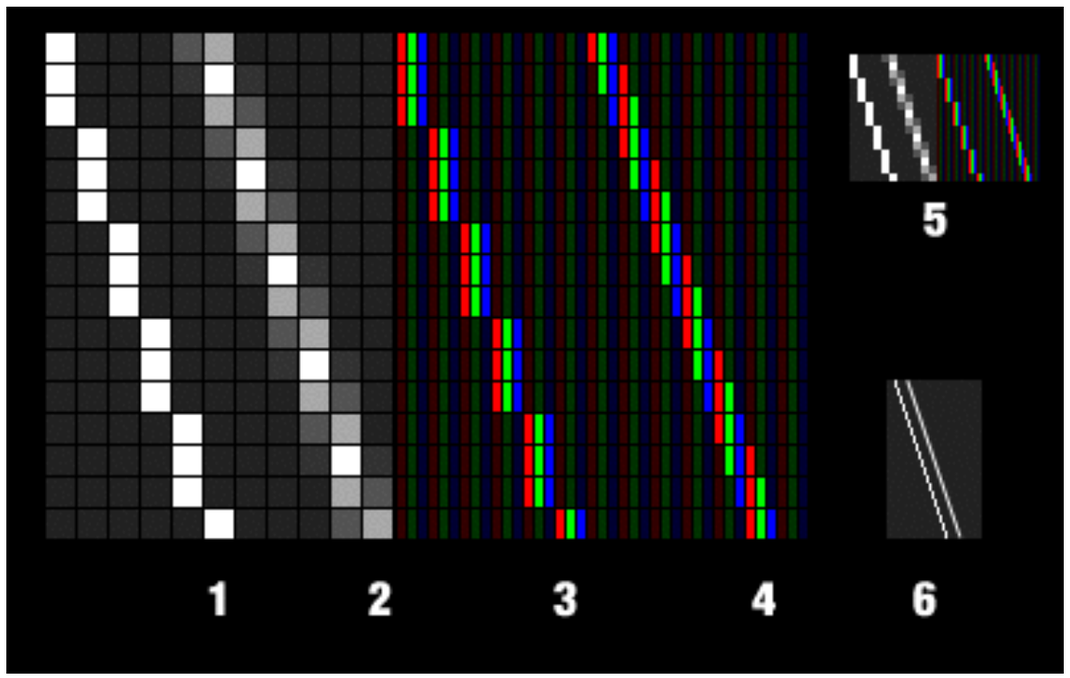
Anti-aliasing

Anti-aliasing

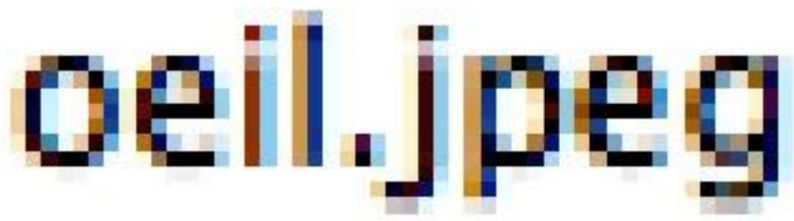
- avoid the e ff ect of stairs
- especially useful for fonts

subpixel rendering

- Examples: ClearType text MacOSX



ClearType
(Wikipedia)



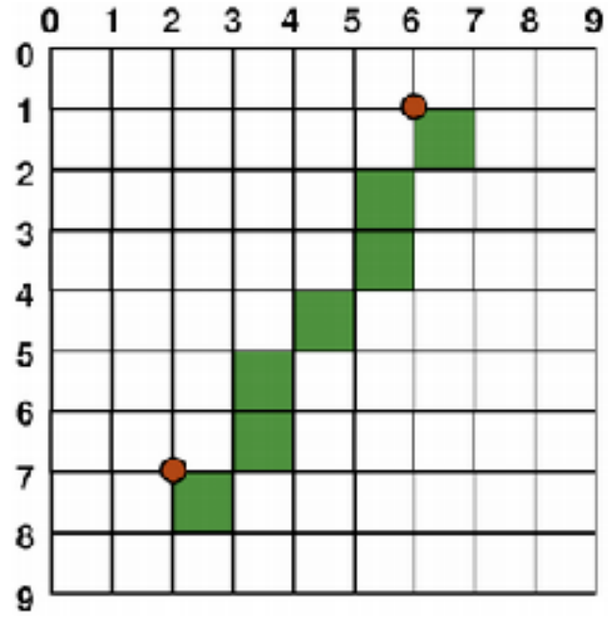
MacOSX

Anti-aliasing

Anti-aliasing in Qt

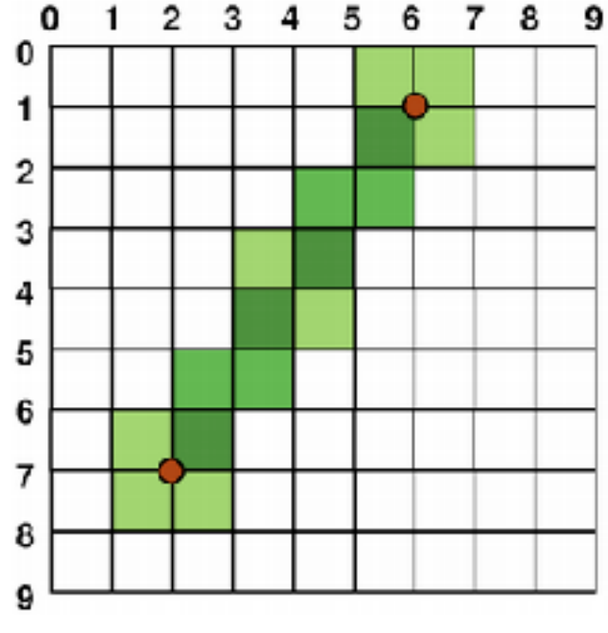
QPainter painter (this); painter. setRenderHint (QPainter.Antialiasing); painter. SetPen

(Qt.darkGreen); painter. drawLine (2, 7, 6, 1);



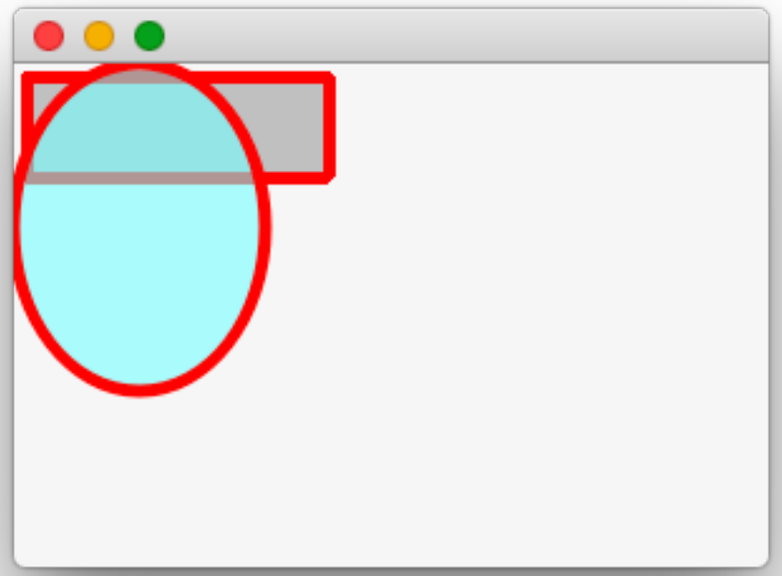
rendering hints

- "Hint" = rendering option
  - eff and unsecured
  - depends on the implementation and equipment
- method **setRenderingHints ( )** of **QPainter**

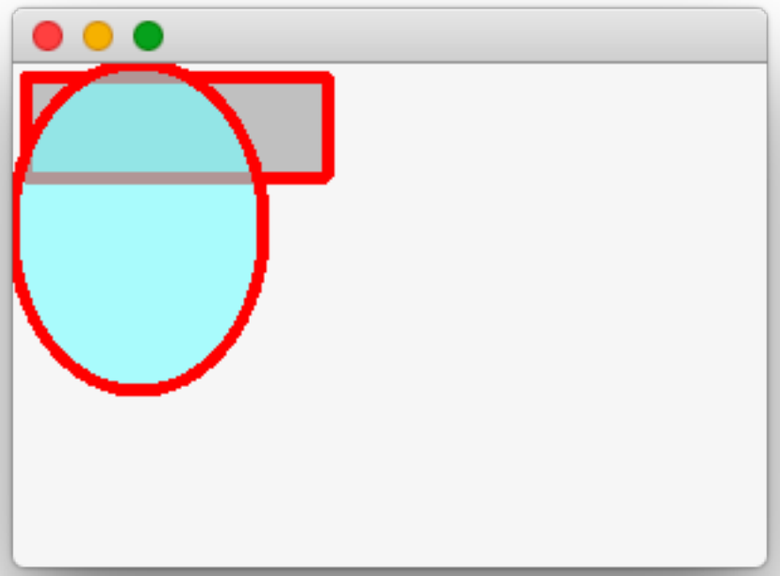


Anti-aliasing

```
painter . setRenderHints (QPainter . antialiasing)
```



With



Without

Antialiasing and coordinates

odd Thicknesses

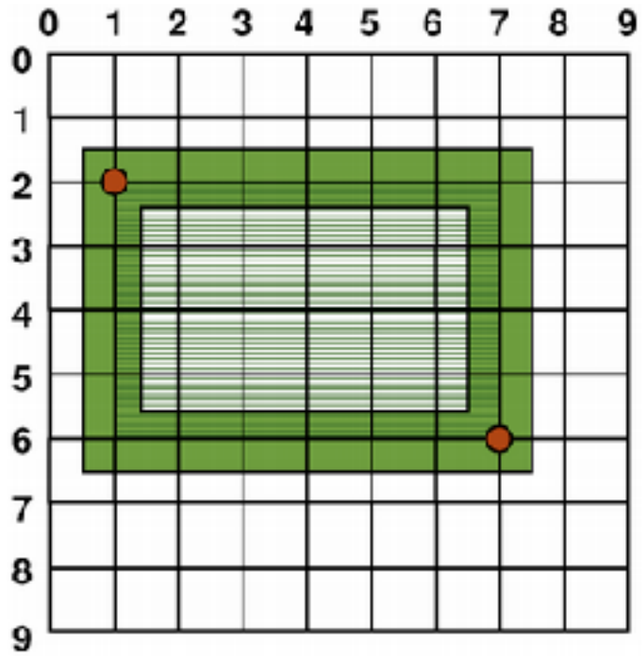
- pixels drawn to the right and below

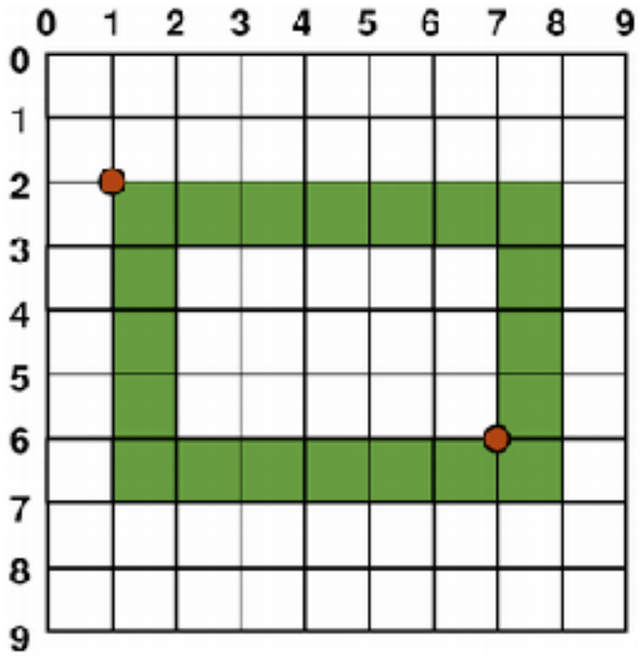
antialiasing drawing

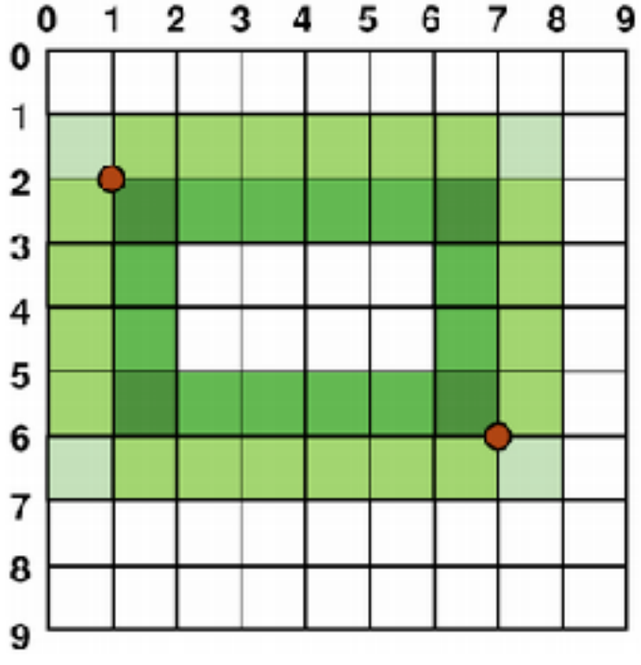
- pixels distributed around the ideal line

QRect. right () = left () + width () - **1 QRect.** bottom () top = () + height () - **1**

better: **QRectF (** floating)







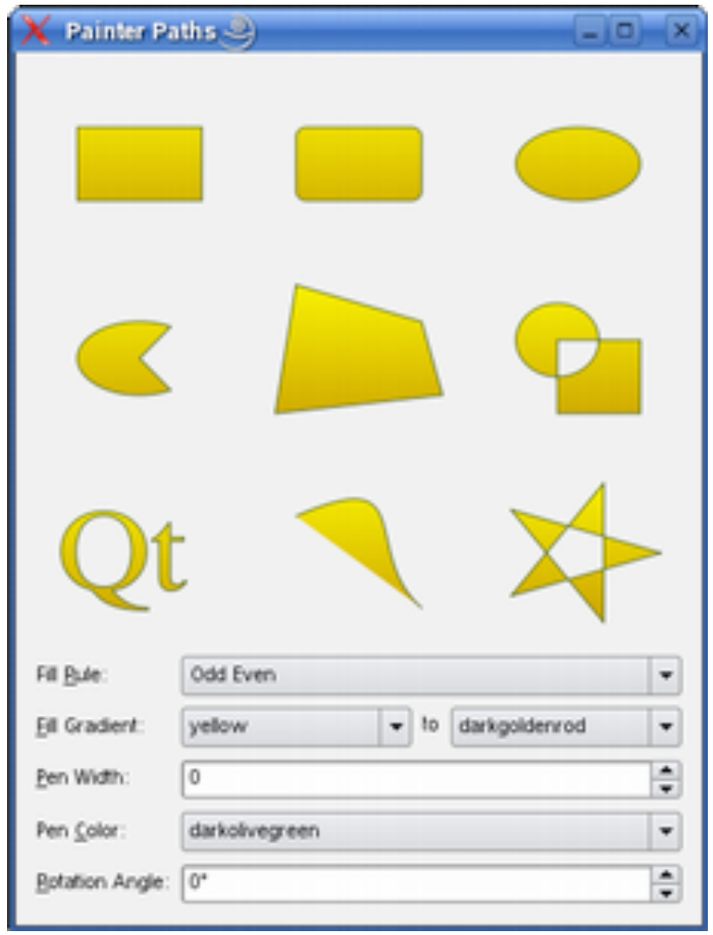
Paths

QPainterPath

- Figure composed of an arbitrary sequence of lines and curves
  - a ffi chée by: **QPainter. drawPath ()**
  - can also be used for filling, profiling cutting

Methods

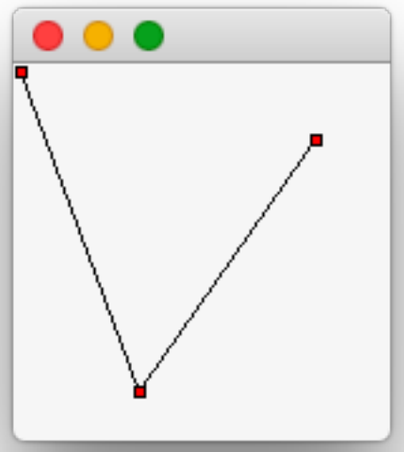
- **travel:** moveTo (), arcMoveTo ()
- **drawing:** lineTo (), arcTo ()
- **curves** Bezier: quadTo (), cubicTo ()
- addRect (), addEllipse (), addPolygon (), addPath () ...
- addText ()
- translate (), Union, addition, subtraction ...
- and even more ...



Path

examples

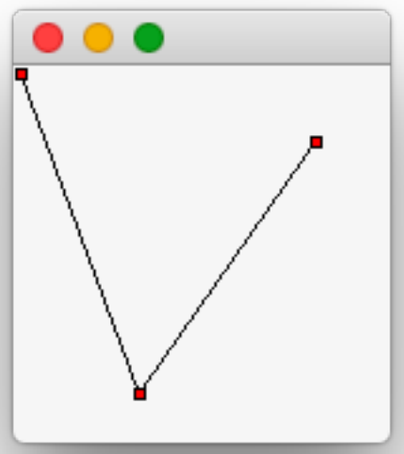
```
    # event QPaintEvent
    def paintEvent ( self , Events): painter = QPainter ( self
        ) path = QPainterPath ()
                                                    # instantiates the path
        path . moveTo ( 3 , 3 )                     #   initial position of the path
        path . lineTo ( 50 , 130 )                      # Right Path
        path . lineTo ( 120 , 30 )                      # Right track
        path . cubicTo ( 120 , 30 , 60 , 50 , 3 , 3 ) # bezier path
        painter . drawPath (path)                   # draw the path
```

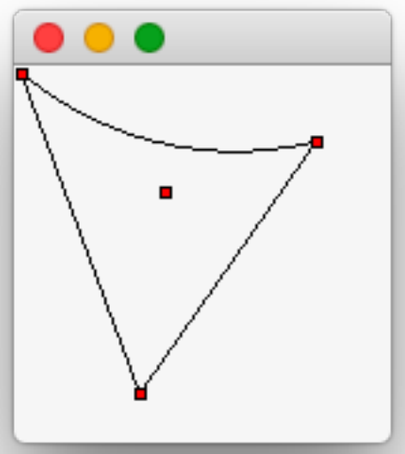


Path

examples

```
    # event QPaintEvent
    def paintEvent ( self , Events): painter = QPainter ( self
        ) path = QPainterPath ()
                                                    # instantiates the path
        path . moveTo ( 3 , 3 )                     #   initial position of the path
        path . lineTo ( 50 , 130 )                      # Right Path
        path . lineTo ( 120 , 30 )                      # Right track
        path . cubicTo ( 120 , 30 , 60 , 50 , 3 , 3 ) # bezier path
        painter . drawPath (path)                   # draw the path
```





Paths

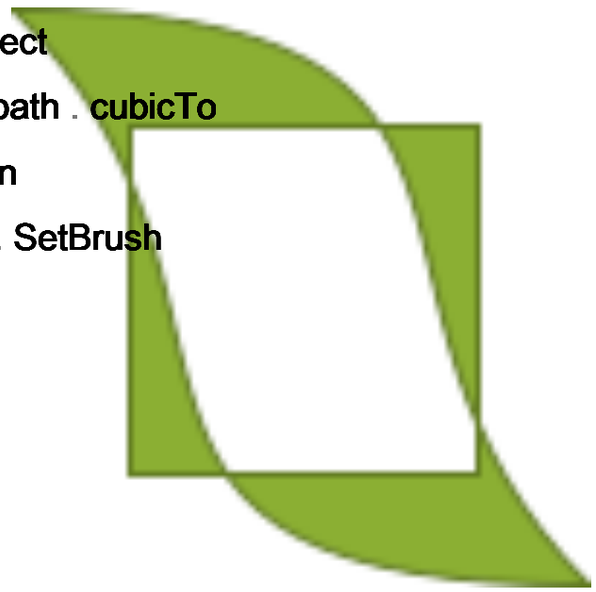
examples

```
#event QPaintEvent
def paintEvent ( self , Events): 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)
```

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

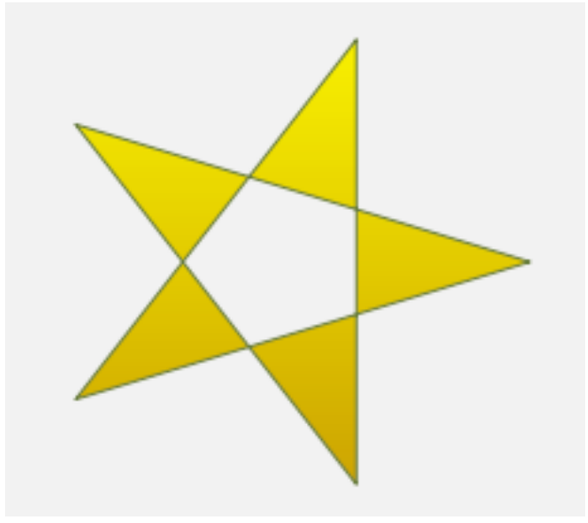
Paths

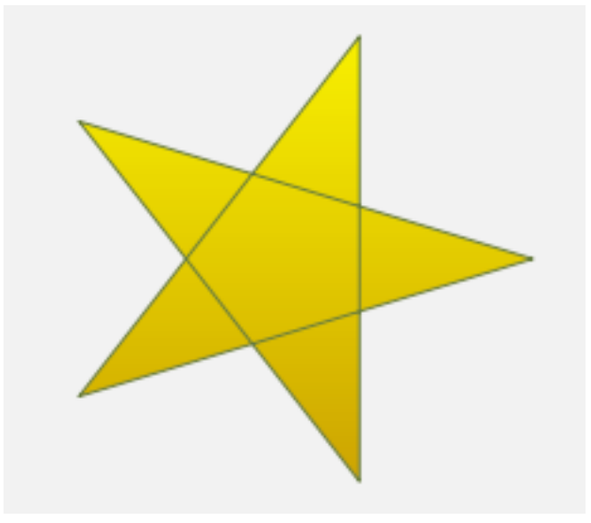
```
#event QPaintEvent
def paintEvent ( self , Events): painter = QPainter ( self ) path = QPainterPath () path . addRect
        ( 20 , 20 , 60 , 60 ) path . moveTo ( 0 , 0 ) path . cubicTo ( 99 , 0 ,  50 , 50 ,  99 , 99 ) path . cubicTo
        ( 0 , 99 ,  50 , 50 ,  0 , 0 ) painter . fillRect ( 0 , 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)

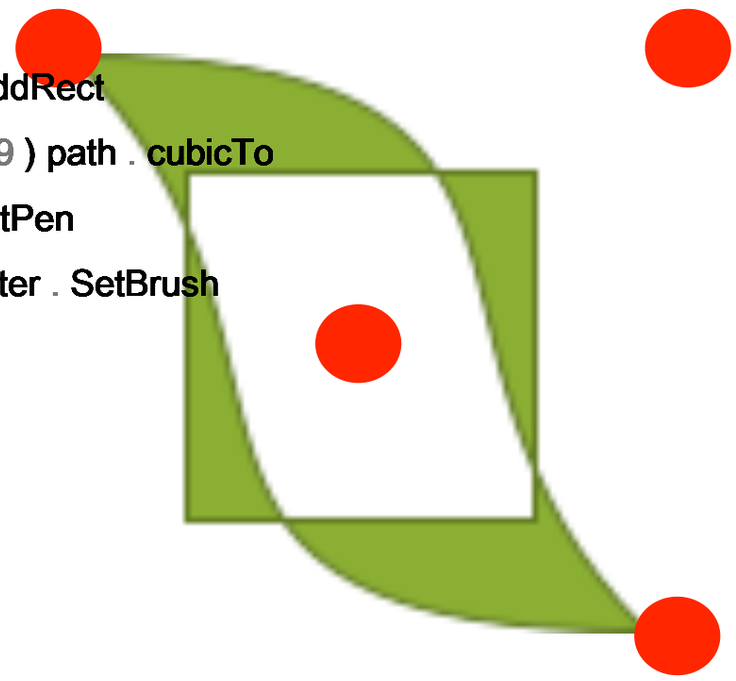




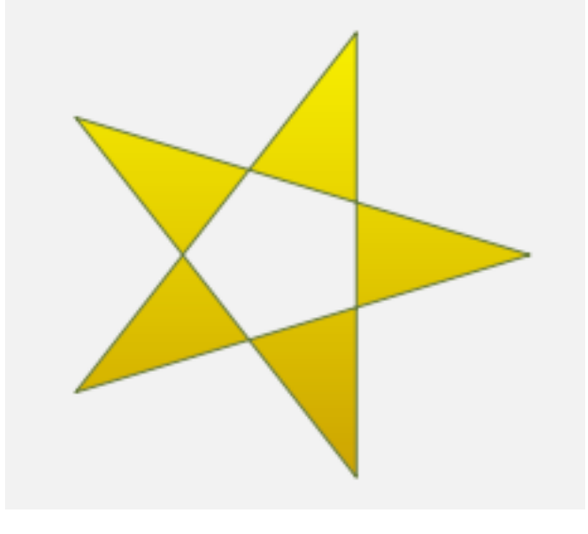
Qt.WindingFill

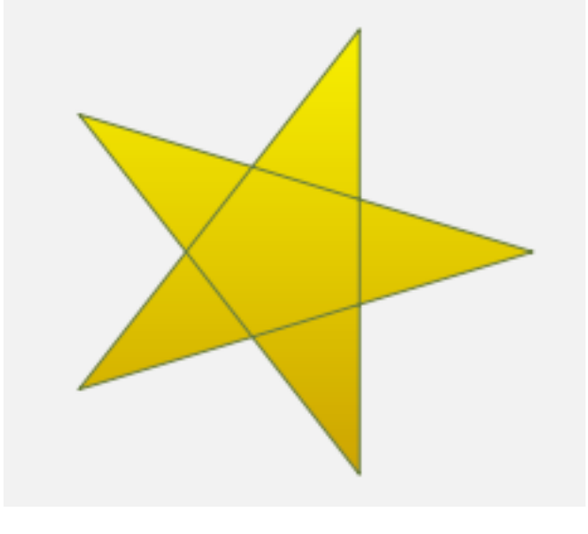
Paths

```
#event QPaintEvent
def paintEvent ( self , Events): 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)

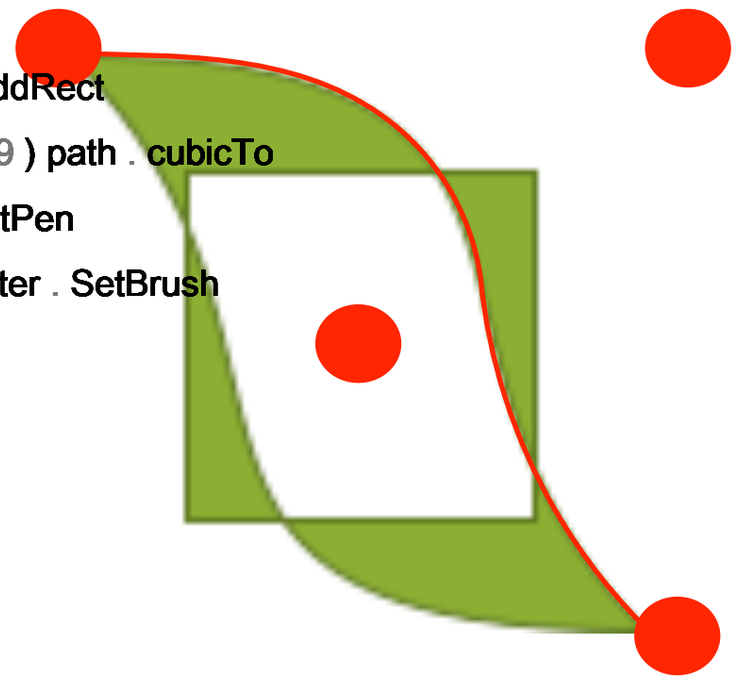




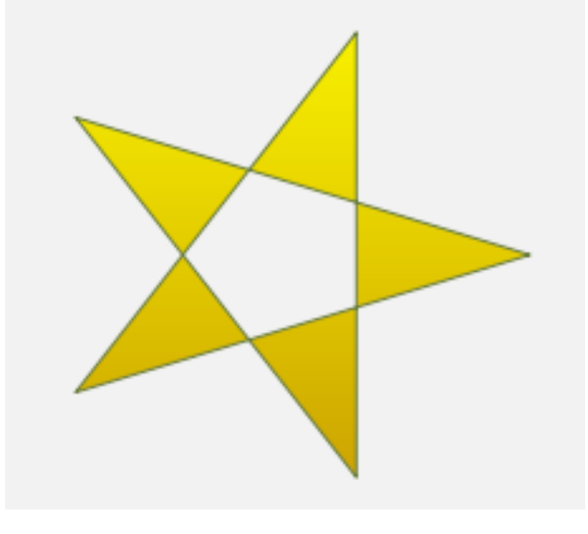
Qt.WindingFill

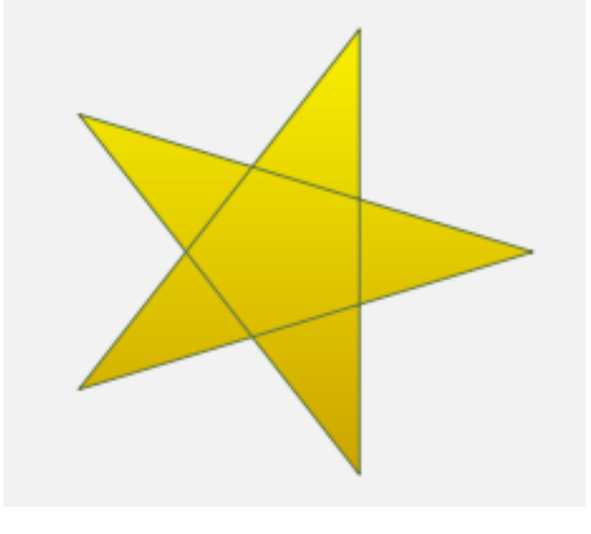
Paths

```
#event QPaintEvent
def paintEvent ( self , Events): 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);
```



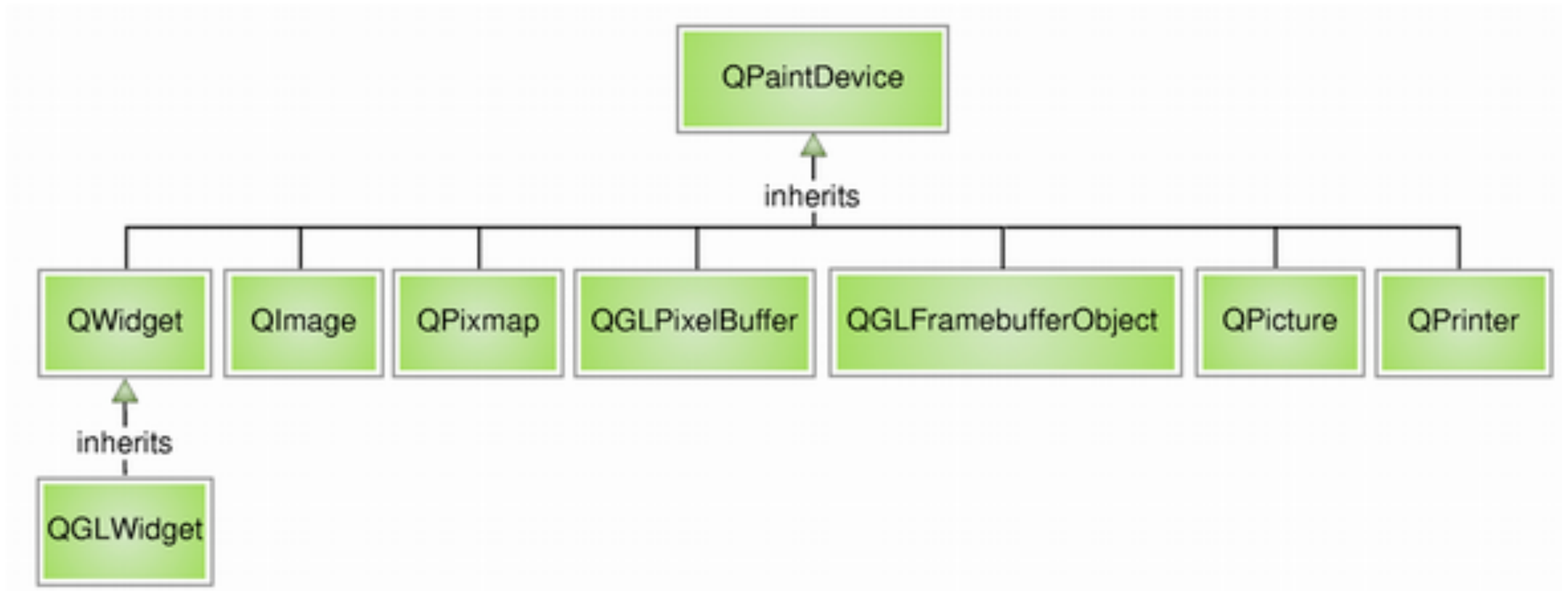
Qt.OddEvenFill
(default)





Qt.WindingFill

has surfaces ffi chage



imagery

Image types

- **QImage :** optimized for I / O access and / pixel manipulation
  - **QPixmap, QBitmap :** optimized for a ffi drying screen
- **QPicture :** to record and replay commands a **QPainter**
- in all cases: one can draw in with a **QPainter**

Entries exits

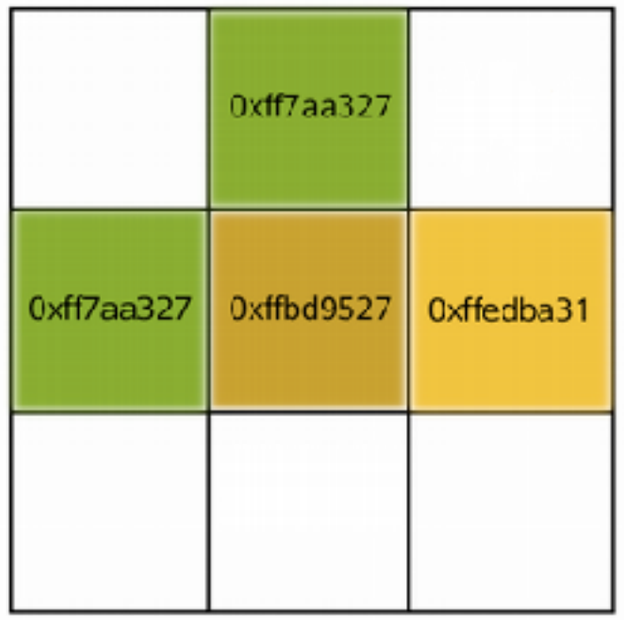
- **load () / save ():** / from a file, the main supported formats
- **loadFromData ():** from the memory

Access to pixels

32-bit format: direct access

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

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



8 bit format: indexed

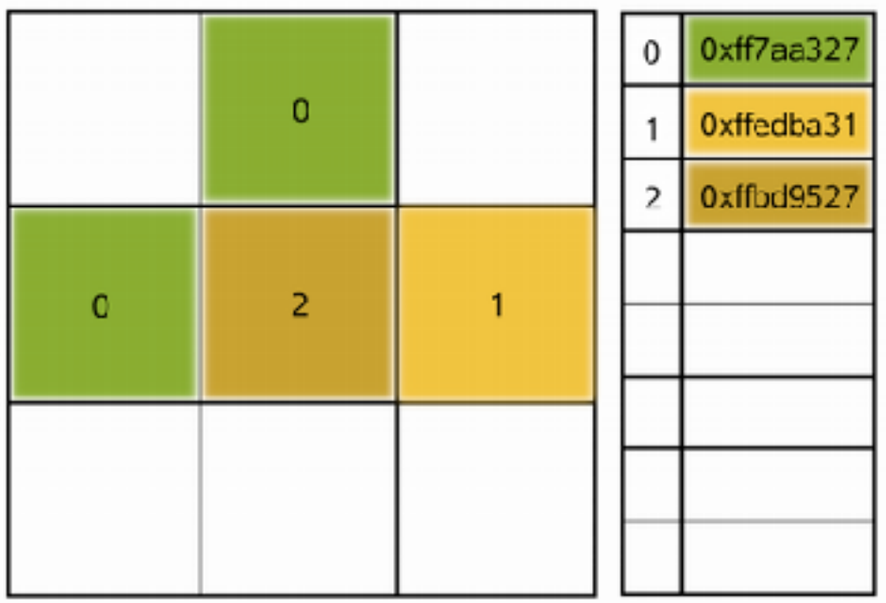
image = QImage (3, 3, **QImage :: Format_Indexed8)**

value = **QRgb (122, 163, 39) // *0xff7aa327***
picture. setColor (0, value)

QRgb value = (237, 187, 51) // 0xffedba31

image.setColor (1, value) image.setPixel (0, 1, 0)

image.setPixel (1, 0, 1)



Other surfaces a ffi chage

SVG ▸ QSvgWidget

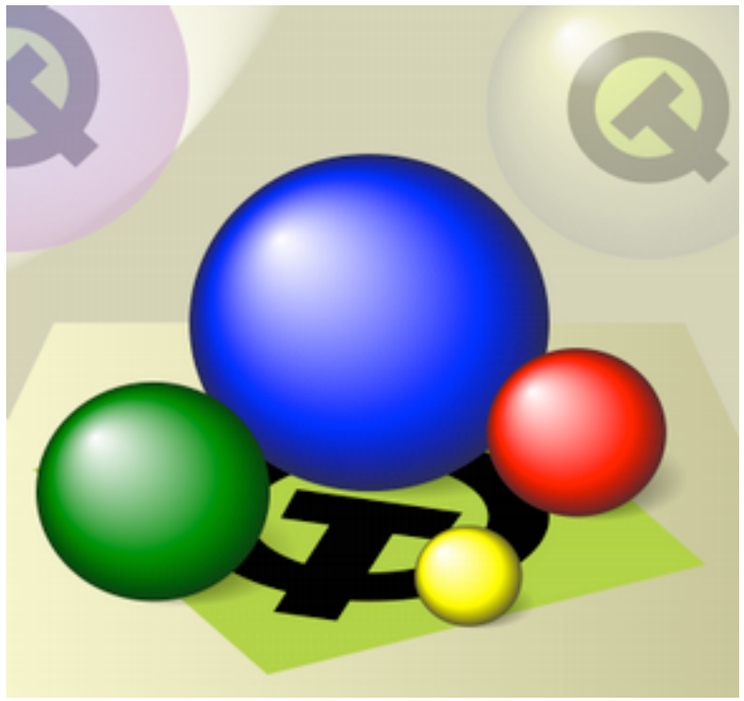
- QSvgRenderer

OpenGL

- ▸ QGLWidget
  - QGLPixelBu ff er
  - QGLFramebu ff erObject

Printing

- ▸ QPrinter



QSvgWidget

4 Interaction with geometric shapes

picking

Picking with QRect, QRectF

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

Picking with QPainterPath

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

Returns the intersection

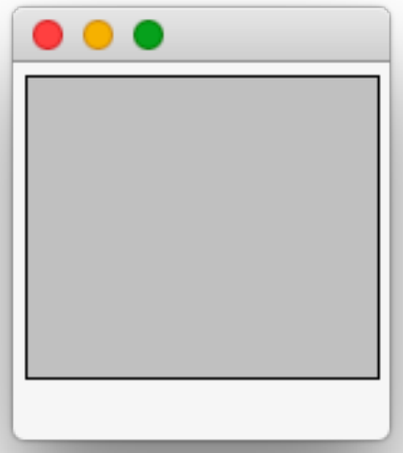
- QPainterPath **intersected** (QPainterPath const & path)

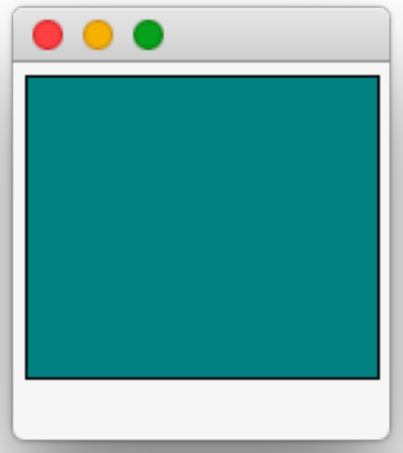
Picking / Interaction

Example

- test if the mouse is in the rectangle when you press the mouse button
- changes the color of the rectangle for each click

```
def __ init__ ( self ):
        Great () . __init __ ()
        self . myBool = true                                    #   boolean instance variable
        self . rect = QRect ( 5 , 5 , 140 , 120 )                   #   instance variable rectangle

def mousePressEvent ( self , Events):                           #   event mousePress
        self . PStart = event . pos ()
        yew self . rect . contains (event . pos ()):                #   test position
                self . myBool = not self . myBool
        self . update ()                                        #   asks the SHIFT drawing


#event QPaintEvent
def paintEvent ( self , Events): painter = QPainter ( self
        )
        yew self . myBool:
                painter . SetBrush (Qt . lightgray)
        else :
                painter . SetBrush (QColor (Qt . darkCyan)) painter . drawRect ( self
        . rect)
```

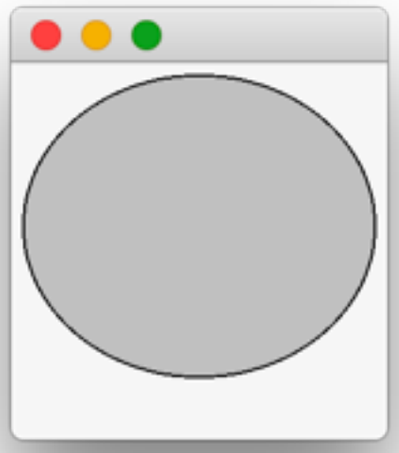


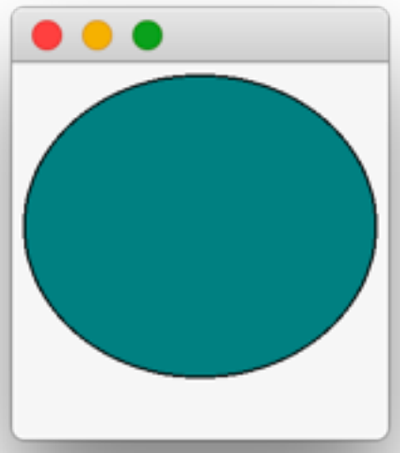


Non-rectangular shapes

using **regions**

```
def __ init__ ( self ):
        Great () . __init __ ()
        self . myBool = true                                    #   boolean instance variable
        self . rect = QRect ( 5 , 5 , 140 , 120 )                   #   instance variable rectangle

def mousePressEvent ( self , Events):                           #   event mousePress
        self . PStart = event . pos () = QRegion ellipse ( self . rect, QRegion . Ellipse) # defined elliptical area

        yew ellipse . contains (event . pos ()):                #   test position
                self . myBool = not self . myBool
        self . update ()                                        #   asks the SHIFT drawing


# event QPaintEvent
def paintEvent ( self , Events): painter = QPainter ( self
        )
        yew self . myBool:
                painter . SetBrush (Qt . lightgray)
        else :
                painter . SetBrush (QColor (Qt . darkCyan)) painter . drawEllipse ( self
        . rect)
```





5 Performance of a ffi chage

Performance of a ffi chage

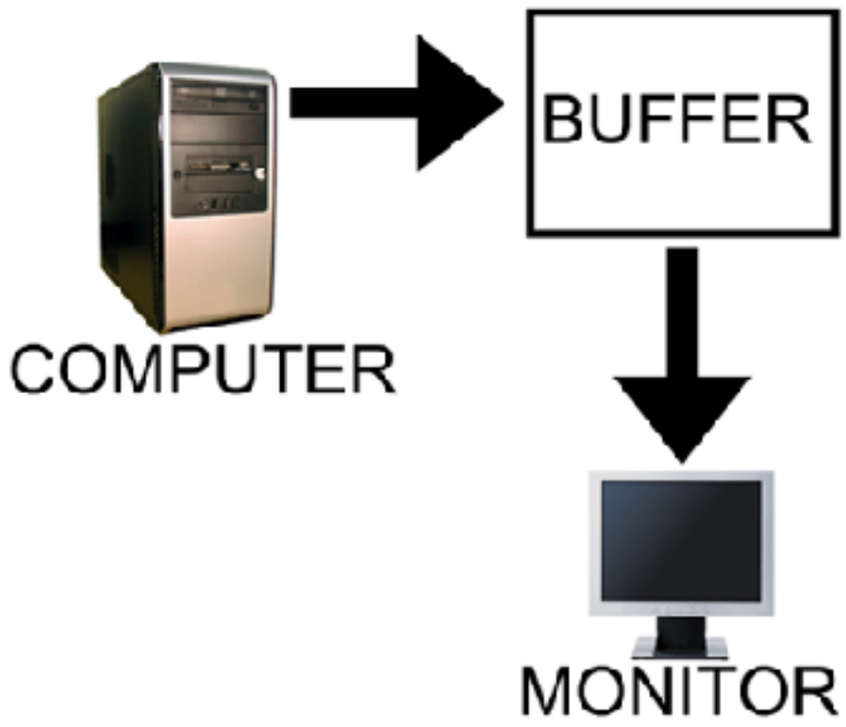
Common problems:

- **Flickering** and **tearing (** flicker and tear)
- **The G (** latency)

Flickering

Flickering

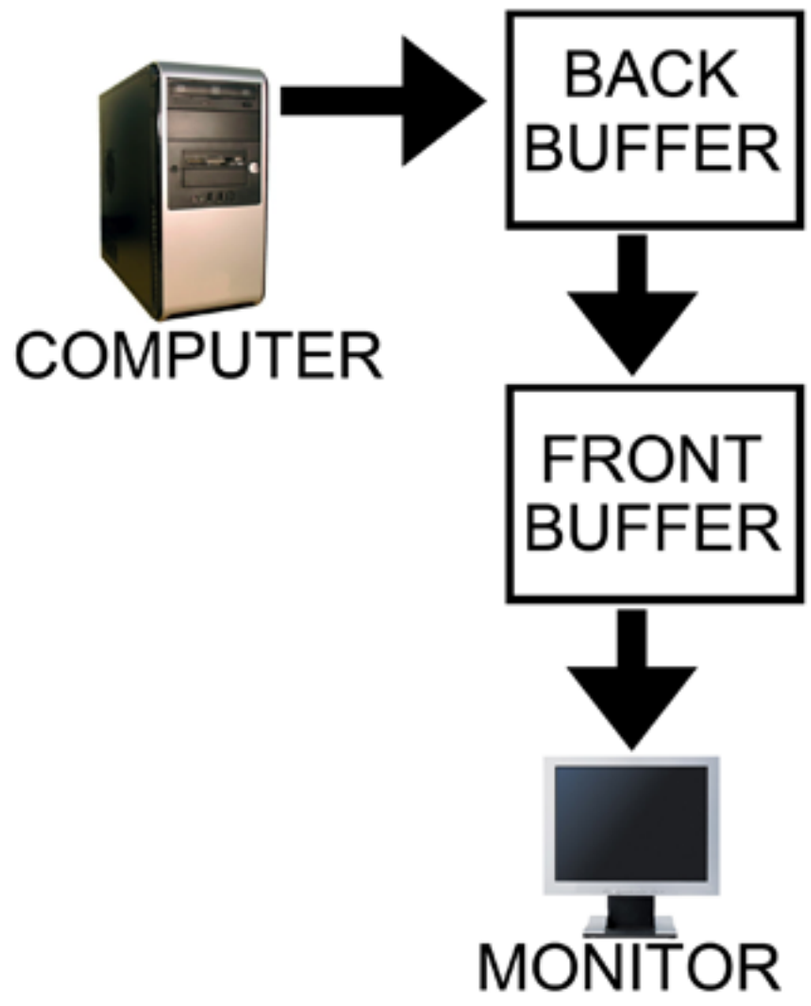
- flicker of a ffi drying because the eye perceives intermediate images
- example: to refresh this page must:
  - 1) any "e ff acer" (repaint the background)
  - 2) redraw all
  - => Flicker if transparent background is dark as the bottom of the window is white



Double bu ff ering

Double bu ff ering

- solution to fl ickering:
  - drawing in the back bu ff er
  - recopy the drunk ff er front (bu ff er video which controls what is a ffi ket on the screen)
- default with Qt4

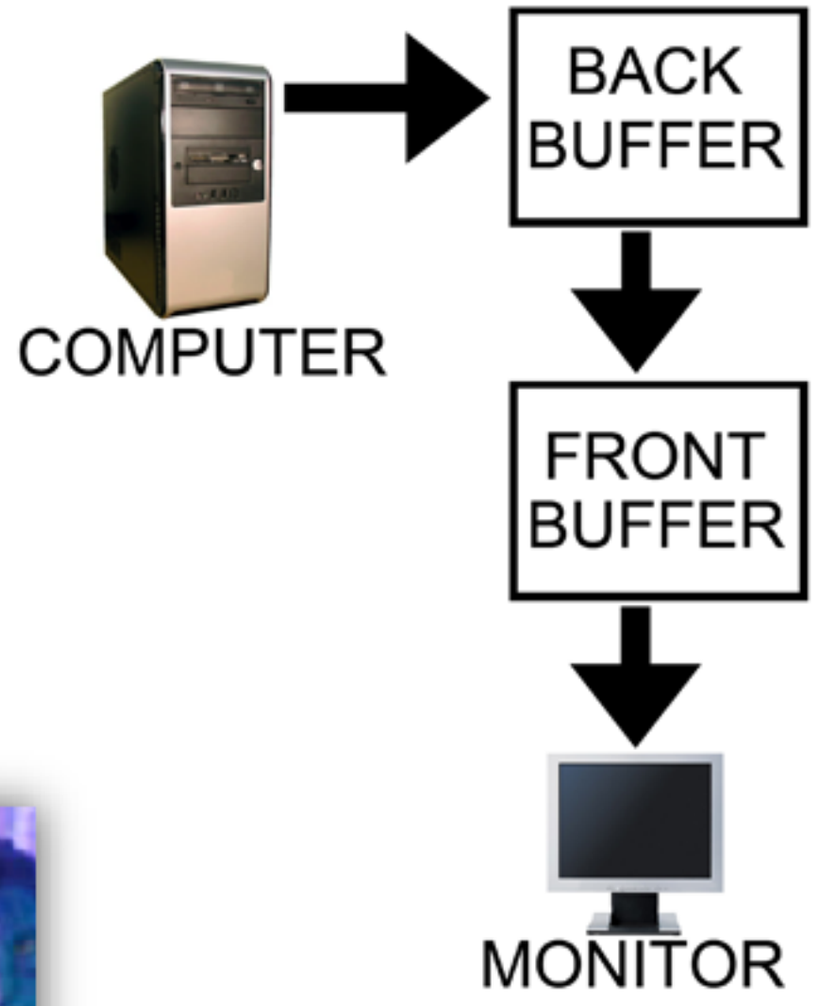


Source: AnandTech

Tearing

Possible problem: **Tearing**

- the image appears in 2 (or 3 ...) horizontal portions
- problem copying the back bu ff er
  before the design is complete
  - mixture of several "frames" video
  - especially with video and other games
    graphically demandantes applications



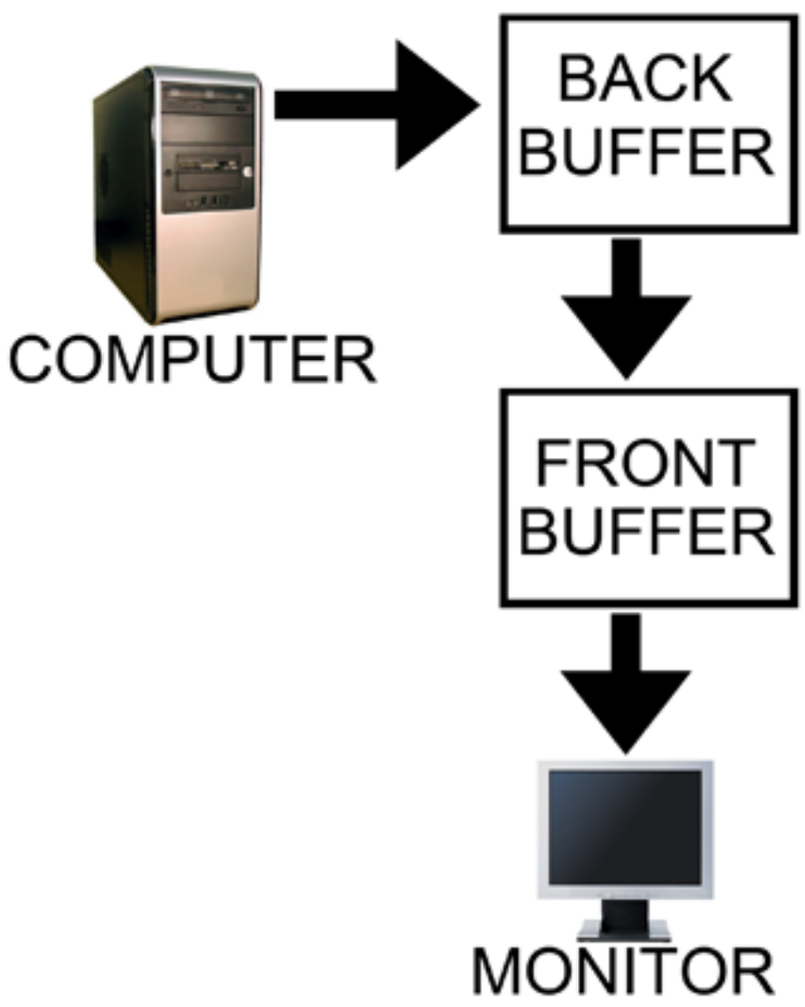
Example:

Source: AnandTech

Tearing

solution to **Tearing**

- **VSync (** Vertical synchronization)
- made synchronized with a ffi chage
- disadvantage slows a ffi chage



Source: AnandTech

Performance of a ffi chage

Latency (lag)

- and e ff: ff has chage "does not follow" the interaction
- reason: the rendering is not fast enough

Performance of a ffi chage

Latency (lag)

- and e ff: ff has chage *do not follow* interaction
- reason: the rendering is not fast enough

Solutions

- a ffi expensive fewer things:
  - in the space
  - in time
- a ffi expensive fashion XOR

Performance of a ffi chage

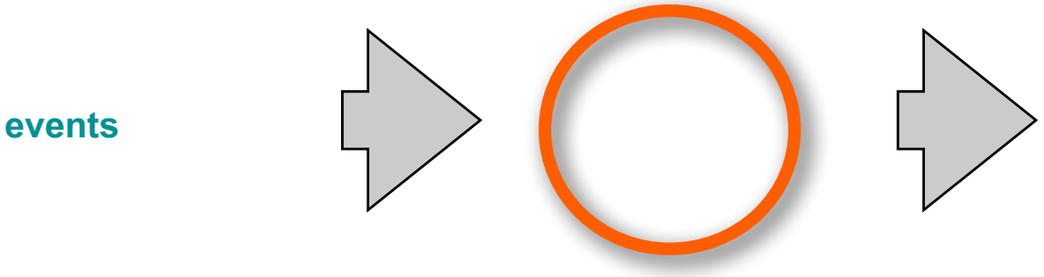
A ffi less expensive things in space

- **clipping:** reduce a ffi drying area
  - methods rect () and area () of QPaintEvent
- **" Backing store "** or equivalent
  - 1) copy which does not change in a **picture**
  - 2) a ffi expensive this image in the drawing area
  - 3) a ffi expensive part that changes over

Performance of a ffi chage

A ffi less expensive things in time

- skip the intermediate images:
  - ffi expensive sheave half the time ... or by time
- Timers can be useful (cf. **QTimer)**




```
void mouseMoveEvent ( QMouseEvent * e)
        ……
        if (delay <MAX_DELAY) update ();
}
```

Do not reaf fi expensive if the delay is too long Depending on the case (and the toolkit) test in one of these 2 functions

```
void paintEvent ( QPaintEvent * e) {
        if (delay> MAX_DELAY) return;
        ……
}
```