Qt Essentials - Painting Module

Training Course

Visit us at `http://qt.digia.com`

Produced by Digia Plc.

Material based on Qt 5.0, created on September 27, 2012

Digia Plc.

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Custom painting or stylesheets

- Painting
  - You paint with a painter on a paint device during a paint event
  - Qt widgets know how to paint themselves
  - Often widgets look like we want
  - Painting allows device independent 2D visualization
  - Allows to draw pie charts, line charts and many more
- StyleSheets
  - Fine grained control over the look and feel
  - Easily applied using style sheets in CSS format

Covers techniques for general 2D graphics and styling applications.

- **Painting**
  - Painting infrastructure
  - Painting on widget
- **Color Handling**
  - Define and use colors
  - Pens, Brushes, Palettes
- **Shapes**
  - Drawing shapes
- **Transformation**
  - 2D transformations of a coordinate system
- **Style Sheets**
  - How to make small customizations
  - How to apply a theme to a widget or application

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Low-level painting with QPainter

- Paints on paint devices (`QPaintDevice`)
- `QPaintDevice` implemented by
  - On-Screen: `QWidget`
  - Off-Screen: `QImage`, `QPixmap`
  - And others ...
- Provides drawing functions
  - Lines, shapes, text or pixmaps
- Controls
  - Rendering quality
  - Clipping
  - Composition modes

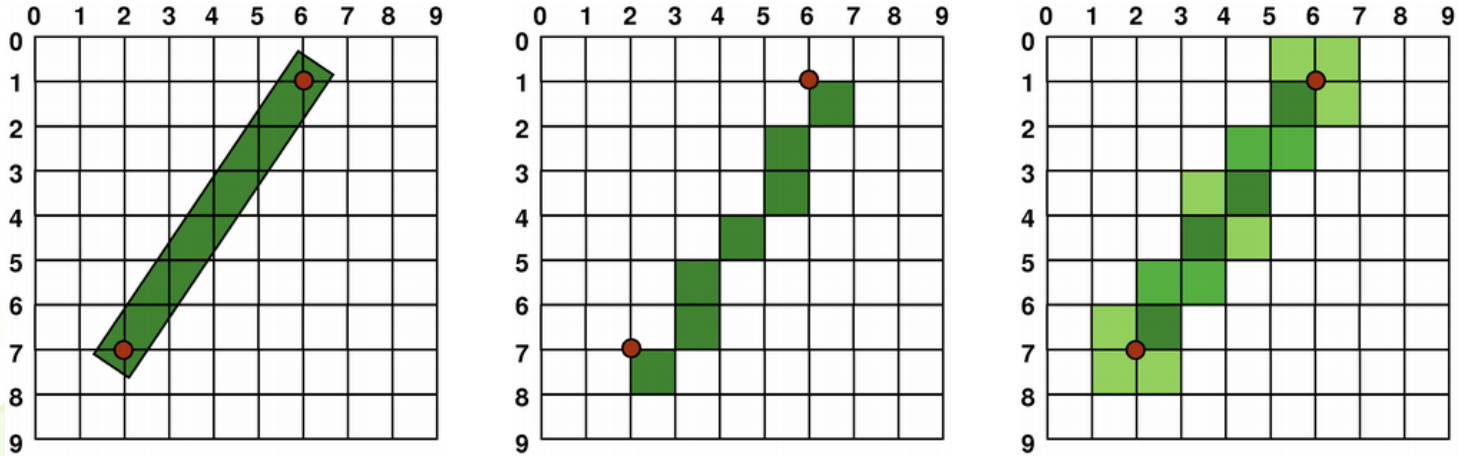
- Override `paintEvent(QPaintEvent*)`

```
void CustomWidget::paintEvent(QPaintEvent *) {
   QPainter painter(this);
   painter.drawRect(0,0,100,200); // x,y,w,h
}
```

- Schedule painting
  - `update()`: schedules paint event
  - `repaint()`: repaints directly
- Qt handles double-buffering
- To enable filling background:
  - `QWidget::setAutoFillBackground(true)`

Coordinate System - Surface to render

- Controlled by QPainter
- Origin: Top-Left
- Rendering
  - Logical - mathematical
  - Aliased - right and below
  - Anti-aliased - smoothing

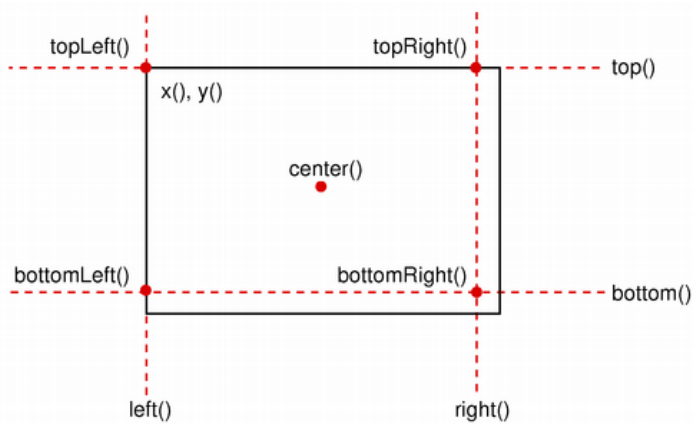


- Rendering quality switch
  - `QPainter::setRenderHint()`

Geometry Helper Classes

- `QSize(w,h)`
  - scale, transpose
- `QPoint(x,y)`
- `QLine(point1, point2)`
  - translate, dx, dy
- `QRect(point, size)`
  - adjust, move
  - translate, scale, center

```
QSize size(100,100);
QPoint point(0,0);
QRect rect(point, size);
rect.adjust(10,10,-10,-10);
QPoint center = rect.center();
```



- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Creating Color Values

- Using different color models:
  - `QColor(255,0,0)` // RGB
  - `QColor::fromHsv(h,s,v)` // HSV
  - `QColor::fromCmyk(c,m,y,k)` // CMYK
- Defining colors:

```
QColor(255,0,0); // red in RGB
QColor(255,0,0, 63); // red 25% opaque (75% transparent)
QColor("#FF0000"); // red in web-notation
QColor("red"); // by svg-name
Qt::red; // predefined Qt global colors
```

- Many powerful helpers for manipulating colors

```
QColor("black").lighter(150); // a shade of gray
```

- `QColor` always refers to device color space

See QColor Details Documentation

Drawing lines and outlines with QPen

- A pen (`QPen`) consists of:
  - **a color or brush**
  - **a width**
  - **a style** (e.g. `NoPen` or `SolidLine`)
  - **a cap style** (i.e. line endings)
  - **a join style** (connection of lines)
- Activate with `QPainter::setPen()`.

```
QPainter painter(this);
QPen pen = painter.pen();
pen.setBrush(Qt::red);
pen.setWidth(3);
painter.setPen(pen);
// draw a rectangle with 3 pixel width red outline
painter.drawRect(0,0,100,100);
```

Rule

The outline equals the size plus half the pen width on each side.

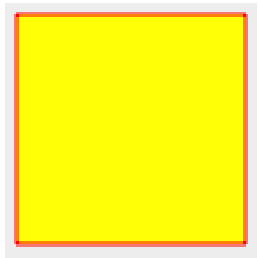
- For a pen of width 1:

```
QPen pen(Qt::red, 1); // width = 1
float hpw = pen.widthF()/2; // half-pen width
QRectF rect(x,y,width,height);
QRectF outline = rect.adjusted(-hpw, -hpw, hpw, hpw);
```

- *Due to integer rounding on a non-antialiased grid, the outline is shifted by 0.5 pixel towards the bottom right.*
- Demo painting/ex-rectoutline

Filling shapes with QBrush

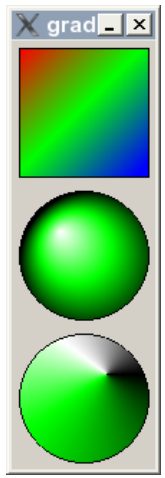
- `QBrush` defines fill pattern of shapes
- Brush configuration
  - `setColor(color)`
  - `setStyle(Qt::BrushStyle)`
    - NoBrush, SolidPattern, ...
  - `QBrush(gradient)` // QGradient's
  - `setTexture(pixmap)`
- Brush with solid red fill



```
painter.setPen(Qt::red);
painter.setBrush(QBrush(Qt::yellow, Qt::SolidPattern));
painter.drawRect(rect);
```

Drawing gradient fills

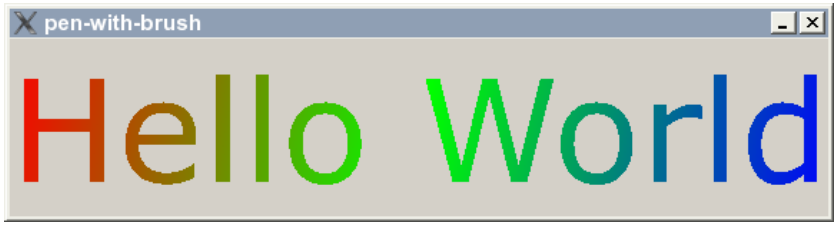
- Gradients used with `QBrush`
- Gradient types
  - `QLinearGradient`
  - `QConicalGradient`
  - `QRadialGradient`
- Gradient from P1(0,0) to P2(100,100)




```
QLinearGradient gradient(0, 0, 100, 100);
// position, color: position from 0..1
gradient.setColorAt(0, Qt::red);
gradient.setColorAt(0.5, Qt::green);
gradient.setColorAt(1, Qt::blue);
painter.setBrush(gradient);
// draws rectangle, filled with brush
painter.drawRect(0, 0, 100, 100 );
```

- Demo painting/ex-gradients

- Possible to set a brush on a pen
- Strokes generated will be filled with the brush



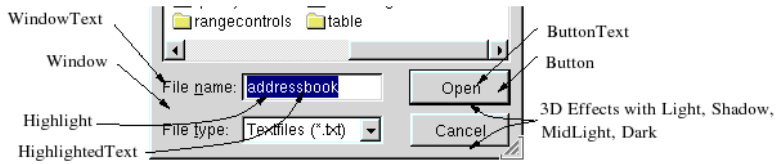
- Demo painting/ex-penwithbrush

- To support widgets color theming
  - `setColor(blue)` not recommended
  - Colors needs to be managed
- `QPalette` manages colors
  - Consist of color groups

- `enum QPalette::ColorGroup`
- Resemble widget states
  - `QPalette::Active`
    - Used for window with keyboard focus
  - `QPalette::Inactive`
    - Used for other windows
  - `QPalette::Disabled`
    - Used for disabled widgets

Color Groups and Roles

- Color group consists of color roles
- enum `QPalette::ColorRole`
- Defines symbolic color roles used in UI



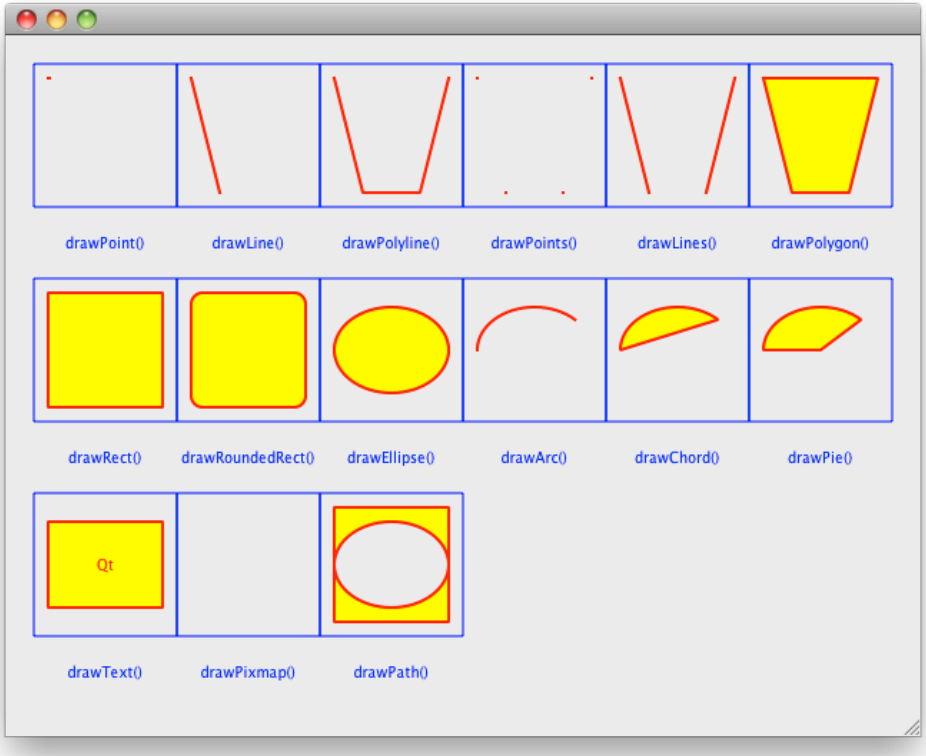

```
QPalette pal = widget->palette();
QColor color(Qt::red);
pal.setColor(QPalette::Active, QPalette::Window, color);
// for all groups
pal.setBrush(QPalette::Window, QBrush(Qt::red));
widget->setPalette(pal);
```

- `QApplication::setPalette()`
  - Sets application wide default palette

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

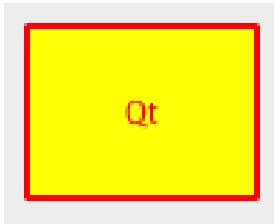
Drawing Figures

- Painter configuration
  - pen width: 2
  - pen color: red
  - font size: 10
  - brush color: yello
  - brush style: solid
- Demo painting/ex-figures



- `QPainter::drawText(rect, flags, text)`

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

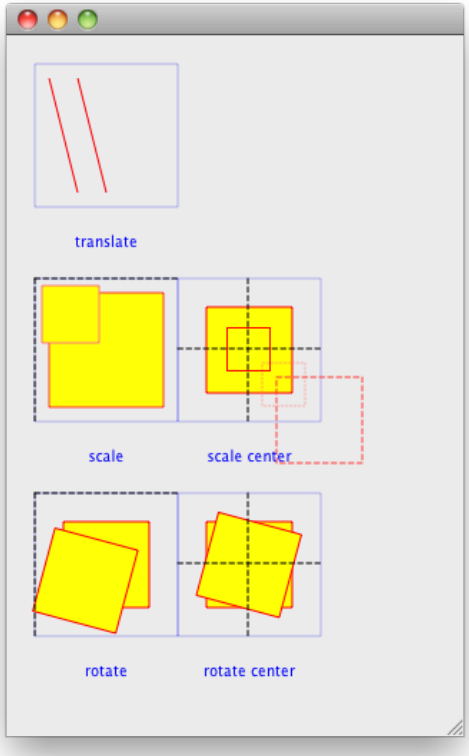


- `QFontMetrics`
  - calculate size of strings

```
QFont font("times", 24);
QFontMetrics fm(font);
int pixelsWide = fm.width("Width of this text?");
int pixelsHeight = fm.height();
```

- Manipulating the coordinate system
  - translate(x,y)
  - scale(sx,sy)
  - rotate(a)
  - shear(sh,sv)
  - reset()

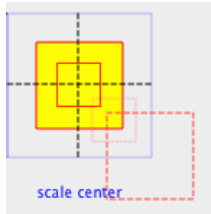
Demo painting/ex-transform



Transform and Center

- scale(sx, sy)
  - scales around QPoint(0,0)
- Same applies to all transform operations
- Scale around center?

```
painter.drawRect(r);
painter.translate(r.center());
painter.scale(sx,sy);
painter.translate(-r.center());
// draw center-scaled rect
painter.drawRect(r);
```



Demo painting/ex-transform (scale center)

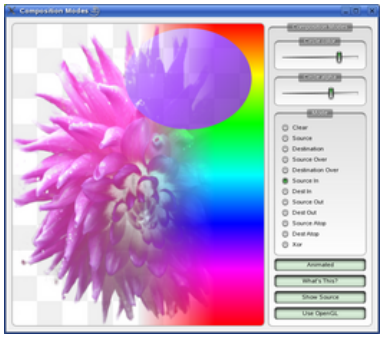
- Container for painting operations
- Enables reuse of shapes

```
QPainterPath path;
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.drawPath(path);
```

- Path information
  - `controlPointRect()` - rect containing all points
  - `contains()` - test if given shape is inside path
  - `intersects()` - test given shape intersects path

Demo $QTDIR/examples/painting/painterpaths

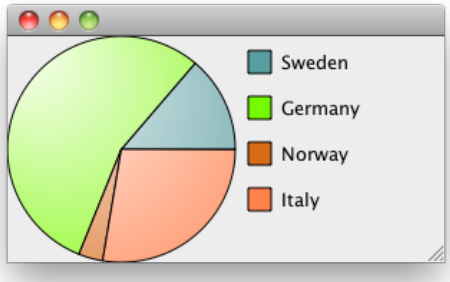
- Clipping
  - Clip drawing operation to shape
- Composition modes:
  - Rules for digital image compositing
  - Combining pixels from source to destination
- Rubber Bands - `QRubberBand`
  - Rectangle or line that indicate selection or boundary
  - See QRubberband Documentation



Lab: Pie Chart Widget

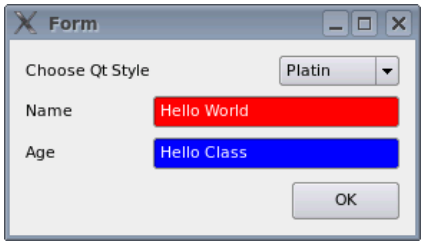
- Task to implement a pie chart
- Draw pies with painters based on data.
- Data Example: Population of 4 countries
  - Sweden
  - Germany
  - Norway
  - Italy
- Guess the population in millions of citizens ;-)
- **Legend is optional**
- See lab description for details

Lab painting/lab-piechart



- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

- Mechanism to customize appearance of widgets
  - Additional to subclassing `QStyle`
- Inspired by HTML CSS
- Textual specifications of styles
- Applying Style Sheets
  - `QApplication::setStyleSheet(sheet)`
    - On whole application
  - `QWidget::setStyleSheet(sheet)`
    - On a specific widget (incl. child widgets)



Demo painting/ex-simpleqss

CSS Rule

selector { property : value; property : value }

- Selector: specifies the widgets
- Property/value pairs: specify properties to change.

```
QPushButton {color:red; background-color:white}
```

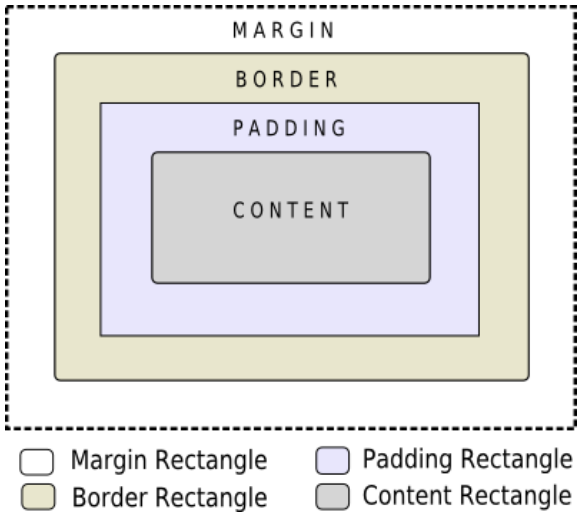
- Examples of stylable elements
  - Colors, fonts, pen style, alignment.
  - Background images.
  - Position and size of sub controls.
  - Border and padding of the widget itself.
- Reference of stylable elements

See Qt Style Sheets Reference Documentation

The Box Model

- Every widget treated as box
- Four concentric rectangles
  - Margin, Border, Padding, Content
- Customizing QPushButton

```
QPushButton {
  border-width: 2px;
  border-radius: 10px;
  padding: 6px;
  // ...
 }
```



- By default, `margin`, `border-width`, and `padding` are 0

- `*{ }` // Universal selector
  - All widgets
- `QPushButton { }` // Type Selector
  - All instances of class
- `.QPushButton { }` // Class Selector
  - All instances of class, but not subclasses
- `QPushButton#objectName` // ID Selector
  - All Instances of class with objectName
- `QDialog QPushButton { }` // Descendant Selector
  - All instances of QPushButton which are child of QDialog
- `QDialog > QPushButton { }` // Direct Child Selector
  - All instances of QPushButton which are direct child of QDialog
- QPushButton[enabled="true"] // Property Selector
  - All instances of class which match property

- Property Selector
  - If property changes it is required to re-set style sheet
- Combining Selectors
  - `QLineEdit, QComboBox, QPushButton { color: red }`
- Pseudo-States
  - Restrict selector based on widget's state
  - Example: `QPushButton:hover {color:red}`
- Demo painting/ex-qssselector
- Selecting Subcontrols
  - Access subcontrols of complex widgets
    - as QComboBox, QSpinBox, ...
  - `QComboBox::drop-down { image: url(dropdown.png) }`
- Subcontrols positioned relative to other elements
  - Change using `subcontrol-origin` and `subcontrol-position`

Conflict Resolution - Cascading

- Effective style sheet obtained by merging
  1. Widgets's ancestor (parent, grandparent, etc.)
  2. Application stylesheet
- On conflict: widget own style sheet preferred

```
qApp->setStyleSheet("QPushButton { color: white }");
button->setStyleSheet("* { color: blue }");
```

- Style on button forces button to have blue text
  - In spite of more specific application rule

Demo painting/ex-qsscascading

Conflict Resolution - Selector Specifity

- Conflict: When rules on same level specify same property
  - Specificity of selectors apply

```
QPushButton:hover { color: white }
QPushButton { color: red }
```

  - Selectors with pseudo-states are more specific
- Calculating selector's specificity
  - **a** Count number of ID attributes in selector
  - **b** Count number of property specifications
  - **c** Count number of class names
  - Concatenate numbers **a**-**b**-**c**. Highest score wins.
  - If rules scores equal, use last declared rule

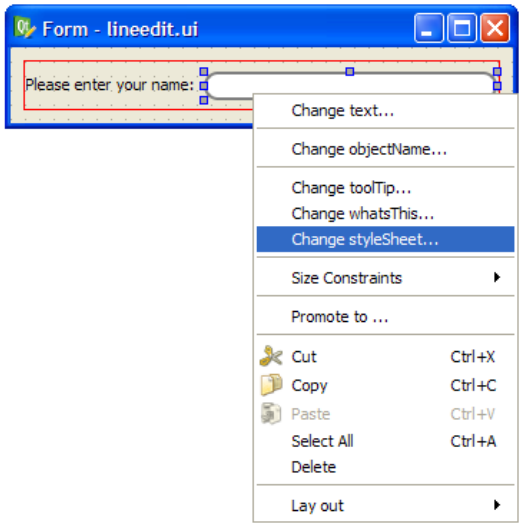
```
QPushButton {}      /* a=0 b=0 c=1 -> specificity =   1 */
QPushButton#ok {}   /* a=1 b=0 c=1 -> specificity =   101 */
```

Demo painting/ex-qssconflict

Qt Designer Integration

- Excellent tool to preview style sheets
- Right-click on any widget
  - Select *Change styleSheet..*
- Includes syntax highlighter and validator

Demo Editing Style Sheets in Designer



- Tasks
  - Investigate style sheet
  - Modify style sheet
  - Remove style sheet
    and implement your own
- Example does not save changes.
  Use designer for this.
- Edit style sheet using
  `File -> Edit StyleSheet`

Lab $QTDIR/examples/widgets/stylesheet