



Qt 绘图





课程主要内容



- ◆ Qt绘制事件
- ◆ Qt 2D绘图
- ◆ 画笔
- ◆ 画刷
- ◆ 基本图形和文本绘制
- ◆ 渐变填充
- ◆ 绘制文本
- ◆ 图像处理
- ◆ 坐标系统与坐标变换
- ◆ 绘图举例：表盘



Qt绘制事件



事件处理和绘制（Painting）



- ◆ 当应用程序收到绘制事件时，就会调用 **QWidget::paintEvent()**，该函数就是绘制控件的地方
- ◆ 有两种方法要求重绘一个控件
 - ⊕ **update()** – 把重绘事件添加到事件队列中
 - ◆ 重复调用**update()** 会被Qt合并为一次
 - ◆ 不会产生图像的闪烁
 - ◆ 可带参数指定重绘某个区域
 - ⊕ **repaint()** – 立即产生绘制事件
 - ◆ 一般情况下不推荐使用此方法
 - ◆ 只使用在需要立即重绘的特效情况下
 - ◆ 可带参数指定重绘某个区域



事件处理和绘制 (Painting)



- ◆ 为处理绘制事件，只需要重写**paintEvent**函数，并在该函数中实例化一个**QPainter**对象进行绘制

```
class MyWidget : public QWidget
{
    ...

protected:
    void paintEvent(QPaintEvent*);
```

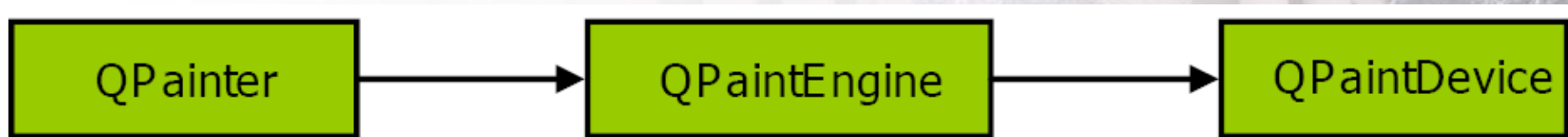
```
void MyWidget::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);
    ...
}
```



基本绘制Pipeline



- ◆ **QPainter**类提供绘制操作
- ◆ **QPaintEngine**类提供平台相关的API
- ◆ **QPaintDevice**代表绘制2D图像的画布
- ◆ 如下继承**QPaintDevice**的类对象都可用于**QPainter**绘制
 - ⊕ **QWidget, QImage, QPixmap, QPicture, QPainter, QSvgGenerator, QGLPixelBuffer, QGLFrameBufferObject, ...**





Qt 2D绘图



QT 2D绘图



- ◆ **Qt4中的2D绘图部分称为Arthur绘图系统。它由3个类支撑整个框架，QPainter，QPaintDevice和QPaintEngine**
 - ⊕ **QPainter**用来执行具体的绘图相关操作如画点，画线，填充，变换，alpha通道等。
 - ⊕ **QPaintDevice**是**QPainter**用来绘图的绘图设备，Qt中有几种预定义的绘图设备，如**QWidget**，**QPixmap**，**QImage**等。他们都从**QPaintDevice**继承。
 - ⊕ **QPaintEngine**提供了**QPainter**在不同设备上绘制的统一接口，通常对开发人员是透明的。使用**QPainter**在**QPaintDevice**上进行绘制，它们之间使用**QPaintEngine**进行通讯。
- ◆ **从Qt4.2开始，Graphics View框架取代了QCanvas，QGraphics View框架使用了MVC模式，适合对大量2D图元的管理，Graphics View框架中，场景(scene)存储了图形数据，它通过视图(view)以多种表现形式，每个图元(item)可以单独进行控制。**



QPainter



- ◆ 线和轮廓都可以用画笔(**QPen**)进行绘制, 用画刷(**QBrush**)进行填充。
- ◆ 字体使用**QFont**类定义, 当绘制文字时, **Qt**使用指定字体的属性, 如果没有匹配的字体, **Qt**将使用最接近的字体
- ◆ 通常情况下, **QPainter**以默认的坐标系统进行绘制, 也可以用**QMatrix**类对坐标进行变换



QPainter



- ◆ 当绘制时，可以使用**QPainter::RenderHint**来告诉绘图引擎是否启用反锯齿功能使图变得平滑
- ◆ **QPainter::RenderHint**的可取值
 - ⊕ **QPainter::Antialiasing**: 告诉绘图引擎应该在可能的情况下进行边的反锯齿绘制
 - ⊕ **QPainter::TextAntialiasing**: 尽可能的情况下文字的反锯齿绘制
 - ⊕ **QPainter::SmoothPixmapTransform**: 使用平滑的pixmap变换算法(双线性插值算法)，而不是近邻插值算法



QPainter的绘图函数



◆ drawArc()	弧	◆ drawPixmap()	QPixmap 表示的图像
◆ drawChord()	弦	◆ drawPoint()	点
◆ drawConvexPolygon()	凸多边形	◆ drawPoints()	多个点
◆ drawEllipse()	椭圆	◆ drawPolygon()	多边形
◆ drawImage() 示的图像	QImage表	◆ drawPolyline()	多折线
◆ drawLine()	线	◆ drawRect()	矩形
◆ drawLines()	多条线	◆ drawRects()	多个矩形
◆ drawPath()	路径	◆ drawRoundRect()	圆角矩形
◆ drawPicture() QPainter指令绘制	按	◆ drawText()	文字
◆ drawPie()	扇形	◆ drawTiledPixmap()	平铺图像
		◆ drawLineSegments()	绘制折线



画笔



画笔



- ◆ 画笔的属性包括线型、线宽、颜色等。画笔属性可以在构造函数中指定，也可以使用 `setStyle()`, `setWidth()`, `setBrush()`, `setCapStyle()`, `setJoinStyle()` 等函数设定
- ◆ Qt 中，使用 `Qt::PenStyle` 定义了 6 种画笔风格，分别是
 - ⊕ `Qt::SolidLine`, `Qt::DashLine`, `Qt::DotLine`, `Qt::DashDotLine`, `Qt::DashDotDotLine`, `Qt::CustomDashLine`。
 - ⊕ 自定义线风格(`Qt::CustomDashLine`)，需要使用 `QPen` 的 `setDashPattern()` 函数来设定自定义风格。



线型



◆ Qt::SolidLine

◆ Qt::DashLine

◆ Qt::DotLine

◆ Qt::DashDotLine

◆ Qt::DashDotDotLine

◆ Qt::CustomDashLine – 由dashPattern控制





◆ 端点风格(cap style)

- ⊕ 端点风格决定了线的端点样式，只对线宽大于1的线有效。
- ⊕ Qt种定义了三种端点风格用枚举类型Qt::PenCapStyle表示，分别为Qt::SquireCap, QT::FlatCap, Qt::RoundCap。

◆ 连接风格(Join style)

- ⊕ 连接风格是两条线如何连接，连接风格对线宽大于等于1的线有效。
- ⊕ Qt定义了四种连接方式，用枚举类型Qt::PenStyle表示。分别是Qt::MiterJoin, Qt::BevelJoin, Qt::RoundJoin, Qt::SvgMiterJoin。

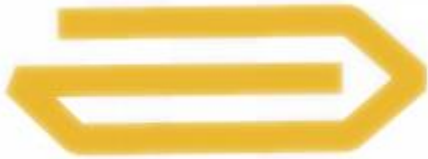


端点风格和连接风格

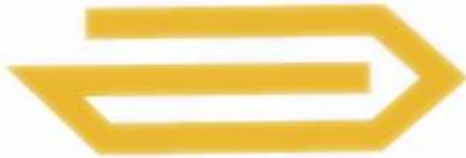


◆ 连接风格

⊕ Qt::BevelJoin
(default)



⊕ Qt::MiterJoin



⊕ Qt::RoundJoin



◆ 端点风格

⊕ Qt::SquareCap (default):
矩形封线尾



⊕ Qt::FlatCap: 不封线尾



⊕ Qt::RoundCap

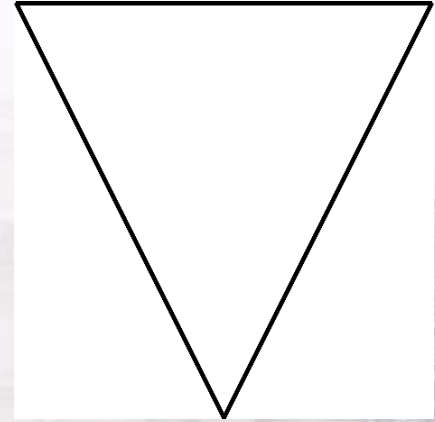




画笔示例



```
QPainter p(this);  
QPen pen(Qt::black, 5);  
p.setPen(pen);  
p.drawPolygon(polygon);
```





画刷



- ◆ 在Qt中图形使用**QBrush**进行填充，画刷包括填充**颜色**和**风格(填充模式)**。
- ◆ 在Qt中，颜色使用**QColor**类表示，**QColor**支持**RGB**，**HSV**，**CMYK**颜色模型。**QColor**还支持**alpha**混合的轮廓和填充。
 - ⊕ **RGB**是面向硬件的模型。颜色由红绿蓝三种基色混合而成。
 - ⊕ **HSV/HSL**模型比较符合人对颜色的感觉，由色调(**0-359**)，饱和度(**0-255**)，亮度(**0-255**)组成，主要用于颜色选择器。
 - ⊕ **CMYK**由青，洋红，黄，黑四种基色组成。主要用于打印机等硬件拷贝设备上。每个颜色分量的取值是**0-255**。
 - ⊕ 另外**QColor**还可以用**SVG1.0**中定义的任何颜色名为参数初始化。
- ◆ 基本模式填充包括有各种点、线组合的模式。



◆ QColor的构造函数

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

◆ **r (red) , g (green) , b (blue) , a (alpha)** 的取值范围为**0-255**

◆ **Alpha控制透明度**

⊕ **255**: 不透明

⊕ **0**: 完全透明

◆ **Qt预定义颜色**

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



颜色微调



◆ 颜色可以通过如下函数进行微调

⊕ **QColor::lighter(int factor)**

⊕ **QColor::darker(int factor)**



darker



Qt::red



lighter



QRgb



- ◆ **QRgb**类可以用于保存颜色值，可与**QColor**相互转换获取
 - ⊕ 32-bit的RGB颜色值+alpha值
- ◆ 创建新颜色

```
QRgb orange = qRgb(255, 127, 0);  
QRgb overlay = qRgba(255, 0, 0, 100);
```

- ◆ 获取单独某个颜色值：**qRed**, **qGreen**, **qBlue**, **qAlpha**

```
int red = qRed(orange);
```

- ◆ 获取灰度值

```
int gray = qGray(orange);
```



实色画刷

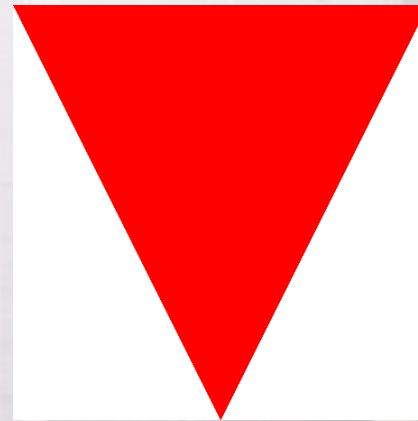


◆ 调用画刷构造函数

```
QBrush red(Qt::red);
```

```
QBrush odd(QColor(55, 128, 97));
```

```
QPainter p(this);  
p.setPen(Qt::NoPen);  
p.setBrush(Qt::red);  
p.drawPolygon(polygon);
```



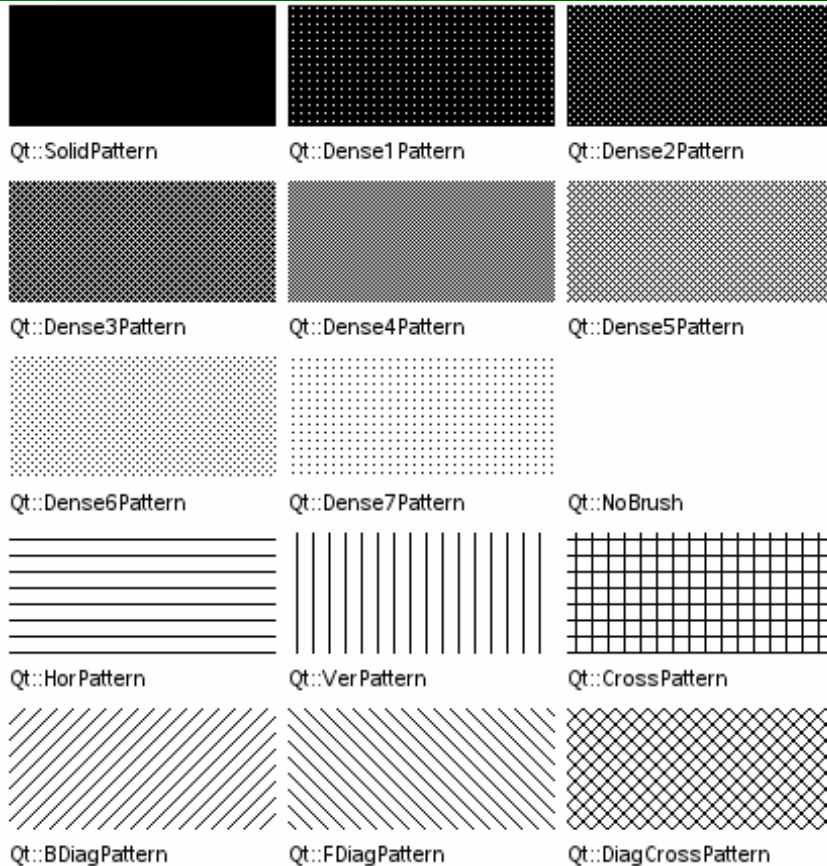


模式画刷



◆ 模式化刷构造函数

QBrush(const QColor &color, Qt::BrushStyle style)





带纹理的画刷

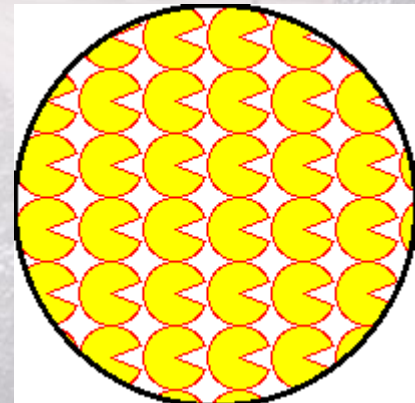


◆ 以QPixmap为参数的构造函数

- ⊕ 如果使用黑白的pixmap，则用画刷颜色
- ⊕ 如果使用彩色pixmap，则用pixmap的颜色

```
QBrush( const QPixmap &pixmap )
```

```
QPixmap pacPixmap("pacman.png");  
  
painter.setPen(QPen(Qt::black, 3));  
painter.setBrush(pacPixmap);  
painter.drawEllipse(rect());
```





基本图形和文本绘制



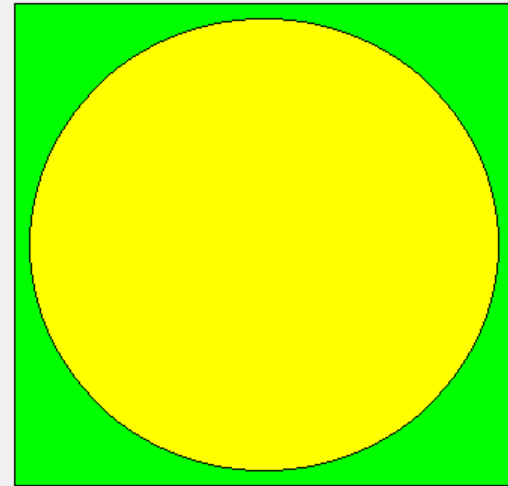
基本图形绘制



◆ 实现paintEvent函数

```
void RectWithCircle::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);

    p.setBrush(Qt::green);
    p.drawRect(10, 10, width()-20, height()-20);
    p.setBrush(Qt::yellow);
    p.drawEllipse(20, 20, width()-40, height()-40);
}
```





基本文本绘制



◆ QPainter::drawText

```
QPainter p(this);
```

```
QFont font("Helvetica");  
p.setFont(font);  
p.drawText(20, 20, 120, 20, 0, "Hello World!");
```

```
font.setPixelSize(10);  
p.setFont(font);  
p.drawText(20, 40, 120, 20, 0, "Hello World!");
```

```
font.setPixelSize(20);  
p.setFont(font);  
p.drawText(20, 60, 120, 20, 0, "Hello World!");
```

```
QRect r;  
p.setPen(Qt::red);  
p.drawText(20, 80, 120, 20, 0, "Hello World!", &r);
```

Hello World!

Hello World!

Hello World!

Hello World!

r返回文本
外边框的矩形区域



渐变填充



渐变填充



- ◆ Qt4提供了**渐变填充**的画刷，渐变填充包括两个要素：
颜色的变化和路径的变化。
 - ⊕ 颜色变化可以指定从一种颜色渐变到另外一种颜色。
 - ⊕ 路径变化指在路径上指定一些点的颜色进行分段渐变。
- ◆ Qt4中，提供了三种渐变填充
 - ⊕ **线性(QLinearGradient)**
 - ⊕ **圆形(QRadialGradient)**
 - ⊕ **圆锥渐变(QConicalGradient)**
 - ⊕ 所有的类都从**QGradient**类继承
- ◆ 构造渐变填充的画刷

```
QBrush b = QBrush( QRadialGradient( ... ) );
```





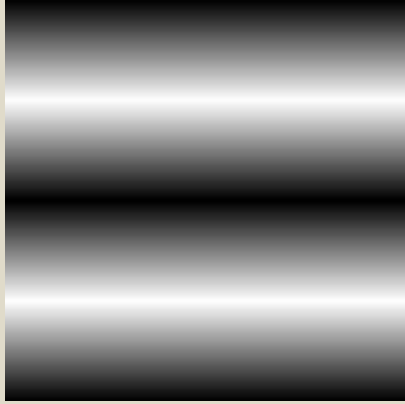
填充设置



- ◆ 从图形的起点到终点，以从0至1的比例渐变填充

```
QGradient::setColorAt( qreal pos, QColor );
```

- ◆ 完成0-1范围的填充后，后续颜色铺开的方式可以不同，通过 **setSpread()** 函数来设置

QGradient::PadSpread (default)	QGradient::RepeatSpread	QGradient::ReflectSpread
		



线性渐变填充



- ◆ 线性渐变填充指定两个控制点，画刷在两个控制点之间进行颜色插值。
- ◆ 通过创建**QLinearGradient**对象来设置画刷。

```
QPainter p(this);
```

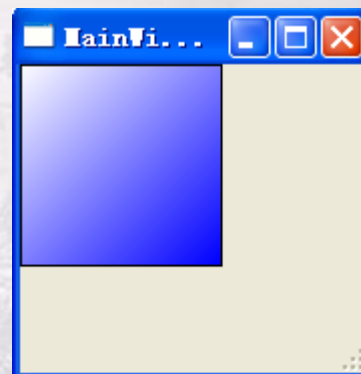
```
QLinearGradient g(0, 0, 100, 100);
```

```
g.setColorAt(0.0, Qt::white);
```

```
g.setColorAt(1.0, Qt::blue);
```

```
p.setBrush(g);
```

```
p.drawRect(0, 0, 100, 100);
```



- ◆ 在**QGradient**构造函数中指定线性填充的两点分别为(0,0)，(100,100)。**setColorAt()**函数在0-1之间设置指定位置的颜色。



线型填充示例



`setColorAt(0.0,
QColor(0, 0, 0))`

`setColorAt(0.7,
QColor(255, 0, 0))`

`setColorAt(1.0,
QColor(255, 255, 0))`



圆形渐变填充



- ◆ 圆形渐变填充需要指定圆心，半径和焦点，**QRadialGradient (qreal cx, qreal cy, qreal radius, qreal fx, qreal fy)**。画刷在焦点和圆上的所有点之间进行颜色插值。创建**QRadialGradient**对象设置画刷

QPainter painter(this);

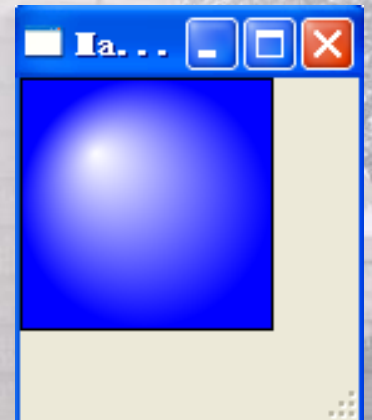
QRadialGradient radialGradient(50, 50, 50, 30, 30);

radialGradient.setColorAt(0.0, Qt::white);

radialGradient.setColorAt(1.0, Qt::blue);

painter.setBrush(radialGradient);

painter.drawRect(0, 0, 100, 100);



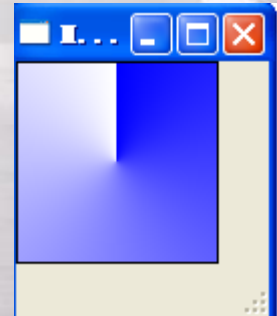


圆锥渐变填充



- ◆ 圆锥渐变填充指定圆心和开始角，**QConicalGradient** (**qreal cx, qreal cy, qreal angle**)。画刷沿圆心逆时针对颜色进行插值，创建**QConicalGradient**对象并设置画刷。

```
QPainter painter(this);  
QConicalGradient conicalGradient(50, 50, 90);  
conicalGradient.setColorAt(0, Qt::white);  
conicalGradient.setColorAt(1, Qt::blue);  
painter.setBrush(conicalGradient);  
painter.drawRect(0, 0, 100, 100);
```



- ◆ 为了实现自定义填充，还可以使用**QPixmap**或者**QImage**对象进行纹理填充。两种图像分别使用**setTexture()**和**setTextureImage()**函数加载纹理。



绘制文本



文本绘制



◆ 使用QPainter进行文本绘制

⊕ 基本文本绘制

drawText(QPoint, QString)

⊕ 带选项的文本绘制

drawText(QRect, QString, QTextOptions)

⊕ 带返回信息的文本绘制

drawText(QRect, flags, QString, QRect*)



使用字体



- ◆ Qt提供了**QFont**类来表示字体，当创建**QFont**对象时，Qt会使用指定的字体，如果没有对应的字体，Qt将寻找一种最接近的已安装字体
 - ⊕ **Font family**
 - ⊕ **Size**
 - ⊕ **Bold / Italic / Underline / Strikeout / ...**
- ◆ 字体信息可以通过**QFontInfo**取出，并可用**QFontMetrics**取得字体的相关数据。
- ◆ 使用**QApplication::setFont()**可以设置应用程序默认的字体
- ◆ 当**QPainter**绘制指定的字体中不存在的字符时将绘制一个空心的正方形。



Font Family



◆ 在构造函数中指定字体

```
QFont font("Helvetica");  
font.setFamily("Times");
```

◆ 得到可用字体列表

```
QFontDatabase database;  
QStringList families = database.families();
```



Font Size



- ◆ 字体尺寸可以用像素尺寸（**pixel size**）或点阵尺寸（**point size**）

```
QFont font("Helvetica");
```

```
font.setPointSize(14); // 14 points high  
// depending on the paint device's dpi
```

```
font.setPixelSize(10); // 10 pixels high
```




字体效果



◆ 可以激活字体效果

Hello Qt!

Hello Qt!

Hello Qt!

~~Hello Qt!~~

Hello Qt!

Hello Qt!

Normal, bold,
italic, strike out,
underline,
overline

◆ QWidget::font函数和QPainter::font函数返回 现有字体的const引用，因而需要先拷贝现有font，再做修改

```
QFont tempFont = w->font();  
tempFont.setBold( true );  
w->setFont( tempFont );
```



测量文本大小



- ◆ **QFontMetrics** 可用于测量文本和font的大小
- ◆ **boundingRect** 函数可用于测量文本块的大小

```
QImage image(200, 200, QImage::Format_ARGB32);  
QPainter painter(&image);  
QFontMetrics fm(painter.font(), &image);  
  
qDebug("width: %d", fm.width("Hello Qt!"));  
qDebug("height: %d", fm.boundingRect(0, 0, 200, 0,  
    Qt::AlignLeft | Qt::TextWordWrap, loremIpsum).height());
```



中文显示问题



◆ 使用QTextCodec类

⊕ In main.cpp

```
#include <QTextCodec>
```

```
...
```

```
QTextCodec *codec = QTextCodec::codecForName("GB2312");
```

```
// or // QTextCodec *codec = QTextCodec::codecForName("UTF-8");
```

```
QTextCodec::setCodecForLocale(codec);
```

⊕ In mainwindow.cpp

```
...
```

```
int ret = QMessageBox::warning(0, tr("PathFinder"), tr("您真的想要退出? "), QMessageBox::Yes | QMessageBox::No);
```



图像处理



图像处理



- ◆ Qt提供了4个处理图像的类。**QImage**, **QPixmap**, **QBitmap**, **QPicture**。它们有着各自的特点。
- ◆ **QImage**优化了I/O操作，可以直接存取操作像素数据。
- ◆ **QPixmap**优化了在屏幕上显示图像的性能。
- ◆ **QBitmap**从**QPixmap**继承，只能表示两种颜色。
- ◆ **QPicture**是可以记录和重启**QPrinter**命令的类。



转换



◆ 在QImage和QPixmap之间转换

```
QImage QPixmap::toImage();
```

```
QPixmap QPixmap::fromImage( const QImage& );
```



读入和保存



- ◆ 如下代码使用**QImageReader**和**QImageWriter**类进行，这些类在保存时通过文件的扩展名确定文件格式

```
QPixmap pixmap( "image.png" );  
pixmap.save( "image.jpeg" );
```

```
QImage image( "image.png" );  
image.save( "image.jpeg" );
```

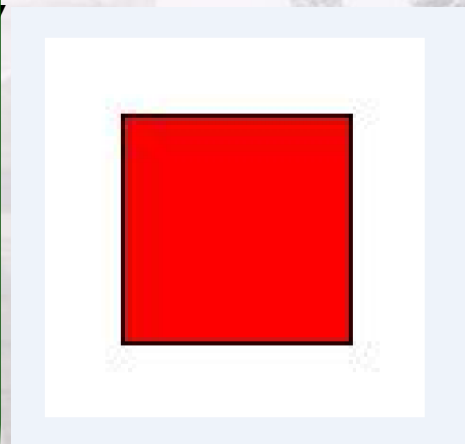


在QImage上绘制



- ◆ QImage是QPaintDevice的子类，因而QPainter可以在其上绘制

```
QImage image( 100, 100, QImage::Format_ARGB32 );  
QPainter painter(&image);  
  
painter.setBrush(Qt::red);  
  
painter.fillRect( image.rect(), Qt::white );  
painter.drawRect(  
    image.rect().adjusted( 20, 20, -20, -20 ) );  
  
image.save( "image.jpeg" );
```





在QPixmap上绘制

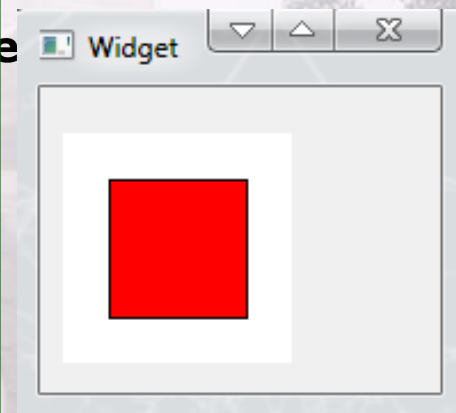


◆ QPixmap是QPaintDevice的子类，因而
QPainter可以在其上绘制

⊕ 主要用于屏幕绘制

```
void MyWidget::imageChanged( const QImage &image )
{
    pixmap = QPixmap::fromImage( image );
    update();
}

void MyWidget::paintEvent( QPaintEvent* )
{
    QPainter painter( this );
    painter.drawPixmap( 10, 20, pixmap );
}
```





坐标系统与坐标变换



坐标系统



- ◆ Qt坐标系统由**QPainter**控制，同时也由**QPaintDevice**和**QPaintEngine**类控制。
- ◆ Qt绘图设备默认坐标原点是左上角，**X**轴向右增长，**Y**轴向下增长，默认的单位在基于像素的设备上是像素，在打印机设备上是**1/72英寸(0.35毫米)**
- ◆ **QPainter**的逻辑坐标与**QPaintDevice**的物理坐标之间的映射由**QPainter**的变换矩阵**worldMatrix()**、视口**viewport()**和窗口**window()**处理。
 - ⊕ 未进行坐标变换的情况下，逻辑坐标和物理坐标是一致的



坐标值的表示方法



- ◆ 如果不进行坐标变换，直接进行绘图
 - ⊕ 可用**QPainter**的**window()**函数取得绘图窗口
 - ⊕ 然后在此绘图窗口内进行绘制
- ◆ 使用**QPoint**, **QSize**, 和**QRect**表示坐标值和区域
 - ⊕ **QPoint**: `point(x, y)`
 - ⊕ **QSize**: `size(width, height)`
 - ⊕ **QRect**: `point` 和 `size (x, y, width, height)`
- ◆ **QPointF/QSizeF/QRectF**用于表示浮点数坐标



坐标变换



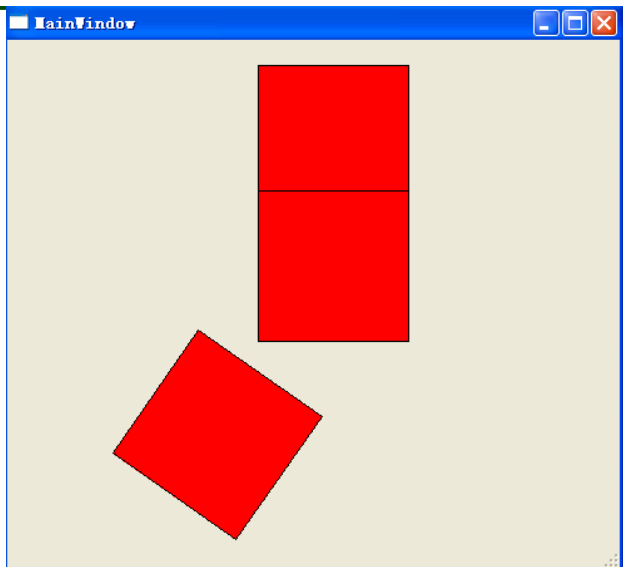
- ◆ 通常 **QPainer** 在设备的坐标系统上绘制图形，但 **QPainter** 也支持坐标变换。
 - ⊕ **QPainter::scale()** 函数：比例变换
 - ⊕ **QPainter::rotate()** 函数：旋转变换
 - ⊕ **QPainter::translate()** 函数：平移变换
 - ⊕ **QPainter::shear()** 函数：图形进行扭曲变换
- ◆ 所有变换操作的变换矩阵都可以通过 **QPainter::worldMatrix()** 函数取出。不同的变换矩阵可以使用堆栈保存。
 - ⊕ 用 **QPainter::save()** 保存变换矩阵到堆栈，用 **QPainter::restore()** 函数将其弹出堆栈。



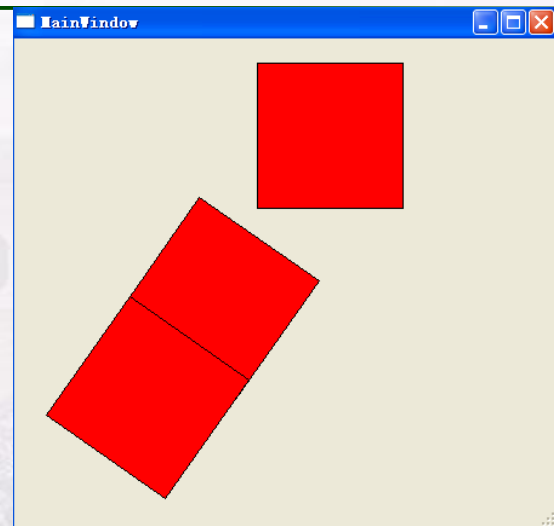
坐标变换



- ◆ 坐标变换的顺序很重要
- ◆ 在做平移变换、旋转变换和扭曲变换时，原点也很重要



```
p.setBrush(Qt::red);  
p.drawRect(200, 20, 120, 120);  
p.translate(0, 100);  
p.drawRect(200, 20, 120, 120);  
p.rotate(35);  
p.drawRect(200, 20, 120, 120);
```



```
p.setBrush(Qt::red);  
p.drawRect(200, 20, 120, 120);  
p.rotate(35);  
p.drawRect(200, 20, 120, 120);  
p.translate(0, 100);  
p.drawRect(200, 20, 120, 120);
```



坐标变换的保存和恢复



- ◆ 通过**save**和**restore**函数，可以将坐标变换的状态保存和恢复

```
QPoint rotCenter(50, 50);  
qreal angle = 42;
```

```
p.save();  
p.translate(rotCenter);  
p.rotate(angle);  
p.translate(-rotCenter);
```

```
p.setBrush(Qt::red);  
p.setPen(Qt::black);  
p.drawRect(25,25, 50, 50);
```

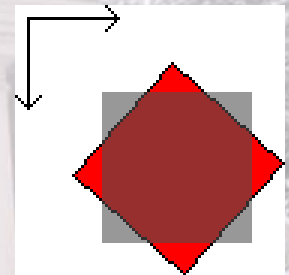
```
p.restore();
```

```
p.setPen(Qt::NoPen);  
p.setBrush(QColor(80, 80, 80, 150));  
p.drawRect(25,25, 50, 50);
```

应用变换

画红色矩形

画灰色矩形



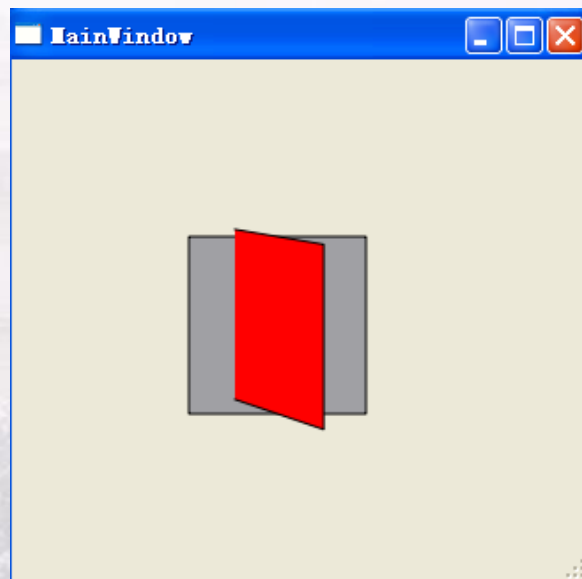


2.5D坐标变换



◆ 可以以任何坐标轴做旋转操作，以产生**3D**效果

```
p.setBrush(Qt::gray);  
p.setRenderHint(QPainter::Antialiasing);  
p.drawRect(100,100, 100, 100);  
QTransform t;  
t.translate(150,0);  
t.rotate(60, Qt::YAxis);  
p.setTransform(t, true);  
p.setBrush(Qt::red);  
p.drawRect(-50,100, 100, 100);
```





视口和窗口



- ◆ 视口表示物理坐标下的任意矩形。而窗口表示在逻辑坐标下的相同矩形。
 - ⊕ 视口由**QPainter**的**viewport ()**函数获取
 - ⊕ 窗口由**QPainter**的**window ()**函数获取
- ◆ 默认情况下逻辑坐标与物理坐标是相同的，与绘图设备上的矩形也是一致的。
- ◆ 使用窗口—视口变换可以使逻辑坐标符合自定义要求，这个机制通常用来完成设备无关的绘图代码。
 - ⊕ 通过调用**QPainter::setWindow()**函数可以完成坐标变换
 - ⊕ 设置窗口或视口矩形实际上是执行线性变换。本质上是窗口四个角映射到对应的视口四个角，反之亦然。因此，应注意保持视口和窗口**x**轴和**y**轴之间的比例变换一致，从而保证变换不会导致绘制变形。



绘图举例：表盘



表盘

- ◆ 自定义绘制
- ◆ 可以与键盘和鼠标交互





◆ 画表盘的背景

```
void CircularGauge::paintEvent(QPaintEvent *ev)
```

```
{
```

```
    QPainter p(this);
```

```
    int extent;
```

```
    if (width() > height())
```

```
        extent = height() - 20;
```

```
    else
```

```
        extent = width() - 20;
```

```
    p.translate((width() - extent) / 2, (height() - extent) / 2);
```

```
    p.setPen(Qt::white);
```

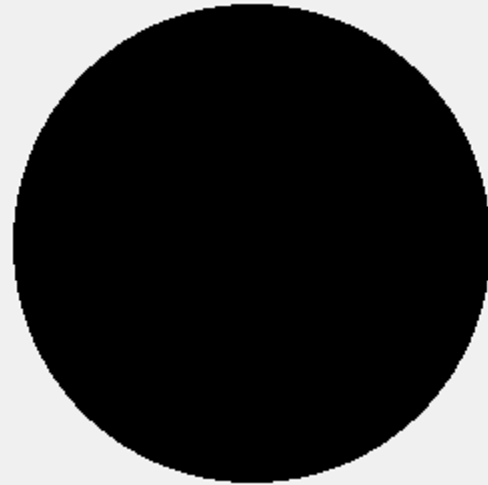
```
    p.setBrush(Qt::black);
```

```
    p.drawEllipse(0, 0, extent, extent);
```

```
    ...
```

将油表放在
中心位置

画背景圆形



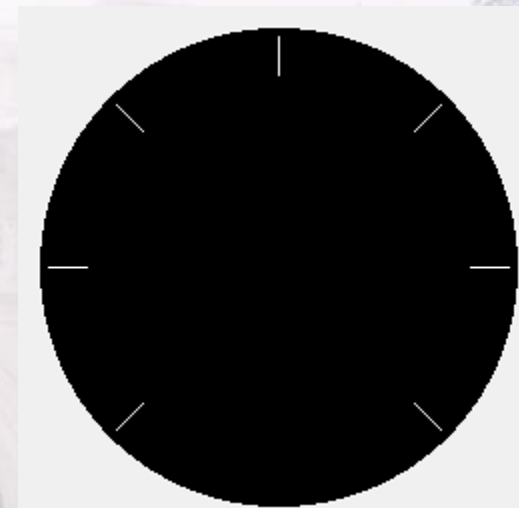


◆ 画表盘的刻度

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    ...

    p.translate(extent/2, extent/2);
    for(int angle=0; angle<=270; angle+=45)
    {
        p.save();
        p.rotate(angle+135);
        p.drawLine(extent*0.4, 0, extent*0.48, 0);
        p.restore();
    }

    ...
}
```



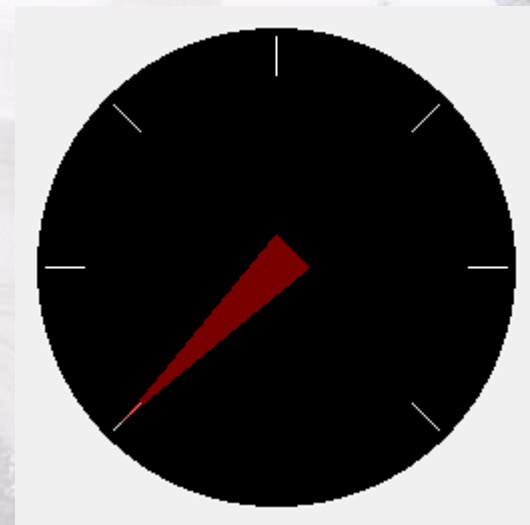
注意save和restore函数
简单调用rotate(45)会增大舍入误差



◆ 画表盘的指针

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    ...

    p.rotate(m_value+135);
    QPolygon polygon;
    polygon << QPoint(-extent*0.05, extent*0.05)
              << QPoint(-extent*0.05, -extent*0.05)
              << QPoint(extent*0.46, 0);
    p.setPen(Qt::NoPen);
    p.setBrush(QColor(255,0,0,120));
    p.drawPolygon(polygon);
}
```





响应事件



◆ 除了 **paintEvent**，还有

- ⊕ 键盘事件
- ⊕ 鼠标事件
- ⊕ 窗口事件
- ⊕ 定时器事件
- ⊕ 。 。 。



响应键盘事件



- ◆ 重写 **keyPressEvent**
- ◆ 键按下时响应
- ◆ 将未处理的按键传给基类处理

```
void CircularGauge::keyPressEvent(QKeyEvent *ev)
{
    switch(ev->key())
    {
        case Qt::Key_Up:
        case Qt::Key_Right:
            setValue(value()+1);
            break;
        case Qt::Key_Down:
        case Qt::Key_Left:
            setValue(value()-1);
            break;
        case Qt::Key_PageUp:
            setValue(value()+10);
            break;
        case Qt::Key_PageDown:
            setValue(value()-10);
            break;
        default:
            QWidget::keyPressEvent(ev);
    }
}
```




响应鼠标事件



- ◆ 鼠标事件通过重写如下函数来处理
 - ⊕ **mousePressEvent**和**mouseReleaseEvent**
 - ⊕ **mouseMoveEvent**: 除非**mouseTracking**为真, 否则只有一个鼠标按键按下时才被调用
- ◆ **setValueFromPos**是一个私有函数, 用于将点转换为角度

```
void CircularGauge::mousePressEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}

void CircularGauge::mouseReleaseEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}

void CircularGauge::mouseMoveEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
```



加速绘制



- ◆ **paintEvent**函数有一个**QPaintEvent**参数
- ◆ **QPaintEvent**有两个方法
 - ⊕ **QRect rect()**: 返回需要重绘的矩形
 - ⊕ **QRegion region()**: 返回需要重绘的区域
- ◆ 重绘时, 尽量避免在**QPaintEvent**返回的矩形/区域外绘制复杂图形



为表盘添加事件过滤器



◆ 按键0时，时油表指向0

```
class KeyboardFilter : public QObject ...
```

```
{bool KeyboardFilter::eventFilter(QObject *o, QEvent *ev)
```

```
    if (ev->type() == QEvent::KeyPress)
```

```
        if (QKeyEvent *ke = static_cast<QKeyEvent*>(ev))
```

```
            if (ke->key() == Qt::Key_0)
```

```
                if (o->metaObject()->indexOfProperty("value") != -1 )
```

```
                {
```

```
                    o->setProperty("value", 0);
```

```
                    return true;
```

```
                }
```

```
    return false;
```

```
}
```

返回true，停止
对该事件的响应



安装事件过滤器



- ◆ 调用**installEventFilter**函数
- ◆ 由于该**filter**对象是应用于属性（**property**）的，它可以用于任何具有该属性的对象，如**QSlider**, **QDial**, **QSpinBox**等
 - ⊕ 如果勇于尝试，可以为**QApplication**添加事件过滤器

```
ComposedGauge compg;  
CircularGauge circg;
```

```
KeyboardFilter filter;
```

```
compg.installEventFilter(&filter);  
circg.installEventFilter(&filter);
```




谢谢！