



Qt 绘图

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn



课程主要内容

- 基本流程
- 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



Qt绘制事件（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（**发出绘制事件**）
 - **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

安全性高，有优化
 - **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

安全性低，无优化



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);

    ...
```



基本绘制流程

- **QPainter**类提供绘制操作
- **QPaintEngine**类提供平台相关的API（通常隐藏）
- **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, QPrinter, QSvgGenerator, QGLPixelBuffer, QGLFramebufferObject, ...



这种设计方式的优点在于确保了所有的绘图操作遵循**相同的流程**，可以很容易地开发相应的功能特性来支持**其它的设备类型**。



QT 2D绘图

- Qt4中的2D绘图部分由3个类支撑整个框架：
 - **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。



Qt 2D绘图

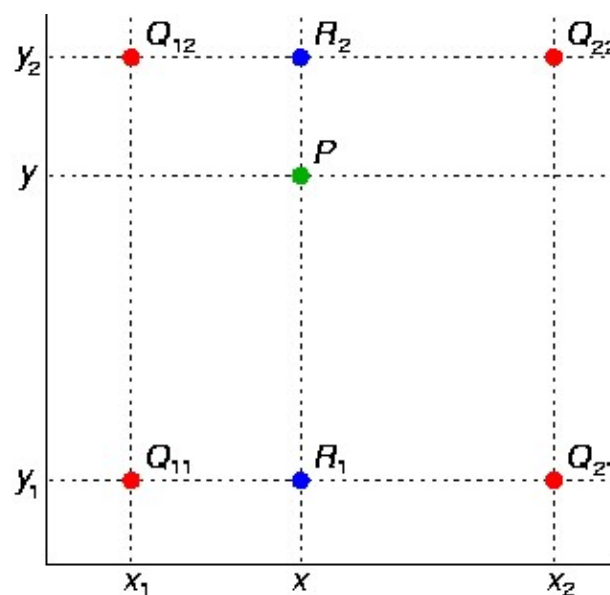


QPainter

- 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() | QImage表示的图像 | • 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() | 按QPainter指令绘制 | • drawText() | 文字 |
| • drawPie() | 扇形 | • drawTiledPixmap() | 平铺图像 |
| | | • drawLineSegments() | 绘制折线 |



QPaintDevice

- QPaintDevice是所有可绘图设备的基类
 - **QWidget**在Qt Widgets模块中是所有用户界面元素类的基类，可以接收鼠标，键盘及其它系统信息并且绘制自身呈现在屏幕上
 - **QImage**提供了硬件无关的图像表示形式，极大简化了I/O与像素存取，支持单色，8位，32位和alpha透明图像。
QImage的优点在于可以在不同平台确保像素的精确度，并且绘图过程是另外的线程而非当前GUI线程 **I/O处理**
 - **QPixmap**提供了与屏幕无关的图像显示方式，简化了图像在屏幕上的呈现。与QImage不同的是，QPixmap的像素数据是被底层的操作系统管理的，只能通过QPainter函数来操作或者转化为QImage来操作 **Qpixmap被直接绘制在屏幕**



画笔



画笔（QPen）

- 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





画笔

- 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中定义了三种端点风格用枚举类型Qt::PenCapStyle表示，分别为Qt::SquareCap, 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

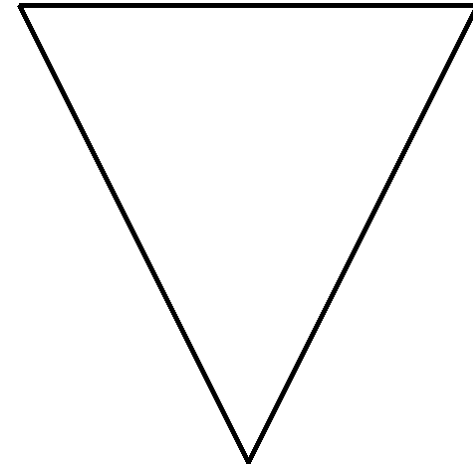


- ⊕ Qt::RoundCap



画笔示例

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





画刷



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/HSB**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。
- 填充模式包括有各种点、线组合的模式。



QColor

- QColor的构造函数

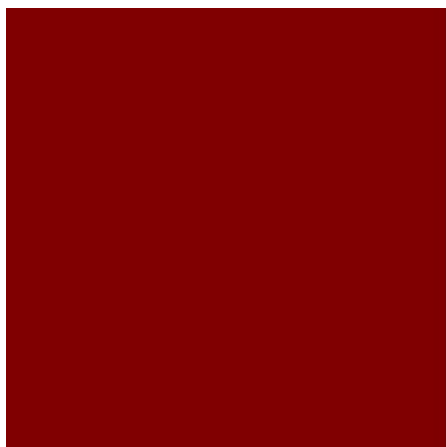
QColor(int r, int g, int b, int a)

- 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);
```

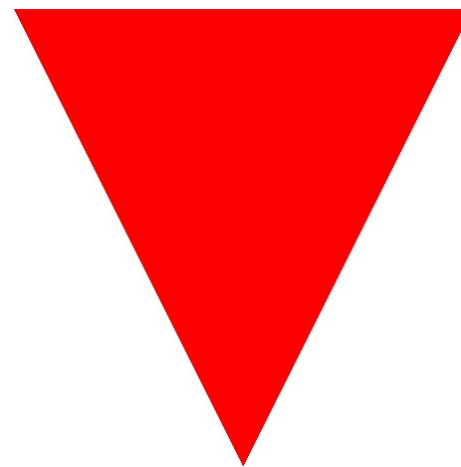
实色画刷

- 调用画刷构造函数

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

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

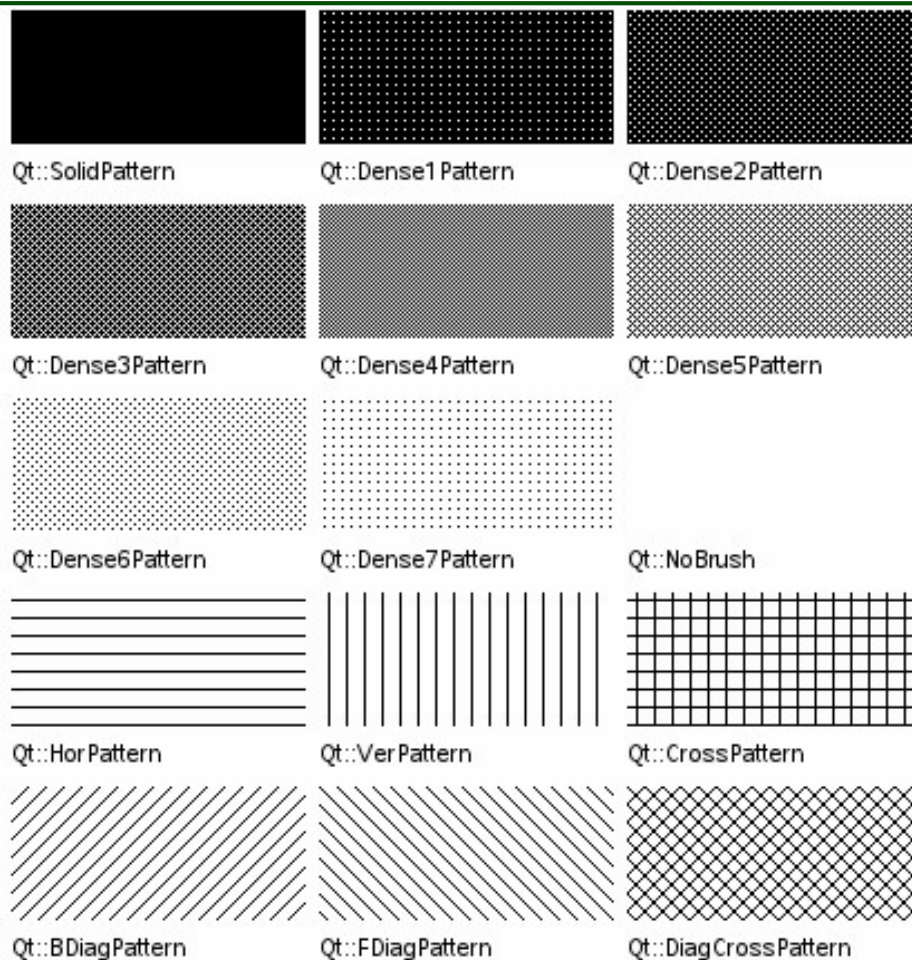
```
QPainter p(this);  
p.setPen(Qt::NoPen);  
p.setBrush(Qt::red);  
p.drawPolygon(polygon);
```



模式画刷

- 模式画刷构造函数

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

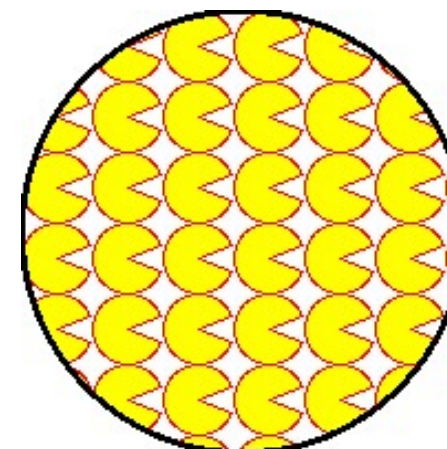


带纹理的画刷

- 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());
```



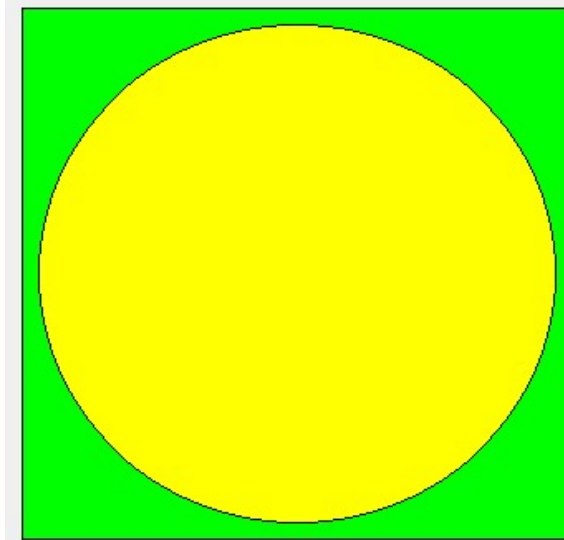


基本图形和文本绘制

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);  
}
```





绘制文本



文本绘制

- 文字对齐方式，文字的自动换行功能等

drawText(QPoint, QString)

drawText(QRect, QTextOptions, QString)

[QTextOption](#):文字对齐方式，文字的自动换行功能等

drawText(QRect, QTextOptions, 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，或之后设置字体

```
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 );
```



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);
```



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.setFont(font);
```

```
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



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

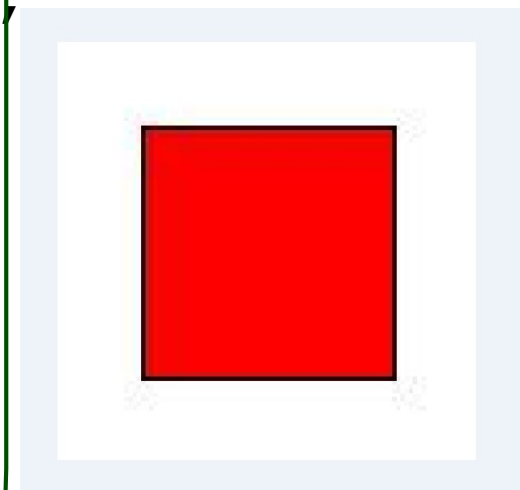
```
QPixmap pixmap( "image.png" );  
pixmap.save( "image.jpeg" );
```

```
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

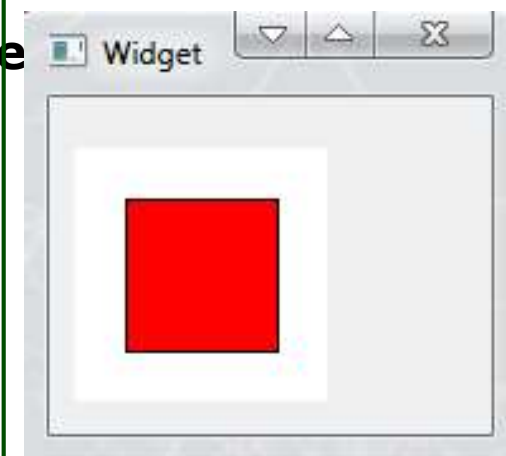
```
QImage image( 100, 100, QImage::Format_ARGB32 );  
QPainter painter(&image);  
  
painter.setBrush(Qt::red);  
  
painter.fillRect( image.rect(), Qt::white );  
painter.drawRect(  
    image.rect().adjusted( 20, 20, -20, -20 ) );  
  
image.save( "image.jpeg" );
```



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 $1/72$ 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

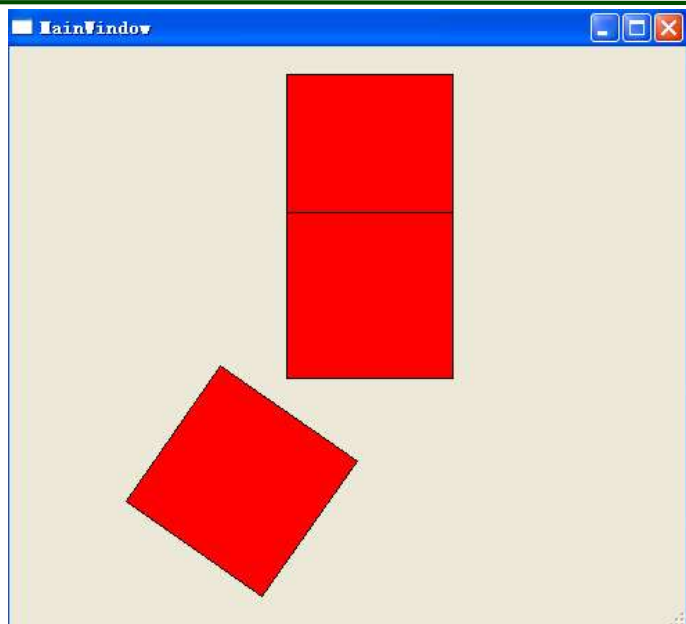


坐标变换

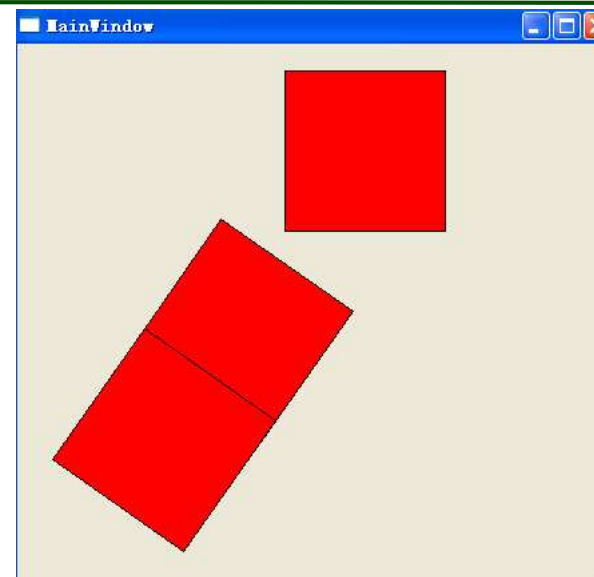
- 通常QPainer在设备的坐标系统上绘制图形，但QPainter也支持坐标变换。
 - QPainter::scale()函数：比例变换
 - QPainter::rotate()函数：旋转变换
 - QPainter::translate()函数：平移变换
 - QPainter::shear()函数：图形进行扭曲变换
- 所有变换操作的变换矩阵都可以通过QPainter::worldMatrix()函数取出。不同的变换矩阵可以使用堆栈保存。
 - 用QPainter::save()保存变换矩阵到堆栈，用QPainter::restore()函数将其弹出堆栈。

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



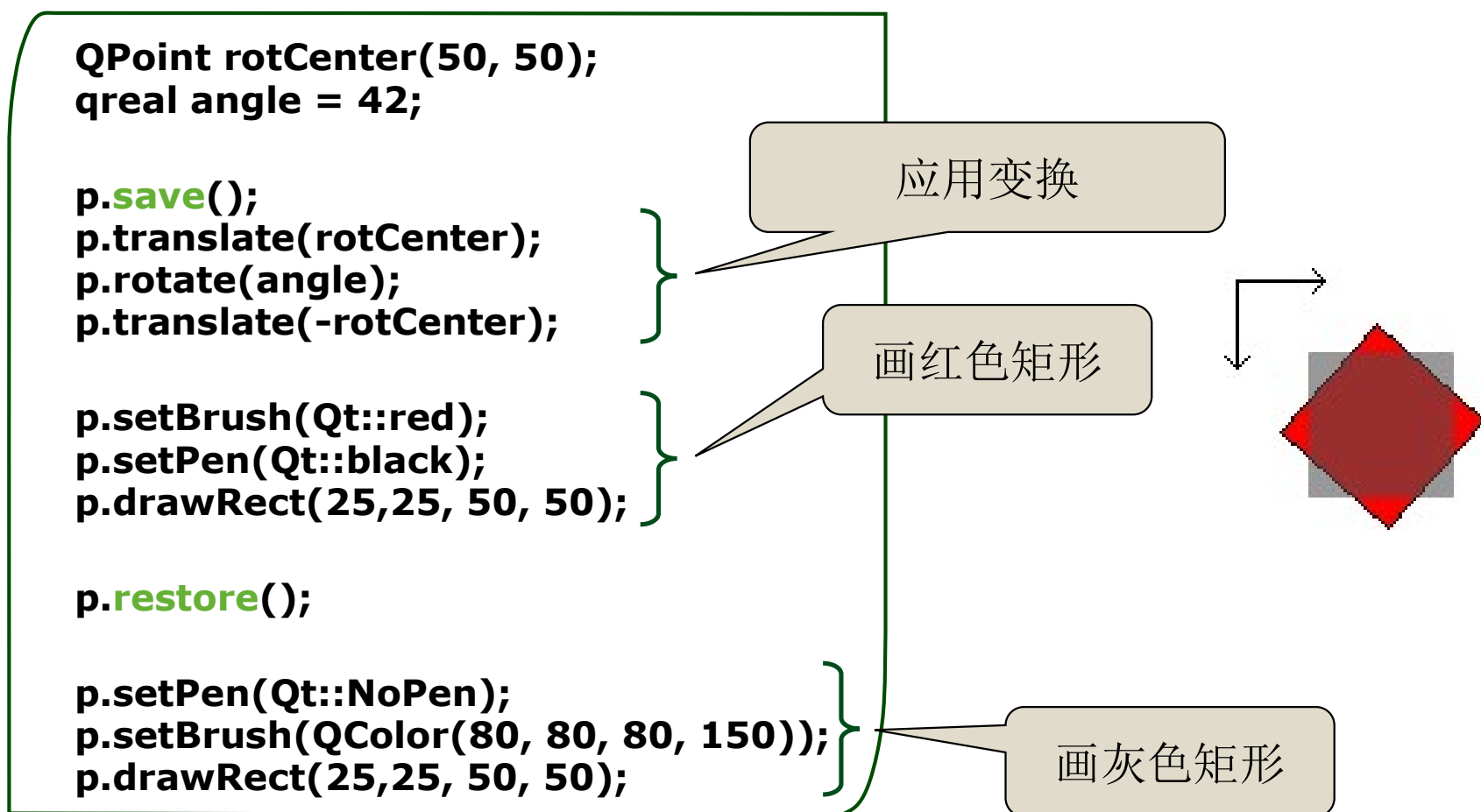
```
p.setBrush(Qt::red);  
p.drawRect(200, 20, 120, 120);  
p.translate(0, 100);  
p.drawRect(200, 20, 120, 120);  
p.rotate(35);  
p.drawRect(200, 20, 120, 120);
```



```
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

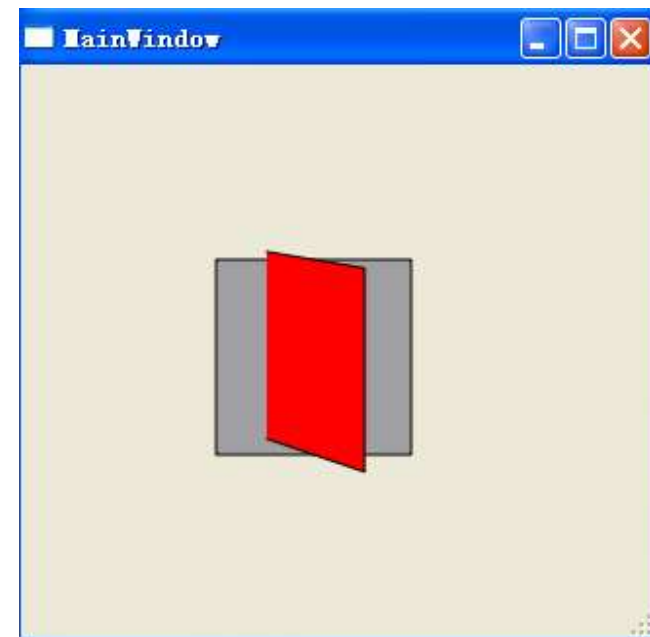
- 通过save和restore函数，可以将坐标变换的状态保存和恢复



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);
```





绘图举例：表盘

绘图举例：表盘

- 自定义绘制
- 可以与键盘和鼠标交互



表盘

- 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);
```

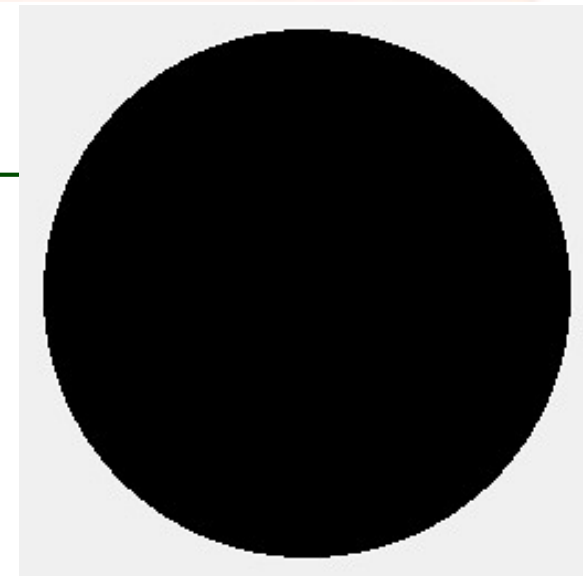
```
    p.setBrush(Qt::black);
```

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

```
    ...
```

将油表放在
中心位置

画背景圆形



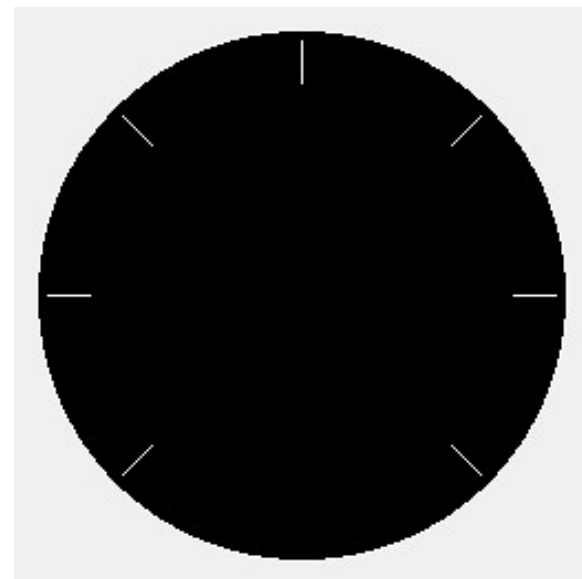
表盘

- 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

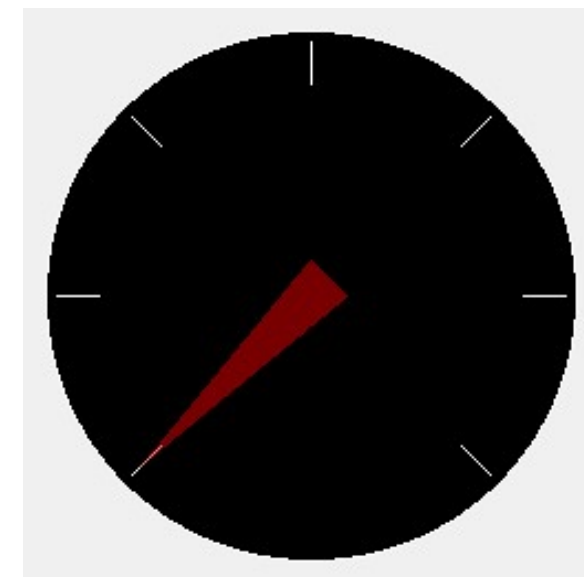


表盘

- 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可能被调用
- 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());
}
```



总结

- 基本流程
- 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

谢谢！