坐标系统是由 QPainter 的类来控制。再加上一个 QPaintDevice 和 QPaintEngine 类，QPainter 的形成 Qt 的绘画系统的基础。 QPainter 的用于执行绘制操作，QPaintDevice 是一个抽象的可以使用 QPainter 绘画的二维空间，并且 QPaintEngine 提供了一个让绘画者能将图绘制到不同类型设置的接口。

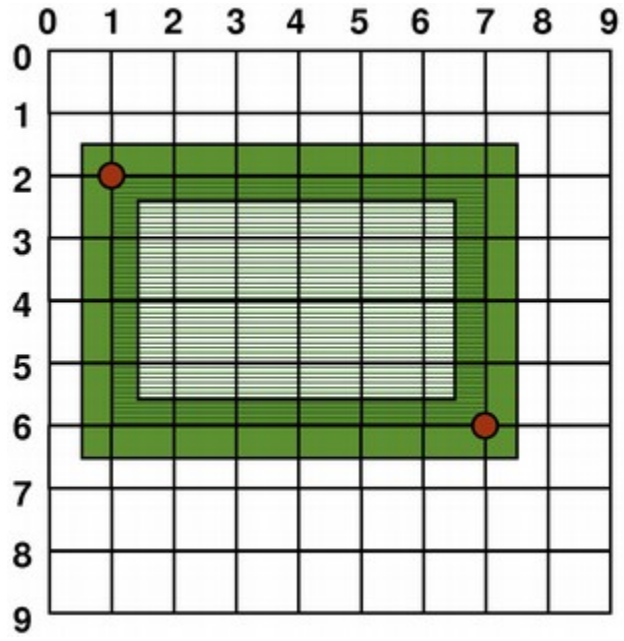
QPaintDevice 类是可绘图对象的基类：它的绘图功能被 QWidget，QImage，QPixmapQPicture 和 QOpenGLPaintDevice 类继承。绘图设备默认的坐标系统的坐标原点在坐上角。 x 值增加向右增加，y 值向下增加。在以像素为基础的设备上，默认的单位 1 像素，而在打印机上默认的单位为 1/72 英寸。

逻辑的 QPainter 坐标到 QPainter 的映射是通过 QPainter 的转换矩阵处理的，视口和“窗口”。逻辑和物理坐标系默认是一致的。 QPainter 的还支持坐标变换（如旋转和缩放）。

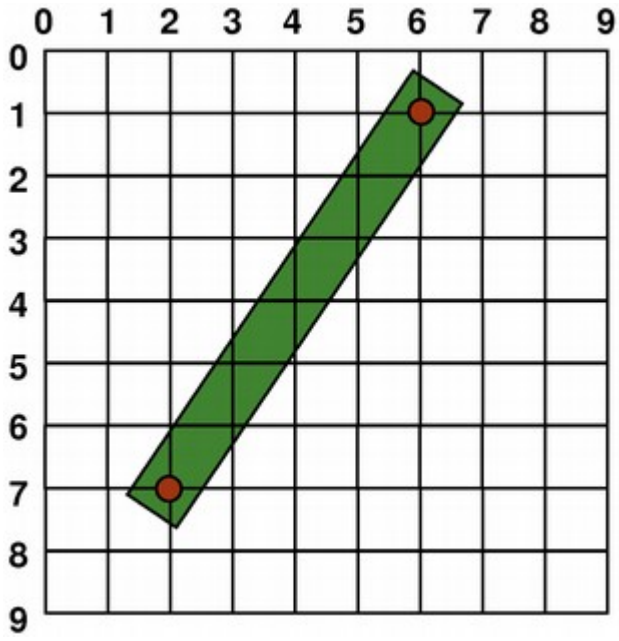
渲染

逻辑表示

原始图像的尺寸（宽度和高度）总是对应于它的数学模型，忽略粉刷笔的宽度：



QRect(1, 2, 6, 4)



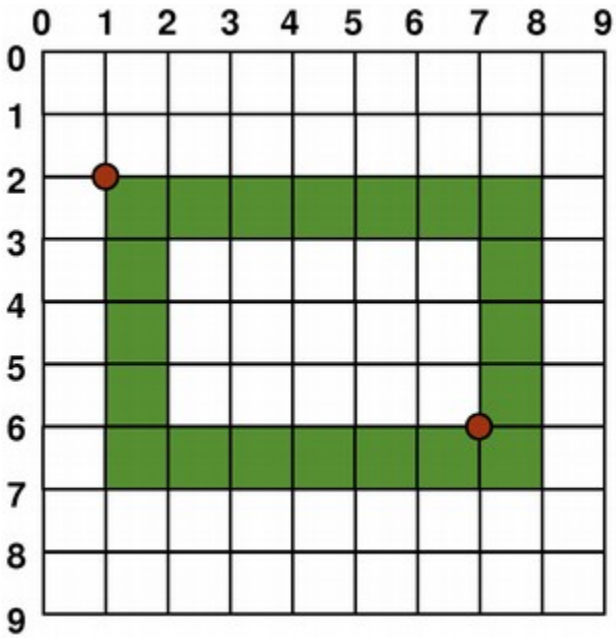
QLine(2, 7, 6, 1)

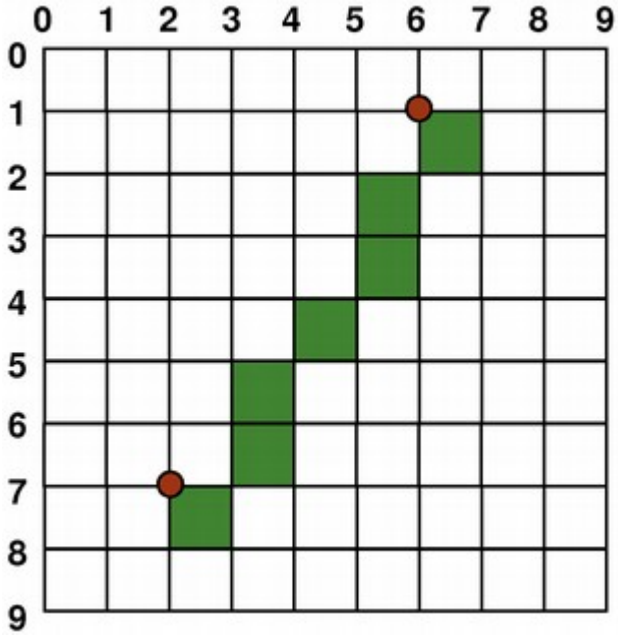
别名绘画

在绘制时，像素渲染通过 QPainter ::Antialiasing 渲染提示控制。

RenderHint 枚举用于为 QPainter 指定标志可以或不可以被任何给定的引擎遵守。QPainter ::Antialiasing 值指示引擎如果可能的话应该边缘反走样，即通过使用不同的颜色强度平滑边缘。

但默认绘画的别名和其他的规则：当一个像素宽的笔将被渲染到数学上的点的右下方。例如：





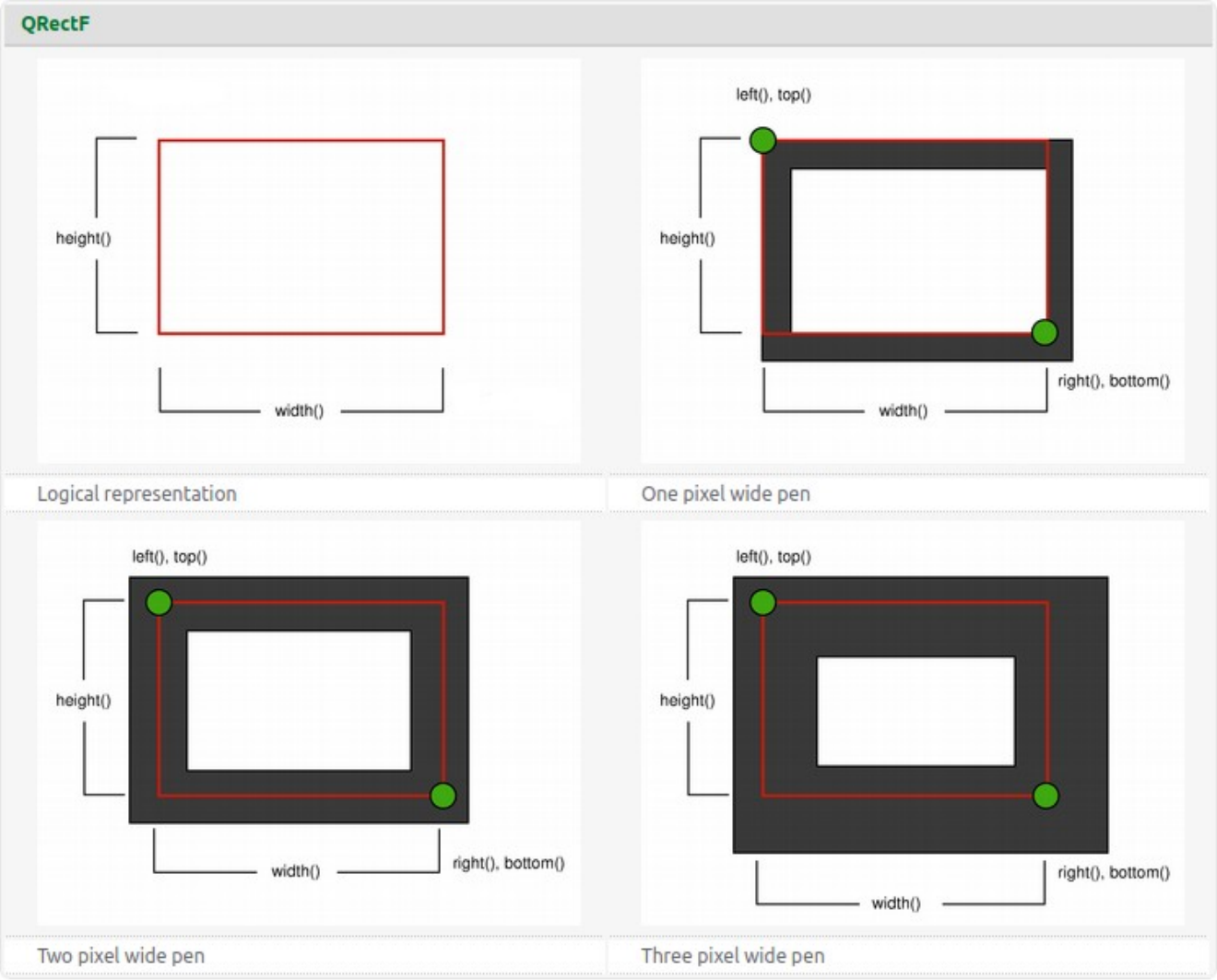

```
QPainter painter(this);

painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```

```
QPainter painter(this);

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

当用钢笔渲染偶数的数字的像素时，像素将对称地呈现周围的数学定义的点，当用钢笔渲染奇数的数字的像素时，备用像素将呈现到点的右侧，下面的数学点如是一个像素的情况下。请参见下面的 QRectF 图的具体例子。



需要注意的是由于历史原因 QRect::right()和 QRect::bottom()的返回值偏离矩形的真实右下角。

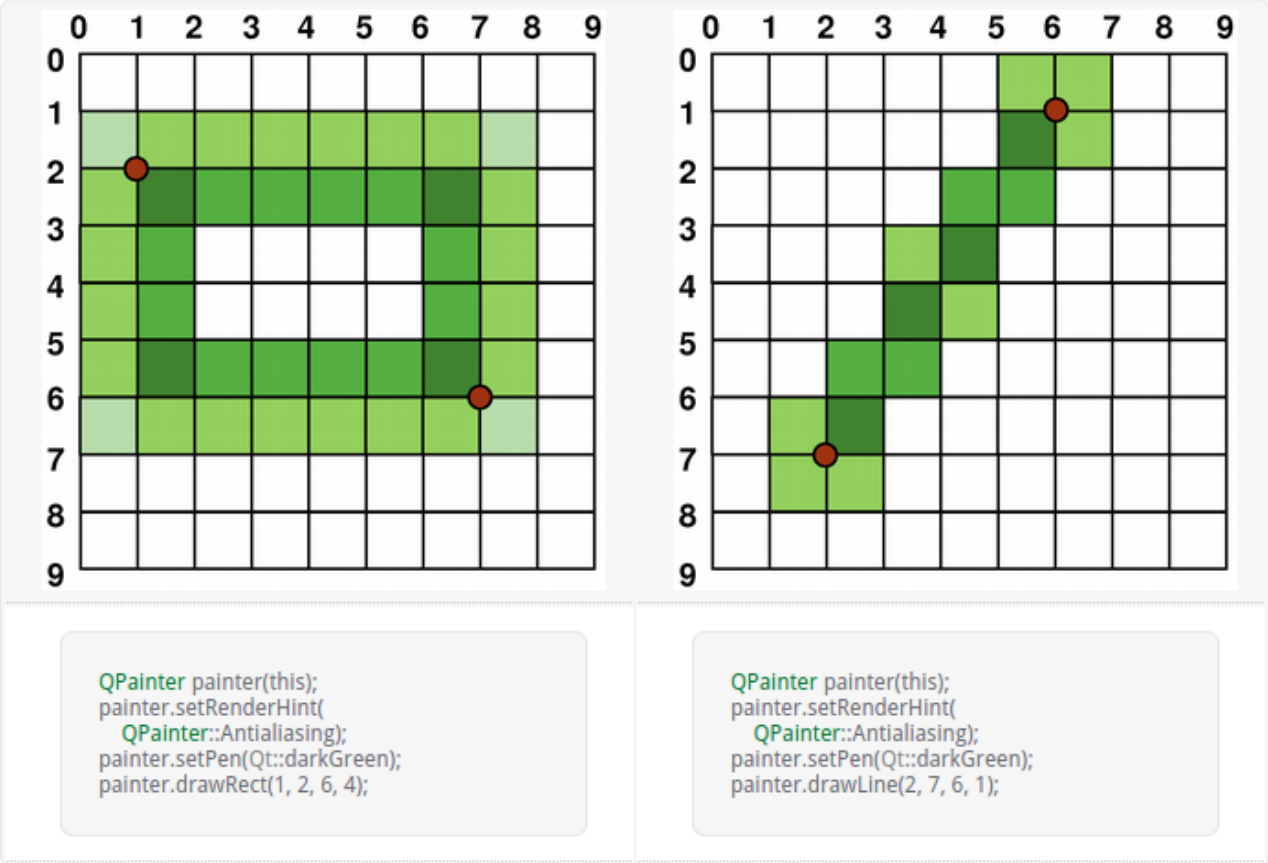
QRect 的 right()函数返回 left() + width() - 1，bottom()函数返回 top() + height() - 1。图中右下绿色的点显示的是这些函数的返回坐标。

我们建议您只需使用 QRectF 代替：QRectF 类使用浮点坐标精度（QRect 使用整数坐标）定义了一个矩形平面，并且 QRectF ::right()和 QRectF ::botton()函数返回真正的右下角。

可替代地，使用 QRect，应用 X()+width()和 y()+hight()寻找右下角，并避免 right()和 botton()函数。

抗锯齿绘画

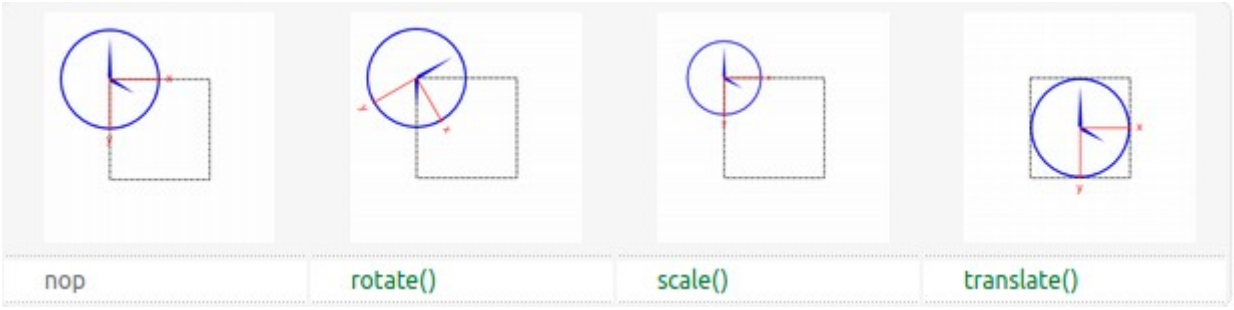
如果设置 QPainter 的抗锯齿渲染提示，像素将被对称地呈现在数学上定义的点的两侧：



转换

默认情况下，QPainter 操作在自身的坐标系相关设备上，但它也有对坐标转换的完整支持。

您可以通过给定的偏移使用了 QPainter ::scale()函数缩放坐标系统，可以顺时针旋转使用了 QPainter ::rotate()函数，你可以使用 QPainter ::translate()函数把转换它（即给点加一个给定的偏移）。



您也可以使用 QPainter ::shear()函数围绕原点扭曲坐标系统。全部转换操作均将操作在 QPainter 的转换矩阵上，你可以使用 QPainter :: worldTransform()函数恢复。矩阵转换平面上的一个点到另一个点。

如果你一遍又一遍需要相同的变换，你也可以使用 QTransform 对象和 QPainter::

worldTransform()和 QPainter:: setWorldTransform()函数。您可以在任何时候通过调用 QPainter ::save()函数保存 QPainter 的转换矩阵到内部栈。该 QPainter 的::restore()函数弹出回来。

一个经常需要变换矩阵，适用于各种绘画设备需要重复使用相同的绘制代码的时候。如果不转换，结果紧密绑定到绘画设备的分辨率上。打印机具有高分辨率，如每英寸 600 点，而屏幕通常每英寸具有 72 和 100 个点。

模拟时钟示例

模拟时钟示例说明了如何使用 QPainter 的变换矩阵来绘制一个自定义窗口控件。

我们建议任何进一步的阅读之前，编译和运行这个例子。尤其是，尝试调整窗口的不同的大小。


```
void AnalogClockWindow::render(QPainter *p)
{
    static const QPoint hourHand[3] = {
        QPoint(7, 8),
        QPoint(-7, 8),
        QPoint(0, -40)
    };
    static const QPoint minuteHand[3] = {
        QPoint(7, 8),
        QPoint(-7, 8),
        QPoint(0, -70)
    };

    QColor hourColor(127, 0, 127);
    QColor minuteColor(0, 127, 127, 191);

    p->setRenderHint(QPainter::Antialiasing);
    p->translate(width() / 2, height() / 2);
```

```
int side = qMin(width(), height());
p->scale(side / 200.0, side / 200.0);
```

我们转换坐标系统，使点（0,0）在控件中心而不是在左上角。我们通过 side/100 扩展系统，其中一面可能是控件的宽度，也可能是或高度，取其中最短的。我们希望时钟是正方形，即使该设备不是。

这将为我们提供一个 200×200 正方形区域，原点中心在（0,0）。我们绘制的东西适合窗口的最大的正文形。

```
QTime time = QTime::currentTime();

p->save();
p->rotate(30.0 * ((time.hour() + time.minute() / 60.0)));
p->drawConvexPolygon(hourHand, 3);
p->restore();
```

我们绘制时钟的时针旋转坐标系，并调用 QPainter :: drawConvexPolygon()。谢谢旋转，它的绘制指出了正确的方向。

多边形被指定为交替 x、y 值的阵列，将存储在 hourHand 静态变量（在函数开头定义），其对应于四个点（2,0），（0,2），（ - 2，0）和（0，-25）。

调用 QPainter :: save()和 QPainter::restore()围绕代码保证下面的代码不会被我们使用的转化受到干扰。

```
p->save();
p->rotate(6.0 * (time.minute() + time.second() / 60.0));
p->drawConvexPolygon(minuteHand, 3);
p->restore();
```

我们做相同的时钟的分针，这是由四个点（1,0）、（0,1）（ - 1，0）和（0，-40）来定义的。

```
p->setPen(minuteColor);

for (int j = 0; j < 60; ++j) {
    if ((j % 5) != 0)
        p->drawLine(92, 0, 96, 0);
    p->rotate(6.0);
}
```

最后，我们画出的时钟界面，其中包含十二个短行以 30 度间隔。最后，以旋转的方式绘

制不是很有用，但我们绘画这样做也没关系。

有关变换矩阵的详细信息，请参阅 QTransform 文档。

窗口视口转换

使用 QPainter 绘图时，我们使用逻辑坐标指定点，逻辑坐标将会转换成物理坐标。

逻辑坐标到物理坐标的映射是由 QPainter 的世界变换 worldTransform()（在转换部分所述）、QPainter 的 virwport()和 window()来处理的。视口表示物理坐标指定一个任意的矩形。“窗口”用逻辑坐标描述相同的矩形。默认的逻辑和物理坐标系一致，并且等同于绘制设备的矩形。

使用窗口-视口转换可以使逻辑坐标系符合你的喜好。该机制还可以用于使绘图代码独立于绘画设备。例如，你可以通过调用了 QPainter :: setWindow()函数使逻辑坐标从（-50，-50）延伸至（50，50），（0，0）为中心原点：

```
QPainter painter(this);
painter.setWindow(QRect(-50, -50, 100, 100));
```

现在，逻辑坐标（-50，-50）对应于绘制设备的物理坐标（0，0）。绘画设备独立，你代码总是操作在指定的逻辑坐标。

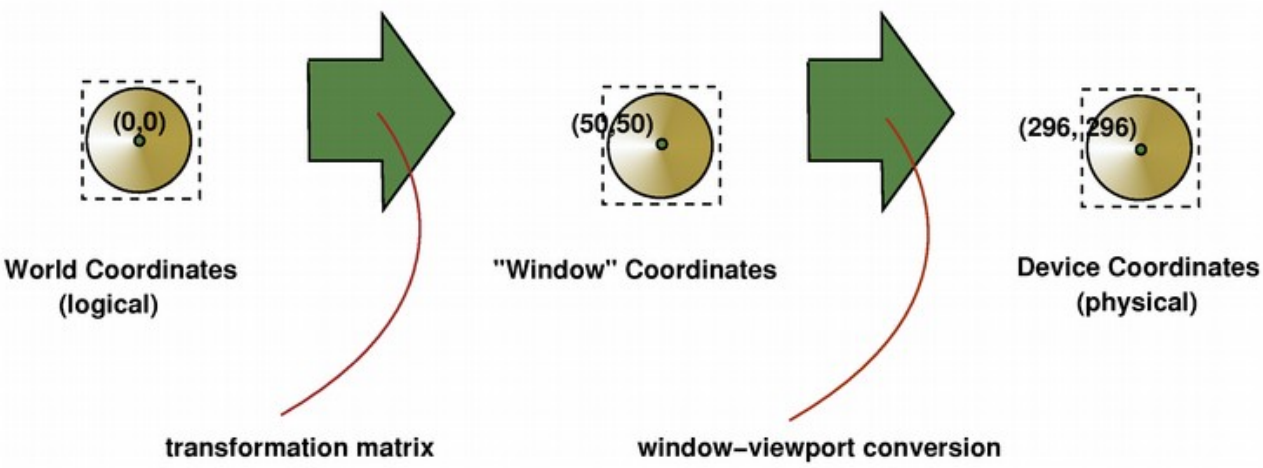
通过设置“窗口”或视口矩形，执行坐标的线性变换。请注意，“窗口”的每个角落映射为视口的相应的拐角处，反之亦然。由于这个原因，通常让视口和“窗口”保持相同的纵横比，以防止变形是一个好主意：

```
int side = qMin(width(), height())
int x = (width() - side )/ 2;          注意：这里原文档 int x=(width() - side / 2）应该是错的
int y = (height() - side) / 2;

painter.setViewport(x, y, side, side);
```

如果我们的逻辑坐标系成为正文形，那我们也应该使用 QPainter :: setViewport()函数使视口成为正文形。在上面的例子中，我们使它成为适合绘画设备的最大的正文形。通过设备窗口或视口时考虑绘画设备的尺寸，保存绘画代码相对于绘画设备的独立性是可能的。

注意，窗口视口变换是唯一的线性变换，即它不执行剪裁。这意味着，如果你在当前设置窗口之外绘画，你的绘画依然采用了相同的线性代数方法转化为视口。



视口，“窗口”和变换矩阵确定逻辑 QPainter 坐标映射到绘制设备的物理坐标。默认情况下，世界变换矩阵为单位矩阵，和“窗口”和视口设置等同于绘画设备的设置，即世界、“窗口”和设备坐标系统是等价的，但正如我们所看到的，系统可以使用变换运算和窗口视口变换进行操作。上图描述了这个过程。