The Graphics View Framework

关键词翻译对照表：

Graphics View：图形视图。

Scene：场景 /场景管理器（ Scene 同时担负着管理场景中的对象，建立索引等工作）。

Item：这里翻译为对象， Graphics View Framework 下的 GraphicsItem 是场景中可以被显示的元素。这里翻译成对象便于理解。

Graphics Item：图形对象。

Event：事件，等同于 Windows 下的消息。

正文：

图形视图（ Graphics View ）提供了支持大量自定义的二维图形对象（ Item ，这里译为“ 对象 ”，方便大家理解）交互（ Interaction ）的管理器，以及一个支持缩放和旋转操作的视图 widget 用于显示这些元素。

该框架包含了事件（ Event ，在 Windows 下可以理解为“ 消息 ”）传播的框架，支持场景管理器中精确的交互能力，以双精度浮点数表示对象位置、大小等属性的变化。图形元素还能处理键盘事件、鼠标按下 / 移动 / 释放和双击的时间，同时也能跟踪鼠标移动。

图形视图使用 BSP 树（ Binary Space Partitioning ，二叉空间分割）提供对图形对象的快速查找，可以想像，即使是包含数以百万计对象的超大场景，也能够进行实时显示。

图形查看 Qt 中引入 4.2 ，取代其前身 QCanvas。 如果您要从 QCanvas 中移植过来，见 移植到图形视图 。

主题：

- 图形视图架构
  - 场景
  - 视图
  - 对象
- 图形视图框架中的类
- 图形视图坐标系
  - 对象坐标
  - 场景坐标
  - 视图坐标
  - 坐标映射
- 主要特点

        - 缩放和旋转
        - 打印
        - 拖放
        - 鼠标指针和 tooltip
        - 动画
        - OpenGL 渲染
        - 元素组
        - widgets 和布局
            - QGraphicsWidget
            - QGraphicsLayout
        - 嵌入式 widget 支持
- 性能
    - 浮点运算指令

图形视图架构

图形视图提供基于图像对象的方式来实现 model-view 的编程模式，这一点很像例程 InterView 中的辅助类 QTableView， QTreeView 和 QListView。不同的视图可以显示一个场景，场景则包含了不同的几何形状的对象。

场景

QGraphicsScene 提供了图形视图的场景管理器。场景管理器有如列职责：

- 提供一个用于管理大量对象的快速接口
- 将事件传递到每个对象上
- 管理对象的状态
- 提供未进行坐标变换的渲染功能，主要用于打印

场景管理器是图形对象 QGraphicsItem 的容器。调用 QGraphicsScene::addItem()将对象添加到场景中后，你就可以通过调用场景管理器中的不同的查找函数来查找其中的图形对象。QGraphicsScene::items()函数及其重载函数可以返回所有通过点、矩形多边形或路径等不同方式选中的所有对象。 QGraphicsScene::itemAt()返回在指定点位置上最上面的对象。所有找到的对象保持按照层叠递减的排列顺序（即第一个返回的对象是最顶层，和最后一个项目是最底层的对象）。

```
QGraphicsScene scene;
QGraphicsRectItem *rect = scene.addRect(QRectF(0, 0, 100, 100));

QGraphicsItem *item = scene.itemAt(50, 50);
// item == rect
```

QGraphicsScene 的事件传递机制负责将场景时间传递给图形对象，同时也管理对象之间的时间传递。如果场景在某个位置得到鼠标按下的消息，就将该事件传递给这个位置上的对象。

QGraphicsScene 同时还管理对象的状态，例如对象的选中状态和焦点状态。您可以通过调用 QGraphicsScene::setSelectionArea()，传递一个任意形状给场景管理器，选中其中包含的对象。此功能也是 QGraphicsView 中拉框选择（ rubber band ）的基础。通过调用 QGraphicsScene::selectedItems()可以获取当前选择集中的所有对象。另外一种通过 QGraphicsScene 来管理的状态是：一个图形对象是否能够相应键盘的焦点切换 .你可以调用 QGraphicsScene::setFocusItem ()或者 QGraphicsItem::setFocus ()将键盘焦点切换到对于的图形对象上，或者通过 QGraphicsScene::focusItem ()获取当前的焦点对象。

最后， QGraphicsScene 允许你通过 QGraphicsScene::render ()将部分场景绘制到 paint device 上。你可以在本文档中关于“打印”的章节了解更多关于这一点的更多细节。

视图

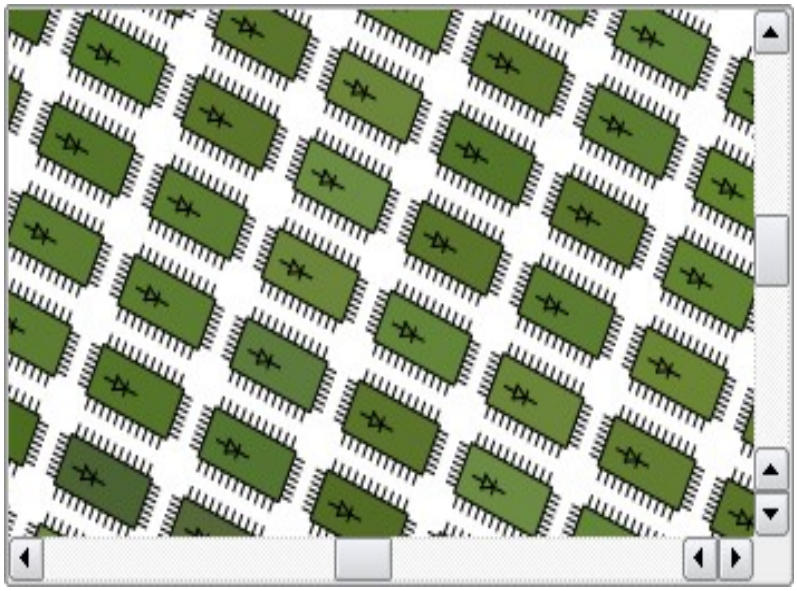
QGraphicsView 提供了视图 widget，将场景中的内容显示出来。你可以用几个不同的视图来观察同一个场景，从而实现对于同一数据集的不同 viewport。该 Widget 同时也是 scroll area，为大场景提供滚动条。如果要启用 OpenGL 支持，可调用 QGraphicsView::setViewport 将 QGLWidget 设置为其 viewport。

```
QGraphicsScene scene;
myPopulateScene(&scene);

QGraphicsView view(&scene);
view.show();
```

视图接受键盘和鼠标消息，并将这些消息转换成场景事件（同时将视图坐标转换为场景坐标），然后将事件发送给可见视图。

通过操作变换矩阵 QGraphicsView::transform ，视图可以对场景的坐标系统进行变换，从而实现缩放、旋转等高级查看功能。为了方便起见， QGraphicsView 同时也提供了在视图坐标和场景坐标之间变化的函数： QGraphicsView::mapToScene () 和 QGraphicsView::mapFromScene ()。



对象

QGraphicsItem 是场景中所有图形独享的基类。图形视图提供了几种标准的对象：矩形 (

QGraphicsRectItem)，椭圆（ QGraphicsEllipseItem ）和文本对象（ QGraphicsTextItem)。但是 QGraphicsItem 最强大的功能是支持定制的图形对象。 QGraphicsItem 支持如下特征：

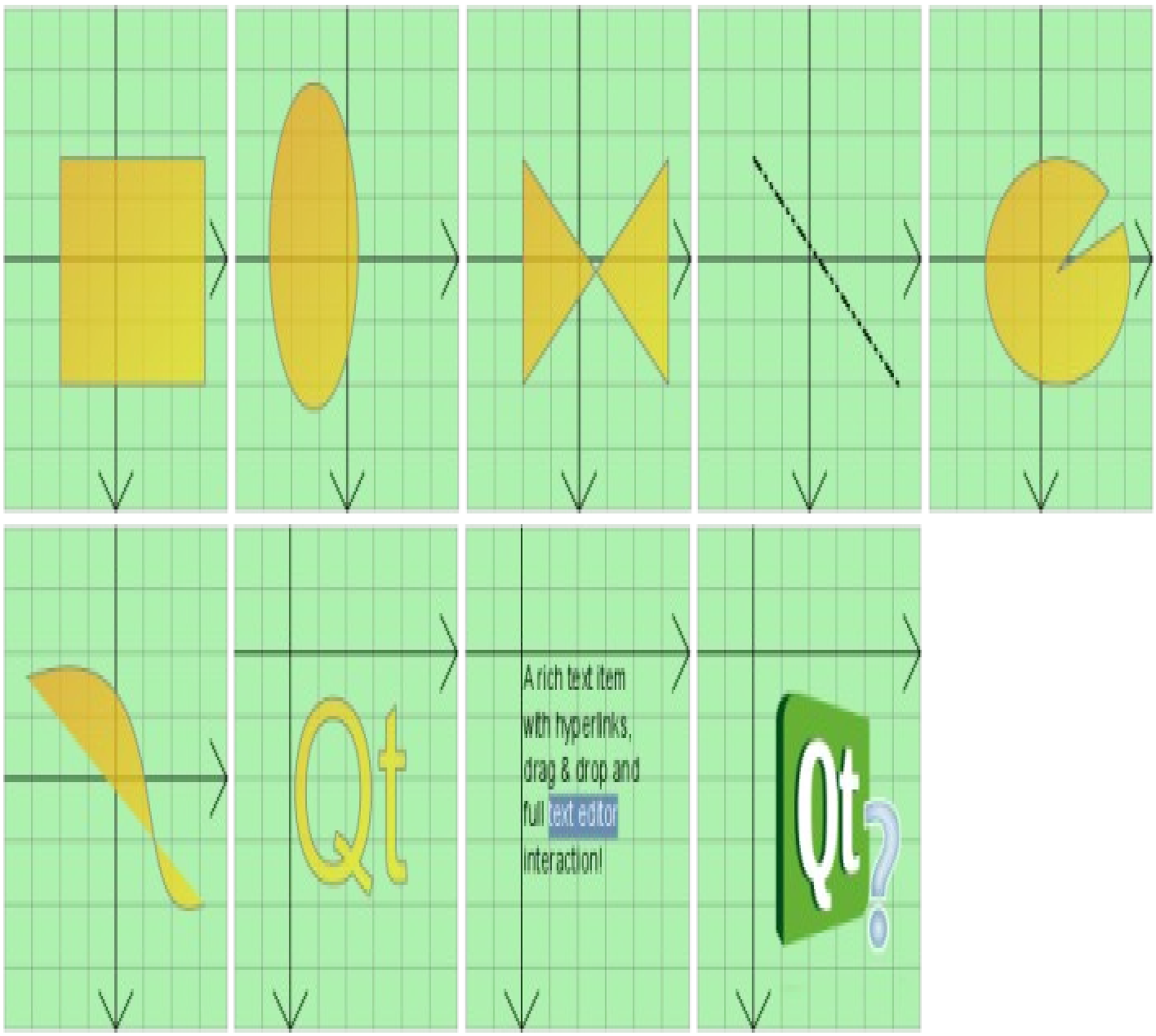
- 鼠标按下、移动、释放和双击事件，同时还支持鼠标悬浮事件、滚轮事件和上下文菜单事件。
- 键盘输入焦点和键盘事件。
- 拖放。
- 组合：通过父对象 -子对象进行组合，或者通过 QGraphicsItemGroup 组合。
- 碰撞检测。

与 QGraphicsView 类似，处于局部坐标系下的图形对象，也提供了图形对象和场景之间的映射函数。和 QGraphicsView 一样，图形对象同时还可以通过矩阵来变换其自身的坐标系统，这一点对于单个图形对象的旋转和缩放非常有用。

图形视图架构

个图形对象可以包含其他对象（子对象）。父对象的变换矩阵同样也会应用到子对象上。但是，不管一个对象累积了多少变换， QGraphicsItem::collidesWith()仍然会在局部坐标系下进行计算。

QGraphicsItem 通过 QGraphicsItem::shape 和 QGraphicsItem::collidesWith 来实现碰撞检测，这两个函数都是虚函数。 QGraphicsItem 通过 QGraphicsItem::shape 获取局部坐标系下的 QPainterPath 对象，来完成碰撞检测。不过如果你想提供你自己的碰撞检测机制，你可以通过自定义 QGraphicsItem::collidesWith 函数来实现。



图形视图架构 中的类

下面的类提供了创建交互 式应用 程序的框架：

QAbstractGraphicsShapeItem	所有路径对象的共同基类
QGraphicsAnchor	代表了 QGraphicsAnchorLayout 中 两个项目之间的 anchor
QGraphicsAnchorLayout	如何将 widgets anchor 到图形视图中
QGraphicsEffect	所有图形特效的基类
QGraphicsEllipseItem	椭圆对象，可以直接添加到 QGraphicsScene

QGraphicsGridLayout	管理 widgets 在图形视图中的布局
QGraphicsItem	在 QGraphicsScene 中所有图形对象的基类
QGraphicsItemAnimation	为 QGraphicsItem 提供简单的 动画支持
QGraphicsItemGroup	将多个图形对象组合成一个对象
QGraphicsLayout	图形视图中所有布局类的基类
QGraphicsLayoutItem	允许布局类管理的自定义对象
QGraphicsLineItem	直线对象，可以直接添加到 QGraphicsScene
QGraphicsLinearLayout	管理 widgets 在图形视图中的的水平或者垂直方向上的布局
QGraphicsObject	所有需要处理信号 /槽 /属性的图形对象。
QGraphicsPathItem	路径对象，可以直接添加到 QGraphicsScene
QGraphicsPixmapItem	位图对象，可以直接添加到 QGraphicsScene
QGraphicsPolygonItem	多边形对象，可以直接添加到 QGraphicsScene
QGraphicsProxyWidget	widget 代理，用于将一个 QWidget 对象嵌入 一个 QGraphicsScene 中
QGraphicsRectItem	矩形对象，可以直接添加到 QGraphicsScene
QGraphicsScene	管理大量二维图形对象的管理器
QGraphicsSceneContextMenuEvent	在图形视图框架中的上下文菜单事件
QGraphicsSceneDragDropEvent	图形视图框架中的拖放拖放事件
QGraphicsSceneEvent	图形视图框架中所有事件的基类
QGraphicsSceneHelpEvent	Tooltip 显示时发出的事件
QGraphicsSceneHoverEvent	图形视图框架中的悬停事件
QGraphicsSceneMouseEvent	图形视图框架中的鼠标事件
QGraphicsSceneMoveEvent	图形视图框架中的 widget 移动事件

QGraphicsSceneResizeEvent	图形视图框架中的 widget 大小改变的事件视
QGraphicsSceneWheelEvent	图形视图框架中的鼠标滚轮时间
QGraphicsSimpleTextItem	简单的文本对象，可以直接添加到 QGraphicsScene 中
QGraphicsSvgItem	可以用来呈现的 SVG 文件内容的 QGraphicsItem 对象
QGraphicsTextItem	文本对象，可以直接添加到 QGraphicsScene ，用于 显示带格式的文本
QGraphicsTransform	创建 QGraphicsItems 高级矩阵变换的抽象类
QGraphicsView	显示 QGraphicsScene 内容的 widget
QGraphicsWidget	QGraphicsScene 中所有 widget 的基类
QStyleOptionGraphicsItem	用于描述绘制 QGraphicsItem 所需的 参数

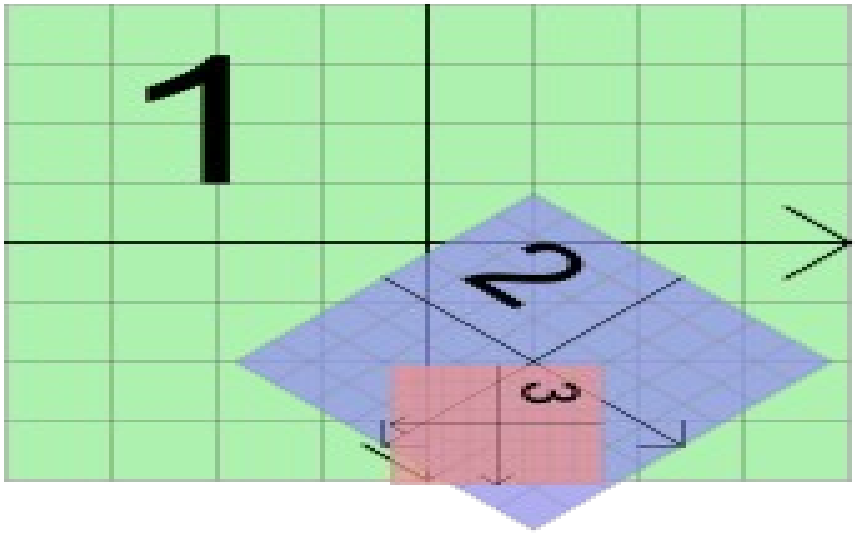
图形视图坐标系

图形视图建立在直角坐标系基础上，图形对象的位置和几何形状由两组数据来表示： x 坐标和 y 坐标。如果使用未变换的视图来观察场景，场景中的一个单元将会表现为屏幕上的一个像素。

注意 ： 图形视图使用了 Qt 的坐标系，不支持反转的 y 轴坐标系统（即 y 向上为正方向）。

图形视图使用了三种有效的坐标系：对象坐标，场景坐标和视图坐标。为了简化你的实现工作，图形视图提供了一系列非常方便的函数来进行三个坐标系下的坐标变换。

绘制的时候，图形视图的场景坐标对应于 QPainter 的逻辑坐标，视图坐标与设备坐标一直。在 The Coordinate System 一文中你可以阅读更多关于逻辑坐标和设备坐标关系的内容。



对象坐标系

图形对象建立其自身的局部坐标系下。该坐标系通常以中心点（ 0， 0）为中心，同时该中心点也是各种矩阵你变换的中心。对象坐标系下的几何元素通常用点、线或者矩形来表示。

创建自定义图形对象的时候，你只需关注对象坐标系即可。 QGraphicsScene 和 QGraphicsView 会执行所有相关的变换，这样一来我们实现自定义对象就容易多了。比如说，当你接受鼠标按下或者拖拽事件的时候，事件位置已经被转换到了对象坐标系下。类似的，对象的包围盒和形状都是基于对象坐标系的。

对象的*位置* 是对象坐标系下的中心点在其父对象坐标系下的位置。对于所有没有父对象的对象来说，场景就是其父对象。因此最顶层对象的位置就是其在场景中的位置。

子 对象坐标系是相对于父对象坐标系来说的一个概念。如果子节点没有进行矩阵变换，那么在子对象坐标系和父对象坐标系的差异就和这些对象在父对象中的偏移。比 如说，如果一个未经变换的子对象精确的位于父对象的中心点，那么这两个对象的坐标系就是完全一致的。如果子对象的位置是（ 10， 0），那么子对象的（ 0， 10）点就位于父对象的（ 10， 10）点的位置。

由于对象的位置和变换是相对于父对象来说的，因此虽然父对象的变换隐式地变换了子对象，子对象的坐标系不会因父对象坐标系改变而改变。在上面的例子中，即使父对象经过了旋转和缩放，子对象的（ 0， 10）点依然相对于父对象是（ 10， 10）点。不过相对于场景来说，子对象将随着父对象进行变换和 偏移 。如果父对象缩放了（ 2x， 2x），那么子对象在场景坐标系下将会位于（ 20， 0）的位置，同时其（ 10， 0）点将会对应于场景中的（ 40， 0）点。

不管对象自身或者父对象进行了什么样的变化， QGraphicsItem 的函数一般总是表示在对象坐标系下的位置，其操作也作用于对象坐标系内。比如，一个对象的包围盒（ QGraphicsItem::boundingRect ())总是在对象坐标系下给出的。但是 QGraphicsItem::pos 是例外之一，该函数表示其在父对象中的位置 。（致谢：此处由 tnt_vampire 指出并修正 ，见注释）

场景坐标系

场景表示了其中所有对象的基础坐标系。场景坐标系描述了每一个顶层对象的位置，同时也是所有从视图传递到场景的事件的变化基础。场景中的每一个对象都有其在场景中的位置和包围盒（ QGraphicsItem::scenePos (), QGraphicsItem::sceneBoundingRect

())，另外，也有其自身的位置和包围盒。场景位置描述了对象在场景坐标系下的位置，场景包围盒则提供给 QGraphicsScene 来决定场景中的哪一块区域已经被改变了。场景中的变化通过 QGraphicsScene::changed ()信号发出，参数是场景坐标系下的矩形列表。

视图坐标系

视图坐标系是 widget 的坐标系。视图坐标系下的每个单位对应于一个像素。对于该坐标系来说比较特殊的一点是，它是相对于 widget 或者 viewport 的坐标系，不会受到被显示的场景的影响。 QGraphicsView 的 viewport 左上角坐标总是（ 0， 0 ），右下角坐标总是（ viewport 宽度， viewport 高度）。所有的鼠标事件和拖拽事件都在视图坐标系下接收，你需要将这些坐标映射到场景中后才能与场景中的图形对象进行交互。

坐标映射

通常当我们处理场景中的对象的时候，坐标变换将会非常有用，我们可以把坐标或者任意形状从场景变换到对象坐标系下，从一个对象坐标系变换到另一个对象坐标系，或者从视图变换到场景。例如当你用鼠标点击了 **QGraphicsView** 的 **viewport**，你可以向场景管理器查询当前鼠标下方的对象（调用 **QGraphicsView::mapToScene ()**变换坐标，然后通过 **QGraphicsScene::itemAt ()**查询对象）。如果你想知道一个对象处于 **viewport** 中的位置，你可以调用对象的函数 **QGraphicsItem::mapToScene ()** ，然后在调用视图的函数 **QGraphicsView::mapFromScene ()** 。最后，如果你想查找位于一个椭圆区域内的对象，你可以把一个 **QPainterPath** 对象传递给 **mapToScene()**然后将变换之后的 **path** 传递给 **QGraphicsScene::items ()**。

你可以通过调用 **QGraphicsItem::mapToScene ()** 将坐标或者任意形状 映射到对象的场景中去，同样也可以通过调用 **QGraphicsItem::mapFromScene ()**将其映射回来。你也可以调用 **QGraphicsItem::mapToParent ()**将对象映射到其父对象的坐标系下，同样也可以通过调用 **QGraphicsItem::mapFromParent ()**将其映射回来。甚至可以用过调用 **QGraphicsItem::mapToItem ()** 和 **QGraphicsItem::mapFromItem ()**在不同的对象的坐标系之间进行映射。所有的映射函数均支持点、举行、多边形和路径。

在视图和场景之间也存在着同样的映射函数： **QGraphicsView::mapFromScene ()** 和 **QGraphicsView::mapToScene ()**。要从视图映射到对象，第一步是映射到场景，然后才能从场景映射到对象坐标系下。

主要特点

缩放和旋转

和 QPainter 一样 QgraphicsView 也可以通过 QGraphicsView::setMatrix ()支持仿射变换。通过这种将变换矩阵应用到视图上的方式，你能很容易的添加对缩放和旋转等操作的支持。

这里是一个如何通过 QGraphicsView 子类来实现旋转和缩放操作的例子：

```
class View : public QGraphicsView
 {
 Q_OBJECT
   ...
 public slots:
   void zoomIn() { scale(1.2, 1.2); }
   void zoomOut() { scale(1 / 1.2, 1 / 1.2); }
   void rotateLeft() { rotate(-10); }
   void rotateRight() { rotate(10); }
   ...
 };
```

槽（ Slots ）可以关联到启用了“ autoRepeat ”属性的 QToolButtons。在变换视图的过程中， QgraphicsView 始终保持与视图中心对齐。参阅例程 Elastic Nodes 了解如何实现这种基本的缩放操作。

打印

图形视图通过其绘制函数 QGraphicsScene::render () 和 QGraphicsView::render ()提供了非常简单的打印功能。这两个函数提供了相同的 API：你可以让场景或者视图将全部或者部分的内容会知道任何 paint device（注： QImage、 QPrinter、 QWidget 均属于 paint device ）上，只需要将将 QPainter 传给绘制函数即可。下面的例子展示了如何利用 QPrinter 将整个场景绘制到一页上：

```
QGraphicsScene scene;
scene.addRect(QRectF(0, 0, 100, 200), QPen(Qt::black), QBrush(Qt::green));

QPrinter printer;
if (QPrintDialog(&printer).exec() == QDialog::Accepted) {
  QPainter painter(&printer);
  painter.setRenderHint(QPainter::Antialiasing);
  scene.render(&painter);
}
```

场景和视图绘制函数的区别在于，前者操作的是场景坐标，后者操作的则是视图坐标。 QGraphicsScene::render 多用于打印未经变换的整个场景，比如将几何数据图表打印出来，或者打印文本文档。 QGraphicsView::render 则比较适合用于抓取屏幕截图，其缺省行为是将 viewport 中确切的内容绘制到所提供的 painter 上。

```
QGraphicsScene scene;
```

```
scene.addRect(QRectF(0, 0, 100, 200), QPen(Qt::black), QBrush(Qt::green));

QPixmap pixmap;
QPainter painter(&pixmap);
painter.setRenderHint(QPainter::Antialiasing);
scene.render(&painter);
painter.end();

pixmap.save("scene.png");
```

当源区域和目标区域不匹配的时候，源区域内容将会被缩放到适合目标区域。通过传递 **Qt::AspectRatioMode** 参数给绘制函数，你可以在需要缩放的时候选择保持或者忽略比例进行缩放。

拖放

由于 **QGraphicsView** 直接从 **QWidget** 继承，因此 **QGraphicsView** 也提供了和 **QWidget** 一样的拖放功能。另外，为了方便起见，图形视图框架为场景中的每个对象提供了拖放支持。当 **view** 对象接收到一个拖拽消息的时候，它将其转换为拖放事件 **QGraphicsSceneDragDropEvent**，然后将其转发给场景管理器。场景管理器则会接管该事件，并将其发送给鼠标下面第一个接受拖拽消息的对象。

要拖拽一个对象，只需要创建一个 **QDrag** 对象，将指针传给开始拖拽的 **widget**。对象可以同时被多个视图观察，但是只有一个视图可以进行拖拽。在大多数情况下，拖拽都从鼠标按下或者移动开始，因此在 **mousePressEvent()**或者 **mouseMoveEvent()**事件中，你可以从事件中拿到原始 **widget** 的指针，例如：

```
void CustomItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
   QMimeData *data = new QMimeData;
   data->setColor(Qt::green);

   QDrag *drag = new QDrag(event->widget());
   drag->setMimeData(data);
   drag->start();
}
```

如果你想拦截发送给场景的拖放消息，只需要实现 QGraphicsScene::dragEnterEvent ()事件，选择你需要处理的事件，然后在 QGraphicsItem:: dragEnterEvent 中进行相应处理即可。你可以到 QGraphicsScene 的文章中查看更多关于拖放的内容。对象可以通过调用 QGraphicsItem::setAcceptDrops 来允许或者禁止对拖放的支持。可以通过实现 QGraphicsItem::dragEnterEvent (), QGraphicsItem::dragMoveEvent (), QGraphicsItem::dragLeaveEvent (), 和 QGraphicsItem::dropEvent () 这几个事件来处理拖动。

鼠标指针和 **tooltip**

和 **QWidget** 一样， **QGraphicsItem** 也支持设置鼠标和 **tooltip**。当鼠标进入对象区域（由 **QGraphicsItem::contains** 检测）时，通过 **QGraphicsView** 来设置鼠标和 **tooltip**。

你也调用 **QGraphicsView::setCursor** 设置视图的确实鼠标指针。在 **Drag and Drop Robot** 这里例子中有实现 **tooltip** 和鼠标指针的代码。

动画

图形视图在几个层面上提供了对动画的支持。你可以用 **Animation Framework** 轻松地设置动画：只需要让你的对象从 **QGraphicsObject** 继承，然后将 **QPropertyAnimation** 绑定到上面。 **QPropertyAnimation** 允许你通过对对象的属性进行操作来实现动画效果。

另外一个选择是创建一个从 **QObject** 和 **QGraphicsItem** 继承的对象。该对象设置自己的时钟，然后在 **QObject::timerEvent** 中控制动画。

第三个选择仅限于与 **Qt3** 中的 **QCanvas** 兼容。调用 **QGraphicsScene::advance** 从而使 **QGraphicsItem::advance** 得以执行。

OpenGL 绘制

要使用 OpenGL 进行绘制，只需简单地创建一个新的 QGLWidget 对象，并调用 QGraphicsView::setViewport () 将其作为视口设置为视图即可。如果你希望使用 OpenGL 的反锯齿，则需要 OpenGL 支持采样缓存（请参见 QGLFormat::sampleBuffers () ）。

```
QGraphicsView view(&scene);
view.setViewport(new QGLWidget(QGLFormat(QGL::SampleBuffers)));
```

对象编组

通过将一个对象设置为另一个对象的子对象，你就可以得到对象编组最重要的功能：对象会一起移动，所有变换都会从父对象传播到子对象中。

另外， **QGraphicsItemGroup** 是一个特殊的对象，它提供了对子对象事件的支持，同时还提供了用于添加和删除子对象的接口。将一个对象添加到 **QGraphicsItemGroup** 将保持对象原始的位置和坐标变换，不过重新设置对象的父对象则会导致对象重新定位到相对于父对象的位置。为了方便起见，你可以 调用 **QGraphicsScene::createItemGroup ()**来 创建 **QGraphicsItemGroup** 对象。

Widgets 和布局

Qt4.4 引入了对几何体和对布局敏感的对象的支持，具体实现在 QGraphicsWidget 中。这一特殊的基类对象和 QWidget 类似，但是不想 QWidget，它没有从 QPaintDevice 继承，而是从 QGraphicsItem。这样就允许你完全实现能够处理事件、信号与槽、大小调整策略的 widget，同时你还可以通过 QGraphicsLinearLayout 和 QGraphicsGridLayout 来管理 widget 的几何元素。

QGraphicsWidget

QGraphicsWidget 建立在 QGraphicsItem 之上，具有 QGraphicsItem 的所有功能，保持了较小的资源占用，同时提供了两者的优势：来自 QWidget 的额外的功能，比如样式、字体、调色板、布局、几何形状，来自 QGraphicsItem 的分辨率独立性和坐标变换的支持。因为几何视图使用真实的坐标而不是整数，因此 QGraphicsWidget 的几何形状函数可以同时操作 QRectF 和 QPointF。同时也能应用到边框的大小、边距和间距上，例如在 QGraphicsWidget 上将对象的边距设置为（ 0.5， 0.5， 0.5， 0.5）是非常常见的。你也可以创建子 widget 对象，甚至是“顶级”窗口。在某些情况下，你甚至可以将图形视图用于高级多文档界面的应用程序。

QGraphicsWidget 支持部分 QWidget 属性，包括窗口标志位和属性，但是并非全部支持。你需要通过 QGraphicsWidget 文档获取哪些支持以及哪些不支持的完整列表。例如，你可以在创建 QGraphicsWidget 时赋予 Qt::Window 标志，从而得到封装的窗口，但是图形视图目标并不支持 Qt::Sheet 和 Qt::Drawer 标志，这两者在 Mac Os X 上非常常见。

QGraphicsWidget 的功能将会逐步丰富起来，这一点由社区的反馈决定。

QGraphicsLayout

QGraphicsLayout 是第二代布局框架的内容之一，专门为 QGraphicsWidget 设计。其 API 和 QLayout 非常相似。你可以在 QGraphicsLinearLayout 或者 QGraphicsGridLayout 中对 widget 或者子布局进行管理，也可以通过派生 QgraphicsLayout 实现你自己的布局类，你还可以通过派生 QGraphicsLayoutItem 来实现你自己的 QGraphicsItem 对象的适配器从而将其加入到布局中。

嵌入式 widget 的支持

图形视图对将任何 **widget** 嵌入到场景中提供无缝的支持。你可以嵌入简单的 **widget**，比如 **QLineEdit** 或者 **QPushButton**，也可以是复杂的 **widget**，比如 **QTabWidget**，甚至是完整的主窗口。要将 **widget** 嵌入场景中，只需要简单地调用 **QGraphicsScene::addWidget** 或者创建一个 **QGraphicsProxyWidget** 对象并将 **widget** 手工的嵌入其中。通过 **QGraphicsProxyWidget** 图形视图能够完全继承客户端 **widget** 对象的功能，包括其鼠标指针、 **tooltip**、鼠标事件、平板电脑相关事件、键盘事件、子窗口、动画、弹出（比如 **QComboBox** 或者 **QCompleter**），以及 **widget** 输入焦点和激活状态。**QGraphicsProxyWidget** 甚至集成了嵌入式 **widget** 的 **tab** 切换顺序，这样你就可以通过 **tab** 键让焦点进入或者移出嵌入式 **widget**。你甚至可以嵌入一个新的 **QGraphicsView** 到你的场景中，从而提供复杂的嵌套的视图。

性能

精度浮点数指令

为了精确和快速的将坐标变换和特效应用到图形对象上，图形视图在编译的时候默认用户的硬件能够为浮点指令提供合理的性能。

很多工作站和桌面电脑都配备了适当的硬件来加速这种类型的计算，但是一些嵌入式设备可能仅仅提供了处理数学运算的库，或者需要用软件来模拟浮点指令。

这样，在某些设备上，某些类型的特效可能要比预期的慢。有可能可以在其他方面进行优化来弥补性能上的损失，比如说用 OpenGL 来绘制场景。不过，如果优化本身是依赖于浮点计算硬件的话，可能都会带来性能上的损失。