Qt Essentials - Graphics View 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.

Module: Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Creating Custom Items

Objectives

- Using QGraphicsView-related classes
- Coordinate Schemes, Transformations
- Extending items
  - Event handling
  - Painting
  - Boundaries

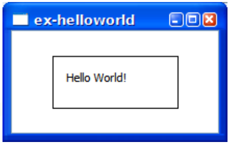
- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Creating Custom Items

GraphicsView Framework

- Provides:
  - A surface for managing interactive 2D graphical items
  - A view widget for visualizing the items
- Uses MVC paradigm
- Resolution Independent
- Animation Support
- Fast item discovery, hit tests, collision detection
  - Using Binary Space Paritioning (BSP) tree indexes
- Can manage large numbers of items (tens of thousands)
- Supports zooming, printing and rendering

Hello World

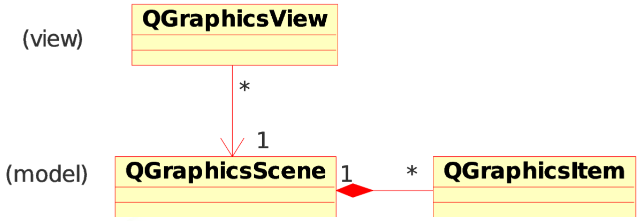
```
#include <QtWidgets>
int main(int argc, char **argv) {
  QApplication app(argc, argv);
  QGraphicsView view;
  QGraphicsScene *scene = new QGraphicsScene(&view);
  view.setScene(scene);
  QGraphicsRectItem *rect =
      new QGraphicsRectItem(-10, -10, 120, 50);
  scene->addItem(rect);
  QGraphicsTextItem *text = scene->addText("Hello World!");
  view.show();
  return app.exec();
}
```



Demo graphicsview/ex-helloworld

UML relationship

- `QGraphicsScene` is:
  - a "model" for `QGraphicsView`
  - a "container" for `QGraphicsItems`



- Container for Graphic Items
  - Items can exist in only one scene at a time
- Propagates events to items
  - Manages Collision Detection
  - Supports fast item indexing
  - Manages item selection and focus
- Renders scene onto view
  - z-order determines which items show up in front of others

QGraphicsScene important methods

- `addItem()`
  - Add an item to the scene
    - (remove from previous scene if necessary)
  - Also `addEllipse()`, `addPolygon()`, `addText()`, etc

```
QGraphicsEllipseItem *ellipse =
    scene->addEllipse(-10, -10, 120, 50);
QGraphicsTextItem *text =
    scene->addText("Hello World!");
```

- `items()`
  - returns items intersecting a particular point or region
- `selectedItems()`
  - returns list of selected items
- `sceneRect()`
  - bounding rectangle for the entire scene

QGraphicsView

- Scrollable widget viewport onto the scene
  - Zooming, rotation, and other transformations
  - Translates input events (from the View) into `QGraphicsSceneEvents`
  - Maps coordinates between scene and viewport
  - Provides "level of detail" information to items
  - Supports OpenGL

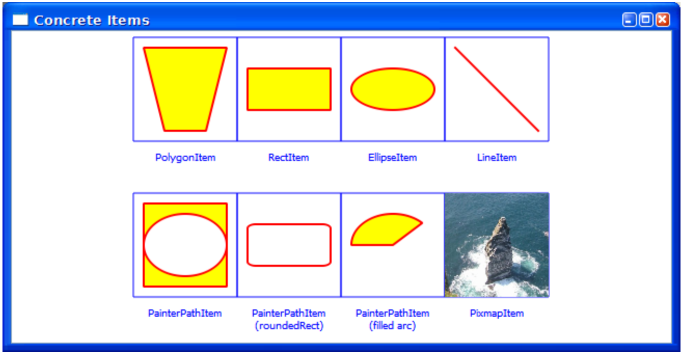
QGraphicsView important methods

- `setScene()`
  - sets the QGraphicsScene to use
- `setRenderHints()`
  - antialiasing, smooth pixmap transformations, etc
- `centerOn()`
  - takes a QPoint or a QGraphicsItem as argument
  - ensures point/item is centered in View
- `mapFromScene()`, `mapToScene()`
  - map to/from scene coordinates
- `scale()`, `rotate()`, `translate()`, `matrix()`
  - transformations

QGraphicsItem

- Abstract base class: basic canvas element
  - Supports parent/child hierarchy
- Easy to extend or customize concrete items:
  - `QGraphicsRectItem`, `QGraphicsPolygonItem`, `QGraphicsPixmapItem`, `QGraphicsTextItem`, etc.
  - SVG Drawings, other widgets
- Items can be transformed:
  - move, scale, rotate
  - using local coordinate systems
- Supports Drag and Drop similar to QWidget

Concrete QGraphicsItem Types



Demo graphicsview/ex-concreteitems

QGraphicsItem important methods

- `pos()`
  - get the item's position in scene
- `moveBy()`
  - moves an item relative to its own position.
- `zValue()`
  - get a Z order for item in scene
- `show()`, `hide()` - set visibility
- `setEnabled(bool)` - disabled items can not take focus or receive events
- `setFocus(Qt::FocusReason)` - sets input focus.
- `setSelected(bool)`
  - select/deselect an item
  - typically called from `QGraphicsScene::setSelectionArea()`

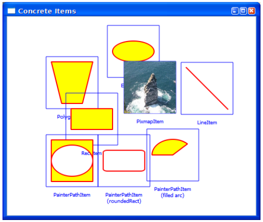
Select, Focus, Move

- `QGraphicsItem::setFlags()`
  - Determines which operations are supported on an item
- `QGraphicsItemFlags`
  - `QGraphicsItem::ItemIsMovable`
  - `QGraphicsItem::ItemIsSelectable`
  - `QGraphicsItem::ItemIsFocusable`

```
item->setFlags(
  QGraphicsItem::ItemIsMovable|QGraphicsItem::ItemIsSelectable);
```

Groups of Items

- Any `QGraphicsItem` can have children
- `QGraphicsItemGroup` is an invisible item for grouping child items
- To group child items in a box with an outline (for example), use a `QGraphicsRectItem`
- Try dragging boxes in demo:



Demo graphicsview/ex-concreteitems

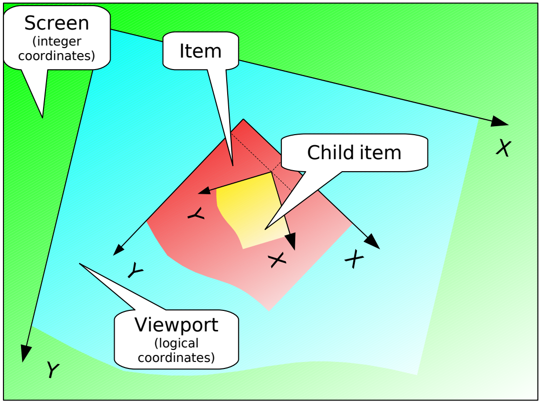
Parents and Children

- Parent propagates values to child items:
  - `setEnabled()`
  - `setFlags()`
  - `setPos()`
  - `setOpacity()`
  - etc...
- Enables composition of items.

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Creating Custom Items

- Each View and Item has its own local coordinate system

(0,0)



- Coordinates are local to an item
  - Logical coordinates, not pixels
  - Floating point, not integer
  - Without transformations, 1 logical coordinate = 1 pixel.
- Items inherit position and transform from parent
- zValue is relative to parent
- Item transformation does not affect its local coordinate system
- Items are painted recursively
  - From parent to children
  - in increasing zValue order

QTransform

- Coordinate systems can be transformed using `QTransform`
- `QTransform` is a 3x3 matrix describing a linear transformation from (x,y) to (xt, yt)

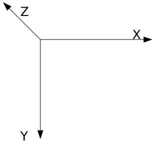
m11	m12	m13
m21	m22	m23
m31	m32	m33

```
xt = m11*x + m21*y + m31
yt = m22*y + m12*x + m32
if projected:
    wt = m13*x + m23*y + m33
    xt /= wt
    yt /= wt
```

- m_{13} and m_{23}
  - Control perspective transformations
- See Affine Transformations Wikipedia Article

Common Transformations

- Commonly-used convenience functions:
  - `scale()`
  - `rotate()`
  - `shear()`
  - `translate()`
- Saves you the trouble of defining transformation matrices
- `rotate()` takes optional 2nd argument: axis of rotation.
  - Z axis is "simple 2D rotation"
  - Non-Z axis rotations are "perspective" projections.



View transformations

```
t = QTransform();           // identity matrix
t.rotate(45, Qt::ZAxis); // simple rotate
t.scale(1.5, 1.5)           // scale by 150%
view->setTransform(t);    // apply transform to entire view
```

- `setTransformationAnchor()`
  - An **anchor** is a point that remains fixed before/after the transform.
  - `AnchorViewCenter`: (Default) The *center point* remains the same
  - `AnchorUnderMouse`: The *point under the mouse* remains the same
  - `NoAnchor`: Scrollbars remain unchanged.

Item Transformations

- `QGraphicsItem` supports same transform operations:
  - `setTransform()`, `transform()`
  - `rotate()`, `scale()`, `shear()`, `translate()`

An item's effective transformation:

The product of its own and all its ancestors' transformations

TIP: When managing the transformation of items, store the desired rotation, scaling etc. in member variables and build a `QTransform` from the identity transformation when they change. Don't try to deduce values from the current transformation and/or try to use it as the base for further changes.

Zooming

- Zooming is done with `view->scale()`

```
void MyView::zoom(double factor)
{
    double width =
        matrix().mapRect(QRectF(0, 0, 1, 1)).width();
    width *= factor;
    if ((width < 0.05) || (width > 10)) return;

    scale(factor, factor);
}
```

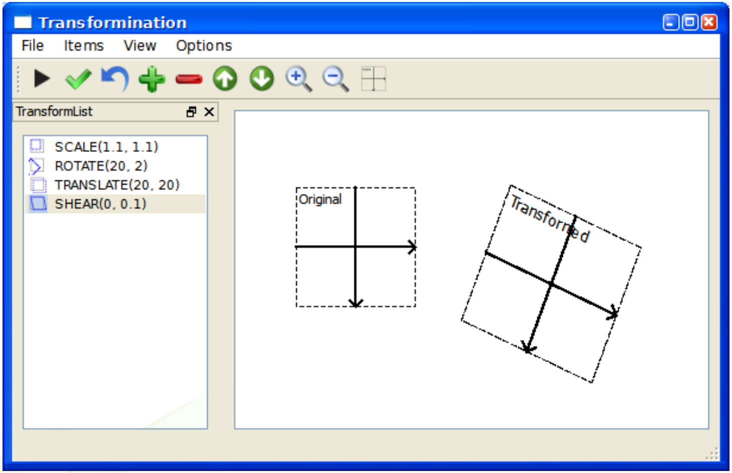
Mapping between Coordinate Systems

- Mapping methods are overloaded for `QPolygonF`, `QPainterPath` etc
  - `mapFromScene(const QPointF&)`:
    - Maps a point from scene coordinates to item coordinates. Inverse: `mapToScene(const QPointF&)`
  - `mapFromItem(const QGraphicsItem*, const QPointF&)`
    - Maps a point from another item's coordinate system to this item's. Inverse: `mapToItem(const QGraphicsItem*, const QPointF&)`.
  - Special case: `mapFromParent(const QPointF&)`.

Ignoring Transformations

- Sometimes we don't want particular items to be transformed before display.
- View transformation can be disabled for individual items.
- Used for text labels in a graph that should not change size when the graph is zoomed.

```
item->setFlag( QGraphicsItem::ItemIgnoresTransformations);
```



SCALE(1.1, 1.1)
ROTATE(20, 2)
TRANSLATE(20, 20)
SHEAR(0, 0.1)

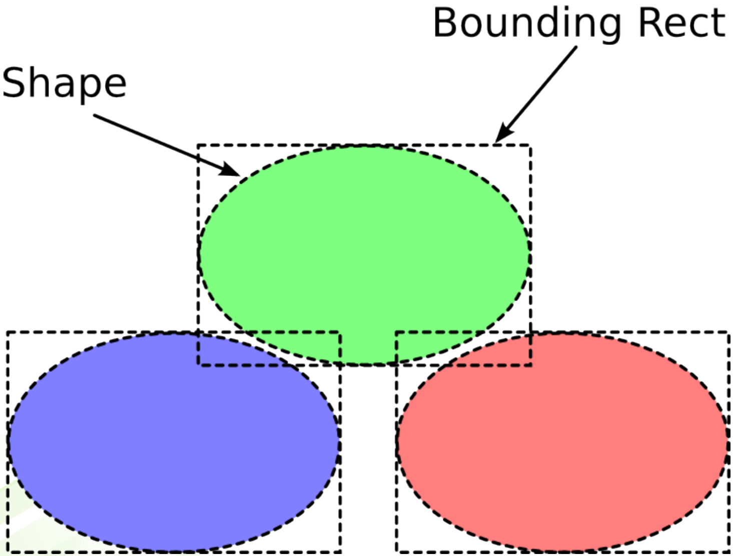
Demo graphicsview/ex-transformination

Extending QGraphicsItem

`QGraphicsItem` pure virtual methods (required overrides):

- `void paint()`
  - Paints contents of item in local coordinates
- `QRectF boundingRect()`
  - Returns outer bounds of item as a rectangle
  - Called by `QGraphicsView` to determine what regions need to be redrawn
- `QPainterPath shape()` - shape of item
  - Used by `contains()` and `collidesWithPath()` for collision detection
  - Defaults to `boundingRect()` if not implemented

Boundaries

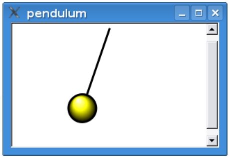


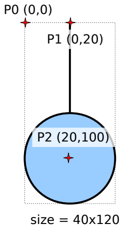
Painting Items

- Item is in complete control of drawing itself
- Use standard `QPainter` drawing methods
  - `QPen`, `QBrush`, pixmaps, gradients, text, etc.
- No background to draw
- Dynamic boundary and arbitrary shape
  - Polygon, curved, non-contiguous, etc.

Custom Item example

```
class PendulumItem : public QGraphicsItem {
public:
    QRectF boundingRect() const;
    void paint(QPainter* painter,
               const QStyleOptionGraphicsItem* option,
               QWidget* widget);
};
```





paint() and boundingRect()

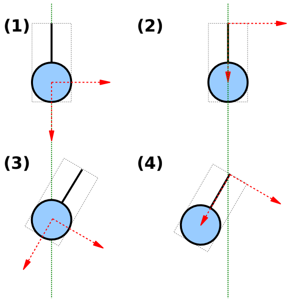
- `boundingRect()` must take the pen width into consideration

```
QRectF PendulumItem::boundingRect() const {
    return QRectF(-20.0 - PENWIDTH/2.0, -PENWIDTH/2.0,
                  40.0 + PENWIDTH, 140.0 + PENWIDTH );
}

void PendulumItem::paint( QPainter* painter,
    const QStyleOptionGraphicsItem*, QWidget*) {
    painter->setPen( QPen( Qt::black, PENWIDTH ) );
    painter->drawLine(0,0,0,100);
    QRadialGradient g( 0, 120, 20, -10, 110 );
    g.setColorAt( 0.0, Qt::white );
    g.setColorAt( 0.5, Qt::yellow );
    g.setColorAt( 1.0, Qt::black );
    painter->setBrush(g);
    painter->drawEllipse(-20, 100, 40, 40);
}
```

Choosing a `boundingRect()`

- `boundingRect()`
  - Influences drawing code
  - Influences "origin" of item transforms
- i.e. for Pendulum that swings:
  - Good origin is non-weighted end of line
  - Can rotate around (0,0) without translation



QGraphicsItemGroup

- Easier approach to making a Pendulum:
  - Extend `QGraphicsItemGroup`
  - Use other concrete items as elements, add as children
  - No need to override `paint()` or `shape()`

```
PendulumItem::PendulumItem(QGraphicsItem* parent)
  : QGraphicsItemGroup(parent) {
  m_line = new QGraphicsLineItem( 0,0,0,100, this);
  m_line->setPen( QPen( Qt::black, 3 ) );
  m_circle = new QGraphicsEllipseItem( -20, 100, 40, 40, this );
  m_circle->setPen( QPen(Qt::black, 3 ));
  QRadialGradient g( 0, 120, 20, -10, 110 );
  g.setColorAt( 0.0, Qt::white );
  g.setColorAt( 0.5, Qt::yellow );
  g.setColorAt( 1.0, Qt::black );
  m_circle->setBrush(g);
}
```

Demo graphicsview/ex-pendulum

Event Handling

- `QGraphicsItem::sceneEvent(QEvent*)`
  - Receives all events for an item
  - Similar to `QWidget::event()`
- Specific typed event handlers:
  - `keyPressEvent(QKeyEvent*)`
  - `mouseMoveEvent(QGraphicsSceneMouseEvent*)`
  - `wheelEvent(QGraphicsSceneWheelEvent*)`
  - `mousePressEvent(QGraphicsSceneMouseEvent*)`
  - `contextMenuEvent(QGraphicsSceneContextMenuEvent*)`
  - `dragEnterEvent(QGraphicsSceneDragDropEvent*)`
  - `focusInEvent(QFocusEvent*)`
  - `hoverEnterEvent(QGraphicsSceneHoverEvent*)`

When overriding mouse event handlers:

Make sure to call base-class versions, too. Without this, the item select, focus, move behavior will not work as expected.

Event Handler examples

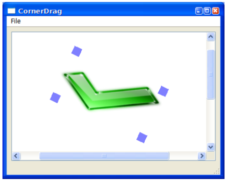
```
void MyView::wheelEvent(QWheelEvent *event) {
    double factor =
        1.0 + (0.2 * qAbs(event->delta()) / 120.0);
    if (event->delta() > 0) zoom(factor);
    else                    zoom(1.0/factor);
}
void MyView::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
      case Qt::Key_Plus:
          zoom(1.2);
          break;
      case Qt::Key_Minus:
          zoom(1.0/1.2);
          break;
      default:
          QGraphicsView::keyPressEvent(event);
    }
}
```

Collision Detection

- Determines when items' shapes intersect
- Two methods for collision detection:
  - `collidesWithItem(QGraphicsItem* other)`
  - `collidingItems(Qt::ItemSelectionMode)`
- `shape()`
  - Returns `QPainterPath` used for collision detection
  - Must be overridden properly
- `items()`
  - Overloaded forms take `QRectF`, `QPolygonF`, `QPainterPath`
  - Return items found in rect/polygon/shape

Lab: Corner drag button

- Define a `QGraphicsItem` which can display an image, and has at least 1 child item, that is a "corner drag" button, permitting the user to click and drag the button, to resize or rotate the image.
- Start with the handout provided in `graphicsview/lab-cornerdrag`
- Further details
  are in the `readme.txt` in the same directory.


CornerDrag
File