Qt Essentials - Graphics View 2 Module

Qt Essentials - Training Course

Produced by Nokia, Qt Development Frameworks

Material based on Qt 4.7, created on December 15, 2010

http://qt.nokia.com

Module: Graphics View 2

- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Module: Graphics View 2

- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Widgets in a Scene

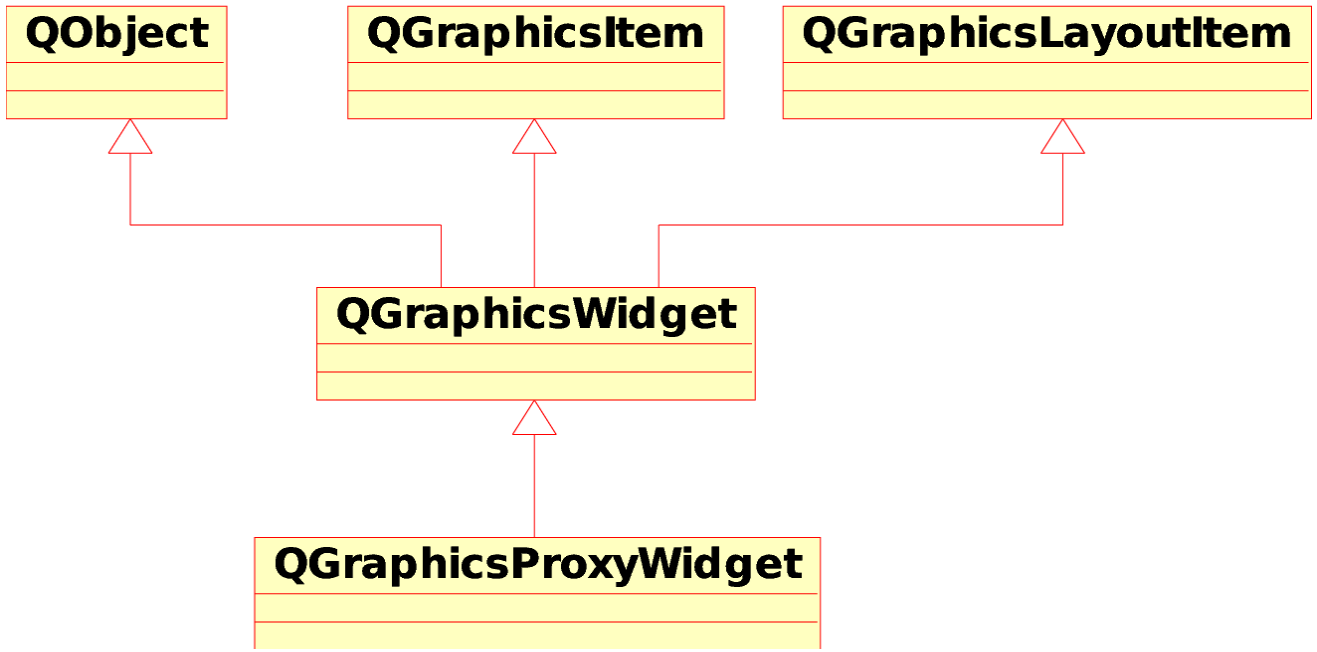
Demo $QTDIR/examples/graphicsview/padnavigator

Items are not widgets

- `QGraphicsItem`:
  - Lightweight compared to `QWidget`
  - No signals/slots/properties
  - Scenes can easily contain thousands of Items
  - Uses different `QEvent` sub-hierarchy (derived from `QGraphicsSceneEvent`)
  - Supports transformations directly
- `QWidget`:
  - Derived from `QObject` (less light-weight)
  - supports signals, slots, properties, etc
  - can be embedded in a `QGraphicsScene` with a `QGraphicsProxyWidget`

QGraphicsWidget

- Advanced functionality graphics item
- Provides signals/slots, layouts, geometry, palette, etc.
- *Not* a QWidget!
- Base class for `QGraphicsProxyWidget`



QGraphicsProxyWidget

- `QGraphicsItem` that can embed a `QWidget` in a `QGraphicsScene`
- Handles complex widgets like `QFileDialog`
- Takes *ownership* of related widget
  - Synchronizes states/properties:
    - `visible`, `enabled`, `geometry`, `style`, `palette`, `font`, `cursor`, `sizeHint`, `windowTitle`, etc
    - Proxies events between Widget and GraphicsView
  - If either (widget or proxy) is deleted, the other is also!
- Widget must not already have a parent
  - Only top-level widgets can be added to a scene

Embedded Widget Example

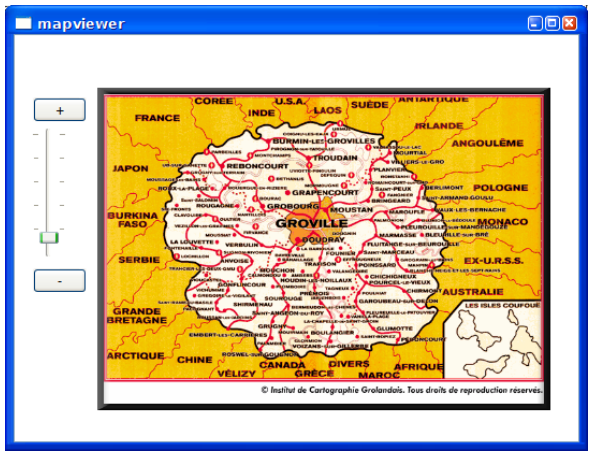
```
#include <QtGui>
int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QCalendarWidget *calendar = new QCalendarWidget;

    QGraphicsScene scene;
    QGraphicsProxyWidget *proxy = scene.addWidget(calendar);

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

    return app.exec();
}
```

QGraphicsLayout

- For layout of `QGraphicsLayoutItem` (+derived) classes in `QGraphicsView`
- Concrete classes:
  - `QGraphicsLinearLayout`: equivalent to `QBoxLayout`, arranges items horizontally or vertically
  - `QGraphicsGridLayout`: equivalent to `QGridLayout`, arranges items in a grid
- `QGraphicsWidget::setLayout()` - set layout for child items of this `QGraphicsWidget`

Lab: Widgets in a Scene

- Starting with the `graphicsview/lab-mapviewer` handout, add zooming controls.
- Suggested widgets:
  - `QPushButtons` for +/-
  - `QSlider` for selecting zoom level directly.
- Use `QGraphicsLayout` to lay out the widgets
- Make the mouse work like a "hand-grab" tool on drag, so we can see different zoomed areas.



Module: Graphics View 2

- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Drag and Drop

- Items can be:
  - Dragged
  - Dropped onto other items
  - Dropped onto scenes
    - for handling empty drop areas

Start Drag

Starting an item drag is similar to dragging from a QWidget.

- Override event handlers:
  - `mousePressEvent()`
  - `mouseMoveEvent()`
- In `mouseMoveEvent()`, decide if drag started? if so:
  - Create a `QDrag` instance
  - Attach a `QMimeData` to it
    - See section on Drag and Drop for `QMimeData info`
  - Call `QDrag::exec()`
    - Function returns when user drops
    - Does not block event loop

Drop on a scene

- Override `QGraphicsScene::dropEvent()`
  - To accept drop:
    - `acceptProposedAction()`
    - `setDropAction(Qt::DropAction); accept();`
- Override `QGraphicsScene::dragMoveEvent()`
- Optional overrides:
  - `dragEnterEvent()`, `dragLeaveEvent()`

startDrag example

```
void startDrag( Qt::DropActions supportedActions ) {
    QList<QListWidgetItem *> items = selectedItems();
    if ( items.size()>0 ) {
        QDrag* drag = new QDrag( this );
        QMimeData *mimeData = new QMimeData; [...]
        QGraphicsItem* gitem =
               DiagramItem::createItem( item->toolType() );
        mimeData->setData( "application/x-qgraphicsitem-ptr",
                           QByteArray::number( ( qulonglong )gitem ) );
        drag->setMimeData( mimeData );
        QPixmap pix = item->icon().pixmap( 111,111 );
        drag->setPixmap( pix );
        drag->setHotSpot( pix.rect().center() );
        if ( drag->exec(supportedActions) == Qt::IgnoreAction ) {
            delete gitem; // drag cancelled, must delete item
        }
```

Demo graphicsview/ex-dragdrop

dropEvent() on a scene

```
void DiagramScene::dropEvent( QGraphicsSceneDragDropEvent* event ) {
    if (event->mimeData()->hasFormat("application/x-qgraphicsitem-ptr
        QGraphicsItem* item = reinterpret_cast<QGraphicsItem*>(
             event->mimeData()->data (
             "application/x-qgraphicsitem-ptr").toULongLong() );
        if ( item ) {
            addItem( item );
            item->setFlag( QGraphicsItem::ItemIsMovable );
            item->setFlag( QGraphicsItem::ItemIsSelectable );
            item->setFlag( QGraphicsItem::ItemIsFocusable );
            item->setPos( event->scenePos() );
            event->acceptProposedAction();
        }
    } else
        /* Call baseclass to allow per-item dropEvent */

        QGraphicsScene::dropEvent( event );
}
```

emo graphicsview/ex-dragdrop

Drop on an item

- To drop into an item:
  - Override `dragEnterEvent()`
  - Optional override: `dragMoveEvent()` (if the item can only accept drops in some parts of its area)

```
void DiagramItem::dragEnterEvent(QGraphicsSceneDragDropEvent* e){
    if ( e->mimeData()->hasColor() )
        e->acceptProposedAction();
}

void DiagramScene::dragMoveEvent(QGraphicsSceneDragDropEvent* e){
     if (e->mimeData()->hasFormat(
     "application/x-qgraphicsitem-ptr"))
         e->acceptProposedAction();
     else
         QGraphicsScene::dragMoveEvent(e);
}
```

Demo graphicsview/ex-dragdrop

Module: Graphics View 2

- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Graphics Effects

Effects can be applied to graphics items:

- Base class for effects is `QGraphicsEffect`.
- Standard effects include blur, colorize, opacity and drop shadow.
- Effects are set on items.
  - `QGraphicsItem::setGraphicsEffect()`
- Effects cannot be shared or layered.
- Custom effects can be written.

Using a Graphics Effect

- Applying a blur effect to a pixmap.

```
QPixmap pixmap(":/images/qt-banner.png");

QGraphicsItem *blurItem = scene->addPixmap(pixmap);

QGraphicsBlurEffect *blurEffect = new QGraphicsBlurEffect();
blurItem->setGraphicsEffect(blurEffect);
blurEffect->setBlurRadius(5);
```

- An effect is owned by the item that uses it.
- Updating an effect causes the item to be updated.

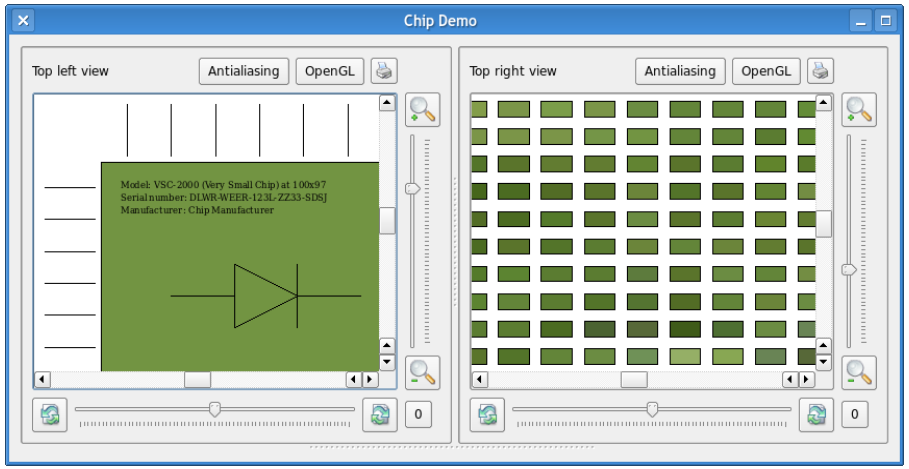
Module: Graphics View 2

- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Level of Detail

- Don’t draw what you can’t see!
- `QStyleOptionGraphicsItem` passed to `paint()`
  - Contains `palette`, `state`, `matrix` members
  - `qreal levelOfDetailFromTransform(QTransform T)` method
- "levelOfDetail" is max width/height of the unity rectangle needed to draw this shape onto a QPainter with a QTransform of T.
- use `worldTransform()` of `painter` for current transform.
  - Zoomed out: `levelOfDetail` < 1.0
  - Zoomed in: `levelOfDetail` > 1.0

Level of detail: Chip demo



Demo $QTDIR/demos/chip

Level of detail: Chip demo 2

```
void Chip::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *)
{
    const qreal lod = option->levelOfDetailFromTransform(
                      painter->worldTransform());
    [ ... ]
    if (lod >= 2) {
        QFont font("Times", 10);
        font.setStyleStrategy(QFont::ForceOutline);
        painter->save();
        painter->setFont(font);
        painter->scale(0.1, 0.1);
        painter->drawText(170, 180, QString("Model: VSC-2000 ..."
        painter->drawText(170, 220, QString("Manufacturer: ..."
        painter->restore();
    }
```

Demo $QTDIR/demos/chip

Caching tips

- Cache item painting into a pixmap
  - So `paint()` runs faster
- Cache `boundingRect()` and `shape()`
  - Avoid recomputing expensive operations that stay the same
  - Be sure to invalidate manually cached items after zooming and other transforms

```
QRectF MyItem::boundingRect() const {
    if (m_rect.isNull()) calculateBoundingRect();
    return m_rect;
}

QPainterPath MyItem::shape() const {
    if (m_shape.isEmpty()) calculateShape();
    return m_shape;
}
```

setCacheMode()

- Property of `QGraphicsView` and `QGraphicsItem`
- Allows caching of pre-rendered content in a `QPixmap`
  - Drawn on the viewport
  - Especially useful for gradient shape backgrounds
  - Invalidated whenever view is transformed.

```
QGraphicsView view;
view.setBackgroundBrush(QImage(":/images/backgroundtile.png"));
view.setCacheMode(QGraphicsView::CacheBackground);
```

Tweaking

The following methods allow you to tweak performance of view/scene/items:

- `QGraphicsView::setViewportUpdateMode()`
- `QGraphicsView::setOptimizationFlags()`
- `QGraphicsScene::setItemIndexMethod()`
- `QGraphicsScene::setBspTreeDepth()`
- `QGraphicsItem::setFlags()`
  - `ItemDoesntPropagateOpacityToChildren` and `ItemIgnoresParentOpacity` especially recommended if your items are opaque!

See API documentation for details.

Tips for better performance

- `boundingRect()` and `shape()` are called frequently so they should run fast!
  - `boundingRect()` should be as small as possible
  - `shape()` should return simplest reasonable path
- Try to avoid drawing gradients on the painter. Consider using pre-rendered backgrounds from images instead.
- It is costly to dynamically insert/remove items from the scene. Consider hiding and reusing items instead.
- Embedded widgets in a scene is costly.
- Try using a different paint engine (OpenGL, Direct3D, etc)
  - `setViewport (new QGLWidget);`
- Avoid curved and dashed lines
- Alpha blending and antialiasing are expensive