



Qt in Education

The Graphics View Canvas



NOKIA



© 2011 Nokia Corporation and its Subsidiary(-ies).

The enclosed Qt Materials are provided under the Creative Commons Attribution-Share Alike 2.5 License Agreement.



The full license text is available here:
<http://creativecommons.org/licenses/by-sa/2.5/legalcode>.

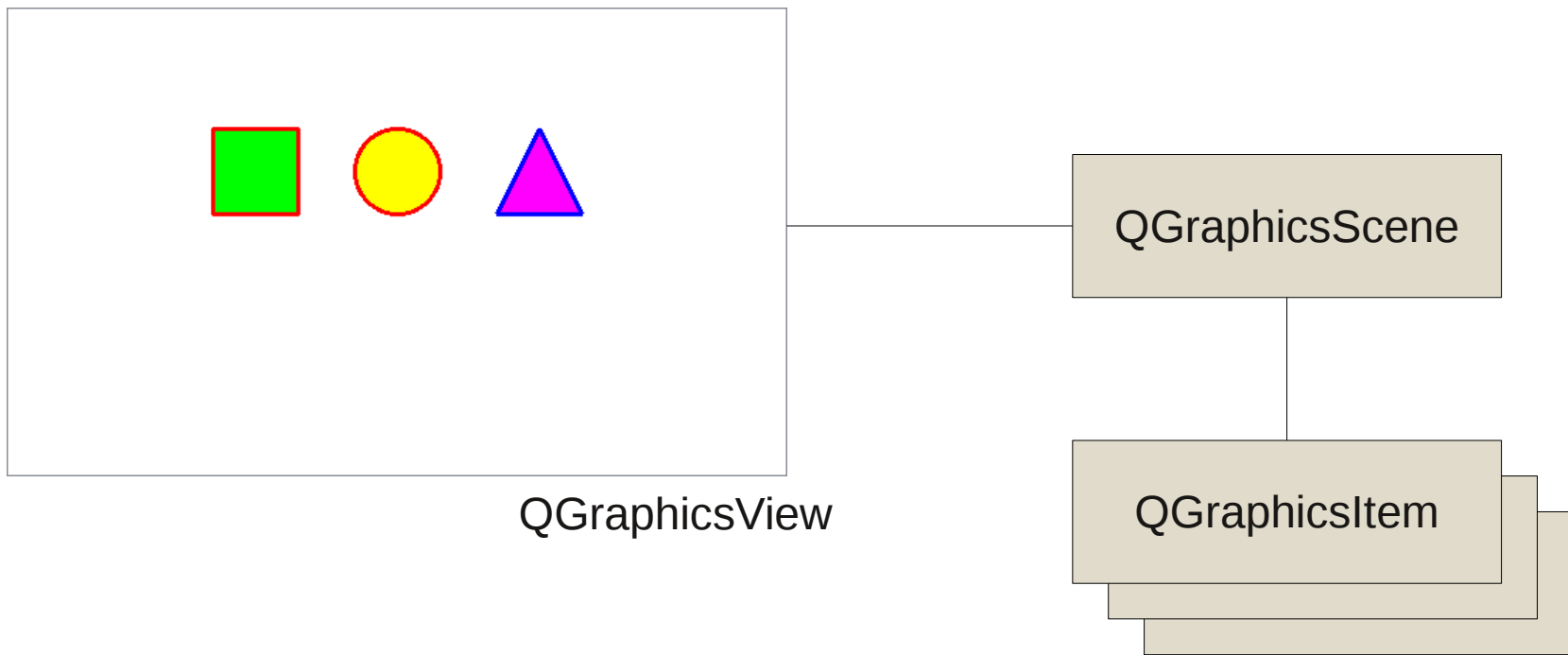
Nokia, Qt and the Nokia and Qt logos are the registered trademarks of Nokia Corporation in Finland and other countries worldwide.



The Graphics View Framework



- The Graphics View Framework is a scene based approach to graphics





Item scenes compared to widgets

| Widget | Scene / Items |
|--------------------------------------|--|
| Rectangular | Arbitrary shape |
| Non-overlapping (when in layouts) | Arbitrary position |
| Optimized for a native platform look | Optimized for animations and effects |
| Integer (pixel) coordinates | Floating point (sub-pixel) coordinates |

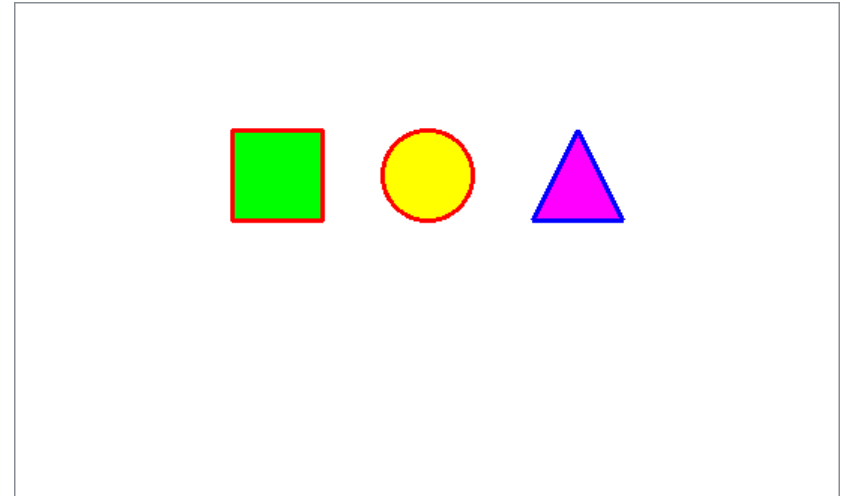


Painting Using QWidget and paintEvent

```
void Widget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);

    painter.setPen(QPen(Qt::red, 3));
    painter.setBrush(Qt::green);
    painter.drawRect(20, 20, 60, 60);

    ...
}
```



- What about supporting more items?
- What about moving items about?



Painting Using Graphics View

```
void Widget::setupScene()
{
    QGraphicsView *view = new QGraphicsView();
    QGraphicsScene *scene = new QGraphicsScene(0, 0, 300, 200, this);

    scene->addRect(20, 20, 60, 60, QPen(Qt::red, 3), QBrush(Qt::green));
    scene->addEllipse(120, 20, 60, 60, QPen(Qt::red, 3), QBrush(Qt::yellow));
    scene->addPolygon(QPolygonF() << QPointF(220, 80) << QPointF(280, 80)
        << QPointF(250, 20), QPen(Qt::blue, 3), QBrush(Qt::magenta));

    view->setScene(scene);
    view->show();
}
```





The Scene



- The QGraphicsScene class contains all items and acts as an interface between the view and the items
 - Owns all items
 - Distributes paint events
 - Distributes other events
 - Provides methods for locating items
 - itemAt – the top item at a given location
 - items – all items in a given area



Initializing a Scene

- Each scene has a sceneRect defining the extent of the scene
 - If it is not specified, it will be the largest rectangle containing (or having contained) the scene items

```
QGraphicsScene *scene = new QGraphicsScene(this);  
scene->setSceneRect(-100, -100, 201, 201);
```

The rectangle does not have to start at the origin (0, 0)



Populating a Scene

- The QGraphicsScene class makes it easy to add basic shapes to a scene

```
QGraphicsItem *item =  
    scene->addRect(20, 20, 60, 60,  
        QPen(Qt::red, 3), QBrush(Qt::green));
```

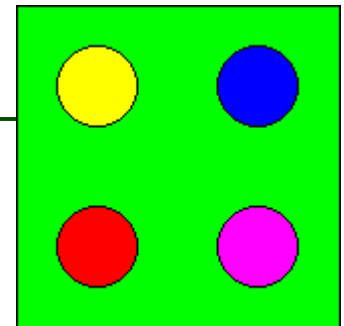
- Supports ellipses, lines, painter paths, pixmaps, polygons, rectangles and text
 - Each item type is represented by a separate class derived from QGraphicsItem



Populating a Scene

- For custom items and complex cases, it is possible to create items and then add them to a scene

```
QGraphicsRectItem *rootItem =  
    new QGraphicsRectItem(-50, -50, 101, 101);  
rootItem->setBrush(Qt::green);  
  
QGraphicsEllipseItem *item;  
item = new QGraphicsEllipseItem(-40, -40, 30, 30, rootItem);  
item->setBrush(Qt::yellow);  
item = new QGraphicsEllipseItem( 10, -40, 30, 30, rootItem);  
item->setBrush(Qt::blue);  
item = new QGraphicsEllipseItem(-40,  10, 30, 30, rootItem);  
item->setBrush(Qt::red);  
item = new QGraphicsEllipseItem( 10,  10, 30, 30, rootItem);  
item->setBrush(Qt::magenta);  
  
scene->addItem(rootItem);
```

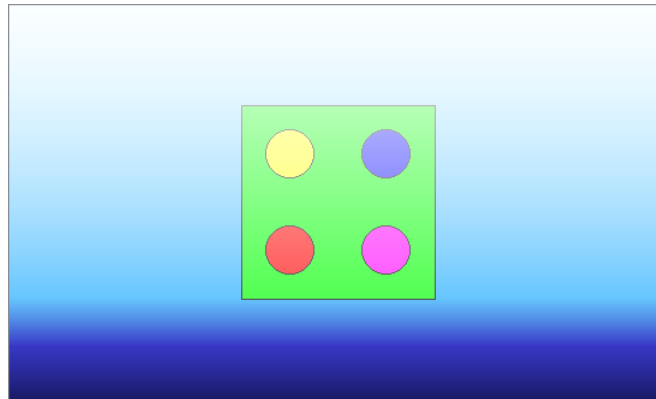


QGraphicsItems can
be placed in an
object hierarchy



Background and Foreground

- It is possible to set both foreground and background brushes for a scene or view



```
scene->setForegroundBrush(hazeBrush);  
scene->setBackgroundBrush(blueToBlackBrush);
```

- Sub-class the view and reimplement drawBackground and drawForeground for custom painting



Setting up a view to a scene

- The QGraphicsView widget serves as the viewport in which the scene is shown

```
QGraphicsView *view = new QGraphicsView();  
QGraphicsScene *scene = setupScene();  
view->setScene(scene);
```

- By default, the scene is centered. Use the alignment property to control this
- The view class is derived from QAbstractScrollArea. From this class the horizontalScrollBarPolicy and verticalScrollBarPolicy properties are inherited



Basic Transformations

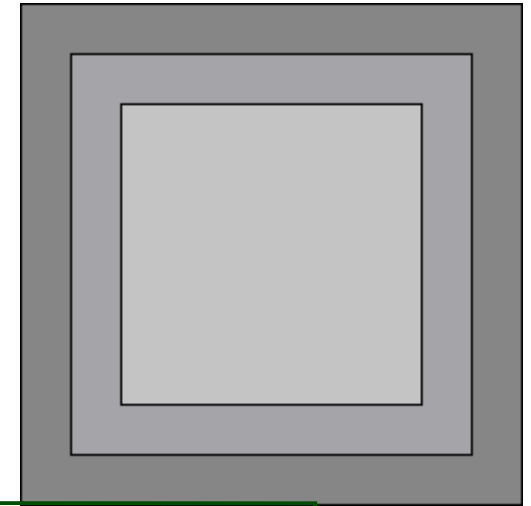


- Both the QGraphicsItem class and the QGraphicsView class can be transformed
 - scaling
 - translating
 - rotating
 - shearing
 - 2.5D effects



Nested Transformations

- When transforming parent items, the children are also transformed



```
QGraphicsRectItem *rootItem = new QGraphicsRectItem(...);  
rootItem->setBrush(Qt::darkGray);
```

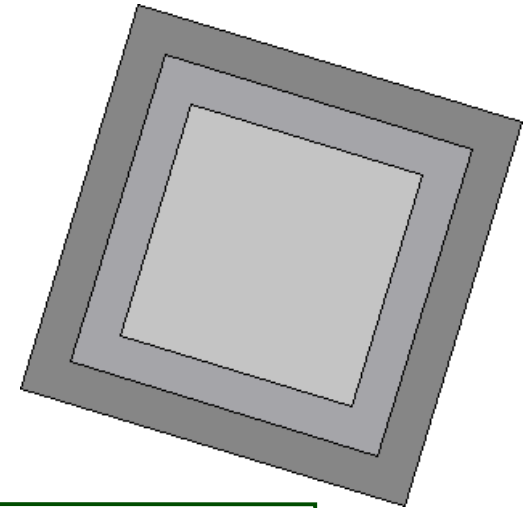
```
QGraphicsRectItem *midItem = new QGraphicsRectItem(..., rootItem);  
midItem->setBrush(Qt::gray);
```

```
QGraphicsRectItem *topItem = new QGraphicsRectItem(..., midItem);  
topItem->setBrush(Qt::lightGray);
```



Nested Transformations

- When transforming parent items, the children are also transformed



```
QGraphicsRectItem *rootItem = new QGraphicsRectItem(...);  
rootItem->setBrush(Qt::darkGray);  
rootItem->setRotation(17);
```

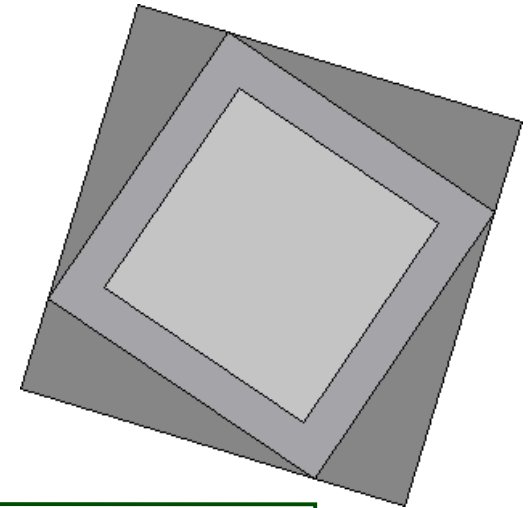
```
QGraphicsRectItem *midItem = new QGraphicsRectItem(..., rootItem);  
midItem->setBrush(Qt::gray);
```

```
QGraphicsRectItem *topItem = new QGraphicsRectItem(..., midItem);  
topItem->setBrush(Qt::lightGray);
```



Nested Transformations

- When transforming parent items, the children are also transformed

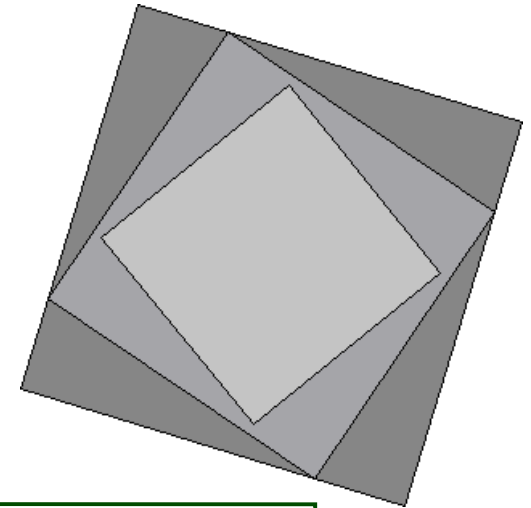


```
QGraphicsRectItem *rootItem = new QGraphicsRectItem(...);  
rootItem->setBrush(Qt::darkGray);  
rootItem->setRotation(17);  
  
QGraphicsRectItem *midItem = new QGraphicsRectItem(..., rootItem);  
midItem->setBrush(Qt::gray);  
midItem->setRotation(17);  
  
QGraphicsRectItem *topItem = new QGraphicsRectItem(..., midItem);  
topItem->setBrush(Qt::lightGray);
```




Nested Transformations

- When transforming parent items, the children are also transformed



```
QGraphicsRectItem *rootItem = new QGraphicsRectItem(...);  
rootItem->setBrush(Qt::darkGray);  
rootItem->setRotation(17);  
  
QGraphicsRectItem *midItem = new QGraphicsRectItem(..., rootItem);  
midItem->setBrush(Qt::gray);  
midItem->setRotation(17);  
  
QGraphicsRectItem *topItem = new QGraphicsRectItem(..., midItem);  
topItem->setBrush(Qt::lightGray);  
topItem->setRotation(17);
```



Coordinate systems



- The view, scene and each item has a local coordinate system
 - The view can
 - `mapFromScene` / `mapToScene`
 - The items can
 - `mapFromScene` / `mapToScene`
 - `mapFromParent` / `mapToParent`
 - `mapFromItem` / `mapToItem`
 - The scene always uses its own coordinate system

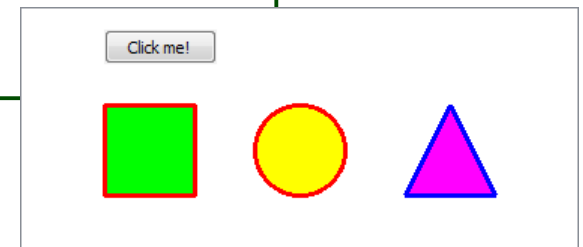


Embedding widgets



- It is possible to add widgets to a scene
- QGraphicsScene::addWidget returns a QGraphicsProxyWidget – a wrapper for the widget in the scene

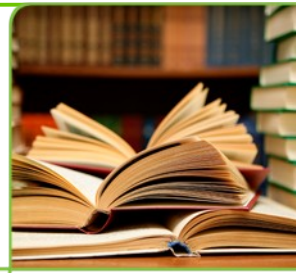
```
QGraphicsProxyWidget *button = scene->addWidget(  
    new QPushButton(tr("Click me!")));  
button->setPos(20, -30);
```



- This is a convenience solution and not a high performance option



Adding Interaction



- The flags of the items control how they can be interacted with
 - `ItemIsMovable` – a convenience feature, the original mouse event methods let the user drag the item
 - `ItemIsSelectable` – the item can be selected using `setSelected` and the `QGraphicsScene::setSelectionArea` method
 - `ItemIsFocusable` – the item can receive keyboard focus



Moving Items

- By setting the `ItemIsMovable` flag, items can be moved around using the mouse

```
QGraphicsItem *item;  
item = scene->addRect(...);  
item->setFlag(QGraphicsItem::ItemIsMovable, true);
```

- When an item is moved, the item receives `ItemPositionChange` events
- Using an event filter it is possible to trace movements in the standard items without sub-classing



Sub-classing Items

- When sub-classing items, there are numerous events that can be intercepted
 - `hoverEnter` / `hoverMove` / `hoverLeave`
 - `itemChange` (move, transform, selection, etc)
 - `keyPress` / `keyRelease`
 - `mousePress` / `mouseMove` / `mouseRelease`
 - etc
- It is also possible to implement the `sceneEventFilter` method and install the item as an event filter on selected items



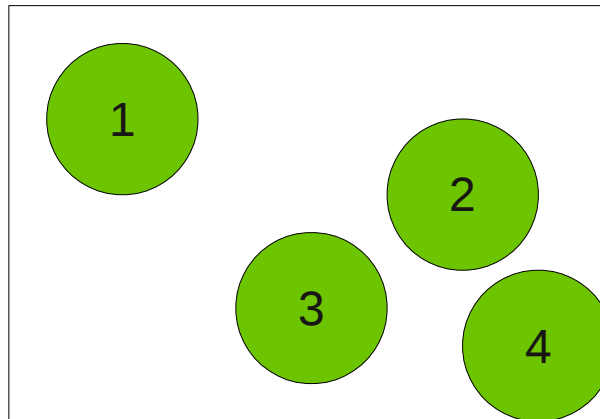
Optimizing itemAt tests

- QGraphicsScene::itemAt is frequently called when the user interacts with a scene
 - Relies on a BSP tree of all the items for performance reasons
- When items are moved, the BSP tree is updated
 - In a scene with lots of movements, updating the BSP tree can be heavier than using a less efficient itemAt
- Using the scene's bspTreeDepth and itemIndexMethod properties, the BSP tree can be tuned or disabled



What is a BSP tree

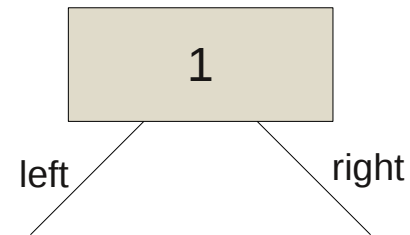
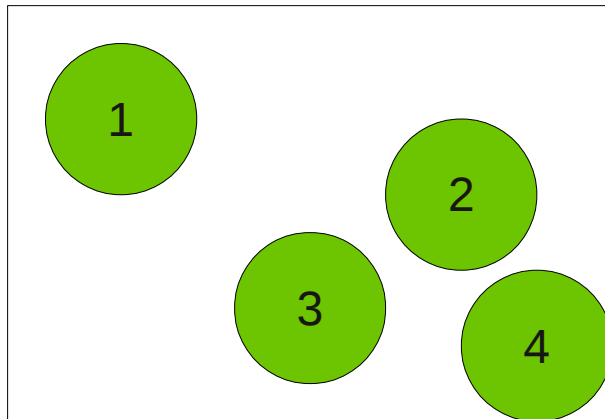
- Binary Space Partitioning trees store items in a tree, depending on their location in space





What is a BSP tree

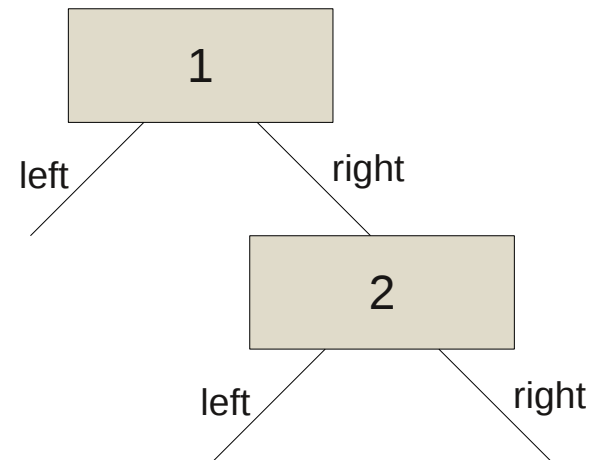
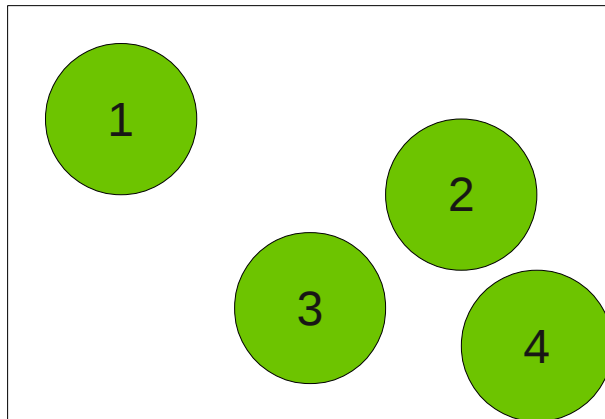
- Binary Space Partitioning trees store items in a tree, depending on their location in space





What is a BSP tree

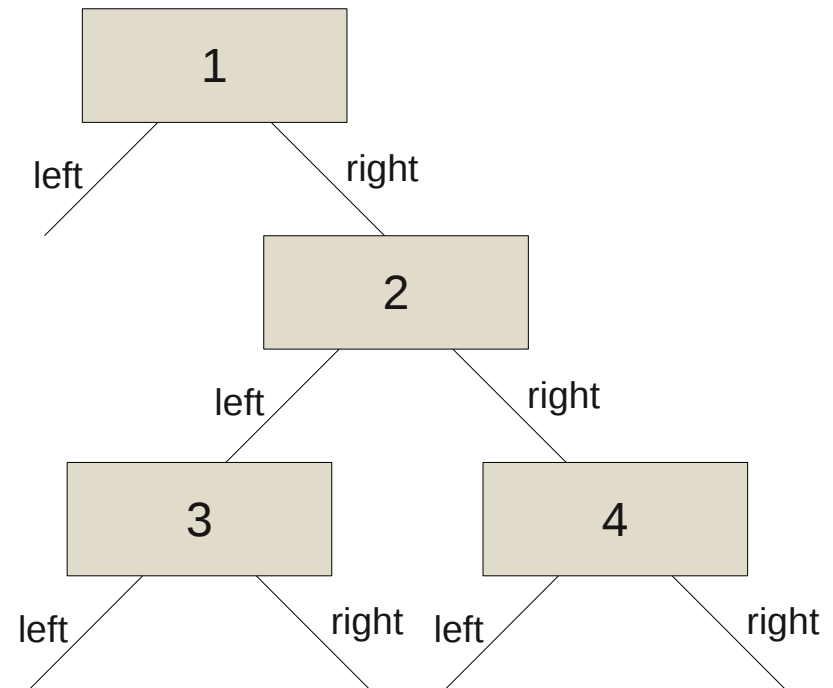
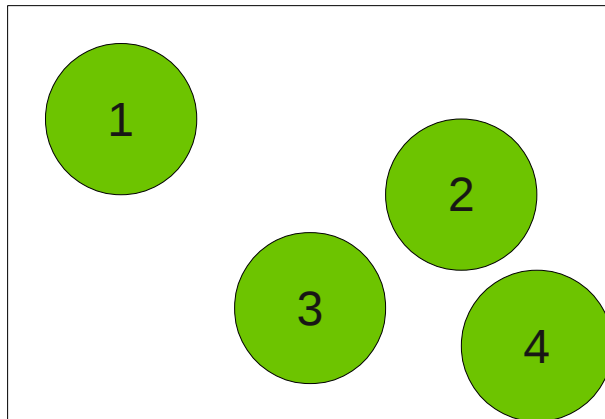
- Binary Space Partitioning trees store items in a tree, depending on their location in space





What is a BSP tree

- Binary Space Partitioning trees store items in a tree, depending on their location in space





Tuning view update regions



- To optimize performance for scene updates, Qt tries to repaint smartly
- The view's viewportUpdateMode controls how the repainting is done
 - FullViewportUpdate – the entire view is updated on all changes
 - MinimalViewportUpdate – only a minimal update is requested
 - SmartViewportUpdate – Qt tries to optimize
 - BoundingRectViewportUpdate – a minimal rectangular update
 - NoViewportUpdate – the updates must be triggered externally



Break



Custom Items



- There are basically two starting points when creating a custom graphics item
 - QGraphicsItem – simple graphics items
 - QAbstractGraphicsShapeItem – adds properties pen and brush
 - QGraphicsObject – graphics items that need QObject features such as signals and slots



A Basic Custom Item

- A basic custom item only provides visuals
 - A bounding rectangle defining the extent of the item
 - A paint method, painting the actual item

```
class BasicItem : public QGraphicsItem
{
public:
    BasicItem(QGraphicsItem *parent=0);

    QRectF boundingRect() const;

    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option,
               QWidget *widget);
};
```



A Basic Custom Item

```
QRectF BasicItem::boundingRect() const
{
    return QRectF(0, 0, 100, 100);
}

void BasicItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option,
    QWidget *widget)
{
    painter->setPen(Qt::NoPen);

    QRadialGradient gradient = radialGradient();
    painter->setBrush(gradient);

    painter->drawEllipse(boundingRect());
}
```



A partially
transparent
brush gives
a shading item



Custom Items

- The paint method is called when needed
 - not necessarily for all repaints
 - Call update to trigger a repaint
- The boundingRect must contain all of the item's graphics
 - Do not forget that wide pens paint on both sides of the specified line
 - When resizing, make sure to call prepareGeometryChange *before* you change the size of the item



Interacting



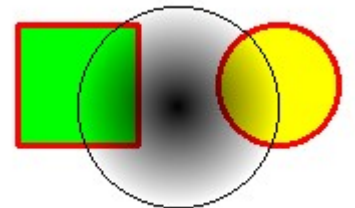
- The flag `ItemIsMovable` gives basic movability

```
item->setFlag(QGraphicsItem::ItemIsMovable, true);
```

- The flag `ItemIsSelectable` makes it possible to select the item in question
 - The item is automatically requested to repaint itself

```
item->setFlag(QGraphicsItem::ItemIsSelectable, true);
```

```
void BasicItem::paint(...)
{
    if(isSelected())
        painter->setPen(Qt::black);
    else
        painter->setPen(Qt::NoPen);
}
```



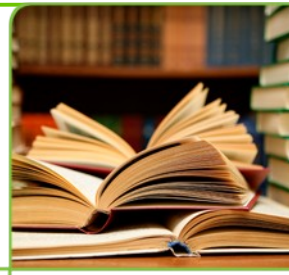


Custom Interaction

- The movability and item selection is implemented in the default mouse event handling methods
- To gain full control, you can re-implement the event handling functions directly
 - `setAcceptHoverEvents` enables hover events
 - `setAcceptTouchEvent` enables touch events
 - `setAcceptedMouseButtons` defines which buttons are handled by the item (default is to accept all buttons)



Interaction Example



- An interactive custom item listening to hover events and mouse press events
 - When hovered, the item expands
 - When pressed, it changes appearance



inactive



hovered



pressed



Interaction Example Class Declaration

```
class InteractiveItem : public QGraphicsItem
{
public:
    InteractiveItem(QGraphicsItem *parent=0);

    QRectF boundingRect() const;
    void paint(...);

protected:
    void hoverEnterEvent(QGraphicsSceneHoverEvent*);
    void hoverLeaveEvent(QGraphicsSceneHoverEvent*);

    void mousePressEvent(QGraphicsSceneMouseEvent*);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent*);

protected:
    bool m_pressed;
    bool m_hovered;
};
```



Interaction Example Constructor

- Initializing the item, making sure that both internal states are false

```
InteractiveItem::InteractiveItem(QGraphicsItem *parent) :  
    QGraphicsItem(parent),  
    m_pressed(false), m_hovered(false)  
{  
    setAcceptHoverEvents(true);  
}
```



Interaction Example

Geometry and Painting

- The bounding rectangle depends on the hovered state, while appearance depends on both hovered and pressed

```
QRectF InteractiveItem::boundingRect() const
{
    if(m_hovered)
        return QRectF(-50, -50, 101, 101);
    else
        return QRectF(-30, -30, 61, 61);
}

void InteractiveItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    QRadialGradient gradient;
    if(m_hovered)
        ... // Setup gradient

    if(m_pressed)
        ... // Setup gradient

    ... // Paint here
}
```



Interaction Example

Mouse Events

- The mouse events only affect the appearance
 - State change is followed by call to update

```
void InteractiveItem::mousePressEvent(QGraphicsSceneMouseEvent*)
{
    m_pressed = true;
    update();
}

void InteractiveItem::mouseReleaseEvent(QGraphicsSceneMouseEvent*)
{
    m_pressed = false;
    update();
}
```




Interaction Example

Hover Events

- The hover events affect the bounding rectangle
 - First call `prepareGeometryChange`, then alter state

```
void InteractiveItem::hoverEnterEvent(QGraphicsSceneHoverEvent*)
{
    if(!m_hovered)
    {
        prepareGeometryChange();
        m_hovered = true;
    }
}

void InteractiveItem::hoverLeaveEvent(QGraphicsSceneHoverEvent*)
{
    if(m_hovered)
    {
        prepareGeometryChange();
        m_hovered = false;
    }
}
```



QGraphicsObject



- The QGraphicsItem class is not derived from QObject – this means that
 - Items cannot have properties
 - Items cannot have slots
 - Items cannot emit signals
- The QGraphicsObject class is a QObject derived QGraphicsItem class



QGraphicsObject

- When sub-classing QGraphicsObject, there are some things to keep in mind
 - Relationships between items are explored using `parentItem` and `childItems`
 - Relationships between QObjects are explored using `parentObject` and `parentWidget`
 - Use QGraphicsItem's features for modifying ownership trees, e.g. `setParentItem`



Interactive QGraphicsObject

- Start from the InteractiveItem class
 - Change the name to InteractiveObject
 - Change the base class to QGraphicsObject

```
class InteractiveObject : public QGraphicsObject
{
    Q_OBJECT
public:
    explicit InteractiveObject(QGraphicsItem *parent = 0);
```

The parent is
still an item

- In the mouse release event, emit a signal if the mouse button is released while over the circle

```
QPointF delta = boundingRect().center()-ev->pos();
qreal radius = boundingRect().width()/2.0;
if(delta.x()*delta.x()+delta.y()*delta.y() <= radius*radius)
    emit clicked();
```



Widgets in a scene

- It is possible to place QWidgets in a graphics view scene
 - The widgets are managed by a QGraphicsProxyWidget
 - Performance is not very good
 - Complex widgets are not always rendered correctly
 - Use for quick hacks or when migrating code

```
QWidget *myWidget = new QPushButton(tr("Qt"));  
QGraphicsProxyWidget *proxyWidget =  
    scene->addWidget(myWidget);
```



Rendering Hints



- It is possible to tune the QGraphicsView to render the scene more quickly or in greater detail
 - renderHints – a set of flags controlling rendering
 - Antialias – enables anti-aliasing
 - SmoothPixmapTransform – enables smoother pixmap transformations



Caching

- The caching of individual QGraphicsItems can greatly affect performance
- Caching is tuned using QGraphicsItem::setCacheMode
 - ItemCoordinateCache
 - Caches the item in its local coordinate system
 - Can reduce rendering quality
 - Call setCacheMode again to resample the item
 - DeviceCoordinateCache
 - Perfect for items that do not apply other transformations than translation



Enabling Hardware Acceleration

- To use OpenGL for rendering the scene, change the viewport widget of the view

```
QGraphicsView *view = new QGraphicsView();  
view->setViewport(new QGLWidget);
```

- The renderHint flag `HighQualityAntialiasing` can be set to enable fragment programs and off-screen rendering for antialiasing when using OpenGL



Enabling Hardware Acceleration

- To use OpenGL for rendering the scene, change the viewport widget of the view

```
QGraphicsView *view = new QGraphicsView();  
view->setViewport(new QGLWidget);
```

- The renderHint flag `HighQualityAntialiasing` can be set to enable fragment programs and off-screen rendering for antialiasing when using OpenGL