Exercises Lecture 6 – The Graphics View Canvas

Aim: This exercise will take you through the process of using the Graphics View framework as well as extending it with custom items.

Duration: 1h

© 2012 Digia Plc.

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.

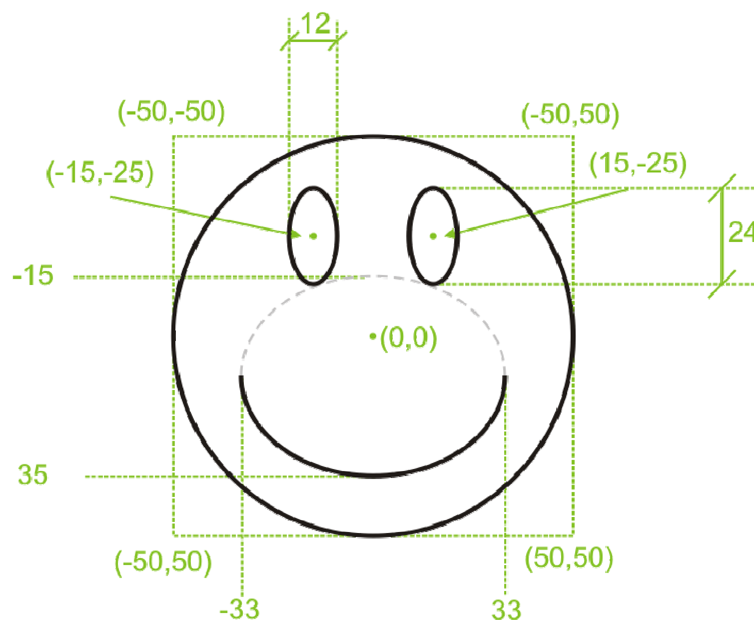
Digia, Qt and the Digia and Qt logos are the registered trademarks of Digia Plc. in Finland and other countries worldwide.

Composing Items

This exercise comes with a source code package. Extract that package to a location of your choice. In it, you will find a number of projects that will serve as starting points for the steps of this exercise.

The first task is about creating a smiling face from existing graphics view items. These will be composed into a single item. The starting point is the *composedgraphicsitem* project.

Create the face by implement the `addSmiley()` function. The figure below suggests coordinates for different parts of the smiley. The starting point only draws the face itself. Your task is to add two eyes (`QGraphicsEllipseItem`) and the smile (`QGraphicsPathItem`).



To get you started, the code for the left eye is shown below. Notice that the rectangle of the ellipse needs to be calculated from the center point (-15, -25) and the dimensions of the eyes (12x24). Also notice that the parent of the eye is `face`.

```
QGraphicsEllipseItem *leftEye =
    new QGraphicsEllipseItem(QRect(-21, -37, 12, 24), face);
leftEye->setPen(QPen(Qt::black));
leftEye->setBrush(Qt::white);
```

The smile is created using the `QGraphicsPathItem` class. When using a path item, you first create a `QPainterPath`. Using the `arcMoveTo` and `arcTo` methods you can create the smile. Finally, you provide the path to the path item using the `setPath` method.

Transforming items

Start from the *transformedsmiley* project. It is the same project that you used in the last step, but the smiley is added three times to the graphics view scene.

The purpose of adding the smiley several times is to be able to compare them after having applied different transformations to them.

For `smiley1`, scale the item to double size using the `scale` method, then translate 200 pixels to the

© 2012 Digia Plc.

left using the `translate` method.

For `smiley2`, rotate the item 45 degrees counter-clockwise using the `rotate` method.

For smiley3, rotate the object 70 degrees around the X-axis. To do this, you must create a `QTransform` object. When using the `rotate` method of the transformation object, you can specify the axis to rotate around. When the item has been rotated, make sure to translate it 200 pixels to the right using the `translate` method. To apply the `QTransform` object to the `smiley3` item, use the `setTransform` method.

Moving items

The next step is to make the items movable. You can either continue from your last project or use the *movablesmiley* project as a starting point.

Now, make the items movable by setting the appropriate flags on them. You should only have to add one line of code per item and not write any item movement code of your own.

Custom item

Until now, you have relied on using the graphics items provided as a part of the graphics view framework. In this step, you will create an custom item of your own. The item in question is the smiley from the first step of this exercise, but now in the form of the `SmileyItem` class.

Use the *customitem* project as a starting point. It already contains a `SmileyItem` skeleton class that holds a basic implementation of a custom item.

You will have to implement the boundingRect method, as well as the paint, paintEye and paintSmile methods.

Start by implementing the `boundingRect` method, so that the returned rectangle, `br`, is a rectangle containing the entire smiley that you will paint. This rectangle is used by the graphics view framework to determine if your item needs repainting, or if it has been clicked by the user.

The painting is handled by the three methods `paint`, `paintEye` and `paintSmile`. As this is a custom item, all painting is done from scratch using the `QPainter` passed as the `painter` argument.

Start by implementing the `paint` method. There, add code to draw the outline of the smiling face. Also fix the `paintEye` calls so that they use correct coordinates and ensure that the `paintSmile` gets the correct rectangle.

In the `paintEye` method, draw an ellipse for the eye. In the `paintSmile` method, draw an arc for the smile.

In all painting methods, ensure that you have set up a proper pen and brush for each painting operation.

Interactive items

The next step is to add interactivity to the custom item. As there is some support code added to the project, use the *interactivesmiley* project as a starting point. Feel free to compare the projects to learn what has been added.

The first interaction for you to add, is to have the smiley's eyes follow the mouse when the mouse pointer is hovering the item. The new paintEye method has updated code using the m_focus member variable as the focus point of the eyes.

© 2012 Digia Plc.

Add code to the methods `hoverEnterEvent`, `hoverMoveEvent` and `hoverLeaveEvent`. Make the `m_focus` point follow the point given as `event->pos()` in the enter and move methods, and then set the point to `QPointF()` in the leave method. Also, ensure to call `update` from all three methods, as the item needs to be repainted.

When testing the eye movement functionality, you will notice that they eyes start moving even if the pointer is not over the circle of the face. The eyes start moving as soon as the pointer is inside the bounding rectangle. To remedy this, implement the `shape` method.

The shape method returns a `QPainterPath` describing the exact shape of the item. In this case, an ellipse. Simply create a painter path, call `addEllipse` to add an ellipse to the path and then return said path. This should give you perfect eye movement behavior.

The other feature of the painter method of this `SmileyItem` incarnation is the `m_bigEyes` boolean. If it is true, the eyes are enlarged. By modifying the flag from the `mousePressedEvent` and `mouseReleasedEvent` methods, you can make the eyes grow when a mouse button is pressed while over the face. Again, do not forget to call the `update` method when you have modified a member variable affecting the appearance of an item.

Adding signals and slots

In this final step of the exercise you will add signal and slot capabilities to your graphics view item. To do that, you need to inherit `QGraphicsObject` instead of `QGraphicsItem`, as well as adding a set of functions. Part of this has been done in the *objectitem* project, which is your recommended starting point.

First, lets complete the interface. The starting point adds the methods `setSmileSize` and `smileSize` to the class along with the private member `m_smileSize`. To complete this interface, add the signal `smileSizeChanged` to the class declaration and ensure that the class starts with the `Q_OBJECT` macro.

In the class, implement the `setSmileSize` method so that the smile size only can be in the range 0-100. If the size of the smile has changed, emit the `smileSizeChanged` signal and call update to request a repaint of the item.

You can check the results of your work by running the program and dragging the slider handle. If it doesn't work and you get some warnings from Qt on the console, read them carefully and check if you didn't omit any of the steps mentioned in this instruction.

© 2012 Digia Plc.

Solution Tips

Step 1

To make the smile item render the smile you have to create a `QPainterPath` object for it and and an arc to it. To do that first move the path cursor to appropriate position using `QPainterPath::moveArcTo()` and then add an arc to the path.

```
QPainterPath smileArc;
QRect rect(-33, -15, 66, 50);
smileArc.arcMoveTo(rect, 0);
smileArc.arcTo(rect, 0, -180);
```

Step 3

To make an item movable, use `QGraphicsItem::setFlag()` to set the `ItemIsMovable` flag on the item.

Step 4

Have the origin (point (0,0)) of the item point to the center of the face by returning a proper rectangle from the boundingRect() implementation:

```
return QRect(-50,-50, 100, 100);
```

Step 5

The position of the cursor is carried by `QGraphicsSceneHoverEvent::pos()`. Have its result assigned to the `m_focus` variable before calling `update()`.

```
m_focus = event->pos();
update();
```

The item needs to report a proper shape. To do that, create a painter path containing an ellipse covering the area of the smiley's face. Use the same coordinates as for drawing the face.

```
QPainterPath path;
path.addEllipse(boundingRect());
return path;
```

Step 6

Having added the `Q_OBJECT` macro, you have to rerun qmake on the project for Qt to recognize the change.

Use the `qBound(min, value, max)` macro to limit the range of a variable.

© 2012 Digia Plc.