A Note of the `QGraphicsScene`/`QGraphicsView` System

Peter Rockett

21st April 2021

Like all GUI libraries, QtPython (a.k.a. PySide6) is able to perform various low-level drawing operations. To do this, QtPython uses the `QPainter` class that has various methods for rendering geometric primitives such as ellipses, lines, *etc.* It would be entirely possible to implement the whole project using nothing more than the `QPainter` class—indeed, there are some arguments as to why you might want to do this. However, QtPython has also introduced a higher-level set of functionality (that interfaces with and uses `QPainter` to perform the low-level rendering): the `QGraphicsScene` system. Using `QGraphicsScene` and friends has a number of advantages, especially for the management and rendering of complex scenes comprising hundreds or thousands of elements. The `QGraphicsScene` system comprises three inter-related classes:

1. The `QGraphicsScene` class that acts as a *container* for graphical elements
2. The `QGraphicsView` class that is associated with a `QGraphicsScene` instance; in fact, a `QGraphicsScene` can be associated with a number of `QGraphicsView` instances providing a number of simultaneous views of the same graphical elements—for example, providing a zoomed-in view of a larger grouping.
3. The `QGraphicsItem` class that represents specific graphical primitives, such as an ellipse. `QGraphicsItem` instances are contained in a `QGraphicsScene` instance—in some sense, the `QGraphicsScene` class acts as a sort of database of `QGraphicsItem` instances.

(To this, we should add the `QGraphicsItemGroup` class which allows more than one `QGraphicsItem` instance to be grouped together, *i.e.* treated as a single entity. It is probably unlikely that the `QGraphicsItemGroup` class will be of interest in this project.)

The `QGraphicsScene` architecture can be summarised by Figure 1. The `QGraphicsScene` instance contains a number of `QGraphicsItem` instances that are rendered using the associated `QGraphicsView` instance.

Event Handling

Clearly event handling is a major part of GUIs, in particular, mouse events. In general, a mouse event comprises a short message that contains: an integer identifier (*e.g.* signifying left mouse button up), as well as the (x, y) coordinates of the cursor at the time the event was triggered.

Since it is concerned with graphical presentation, the `QGraphicsView` instance receives the low-level mouse events and routes them to its associated `QGraphicsScene` instance. The `QGraphicsScene` instance then directs the mouse event to the `QGraphicsItem` that the mouse has selected.

- The *mechanism* by which this happens is that each `QGraphicsItem` has a *bounding rectangle*—a rectangle that only just encloses the graphical shape. When a `QGraphicsScene` instance receives a mouse event, it uses the coordinates of the mouse event to locate the matching `QGraphicsItem` instance by searching for the item whose bounding rectangle contains the mouse coordinates. In fact, the items stored in the `QGraphicsScene` instance are arranged as a binary tree structure which means the search can run in $\mathcal{O}(\lg n)$ time; this is very useful for something like a CAD application where the scene may contain thousands of individual graphical elements.

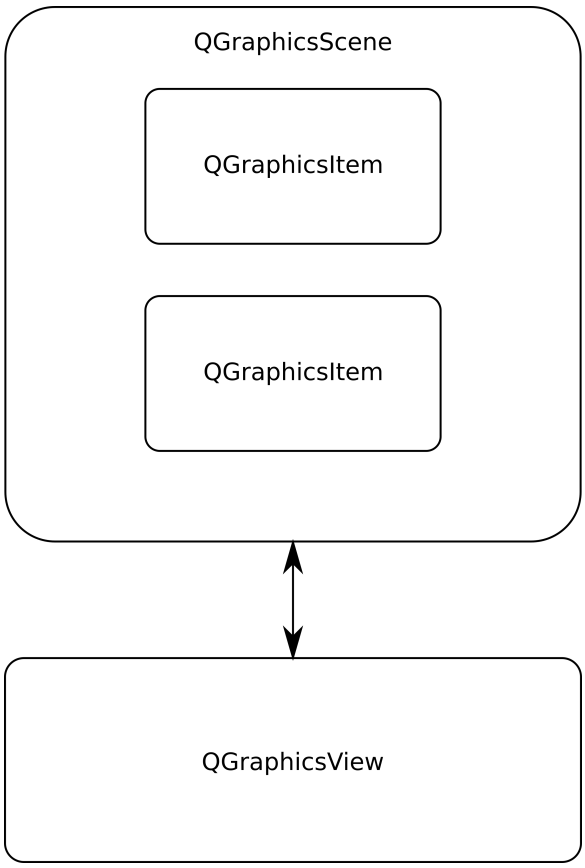

Figure 1: Illustration of the `QGraphicsScene` architecture.

- Since the mouse event is (ultimately) routed to an instance of a `QGraphicsItem`, it should be clear that the way to use this class is to create a sub-class, and implement a specific handler for the mouse event in each sub-class. In that way, a click on, say, a node can perform a different action compared to a click on an arc.
- When a mouse event handler make a change to the geometry of a `QGraphicsItem`, it is usually necessary to call its `prepareGeometryChange` method to update the information held in the `QGraphicsScene` instance You would then typically all the `update` method to trigger `QGraphicsScene` to send a repaint event to the `QGraphicsView` instance so the item is updated on the screen. Notice this chain of operations is 'hidden' in the `QGraphicsScene` framework whereas you would have to explicitly program it if you were using `QPainter` directly.

Coordinate Systems

The Qt graphical framework has a number of simultaneous coordinate frames that each make sense for the entity you are dealing with—in general, you have to map from one coordinate frame to another to interface components. (In fact, the erratic mouse drag behaviour of the initial example program is due to the failure to properly map between different coordinate systems.) Fortunately, Qt provides a number of maps-to and from convenience methods. You will have to ensure that you are performing the correct mappings between coordinate frames.

Pen Widths and Bounding Rectangles

Finally, let me point out a GUI gotcha connected with drawing object that have bounding rectangles. If you redraw a graphical primitive with a pen of width, say, 2 pixels, half of the width of the pen will

fall outside the bounding rectangle, and so the shape will be drawn with an outline of one pixel width. You need to be aware of this and possibly adjust the size of bounding rectangles accordingly.

This and related phenomenon is simply an inevitable result of the mismatch between specifying computer graphics elements on an integer grid, and the fact that classical geometry is defined in terms of the real numbers $\mathbb{R}$ and with concepts like lines of zero width.