UNIVERSITY OF LIVERPOOL

Application Development with C++ (ELEC362)

Lecture 17: Drawings in Qt Applications

mihasan@liverpool.ac.uk

Previous lecture

- The concept of events and event handling was discussed.
- The coordinates systems in Qt were discussed.
- The concept of actions has been discussed.
- Constructing menus in menu bars, toolbars and context menus was shown
- Creating a child window in an application was shown.
- Classes introduced: **`QEvent, QKeyEvent, QMouseEvent, QTouchEvent, QAction, QMenu, QContextMenuEvent, QTextEdit`**

This lecture

- What is covered in this lecture?

  Classes related to drawings, paintings, and image handling

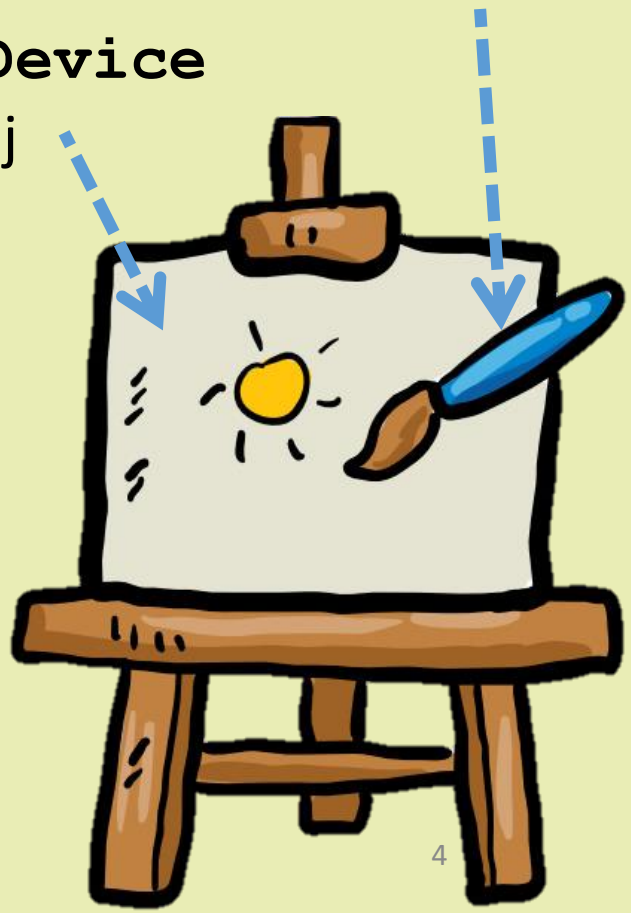
- Why it is covered?

  Almost every application in existence has one aspect related to paints/drawings

- How are topics covered in this lecture: Live demonstrations

`QPainter` Class

- This class is responsible for low-level drawings ranging from simple lines to complex polygons and images.
- Reference: https://doc.qt.io/qt-5/qpainter.html
- Objects of this class can operate on any object of a class inheriting **`QPaintDevice`** (such as **`QWidget`**).
- A **`QPainter`** object can operate on a widget by implementing the virtual function **`paintEven(QPaintEvent *event)`**.
- A **`QPainter`** object initiates the drawing process, it can draw independently, or using other objects.



Classes that work with **QPainter**

- This is a brief list of selected classes used for painting with **QPainter** objects:

Class	What it does
QPen	Defines a “pen” to draw shapes (its style, width, and colour).
QBrush	Defines the colour and the fill pattern of closed shapes.
QRectF	Defines a rectangle (F stands for Float).
QPolygon	Defines polygons by having a set of points.
QPointF	Defines a point (F stands for Float).
QImage	Defines an image imported to the project.

- A simple way to think of it is to imagine taking the widget to MS Painter to paint on it and return it back to Qt.

Painting events

- The function **`paintEvent()`** is implicitly called in the constructor.
- To change a painting during the operation of the application, the function **`repaint()`** or **`update()`** should be called. Both functions belong to the widget where the paint change should occur.
- The function **`repaint()`** redraws the widget immediately, but can cause flickering for widgets with heavy paints. The function **`update()`** redraws the widget in the next event loop iteration, thus it minimises the flicker.
- Example of flickering: https://www.youtube.com/watch?v=CxLvnNrutG4. It can be resolved by a technique known as double-buffering, which is implemented by default in Qt.

A sample application: Mouse tracker

- The task: Modify the Mouse tracker application such that the type of the click (left, right, or middle) is indicated by a changing colour of a rectangle in the application window.
- Build the application (demonstration).

Good practice note:

- Remember to include the header of every class that was not defined in Qt designer.
- If dealing with large paint, define shapes as objects rather than using drawing functions of **`QPainter`**

The graphics view framework

- For large drawings, dealing with every line, shape, or colour becomes a burden.
- A high-level alternative is the Graphics View Framework.
- Reference: https://doc.qt.io/qt-5/graphicsview.html
- It is far more efficient to deal with drawings as composition of objects rather than a composition of lines and colours.
- The graphics view framework is a special case of the model-view programming in Qt.
- Reference: https://doc.qt.io/qt-5/model-view-programming.html

The graphics view framework

- The graphics view framework consists primarily of three classes:
- **`QGraphicsScene:`** The area of the drawing.
- **`QGraphicsView:`** The window showing the visible part of the drawing.
- **`QGraphicsItem:`** The building blocks of the drawing.

`QGraphicsItem` obj

`QGraphicsScene` obj

`QGraphicsScene` Class

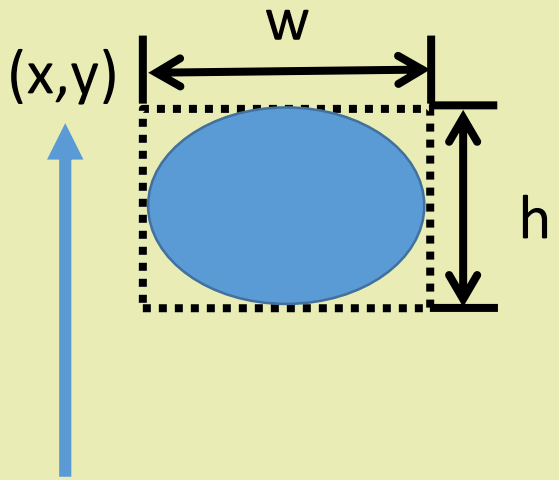
- This class provides surface to draw large 2D graphic items, all of which are **`QGraphicsItem`** objects.
- Reference: https://doc.qt.io/qt-5/qgraphicsscene.html
- An object of this class can be thought of as a hidden drawing, it needs a **`QGraphicsView`** object to be seen.
- It can efficiently manage millions of items, providing functionalities such as zooming, locating and tracking of items with simple lines of code.

`QGraphicsView` Class

- This class provides a widget to visualise the content of a **`QGraphicsScene`** object.
- Reference: https://doc.qt.io/qt-5/qgraphicsview.html
- An object of this class can be added to the UI from Qt Designer.
- An object of this class can be used to visualise a whole scene or parts of it.
- It provides scroll bars, in addition to pan functionality to the scene it is visualising with minimal coding.

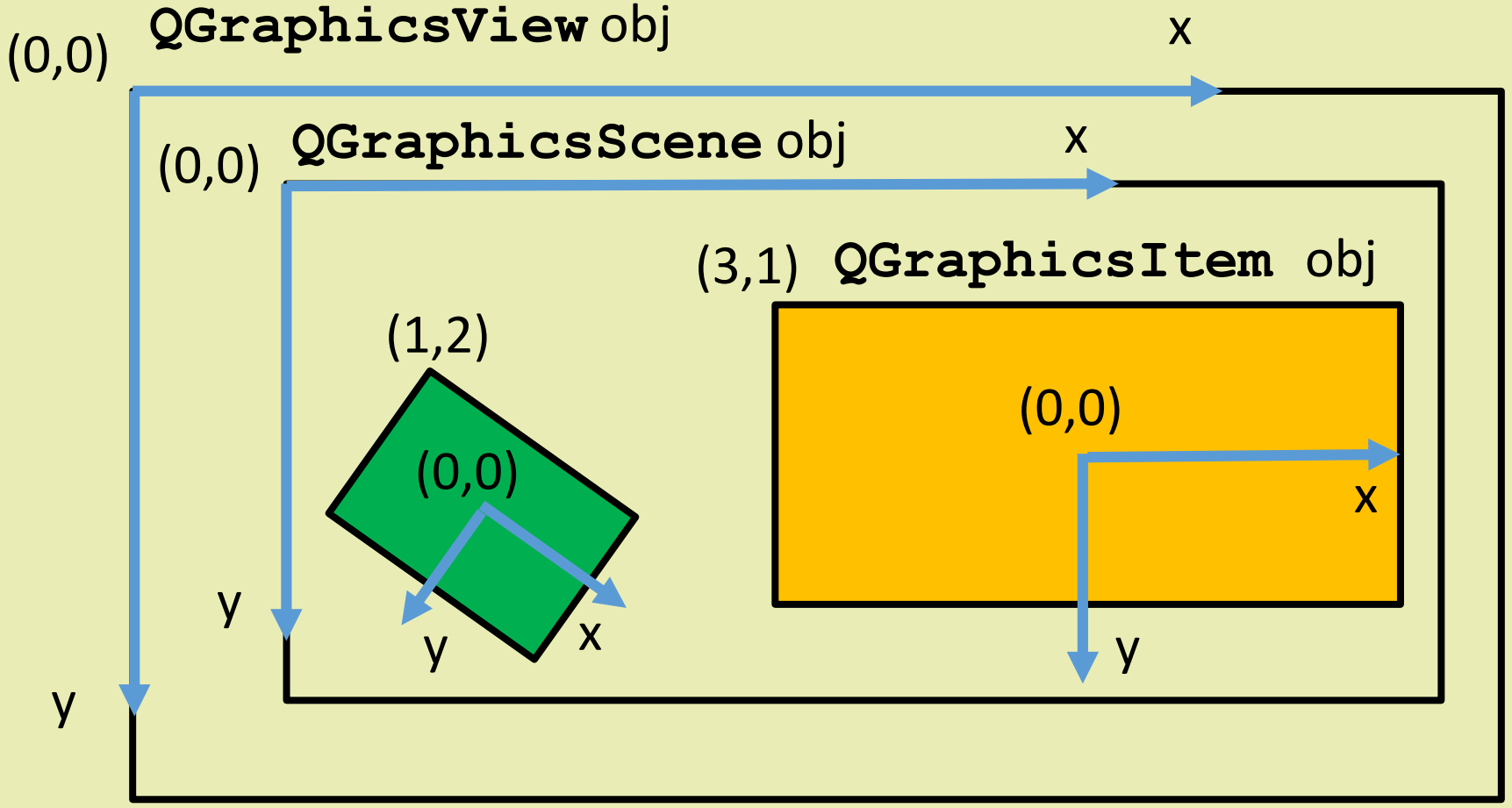
`QGraphicsItem` Class

- An abstract class that is the base class for all items in any **`QGraphicsScene`** object.
- Reference: https://doc.qt.io/qt-5/qgraphicsitem.html
- In addition to a large number of predefined items, such as rectangles, polygons, pixelmaps and paths, it allows the developers to define their custom items.
- It has lots of virtual event handlers, allowing the addition of complex functionality to the drawing, including item-item interaction.
- Every object of this class has a bounding rectangle



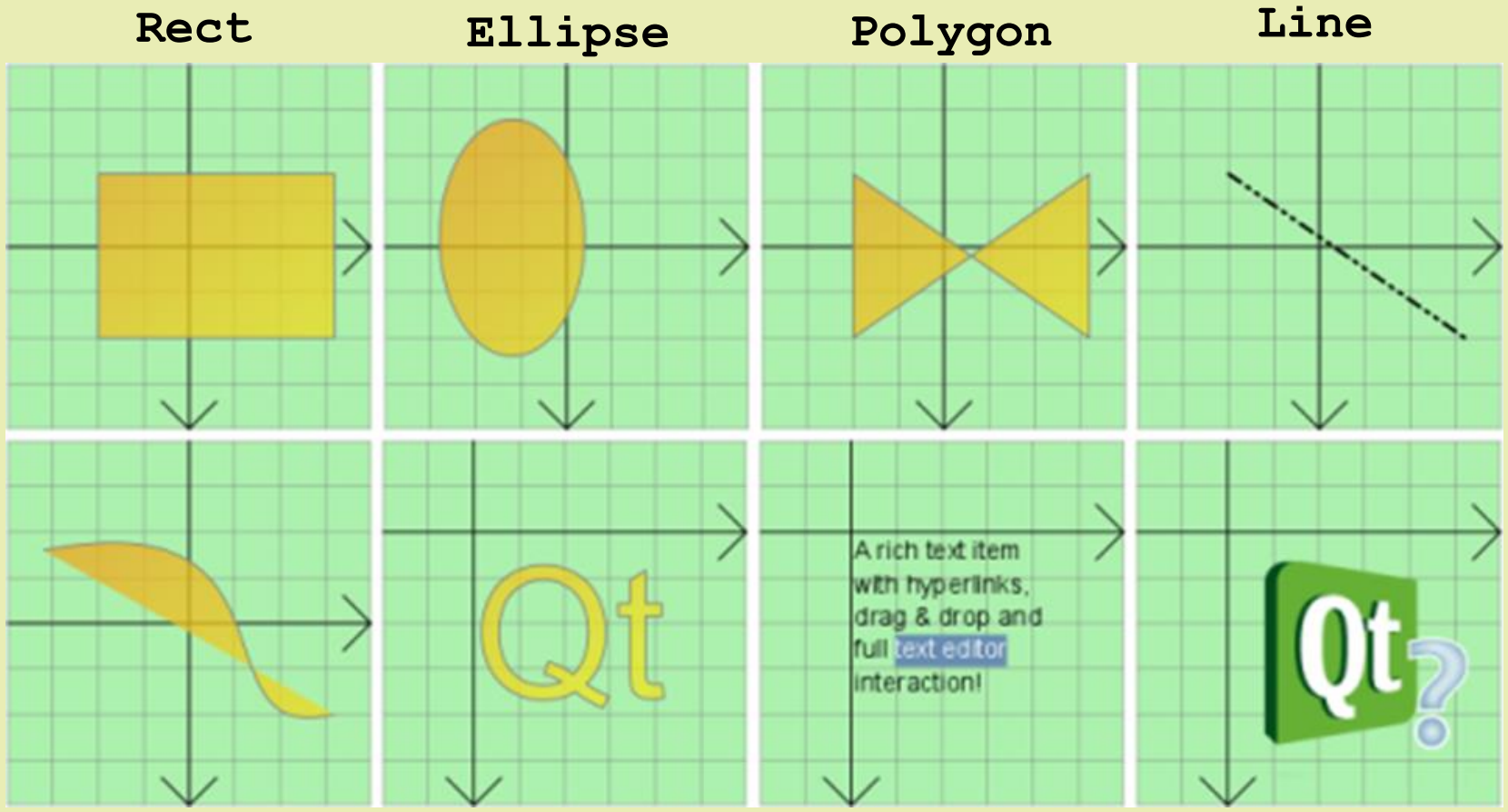
Coordinates transformation

- In the Graphics View Framework, every element has its own coordinates!



- The view has the coordinates of the widget.
- Items are placed using the scene's coordinates.
- Mapping is done using **`mapFromScene()`**, **`mapToScene()`**, and **`mapFromItem()`**

Derived classes from `QGraphicsItem`



- The name of any of these classes is:

  `QGraphics"word"Item`

where **`"word"`** is one of these indicated to the left.

- You can define your own custom items.

A sample application: Orbit animation

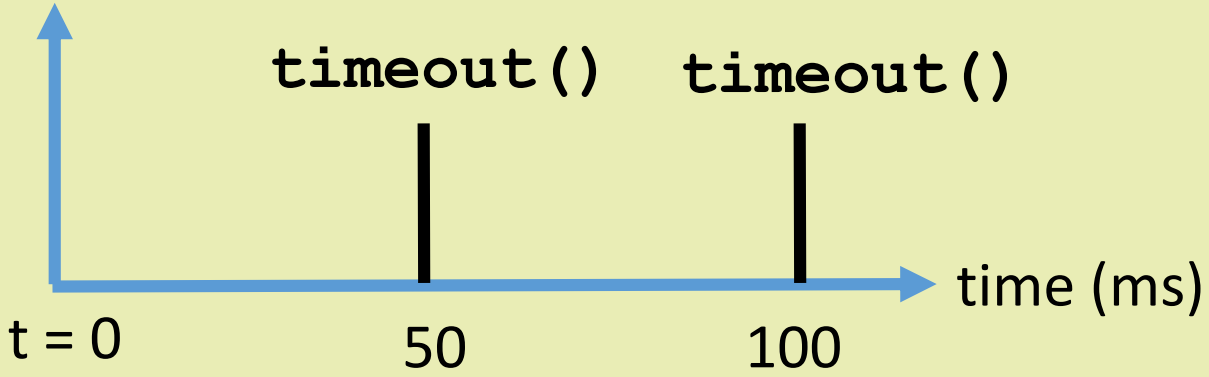
- The task: Design and implement a Qt-based GUI application that animates the motion of the earth around the sun.
- Build the application (demonstration).

Good practice note:

- Remember to set the scene in the constructor when working with graphics view framework.
- Because the scene object and the view object are highly dynamic, verify that they interact as intended while developing the code.
- When you need to scale, rotate or shear items, draw a small circle centered at the origin of the scene to keep track of your work.

`QTimer` Class

- This class provides time synchronised signals in single shot mode and in repetitive mode.
- Reference: https://doc.qt.io/qt-5/qtimer.html
- The signal emitted from an object of this class is called **`timeout()`**.
- Combining a Timer object with paint/graphics is the basis for simple animations.
- They are best implemented using the signal-slot mechanism.



`QTransform` Class

- This class controls the translation, rotation, scaling, and shearing of coordinate system .
- Reference: https://doc.qt.io/qt-5/qtransform.html
- Typically used when dealing with graphics, as well as simple drawings using **`QPainter`**.
- It works by multiplying the coordinates of every point in the coordinate by a matrix to obtain the new coordinates. The matrix can be automatically generated or specified by the developer.
- The origin point in any transform is the origin of the view. Unless the function **`translate()`** is called, which moves the origin point of the transform.

Graphics View Framework vs `QPainter`

- A comparison between the two approaches:

Aspect	`QPainter`	Graphics View Framework
Simplified concept	Similar to changing canvases when a modification is required	Similar to changing “stickers” on a single canvas.
Interaction with user	Not possible as the drawing is static	Ideal for interaction as every element is an object
Ease of coding	Relatively easy	Requires invoking `QTransform` when the objects change shape/position

Summary

- In this lecture, different drawing/graphics approaches in Qt were discussed.
- Painting on widgets was discussed.
- Handling graphics using the Graphics View Framework was discussed.
- Classes introduced: **`QPainter`**, **`QPen`**, **`QBrush`**, **`QRectF`**, **`QTimer`**, **`QGraphicsView`**, **`QGraphicsScene`**, **`QGraphicsItem`**, **`QTransform`**