Collision Detection In QtQuick

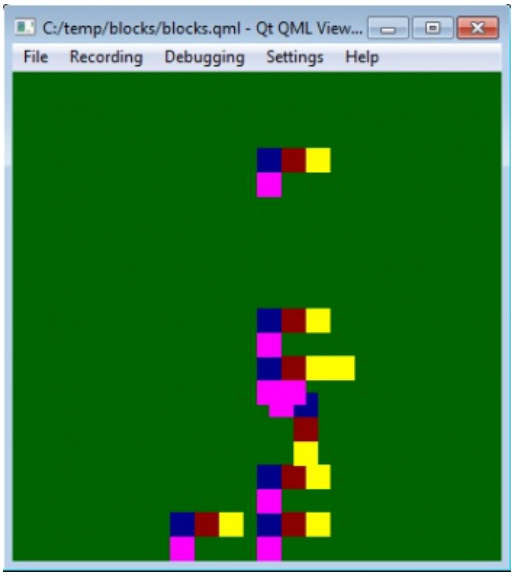
This article explains how easy pixel collision detection can be implemented in QML by using QML features as object property binding, object transition and object signal-slots communication mechanism. Also the article shows number of limitations of that implementation approach and ways how to workaround them

Introduction

Collision detection is a crucial part of any game implementation. There are number of physic engines available for developers and there are tries to apply some of them in QML projects as for example MotoTrialRacer game that uses Box2D engine. Physic engines implement sophisticated algorithms to emulate physic of object interaction in real world. The API requires C++ components implementation for QML projects. Sometimes that is too much for casual QML games. Can we find a cheaper solution ? QML offers great things as declarative programming approach , object transitions, signal-slot mechanism. Do we need extra API that will duplicate these features in some sense? That is what this article about

The project prototype

There is a game board a QML Rectangle element that serves as visual container for all the rest game components. Game component is an object that can fall and collide with other elements. On the elements collision falling object stops. Game component can consist of several QML Rectangle elements that can detect their position on screen in device coordinates then each element checks point ahead of that position in desired direction whether there is other game component. Thanks to QML the implementation code is very concise. There is no explicit loops in the code -- each element catches Y coordinate change signal broadcasted from game component. We unify the elements behavior just for simplicity sake and amazingly we do not get visible performance impact -- QML does the job nicely!



While figure is falling you can control its horizontal movement by mouse click from right or left side of the figure. If you click on the figure, it will rotate by 90 degree.

Implementation details

The actors

- **game board** is implemented in blocks.qml file (please see project package attached to the article File:Blocks.zip).

  It instantiates **figure** and makes it falling. When figure reached the bottom of the game board , game board creates new figure and makes it falling. It captures mouse click outside of the falling figure and passes the signal to figure to allow it some action. As you see we try to follow good old object incapsulation implementation principle.
- **figure** is an composition of smaller elements. It is implemented in figure.qml file

  it handles mouse clicks inside the figure and rotates itself
  it implements horizontal movement signalled from the game board
  it is container for smaller elements
  it has internal states : falling , stopped, reached-the-board-bottom. The latter is kind of axillary state needed to detect situation when no collision is expected but figure reached the board bottom and must stop
- **block** is a smallest visible elements in the game. It is implemented in block.qml file

  it implements collision detection
  when collision happens it signals to its parent , figure to change state from 'falling' to 'stop'

Collision detection

This humble function below does this job --- sorry no math at all.

```
.....
    function collisionDetection(){
```


(C) Copyright Nokia 2013. All rights reserved.

```
        var figure = parent.parent;
        var board = figure.parent;
        var stepahead = 1;

        var obj = parent.mapToItem(board, x, y);
        var pointx = obj.x;
        var pointy = obj.y;
        pointx += width / 2;
        pointy += height + stepahead;

        var foreign = board.childAt(pointx, pointy);
        if(foreign !== null && foreign.objectName !== '' && foreign.objectName !== figure.objectName){
            figure.state = "stop";
        }
    }

    Connections{
        target: figure
        onCheckmove:{
            collisionDetection();
        }
    }
......
```

The main point is : first you need to translate object's coordinate to game board coordinate and then by using game board coordinates query for board's children. When found children is the other figure then you have detected collision. There is small problem -- the game board has several children object that we do not want to count, as for example MouseArea and figure itself. How we can recognize other figure? --- Simple! Using time stamp as figure object name does the trick.

here is figure creation dynamically in the game board:

```
.....
        var figure = component.createObject(board, {
                                            objectName: Qt.formatTime(new Date(), "hhmmsszzz"),
                                            x: (board.width / 2),
                                            y:0,
                                            boardWidth: board.width,
                                            boardHeight: board.height,
                                            control: control
                                        });
        figure.reachedBottom.connect(createFigure)
.....
```

Note the last line -- is another trick i like to demonstrate. It connects signal from dynamically created object to the parent object. I see that is the best pattern to do so because you do not need to keep reference to variable 'figure' to use it later when the variable is gone when you go out of the function scope.

...And finally the implementation problem

It looks pretty nice, but try to rotate falling object and see how it will collide. It calculates figure position incorrectly! What is wrong! -- Simple: that is QML transformation in action. When you rotate the figure, 'x' and 'y' remains unchanged but transformation is applied dynamically when the figure is drawn. Such behavior makes it easy to layout objects in standard UI but leaves developer hopeless. On this problem there is an open bug report https://bugreports.qt-project.org/browse/QTBUG-18161 At least it is accepted and being analyzed by Qt team.

I see two workarounds here:

- not to use transformations. In that case you need to implement matrix rotation -- (don't confuse with rotating matrix) -- not big deal to google for ready solution
- subclass Rectangle element with method that will allow you to convert element's coordinates to device coordinates and vice versa. check QPainter class. I consider this missing as QML implementation flaw. Please argue me

Downloads

QML UI project File:Blocks.zip

Commercial release

qblocks

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.