Qt Kinetic scrolling - from idea to implementation

Introduction

16 Jan 2011

Kinetic scrolling is the combination of regular, drag-finger-on-screen scrolling with an additional movement after the finger is lifted off the screen. Based on how fast the finger was dragged on the screen, the duration, speed and deceleration of the additional movement can vary. This video shows kinetic scrolling in action on a Nokia Symbian touch screen phone. A notable difference from using a scrollbar is that when tapping and dragging, the list moves in the same direction that the user is dragging in, which is the opposite of how a scrollbar works. For a scrollbar, when one drags down the list moves up, for kinetic scrolling when one drags down, the list also moves down – it feels more natural this way.

Advanced kinetic scrolling implementations also feature „bounce" and „overshoot": in the first case the scrollable elements bounce off the scroll area's edge once the scroll bar reaches the end, while in the second case one can scroll past the end of the scroll area and it will exhibit drag resistance and snap back into place when released.

Recent Symbian OS touch screen phones feature kinetic scrolling prominently in the user interface, but surprisingly Qt apps do not have it enabled by default. A fully featured implementation is still in the works for Qt version 4.8. Additionally there is a back-port solution for Qt 4.6/4.7. In the mean time, a proof of concept implementation is available in the Qt labs under the name Flickable. The ideas in the latter example were used as a starting point for the simple kinetic scroller described in this article - QsKineticScroller. It can be used on any descendant of QAbstractScrollArea, including QScrollArea, QListView, QListWidget and QTreeView.

Algorithm

Note: the algorithm is designed for touch screen use with fingers, but Qt works with mouse cursors and mouse events. The algorithm is presented with Qt's terminology, with a list playing the role of scroll area.

Click & drag scrolling

Since kinetic scrolling can be viewed as the sum of two features, it can be implemented in two steps.

The first step is click & drag scrolling. It can be achieved by installing an event filter and intercepting mouse press, move and release events. When a press event is received the scrolling starts, when a move event is received the list is scrolled, and finally when a release event is received the scrolling stops. To avoid accidental clicks, all the events are blocked inside the filter function.

Consuming the mouse events is a necessary step that leads to an unpleasant problem: regular clicks are no longer registered by the target list. This issue can be avoided by making the algorithm guess when the user is clicking and dragging as opposed to when they're just clicking to select an item in the list. In the QsKineticScroller implementation a press & release event sequence is considered a click when it has less than five move events in between. Personal experiments have shown that a finger tap is much less precise than a pointer click, with one to four move events received between the time the finger is pressed on the screen and then lifted.

By the time the scroller has figured out that the user wanted to tap the screen, the events have already been consumed. To get around this last obstacle, the scroller records the screen position of the last press event and simulates a mouse click at that position.

Finally, by tracking mouse move events and updating the scrollbar position, the scroller makes the item list follow the user's finger. Implementing this first part of the algorithm allows Symbian Qt application users to scroll much easier than with scrollbars. The second step makes scrolling more visually interesting and easier to do, especially on longer lists.

Kinetic scrolling

For step two, the scroller continues to scroll the list automatically after the user has lifted their finger off the screen, gradually slows down and then stops. To display a pleasing effect, the scroller must decide how fast to scroll, how far to scroll and how fast to slow down.

A good starting point is „how fast to scroll". In physics velocity represents the direction in which and magnitude by which an object changes its position. Speed is another word for magnitude in this context. The „how fast to scroll" question can be answered by recording the cursor's drag velocity on the screen. A simple but imprecise way to do this is to poll the cursor position at specific time intervals; the difference in positions represents the speed (measured in pixels / timer interval) and the mathematical sign of the difference represents the direction. This algorithm will give a good enough idea on whether the cursors is moving fast or slow

and it is popular enough, since it can be found in other implementations such as Sacha Barber's Scrollable canvas.

Next up is „how far to scroll". How far is actually connected to how fast to slow down because the list is scrolled with a certain velocity and then it decelerates until it stops. Since the velocity has previously been established, the only thing left is to calculate the deceleration based on friction. In physics, kinetic friction is the resistance encountered when one body is moved in contact with another. Of course, there can be no friction between pixels, but kinetic scrolling is a simulation and one can pretend that the list items are moving over the list container and that this movement generates friction. In reality friction is calculated based on the nature of the materials, mass, gravitational force and so on. In the simulation a numeric value is used to alter the speed of scrolling. QsKineticScroller reduces the speed by a value of 1 at certain time intervals – a very simplified model indeed, but it works.

Having determined the deceleration, „how far" the list scrolls kinetically is simply a function of the time that it needs to reach a speed of zero.

= Implementation =

Note: comments have been removed and some of the code has been truncated and replaced with a [...] marker. The complete commented, working source code is attached at the end of this section.

Step-by-step source code description

QsKineticScroller is implemented as a stand-alone class and most of the implementation is hidden behind a d-pointer. The event filter function makes QObject inheritance necessary, but with a bit of work the event machinery can be moved inside the d-pointer to make the class implementation truly opaque.

```
class QsKineticScroller: public QObject
{
   [...]
protected:
   bool eventFilter(QObject* object, QEvent* event);
   [...]
private:
   QScopedPointer<QsKineticScrollerImpl> d;
};
```

Moving on to the cpp file, a few variables of interest can be seen at the top. The class user can experiment with these variables to influence the scrolling behavior. Indeed, the default values have also been chosen based on experimentation. As an example, changing the timer interval will affect the scrolling speed and smoothness, while changing the friction will influence the deceleration.

```
static const int gMaxIgnoredMouseMoves = 4;
static const int gTimerInterval = 30;
static const int gMaxDecelerationSpeed = 30;
static const int gFriction = 1;
```

The private implementation is aimed at hiding unneeded information from the compiler and component users. The `isMoving` and `isPressed` variables are the simplest way to keep track of what state the scroller is in. e.g: not moving, scrolling by finger and so on. Some implementations assign an explicit state and one can go as far as using the Qt state machine implementation. The rest of the variables are described at the point of use.

```
class QsKineticScrollerImpl
{
   [...]
   bool isPressed;
   bool isMoving;
   QPoint lastPressPoint;
   int lastMouseYPos;
```

```
    int lastScrollBarPosition;
    int velocity;
    int ignoredMouseMoves;
    int ignoredMouseActions;
    QTimer kineticTimer;
};
```

When installing the event filter, the important thing to notice is that it has to be installed for both the scroll area and its viewport. A scroll area is not a single item, and failing to install the filter on the viewport will result in not getting any mouse events at all.

```
void QsKineticScroller::enableKineticScrollFor(QAbstractScrollArea* scrollArea)
{
    [...]
    scrollArea->installEventFilter(this);
    scrollArea->viewport()->installEventFilter(this);
    d->scrollArea = scrollArea;
}
```

The largest part of the implementation lies inside the event filter. It is described in multiple blocks.

The first job of the filter is to make sure that it only works on mouse events, since the scroll area will also receive paint events, resize events and so on. When a mouse press is registered, the press point is stored in case it is needed later for a simulated click and the scroll bar position is stored so that it can be used when calculating by how much to scroll the list.

```
bool QsKineticScroller::eventFilter(QObject* object, QEvent* event)
{
    const QEvent::Type eventType = event->type();
    const bool isMouseAction = QEvent::MouseButtonPress == eventType
       || QEvent::MouseButtonRelease == eventType;
    const bool isMouseEvent = isMouseAction || QEvent::MouseMove == eventType;
    if( !isMouseEvent || !d->scrollArea )
      return false;
    [...]
    switch( eventType )
    {
    case QEvent::MouseButtonPress:
       {
          d->isPressed = true;
          d->lastPressPoint = mouseEvent->pos();
          d->lastScrollBarPosition = d->scrollArea->verticalScrollBar()->value();
          [...]
```

This code block does the click versus click & drag differentiation. If it weren't for it, the scroller would ignore legitimate clicks. Once the scroller has established that the user is indeed dragging on the screen, it starts a timer that calculates the drag speed. `lastMouseYPos` will be used later in the speed calculation.

```
    case QEvent::MouseMove:
       {
          if( !d->isMoving )
          {
             if( d->ignoredMouseMoves < gMaxIgnoredMouseMoves )
                ++d->ignoredMouseMoves;
             else
             {
                d->ignoredMouseMoves = 0;
```

```
                d->isMoving = true;
                d->lastMouseYPos = mouseEvent->pos().y();
                if( !d->kineticTimer.isActive() )
                   d->kineticTimer.start(gTimerInterval);
             }
          }
         [...]
```

When the user lifts their finger off the screen a mouse release event shall be received. This is where the click versus drag differentiation makes a difference: `d->isMoving` will be false for clicks, but true for drags. The simulated click will be swallowed by the next filter call unless the filter is told to ignore it.

```
   case QEvent::MouseButtonRelease:
      {
         [...]
         if( !d->isMoving )
         {
            QMouseEvent* mousePress = new QMouseEvent(QEvent::MouseButtonPress,
               d->lastPressPoint, Qt::LeftButton, Qt::LeftButton, Qt::NoModifier);
            QMouseEvent* mouseRelease = new QMouseEvent(QEvent::MouseButtonRelease,
               d->lastPressPoint, Qt::LeftButton, Qt::LeftButton, Qt::NoModifier);

            d->ignoredMouseActions = 2;
            QApplication::postEvent(object, mousePress);
            QApplication::postEvent(object, mouseRelease);
       [...]
}
```

As mentioned in the algorithm description, both speed calculation and deceleration are done at certain time intervals. This implementation uses a single timer for both. The kinetic scrolling happens in the else branch: since the speed measurement is imprecise it is restricted to a more reasonable value at first. Then it is adjusted by the friction value until it reaches a minimum boundary, which means that kinetic scrolling should stop.

```
void QsKineticScroller::onKineticTimerElapsed()
{
   if( d->isPressed && d->isMoving )
   {
      const int cursorYPos = d->scrollArea->mapFromGlobal(QCursor::pos()).y();
      d->velocity= cursorYPos - d->lastMouseYPos;
      d->lastMouseYPos = cursorYPos;
   }
   else if( !d->isPressed && d->isMoving )
   {
      d->velocity = qBound(-gMaxDecelerationSpeed, d->velocity, gMaxDecelerationSpeed);
      if( d->velocity> 0 )
         d->velocity -= gFriction;
      else if( d->velocity < 0 )
         d->velocity += gFriction;
      if( qAbs(d->velocity) < qAbs(gFriction) )
          d->stopMotion();

      const int scrollBarYPos = d->scrollArea->verticalScrollBar()->value();
      d->scrollArea->verticalScrollBar()->setValue(scrollBarYPos - d->velocity);
   }
   else
```

```
        d->stopMotion();
}
```

Download and use guide

The source code can be downloaded from the following link - File:QsKineticScroller.zip. This component is licensed under a BSD license and can be used both for commercial and Free software.

Using QsKineticScroller is easy:

1. Unzip the downloaded file.
2. Add the unzipped cpp and h files to a Qt pro file.
3. Create a scroller object instance. It's a good idea to make the scroller a child of the dialog or window that also contains the target scroll area.
4. Call the enableKineticScrollFor function with the target scroll area as a parameter.
5. If the scroll area is a list view or list widget, it must have the scroll mode set to ScrollPerPixel.

That's it, the scroller will take care of everything else.

Future improvements

It is becoming obvious that this type of scrolling is turning into a mandatory feature, to the point that reviews mention the lack of kinetic scrolling as a minus. QsKineticScroller is a stopgap solution that can be used until the Qt team will publish the official component.

As previously mentioned, bounce and overshoot can make kinetic scrolling more realistic and even more fun – giving a physical feel to the pixels on the screen. Easing curves and the property animator should make the deceleration smoother. All of these make logical candidates for future improvements, which will be discussed in the following version of this article.