St.-Petersburg and Moscow Metro Map for Mobile Devices With Touch Screen

Nikita Karpinsky, Evgeny Linsky, Alexander Malakhov
State University of Aerospace Instrumentation
Saint-Petersburg, Russia
nikita.karpinsky@gmail.com,
{evlinsky, admalakhov}@vu.spb.ru

Abstract

This article is devoted to interactive metro map application which calculates the way between two chosen stations. It describes structure of the program and input data format so that users can add their own maps for different cities. Main emphasis is on implementing map for dragging and zooming it quickly.

I. Introduction

This application implements interactive metro map and is called Underground. It provides an opportunity to get travel time and the shortest way between two chosen stations.

It is developed in Qt for Symbianˆ1 and MeeGo. By the time development began there was:

- no such an application for MeeGo at all;
- one application in Java which was too slow for Symbianˆ1 (Yandex.Metro).

Initial requirements for the app are:

- MeeGo and Symbianˆ1 support.
- Fast calculating travel time and the shortest way between stations with an opportunity to switch similar routes.
- Dragging and zooming implementation.
- Pinch-to-zoom for MeeGo support.
- GPS for finding the closest station support.
- Storing maps in XML files for opportunity to add new maps.
- UI is designed in Qt-Quick and program logic is in C++.

II. Implementation details

A. Choosing technologies

Qt was chosen because it allows to implement UI once for both Maemo and Symbian. Program logic is written in C++ and UI is in Qt-Quick to make development faster. XML was chosen because it is self-describing. It allows users to add new maps on their own.

B. Program structure

1) Program logic part: All program logic is in C++. Main class for logic is *Model* class. There is one *Model* for each map. First of all, constructor of *Model* calls *parseXMLdata* function which is implemented in *xmlParser* class to get all input data. Besides providing functions to get input data, *Model* also provides functions such as:

[0]With the financial support of the Ministry of Education and Science of the Russion Federation

Fig. 1. Route on the map

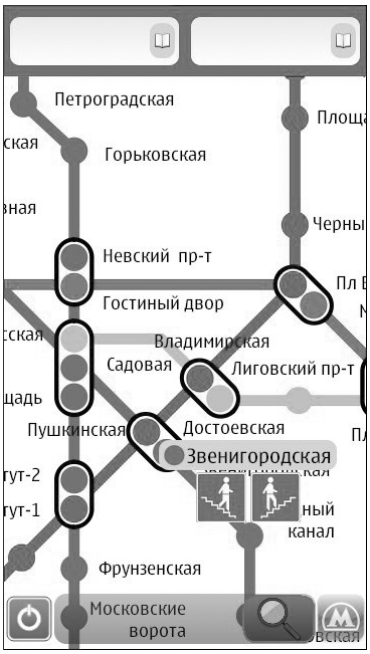

Fig. 2. Choosing station

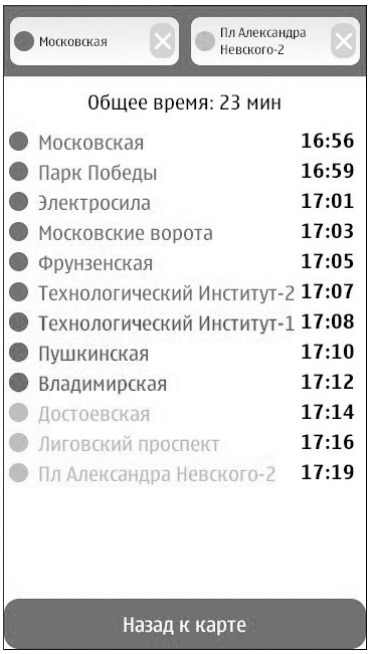

Fig. 3. Stations list from the route

- *getClickedStation* which gets clicked screen coordinates and returns clicked station if there is one with such coordinates;
- *getRoute* which gets two stations, then returns a route between them.

To switch models there is *ModelChooser* class which implements all the *Model* functions. If a *ModelChooser* function is called then it is redirected to current map *Model*.

2) UI part: All UI is in QML except map implementation which is in C++. There is Main.qml file which contains all the biggest parts of UI:

- Two buttons for opening station lists in the top part of the screen where stations can be chosen (*UpFields.qml*).
- Slider for zooming map (*Slider.qml*).
- Map which can be dragged and zoomed (Map is implemented in *MainWidget* class).
- Route list which consists of stations in calculated route (*RouteList.qml*).

There are some states for Main.qml:

- ”mainWin”
  Map without route is shown;
- ”stationList1” and ”stationList2”
  List for chosing first or last station is opened;
- ”routeList”
  There is a calculated route and stations list from this route is opened;
- routeMap
  There is a calculated route. Map is transparent except the stations and edges which are used in this route.

3) Connections between Model and UI part: All connections between Model and UI part are implemented in JavaScript. Data for Stations and Edges is stored in QVariantLists of QVariantMaps so that it can be read in both C++ and JavaScript.

C. Functionality description

In the Fig. 1 state is ”routeMap”.

In the Fig. 2 state is ”mainWin”. Dialog for chosing station is shown. User can choose whether the clicked station is first or last in the route.

In the Fig. 3 state is ”routeList”. Times for each station are absolute.

D. Implementing Map

The biggest problem of the development was how to make dragging and zooming work fast for Symbianˆ1. There were some ideas which appeared to work bad.

1) Map is a single image
   There is a map image which is loaded in QML using Image element and is contained in Flickable. When map is scaled the image is zoomed.
   The biggest disadvantage of this method is that dragging is very slow because of the way Qt-Quick stores an image. If map is scaled, default image is taken and zoomed every time it should be repainted while dragging.
   Here is simplified code:

```
Flickable {
    id: flickable
    function recount() {
        map.height = main.height * slider.value
        map.width = main.width * slider.value
    }

    Image {
        id: map
        source: ”Icons/spb.svg”
    }
}
```

2) There are some different-sized images
   To get rid of lack of the previous method image size is constant so dragging becomes fast. To provide scaling there are several different-sized map images which replace each other according to current scale value.
   The disadvantage of this method is that it takes a lot of memory because images are not unloaded properly in Qt-Quick while replacing each other. We tried a lot of ways to solve this problem such as using *Loader* elements, a lot of different combinations of using *Image* elements but it wasn`t fixed. Also scaling becomes discrete and it looks awful for pinch-to-zoom.
   Here is simplified code:

```
Flickable {
    id: flickable
    function recount() {
         map.source=”Icons/spb” + slider.value + ”.svg”
    }

    Image {
        id: map
        source: ”Icons/spb0.svg”
    }
}
```

3) Visible part of the map is drawn in C++
   As all attempts to implement map using only Qt-Quick collapsed map is rendered using C++. Map is a vector image which is described in XML-file and is drawn by *QPainter* draw functions. Only elements which are shown on the screen are drawn.

As the result we get *QDeclarativeItem* which is loaded in QML instead of images in previous methods.
Dragging works very slow because of big amount of lines, stations circles, station names (the most slowly part), etc. which should be drawn while dragging. (For Moscow it is 185 station circles and names and about 250 lines between stations)
Here is simplified code:

```
void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget
    *widget)
{
        QRectF curRect = boundingRect ();

        for (int i = 0; i < allEdges.count(); ++i) {
            QVariantMap map = allEdges.at(i).toMap();
            if(curRect.contains(map["x1"] * scale, map["y1"] * scale) ||
                curRect.contains(map["x2"] * scale, map["y2"] * scale)){
                QPen pen(QColor(map["color"]), penWidth * scale, Qt::SolidLine,
                    Qt::RoundCap, Qt::RoundJoin);
                painter->setPen(pen);
                painter->drawLine(map["x1"] * scale, map["y1"] * scale, map["x2"] *
                    scale, map["y2"] * scale);
            }
        }
        for (int i = 0; i < allStations.count(); ++i) {
            QVariantMap map = allStations.at(i).toMap();
            if(curRect.contains(map["xCenter"] * scale,(map["yCenter"] * scale)){
                painter->setBrush(QColor(map["stationColor"]));
                painter->drawEllipse(map["xCenter"] * scale, map["yCenter"] * scale,
                    cirlceSize * scale, cirlceSize * scale);
            }
        }
}
```

4) Map is drawn in C++ once for each scale
Map is a *QPixmap* which is created every time map is scaled. Creating *QPixmap* is a function which is similar to *paint* function from above. There is only one difference: we should draw all the map, not only it's visible part (*createPixmap* function). When map is dragged, a fragment of this pixmap is copied using the *QPainter* of the *paint* function.
To provide fast rendering while scaling there is a pixmap copy (*smallPixmap*). While scaling, *smallPixmap* is being zoomed so that it is quite fast although the quality isn't high (*createFastScaledPixmap* function). It allows to solve our problem and dragging, zooming and pinch-to-zooming work fine.
The *paint* function is so fast that we can consider that rendering time of the map depends only on QML rendering, not on C++ part.
Here is simplified code:

```
void createPixmap()
{
        pixmap = new QPixmap(width * scale, height * scale);
        QPainter* painter = new QPainter(pixmap);

        for (int i = 0; i < allEdges.count(); ++i) {
            QVariantMap map = allEdges.at(i).toMap();
            QPen pen(QColor(map["color"]), penWidth * scale, Qt::SolidLine,
                Qt::RoundCap, Qt::RoundJoin);
            painter->setPen(pen);
            painter->drawLine(map["x1"] * scale, map["y1"] * scale, map["x2"] *
                scale, map["y2"] * scale);
        }
        for (int i = 0; i < allStations.count(); ++i) {
            QVariantMap map = allStations.at(i).toMap();
            painter->setBrush(QColor(map["stationColor"]));
```

```
15              painter->drawEllipse(map["xCenter"] * scale, map["yCenter"] * scale,
                    cirlceSize * scale, cirlceSize * scale);
16          }
17  }
18  void MainWidget::createFastScaledPixmap() {
19
20      pixmap = new QPixmap(width * scale, height * scale);
21      QPainter* painter = new QPainter(pixmap);
22      QPainter::PixmapFragment fragment;
23      //Setting parameters to the fragment is skipped
24      painter->drawPixmapFragments(&fragment,1, *smallPixmap, QPainter::OpaqueHint);
25      update();
26  }
27  void MainWidget::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
        QWidget *widget)
28  {
29      QRectF curRect = boundingRect();
30      painter->drawPixmap(curRect, *pixmap, curRect);
31  }
```

5) Using Qt SVG
   This is an attempt to improve *createPixmap* function.
   First of all, an SVG-image was created using *createPixmap* from above where *QPixmap* is replaced with *QSvgGenerator*. Then if there is no route on the map only SVG is shown. If there is a route it is drawn over this SVG.
   However it appeared that it works several times more slowly than previous method.
6) QGraphicsScene and QGraphicsView
   We didn't use *QGraphicsView* and *QGraphicsScene* because initial requirements were to use Qt-Quick for all visible parts of the app and to use C++ for program logic. After we realized that vector graphic was the only way for implementing we didn't use *QGraphicsView* because it meant that we had to change all the program structure (Instead of having main class in Qt-Quick we should have used *QGraphicsScene* to load interface and create new main class to control the map).

Rendering times

All the times were obtained using QML profiler and time.h functions for C++ on PC and Moscow map was used (times for QML are very inaccurate).
(X + Y) ms means rendering time in QML + rendering time in C++.

TABLE I
RENDERING TIMES

Method	Dragging	Scaling
Map is a single image	(10 + 0)ms	(10 + 0)ms
There are some different-sized images	(4 + 0)ms	(6 + 0)ms
Visible part of the map is drawn in C++	(3 + 20)ms	(3 + 20)ms
Map is drawn in C++ once for each scale	(3 + 0.5)ms	(3 + 60)ms
Using Qt SVG	(3 + 0.5)ms	(3 + 230)ms

E. Platforms features

As the app was developed in Qt and Qt-Quick all the version differences for Symbian and MeeGo are because of sreen size and MeeGo supporting pinch-to-zoom. Some QML files are different for not changing code for compiling. There are different main files (Main_S60.qml and Main_MeeGo.qml) and different Flickables which hold the map (MapFlickable_MeeGo.qml and MapFlickable_S60.qml)

III. Conclusion

In this article interactive metro map was described. It appeared that for this application custom-made engine works faster than Qt and Qt-Quick engines.
This open source application is called Underground and can be found here https://projects.developer.nokia.com/icthub/browser#underground.