ECE155: Engineering Design with Embedded Systems Winter 2013

Lab 4 (Navigation) — Weeks 9, 11

Kirill Morozov *version 1*

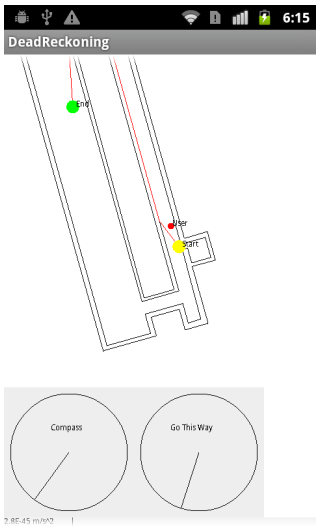

Figure 1: A complete lab 4 implementation.

Deadline: You must submit the lab to the SVN repository and be prepared to demonstrate Lab 4 to a TA by the midpoint of your second assigned Lab 4 session. The best way to demo is during a lab session, but any earlier time where you can convince a TA to watch is OK too. **Because March 29 is a holiday, the Friday lab groups must demo the lab by Thursday, March 28th; we will schedule demo sessions on Thursday.**

1 Objective

The goal of this lab is to make conclusions about the real world based on sensor data. In particular, you will create an application that reads map information and guides a user through the corresponding physical environment. You will build this application on top of the dead reckoning application from lab 3.

During this lab, you will:

1. Track the user's position on a model of the physical world.
2. Account for errors in sensor readings based on knowledge of the physical world.
3. Implement a path-finding algorithm which can guide a user to a destination, correcting the user's wrong turns along the way.

2 Interacting With the Real World

For this lab, we will provide you with the `MapView` Java class. The `MapView` enables the user to select a starting point and a destination on an interactive map. We will also provide you with map files of the testing area (our lab room, E2-2364) ready for use by the `MapView`.

You can find the complete `mapper` package in the SVN repository under `materials/mapper`. Copy this package into your project. The `MapView` provides enough functionality to complete lab 4. However, if you would like to modify the mapper to make it prettier or to add additional functionality, go ahead! (For instance, you could implement multitouch functionality to zoom your map.)

Setting up the MapView. The `MapView` is an Android `View`, like the `LineGraphView` from Lab 1. Thus, you can use the techniques from Lab 1 to add the `MapView` to your applications.

Here's what the `MapView` needs. The `MapView`'s constructor takes five parameters. The first, the application context, is common to all `Views`. The second and third parameters are the width and height of the mapper in pixels. The fourth and fifth parameters are the X and Y scale that should be used to display the map. The scale is represented as pixels displayed per meter represented. As an example:

```
MapView mv = new MapView(getApplicationContext(), 400, 400, 60, 60)
```

You will need to tune the parameters so that the map fits on your phone screen.

The `MapView` comes with a context menu; add the following lines to your application to use it:

```
// In your Activity's onCreate() method:
registerForContextMenu(mv);

// In your Activity:
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);
  mv.onCreateContextMenu(menu, v, menuInfo);
}

@Override
public boolean onContextItemSelected(MenuItem item) {
  return super.onContextItemSelected(item) || mapView.onContextItemSelected(item);
}
```

Loading a map. The `MapView` accepts SVG (Scalable Vector Graphics) files. SVG is a file format for storing vector-based graphics as XML. You can create your own SVG files with the open-source program Inkscape[1]. The mapper understands a limited subset of SVG: it only understands paths, which it interprets as walls. The mapper also looks for "xScale" and "yScale" attributes in the SVG file's top-level tag, which are interpreted as the scale, in meters per pixel, of the map. By default, the `MapLoader` assumes a scale of 0.01 for both of these values.

You can load a new map like this:

```
NavigationalMap map = MapLoader.loadMap(getExternalFilesDir(null), "Lab-room-peninsula.svg");
mapView.setMap(map);
```

For the map to be found by the loader, you will need to place the SVG file on your phone's SD card in the directory `<SDCard>/Android/data/<yourpackagename>/files`[2].

[1] `http://inkscape.org/download/`

[2] The AirDroid app is a good way to put files onto your phone.

Interacting with the MapView. The `MapView` informs your application when the user changes location or destination, through callbacks. To receive the callbacks, you need to provide a class (possibly your `MainActivity`) implementing the `PositionListener` interface. Call `addListener()` on your `MapView` with a `PositionListener`. The `MapView` will call the `originChanged()` method on your `PositionListener` when the user selects a new starting location, and it will call the `destinationChanged()` method when the user selects a new destination.

The `MapView` also has `setUserPoint()` method, which draws a red dot at the given location on the map. You will want to call `setUserPoint()` immediately whenever `originChanged()` is called. Similarly, the `MapView` has a method `setUserPath()`, which takes a list of `PointF` objects. It draws this path on the screen. I recommend using that to show your current heading, that is, which way your next step will take you on the map.

Utility functions. Three useful classes are the `NavigationalMap` class, the `VectorUtils` class, and the `LineSegment` class. `VectorUtils` should be familiar to you from high-school linear algebra; I used `angleBetween` to compute angles between vectors. `LineSegment` also contains potentially useful methods, although I didn't use any of them directly. `NavigationalMap` contains one particularly useful method, `calculateIntersections()`, which returns a list of walls between two `PointF`s.

When comparing numbers which come from the real world and are potentially jittery, you'll often want to just see if they are "close enough", by seeing if their difference is less than a given tolerance. It's good design to use a static final field to specify these tolerances. Locations and angles are numbers that often need to only be "close enough".

Keep in mind that our classes operate only in meters. All values should be in meters.

3 Background

For this lab, you will need to calculate a route between two points in the presence of obstacles. You do not need to come up with an algorithm from scratch, and may use an existing design, as long as you reference your sources.

There are a number of simple-to-implement ways to solve mazes. We'll start with a simple heuristic and then present two potential full solutions.

Simple heuristic. Here's a heuristic that should work well enough.

- is there a path to the destination free of walls? (`calculateIntersections()` is your friend here!) If so, take it.
- no path—then navigate to the first wall between you and the destination, and arbitrarily pick an end of the wall to walk towards. When the user reaches that end of the wall, your algorithm will calculate something different.

Such an algorithm should give you full marks in this grading scheme, but is not a complete solution; it gets stuck in corners. You'll get a bonus mark if you implement a solution that never gets stuck.

Wall-following algorithm. You can implement a wall-following algorithm. As long as every wall in the maze touches the edges of the maze, you can reach the exit from the start by walking forward and keeping your hand on the left (or right) wall.

Converting maps into graphs. Physical environments (like rooms) tend to be reasonably simple, consisting of mostly-straight lines meeting at right angles. You can use this idea to break up the map into a grid. Each square of the grid becomes a node in the graph. Nodes are connected to adjacent nodes as long as there is no intervening wall.

You will find that a grid creates many graph nodes which are connected to all of their neighbours. With a bit of cleverness, you could remove these nodes or not generate them in the first place.

Magnetic North. Be aware that a compass does not point to the north pole ("True North"), but rather to the magnetic north pole. This is the location on Earth where all magnetic field lines point vertically downwards. The maps we provide for use with the mapper will be arranged such that the vertical axis of the map aligns with magnetic north. We mention this because, if you make your own maps, you will need to account for the discrepancy between magnetic and true north.

The way to convert magnetic north into true north is via "magnetic declination." This is the angle between true north and magnetic north. You can consult `http://magnetic-declination.com/` to find the magnetic declination of any point on earth. The magnetic declination at Waterloo, Ontario is -9° 41'.

Adjusting for the real world. Due to user error and sensor error, any dead reckoning device will slowly drift off course. To correct for this, you can use what you know about the world to correct your calculations. For example, if the algorithm thinks the user just took a step through a wall, which is of course not possible, it should silently drop the step.

You can detect a wall using `calculateIntersections()` and just a bit of cleverness: calculate the coordinates of the prospective step and see if there are any walls between the current coordinates and the prospective coordinates.

Over-adjusting for the real world. You will find that aggressive techniques for taking the real world into account will sometimes cause you to "get stuck" on a wall. For example, in the real world, the user may turn a corner, but your application may think that the user was actually two meters behind that. Correcting for this is tricky, so you are not required to do it for this lab.

4 Demonstration

Same as with all the previous labs. Don't plagiarise. Again, the Friday lab section must demonstrate by the end of Thursday.

Requirements. Your app must:

1. Display the current location of the phone in the `MapView` view, updating after each step.
2. Announce when the destination is reached.
3. Display instructions for the user to reach the destination point selected using the `MapView`. These instructions may take the form of orders (e.g. "4 steps North and 1 step East"—show which way is north!), or they may take the form of a direction and distance (e.g. "Walk forward"; "Turn left by 10 degrees".) It may be easiest to have a `TextView` which shows the next instruction.

We will measure your app on up to three routes, each with between 50 and 100 steps, and give points based on what your algorithm can do. First, we will test your application on a direct route, free of obstacles, from the origin to the destination. You can earn 2 points for this trial by 1) updating the position on the map and 2) pointing the user in the right direction. Next, we will test the behaviour of your application in going around an obstacle. You can earn 1 point on this trial simply by preventing the user position from being inside a wall—the TA will simulate a step through a wall and observe the app's behavious. You can earn 1 additional point by navigating the user along an indirect route which reaches the destination. Finally, on the third trial, the TA will walk in the wrong direction on an indirect route for a while. If your application correctly handles this situation and directs the user to the right destination, you will earn the 1 final mark on the lab, for full marks. In all cases, your application should tell the user when to stop upon reaching the destination.

We will also give a bonus mark for implementing a more robust navigational algorithm than the one that I've described. If you tell us that you'd like this mark, we will carry out a fourth trial where we test your application with a more complicated map and a route which will foil the simple algorithm. If it works, you'll get an extra mark.

It is OK if your application tells the user to cut through corners (but not walls).

For this lab, you will need to convert the number of steps that your pedometer detects into a distance. In our experience this is non-trivial, so you may use one of the following options to make it easier for you, in order of increasing coolness:

- You may hard-code a value. We will provide you with the step lengths for the teaching assistants who will be testing your application.
- You may add a UI widget where the user may enter a step-length.
- You may add a calibration mode to your device where you ask the user to walk some number of meters and then count how many steps that takes.
- You may calculate distance travelled from accelerometer readings.

We have provided you with two different maps of the testing area. One has "peninsulas" instead of the tables in the centre of the room—you will find that the wall following algorithm (but not the simple heuristic) is fooled by free standing obstacles. If you can find a way to overcome this problem, you will get a bonus mark. The provided map "`Lab-room-bonus-destination-and-start.svg`" shows you two example points that fool basic wall following.

5 Submission of project files

Upload the version of your code used in the demo to your subversion repository and commit it. The address is: `https://ecesvn.uwaterloo.ca/courses/ece155/w13/groups/group-NNN-MM`

Email Patrick Lam if you can't commit to that address.

6 Tips and hints

Compass not working? Make sure that your device isn't in an interfering case.

You'll need to make sure that your app has at least the permission:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

or your app will just crash with a `NullPointerException`. It will also throw a `RuntimeException` if there are no maps in the directory. You may want to give the app `WRITE_EXTERNAL_STORAGE`; that way, it will create the appropriate directory for you, and then you can just put the file into that directory.

If you have points moving when they shouldn't ("Why is my origin point moving around?"), make sure that you're creating a fresh `PointF` object instead of just copying a reference, e.g.
`originPoint = new PointF(op.x, op.y);`

You'll find that it will be much easier to develop and debug your app if you include a button which simulates a step instead of having to take an actual step. You may want to disable step detection during development and only enable it for final testing.

7 Marking scheme

Here's the marking scheme I intend to use. 1 point for each of the following items:

- updates position on map following steps;
- finds and presents direct route to destination;
- does not walk through walls;
- finds and presents indirect route to destination (going around one obstacle); and
- compensates for user errors while walking on an indirect route.

There's also a bonus mark. We've simplified the default map. If you use the complex map (`Lab-room.svg` in the repository) and correctly solve all of navigational challenges without getting stuck in corners, you will get a bonus mark.

We'll also give a bonus mark for implementing multitouch to zoom your map in and out.