Fall Detection with Android

Ubiquitous Computing – 2010-2011

Samuel Esposito & Grayson Kleine

1 Introduction

In this document we discuss the Android based fall detection software we built for the Ubiquitous Computing course at the University of Groningen. The intent of the project was to create an Android application that would detect a person's fall while carrying an Android smartphone. The application should cleverly process the smartphone's sensor data in order to distinguish a dangerous fall from other daily activities that involve sudden movements, such as sitting down or running. Unless the user cancels the application within 10 seconds after a fall was detected, it should send out a call for aid.

In the next section we briefly discuss a set of algorithms for the calculation of fall features that are indispensable to sensitive and specific fall detection. After that we discuss our implementation of the fall detection software on Android and document the steps to take in order to deploy it, including simulation. Finally we evaluate the project and discuss our main results.

2 Fall Features

There is plenty of literature describing fall features that can be calculated from raw sensor data. We chose to use the article on fall detection of A. K. Bourke et. *al.*[1] as a base for this project because it gives an up to date overview of the state of the art and compares different algorithms for sensitivity (detecting a fall when it occurs) and specificity (discerning a fall from other activities). The results of their research is that the combination of three features yields the best results in terms of sensitivity and specificity: vertical velocity, fall impact and posture. We discuss each of these features separately below.

The fall impact is the most commonly used fall feature as it is most easy to derive from the raw data of a tri-axial accelerometer. It is calculated as the root sum of squares of the acceleration on each axis, from now on referred to as RSS:

$$RSS = \sqrt{x^2 + y^2 + z^2}$$

The RSS gives an indication of the forces that act on the smartphone during impact measured in g (gravity). The harder a person falls, the bigger the g-forces on the smartphone will be and the higher the RSS value is. Based on their findings and the consensus in the field of fall detection, Bourke et. *al.* put forward a threshold of $2.8g$ for the RSS value: every value higher than this threshold is considered as a fall feature.

Another less straightforward feature of a fall is the vertical velocity. When a person makes a free fall, the g-force on the smartphone is compensated by the downward acceleration for a small period of time: the smartphone experiences zero-gravity. This fall feature, henceforth referred to as VVE (vertical velocity), is calculated by numerically integrating the difference of the RSS with $1g$ (9.80665) for a small time window ($\Delta T = 0.6s$):

$$VVE = \int_{\Delta T} (RSS(t) - 1g)\,\mathrm{d}t$$

[1] Assessment of waist-worn tri-axial accelerometer based fall-detection algorithms using continuous unsupervised activities. Bourke AK, van de Ven P, Gamble M, O'Connor R, Murphy K, Bogan E, McQuade E, Finucane P, Olaighin G, Nelson J. Conf Proc IEEE Eng Med Biol Soc. 2010;2010:2782-5. PMID: 21095967 [PubMed - indexed for MEDLINE]

The higher the vertical velocity of the smartphone, the lower the value of the VVE feature. Bourke et. *al.* determined that a fall should be detected when the value of VVE drops below -0.7.

Finally there is the human posture to take into account. When a person makes a dangerous fall for which aid is needed, he or she is expected to remain in a horizontal position for some time after the fall. When the smartphone is carried in a pocket, it should also maintain this horizontal position. A horizontal position in the context of fall detection is defined as follows: the smartphone must have an angle between 0° and 30° from the horizon for 75% of the time during a 2 second interval 1 second after an RSS or VVE feature was detected.

As mentioned above, Bourke et. textital. showed in their article that the best results are obtained when the combination of vertical velocity, fall impact and posture is used for detecting a fall. More specifically the detection of a fall should be marked by either a VVE feature followed by a posture feature or an RSS feature followed by a posture feature.

Now that the theoretical details behind fall detection are clear we will shift focus to the implementation of the above algorithms on Android.

3 Fall Detection on Android

In this section we discuss the implementation of the fall detection application for the Android platform. First we explain how to retrieve sensor data on an Android smartphone and implement the algorithms for calculating the different fall features[2] from this raw data. Next we handle sending a fall notification to a server and allowing the user to cancel the sending within 10 seconds. Finally we wrap this all together into an Android activity.

3.1 Calculating Fall Features

To get access to sensor data on the Android platform one has to create a class implementing the 'SensorEventListener' interface and register it with the android 'SensorManager' as shown in the listing below.

```java
public class FallDetector implements
    SensorEventListener {

  private SensorManager mSensorManager =
    (SensorManager) activity.getSystemService(activity.SENSOR_SERVICE);

  public void registerListeners() {
    mSensorManager.registerListener(this,
        mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
        SensorManager.SENSOR_DELAY_UI);
    mSensorManager.registerListener(this,
        mSensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION),
        SensorManager.SENSOR_DELAY_UI);
  }
  ...
}
```

[2] VVE (vertical velocity), RSS (fall impact) and posture (See chapter 2: Fall Features).

By implementing the 'onSensorChanged' method the sensor data can be captured and processed. We are particularly interested in the accelerometer data for calculating the RSS and VVE features and the orientation sensor for deriving the posture feature. The listing below shows how to filter for the right type of sensor and access the available data.

```
@Override
public void onSensorChanged(SensorEvent event) {
  synchronized (this) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
      event.values[0]; // Acceleration minus Gx on the x-axis
      event.values[1]; // Acceleration minus Gy on the y-axis
      event.values[2]; // Acceleration minus Gz on the z-axis
    } else if (event.sensor.getType() == Sensor.TYPE_ORIENTATION) {
      event.values[0]; // Azimuth, angle between the magnetic north direction and the
                       // y-axis, around the z-axis (0 to 359).
                       // 0=North, 90=East, 180=South, 270=West
      event.values[1]; // Pitch, rotation around x-axis (-180 to 180)
      event.values[2]; // Roll, rotation around y-axis (-90 to 90)
    }
  }
}
```

The RSS feature is calculated by taking the root sum of squares of the accelerometer event values. When the values exceeds the threshold ($2.8g$), an RSS feature is detected. This is implemented below.

```
protected final float RssTreshold = 2.8f;
...
float rss = (float) Math.sqrt(Math.pow(event.values[0], 2)
                            + Math.pow(event.values[1], 2)
                            + Math.pow(event.values[2], 2));
if (rss > RssTreshold * SensorManager.STANDARD_GRAVITY) {
  // Rss feature detected!
}
```

The VVE feature in turn is calculated by taking the numerical integer[3] of the RSS minus $1g$ in a time window of $0.6s$. When this value goes below the VVE threshold ($-0.7g$), a VVE feature is detected. This is implemented in the listing below.

```
private float mRssValues[] = new float[256];
private int mRssCount = 0;
private int mRssIndex = 0;
private long RssStartTime = 0;
protected final float VveWindow = 0.6f;
protected final float VveTreshold = -0.7f;
...
// Store all RSS values in the window in a circular array
if (RssStartTime == 0) {
  RssStartTime = date.getTime();
  mRssCount++;
} else if (date.getTime() - RssStartTime <= VveWindow * 1000
    && mRssCount < mRssValues.length) {
  mRssIndex = mRssCount++;
} else {
  mRssIndex = ++mRssIndex % mRssCount;
}
mRssValues[mRssIndex] = rss
    - SensorManager.STANDARD_GRAVITY;
// Calculate the numerical integer over all stored RSS values
float vve = 0;
for (int i = 0; i < mRssCount; i++) {
  vve += mRssValues[i];
}
vve = (vve * VveWindow) / mRssCount;
if (vve < VveTreshold * SensorManager.STANDARD_GRAVITY) {
  // Vve feature detected!
}
```

[3] The numerical integer is approximated using Simpson's rule (2^{nd} order polynomial).
Source: http://en.wikipedia.org/wiki/Simpson%27s_rule

One second after either a VVE feature or an RSS feature was detected, posture data is collected in a 2 second time window in order to detect the posture feature. The posture data itself is derived from the orientation sensor pitch (see above). When more than 75% of the collected posture data in the window has an angle smaller than 30°, the posture feature has been detected.

```
private final int OriOffset = 1000;
private final int OriWindow = 2000;
private long OriStartTime = 0;
protected final float OriTreshold = 60;
private final float OriConstraint = 0.75f;
private float OriValues[] = new float[256];
private int ori_index = 0;
...
// Calculate orientation wrt horizon
float ori = (90 - Math.abs(event.values[1]));
// Wait one second
long wait_interval = (activity.RssTime != 0 ? date
    .getTime() - activity.RssTime
    : (activity.VveTime != 0 ? date.getTime()
        - activity.VveTime : 0));
if (wait_interval >= OriOffset) {
  // Collect ori values for 2 seconds
  if (OriStartTime == 0)
    OriStartTime = date.getTime();
  else if (date.getTime() - OriStartTime < OriWindow) {
    if (ori_index < OriValues.length)
      OriValues[ori_index++] = ori;
  } else {
    // Calculate percentage above threshold
    int count = 0;
    for (int i = 0; i < ori_index; i++) {
      if (OriValues[i] > OriTreshold)
        count++;
    }
    if (count / ori_index >= OriConstraint) {
       // Posture feature detected
    }
  }
}
```

When either a VVE feature or an RSS feature has been detected followed by the detection of a posture feature, a fall has been detected. The fall is handled by sending a call for aid to a server, which is described in the next section.

3.2 RESTfull Fall Notification

For every fall that has been detected, a fall notification is sent to a RESTfull web service. Before sending the notification, a progress dialog is shown offering the user the opportunity to cancel it within 10 seconds. With every fall notification the current user location, if available, is included in the message. Below the code for accessing the smartphone's location is listed.

```
public class LocationUpdateHandler implements LocationListener {

  private FallActivity activity;
  protected LocationManager locationManager;

  public LocationUpdateHandler(FallActivity activity) {
    this.activity = activity;
    // Get location manager
    locationManager = (LocationManager) activity
        .getSystemService(Context.LOCATION_SERVICE);
    // Request location updates
    locationManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER, 0, 0,
        activity.locationUpdateHandler);
  }

  // Handle location updates
  public void onLocationChanged(Location loc) {
    synchronized (this) {
      activity.latitude = loc.getLatitude();
      activity.longintude = loc.getLongitude();
    }
  }
}
```

Beneath the actual fall notification is compiled and sent.

```
// Making an HTTP post request and reading out the response
HttpClient httpclient = new DefaultHttpClient();
httpclient.getParams().setParameter(
    CoreConnectionPNames.CONNECTION_TIMEOUT, 10000);
HttpPost httppost = new HttpPost("http://web.service.host/falls");
List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(2);
// Set fall timestamp
nameValuePairs.add(new BasicNameValuePair("datetime",
    (activity.VveTime != 0 ? Long.toString(activity.VveTime)
        : (activity.RssTime != 0 ? Long
            .toString(activity.RssTime) : "")))));
// Set RSS feature
nameValuePairs.add(new BasicNameValuePair("rss",
    (activity.RssVal == 0 ? "" : Float.toString(activity.RssVal))));
// Set VVE feature
nameValuePairs.add(new BasicNameValuePair("vve",
    (activity.VveVal == 0 ? "" : Float.toString(activity.VveVal))));
// Set user location
nameValuePairs
    .add(new BasicNameValuePair("lat", Double.toString(latitude)));
nameValuePairs
    .add(new BasicNameValuePair("lon", Double.toString(longitude)));
try {
  httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
} catch (UnsupportedEncodingException e) {
  ...
}
// Send the notification
HttpResponse response;
try {
  response = httpclient.execute(httppost);
} catch (Exception e) {
  ...
}
```

3.3 Android Activity

All of the applications functionality is bundled together in a so called Android activity. In this class the applications view is set and the objects for processing the sensor data are initialized.

```
public class FallActivity extends Activity {

  protected GraphView mGraphView;

  protected FallDetector mFallDetector;

  protected LocationUpdateHandler locationUpdateHandler;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // Be sure to call the super class.
    super.onCreate(savedInstanceState);
    // Create the view
    mGraphView = new GraphView(this);
    setContentView(mGraphView);
    // Create the fall detector
    mFallDetector = new FallDetector(this);
    // Initialize location manager
    locationUpdateHandler = new LocationUpdateHandler(this);
    // Check whether gps is turned on
    locationUpdateHandler.checkGPS();
    // Set app orientation to landscape
    setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
  }
}
```

4 Deployment

If you are to deploy the Android fall detection application on a smartphone or emulator starting from the source code, you need to install an appropriate development environment on your computer. One good option is to go with Eclipse 3.5 or greater[4] including the Java Development Tools plugin. Make sure you have the Java Development Kit 5 or 6 installed. After that you have to download and unpack/install the latest version of the Android SDK Starter Package[5] and install the Android Development Tools plugin[6] for Eclipse.

Now that all necessary software is in place, we will explain how to run the application on the Android emulator. Start by opening the Android SDK manager (from the Eclipse Window menu) and install the Android Platform version 2.1 (the application was not tested on later versions) from the Android Repository under the ‘Available packages’ window. Next create a new virtual device that runs on this platform in the ‘Virtual devices’ window. Now it is time to add the source code to a fresh Android project in Eclipse. Download the code from https://github.com/krikis/FallDet/zipball/master and extract the zipball it in some convenient folder. Next create a new Android project in Eclipse, select the option ‘Create project from existing source’, browse for the previously downloaded source code and click finish. Right click on the project folder and select ‘Run As > Android Application’ to run the fall detection application on the emulator[7]. Yes it runs in landscape mode: press Ctrl-F11 to turn the screen of your emulator.

When you run the application on the Android emulator, a progress dialog will be displayed waiting for the GPS to be activated. Click ‘Just go’ to skip this step, we will explain it later. Now a graph is shown in which nothing much happens, as there is no sensor data coming in on the Emulator. To get some interesting results we need simulation. For this we use the SensorSimulator from OpenIntents[8]. This requires a number of steps: open the ‘platform-tools’ folder of the android SDK in a console and type in the following command to install the SensorSimulatorSettings apk on the emulator: .adb install /full/path/to/project/folder/doc/SensorSimulatorSettings-1.1.0-rc1.apk. When the application was successfully installed, start it up. Next fire up the SensorSimulator jarfile in the project’s doc directory (sensorsimulator-

[4] http://www.eclipse.org/downloads/
[5] http://developer.android.com/sdk/index.html
[6] http://developer.android.com/sdk/eclipse-adt.html
[7] Select ‘Run As > Run Configuration...’ and manually select the target to directly run it on your smartphone.
[8] http://code.google.com/p/openintents/wiki/SensorSimulator

1.1.0-rc1.jar). This will open a simulator interface displaying a set of controls and in a text-area a list of possible IP addresses and a port on which it listens. Enter these details in the corresponding text-fields in the SensorSimulatorSettings app on the emulator, then click the testing tab and click connect. Check all available sensors and see the values change as you fiddle with the controls in the simulator interface.

The connection between the SensorSimulator and the emulator is now successfully set up. This leaves us only to change the project code to make it get its sensor data from the simulator. First make sure the jar file that resides in the project lib directory (sensorsimulator-lib-1.1.0-rc1.jar) is in the class path: right click the jar file and select 'Build Path > Add to Build Path' To do this, open the 'FallDetector' java file in the project 'src' folder and change it according to the instructions below[9].

The import lines for the sensor classes

```
//import org.openintents.sensorsimulator.hardware.Sensor;
//import org.openintents.sensorsimulator.hardware.SensorEvent;
//import org.openintents.sensorsimulator.hardware.SensorEventListener;
//import org.openintents.sensorsimulator.hardware.SensorManagerSimulator;

import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
```

should be changed to this

```
import org.openintents.sensorsimulator.hardware.Sensor;
import org.openintents.sensorsimulator.hardware.SensorEvent;
import org.openintents.sensorsimulator.hardware.SensorEventListener;
import org.openintents.sensorsimulator.hardware.SensorManagerSimulator;

// import android.hardware.Sensor;
// import android.hardware.SensorEvent;
// import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
```

The declaration of the sensor manager

```
private SensorManager mSensorManager;
```

should be this:

```
private SensorManagerSimulator mSensorManager;
```

The initialization of the manager

```
mSensorManager = (SensorManager) activity
                 .getSystemService(activity.SENSOR_SERVICE);
```

has to be changed to this:

```
mSensorManager = SensorManagerSimulator.getSystemService(activity,
    FallActivity.SENSOR_SERVICE);
mSensorManager.connectSimulator();
```

Finally the TWO references to the event sensor type

```
event.sensor.getType() == Sensor.TYPE_ACCELEROMETER
```

should look like this:

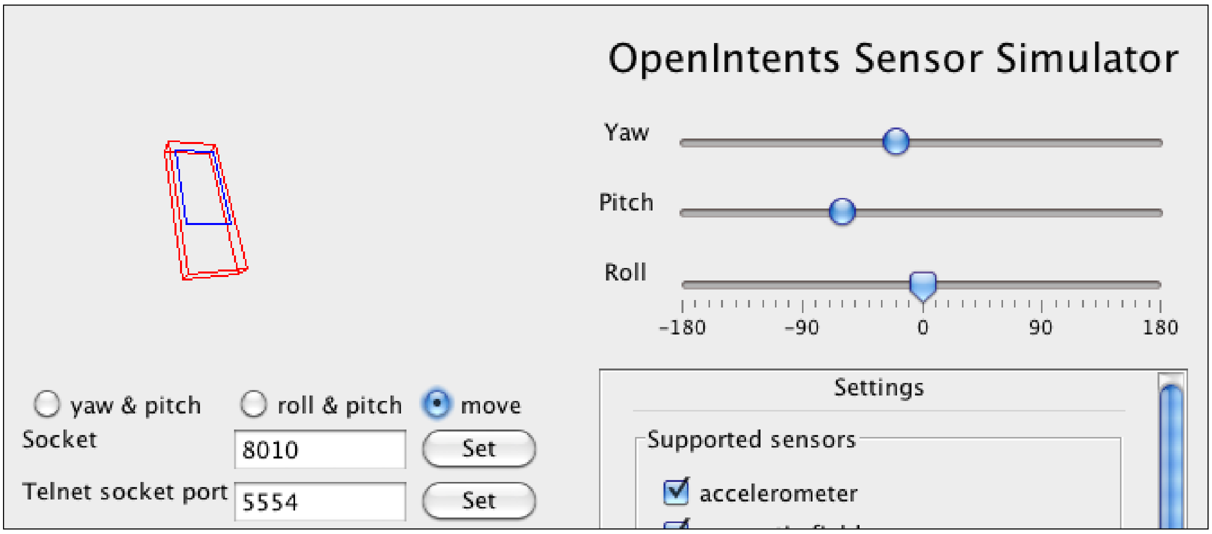
```
event.type == Sensor.TYPE_ACCELEROMETER
```

The application is now ready for interfacing with the SensorSimulator. Fire it up again by right clicking on the project folder and selecting 'Run As > Android Application' (or just clicking the play button in the Eclipse toolbar). The application will once more try to activate the GPS. Open a telnet connection to the emulator and type in a geo fix with a longitude and latitude to provide the emulator with location data.

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
geo fix 6.5365 53.24015
OK
```

[9] If you are a Git man, just run 'git apply enable_simulator.patch' and you are ready to go.

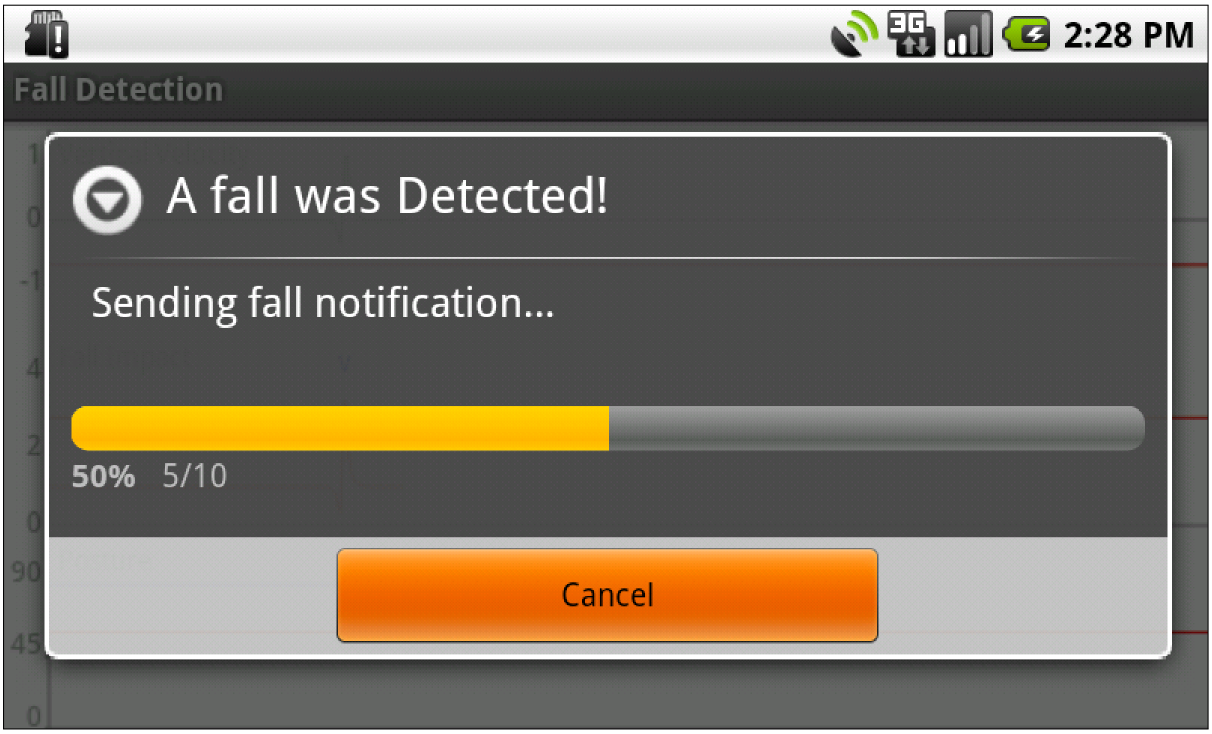
The application on the emulator will now happily proceed and a GPS icon appeared in the top bar of the interface. Now you will see some lines proceed horizontally through the graph. The top graph displays the vertical velocity of the smartphone (VVE). In the middle the fall impact is shown (RSS). In the bottom the posture is shown. Switch back to the simulator interface to generate some sensor data (make sure to maximize the window). Select the 'move' option and drag the wireframe of the phone around to generate some acceleration data.



To simulate the RSS feature ('Fall Impact' in the graph), just shake the phone's wireframe brusquely or smash it against the sides of your screen. The fall impact line in the graph will peak above the red line of the RSS threshold and a blue mark will indicate the feature was detected. Simulating the VVE feature ('Vertical Velocity' in the graph) is less straightforward as it requires smoothly accelerating the wireframe towards gravity for some period of time ($0.6s$). We were only able to hit a threshold of $-0.3g$ having the wireframe 'fall' vertically along two screens. This is naturally because the threshold was selected to detect the fall of a person and not something like a 30cm fall. In order to simulate a complete fall, the posture feature also has to be simulated. This can simply be done by changing the pitch in the simulator interface so that the posture line surfaces the 60° threshold line.

In the image below you see a fall for which all three features are detected. In the uppermost graph the vertical velocity feature is detected where the green line dips below the red threshold line. Shortly after that the fall impact feature is detected in the middle graph when the red line peaks above the threshold line. During the fall the posture in the bottom graph changed from standing to lying. 1 second after the VVE feature was detected the position remains the same for two seconds, so the posture feature is detected.

Now a progress dialog is presented to the user to allow him to cancel the sending of a fall notification.

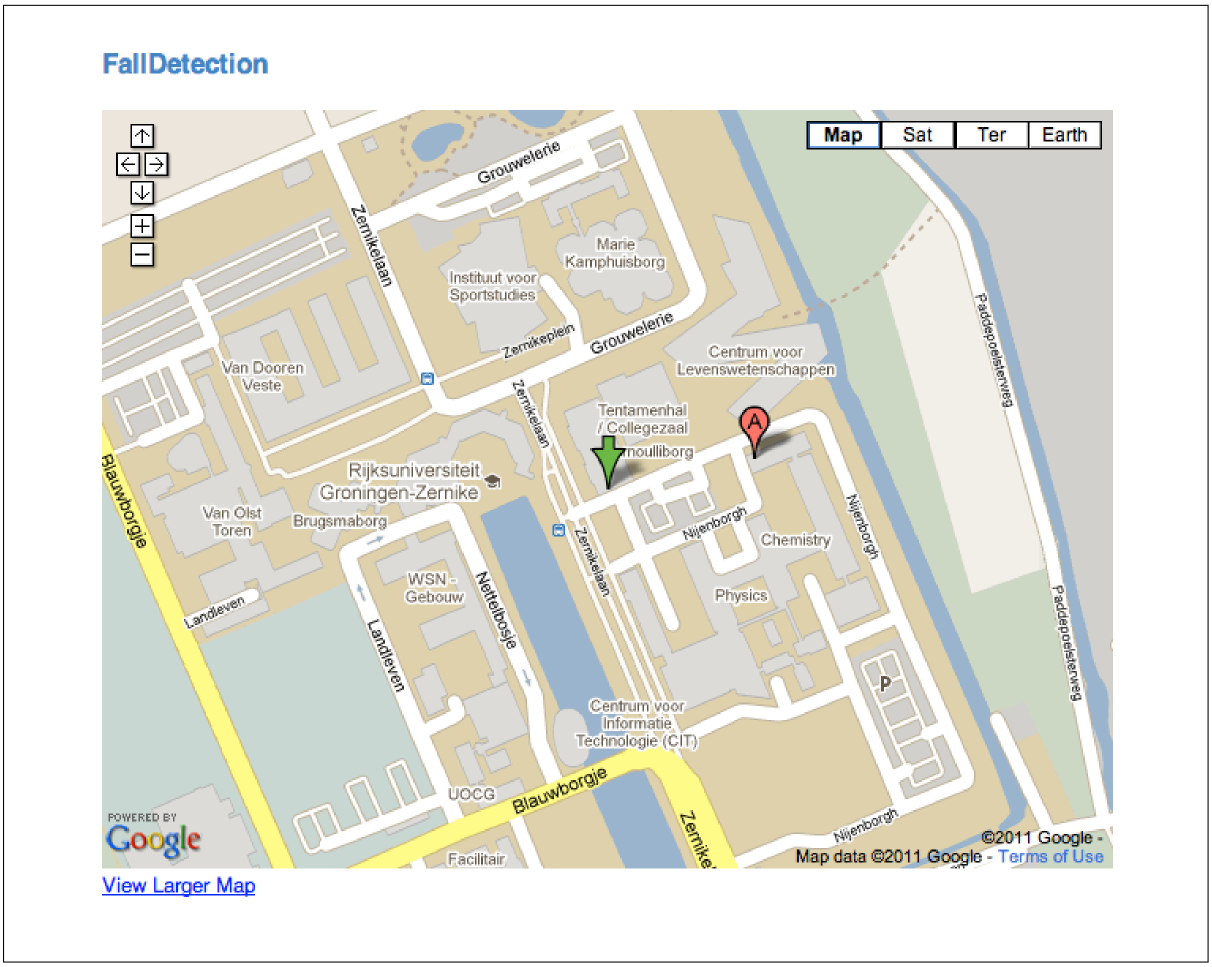


If the user does not cancel the dialog, a notification is sent to a RESTfull webservice. The folder 'server_side' contains a Rails powered web application that receives the notifications and lists them in a web interface for demonstration purposes.

FallDetection

Fall Date	Vertical Velocity	Fall Impact	Latitude	Longitude
less than 5 seconds ago		70.49148	53.240407	6.535999
3 minutes ago		98.972046	0.0	0.0
3 minutes ago		38.792152	0.0	0.0
5 minutes ago		29.501278	0.0	0.0
40 minutes ago		43.18259	0.0	0.0
about 1 hour ago		27.818087	0.0	0.0

By clicking on a row in the table the location of the fall is shown on a map.



5 Evaluation

6 Results