ECE155: Engineering Design with Embedded Systems Spring 2014

Lab 1 (Reading Sensors & The Android API)

Prepared by Kirill Morozov, Han Xu, & Zheng Ma *version 1.4*

Deadline: You must submit the lab to the SVN repository by the submission deadline (see the syllabus) and be prepared to demonstrate Lab 1 to a TA at the start of your assigned Lab 2 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.

1 Objective

The objectives of this week's lab are for you to 1) familiarize yourself with the Android phones that you will be developing for, and 2) learn how to read the sensors on these phones. During this lab, you will write a simple Android application that reads the various sensors available on the phone and graphs them. Your code will initialize an Android Activity; accept the values from the phone's sensors; and display these values on the screen.

The steps in this lab will be:

1. Use the Eclipse IDE and the Android API to create a basic Android application.
2. Add items to the user interface for your Android application.
3. Create event handlers for a variety of sensors and make them modify the user interface.
4. Include a widget to graph a history of sensor values.

Deliverables. For this lab, you must:

- Create a system that will let you display the output from all the sensors that you are monitoring. Display each sensor's current value upon event reception. (See Figure 1, field "Acceleration (linear) (m/s^2)".)
- Since the reading from the sensors will change quickly, you must also display the all-time highest (in absolute value) readings from each sensor so that we can see concrete numbers. (Field "Record Acceleration (linear) (m/s^2)" in Figure 1.)
- Ensure that the data you display is readable, accessible to the user, and that it does not vanish off the edge of the screen. How you do this is up to you. For instance, you can enable scrolling, or simply format the data effectively. Any functional system is acceptable.

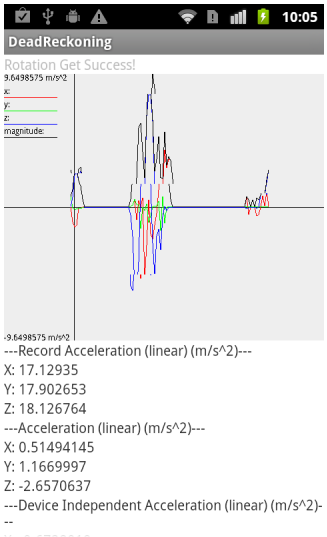

Figure 1: An example of a completed Lab 1.

2 Phases of the lab

This lab contains the following steps:

1. Create infrastructure to display raw data to the user.
2. Create handlers to read and record sensor values.
3. Test.
4. Demonstrate.

3 Create infrastructure to display data to the user

Before we continue, we'll explain the structure of an Android application and about JavaDoc.

3.1 Digression 1: About the Android application structure

Even the empty Android project (e.g. from Lab 0) contains folders and files. Here's what they do.

- The "`src`" folder contains all of your `.java` files for the project. Here you will add the code you'll write. The empty project already contains a "Hello World" activity, `MainActivity`.
- The "`gen`" folder contains automatically generated `.java` files. Of particular interest is the `R.java` file which mirrors the constants and ids you will define in the project's xml. *Do not edit these files directly. Your changes will be lost when you build your code next.*
- The "`bin`" folder is the output directory for the project. Compiled files appear here.
- The "`libs`" folder contains your private libraries. (You can ignore this.)

- The "`res`" folder contains application resources, such as drawable files, layout files, and string values. The "`res/layout`" folder is important. Each xml file in that folder describes the layout of controls in a single activity in your application.

About Activities. The fundamental building block of an Android application is the "Activity." Activities represent screens in your application. When a screen gains focus, Android activates the appropriate activity, which may then interact with the user. When the user switches to a different screen, Android disables the activity, preserving the program state.

The new Android application wizard will create a blank Main activity for you. You can work exclusively with that activity, and you won't need to create any new activities for these labs.

Random testing-related tip. If you did not change anything in your code since the last time you launched your application, Eclipse will not re-upload or restart your application when you click **Run**. Instead, it will just make your application active again. If you are trying to reset your application's state, you will either need to build that option in yourself, or force Eclipse to recompile your application.

3.2 Digression 2: Helping you help yourself: On JavaDoc

For the next step, the rest of the labs, and life in general, you need to learn how to navigate JavaDocs. JavaDoc is documentation automatically extracted from Java programs—while these programs were being developed, developers and technical writers added JavaDoc documentation in the form of specially formatted comments in their source code.

Android's JavaDoc documentation is thorough and easy to read. It is at `http://developer.android.com/reference/packages.html`. If you installed the documentation when you installed the Android SDK, you can also reach the documentation from Eclipse. To do this, hover the mouse over something that you want documentation about. A yellow box will pop up; click on it. Finally, click the right-most button that appears at the bottom of this box. See figure 2 for an example. Keep in mind that you can only hover over something that is defined by the Android SDK. If you try to hover over one of your own classes or variables, Eclipse will look for the JavaDoc attached to your code, and will not find it (unless you add it yourself.)

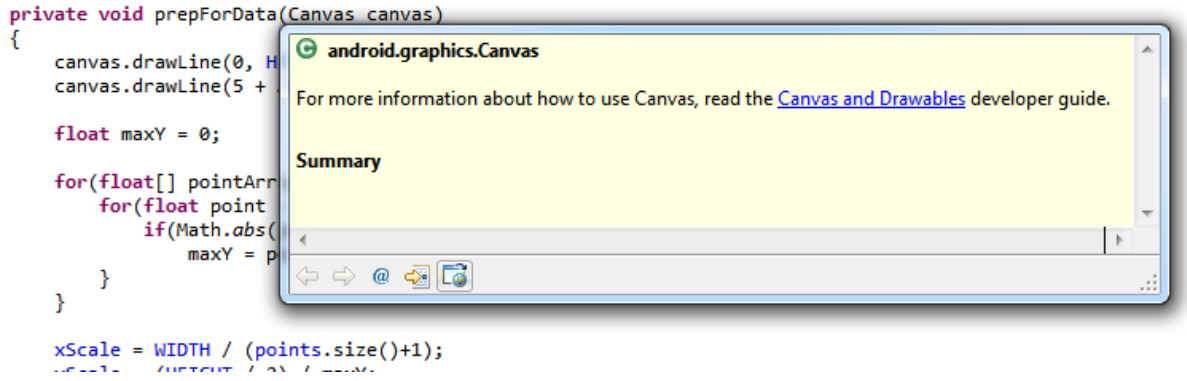

Figure 2: Tool-tip you'll see when mousing over a class name.

3.3 Back to displaying data to the user

The immediate goal of this exercise is to display information to the user. The best way to do that is by creating labels and modifying their contents programmatically. We'll walk you through changing the content of the "Hello World" label from Lab 0.

The Main activity contains the "Hello World" label. To reference this label from your code, you need to give it an ID. Here's how.

1. Open the fragment_main.xml file located in **[your project]** > **res** > **layout**.
2. Switch to the xml tab at the bottom of the editor.
3. Look for the xml tag `<TextView>` and add the attribute `android:id="@+id/label1"`.

In the Android API, all user-visible controls are sub-classes of `android.view.View`. You can get a reference to the "Hello World" label by calling `findViewById()` in your Activity with the label id (which you just added) as a parameter. The Android compilation system will add the ids to the `R` class. For instance, to find the label you just added above, you might call `findViewById(R.id.label1)`.

Labels are `TextView` objects. You can change the text of a label by calling `TextView.setText()`. You can just pass any `String` to `TextView.setText()`, and Android will automatically redraw the label with the new value.

In particular, here's how to modify the "Hello World" label, after you've followed the steps above.

1. Open `MainActivity.java` in **[your project]** > **src** > **ca.uwaterloo.[your project]**.
2. Write the following `PlaceholderFragment()` class:

```
public static class PlaceholderFragment extends Fragment {
 public PlaceholderFragment() {
 }
 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container,
   Bundle savedInstanceState) {
  View rootView = inflater.inflate(R.layout.fragment_main, container,
    false);
       TextView tv = (TextView) rootView.findViewById(R.id.label1);
       tv.setText("I've replaced the label!");
  return rootView;
 }
}
```

Non-deliverable. You should now have an Android application where you can change the label. We are, however, going to take a step back and remove that label, replacing it with automatically-generated labels.

3.4 Adding more labels

You'll need to create additional labels to display all the needed data. Of course, you could do this "by hand": modify the layout xml for your activity directly, or use the graphical layout tool (double-click on the layout xml). This lab requires creating a lot of labels, and doing that by hand is tedious.

Computer scientists ruthlessly automate. Conveniently, the Android API lets you create views programmatically. Here's how to create one label and add it to a view. First, assign an id to the parent:

- Give the top-level layout tag (`<RelativeLayout>` in the auto-generated project) in your xml an ID, like you did for the "Hello World" label. A layout is a special kind of view that can itself contain views (i.e. have child views).

In the `onCreateView()` of the `PlaceHolderFragment`:

1. Create a new `TextView`, e.g. `TextView tv1 = new TextView(rootView.getContext());`
2. Set the text of `tv1`, as above.

Then, in your PlaceholderFragmentÕs `onCreateView()` method:

1. Get a reference to the top-level layout (which you just assigned an id to) by using `findViewByIdwith` the parameter of the ID of the layout (usually starts with `R.id.`. Cast the result to `RelativeLayout`.
2. Create a new `TextView`, e.g. `TextView tv1 = new TextView(rootView.getContext())`
3. Set the text of `tv1`, as above.
4. Add `tv1` to the layout by calling `addView(tv1)` on the top-level layout.

You can do this several times. You will find that you can save some effort if you write a method to create a new label, add it to the layout and return a reference to the new label.

Common Problems At this stage or very soon afterwards, you'll need to create member variables to contain your `TextViews`. You can not call `getApplicationContext()` or `findViewById()` before the `onCreate()` method is executed. For example, the following will cause an null pointer exception:

```
public class MyActivity extends ActionBarActivity {
    TextView globalView = new TextView(getApplicationContext()); // Error
    TextView anotherGlobalView = findViewById(R.id.view1); // Another Error
}
```

Do this instead:

```
public class MyActivity extends ActionBarActivity {

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
             Bundle savedInstanceState) {
        LinearLayout lmain = (LinearLayout) rootView.findViewById(R.id.Label1);
        TextView tv1 = new TextView(rootView.getContext());
        tv1.setText(‘‘Example’’);
        lmain.addView(tv1);
        return rootView;
    }
}
```

RelativeLayout versus LinearLayout. You'll find that after you follow the above directions and add a number of labels, your `TextView`s all run into each other. It is possible to configure the `TextView` objects to not step over each other. But it's easier to change `RelativeLayout` for `LinearLayout`, which automatically puts its contents in non-overlapping positions.

1. In `frament_main.xml`, change `RelativeLayout` to `LinearLayout`. You have to do that in both the `<RelativeLayout>` and `</RelativeLayout>` tags.
2. In `MainActivity.java`, change the line
   ```
   RelativeLayout l = (RelativeLayout)findViewById(R.id.fragment_main);
   ```
   to
   ```
   LinearLayout l = (LinearLayout)findViewById(R.id.fragment_main);
   ```
3. Add the line
   ```
   l.setOrientation(LinearLayout.VERTICAL);
   ```
   which will stack all of the `TextView` objects vertically rather than horizontally.

Deliverable. Add labels for all of the sensors that we'll record in the next stage. Your Android app should now have a number of labels on the screen, which you should have created programmatically. But you might be displaying more labels than will fit on the screen.

3.5 Scrolling

It is easy to implement a scrollbar so that the user can look at all of the labels that you've added.

1. Go back to `fragment_main.xml`, where you should currently have a top-level `LinearLayout`. Before the `LinearLayout`, introduce a `ScrollView`:
   ```
   <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
     android:id="@+id/scroll" android:layout_width="fill_parent"
     android:layout_height="wrap_content">
   ```
   and close the tag after the `LinearLayout`:
   ```
   </ScrollView>
   ```
2. Change the attribute `android:layout_height`'s value from `match_parent` to `wrap_content`.

Deliverable. If you have more content than fits on the screen, make sure that all of the content is somehow viewable.

4 Create handlers to read and record sensor values

Now that you can display data to the user, you need to manufacture data. For this lab, you'll need to get that data from the phone's sensors. A number of Android classes collaborate to allow you to read this data, including:

- The `SensorManager`. Manages the various sensors of the phone.

- `Sensor`s. A software representation of the sensor hardware. You get references to specific instances of `Sensor`s from the `SensorManager`.
- A `SensorEventListener`. One of your classes must implement this interface to receive events from the sensors.
- `SensorEvent`s. Represents a single message from a single sensor. The documentation of the `SensorEvent` has an excellent explanation of the format that each sensor sends its values in.

We'll walk you through how to read the values from the light sensor. You are responsible for reading the other values and for displaying them on the screen.

1. Implement a `SensorEventListener` to receive sensor events[1]. You will want to add some way for the listener to communicate data back to the Activity. In this example, we give the listener a reference to a label, but you have the freedom to do this however you like.

```
class LightSensorEventListener implements SensorEventListener {
 TextView output;

  public LightSensorEventListener(TextView outputView){
   output = outputView;
  }

  public void onAccuracyChanged(Sensor s, int i) {}

  public void onSensorChanged(SensorEvent se) {
    if (se.sensor.getType() == Sensor.TYPE_LIGHT) {
      // TODO put se.values[0] somewhere useful
    }
  }
}
```

2. In your Activity request the sensor manager:
```
SensorManager sensorManager = (SensorManager)
          rootView.getContext().getSystemService(SENSOR_SERVICE);
```

3. In your Activity request the light sensor:
```
Sensor lightSensor =
        sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
```

4. In your Activity create an instance of your new `SensorEventListener` and register it.
```
SensorEventListener l = new LightSensorEventListener();
sensorManager.registerListener(l, lightSensor,
                               SensorManager.SENSOR_DELAY_NORMAL);
```

This should now display the value of the light sensor, if you fill in the TODO line appropriately.

Tip. You saw the C# syntax `String.Format` in ECE150. The corresponding Java syntax is:

```
String s = String.format("(%f, %f, %f)", x, y, z);
```

This will allow you to display (x, y, z) coordinate triples, which will be helpful. You may also want to not display 6 decimal places. I'll leave figuring out how to do that up to you. Use the Internet.

[1] If you know what an inner class is, you can use one. But you might not want to.

Cleaning up after yourself. For many applications, it is appropriate to unregister sensor event listeners in the `onPause()` method of the main activity and reregister them in the `onResume()` method. Unregistering event listeners helps the phone conserve battery power. Unregistering event listeners is perhaps a good idea for this lab, but will cause havoc for Labs 2 and onward unless you stop the phone from going to sleep. We're not requiring that you unregister your event listeners.

Deliverables. Your solution must display current sensor information for these sensors, as well as record values for all sensors but the light sensor.

- (`TYPE_LIGHT`): The readings from the light intensity sensor.
- (`TYPE_ACCELEROMETER`): The three components of linear acceleration from the accelerometer. (You may also display `TYPE_LINEAR_ACCELERATION` data as an alternative.)
- (`TYPE_MAGNETIC_FIELD`): The three components of the magnetic field sensor.
- (`TYPE_ROTATION_VECTOR`): The three components of phone's rotation vector.

The record value is the highest absolute value.

Optional Extra (no marks). Implement a "Clear" button which resets the record values.

4.1 Digression 3: the accelerometer and gravity

You will notice that when the phone is at rest, the accelerometer reports an z-acceleration of approximately 9.81 m/s^2. Surprisingly, this is not the acceleration due to gravity, but rather the acceleration due to the normal force. To understand why this is, we need to look at what exactly an accelerometer measures.

A simple one-axis accelerometer consists of a test mass on a spring. When a force is applied to the case of the accelerometer, the mass moves until the force of the spring matches the force applied to the case. The degree to which the spring is stretched or compressed tells us how much force the case is experiencing.

Now, if we imagine this device in free-fall (don't drop your phones!), we can see that the spring will be at its rest position, while if the device was resting on the ground, the spring would be compressed under the weight of the test mass. Thus, an accelerometer measures acceleration relative to free-fall. In relativity theory, this acceleration is called *proper acceleration*.

In Lab 2, you'll process the data from the accelerometer to measure footsteps. To get the absolute acceleration experienced by your phone, you will need to apply a low-pass filter to the acceleration data. Stay tuned for more information!

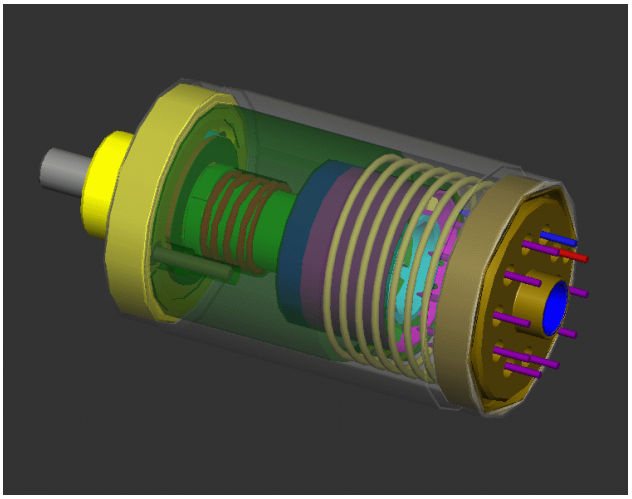

Figure 3: An accelerometer designed by the Archimedes Automated Assembly Planning Project at Sandia National Laboratory.

5 Graphing

For the next labs, it is enormously useful to be able to see how the accelerometer reacts when you manipulate the phone in various ways. We have provided an implementation of a line graph view that will display sets of data points. You will find `LineGraphView.java` in the "materials" directory of the course repository. Figure 1 shows what the graph view looks like. There is a bit of tearing in the screenshot of the graph because the process of taking a screenshot takes longer than one graph update.

Using the LineGraphView The following sample code shows how to set up the `LineGraphView` that we've provided. You are under no obligation to use this code; you can provide your own, if you want, or modify the code as you feel best.

Note that you'll have to include an `import` statement for this class, since it belongs to the `ca.uwaterloo.sensortoy` package. Imports are like `using` statements in C#[2]. Hitting Ctrl-Shift-O in Eclipse often fixes your imports.

```
class MainActivity {
  LineGraphView graph;

  public void onCreate(Bundle savedInstanceState) {
    // ... code was already here ...
    LinearLayout layout = ((LinearLayout)findViewById(R.id.layout));

    graph = new LineGraphView(getApplicationContext(),
                              100,
                              Arrays.asList("x", "y", "z"));
    layout.addView(graph);
    graph.setVisibility(View.VISIBLE);
    // ... other code ...
```

[2] `http://www.harding.edu/fmccown/java_csharp_comparison.html`

```
  }
}
```

Relevant parameters in the constructor call are 100 and `Arrays.asList("a", "b", "c")`. The 100 represents the number of samples to keep on the t-axis. The syntax `Arrays.asList("x", "y", "z")` is Java shorthand for providing a list of labels for the graph.

Once you've created your view, you can start adding data to it. Use methods `addPoint()` and `purge()`. `purge()` will clear the graph. `addPoint()` takes either an array or `List` of data points. The data must be in the same order as the labels that were passed into the constructor. We'll include an example of how to populate the graph below.

Integrating the `LineGraphView`. Now you have code to handle sensor changes. You can hook up the LineGraphView as follows.

```
public void onSensorChanged(SensorEvent event) {
    graph.addPoint(event.values);
}
```

Deliverable. Plot the accelerometer data in graphical format.

6 Test

Your application should now display the current values of the sensors. You must display sensor values showing the three components of each sensor along with the lifetime maximum reading for these components. Also, you must display a graph of the accelerometer sensor readings over time. Your application should not crash or throw exceptions.

7 Demonstration

Once you are satisfied that your design and implementation are working correctly, arrange a demonstration with a teaching assistant. You will need to have the TA complete an assessment form. All of the members of your laboratory group must sign the completed assessment form. Your grade will be based on the results of the project demonstration at the end of the laboratory session and an evaluation of how well your code follows engineering design principles. (We will run plagiarism detection software on the source code, which may subsequently lower your grade and trigger a Policy 71 case. Don't plagiarize!)

Writing Good Code. Your application must follow good engineering design principles. You should therefore avoid: unnecessary code duplication; excessive use of variables the state of which you keep synchronized manually; forgetting to deallocate resources you have requested from the Android OS; and any other such design failures that make your application difficult to maintain or that step on the toes of other applications you share the device with.

8 Submission of project files

Commit the `Lab1_SSS_XX` files to your Subversion repository. The TAs will not mark labs that have not been submitted to SVN. The address is

`https://ecesvn.uwaterloo.ca/courses/ece155/s14/groups/group-NNN-MM`

Email Sanjay Singh if you can't commit to that address.