LE/EECS 4443: Mobile User Interfaces (LAB)

Week 8: Introduction to Sensors & View Flavours

Shogo Toyonaga[1]

[1]Lassonde School of Engineering
York University

February 6, 2025

Table of Contents

1 Introduction

2 Sensors

3 View Flavours

4 Conclusion

Introduction

By the end of this tutorial, you will be able to...

1. Utilize sensor data from your device
2. Manage power consumption from registering / polling sensors
3. Implement & customize views (Widgets)
4. Save & restore states of sophisticated views

Introduction: Sensors

Remark

"Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device..."

https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview

Introduction: Sensors

1. **Motion Sensors**
   - Measure acceleration and rotational forces along $< x, y, z >$ axes.
   - e.g., Accelerometer, Gravity, Gyroscope, Rational Vector(s)
2. **Environmental Sensors**
   - Measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity.
   - e.g., Barometers, photometers, & thermometers
3. **Position Sensors**
   - Measure physical position of a device.
   - e.g., Orientation, Magnometers

Introduction: Sensor Framework I

1. **SensorManager**
   - Creates an instance of the sensor service which provides methods for accessing, listing, registering, and unregistering data through event listener(s).
   - Provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors
2. **Sensor**
   - Creates an instance of a specific sensor.
3. **SensorEvent**
   - Create a sensor event object.
   - A sensor event object retrieves raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

Introduction: Sensor Framework II

4. **SensorEventListener**
   - Creates callback methods that receive sensor events when its values or accuracy change.

Big Picture

"Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: (1) the name of the sensor that triggered the event, (2) the timestamp for the event, (3) the accuracy of the event, and (4) the raw sensor data that triggered the event."

Monitoring Sensor Events

1. Implement the SensorEventListener interface in your Activity class and override two callback methods:
   - onAccuracyChanged()
   - onSensorChanged()
2. Register the Sensor manager listener in onResume() and set the appropriate delay (sampling rate).

   Note: "As a best practice, you should specify the largest delay that you can because the system typically uses a smaller delay than the one you specify. Using a larger delay imposes a lower load on the processor and therefore uses less power."
3. Unregister the Sensor manager listener in onPause(). Failing to unregister the sensor when you don't need it can drain your phones battery. The system will not disable sensors when the screen turns off.

Targetting Sensors

Remark

"By specifying the features that your application requires in the Manifest, you enable Google Play to present your application only to users whose device meets the application's feature requirements, rather than presenting it to all users."

"The sensors you can list include: accelerometer, barometer, compass (geomagnetic field), gyroscope, light, and proximity".

Targetting Sensors: Example

```
1
2  <?xml version="1.0" encoding="utf-8"?>
3  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:tools="http://schemas.android.com/tools">
5
6      <uses-feature
7          android:name="android.hardware.sensor.gyroscope"
8          android:required="true" />
9      <application ...>
10         ...
11     </application>
12 </manifest>
```

See: Demo_Sensors

Sensors: References

1. **Sensors Overview | Sensors & Location**
2. **How to Use Device Sensors the Right Way in Android (Phillip Lackner)**
3. **Professor Mackenzie's Demo Sensors (API)**
4. **Professor Mackenzie's Demo TiltMeter (API)**
5. **Professor Mackenzie's Demo TiltBall (API)**

Introduction

Recall that UI widgets can be implemented in two ways:

1. **Views**, which represent objects (widgets) that are drawn on a screen. Users can interact with them through event listeners.
2. **ViewGroups**, which represent layouts that hold other View or ViewGroup(s).

https://developer.android.com/develop/ui/views/layout/declaring-layout

Introduction: Custom Views

- If none of the prebuilt widgets or layouts meet your needs, you can create your own View subclass.
- If you only need to make small adjustments to an existing widget or layout, you can subclass the widget or layout and override its methods
- When you implement custom Views, it is critical that you consider:
  1. Form Factor
  2. Device Capabilities (Processing)
  3. User Experience

Introduction: Custom Views

Step-by-Step Instructions

1. Extend an existing **View** class or subclass with your own class.
2. Override some of the methods from the superclass such as: **onDraw(), onMeasure(), onKeyDown(), etc.,**
3. Use your new extension class. Once completed, you can use your new extension class in place of the view it was based on.

https://developer.android.com/develop/ui/views/layout/custom-views/custom-components

Introduction: Custom Views (Custom Class) I

```
1  public class CustomButton extends View {
2
3      // Custom Attributes
4      final private Paint panel = new Paint();
5      private int BACKGROUND_COLOUR = Color.GREEN;
6      final int DEFAULT_WIDTH = 250;
7      final int DEFAULT_HEIGHT = 100;
8      final int TEXT_SIZE = 50;
9      ...
10     // We use this method in each of the different constructors
11    private void initialize() {
12         setClickable(true);
13         setFocusableInTouchMode(true);
14         setFocusable(true);
15         // Background Colour
16         panel.setColor(BACKGROUND_COLOUR);
17         // Text Properties
18         panel.setTextSize(TEXT_SIZE);
19         panel.setTextAlign(Paint.Align.CENTER);
20         panel.setColor(Color.DKGRAY);
21     }
22
23
24
```

Introduction: Custom Views (Custom Class) II

```
25      // you can be a bit more creative with how you draw your custom view....
            haha :P
26      @Override
27      protected void onDraw(@NonNull Canvas canvas) {
28          super.onDraw(canvas);
29          setBackgroundColor(BACKGROUND_COLOUR);
30          // Get center of view coordinates
31          float x = getWidth() / 2f;
32          // Vertical centering
33          float y = getHeight() / 2f - ((panel.descent() + panel.ascent()) / 2);
34          // Draw the text at the center
35          canvas.drawText("Button", x, y, panel);
36      }
37
38      @Override
39      protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
40          super.onMeasure(widthMeasureSpec, heightMeasureSpec);
41          setMeasuredDimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
42      }
```

Introduction: Custom Views (Controller) I

```
1  public class MainActivity extends AppCompatActivity implements View.
       OnClickListener {
2      // Declare our Custom View Class here
3      CustomButton custom_button;
4      int count;
5      ToneGenerator tone;
6
7      private void initialize() {
8          custom_button = findViewById(R.id.custom_button);
9          count = 0;
10         tone = new ToneGenerator(AudioManager.STREAM_NOTIFICATION, ToneGenerator
               .MAX_VOLUME);
11     }
12
13
14
15
16
17
18
19
20
21
22
```

Introduction: Custom Views (Controller) II

```
23
24
25      @Override
26      protected void onCreate(Bundle savedInstanceState) {
27          super.onCreate(savedInstanceState);
28          EdgeToEdge.enable(this);
29          setContentView(R.layout.activity_main);
30          initialize();
31          custom_button.setOnClickListener(this);
32          ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v,
                insets) -> {
33              Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.
                    systemBars());
34              v.setPadding(systemBars.left, systemBars.top, systemBars.right,
                    systemBars.bottom);
35              return insets;
36          });
37      }
38  ...
```

Introduction: Custom Views (Layout) I

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:id="@+id/main"
6      android:layout_width="match_parent"
7      android:layout_height="match_parent"
8      android:gravity="center"
9      android:orientation="vertical"
10     tools:context=".MainActivity">
11
12     <!-- This is how we load the view in our XML -->
13
14     <view
15         android:id="@+id/custom_button"
16         class="com.example.democustomviews.CustomButton"
17         android:layout_width="wrap_content"
18         android:layout_height="wrap_content" />
19 </LinearLayout>
```

Demo_CustomViews

Introduction: Display Considerations

- When you are selecting or designing custom widgets to design your application, it is critical that you consider the following **display properties:**
  1. Physical Size
  2. Resolution
  3. Orientation (Natural, Current, Rotation(s))
- To some extent, you can see how your application appears in your layouts "Design" tab with respect to a reference device.
- You can obtain information about the display programmatically. See: https://developer.android.com/reference/android/view/Display

Interesting Views I

1. **ImageView**: Displays image resources (BitMap, Drawable) and supports native tinting and scaling.
2. **VideoView**: Displays a video file. It can also load images from content providers. Do note that VideoView does not retain its full state when an activity is paused. You should save and restore the play state, play position, selected tracks, or subtitle tracks..
3. **WebView**: Displays a web page as part of the application. In particular, it is useful when your app needs to provide data to the user which requires an internet connection.

Interesting Views II

4. **MapView**: Displays a map obtained from Google Maps Services. You must have an API key associated with your project.
5. **SurfaceView**: Displays and provides support for a dedicated drawing surface.

   Note: "One of the purposes of this class is to provide a surface in which a secondary thread can render into the screen. If you are going to use it this way, you need to be aware of some threading semantics [...]"

Conclusion

Conclusion

Thank you for your attention!
Questions?