CSCI3310 Mobile Computing & Application Development

Lab 8 – Sensors: Bubble Level

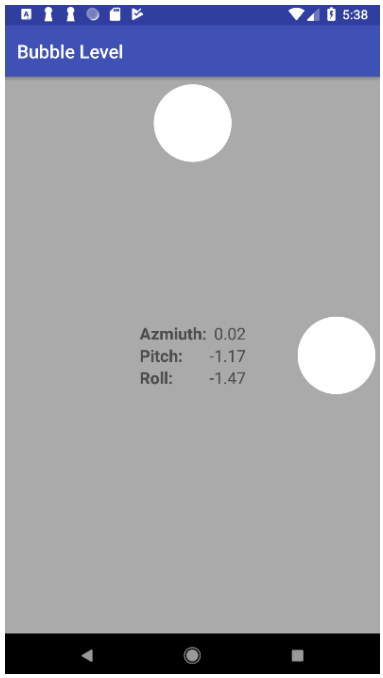
Objective

A common use of motion and position sensors, especially for games, is to determine the orientation of the device, i.e. the device's tilt. In this lab, you create a sensor-based app called `Bubble Level` for determining the device's orientation.
To determine device orientation, it involves using both the accelerometer and geomagnetic field sensor, which are common even on older devices. What you learn in this lab:

1) What the accelerometer and magnetometer sensors do.
2) The differences between the device-coordinate system and the Earth coordinate system, and which sensors use which systems.
3) Orientation angles (azimuth, pitch, roll), and how they relate to other coordinate systems.
4) How to use methods from the sensor manager to get the device orientation angles. How activity rotation (portrait or landscape) affects sensor input.

Todo

1) Download and explore a starter app.
2) Get the orientation angles from the accelerometer and magnetometer, and update text views in the activity to display those values.
3) Change the color of a shape drawable to indicate which edge of the device is tilted up.
4) Handle changes to the sensor data when the device is rotated from portrait to landscape.



The BubbleLevel app displays the device orientation angles as numbers and as colored bubbles along the four edges of the device screen. There are three components to device orientation:

- *Azimuth*: The direction (north/south/east/west) the device is pointing. 0 is magnetic north.
- *Pitch*: The top-to-bottom tilt of the device. 0 is flat.
- *Roll*: The left-to-right tilt of the device. 0 is flat.

When you tilt the device, the bubbles along the edges that are tilted up become lighter in color.

1. Build the BubbleLevel app

In this task, you download and open the starter app for the project and explore the layout and activity code. Then you implement the `onSensorChanged()` method to get data from the sensors, convert that data into orientation angles, and report updates to sensor data in several text views.

1.1 Download and explore the starter app

1) Download the `BubbleLevel start` app and open it in Android Studio.
2) Open `res/layout/activity_main.xml`.

The initial layout for the BubbleLevel app includes several text views to display the device orientation angles (azimuth, pitch, and roll)—you learn more about how these angles work later in the lab. All those textviews are nested inside their own constraint layout to center them both horizontally and vertically within the activity. You need the nested constraint layout because later in the lab you add the bubbles around the edges of the screen and this inner text view.

3) Open `MainActivity`.

MainActivity in this starter app contains much of the skeleton code for managing sensors and sensor data.

4) Examine the `onCreate()` method.

This method gets an instance of the `SensorManager` service, and then uses the `getDefaultSensor()` method to retrieve specific sensors. In this app, those sensors are the accelerometer (`Sensor.TYPE_ACCELEROMETER`) and the magnetometer (`Sensor.TYPE_MAGNETIC_FIELD`).

The Android sensor framework can combine data from both accelerometer and magnetometer to get a fairly accurate (for this app at least) device orientation

5) At the top of `onCreate()`, note this line:

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT)
```

This line locks the activity in portrait mode, to prevent the app from automatically rotating the activity as you tilt the device. Activity rotation and sensor data can interact in unexpected ways. Later in the lab, you explicitly handle sensor data changes in your app in response to activity rotation and remove this line.

6) Examine the `onStart()` and `onStop()` methods. The `onStart()` method registers the listeners for the accelerometer and magnetometer, and the onStop() method unregisters them.

7) Examine onSensorChanged() and onAccuracyChanged(). These are the methods from the SensorEventListener interface that you have to implement. The onAccuracyChanged() method is empty because it is unused in this class. You implement onSensorChanged() in the next task.

1.2 Get sensor data and calculate orientation angles

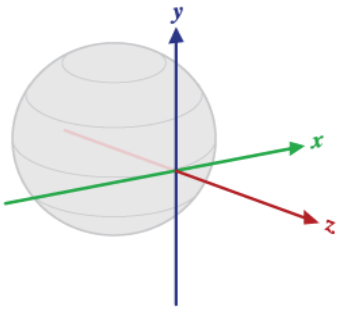
In this task you implement the onSensorChanged() method to get raw sensor data, use methods from the SensorManager to convert that data to device orientation angles, and update the text views with those values.

1) Open MainActivity.

2) Add member variables to hold copies of the accelerometer and magnetometer data.

```
private var mAccelerometerData = FloatArray(3)
private var mMagnetometerData = FloatArray(3)
```

When a sensor event occurs, both the accelerometer and the magnetometer produce arrays of floating-point values representing points on the *x*-axis, *y*-axis, and *z*-axis of the device's coordinate system. You will combine the data from both these sensors, and over several calls to onSensorChanged(), so you need to retain a copy of this data each time it changes.



1. Scroll down to the onSensorChanged() method. Add a line to get the sensor type from the sensor event object:

```
val sensorType = sensorEvent.sensor.type
```

2. Add tests for the accelerometer and magnetometer sensor types, and clone the event data into the appropriate member variables:

```
when (sensorType) {
    Sensor.TYPE_ACCELEROMETER -> mAccelerometerData = sensorEvent.values.clone()
    Sensor.TYPE_MAGNETIC_FIELD -> mMagnetometerData = sensorEvent.values.clone()
    else -> return
}
```

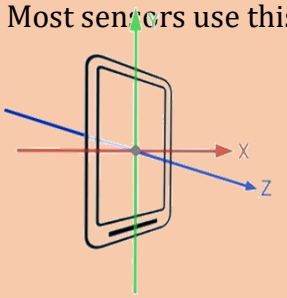
You use the clone() method to explicitly make a copy of the data in the values array. The SensorEvent object (and the array of values it contains) is reused across calls to onSensorChanged(). Cloning those values prevents the data you're currently interested in from being changed by more recent data before you're done with it.

3) After the switch statement, use the SensorManager.getRotationMatrix() method to generate a rotation matrix (explained below) from the raw accelerometer and magnetometer data. The matrix is used in the next step to get the device orientation, which is what you're really interested in.

```
val rotationMatrix = FloatArray(9)
val rotationOK = SensorManager.getRotationMatrix(rotationMatrix, null,
    mAccelerometerData, mMagnetometerData)
```

A rotation matrix is a linear algebra term that translates the sensor data from one coordinate system to another—in this case, from the device's coordinate system to the Earth's coordinate system. That matrix is an array of nine float values because each point (on all three axes) is expressed as a 3D vector.

The device-coordinate system is a standard 3-axis (*x*, *y*, *z*) coordinate system relative to the device's screen when it is held in the default or natural orientation. Most sensors use this coordinate system. In this orientation:



The *x*-axis is horizontal and points to the right edge of the device.

The *y*-axis is vertical and points to the top edge of the device.

The *z*-axis extends up from the surface of the screen. Negative *z* values are behind the screen.

The Earth's coordinate system is also a 3-axis system, but relative to the surface of the Earth itself. In the Earth's coordinate system:

The *y*-axis points to the magnetic north along the surface of the Earth.

The *x*-axis is 90 degrees from *y*, pointing approximately east.

The *z*-axis extends up into space. Negative *z* extends down into the ground.

A reference to the array for the `rotation matrix` is passed into the getRotationMatrix() method and modified in place. The second argument to `getRotationMatrix()` is an inclination matrix, which you don't need for this app. You can use `null` for this argument.

The `getRotationMatrix()` method returns a boolean (the `rotationOK` variable), which is true if the rotation was successful. The boolean might be false if the device is free-falling (meaning that the force of gravity is close to 0), or if the device is pointed very close to the magnetic north. The incoming sensor data is unreliable in these cases, and the matrix can't be calculated. Although the boolean value is almost always true, it's good practice to check that value anyhow.

4) Call the `SensorManager.getOrientation()` method to get the orientation angles from the rotation matrix. As with `getRotationMatrix()`, the array of float values containing those angles is supplied to the `getOrientation()` method and modified in place.

```
val orientationValues = FloatArray(3)
if (rotationOK) {
    SensorManager.getOrientation(rotationMatrix, orientationValues)
}
```

The angles returned by the `getOrientation()` method describe how far the device is oriented or tilted with respect to the Earth's coordinate system. There are three components to orientation:

- *Azimuth*: The direction (north/south/east/west) the device is pointing. 0 is magnetic north.
- *Pitch*: The top-to-bottom tilt of the device. 0 is flat.
- *Roll*: The left-to-right tilt of the device. 0 is flat.

All three angles are measured in radians, and range from -π (-3.141) to π.

5) Create variables for azimuth, pitch, and roll, to contain each component of the orientationValues array. You adjust this data later in the lab, which is why it is helpful to have these separate variables.

```
val azimuth = orientationValues[0]
val pitch = orientationValues[1]
val roll = orientationValues[2
```

6. Get the placeholder strings, from the resources, fill the placeholder strings with the orientation angles and update all the text views.

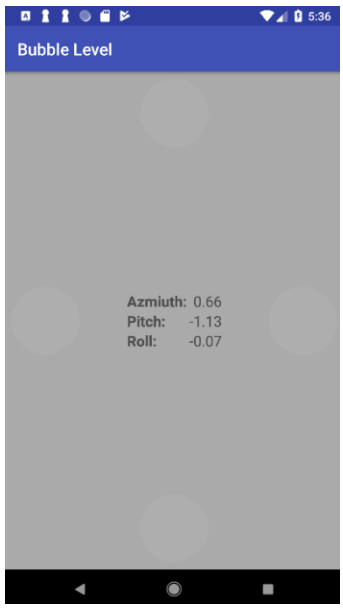
```
mTextSensorAzimuth?.text = resources.getString(
    R.string.value_format, azimuth))
mTextSensorPitch?.text = resources.getString(
    R.string.value_format, pitch))
mTextSensorRoll?.text = resources.getString(
    R.string.value_format, roll))
```

The string (value_format in `strings.xml`) contains placeholder code (`"%1$.2f"`) that formats the incoming floating-point value to two decimal places.

```
<string name="value_format">%1$.2f</string>
```

1.3 Build and run the app

1. Run the app. Place your device flat on the table (Or if you use the emulator, use the virtual sensor to adjust X-Rot to -90 in order to give a horizontal virtual device). The output of the app looks something like this:



the

Even a motionless device shows fluctuating values for the azimuth, pitch, and roll. Note also that even though the device is flat, the values for pitch and roll may not be 0. This is because the device sensors are extremely sensitive and pick up even tiny changes to the environment, both changes in motion and changes in ambient magnetic fields.

2. Turn the device on the table from left to right, leaving it flat on the table.

Note how the value of the azimuth changes. An azimuth value of 0 indicates that the device is pointing (roughly) north.

Note that even if the value of the azimuth is 0, the device may not be pointing exactly north. The device magnetometer measures the strength of any magnetic fields, not just that of the Earth. If you are in the presence of other magnetic fields (most electronics emit magnetic fields, including the device itself), the accuracy of the magnetometer may not be exact.

Note: If the azimuth on your device seems very far off from actual north, you can calibrate the magnetometer by waving the device a few times in the air [in a figure-eight motion](#).

3. Lift the bottom edge of the device so the screen is tilted away from you. Note the change to the pitch value. Pitch indicates the top-to-bottom angle of tilt around the device's horizontal axis.

4. Lift the left side of the device so that it is tilted to the right. Note the change to the roll value. Roll indicates the left-to-right tilt along the device's vertical axis.

5. Pick up the device and tilt it in various directions. Note the changes to the pitch and roll values as the device's tilt changes. What is the maximum value you can find for any tilt direction, and in what device position does that maximum occur?

2. Add the bubbles

In this task, you update the layout to include bubbles along each edge of the screen and change the opaqueness of the bubbles so that they become lighter in color when a given edge of the screen is tilted up.

The color changes in the bubbles rely on dynamically changing the alpha value of a shape drawable in response to new sensor data. The alpha determines the opacity of that drawable so that smaller alpha values produce a lighter shape, and larger values produce a lighter in color shape.

2.1 Add the bubbles and modify the layout

1) Add a new file called `bubble.xml` to the project, in the `res/drawable` directory. (Create the directory if needed.)

2) Replace the selector tag in `bubble.xml` with an oval shape drawable whose color is solid white (`"@android:color/white"`):

```
<shape
   xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
   <solid android:color="@android:color/white"/>
</shape>
```

3) Open `res/values/dimens.xml`. Add a dimension for the bubble size:

```
<dimen name="bubble_size">84dp</dimen>
```

4) In `activity_layout.xml`, add an `ImageView` after the inner `ConstraintLayout`, and before the outer one. Use these attributes:

Attribute field	Enter the following:
android:id	"@+id/bubble_top"
android:layout_width	"@dimen/bubble_size"
android:layout_height	"@dimen/bubble_size"
android:layout_margin	"@dimen/base_margin"
app:layout_constraintLeft_toLeftOf	"parent"
app:layout_constraintRight_toRightOf	"parent"
app:layout_constraintTop_toTopOf	"parent"
app:srcCompat	"@drawable/bubble"
tools:ignore	"ContentDescription"

This view places a bubble drawable the size of the bubble_size dimension at the top edge of the screen. Use the `app:srcCompat` attribute for a vector drawable in an `ImageView` (versus `android:src` for an actual image.) The `app:srcCompat` attribute is available in the `Android Support Library` and provides the greatest compatibility for vector drawables.

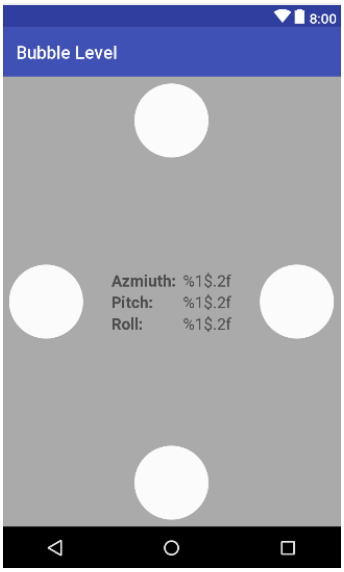
The `tools:ignore` attribute is used to suppress warnings in Android Studio about a missing content description. Generally, `ImageView` views need alternate text for sight-impaired users, but this app does not use or require them, so you can suppress the warning here.

5) Add the following code below that first ImageView. This code adds the other three bubbles along the remaining edges of the screen.

```
<ImageView    android:id="@+id/bubble_bottom"
    android:layout_width="@dimen/bubble_size"
    android:layout_height="@dimen/bubble_size"
    android:layout_marginBottom="@dimen/base_margin"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:srcCompat="@drawable/bubble"
    tools:ignore="ContentDescription" />
<ImageView
    android:id="@+id/bubble_right"
    android:layout_width="@dimen/bubble_size"
    android:layout_height="@dimen/bubble_size"
    android:layout_marginEnd="@dimen/base_margin"
```

```
    android:layout_marginRight="@dimen/base_margin"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/bubble"
    tools:ignore="ContentDescription"/>
<ImageView
   android:id="@+id/bubble_left"
    android:layout_width="@dimen/bubble_size"
    android:layout_height="@dimen/bubble_size"
    android:layout_marginLeft="@dimen/base_margin"
    android:layout_marginStart="@dimen/base_margin"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/bubble"
    tools:ignore="ContentDescription" />
```

The layout preview should now look like this:



6) 6. Add the `android:alpha` attribute to all four `ImageView` elements, and set the value to `"0.05"`. The alpha is the opacity of the shape. Smaller values are less opaque (less visible). Setting the value to 0.05 makes the shape very nearly invisible, but you can still see them in the layout view.

2.2 Update the bubble color with new sensor data

Next you modify the `onSensorChanged()` method to set the alpha value of the bubbles in response to the pitch and roll values from the sensor data. A higher sensor value indicates a larger degree of tilt. The higher the sensor value, the more opaque (the ligher the color) you make the bubble.

1) In `MainActivity`, add member variables at the top of the class for each of the bubble `ImageView` objects:

```
private var mBubbleTop : ImageView? = null
```

```
private var mBubbleBottom : ImageView? = null
private var mBubbleLeft : ImageView? = null
private var mBubbleRight: ImageView? = null
```

2) In `onCreate()`, just after initializing the text views for the sensor data, initialize the bubble views:

```
mBubbleTop = findViewById<View>(R.id.bubble_top) as ImageView
mBubbleBottom = findViewById<View>(R.id.bubble_bottom) as ImageView
mBubbleLeft = findViewById<View>(R.id.bubble_left) as ImageView
mBubbleRight = findViewById<View> (R.id.bubble_right) as ImageView
```

3) In `onSensorChanged()`, right after the lines that initialize the azimuth, pitch, and roll variables, reset the pitch or roll values that are close to 0 (less than the value of the `VALUE_DRIFT` constant) to be 0:

```
if (Math.abs(pitch) < VALUE_DRIFT) {
    pitch = 0f
}
if (Math.abs(roll) < VALUE_DRIFT) {
    roll = 0f
}
```

When you initially ran the `BubbleLevel` app, the sensors reported very small non-zero values for the pitch and roll even when the device was flat and stationary. Those small values can cause the app to flash very light-colored bubbles on all the edges of the screen. In this code, if the values are close to 0 (in either the positive or negative direction), you reset them to 0.

4) Reset all the bubbles to be invisible (sets the alpha to 0) each time `onSensorChanged()` is called.

```
mBubbleTop?.alpha = 0f
mBubbleBottom?.alpha = 0f
mBubbleLeft?.alpha = 0f
mBubbleRight?.alpha = 0f
```

This is necessary because sometimes if you tilt the device too quickly, the old values for the bubbles stick around and retain their lighter in color. Resetting them each time prevents these artifacts.

5) Update the alpha value for the appropriate bubble with the values for pitch and roll.

```
if (pitch > 0) {
    mBubbleBottom?.alpha = pitch
} else {
    mBubbleTop?.alpha = Math.abs(pitch)
} if (roll > 0) {
    mBubbleLeft?.alpha = roll
} else {
    mBubbleRight?.alpha = Math.abs(roll)
}
```

Note that the pitch and roll values you calculated in the previous task are in radians, and their values range from -π to +π. Alpha values, on the other hand, range only from 0.0 to 1.0. You could do the math to convert radian units to alpha values, but you may have noted earlier that the higher pitch and roll values only occur when the device is tilted vertically or even upside down. For the BubbleLevel app, you're only interested in displaying dots in

response to *some* device tilt, not the full range. This means that you can conveniently use the radian units directly as input to the alpha.

6) Build and run the app.

You should now be able to tilt the device and have the edge facing "up" display a dot which becomes lighter in color the further up you tilt the device.

3. Handle activity rotation

The Android system itself uses the accelerometer to determine when the user has turned the device sideways (from portrait to landscape mode, for a phone). The system responds to this rotation by ending the current activity and recreating it in the new orientation, redrawing your activity layout with the "top," "bottom," "left," and "right" edges of the screen now reflecting the new device position.

You may assume that with BubbleLevel if you rotate the device from landscape to portrait, the sensors will report the correct data for the new device orientation, and the bubbles will continue to appear on the correct edges. That's not the case. When the activity rotates, the activity drawing coordinate system rotates with it, but the sensor coordinate system remains the same. The sensor coordinate system *never* changes position, regardless of the orientation of the device.

The second tricky point for handling activity rotation is that the default or natural orientation for your device may not be portrait. The default orientation for many tablet devices is landscape. The sensor's coordinate system is always based on the natural orientation of a device.

The BubbleLevel starter app included a line in onCreate() to lock the orientation to portrait mode:

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT)
```

Locking the screen to portrait mode in this way solves one problem—it prevents the coordinate systems from getting out of sync on portrait-default devices. But on landscape-default devices, the technique forces an activity rotation, which causes the device and sensor coordinate systems to get out of sync.

Here's the right way to handle device and activity rotation in a sensor-based drawing: First, use the `Display.getRotation()` method to query the current device orientation. Then use the `SensorManager.remapCoordinateSystem()` method to remap the rotation matrix from the sensor data onto the correct axes. This is the technique you use in the BubbleLevel app in this task.

The getRotation() method returns one of four integer constants, defined by the Surface class:

- `ROTATION_0`: The default orientation of the device (portrait for phones).

- ROTATION_90: The "sideways" orientation of the device (landscape for phones). Different devices may report 90 degrees either clockwise or counterclockwise from 0.
- ROTATION_180: Upside-down orientation, if the device allows it.
- ROTATION_270: Sideways orientation, in the opposite direction from ROTATION_90.

Note that many devices do not have ROTATION_180 at all or return ROTATION_90 or ROTATION_270 regardless of which direction the device was rotated (clockwise or counterclockwise). It is best to handle all possible rotations rather than to make assumptions for any particular device.

3.1 Get the device rotation and remap the coordinate system

1) In MainActivity, edit onCreate() to remove or comment out the call to setRequestedOrientation().

```
//setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT)
```

2) In MainActivity, add a member variable for the Display object.

```
private var mDisplay: Display? = null
```

3) At the end of onCreate(), get a reference to the window manager, and then get the default display. You use the display to get the rotation in onSensorChanged().

```
val wm = getSystemService(WINDOW_SERVICE) as WindowManager
mDisplay = wm.defaultDisplay
```

4) In onSensorChanged(), just after the call to getRotationMatrix(), add a new array of float values to hold the new adjusted rotation matrix.

```
val rotationMatrixAdjusted = FloatArray(9)
```

5) Get the current device rotation from the display and add a switch statement for that value. Use the rotation constants from the Surface class for each case in the switch. For ROTATION_0, the default orientation, you don't need to remap the coordinates. You can just clone the data in the existing rotation matrix:

```
when (mDisplay?.rotation) {
    Surface.ROTATION_0:= -> rotationMatrixAdjusted = rotationMatrix.clone()
}
```

6) Add additional cases for the other rotations, and call the SensorManager.remapCoordinateSystem() method for each of these cases.

This method takes as arguments the original rotation matrix, the two new axes on which you want to remap the existing *x*-axis and *y*-axis, and an array to populate with the new data. Use the axis constants from the SensorManager class to represent the coordinate system axes.

```
Surface.ROTATION_90 -> SensorManager.remapCoordinateSystem(rotationMatrix,
       SensorManager.AXIS_Y, SensorManager.AXIS_MINUS_X,
      rotationMatrixAdjusted)

Surface.ROTATION_180 -> SensorManager.remapCoordinateSystem(rotationMatrix,
      SensorManager.AXIS_MINUS_X, SensorManager.AXIS_MINUS_Y,
      rotationMatrixAdjusted)
```

```
Surface.ROTATION_270 -> SensorManager.remapCoordinateSystem(rotationMatrix,
        SensorManager.AXIS_MINUS_Y, SensorManager.AXIS_X,
       rotationMatrixAdjusted)
```

7) Modify the call to `getOrientation()` to use the new adjusted rotation matrix instead of the original matrix.

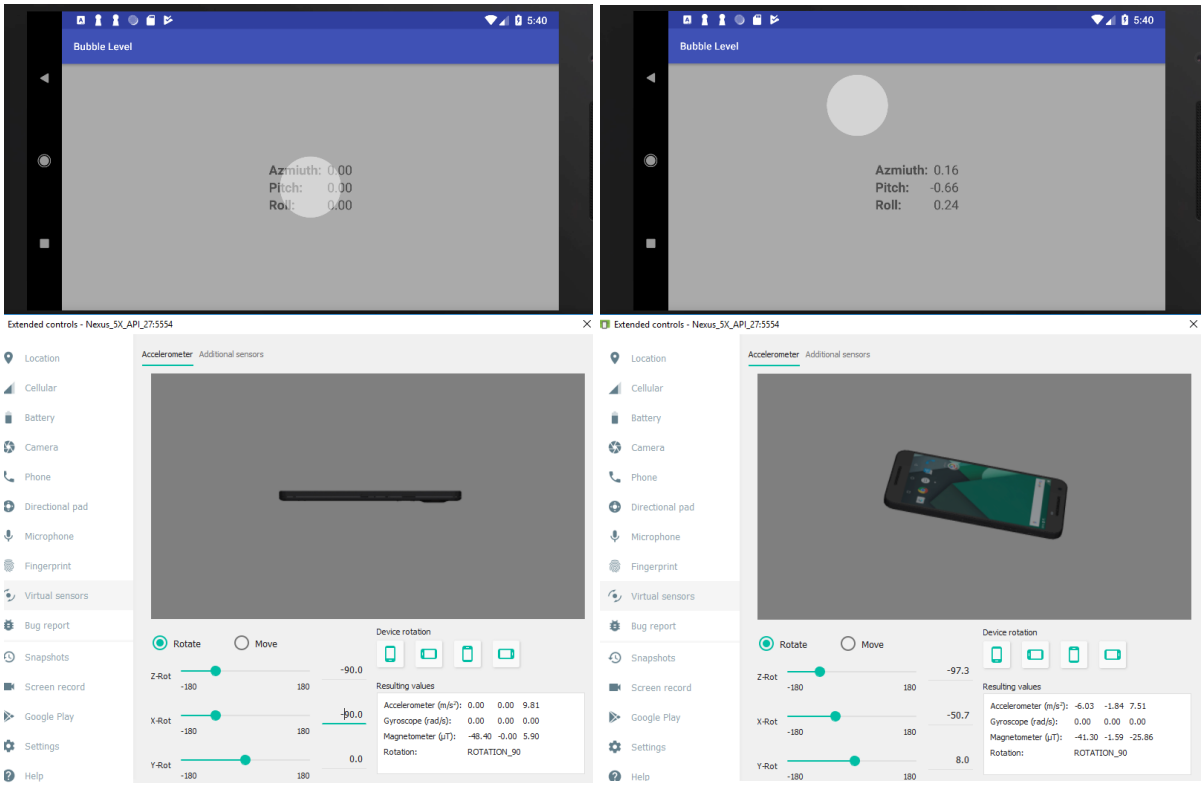
```
SensorManager.getOrientation(rotationMatrixAdjusted, orientationValues)
```

8) Build and run the app again. The colors of the bubbles should now change on the correct edges of the device, regardless of how the device is rotated.

4. (Optional) Floating bubble

Extend the app to simulate a real bubble level along the long or short edges of the device. (A *bubble or spirit level* is a tool that uses colored liquid and an air bubble to show whether a surface is a level.) Use a single bubble as the "bubble." When the device is level (flat on the table), the bubble should appear in the center of the screen. Move the bubble left or right on the screen when one long edge of the device is tilted.

Hint: you can exploit several functions provided by the `View` class to achieve this task, including: `getWidth()`, `getHeight()`, `setX()`, `setY()`. In drawing 2D graphics, the coordinate system is slightly different where the (0, 0) is at the top-left corner while Y-axis is pointing downward. Also, be reminded to remove all constraints of the single bubble in order to let it “float” around.



References

Android developer documentation:

- Sensors Overview
- Motion Sensors
- Position Sensors

Android API reference documentation:

- Sensor
- SensorEvent
- SensorManager
- SensorEventListener
- Accelerometer Basics