InvenSense Inc.
1197 Borregas Ave., Sunnyvale, CA 94089 U.S.A.
Tel: +1 (408) 988-7339 Fax: +1 (408) 988-8104
Website: www.invensense.com

Document Number:
Revision:
Release Date:

Improving the Performance of Accelerometer Apps Using Android Gravity API

TABLE OF CONTENTS

1. Revision History

Revision Date	Revision	Description
06/26/2012	1.0	Document created

2. Purpose

This tutorial teaches the app developers how to improve the performance of their accelerometer-based applications using the “gravity sensor” on Android-powered devices. A simple “steering wheel” example code is used to demonstrate how to use the gravity sensor to detect the tilt angle of the device, and how to control the graphics content with the detected tilt angle for racing games or flight simulator type applications. The tutorial also explores the advantages of using the gravity output over a raw accelerometer output.

3. Requirements

- Eclipse
- Android SDK
- Java Programming Knowledge
- Knowledge of Accelerometers
- Knowledge of quaternion and rotations
- Completion of Android’s OpenGL ES tutorial

4. Making Your App Respond to Device Motion

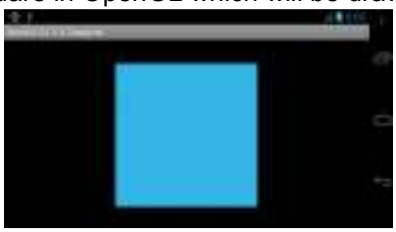
Starting with Android 2.3 (Gingerbread), the Android platform provides several sensors that let you monitor the motion of a device. Two of these sensors are always hardware-based (the accelerometer and gyroscope), and three of these sensors are mostly software-based (the gravity, linear acceleration, and rotation vector sensors). This tutorial teaches the developers how to leverage Android's new "gravity sensor" to improve the performance of their accelerometer-based applications such as portrait-landscape switch, racing games, or flight simulation games.

4.1 Preliminary Setup and Overview

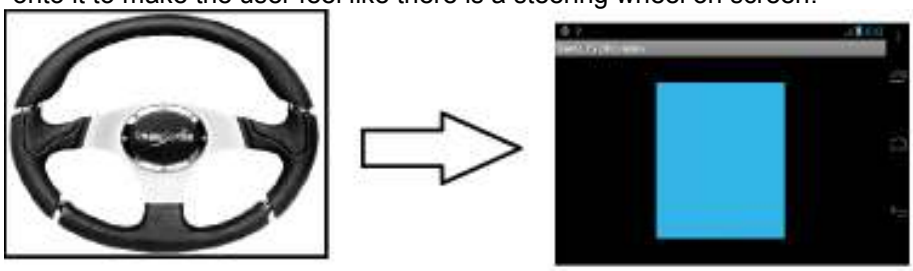
Before starting this tutorial please go through the Android Motion App Tutorial [8] to learn how to use OpenGLES along with setting up and activating SensorManager. This tutorial will assume that the developer has plenty of knowledge with working with OpenGL ES and has development experience with the Android SDK.

From the previous code a new class will be made that will be responsible for drawing a flat 2D steering wheel texture onto the screen. There are many ways to do this and this tutorial will do it through OpenGLES20 with the use of fragment and vertex shaders. For more information on how these work please refer to the OpenGL tutorial section of Android. The steps of drawing a steering wheel and rotating it on screen with OpenGLES would be as follow:

1. Define the vertices of a square in OpenGL which will be drawn on the X-Y Plan:



2. With this square, a steering wheel image with a transparent background can be pasted onto it to make the user feel like there is a steering wheel on screen.



3. The square can then be rotated like any shape, since we want to rotate it like a steering wheel it will be rotated in the center around the Z-axis.
   The outputted texture class will take in a matrix that will rotate the wheel just like the cube. The steering without any motion will look like this:

4.2 Modifying the Renderer to Rotate the Texture Class in Z Axis

Before this wheel can be rotated with sensor motion a public variable mAngle must be declared. This is done so the sensor values outside of this class can input what angle the wheel should steer out. With the angle, the mRotationMatrix value can be written to correspond to this rotation. The previous mRotation code will be commented out so it doesn't affect the new rotation Value. The drawFrame function should look like this:

```
@Override
public void onDrawFrame(GL10 unused) {
    // Draw background color
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);

    // Set the camera position (View matrix)
    Matrix.setLookAtM(mVMatrix, 0, 0, 0, 2, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    // Calculate the projection and view transformation
    Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mVMatrix, 0);


    // Set Rotation Matrix with the quaternion variable
   /* Matrix.setRotateM(mRotationMatrix, 0,
            (float) (2.0f * Math.acos(quat[0]) * 180.0f / Math.PI),
           quat[1], quat[2], quat[3]);*/

    // Set the rotation matrix to rotate mAngle around the Z-axis
    Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, 1);

    //Apply Rotation Matrix to View Matrix
    Matrix.multiplyMM(mMVPMatrix, 0, mRotationMatrix, 0, mMVPMatrix, 0);

   // mCube.draw(mMVPMatrix);
    //Draw the Texture Wheel
    mTexture.draw(mMVPMatrix);
}
```

4.3 Activating the Proper Sensors

After the Renderer class has been setup, the MyGLSurfaceView class should now be modified to activate and listen to the proper sensors. The two types of sensors we are most interested in will be:

- TYPE_GRAVITY
- TYPE_ACCELEROMETER

Since this demo will have the option of choosing between the gravity sensor and the acceleration sensor during runtime, both of these sensors will be activated at the start of the application. The startSensors() function will look like this:

```
// Only start the sensors when the program is running to save power
public void startSensors(){
    //get the sensor list
    List<Sensor> listSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);

    //iterate through sensor list and activate desired sensor
    if (listSensors.size() > 0) {
        for(int i = 0; i<listSensors.size(); i++){
            Sensor itemSensor = listSensors.get(i);

            switch (itemSensor.getType()){
            case Sensor.TYPE_GRAVITY:
                mSensorManager.registerListener(this, itemSensor,
                        SensorManager.SENSOR_DELAY_GAME);
                break;
            case Sensor.TYPE_ACCELEROMETER:
                mSensorManager.registerListener(this, itemSensor,
                        SensorManager.SENSOR_DELAY_GAME);
                break;
            case Sensor.TYPE_ROTATION_VECTOR:
                //Activate only the Invensense's sensor
                if(itemSensor.getVendor().equals("Invensense")){
                    mSensorManager.registerListener(this, itemSensor,
                            SensorManager.SENSOR_DELAY_GAME);
                }
                break;
            default:
                break;
            }
        }
    }

}
```

4.4 Menu Setup to Toggle between Sensors

In order to toggle between the two sensors a Dialog Menu can be set up to switch between the sensors. This dialog will pop up whenever the screen is pushed and held for a certain amount of time. The code to setup the dialog can be seen below (The third toggle field will be used to select a filtered version of the Accelerometer):

```
menu = new Dialog(this);
menu.setContentView(R.layout.sensor_selection);
RadioGroup group = (RadioGroup)menu.findViewById(R.id.radioGroup1);
group.setOnCheckedChangeListener(new OnCheckedChangeListener(){

    @Override
```

```
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                // TODO Auto-generated method stub
                Log.v("text", "checked change");
                int checked = group.getCheckedRadioButtonId();
                if(checked == R.id.radio0){
                    //Log.v("text", "checked change " + checked);
                    mGLView.wheelControl = 0;
                }else if(checked == R.id.radio1){
                    mGLView.wheelControl = 1;
                }else if(checked == R.id.radio2){
                    mGLView.wheelControl = 2;
                }
            }});
```

This is what the dialog menu will look like when the screen is pushed and held:

4.5 Handling the Sensor Data

During the onSensorChanged callback, the variable wheelControl can be used to determine which sensor value can be used to determine the angle. Since the device will be viewed in landscape mode, the accelerometer values we are most interested in will be on the Y axis. Whenever the accelerometer value in the Y axis is at 1G, we can assume the wheel will be rotated at a 90 degree angle. The code to set the mAngle variable during the onSensorChanged callback will be the following:

```
         if(event.sensor.getType() == Sensor.TYPE_GRAVITY){
            if(wheelControl == 0){
                mRenderer.mAngle = event.values[1]*90/9.8f;
                requestRender();
            }

        }else if(event.sensor.getType() == Sensor.TYPE_ACCELEROMETER){
            if(wheelControl == 1){
                mRenderer.mAngle = event.values[1]*90/9.8f;
                requestRender();
            }else if(wheelControl == 2){
                currAccelY = event.values[1] * filterFactor + currAccelY *(1.0f -
filterFactor);
                mRenderer.mAngle = currAccelY*90/9.8f;
```

```
                requestRender();
            }
        }
```

5. Performance Comparison

5.1 Gravity vs. Raw Accel vs. Filtered Accel

Now the steering wheel can be rotated with either of the three sensor values. Using the raw accelerometer output will make the steering wheel very jittery. The jittery behavior comes from not only the noise level on the raw accelerometer data, but also the linear and centripetal accelerations that the accelerometer senses in addition to the earth gravity. Figure 1 shows the raw accelerometer data when the device turns from portrait mode to landscape mode. Notice the overshoot at the transition due to the excessive linear acceleration. To remove this jitter one can low-pass filter the accelerometer values to reduce the noise (in this example we use 1-tap IIR filter with filterFactor = 0.01, meaning that only 1 percent of the current accelerometer value contributes to the accumulated result). The drawback of putting a low-pass filter would be the lag. Figure 1 (left) illustrates the latency due to the low pass filtering. This is one of the reasons why most racing games do not have a fast response when the user turns the device.

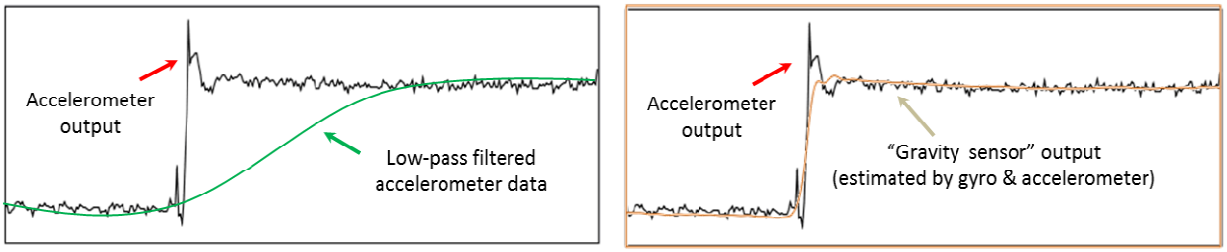


Figure 1. Low-pass filtered accelerometer data (left) and gravity sensor output (right).

To eliminate both noise and lag, we recommend Android app developers use the gravity sensor to detect the device tilt angle. For Android devices equipped with gyroscopes, the gravity sensor is usually implemented using sensor fusion with accelerometers and gyroscopes. Gyroscopes track the rotation movement instantly and with immunity to linear or centripetal accelerations. By fusing the data from gyroscopes and accelerometers, the device can generate gravity vector with high precision without delay. Figure 1 (right) shows the gravity sensor output from Invensense's Motion Processing Library (MPL). Comparing to low-pass filtered accelerometer data, the gravity sensor output is also smooth, but with no latency problem.

5.2 Gravity vs. Rotation Vector

For app developers who use accelerometer readings to determine the tilt angle (e.g. pitch or roll) of the device, replacing accelerometer data with gravity sensor output will give a smoother and instant motion response, as mentioned in previous sections. If the developers want to extend their apps to respond to heading or horizontal rotation as well (e.g. yaw), they can listen to another Android sensor event called the "rotation vector". Android rotation vector reports the 3D orientation of the device instantly using the sensor fusion result from accelerometers, gyroscopes, and magnetometers. Interested readers can refer to [8] for more details about the rotation vector.

6. Reference

[1] http://developer.android.com/training/graphics/opengl/index.html
[2] http://developer.android.com/guide/topics/graphics/opengl.html
[3] http://developer.android.com/guide/topics/sensors/index.html
[4] http://developer.android.com/index.html
[5] http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
[6] http://developer.android.com/guide/topics/sensors/sensors_motion.html#sensors-motion-rotate
[7] http://developer.android.com/training/graphics/opengl/projection.html
[8] Making Android 3D Graphics Applications Respond to Motion"