Android: Gyroscope Basics

Written by Super User

Android: Gyroscope Basics

Discusses basic usage of the Android device's gyroscope sensor. How a gyroscope works, how to compensate for drift and implementing a gyroscope sensor on and Android device.

Author: Kaleb Kircher

Version: 2.0

4.10.14.2.0

Refactoring History

v1.0	12.1.13.1.0	Authored
v2.0	4.10.14.2.0	Refractored

Associated Code Projects:

Gyroscope Explorer:

Gyroscope Explorer is a code example and working project that contains Android classes that demonstrate how to use the Sensor.TYPE_GYROSCOPE and Sensor.TYPE_GYROSCOPE_UNCALIBRATED. This includes integrating the sensor outputs over time to describe the devices change in angles, initializing the rotation matrix, concatination of the new rotation matrix with the initial rotation matrix and providing an orientation for the concatenated rotation matrix. The Android developer documentation covers some of this information, but it is an incomplete example that is fully implemented within Gyroscope Explorer. Gyroscope Explorer provides the Earth frame orientation with the azimuth, pitch and roll and described in a clean graphical view.

Download Gyroscope Explorer for free in the Google Play Store!

Fork the Gyroscope Explorer source code from GitHub!

How Gyroscopes Work:

Most gyroscopes on Android devices are vibrational and measure the rotation of a device with a pair of vibrating arms that take advantage of what is known as the Coriolis effect, which is caused by the Earth's rotation. By measuring changes in the direction of the vibrating arms caused by a rotation and the Coriolis effect, an estimation of the rotation can be produced. The gyroscope is one of three sensors that are always hardware based (the other two are the magnetic and the acceleration sensors) on Android devices. In conjunction with the acceleration sensor, the gyroscope can be used to create other sensors like gravity, linear acceleration or rotation sensors. These sensors are all useful for detecting the movement of the device, which can either be a result of the user inputs (moving the device to control a character on a game) or an external physical environment (like the movement of a car). They can also be used indirectly to determine the position of a device, like tilt-compensation on the magnetic sensor for a compass.

Gyroscope Drift

Like all sensors, a gyroscope is not perfect and has small errors in each measurement. Since the measurements from a gyroscope are integrated over time, these small errors start to add up and result in what is known a drift. Over time, the results of the integration can become unreliable and some form of compensation is required to help compensate for the drift. This requires another sensor to provide a second measurement of the devices orientation that can then be used to augment the gyroscopes integration back towards the actual rotation of the device. This second sensor is usually a acceleration or magnetic sensor, or sometimes both. A weighted average, Kalman filter or complementary filter are common implementations of fusing other sensors to the gyroscope sensor, each with their own advantages and disadvantages. When you really get down into the implementations, you also run into real limitations with the "support" sensors as well. For instance, an acceleration sensor cannot determine the difference between the tilt of the device and linear acceleration, which makes for a vicious circular reference when trying to implement a linear acceleration sensor. In fact, the Android Sensor.TYPE_LINEAR_ACCELERATION is terrible at measuring linear acceleration under the influence of a physical environment such as the acceleration of a car because of the circular reference. The magnetic sensor is another option, but it is limited by the effects of hard and soft iron offsets and it can only measure roll and yaw, so it isn't perfect, either. It can take a lot of

effort, fine tuning and possibly multiple sensor fusions and calibrations to get reliable estimations.

Calibrated versus Uncalibrated:

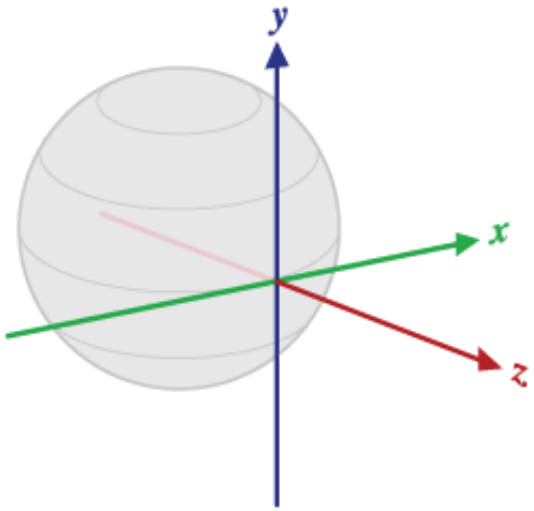
As of Android 4.3, a new uncalibrated gyroscope sensor is available. TYPE_GYROSCOPE UNCALIBRATED is similar to TYPE_GYROSCOPE, but no gyro-drift compensation has been performed to adjust the given sensor values. However, such gyro-drift bias values are returned to you separately in the result values so you may use them for custom calibrations. This allows you to implement your own sensor fusions without having to worry about black-boxed underlying sensor fusions wrecking your carefully implemented custom calibrations and fusions. Gyroscope Explorer displays the orientation from both the TYPE_GYROSCOPE and TYPE_GYROSCOPE_UNCALIBRATED. On the Nexus 4 and Nexus 5 devices, the uncalibrated gyroscope actually works fairly well on its own, but will eventually drift over long periods of time or after a lot of dynamic rotation. Since hardware implementations vary with each device and manufacturer, this may not be the case will all Android devices.

Coordinate Systems:

There are a number of coordinate systems to be aware of when developing with Android devices.

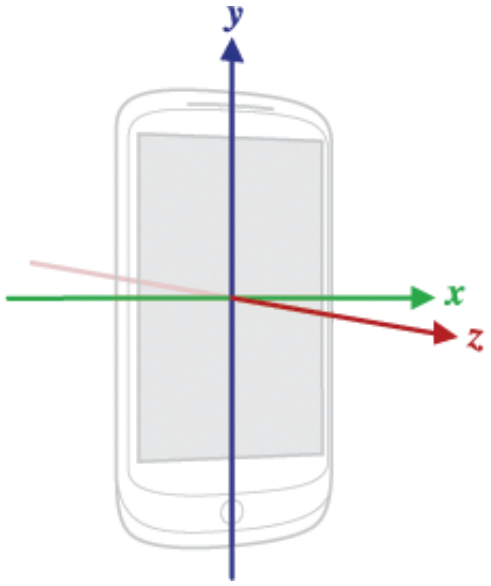
World Coordinate System:

The world coordinate system in Android is the ENU (east, north, up) coordinate system. This is *different* from the NED (north, east, down) coordinate system that is commonly used in aviation.



Local Coordinate System:

The local coordinate system describes the coordinate system of the device. For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation (see figure 1). When a device is held in its default orientation, the X axis is horizontal and points to the right, the Y axis is vertical and points up, and the Z axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative Z values. The most important point to understand about this coordinate system is that the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device moves.

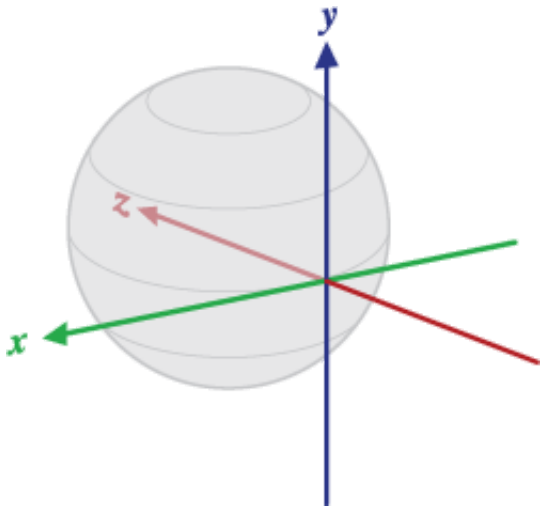


Other Coordinate Systems:

The method SensorManager.getOrientation(), which is commonly used to get the orientation vector of the device from a rotation matrix, uses a third reference coordinate system that is not the world coordinate system. A WND (west, north, down) coordinate system is used, which is different from both the ENU and NED coordinate systems that are more common. Also worth noting is that the order of the axis returned in the method are different from those returned by the sensors.

When SensorManager.getOrientation() returns, the array values is filled with the result:

- values[0]: azimuth, rotation around the Z axis.
- values[1]: pitch, rotation around the X axis.
- values[2]: roll, rotation around the Y axis.



Integration Over Time:

The gist of the integration comes from the Android Developer documentation. First, the axis-angle representation is normailzed (a requirement of quaternions) by taking the magnitude of the vector and then dividing each element in the vector by the magnitude. The derivative is then taken with respect to time and then the axis-angle representation is converted into a quaternion. The quaternion is then converted into a rotation matrix (via SensorManager.getRotationMatrixFromVector()) which can be applied to another rotation matrix representing the current orientation of the device. Finally, the matrix representing the current orientation is converted into a vector representing the orientation of the device (via SensorManager.getOrienation()).

Some things that aren't explained very well in the documentation...

Why divide each element in the axis-angle vector by the magnitude of the vector? SensorManager.getRotationMatrixFromVector() requires a unit quaternion and, by definition, the three elements of the rotation vector are equal to the last three components of a unit quaternion <cos(θ/2), x*sin(θ/2), y*sin(θ/2), z*sin(θ/2)>. A vector, 'u' is a unit quanternion if its norm is one, | u | = 1. We normalize the axis-angle vector so we can convert it into a quaternion.

Why is the sensors delta time (sensor period, dt, time between updates, etc...) divided by 2? This comes from the θ/2 in each of the quaternions elements because we want half angles for our quaternions. Instead of dividing theta by 2 in each of the four elements, we divide it once when we

determine the sensors delta time.

What is a good value for EPSILON? A value that is just large enough to normal the vector, 'u', so very small, just larger than 0.

The Integration Code:

```
private static final float NS2S = 1.0f / 1000000000.0f;
private final float[] deltaRotationVector = new float[4]();
private float timestamp;

public void onSensorChanged(SensorEvent event) {
     // This timestep's delta rotation to be multiplied by the current rotation
     // after computing it from the gyro sample data.
     if (timestamp != 0) {
         final float dT = (event.timestamp - timestamp) * NS2S;
         // Axis of the rotation sample, not normalized yet.
         float axisX = event.values[0];
         float axisY = event.values[1];
         float axisZ = event.values[2];

         // Calculate the angular speed of the sample
         float omegaMagnitude = sqrt(axisX*axisX + axisY*axisY + axisZ*axisZ);

         // Normalize the rotation vector if it's big enough to get the axis
         if (omegaMagnitude > EPSILON) {
             axisX /= omegaMagnitude;
             axisY /= omegaMagnitude;
             axisZ /= omegaMagnitude;
         }

         // Integrate around this axis with the angular speed by the timestep
         // in order to get a delta rotation from this sample over the timestep
         // We will convert this axis-angle representation of the delta rotation
         // into a quaternion before turning it into the rotation matrix.
         float thetaOverTwo = omegaMagnitude * dT / 2.0f;
         float sinThetaOverTwo = sin(thetaOverTwo);
         float cosThetaOverTwo = cos(thetaOverTwo);
         deltaRotationVector[0] = sinThetaOverTwo * axisX;
         deltaRotationVector[1] = sinThetaOverTwo * axisY;
```

```
            deltaRotationVector[2] = sinThetaOverTwo * axisZ;
            deltaRotationVector[3] = cosThetaOverTwo;
        }
        timestamp = event.timestamp;
        float[] deltaRotationMatrix = new float[9];
        SensorManager.getRotationMatrixFromVector(deltaRotationMatrix, deltaRotationVector);
        // User code should concatenate the delta rotation we computed with the current rotation
        // in order to get the updated rotation.
        // rotationCurrent = rotationCurrent * deltaRotationMatrix;
    }
```

We will also need to get the initial rotation matrix from the acceleration and magnetic sensors. Most folks will want to start with a standard basis oriented to the world frame (East, North, Up) based on the current orientation of the device. This requires determining the current orientation of the device in the world coordinate system and can be done with the acceleration and magnetic sensors with a call to SensorManager.getRotationMatrix(). The matrix that is returned will then be multiplied by the delta rotation matrix that is produced from the gyroscope measurements to produce a new rotation matrix representing the current orientation.

If you do not take this step, you can create an initial orientation based on the rotation of the device when the algorithm starts with the identity matrix. This will create a world frame that is oriented to the initial rotation of the device and all further device frame rotations will be relative to the initial world frame. The identity matrix will be multiplied by the delta rotation matrix that is produced from the gyroscope measurements and the integration method. The identity matrix will be multiplied by the delta rotation matrix that is produced from the gyroscope measurements to produce a new rotation matrix representing the current orientation

The Initial Orientation Code:

```
private void calculateInitialOrientation()
{
    hasInitialOrientation = SensorManager.getRotationMatrix(
            initialRotationMatrix, null, acceleration, magnetic);

}
```

The Concatenate the Rotation Code:

We can apply a rotation matrix to another rotation matrix by multiplying the two rotation matrices together. This is how the orientations of the axis-angle vectors produced by the gyroscope are integrated. A rotation matrix representing each axis-angle vector is produced by the gyroscope and then applied to the rotation matrix representing the last known orientation of the device. The easiest way to do this is to create a function to multiply two 3x3 matrices together.

```
private float[] matrixMultiplication(float[] a, float[] b)
    {
        float[] result = new float[9];

        result[0] = a[0] * b[0] + a[1] * b[3] + a[2] * b[6];
        result[1] = a[0] * b[1] + a[1] * b[4] + a[2] * b[7];
        result[2] = a[0] * b[2] + a[1] * b[5] + a[2] * b[8];

        result[3] = a[3] * b[0] + a[4] * b[3] + a[5] * b[6];
        result[4] = a[3] * b[1] + a[4] * b[4] + a[5] * b[7];
        result[5] = a[3] * b[2] + a[4] * b[5] + a[5] * b[8];

        result[6] = a[6] * b[0] + a[7] * b[3] + a[8] * b[6];
        result[7] = a[6] * b[1] + a[7] * b[4] + a[8] * b[7];
        result[8] = a[6] * b[2] + a[7] * b[5] + a[8] * b[8];

        return result;
    }
```

The Orientation Code:

Once we have our new rotation matrix, all we have to do is make a call to SensorManager.getOrientation() to get the orientation of the device. Note that the reference coordinate-system used is different from the world coordinate-system defined for the rotation matrix and the order of the axis returned by the method are different from the order returned by the sensors.

```
SensorManager.getOrientation(currentRotationMatrixCalibrated,
        gyroscopeOrientationCalibrated);
```

The Finished Code:

```
public void onGyroscopeSensorChanged(float[] gyroscope, long timestamp)
```

```
    {
        // don't start until first accelerometer/magnetometer orientation has
        // been acquired
        if (!hasInitialOrientation)
        {
            return;
        }

        // Initialization of the gyroscope based rotation matrix
        if (!stateInitializedCalibrated)
        {
            currentRotationMatrixCalibrated = matrixMultiplication(
                    currentRotationMatrixCalibrated, initialRotationMatrix);

            stateInitializedCalibrated = true;
        }

        // This timestep's delta rotation to be multiplied by the current
        // rotation after computing it from the gyro sample data.
        if (timestampOldCalibrated != 0 && stateInitializedCalibrated)
        {
            final float dT = (timestamp - timestampOldCalibrated) * NS2S;

            // Axis of the rotation sample, not normalized yet.
            float axisX = gyroscope[0];
            float axisY = gyroscope[1];
            float axisZ = gyroscope[2];

            // Calculate the angular speed of the sample
            float omegaMagnitude = (float) Math.sqrt(axisX * axisX + axisY
                    * axisY + axisZ * axisZ);

            // Normalize the rotation vector if it's big enough to get the axis
            if (omegaMagnitude > EPSILON)
            {
                axisX /= omegaMagnitude;
                axisY /= omegaMagnitude;
                axisZ /= omegaMagnitude;
            }

            // Integrate around this axis with the angular speed by the timestep
```

```
43              // in order to get a delta rotation from this sample over the
44              // timestep. We will convert this axis-angle representation of the
45              // delta rotation into a quaternion before turning it into the
46              // rotation matrix.
47              float thetaOverTwo = omegaMagnitude * dT / 2.0f;
48
49              float sinThetaOverTwo = (float) Math.sin(thetaOverTwo);
50              float cosThetaOverTwo = (float) Math.cos(thetaOverTwo);
51
52              deltaRotationVectorCalibrated[0] = sinThetaOverTwo * axisX;
53              deltaRotationVectorCalibrated[1] = sinThetaOverTwo * axisY;
54              deltaRotationVectorCalibrated[2] = sinThetaOverTwo * axisZ;
55              deltaRotationVectorCalibrated[3] = cosThetaOverTwo;
56
57              SensorManager.getRotationMatrixFromVector(
58                      deltaRotationMatrixCalibrated,
59                      deltaRotationVectorCalibrated);
60
61              currentRotationMatrixCalibrated = matrixMultiplication(
62                      currentRotationMatrixCalibrated,
63                      deltaRotationMatrixCalibrated);
64
65              SensorManager.getOrientation(currentRotationMatrixCalibrated,
66                      gyroscopeOrientationCalibrated);
67          }
68
69          timestampOldCalibrated = timestamp;
70      }
```

Acceleration Sensors, Sensor, Gyroscope,