Making Android 3D Graphics Applications
Respond to Motion

This information furnished by InvenSense is believed to be accurate and reliable. However, no responsibility is assumed by InvenSense for its use, or for any infringements of patents or other rights of third parties that may result from its use. Specifications are subject to change without notice. InvenSense reserves the right to make changes to this product, including its circuits and software, in order to improve its design and/or performance, without prior notice. InvenSense makes no warranties, neither expressed nor implied, regarding the information and specifications contained in this document. InvenSense assumes no responsibility for any claims or damages arising from information contained in this document, or from the use of products and services detailed therein. This includes, but is not limited to, claims or damages based on the infringement of patents, copyrights, mask work and/or other intellectual property rights.

Certain intellectual property owned by InvenSense and described in this document is patent protected. No license is granted by implication or otherwise under any patent or patent rights of InvenSense. This publication supersedes and replaces all information previously supplied. Trademarks that are registered trademarks are the property of their respective companies. InvenSense sensors should not be used or sold in the development, storage, production or utilization of any conventional or mass-destructive weapons or for any other weapons or life threatening applications, as well as in any other life critical applications such as medical equipment, transportation, aerospace and nuclear instruments, undersea equipment, power plant equipment, disaster prevention and crime prevention equipment.

Copyright ©2011 InvenSense Corporation.
# Table of Contents

1. Revision History ................................................................. 3
2. Purpose .................................................................................. 3
3. Requirements ........................................................................ 3
4. Making Your App Respond to Device Motion ...................... 4
   4.1 Detecting the Presence of Motion Sensors ..................... 5
   4.2 Adding Motion Sensor Event Listener to an OpenGL ES 2.0 Graphics App ........................................ 6
   4.3 Defining and Drawing 3D Shapes ..................................... 9
   4.4 Move the Graphics Objects with Device Motion .............. 12
5. Reference ............................................................................... 13
1. Revision History

<table>
<thead>
<tr>
<th>Revision Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/26/2012</td>
<td>1.0</td>
<td>Document created</td>
</tr>
</tbody>
</table>

2. Purpose

This document is a quick start guide on how to provide one-to-one motion on a 3D graphics object using OpenGL and Android's Sensor Manager. Knowledge of Android's SDK and application development is needed before reading this document. Some knowledge of OpenGL is also useful when creating this application. Please refer to the documentation provided in the reference section for more information on Android’s OpenGL development and working with Android's Sensor Manager.

3. Requirements

- Eclipse
- Android SDK
- Java Programming Knowledge
- Knowledge of quaternion and rotations
- Completion of Android’s OpenGL ES tutorial
4. **Making Your App Respond to Device Motion**

Most Android-powered devices have built-in sensors that measure motion. 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. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

The Android platform provides six types of sensors that let you monitor the motion of a device. Three of these sensors are always hardware-based (the accelerometer, magnetometer, and gyroscope), and three of these sensors can be either hardware-based or software-based (the gravity, linear acceleration, and rotation vector sensors). For example, on some devices the software-based sensors derive their data from the accelerometer and magnetometer, but on other devices they may also use the gyroscope to derive their data. Most Android-powered devices have an accelerometer, and many now include a gyroscope. The availability of the software-based sensors is more variable because they often rely on one or more hardware sensors to derive their data.

This tutorial document shows you how to use the sensed motion information to animate the graphics object in your app. There are four sections in this tutorial:

1. **Detecting the Presence of Motion Sensors**: This section teaches how to make the app visible only to the devices that have built in motion sensors.
2. **Adding Motion Sensor Event Listener to an OpenGL ES 2.0 Graphics App**: This section teaches how to make the graphic app receive motion sensor data stream.
3. **Defining and Drawing 3D Shapes**: This section shows how to create a simple 3D cube in OpenGL ES 2.0 environment.
4. **Move the Graphics Objects with Device Motion**: This section teaches how to add the quaternion, which is derived from motion sensor input, to the viewing transformation matrix of the graphics object, so that the object responds to device motion.

Note that the example code in this class is built upon the sample code from the prerequisite class “Displaying Graphics with OpenGL ES” in the Android Training site. Readers are encouraged to go through the link first before continuing this class.
4.1 Detecting the Presence of Motion Sensors

Before we make our Android app respond to motion, we first need to check whether the device is equipped with motion sensors or not. To do so we add the following <user-feature> elements to the manifest file of our app:

```xml
<uses-feature android:name="android.hardware.sensor.accelerometer" android:required="true"/>
<uses-feature android:name="android.hardware.sensor.gyroscope" android:required="true"/>
<uses-feature android:name="android.hardware.sensor.compass" android:required="true"/>
```

Google Play uses <user-feature> elements to filter the applications that are visible to users, so that users can see and download only those applications which are compatible with their devices. Therefore, by adding the above three lines to the .xml manifest file of your app, Google Play will guarantee that your app will only be installed to the devices that have motion sensors.

For more detailed information about <user-feature> element and manifest file, please go to the “Google Play and Feature-Based Filtering” section in Android Developer site.

Note that although not all the Android-powered devices have built-in gyroscope, it is highly recommended that the motion-based apps run only on those devices which have gyroscopes. The gyroscope captures the rotation movement in high precision, and therefore it is an essential component to derive the rotation vector. The rotation vector derived from only the accelerometer and magnetometer tends to have delayed response and jitter problems.
4.2 Adding Motion Sensor Event Listener to an OpenGL ES 2.0 Graphics App

To register and activate sensors in the Android SensorManager to the OpenGL ES 2.0 app, we modify the code in the following steps:

1. Implement the SensorEventListener interface to the OpenGL ES SurfaceView class,
2. Declare a public variable in GLRenderer class to store sensor data for graphics use,
3. Modify the callback function OnSensorChanged() so that it stores the sensor data to the designated public variable whenever the data is available,
4. Register the sensor that we want to receive data from (in this example we listen to the rotation vector),
5. Activate the registered sensor.

We start by implementing the SensorEventListener interface to the MyGLSurfaceView class, the dedicated surface in our sample code for displaying OpenGL rendering:

```java
class MyGLSurfaceView extends GLSurfaceView implements SensorEventListener{

    private final MyGLRenderer mRenderer;
    private SensorManager mSensorManager;

    public volatile float[] quat = {1,0,0,0};

    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR){
            SensorManager.getQuaternionFromVector(mRenderer.quat, event.values);
            requestRender();
        }
    }
}
```

For more information about how the rotation vector and quaternion represent the orientation of the device, please check Reference [5] and [6].

Now that SensorEventListener has been implemented, the SensorManager can now be configured to activate Sensors. In the GLSurfaceView constructor get an instance of the SensorManager like this:

```java
mSensorManager = (SensorManager) context.getSystemService(Context.SENSOR_SERVICE);
```

The overall constructor will be:

```java
public MyGLSurfaceView(Context context) {
    super(context);
}
```
// Create an OpenGL ES 2.0 context.
setEGLContextClientVersion(2);

// Set the Renderer for drawing on the GLSurfaceView
mRenderer = new MyGLRenderer();
setRenderer(mRenderer);

// Get instance of SensorManager so that sensors can be registered/unregistered
mSensorManager = (SensorManager) context.getSystemService(Context.SENSOR_SERVICE);

// Render the view only when there is a change in the drawing data
setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
}

The StartSensors() and StopSensors() functions are created in order for the main activity to have control over whether or not the sensors should be activated. The Main Activity should deactivate the sensors whenever it is not running so that power can be saved.

// 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_ROTATION_VECTOR:
                    //Activate only the Invensense's sensor
                    if(itemSensor.getVendor().equals("Invensense")){
                        mSensorManager.registerListener(this, itemSensor,
                            SensorManager.SENSOR_DELAY_GAME);
                    }
                    break;
                default:
                    break;
            }
        }
    }
}

// Stop the sensors if they are not needed to save power.
public void stopSensors(){
    mSensorManager.unregisterListener(this);
Grabbing the sensor list with the `mSensorManager.getSensorList(Sensor.TYPE_ALL);` function call will return a list of the available sensors that can be used. Each type of phone may have varying sensors so it is a good practice to look through this list and view at what sensors are available.

When the `mSensorManager.registerListener` function call is called one parameter to highlight is the data rate that the sensor will be set to. The available data rates that the sensors can be set to are:

<table>
<thead>
<tr>
<th>int</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_DELAY_FASTEST</td>
<td>get sensor data as fast as possible (100 HZ)</td>
</tr>
<tr>
<td>SENSOR_DELAY_GAME</td>
<td>rate suitable for games (50HZ)</td>
</tr>
<tr>
<td>SENSOR_DELAY_NORMAL</td>
<td>rate (default) suitable for screen orientation changes (5HZ)</td>
</tr>
<tr>
<td>SENSOR_DELAY_UI</td>
<td>rate suitable for the user interface (15HZ)</td>
</tr>
</tbody>
</table>

When activating the TYPE_ROTATION_VECTOR sensor it is possible that there can be more than one implementation of this type. To guarantee the performance it is recommended to check the vendor info by function `getVendor()` to ensure that the sensor data comes from the desired vendor. In this example we use the Rotation Vector from InvenSense.

The Main Activity can now be modified to start and stop the sensors in this class whenever it needs to.

```java
@Override
protected void onPause() {
    super.onPause();
    mGLView.stopSensors();
    // The following call pauses the rendering thread.
    // If your OpenGL application is memory intensive,
    // you should consider de-allocating objects that
    // consume significant memory here.
    mGLView.onPause();
}

@Override
protected void onResume() {
    super.onResume();
    // The following call resumes a paused rendering thread.
    // If you de-allocated graphic objects for onPause()
    // this is a good place to re-allocate them.
    mGLView.onResume();
    mGLView.startSensors();
}
```
4.3 Defining and Drawing 3D Shapes

Now that we have linked the motion sensor to the graphics Surface View, the next step is to extend the 2D square object in the OpenGL ES example code to 3D cube, and make it respond to the 3D motion. Defining a 3D cube will be exactly like defining a square except that it will require more coordinates and the drawOrder will increase. Like the square, only the 8 corners of the square will need to be defined since the points can be reused to define all of the triangles. Here’s the code for the cube:

```java
class Cube {
    private final FloatBuffer vertexBuffer;
    private final ShortBuffer drawListBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float cubeCoords[] = {
        //FRONT FACE
        -0.25f, 0.25f, 0.25f, // top left
        -0.25f, -0.25f, 0.25f, // bottom left
        0.25f, -0.25f, 0.25f, // bottom right
        0.25f, 0.25f, 0.25f, // top right
        //BACK FACE
        -0.25f, 0.25f, -0.25f, // top left
        -0.25f, -0.25f, -0.25f, // bottom left
        0.25f, -0.25f, -0.25f, // bottom right
        0.25f, 0.25f, -0.25f// top right
    };

    private final short drawOrder[] = {
        0, 1, 2, 0, 2, 3, //front
        4, 7, 6, 5, 4, //back
        0, 4, 5, 5, 1, 0, //Left
        3, 2, 6, 6, 7, 3, //Right
        1, 5, 6, 6, 2, 1, //Bottom
        0, 3, 7, 7, 4, 0}; // order to draw vertices

    public Cube() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
            // (# of coordinate values * 4 bytes per float)
            cubeCoords.length * 4);
        bb.order(ByteOrder.nativeOrder());
        vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(cubeCoords);
        vertexBuffer.position(0);

        // initialize byte buffer for the draw list
        ByteBuffer dlb = ByteBuffer.allocateDirect(
            // (# of coordinate values * 2 bytes per short)
            drawOrder.length * 2);
        dlb.order(ByteOrder.nativeOrder());
        drawListBuffer = dlb.asShortBuffer();
        drawListBuffer.put(drawOrder);
    }
}
```
Making Android 3D Graphics Applications
Respond to Motion

drawListBuffer.position(0);
}

Figure 1. The cube object defined in this example

If we are to follow the square’s draw method the cube will end up having one color and becoming flat. To avoid this 6 different face colors can be declared and whenever the draw function is called each of those faces can have a different color set.

// Declare different face colors for each of the faces
float color1[] = { 0.8f, 0.0f, 0.0f, 1.0f };
float color2[] = { 0.0f, 0.8f, 0.0f, 1.0f };
float color3[] = { 0.0f, 0.0f, 0.8f, 1.0f };
float color4[] = { 0.8f, 0.8f, 0.0f, 1.0f };
float color5[] = { 0.8f, 0.0f, 0.8f, 1.0f };
float color6[] = { 0.0f, 0.8f, 0.8f, 1.0f };

CULL_FACE is enabled so that OpenGL is able to differentiate whether a drawn face is facing forward or backward. Without setting CULL_FACE the faces will be drawn in order and some of the faces will look like they are in front of one face when in reality they are the opposite.

GLES20.glEnable(GLES20.GL_CULL_FACE);  // for determining whether shape is front facing

In the cube’s draw function ensure that each color is set every time and that each face is treated as a separate draw case. The example code can be seen here:

//draw the cube
//each face will be drawn separately with different colors
drawListBuffer.position(0);
gles20.glUniform4fv(mColorHandle, 1, color1, 0);
gles20.glDrawElements(GLES20.GL_TRIANGLES, drawOrder.length/6,
                     GLES20.GL_UNSIGNED_SHORT, drawListBuffer);

gles20.glUniform4fv(mColorHandle, 1, color2, 0);
drawListBuffer.position(6);
gles20.glDrawElements(GLES20.GL_TRIANGLES, drawOrder.length/6,
                     GLES20.GL_UNSIGNED_SHORT, drawListBuffer);
In order to draw this cube onto the buffer the onDrawFrame can be modified to call the Cube’s draw function. After the Cube object has been initialized that onDrawFrame will look like this:

```java
@override
public void onDrawFrame(GL10 unused) {
    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    Matrix.setLookAtM(mVMatrix, 0, 0, 0, 3, 0f, 0f, 0f, 1.0f, 0.0f);

    mCube.draw(mMVPMatrix);
}
```

The result of this will be a cube with no movement:
4.4 Move the Graphics Objects with Device Motion

We have connected the sensor data stream into our app, and we have created a 3D object in the Surface View. The last step is to make the 3D cube rotate based on the received motion sensor data. We do so by modifying the OnDrawFrame() function. The main operation in the callback function OnDrawFrame() is to calculate the transformation matrix that projects the 3D coordinates of the drawn objects onto the display surface. To make the drawn object respond to the device orientation, we simply add one more transformation matrix to rotate the viewport based on the device's actual orientation, which is indicated by the quaternion data that we derived from the rotation vector that SensorManager gives us:

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

    // Set the camera position (View matrix)
    Matrix.setLookAtM(mVMatrix, 0, 0, 0, 3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    // Calculate the projection and view transformation
    Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mVMatrix, 0);
    Matrix.multiplyMM(mRotationMatrix, 0, (float) (2.0f * Math.acos(quat[0]) * 180.0f / Math.PI), quat[1], quat[2], quat[3]);
    Matrix.multiplyMM(mMVPMatrix, 0, mRotationMatrix, 0, mMVPMatrix, 0);
    mCube.draw(mMVPMatrix);
}

@Override
public void onSurfaceChanged(GL10 unused, int width, int height) {
    // Adjust the viewport based on geometry changes,
    // such as screen rotation
    GLES20.glViewport(0, 0, width, height);
    float ratio = (float) width / height;
    // this projection matrix is applied to object coordinates
    // in the onDrawFrame() method
```
Matrix.frustumM(mProjMatrix, 0, -ratio, ratio, -1, 1, 3, 20);

5. **Reference**