Tegra Android Accelerometer Whitepaper

Version 5

Contents

INTRODUCTION 3

COORDINATE SPACE GLOSSARY 4

ACCELEROMETER CANONICAL AXES 6

WORKING WITH ACCELEROMETER DATA 7

POWER CONSERVATION 10

SUPPORTING OLDER OS VERSIONS 11

Introduction

Portable devices can be used in a variety of orientations. Applications need to consider the ways in which orientation affects them, such as a custom UI for landscape versus portrait modes, or interpreting raw sensor data. Applications making use of the accelerometer must take special care when processing the raw data, given device and operating system expectations, in order to deliver a proper user experience.

While the accelerometer is an important input device to many software applications, some developers are using incorrect methods to process the accelerometer data. More to the point, these applications are not taking into account device orientation, which results in a poor if not failed user experience. Proper orientation handling of accelerometer input is a simple matter, which we will discuss in the sections that follow.

Coordinate Space Glossary

Android reports accelerometer values in absolute terms. The values reported are always the data from the physical sensor adjusted to match the Android "canonical" format so that all devices report such data in the same fashion. Android does not transform accelerometer data to be relative to the device orientation. Applications requiring this must perform their own transformations.

To that end, this document uses the following definitions of coordinate spaces and coordinate space transformations to maintain consistency:

Space	Description
Device Raw	The accelerometer device can output acceleration values in a variety of ways and is not subject to any particular standard.
Canonical	Android specifies that the coordinate frame outputted by the driver must remap *Device Raw* so that the positive X axis is oriented increasing to the right side of the device, the positive Y axis should be increasing to the top of the device, and the positive Z axis is increasing out the display of the device towards the user. See the "Accelerometer Canonical Axes" reference below for further visual guide to *Canonical* accelerometer layout.
Screen	The Android window manager's screen coordinate origin is at the upper left corner and the maximum coordinate is at the lower right corner, i.e. increasing x is right, increasing y is down. Android's display manager will change the screen orientation based on sensor readings. The screen coordinate system is always relative to the current rotation.
World	This coordinate space is specific to OpenGL ES applications. App developers may need to alter the sample code to fit their assumptions in this regard. In this document, it is assumed that applications are using a right-handed coordinate system, up can be any arbitrary vector. NOTE: Many applications will require additional transforms to those shown here to orient their models. Apps using left-handed coordinates may require an additional coordinate inversion as well.

The table below shows transforms of interest and defines a vocabulary for this paper. OpenGL applications will typically be using `canonToWorld`. Android windowing system based applications will use `canonToScreen`. Hybrid applications, such as OpenGL applications that use the Android window system to render widgets, will require both.

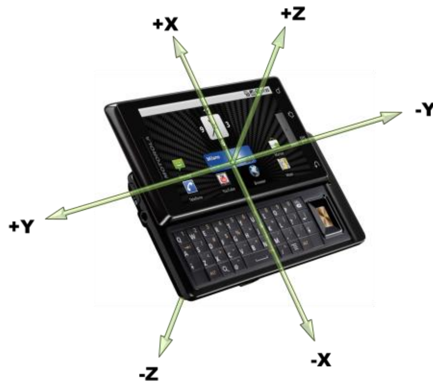
NOTE: The accelerometer device driver handles conversion into Canonical space (handling the `deviceToCanon` transform), this whitepaper focuses on the `canonToScreen` and `canonToWorld` transformations.

		Destination		
		Canonical	*Screen*	*World*
Source	*Device Raw*	`deviceToCanon`		
	Canonical		`canonToScreen`	`canonToWorld`

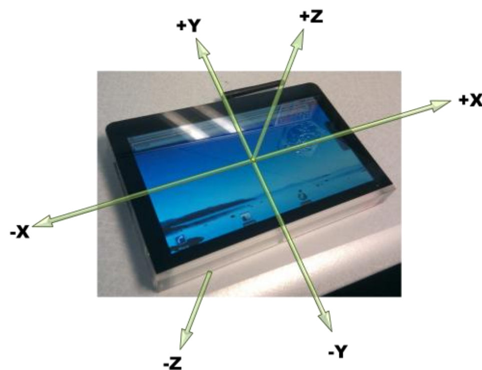
Accelerometer Canonical Axes

The following visuals show the variation in accelerometer values based upon given device and orientation. They include a portrait-native device, a portrait device rotated to landscape, and a landscape-native device, with *Canonical* x/y/z accelerometer axes/values labeled.

Phone Device
Portait Native
Orientation 0



Phone Device
Portrait Native
Orientation 90



Tablet Device
Landscape Native
Orientation 0

Working with Accelerometer Data

Where screen-relative results are desired, the accelerometer values must be rotated according to the display orientation returned by the Android API's `getOrientation()` or `getRotation()` functions. Both functions return the same values, but the former is a deprecated usage.

The value returned from these functions corresponds to the integers/constants defined in `Android.view.Surface`, those prefixed with `ROTATION_`. Below is an example of the invocation of one of these functions. Here `this` is of type `Activity`.

```
WindowManager windowMgr =
    (WindowManager)this.getSystemService(WINDOW_SERVICE);
int rotationIndex = windowMgr.getDefaultDisplay().getRotation();
```

The returned constants are:

constant name	index/value
ROTATION_0	0
ROTATION_90	1
ROTATION_180	2
ROTATION_270	3

Applications can use the rotation value to construct a transformation matrix that will convert Android *Canonical* accelerometer data to other coordinate spaces. In order to transform from *Canonical* aligned accelerometer values into either *screen* or *world* aligned accelerometer values, a `canonAccel` vector needs to be rotated by 90 degree increments based on the `rotationIndex`, (where `ROTATION_0` means no rotation is necessary).

For the `canonToScreen` transform, the rotations follow these equations:

$$screenAccel[0] = canonAccel[0] * \cos\emptyset - canonAccel[1] * \sin\emptyset$$
$$screenAccel[1] = -canonAccel[0] * \sin\emptyset - canonAccel[1] * \cos\emptyset$$
$$screenAccel[2] = canonAccel[2]$$

Where:

$$\emptyset = \frac{\pi}{2} \cdot getRotation()$$

A function implementing the `canonToScreen` transform follows.

```
static void canonicalToScreen(int     displayRotation,
                              float[] canVec,
                              float[] screenVec)
{
    struct AxisSwap
    {
          signed char negateX;
          signed char negateY;
          signed char xSrc;
          signed char ySrc;
    };
    static const AxisSwap axisSwap[] = {
          { 1, -1, 0, 1 },   // ROTATION_0
          {-1, -1, 1, 0 },   // ROTATION_90
          {-1,  1, 0, 1 },   // ROTATION_180
          { 1,  1, 1, 0 } }; // ROTATION_270

    const AxisSwap& as = axisSwap[displayRotation];
    screenVec[0] = (float)as.negateX * canVec[ as.xSrc ];
    screenVec[1] = (float)as.negateY * canVec[ as.ySrc ];
    screenVec[2] = canVec[2];
}
```

For the `canonToWorld` transform, the rotations follow these equations instead:

$$screenAccel[0] = canonAccel[0] * \cos\emptyset - canonAccel[1] * \sin\emptyset$$
$$screenAccel[1] = canonAccel[0] * \sin\emptyset + canonAccel[1] * \cos\emptyset$$
$$screenAccel[2] = canonAccel[2]$$

This axis-aligned transformation can be put into a static array, as shown below in the `canonicalToWorld()` function that uses a simple integer lookup array to avoid costly trigonometric functions when converting a canonical space accelerometer vector into an OpenGL-style world space vector.

```
static void canonicalToWorld( int           displayRotation,
                              const float*  canVec,
                              float*        worldVec)
{
    struct AxisSwap
    {
          signed char negateX;
          signed char negateY;
          signed char xSrc;
          signed char ySrc;
    };
    static const AxisSwap axisSwap[] = {
          { 1,  1, 0, 1 },   // ROTATION_0
          {-1,  1, 1, 0 },   // ROTATION_90
          {-1, -1, 0, 1 },   // ROTATION_180
          { 1, -1, 1, 0 } }; // ROTATION_270

    const AxisSwap& as = axisSwap[displayRotation];
    worldVec[0] = (float)as.negateX * canVec[ as.xSrc ];
    worldVec[1] = (float)as.negateY * canVec[ as.ySrc ];
    worldVec[2] = canVec[2];
}
```

The next function will compute the axis-angle transform necessary to align a model's `localUp` vector with that of the accelerometer. The function returns the vector (`rotAxis`) and angle (`ang`) which is sufficient to build a transformation matrix or to build a quaternion to orient a model vertically in *World* space.

```
void computeAxisAngle(const float* localUp, const float* worldVec,
                      float* rotAxis, float* ang)
{
    const Vec3& lup  =  *(Vec3*)localUp;
    Vec3 nTarget     =  normalize(*(Vec3*)worldVec);
    *rotAxis         =  cross(lup, nTarget);
    *rotAxis         =  normalize(*rotAxis);
    *ang             = -acosf(dot(lup, nTarget));
}
```

The NVIDIA Android NDK Samples includes library functions for building matrices and quaternions from the axis angle representation. It may be necessary to apply an additional rotation to orient objects in the plane orthogonal to the final world vector.

Power Conservation

In order to conserve device power, applications should choose the slowest accelerometer update rate possible to achieve the desired result. Values available for setting the update rate are defined in `android.hardware.SensorManager` and are listed below in order of decreasing update rate.

constant name	relative speed
SENSOR_DELAY_FASTEST	*fastest*
SENSOR_DELAY_GAME	*faster*
SENSOR_DELAY_NORMAL	*slower*
SENSOR_DELAY_UI	*slowest*

A sample of setting the sensor update rate is shown below.

```
if (mSensorManager == null)
    mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
if (mSensorManager != null)
    mSensorManager.registerListener(
        this,
        mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
        SENSOR_DELAY_GAME );
```

Note that the 'delay' values are abstract, with values specific to a given device, and thus rates could vary significantly between different devices. The only way to guarantee a certain update rate is to measure the rate returned by a device at run time.

Supporting Older OS Versions

In order to support OS versions older than Android Froyo/v2.2, it may be necessary to rely on the older deprecated function, `getOrientation()`. Below is a brief snippet of code illustrating dynamic function binding. Please note that `getOrientation()` may disappear from future Android versions, so it may be most prudent to dynamically bind against both functions and use the one that is available.

```
WindowManager wm;
Method getRotation;

wm = (WindowManager)this.getSystemService(WINDOW_SERVICE);
Class<Display> c = (Class<Display>)wm.getDefaultDisplay().getClass();
Method[] methods = c.getDeclaredMethods();
String rotFnName = new String("getRotation");
for( Method method : methods )
{
    Log.d("Methods", method.getName());
    if( method.getName().equals( rotFnName ))
    {
        getRotation = method;
        break;
    }
}

int orientation;
Display display = wm.getDefaultDisplay();

if( getRotation != null )
{
    try
    {
        Integer i = (Integer)getRotation.invoke(d);
        orientation = i.intValue();
    }
    catch(Exception e) {}
}
else
{
    orientation = display.getOrientation();
}
```

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, Tegra, GeForce, NVIDIA Quadro, and NVIDIA CUDA are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2010 NVIDIA Corporation. All rights reserved.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com