Applying Low Pass Filter to Android Sensor's Readings

May 28th 2013, Samir Bhide

Android Mobile Technical

This is a continuation of the AugmentedRealityView project which was released

The Android sensor framework lets you access many types of sensors. Two very basic types are:

1. Hardware Sensors.
2. Software Sensors.

Hardware sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.
For example: `Sensor.TYPE_ACCELEROMETER"` , `Sensor.TYPE_MAGNETIC_FIELD`

Software sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors.
For example: `Sensor.TYPE_ORIENTATION` , `Sensor.TYPE_ROTATION_VECTOR`

Best Practices for Accessing and Using Sensors

1. Unregister sensor listeners.
2. Don't test your code on the emulator.
3. Don't block the onSensorChanged() method.
4. Avoid using deprecated methods or sensor types.
5. Verify sensors before you use them.
6. Choose sensor delays carefully.
7. **Filter the values received in `onSensorChanged()` . Allow only those that are needed.**

After we register the Sensors, the sensor readings get notified in `SensorEventListener` 's `onSensorChanged()` method. However, the rate of change in sensor values is so high that if we map these small changes a.k.a 'Noise' the values jump within a large range of values.

We can also specify the `SensorManager` 's delay properties from one of these:

2. `SENSOR_DELAY_GAME`
3. `SENSOR_DELAY_UI`
4. `SENSOR_DELAY_NORMAL`

This, however, is only a peek into the system. Events may be received faster or slower than the specified rate, but usually events are received faster.

Moral of the story is:

Allow only those values which are useful and discard the unnecessary noise.

The solution for this is to apply a Low-Pass Filter on these values.

A Small Glimpse of Low Pass Filter

A low-pass filter passes low-frequency signals/values and attenuates (reduces the amplitude of) signals/values with frequencies higher than the cutoff frequency.

Take an example of simple signal with values ranging from 0 to 1. Due to an external source (environmental factors such as jerks or vibrations), a considerable amount of noise is added to these signals. These high frequency signals (noise) cause the readings to hop between considerable high and low values.

Programmatically Apply Low Pass Filter

A device's sensor readings contribute noise data due to high sensitivity of its hardware to various factors. For gaming purposes, these highly sensitive values are a boon, but for application hat need smooth readings, these hopping values are a mess.

Lets look at AugmentedRealityView on GitHub, where we have to point markers on `Camera` `SurfaceView` .

The high sensitivity causes the markers to change positions randomly due to noise.
A Low-Pass Filter concept comes to rescue, because we can omit those high frequencies in the input signal, applying a suitable threshold to the filter output reading

With this implementation the markers won't hop randomly because we have removed the unwanted high reading values.

Here is the algorithm implementation:

```
<code>for i from 1 to n
y[i] := y[i-1] + α * (x[i] - y[i-1])
</code>
```

Here, `α` is the cut-off/threshold.

Lets implement it in Android:

```
<code>lowPass(float[] input, float[] output)
</code>
```

The above method filters the input values and applies LPF and outputs the filtered signals. `static final float ALPHA = 0.25f; // if ALPHA = 1 OR 0, no filter applies.`

```
<code>protected float[] lowPass( float[] input, float[] output ) {
    if ( output == null ) return input;
    for ( int i=0; i<input.length; i++ ) {
        output[i] = output[i] + ALPHA * (input[i] - output[i]);
    }
    return output;
}
</code>
```

Low-Pass Filter is finally applied to sensor values in `onSensorChanged(SensorEvent event)` as follows:

```
<code>@Override
public void onSensorChanged(SensorEvent evt) {
    if (evt.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        gravSensorVals = lowPass(evt.values.clone(), gravSensorVals);
    } else if (evt.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {
        magSensorVals = lowPass(evt.values.clone(), magSensorVals);
    }
    if (gravSensorVals != null && magSensorVals != null) {
        SensorManager.getRotationMatrix(RTmp, I, gravSensorVals, magSensorVals);
        int rotation = Compatibility.getRotation(this);
        if (rotation == 1) {
            SensorManager.remapCoordinateSystem(RTmp, SensorManager.AXIS_X, SensorManager.AXIS_MINUS_Z, Rot);
```

```
        UIARView.pitch = (float)(((results[1]*180/Math.PI))+90);
        UIARView.roll = (float)(((results[2]*180/Math.PI)));
        radarMarkerView.postInvalidate();
    }
}
</code>
```

Here i have applied low pass filter for `Sensor.TYPE_ACCELEROMETER` and `Sensor.TYPE_MAGNETIC_FIELD` .

All code in this post and more can be found on GitHub.

Like what you read? Join our community to get more technical information, chances to win prizes, and more: built.io/community

Popular Posts

Connected Sports Venues: Futuristic

Gal Oppenheimer

Working With References In Built.io

Dhaval Majithia

How To Leverage Integration And

Steven Russell

Subscribe to our blog

first name

last name

work email

Integration ▼

Company

News and Awards

Events

Careers

Contact

Whitepaper

Developers

Built.io Flow Docs

Support

Built.io Flow Support

Built.io Status

Social

Get in touch with our sales team at sales@built.io. For any other assistance, contact us at support-flow@built.io.

Terms of Service Privacy Policy Cookie Policy