Making an Accelerometer into a 3d Mouse with Reactive Extensions

by Paul Schrum

Confidential Intellectual Property of Terragrammetry Inc. Do not share or distribute. Github collaborators are okay to read this as they like, but for now keep it not-shared.

24 July 2014

# Introductory Summary

In pursuit of other objectives, I have bought a PhidgetSpatial 3/3/3 1056 for $60.00. As part of getting a .Net .dll that will process the data stream coming from it, and transforming the data stream such that the .dll can present the data in other, more useful forms to a downstream application.

My end goal is to be able to present to the consumer (another dll or an app), a Reactive Extensions (Rx) data stream of either acceleration, velocity, or position, along with either rotation speed or orientation. All of the values are provided to the consumer who simply takes whichever value they are interested in. By using reactive extensions the data is streamed in real time, and is composable with other Rx temporal data sources (such as video or audio).

# Objectives

1. Track the accelerometer’s position and orientation in real time.

2. Adapt the data stream coming from the accelerometer to an IObservable<T>. This adapting of the data includes noise reduction and numeric integration into Velocity, Position, and Orientation.

3. Use the position and orientation data as a 3d mouse (appearing as an arrow or something in a WPF Viewport3d window).

4. Use the position and orientation data as the position and look-direction for the Viewport3d. In other words, as the accelerometer moves and turns, the view itself moves and turns in the virtual 3d world.

5. Persist the datastream to a file so that it can be analyzed or used in unit testing.

# Anticipated Challenges

The data is very noisy. This accelerometer is low-end qualitywise. This means that you can hold the accelerometer perfectly still and the signal coming from it looks like somebody is jiggling it very fast. Also the orientation data has drift in it, which means that if you went by the data stream, it looks like the accelerometer is slowly turning in space at a rate on the order of 2 rpm. The challenge is to cleanse these unwanted and incorrect artifacts while still preserving data representing true movement.

The data stream comes in via arrays being passed into C# Events. I will be converting this data to custom types consisting of points and vectors and shifting from the Event paradigm to IObservable<T> (the basic type for Reactive Extensions).

The mathematics will be rather involved. The noise reduction must happen in near-real-time, so there is no time to do a bunch of averaging and other noise-reduction techniques for all these data points, plus figure out which way is up, plus remove gravity from the data (and remove it in the correct direction).

# Envisioned Steps

Here are the steps I expect to take over the course of this.

1. Get the accelerometer from the manufacturer.

2. Download the manufacturer’s code sample for C# and get it to work.

3. Get my own, from-scratch C# dll to work properly by taking the data from the device and putting the raw (unprocessed) values on a WPF window.

4. Develop a class to encapuslate the the data coming from the accelerometer plus the derived values (Velocity, Position, Orientation). The class is called SpatialData.

5. Save a sample of the data to a file. Each row in the file is a single SpatialData instance.

6. Adapt the raw Event-based data stream to IObservable<SpatialData>.

7. Analyze the data stream for noise and drift.

8. Develop approaches to mitigate or reduce the noise and drift.

9. Consume the data in a WPF Viewport3d view to show location and orientation. I.e., make a 3D mouse.

10. Consume the data in a WPF Viewport3d as the view control (position, look-direction, up-direction). I.e., make a 3d view orientation control handheld device.

8 August 2014

## Progress Notes

I have accomplished steps 1-5 above. After saving a 10-second snapshot of the accelerometer while motionless, I have a general idea of the nature of the data and what I plan to do to flatten it (reduce the noise). My current notion is to have certain parameters which will keep track of the running average of the accelerometer and the gyroscope, squelching low amplitude signals (forcing to zero), and passing larger signals. The class will yield raw values and smoothed values (processed by running average). The running average process is optimized to take a small amount of time.

I have found that most of the timespans between data event calls is about 8 milliseconds. A few of them are at 12 milliseconds. The 8 millisecond rate is 125 Hertz. Over 10 seconds it averages to about 123 Hertz. With standard video being at 29.97 frames per second this means that there are about 4.1 accelerometer frames per video frame.

Further, I plan to have a running average size of 12 frames. (This is a FIFO queue.) However, once I get farther into development, I can adjust this number for optimal performance. Bacially, the larger the number of frames, the quiter the noise is. But the smaller the number of frames in the running average, the more responsive the accelerometer is. Also, if the frame count is too high, then whole movements can be smoothed over and never register. Thus I need to tune it, so I need to design the class so it can be tuned.

So here is the class design as I am thinking of it now.

public enum AccelerometerState

{

Active,

Suspended,

Calibrating,

Disconnected

}

public class AccelerometerFrame

{

public RawData RawData;

public SmoothedData Frame;

// public AccelerometerState State; not sure this should be here

}

public class RawData

{

public Vector3D Acceleration;

public Vector3D Rotation;

}

public class SmoothedData

{

public Vector3D Jerk;

public Vector3D Acceleration;

public Vector3D Velocity;

public Vector3D Position;

public Vector3D Rotation;

public Vector3D Orientation;

public Vector3D LookDirection;

public Vector3D UpDirection;

}

public class RxSpatial : IObservable<AccelerometerFrame>

{

public CalibrationSettings CalibrationSetting;

public IDisposable

Subscribe(IObserver< AccelerometerFrame > observer){…}

}

public class CalibrationSettings

{

public Double AverageNullAcceleration;

public Double AccelerationZeroSquelchThreshold;

public Double AverageNullRotation;

public Double RotationZeroSquelchThreshold;

}

Note that non-linear gain values are possible and would be interesting to work with later.

Now that my direction is set a little bit better, it is time to move forward with the project. (All puns intended.)

17 August 2014

# First Analysis of Noise

I skipped step 6 and partially implemented step 7, analyze the data stream for noise and drift. I took the 10-second record of frames while the accelerometer was motionless. I developed a class that would decompose Double values to bar charts so I could look at the statistical distribution. I was looking at the magnitude of the acceleration value, which should be very close to 1.0 since the accelerometer has no way to remove acceleration due to gravity.

I thought a bar chart would be informative, hoping to find something close to a Gaussian Distribution. I wrote a class to parse the raw values and compose the barchart. I parsed it into 24 bars, and got the following:

This surprised me because it is not the well-formed Gaussian Distribution I expected. This chart does not show it, but the range is 1.000856074 -- 1.001399829 (0.000543755). These units are g (approximately 9.81 m/s2). The average is 1.000856074. The “bandwidth” (my term), range / average = 0.00054329, or 0.054%.

Although this is pretty tight (seems to me), it is noise, so I would like to eliminate it.

My approach to eliminating the noise is to generate a running average of the values. We are still only working with the magnitude of the acceleration and not the individual axes. The frame rate is 8 ms (occasionally a 12 ms comes in). So I decided to take a 12-frame running average to smooth out the noise. This gives a fair sample size without too much latency. (Approximately 0.1 seconds of latency.)

I made this computation and called it the smoothed values. I plotted the distribution on the same barchart as the raw values, and got this:

I was surprised again. I expected my running average approach to suck up all the outliers, and clearly it does not. The standard deviation is now reduced by 7.2%, which is not much.

So is my running average approach failing? How many frames do I need to get rid of all this noise? Well, before going through all that again for a 50-frame running average (a latency of about 0.4 seconds), I decided I should get a look at the noise over time and also see if I can tell what my 12-frame running average is accomplishing.

Here is how that looks. This is a 1.6 second sample of my results:

Now I see what’s going on. The running average *is* smoothing the data, but there are frequency components in the noise which are greater than my running average length. Let’s refocus on the point here: The above graph shows the acceleration value of a perfectly motionless device. It “thinks” it is moving up and down in space a little bit. This is the noise, and it has high frequency components (which I eliminate with the running average), *and* a low frequency component at about 2.5 Hertz.

I think I can now come to a conclusion: In order to keep a motionless accelerometer from moving in virtual space, **I have to squelch values close to 1.0g**. In this context, squelching means ignoring. In software, ignoring means something setting to 0, as in:

if (Math.Abs(value) <= ToleranceThreshold)

value = 0.0;

Of course, in my case “motionless” is not 0.0, but it is close to 1g. Just how close to 1g I will discuss momentarily.

Interestingly, this code snippet is the same software idiom for Fuzzy Equals for Floating Point values. (Also called Tolerant Equals.) This makes me feel a little better because I have seen it before and know how to deal with it. I want to minimize ToleranceThreshold so I don’t squelch real movements. I want to maximize ToleranceThreshold so noise doesn’t get through to the virtual space and become unwanted “random walk” motion.

Before moving on, there was one last question I wanted to get some kind of answer to: what is the scale of this noise? In other words, how “loud” is it. I look at this by seeing how far the accelerometer “random walks” in virtual space when it is motionless in real space.

Still using magnitude to keep it simple, I numerically integrate the accelerometer values once to get virtual velocity, then again to get virtual position. The analysis essentially tells me how far the device drifts in v-space over the 10 seconds of the sample data I have persisted.

In short, I learned two things from the random walk analysis, both of them pleasing. First, I found that I could “tune” the value of “assumed g” until I got a minimum random walk distance. Perhaps I can use this approach in auto-tuning the device (a future challenge). The second pleasant outcome of the random walk analyis is that over 10 seconds, the real-motionless accelerometer virtually moved only 0.03 inches,or 1 inch every 5 minutes or so. This won’t happen in actual implementations because I will be squelching noise. (I also want to squelch hand-quiver, which will be greater.) The finding is important in that it gives me a sense of scale of how bad the noise really is, and it looks like it is not all that bad.

26 October 2014

# Long Silence, Significant Progress

Blog silence here since 17 August. I have been busy with contract work and other stuff, but I have also made some progress.

1. I have implemented the first, most basic use of Reactive Extensions, both for the producer and the consumer.

2. I have decided on refactoring the data structures and the naming and architecture of the producer classes. Implementing this is on tap for tonight.

3. My presentation on this project has been accepted for Raleigh Code Camp. I now have 13 days in which to form up all of this into a presentation. I did a friendly preview presentation at TriNug’s Data Sig. I got some good advice on the presentation and a good response from the group on the content. This encouraged me a lot.

So with all of this, I am now less focused on data smoothing and tuning. These will still be part of the final product, but they are on the back burner for now. The main refactorings are two:

Refactoring 1: Make a class hierarchy as follows:

Base Class is SpatialDataStreamer\_raw and implements IObservable<AccelerometerFrame\_raw>.

Inheriting from this is SpatialDataStreamer\_processed and implements IObservable<AccelerometerFrame\_processed>.

Inheriting from this is SpatialDataStreamer\_smoothed and implements IObservable<AccelerometerFrame\_smoothed>.

Here are the constructors and properties of the frame classes:

public class AccelerometerFrame\_raw

{

internal AccelerometerFrame\_raw(

Double accX,

Double accY,

Double accZ,

Double rotX,

Double rotY,

Double rotZ

)

{

Acceleration = new Vector3D(accX, accY, accZ);

Rotation = new Vector3D(rotX, rotY, rotZ);

}

public Vector3D Acceleration { get; internal set; }

public Vector3D Rotation { get; internal set; }

}

I have this (above) class refactored and implemented.

Next is:

public class AccelerometerFrame\_processed

{

internal AccelerometerFrame\_processed(

AccelerometerFrame\_processed previousFrame,

AccelerometerFrame\_raw currentRawFrame

) : base(currentRawFrame)

{

}

/// <summary>

/// Force of gravity.

/// </summary>

public Vector3D g { get; internal set; }

/// <summary>

/// Acceleration sensed by the accelerometer with g taken out.

/// </summary>

public Vector3D TrueAcceleration { get; internal set; }

public Vector3D Velocity { get; internal set; }

public Vector3D Position { get; internal set; }

public void SetGravityVectorWhileStill(Vector3D newG)

{

g = newG;

}

}

31 October 2014

# Happy Reformation Day

I have the Processed Data Stream classes hooked up and working – kind of.

Issue: My algorithm does not consider the time delta, so I think velocity is too large by a factor of approximately 120. So I have to get the time component added in.

Issue: The window I built to consume the Processed Data stream is a child of the Raw Stream window, and the stream does not show up in it. My theory is that the accelerometer’s SpatialData event handler can only have a single subscriber. What this means is that I have to refactor my code. Either I have to make the Processed Stream View be in its own project, or I have to make the Raw Window unsubscribe from the Raw Stream before it creates the Processed Stream Window. I will attempt approach number 2 first because I want to show it that way at the presentation. Also, I would really like to see if approach 2 can work.

New thought on Issue 2: What if the problem is not that the accelerometer driver can only handle 1 subscription? What is the problem is that my implementation blocks other subscribers, and the best approach is to implement the processed stream window as subscribing to the same, original stream. This solution/approach would entail passing the IObservable to the constructor of the processsed stream window, thus not creating it anew, but subscribing to it nonetheless.

9 November 2014

# Major Progress, Presentation Went Well

I gave my presentation yesterday at Raleigh Code Camp. There were only 9 adults and 1 child in the presentation, which disappointed me. At least they all stayed through the whole presentation, so it was not all bad. Actually, I was quite pleased with how well it went.

A lot has happened since my last entry. I got some simple Reactive Extensions queries working and working correctly. The Processed Data Stream class is no longer “kind of” working. It is all working. I am sending the processed data stream to its own window, and the Process Stream Window and the Raw Stream Window are both updating correctly and concurrently. Yes! Fist pump.

Important notes. SpatialDataStreamer\_raw implemented Rx improperly. For educational purposes, I did not delete it. The class with the correct implementation is SpatialDataStreamer\_raw2. It is a singleton class, so I should inherit \_processe from it. Instead, process attempts to create the singleton, which automatically creates it if needed, but returns itself if it already exists.

\_raw2 no longer is-a IObservable<T>. It has-a. T is still AccelerometerFrame\_raw. So if \_raw2 is not an Observable, how do I make Rx work? \_raw2 creates the Observable FromEventPattern. Then the Window’s viewModel subscribes to it.

SpatialDataStreamer\_processed also subscribes to the raw2 stream and transforms that stream to make its own processed stream. So when my demo app is running, Streamer\_raw2 has two subscribers: The viewModel of the Raw window, and the instance of Streamer\_processed. They do not interfere with each other, thus they are running concurrently. This means I wrote a concurrent application without even realizing it. This last sentence, well the implication of that is that Rx truly is powerful.

I learned that in order to make Streamer\_processed consume the data from Streamer\_raw2, I had to make some minor changes to the Rx query in raw2. I had to add .Publish(); to the query, then .Connect(); I do not now know exactly why this made it work, but it did.

I added code to make \_processed self-calibrate in the first 3/10’s of a second of its lifetime. This assumes that the accelerometer is perfectly still, and that the acceleration that it feels is all from gravity and only from gravity. From this, I get the g-vector, which I later subtract from acceleration to get trueAcceleration. At this point, g-vector does not change even when I rotate the device. So trueAcceleration is totally wrong if I do rotate it. I pointed this out in my presentation yesterday to illustrate why this is an important thing for production code to do.

Where to next:

1. Add a report of the time it takes each processed frame to get created as a percent of how much time is available between frames. So far, I have no idea if I am taking up too much time or if I am doing fine. This metric will also be used once I make SpatialDataStreamer\_smoothed. I expect I may be pushing the limit with all the processing I do in \_smoothed, but how can I know unless I ask.
2. Add code to rotate the direction of g as the device orientation changes. For \_processed, this will still be in error because the rotationRate is not zero for a still device due to noise. But it is consistent with the life-purpose of the \_processed class: just the facts and nothing but the facts. The \_processed frame class would not be consumed in production applications, but I will keep it public for the purpose of educating people on what happens when you have a noisy accelerometer and how the device does a virtual random walk. These items, 1) and 2), are all for SpatialDataStreamer\_processed. The remainder of items are for SpatialDataStreamer\_smoothed.
3. Add averageStillGMagnitude property. After self-calibration, this value will be unchanged for the life of the class. \_smoothed will distinguish true movement from still noise by comparing the acceleration magnitude to averageStillGMagnitude. If it differs by more than the given tolerance (in the Tuning class), then trueAcceleration is non-zero and we are accelerating.
4. Implement approach to Orientation in which orientation during non-accelerating times is computed from the gVector. This will yield greater accuracy for hand-held mouse operations. That is, it is an optimizing assumption available to hand-held mouse usage that is not possible for a generalized accelerometer. Rotation rate, then, would be used in two cases. One is when the device is accelerating in order to account for rotation happening simultaneoulsy with acceleration. The other is when one of the three axes of the device is near vertical. So when Z, for example, is close to azimuth (up from the earth), then rotation rate from the gyroscope is needed to register yaw (rotation about Z). This tolerance is probably just 5 degrees from plumb. More than that and detecting from the g vector is more accurate.
5. Squelch trueAcceleration value to 0.0 when acceleration.Magnitude is close to g.Magnitude. When being used as a 3D mouse, this state will also squelch Velocity to 0. It would not do so for other applications (like wheel-drones).

Time for bed now. It is good to know what course to set. Once I get to item 5), the project is essentially complete – that plus complete unit test coverage.

If the operation of the 3D mouse is successful, then I would like to try to market it. Perhaps I can get it into Microstation or Autocad or find othr 3D apps that need this kind of thing. Then again I may only succeed in proving that accelerometers make lousy 3D mouses. Only time plus work will tell.

10 November 2014

# Time Metrics In Place, Looking Very Good

This morning I added frameProcessingTimePercent. This is the Percent of a frame that all of the processing uses up (Item 1 from yesterday’s entry). The results are very pleasing. Running in Debug mode, I am averaging about 0.4% usage when generating a Processed Frame. Now I am hopeful that the additional work to generate a Smoothed Frame will be less than 8% total on an otherwise idle machine. As other processes come up, Frame Processing Percent increases. Eventually I will move the whole process to another Task, but getting that done is not a high priority for now.

Noteworthy is that when I sit there and watch it, the average is about 0.4%. When I move the accelerometer around, this value does not change. But when I move the system mouse around, the Frame Processing Time Percent goes up to about 0.6%. Clearly, as the CPU increases, my Processing Percent will increase, so I have to keep an eye on this.

Also today: I learned that my Window classes are not being noticed by Git. It turns out that that Project had the same name as the Solution, and Git could not deal with that. I had to refactor and rename and whatnot. That is all taken care of now.

11 November 2014

# Happy Veterans Day

Today my goal is to add code which updates the g vector based on Rotation Rate in Processed Frame. It looks like I can save myself a lot of math heartache by using the framework’s Transform3DGroup and applying transforms to g.

So, after a little work, I am in Quaternions now. Watching this video: *Math for Game Developers - Quaternions and Vectors* https://www.youtube.com/watch?v=Ne3RNhEVSIE

Okay. This video series, Math for Game Developers, is great overall. By watching part of it, I have learned that Quaternions are not where I need to go. But this series has much gold in it, so now I am watching the videos on Matrices.

18:05

I haven’t finished watching the Transform Matrix video yet, but I am pretty sure that Rotation Matrix is what I need to be working with. Also, the “Orientation” values I am keeping track of are Euler Angles, so rotation rate is rate of change per axis of the Euler Angle.

What I really want to do is determine the device’s Rotation Matrix from the g vector at the beginning of the session, then update the Euler Angles from rotation rates then compute a new g vector based on the updated Euler Angles, so that I can then subtract g from acceleration “correctly”. I put “correctly” in quotes because I am still talking about Processed Frames, but not Smoothed Frames. When I implement Smoothed Frame, upon a still device event, I will reset g vector to be read from average g over x seconds, just like when I am first calibrating. ~~In other words, I’m really getting pitch and roll from g whenever possible, so only \*yaw is updated from the gyroscope, and that is only necessary when the device is not close to plumb on one of the three axes. (\*Yaw is from the gyroscope when effectively plumb along Z. Device Pitch is from the gryoscope when effectively plumb along X, and Roll is from the gyroscope when effectively plumb along Y.)~~

12 November 2014

# What a Difference Sleep Makes

11:00 AM

This morning I realized that the last three sentences of yesterday’s entry are conceptually in error, so I marked through them. Orientation can not be determined from the g vector alone, no matter what the orientation of the device may be. That is, the component of Rotation Rate which is about the g-vector as an axis can’t be determined. So the marked-out sentences are not untrue, they are just not complete. There is a loss of visibility on the rotation value about g no matter what the total device orientation is, even when one of the primary axes is not close to plumb.

This means the maths are a little more complex, and definitely beyond me. I will have to enlist some help from some smart friends. Fortunately, I have smart friends.

But this has made me think of something else. At the beginning of this development process, way back in July, I was not getting the magnetometer frames from the device. I just shrugged and proceeded without concerning myself with this minor detail. Now I realize I want these values. I need these values. I have to have these values.

I need this data channel in order to help me with the orientation question. If I can get the magnetic field vector, I can use that as another indicator of orientation. My mathematics for determining orientation about the g axis are then considerably simpler. So this has become my next task – seeing if I can get that magnetomter data and use it.

13:50

Time to contact Phidgets Support. When I consume the Magnetometer Reading, some interesting (and debilitating) things happen. When I get the value in a classic .Net Event Handler, the Magnetometer Reading is the identical same as the Gyroscope Reading. I have no way to know if this particular device is defective or if Phidgets is advertising falsely. Let’s give them the benefit of the doubt. But the other problem is that when I try to read the value via Reactive Extensions’ select statement, it throws an error that crashes the app before I can even catch it. This probaly means I have to get the first chance exception. I have a better idea: Go to a different device.

So a collaborator-friend of mine told me that the 1056 is no longer available and perhaps he could get the