**Software and Electronic Engineering**

**VR headset + GAME Documentation**

Project log

***How does VR work?***

VR glasses contain 2 lenses that help to create a 3D virtual image by angling 2 slightly different 2d images. light passes through the cornea, iris and lens up to finally at the retina. from the retina all the information travels to the brain and is processed

*Tuesday, 8 October 2019:*

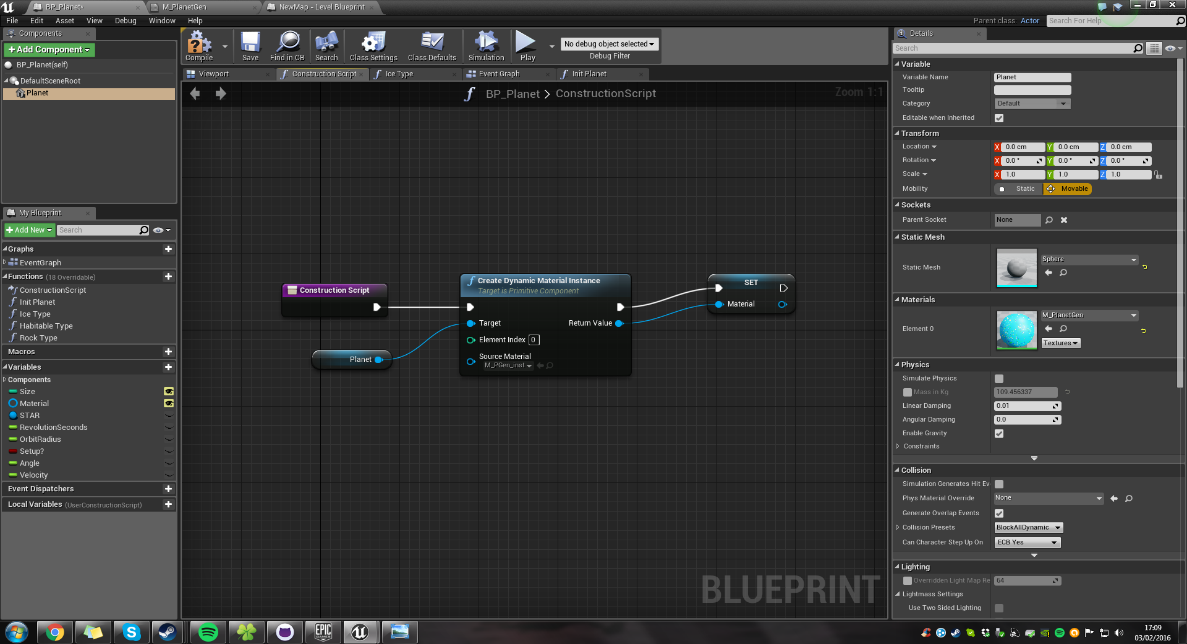
Submitted project proposal

*Friday, 11 October 2019:*

Ordered STM32 and Mpu6050 gyroscope and got 3d printed parts ordered and controller

*Saturday 12 October 2019 Unreal engine:*

Installed unreal engine and started using blueprint and C++ to start my game.



*Monday 15 October 2019:*

STM32 and gyroscope arrived

Started Implementing connection between gyroscope to get reading from gyroscope for later use.

*Wednesday 17 October 2019: STM32 hardware implementation*

Started stm32 implementation with gyroscope

*Friday 19 October 2019: Quaternions*

**Learning about Quaternions in video games**

 a quaternion is a complex number with 4 dimensions. But in game development, Quaternions are often used to describe a rotation in 3d space by encoding:

1. a rotation axis (in the form of a 3-dimensional vector)
2. how far to turn around that axis

An alternative way to describe rotations is by describing how far to turn around the 3 fixed axis' x, y, and z (aka Euler angles) which only requires 3 numbers instead of 4 and is usually more intuitive to use. However, Euler-angles are subject to a problem called [gimbal-lock](https://en.wikipedia.org/wiki/Gimbal_lock#Gimbal_lock_in_applied_mathematics): When you rotate 90° around one axis, the other two axis become equivalent. With quaternions, this problem does not occur

*20 October 2019: Quaternions in coding*

*1ST NOVEMBER 2019 OSVR [Open-Source Virtual Reality](http://www.osvr.org/)*

**What is OSVR**

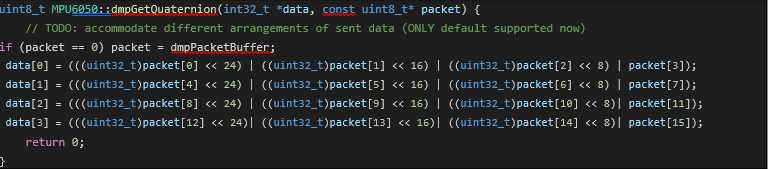
Open Source Virtual Reality is an open-source software project that aims to enable headsets and game controllers from all vendors to be used with any games developed by Razer and Senses. It is also a virtual reality headset that claims to be open-source hardware using the OSVR software

 OSVR is designed to work with several other head-mounted displays and is on a mission to establish an open standard so that existing devices and software can become interoperable.

Developers and consumers with OSVR-compatible HMDs, [can already get started](http://www.osvr.org/getting-started.html) by configuring their equipment. The OSVR website features a perfect [showcase of their software](http://www.osvr.org/featured.html) (also called “experiences”) that are compatible and ready to run. There are already games, media players, emulators and more! There is lots of SteamVR-powered content too.

the data hold packets ! the packets are what we got for the sensor , and the data are the quaternions that we've used the packets to get them . we use packets to form the data . and the data are actually the quaternions ! the data is holding 4 variables . (w,x,y,z) data[0] = w data[1] = x data[2] = y data[3] = z

(1) This what the: mpu.dmpGetQuaternion(&q, fifoBuffer); is referring to : in “ MPU6050\_6Axis\_MotionApps20.h “



This function takes two parameters: a 32bit pointer to data, and 8bit pointer to packet.

It will first check if the packet equals to 0 , if so then it will be equal to dmpPacketBuffer which is another pointer holding default (q) data !

After that, the function updates the data array with the packets by shifting them to a 32bit data variable .

For the data[0] which is a 32bit, it takes the packet[0] and shift it to left by 24bit , takes the packet[1] and shift it to left by 16 bit , takes the packet[2] shift it to left by 8 bit , and takes the packet[3] then add all together with the ‘|’ operator! Keep in mind that each packet is an 8bit, so each packet [] will take 8bits!

This is for example the data [0] : (32bit)

Initial state: 00000000 00000000 00000000 000000

This is for example the packet[0](8bit): XXXXXXXX

This is for example the packet[1](8bit): YYYYYYYY

This is for example the packet[2](8bit): ZZZZZZZZ

\*All these packets are actually what we have read from the sensor in its fifobuffer !

(uint32\_t)packet[0] = 00000000 00000000 00000000 XXXXXXXX // just change the bits from 8 to 32!

(uint32\_t)packet[0] << 24 = XXXXXXXX 00000000 00000000 00000000 // a shift to the left by 24bit

And the same thing will happen to the other packet[] that makes the data[0] :

(uint32\_t)packet[1] << 16 = 00000000 YYYYYYYY 00000000 00000000

// a shift to the left by 16bit

(uint32\_t)packet[2] << 8

= 00000000 00000000 ZZZZZZZZ 00000000

// a shift to the left by 8bit

packet[3]

= VVVVVVVV

// (we don’t put it in a 32bit format)and without shifting

The ‘|’ operator will add them all :

(uint32\_t)packet[0] << 24 = XXXXXXXX 00000000 00000000 00000000 (uint32\_t)packet[1] << 16 = 00000000 YYYYYYYY 00000000 00000000 (uint32\_t)packet[2] << 8 = 00000000 00000000 ZZZZZZZZ 00000000

packet[3] = VVVVVVVV

data [0] = XXXXXXXX YYYYYYYY ZZZZZZZZ VVVVVVVV And so on for all the other data []!

We have 16 packets and 4 data. each data[] will be filled with 4 packets !

\*You see the function takes the &q which is a pointer to the (w,x,y,z) , but in the real function in the library it is replaced with \*data . so at the end, all the data[] will be saved to the q(w,x,y,z) ! and so :

Data[0] would be the x

Data[1] would be the y

Data[2] would be the z

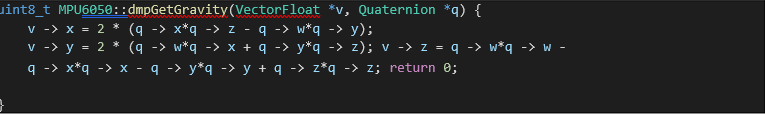
Data[3] would be the w

**In general**:this functionmpu.dmpGetQuaternion();will take the sensor data and transform it to the q(w,x,y,z) datathat we need, to proceed for other parameters calculations !

(2) This is the function that mpu.dmpGetGravity(&down, &q); is referring to :

It takes &down which a pointer to the gravity vector data, and takes &q a pointer to the q(w,x,y,z) data :

It takes the q(w,x,y,z) and calculates the gravity vector data :



We’re using pointers and structures, so the instruction might look ambiguous .

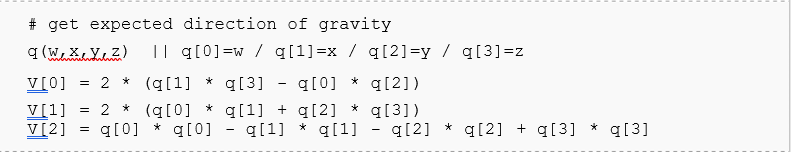
Here is what inside this function mean in another mathematical way :

Vx = 2\*(X\*Z – W\*Y)

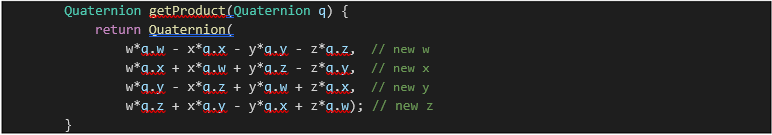
Vy = 2\*(W\*X+Y\*Z)

Vz = W²-X²-Y²+Z²

Same as explained here :



(3) this is what the getProduct() function is referring to :



This function will take any data format as a Quaternion format and outputs it as another Quaternion data format.

For example :



will take the yaw\_correction (yaw\_correction.w, yaw\_correction.x, yaw\_correction.y, yaw\_correction.z) w\* yaw\_correction.w - x\* yaw\_correction.x - y\* yaw\_correction.y - z\* yaw\_correction.z, // new w w\* yaw\_correction.x + x\* yaw\_correction.w + y\* yaw\_correction.z - z\* yaw\_correction.y, // new x w\* yaw\_correction.y - x\* yaw\_correction.z + y\* yaw\_correction.w + z\* yaw\_correction.x, // new y w\* yaw\_correction.z + x\* yaw\_correction.y - y\* yaw\_correction.x + z\* yaw\_correction.w); // new z

at the end it will return a NEW q(w,x,y,z) with corrected yaw.

And the same thing for the tilt correction : qc.getProduct(tilt\_correction); will report a NEW q(w,x,y,z) with corrected tilt.

After that the **qc.getProduct(q);** will apply these corrections to the original q(w,x,y,z) by :

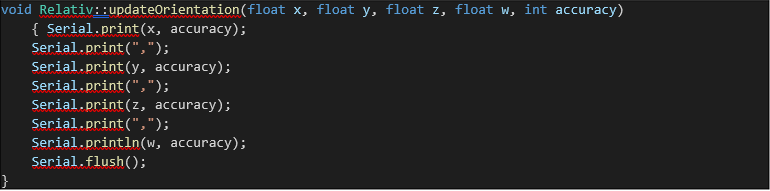
(4) This is the function that relativ.updateOrientation(q.x, q.y, q.z, q.w, 4); is referring to :

We just print all the corrected Quaternions data to the serial (to the pc) for further processing by the VR software tools !

We also set the accuracy, here it is set to 4 . (just how much numbers after the decimal point)!

The pc will get in the serial : the final corrected quaternions separated by a (,).

<the (,) is just for separating these values >.



**In General :**

1. We get the gyroscope+accelerometer data from the MPU6050 sensor through the I²C .

2. We process these data to get the Quaternions we needs q(w,x,y,z).

3. We use the Quaternions to get the gravity vectors .

4. We use the gravity vectors to get the tilt angle.

5. We use the tilt angle to get the drift due to tilt .and we use the default values for the drift due to yaw.

6. We do the tilt and yaw corrections based on the drift due to tilt calculated, and default drift due to yaw .

7. We apply those corrections to the Quaternions q(w,x,y,z).

8. We send the corrected q(w,x,y,z) to the PC via serial communication for further processing by the VR software.

# 

### 

### 

### 

### 

### 

### 

### 

### WHY I AM USING AN STM32?

the speed !

as the clock speed gets higher , the amount of time a microcontroller will need for executing a code instruction will decrease ! say a microcontroller capable of a 400Mhz clock speed . this 400Mhz means that the controller can do 400 million clock cycles on one second . any code instruction will need certain amounts of clock cycles to be totally executed . example: say we have that microcontroller with 400Mhz clock speed .

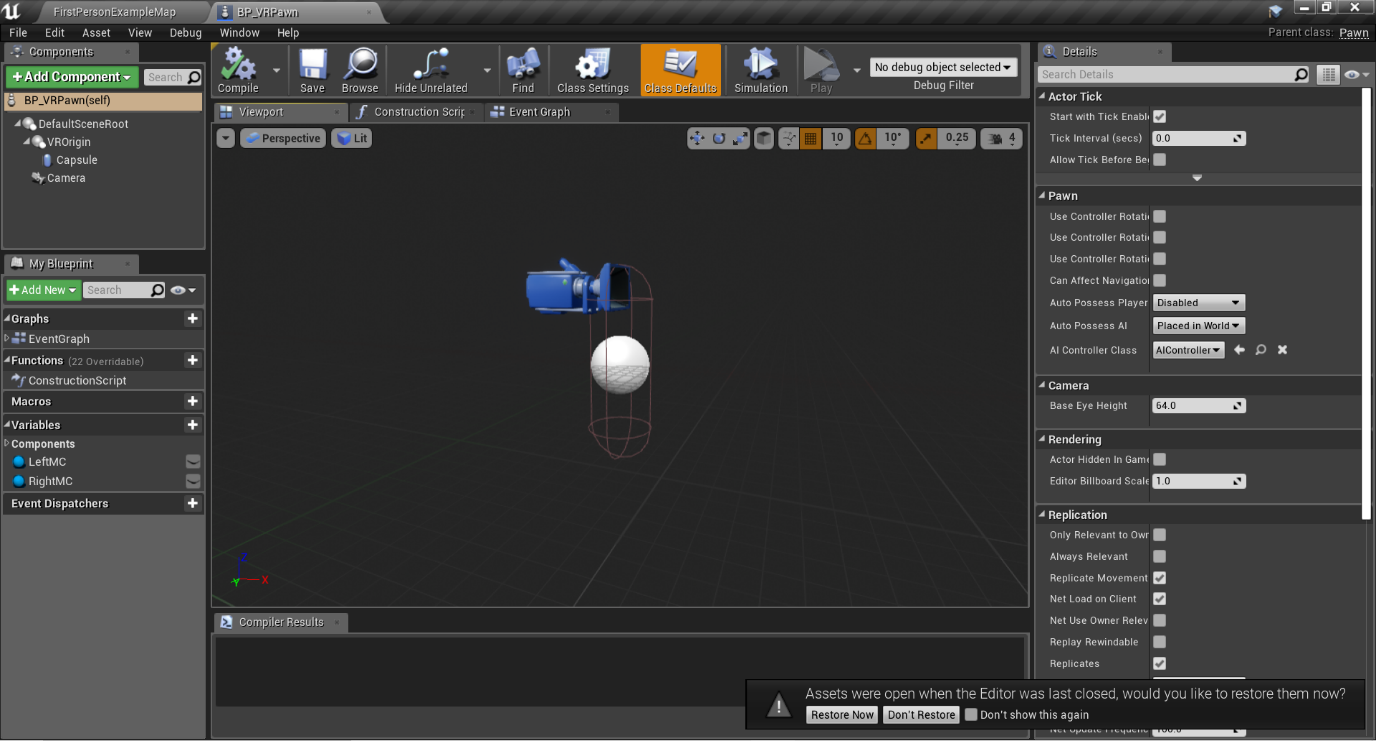
and say we have a code instruction that needs 100 clock cycles to be executed . question: how much time this microcontroller needs to execute this code instruction ? solution: 400Mhz -> 400 million clock cycle per second any clock cycle will then be at : 1/(400 million) = 2.5 nS (nano second) so we have 2.5nS per clock cycle for this microcontroller . so the 100 clock code instructions will need 2.5nS x 100 = [ 250 nS ] to be totally executed by this 400MHz microcontroller . l

et's do all that , for a standard arduino board that usually have a 16Mhz clock speed . any clock cycle will then be at : 1/(16 million) = 62.5 nS (nano second) so we have 62.5nS per clock cycle for this Arduino board . so the 100 clock code instructions will need 62.5nS x 100 = [ 6250 nS ] to be totally executed by this 16MHz Arduino board. Who is the fastest board do you think ? it is the one with higher clock speed ! mathematical equations have lots of code instructions ,

that requires lots of clock cycles . and if the microcontroller is not very fast to execute them within the acceptable amount of time , other followed instructions will got delayed ! and so maybe a Small functioning on the work of the entire code .

in this VR application ,most cases you will have a delay between the movement of the unit and the displayed view on the lcd !

*1ST NOVEMBER 2019 continuation of Unreal engine*

**

This the camera and how tall the player will be and how they will camera will be used and placed

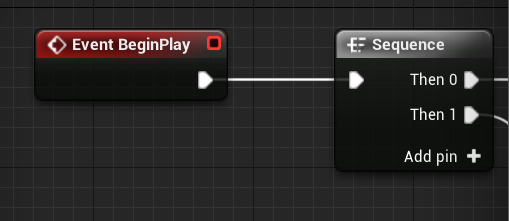
****

So far, I’ve created two hands, a canvas and a pickup test. The goal is to test if I can pick up and drop-down items.

*2nd November 2019*

*Unreal documentation for BP\_VPAWN*

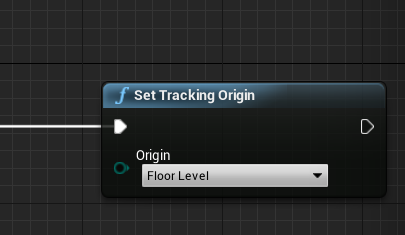
Reacting to the environment



**Event Begin Play**

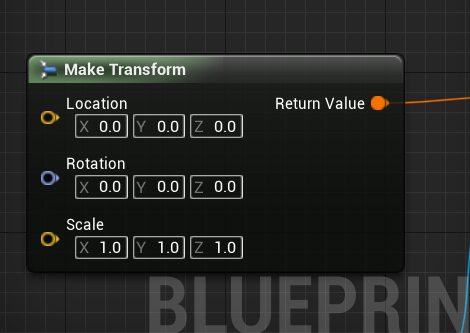
This **event** is triggered for all Actors when the game is started, any Actors spawned after the game is started will have this called immediately. Upon **beginning play**, this Actor will set its Health to 1000 and Score to 0.

The **Level Sequence** is the "container" for your cinematic scenes and must be created in order to begin working inside of the Sequencer Editor.

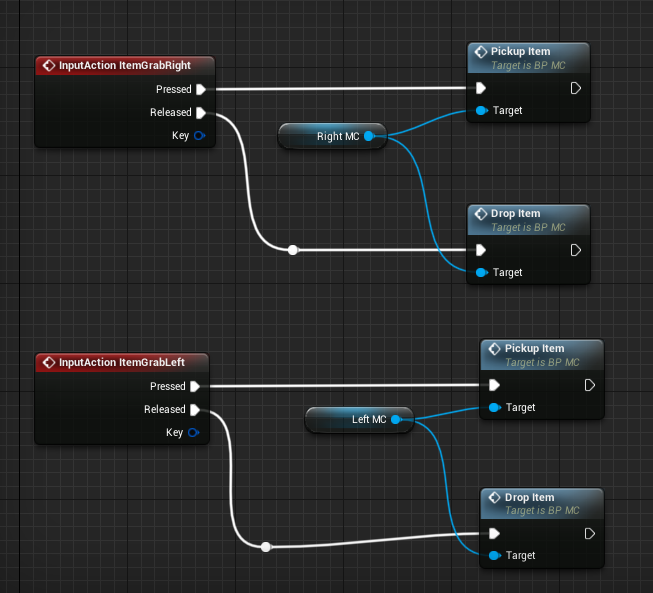


**Sets current tracking** origin type (eye level or floor level).

Target is Head Mounted Display Function Library



**make transform** is just location rotation and scale combined as quaternion



**Action and Axis Mappings** provide a mechanism to conveniently map keys and axes to input behaviours by inserting a layer of indirection between the input behaviour and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.