Design and Implementation of a Low-cost VR Controller

Michael Hamm   
*Computer Science   
Columbus State University*Columbus, Georgia  
hamm\_michael@columbusstate.edu  
Jorge Castro  
*Computer Science  
Columbus State University*Columbus, Georgia  
castro\_jorge@columbusstate.edu

*Abstract*— There are several issues facing cheap virtual reality solutions, such as expense and single-controller only support for many systems. We designed a controller using an Arduino Nano 33 iot, Arduino integrated development environment for a client, Visual Studio integrated development environment for a C# Server, VRidge and Riftcat, along with other physical parts. Our Arduino client connects with our C# Server using TCP first, and then UDP is used to send packets containing controller information to our C# Server. The C# Server sends this information to VRidge, which sends it to Riftcat, which is used in accordance with SteamVR.

Keywords—virtual reality, Arduino, VRidge, Riftcat, SteamVR

# Introduction

In recent years, there have been several VR (virtual reality) systems designed. There is a relatively large degree of difference between many of these systems, namely related to capabilities, ease of use, and, especially, price. For example, Google Daydream is a phone-based VR headset that costs under $100, while the HTC Vive Pro headset costs about $800 alone. In addition, many cheaper phone-based VR headsets such as Google Daydream and Samsung Gear VR only allow the option of one controller, while more expensive systems allow two. The goal of this project was to create an inexpensive, Arduino-based VR controller that would work in conjunction with the native VR controller of a phone-based Arduino headset by using VRidge API to allow the use of two controllers in VR games with a phone-based VR headset.

# Materials and Design

## Materials

The Arduino Nano 33 IoT was the main component of this project. It is an Arduino device that has both an IMU (inertial measurement unit) and a 2.4 GHz WiFi chipset. The IMU was necessary for its use as a VR controller, as accelerometer and gyroscope data from an IMU are needed to be passed to the VRidge application programming interface (API). The other major hardware component used was a phone-based VR system, and physical components used were Micro-USB to USB cables, pushbuttons, wires, resistors, and breadboards, and personal computers.

Software components consisted mainly of the Arduino integrated development environment (IDE), Visual Studio IDE, VRidge, and Riftcat. VRidge is a software that takes positional and rotational data from a phone, along with controller information, and sends it to Riftcat, a software that allows a VR game to be streamed to your phone in the phone-based VR headset [4].

## Design

The controller was designed to have 3 buttons, named according to the data they would send to VRidge when pressed. The figure below shows these 3 buttons, with red being the trigger button (matching to D2 on the Arduino Nano), the blue grip button (matching to D3), and the black reset button (matching to D4). The red cables connect the 3.3V power, the orange cables send the data (D2, D3, D4), and the white and black cables are ground cables.

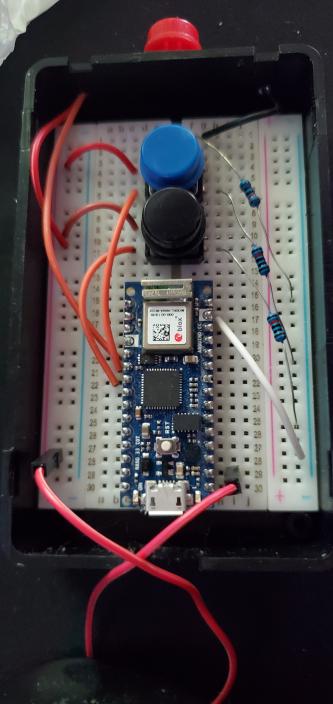


Fig. 1. Picture of the Microcontroller Design

Figure 2 below shows the overall design of the system. First, there is a TCP connection between the Arduino client and the C# server created with Visual Studio. After welcome messages are exchanged, a UDP connection is then made to send packets from the Arduino to the C# server. These packets contain orientation, user input, and controller state information for use with the VRidge API. The VRidge API sends this data to Riftcat, which presents it to OpenVR API (and thus SteamVR) as a controller with the corresponding user input from the microcontroller.

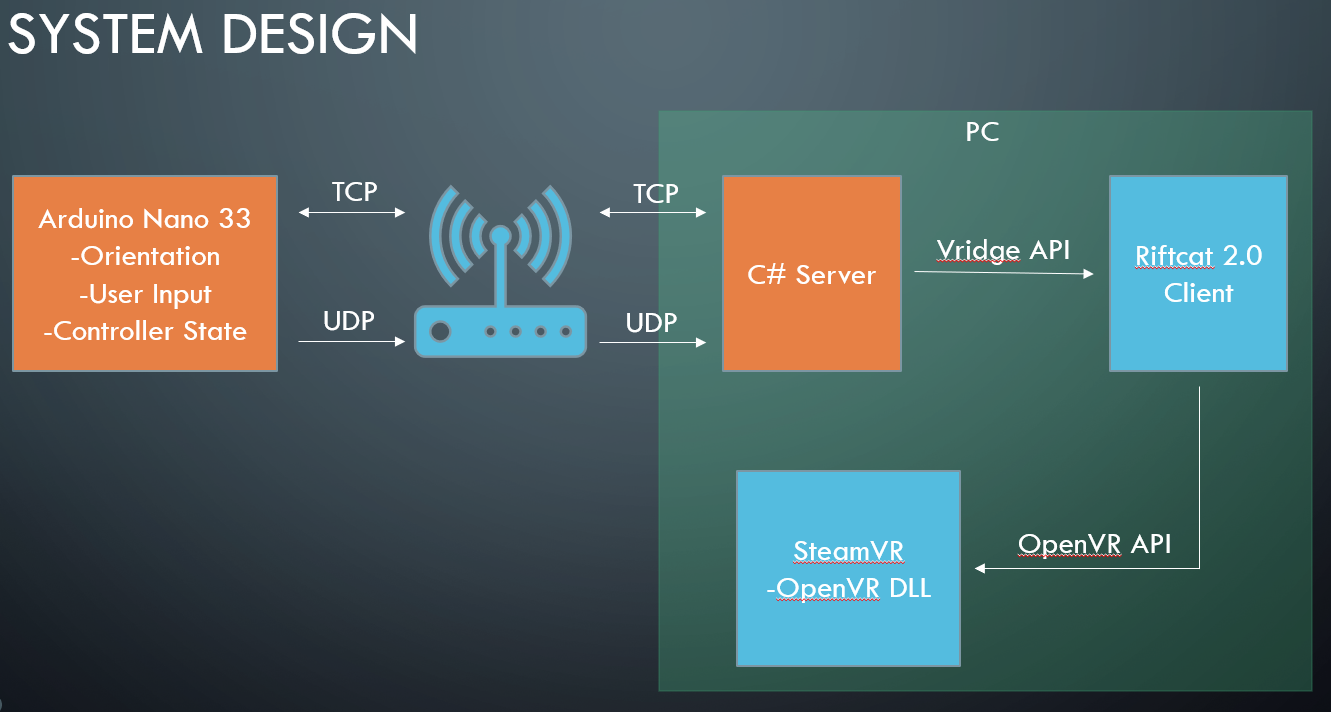


Fig. 2. Overall System Design of the VR controller project

# Code implementation

## C# Server

Our C# server was heavily based on previous work by Tom Weiland’s C# Networking tutorials. Tom Weiland created a series of networking tutorial videos [7] and full code [6] that are useful for a project such as this where game controller (such as the microcontroller) information needs to be taken from a client and sent to a server. His server setup allows information from the client (controller) to be stored in a controller object in the server, which is useful for us as we can pass this controller object to the VRidge class in our C# server, as it contains IMU data and button data. The VRidge class then sends this to Riftcat/VRidge using the SetControllerState method, which passes all the controller data, including controller id, hand, orientation, analog, button, and touchpad data.

## Arduino Client

The Arduino client was responsible for much of the initial setup of the microcontroller and variables. Figure 3 shows that several libraries were imported, such as the IMU library for the Arduino Nano 33 IoT, Arduino\_LSM6DS3 [1], WiFiNINA, the library used for WiFi connection and TCP connection [2], and WiFiUdp, the library used for UDP connections [3]. Especially important to the project was the SensorFusion library, which is a library that can filter IMU positional data for increased positional accuracy and redefine positional data in terms of quaternions. Other inclusions in the Arduino code included arduino\_secrets.h, a file that includes personal information regarding things like network SSID and password, and controller\_packet.h, a file that defines a packet and contains different functions that allow reading of packets from the server and writing of packets to the server.

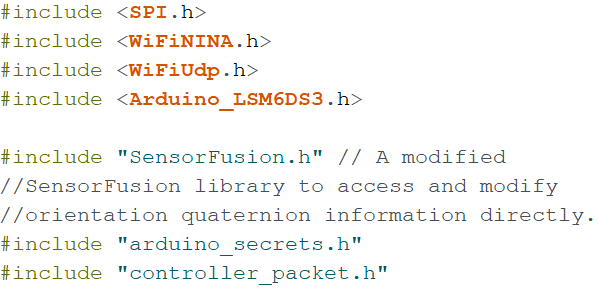


Fig. 3. Library and header file inclusions

The code starts with defining variables after the inclusions, as shown in Figure 4, such as the buttons trigger, grip, reset, and reset\_button\_ms, which is used to make sure the reset button is held for 2 seconds before the controller resets position. G is then defined, which is a variable that is used when calibrating accelerometer values from the IMU. Next, gyroscope values are calibrated, sensor fusion is setup, then sensor fusion and gyroscope-related variables are setup. Last for variable setup, delayMillis, startMillis, and currentMillis are setup, along with the resetButtonCounter, which are all used later for use in timing, so new controller information isn’t sent too early after one set of controller information is sent.

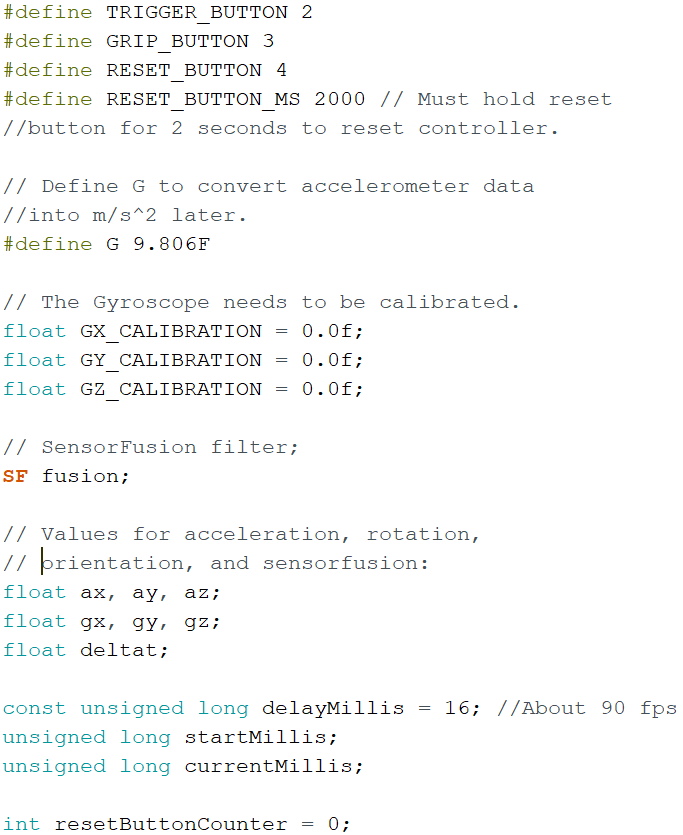


Fig. 4. Setup of global variables

WiFi information is defined next, as variables from the arduino\_secrets.h file are put in Arduino variables ssid[] for network SSID, pass[] for network password, serverIP for IP Address, and serverPort for server port number, as shown in Figure 5. tcpClient and udpClient are then setup using WiFiClient and WiFiUDP, respectively. clientId and clientPort are then setup to take in the client ID and port numbers that the Arduino client will receive from the C# server. Packet is then defined using the Packet class from controller\_packet.h.

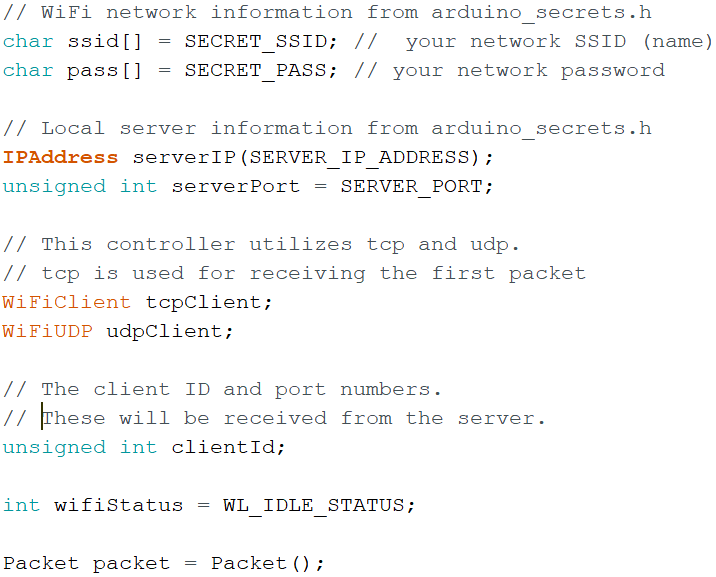


Fig. 5. Setup of WiFi credentials, UDP, and TCP, as well as creation of a packet.

Figure 6 shows the start of the setup portion of the code after serial port connection, where the buttons are initialized as inputs, and then the IMU is initialized. If IMU has begun, the IMU is then calibrated using calibrateIMU().

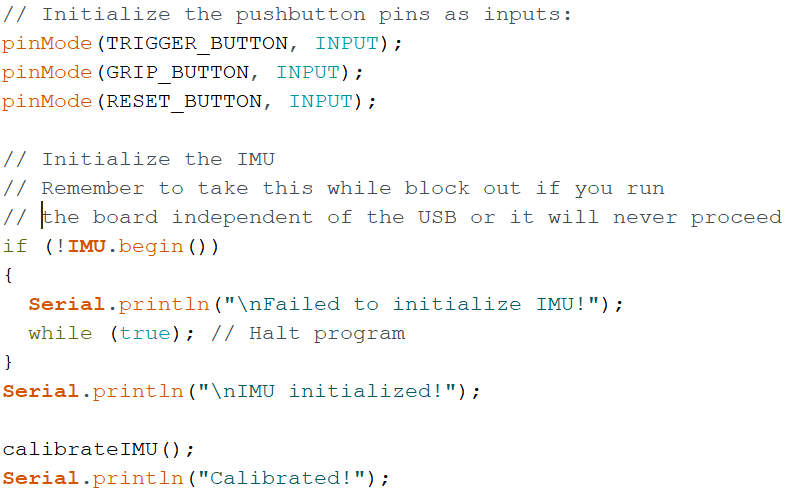


Fig. 6. Initialize buttons and IMU, then calibration of IMU

The calibrateIMU() function calibrates gyroscope data based on the current IMU position. Figure 7 shows how this function works, as it takes multiple gyroscope values over time, and then averages all of them, afterwards subtracting the averaged values of gx, gy, and gz from 0.0f to get GX\_CALIBRATION, GY\_CALIBRATION, and GZ\_CALIBRATION.

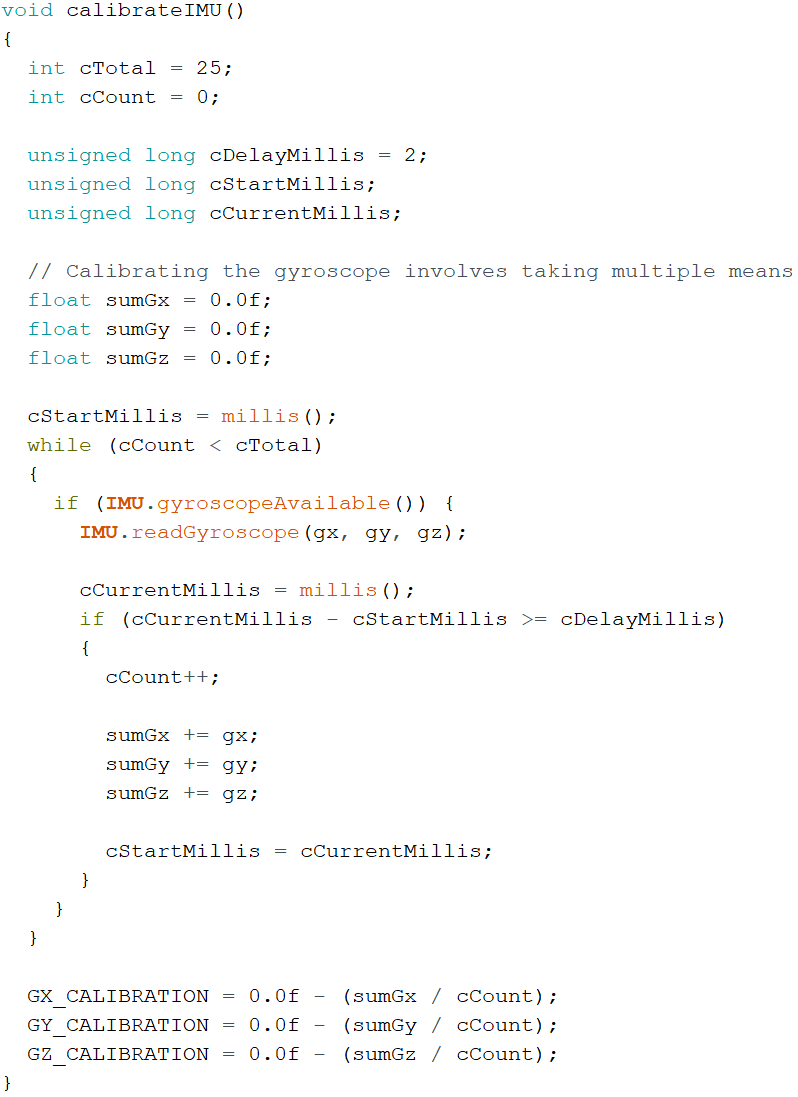


Fig. 7. Calibration of gyroscope values from the IMU

After Calibration, the WiFi variables defined earlier are used to connect to the local WiFi, and details are printed to the Arduino, shown in Figure 8.



Fig. 8. WiFi was connected, and network information is shown.

Next, connection to the local server is done using tcpClient.connect(serverIP, serverPort). If connection is successful, the Arduino client then waits for the response from the server, and when it is, it stores the message using tcpClient.availalbe(). Next, there is a loop that uses tcpClient.read() to store the message in the form of bytes. The msgbyte variable is then written into a packet using packet.writeByte(msgbyte).

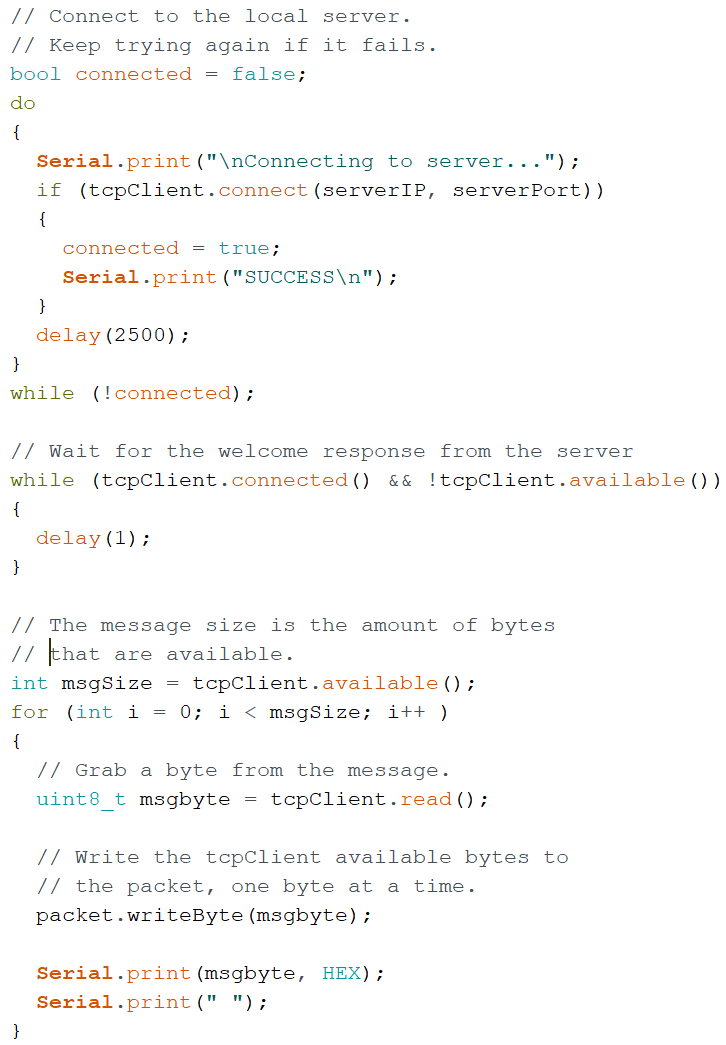


Fig. 9. Connection to C# Server, received welcome response from the server, then the response was written to a packet.

Afterwards, this welcome packet’s contents are printed using packet.readInt() for packet size, packet.readInt() for packet type, and packet.readString() for the packet message, as displayed in Figure 10. The next parts of the packet to be read are stored in the earlier variables mentioned, clientId and clientPort. packet.readInt() is used twice, one for each respectively. Although they are stored, the values are printed first to the Arduino first.

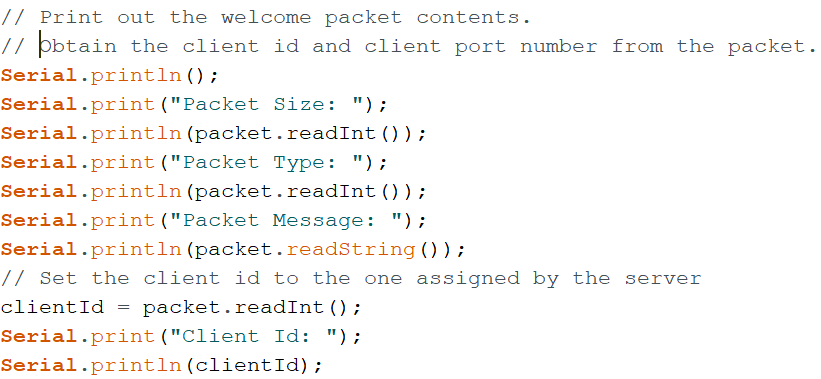


Fig 10. Welcome packet contents from the server are read and displayed on the Arduino client.

The packet is then cleared to allow for the Arduino to send a welcome response back to the C# server. Clearing and reusing the same packet is very useful in keeping memory used by the program in check. Figure 11 shows this code portion, and after clearing the welcome response is written to the packet using writeInt, and the clientId and length of the packet are also written to the packet using another writeInt and writeLength. After this, the welcome packet is sent to the server using tcpClient.write(packet.getBytes(), packet.size()). After this initial TCP connection, the UDP client is begun using udpClient.begin(CLIENT\_PORT), which will allow UDP packets to be sent to the server. Last before the code loop, startMillis is set to current millis() for use in a custom time delay setup that helps with performance.

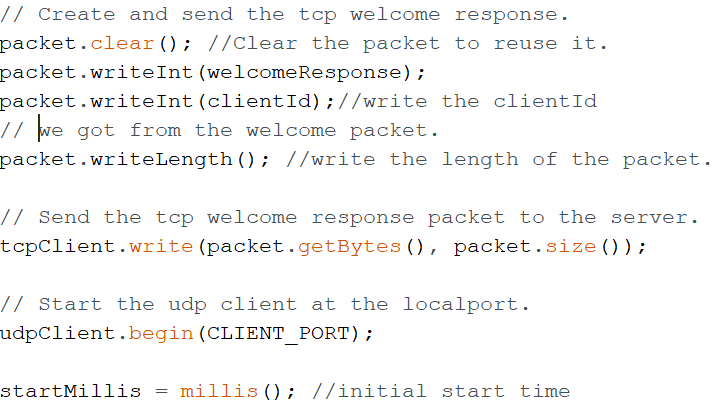


Fig. 11. Packet clear followed by a welcome response to the server. TCP is connected to send the packet and UDP connection is started.

Figure 12 shows the starting portion of the loop, where a Controller struct called vridge is first created, which is a struct that contains controller information that will be sent to VRidge, including head relation, hand type, quaternion, analog, button, and touchpad values. Next, if the IMU accelerometer and gyroscope are available, their values are read, and all the previous accelerometer and gyroscope values are calibrated accordingly. Both functions fusion.deltatUpdate and MahonyUpdate are used to convert IMU to quaternion data that will be used add rotational data to the vridge controller. For the block of code, startMillis, currentMillis, and delayMillis are important when relating to a custom delay period. Defined earlier, delayMillis is the cutoff time that must pass before new controller information can be sent to the vridge controller. startMillis was initialized before the loop, and it is set to current time again at the end of the loop. currentMillis is set to current time right before the if condition for delay, so when currentMillis minus startMillis is greater than or equal to delayMillis (delay time has been exceeded), the program then progresses. If the delay period has elapsed, first there are if statements that check if different buttons were pressed and sets their value to true if so. The only difference between the if statements are for the reset button, where resetButtonCounter and RESET\_BUTTON\_MS are used to determine if the reset button has been held for at least two seconds. If it has, resetCounter is set back to 0, the IMU is calibrated, and the quaternion is reset.



Fig. 12. Controller vridge is created, IMU values are read and calibrated, IMU data is converted to a quaternion, and button press conditions are set.

After the button conditions, the quaternion is obtained using fusion.getQuaternion(), shown in Figure 13. Then the quaternion data is put into vridge rotation variables and the vridge Controller is sent to sendControllerStatusUDP.

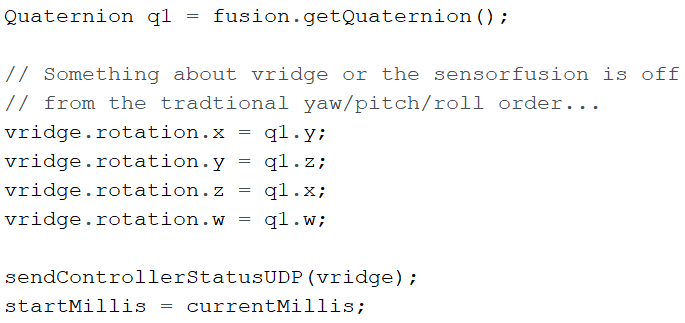


Fig. 13. Quaternion is obtained and it is used to set rotational data to the vridge Controller. Then the Controller is sent to sendControllerStatusUDP

Figure 14 shows that the function sendControllerStatusUDP puts all the controller data into a packet, such as head and hand data, rotational data, button presses, and ClientId. Next, udpClient.beginPacket creates the UDP connection, udpClient.write sends the packet, and last, udpClient.endPakcet closes the connection.



Fig. 14. Controller data is written to a packet, and the packet is sent to the server using a UDP connection.

# Conclusion and Future Outlook

## Conclusion

Our project was mostly successful in terms of implementing the Arduino microcontroller as a second controller using phone- based VR headsets. The second controller showed up in VR games, and the buttons worked correctly. The controller was responsive, but due to imperfections in the IMU, notably that it was 3 degrees of freedom (DoF) instead of 6 DoF, many VR games would not play correctly, and there is a slight drift in the positional variables over time.

## Future Outlook

For the future, implementing a touchpad, as well as home and system buttons would match the functionality of the Arduino controller with other VR controllers. A LiPo battery circuit and 3D printing would also be helpful to create a fully wireless system as the current design is impractical for harsher movements. Lastly, replacing the Arduino Nano 33 IoT with the Arduino Nano 33 BLE would help with accuracy and possibly drifting, as the LSM9DS1 IMU it has is more accurate, with the downside that it is only Bluetooth, not WiFi.

# References

1. Arduino. (2019, December 24). ArduinoLSM6DS3. https://www.arduino.cc/en/Reference/ArduionLSM6DS3
2. Arduion. (2019, December 24). WiFiNINA. https://www.arduino.cc/en/Reference/WiFiNINA
3. Arduino. (2019, December 24). WiFiUDPConstructor. https://www.arduino.cc/en/Reference/WiFiUDPConstructor
4. *RiftCat/vridage-api*. GitHub. https://github.com/RiftCat/vridge-api.
5. Stoffregen, P. (2018, March). aster94/SensorFusion. GitHub. https://github.com/aster94/SensorFusion.
6. Weiland, T. (2019). tom-weiland/tcp-udp-networking. GitHub. https://github.com/tom-weiland/tcp-udp-networking.
7. Weiland, T. (Director). (2019, October 26). Connecting Unity Clients to a Dedicated Server | C# Networking Tutorial [Video file]. https://www.youtube.com/watch?v=uh8XaC0Y5MA&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&ab\_channel=TomWeiland