Schweizerische
Fachschule
TEKO

Projektarbeit Embedded Systems

Drohnensteuerung

Studiengang zum

Dipl. Informatiker HF Applikationsentwicklung

an der TEKO Bern

Klasse:

B-TIN-19-T-a

Eingereicht von

Sascha Dubois

Michael Neuhaus

15. September 2022

Dozent: Herr Xavier Pfaff

Inhaltsverzeichnis

1 Situationsanalyse **1**
1.1 Technische Details Arduino Nano 33 BLE Sense . . . 1

2 Konzept **3**
2.1 UseCase . . . 3
2.2 Sequenzdiagramme . . . 3
2.3 Klassendiagramm . . . 6
2.4 Beschreibung der Klassen und Funktionen . . . 7
2.4.1 Vector_type . . . 7
2.4.2 Arduino_LSM9DS1 . . . 7
2.4.3 "main" Klasse . . . 7
2.4.4 PID Controller . . . 8
2.4.5 ProtocolHandler . . . 8

3 Implementation **9**
3.1 Berechnung der Orientation . . . 9
3.1.1 Ursprünglicher Lösungsansatz (Gyroskop) . . . 9
3.1.2 Erweiterter Lösungsansatz (Beschleunigungssensor) . . . 10
3.1.3 Definitiver Lösungsansatz . . . 11
3.1.4 PID-Loop . . . 12
3.1.5 PID-Algorythmus . . . 13
3.2 Protokoll zum Empfangen und Senden der Daten . . . 14
3.2.1 Datenempfang . . . 15
3.2.2 Senden der Daten . . . 16
3.3 Game Demo . . . 17
3.3.1 Hauptmenü . . . 17
3.3.2 Verbindung herstellen . . . 18
3.3.3 PID-Setup . . . 18
3.3.4 Gameziel . . . 19
3.4 3D Druck . . . 20

4 Ablage Dokumentation und Code **21**

1 Situationsanalyse

Im Rahmen der Gruppenarbeit im Fach Embedded Systems wollen wir mit dem Arduino Nano 33 BLE Sense ein Projekt durchführen.
Der Arduino Nano liefert gleich mehrere verbaute Sensoren mit. Unter anderem ist auf dem Board ein 9-achsige Trägheitsmesseinheit verbaut, welche uns auf die Idee brachte, eine Drohne mit diesem Sensor zu steuern.
In der weiteren Analyse hat sich aber gezeigt, dass das Protokoll zum Senden an die Drohne nicht so einfach umzusetzen ist. Aus diesem Grund haben wir uns dann entschieden, eine einfache Simulation mit Unity zu erstellen, welche mit dem Arduino Nano über USB gesteuert werden kann.

1.1 Technische Details Arduino Nano 33 BLE Sense

Als Übersicht sind in dieser Skizze die Sensoren des Arduino Nano 33 BLE Sense dargestellt. Für unser Projekt verwenden wir nur die beiden Sensoren, das Gyroskop und den Beschleunigungssensor. Über die USB-Schnittstelle findet der Datenaustausch zwischen Computer und Arduino statt.

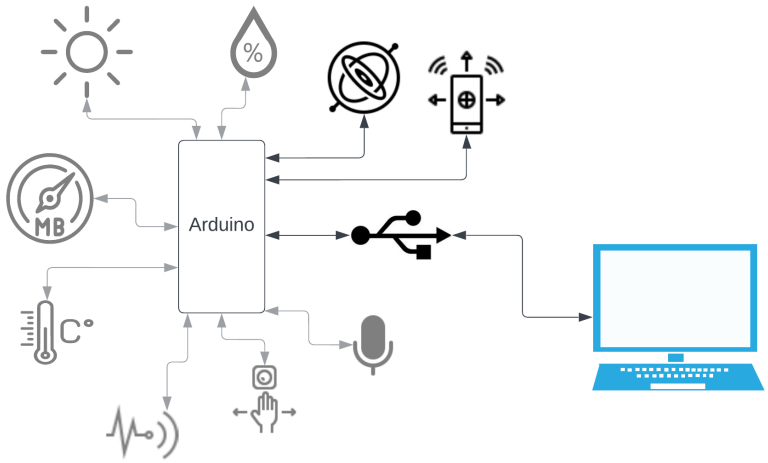

Abb. 1: Arduino Sensoren

Hier werden die wichtigsten Daten des Arduino Boards tabellarisch aufgelistet:

Takt	64Mhz
Bit	32
Mikrocon-troller	nRF52840
Flash	1024KB
SRAM	256KB
Anschlüsse	USB, I2C, SPI, Bluetooth
LSM9DS1	**Das LSM9DS1 ist ein System-in-Package mit einem 3D-Digital-Linearbeschleunigungssensor, ein 3D-Digital Winkelgeschwindigkeitssensor und ein digitaler 3D-Magnet Sensor. Das LSM9DS1 hat einen linearen Beschleunigungsendwert von ± 2 g / ± 4 g / ± 8 g / ± 16 g, ein Magnetfeld mit voller Skala von ±4/±8/±12/±16 Gauss und einer Winkelgeschwindigkeit von ±245/±500/±2000 dps.**
LPS22HB	Sensor für den barometrischen Druck und die Umgebungstemperatur Der Barometer hat eine Range von 260 bis 1260 hPa.
HTS221	Kapazitiver Digitalsensor für relative Feuchte und Temperatur 0 bis 100% relative Feuchte mit einer Genauigkeit von ±3.5%rH Der Temperatursensor misst im Range von 15 bis 40°C mit einer Genauigkeit von ±0.5°C und im Range von 0 bis 60°C mit einer Genauigkeit von ±1°C
APDS-9960	Digitaler Näherungs-, Umgebungslicht-, RGB- und Gestensensor
MP34DT05	Digitales Mikrofon, z.B. für Spracherkennung

Tabelle 1: Technische Details

2 Konzept

2.1 UseCase

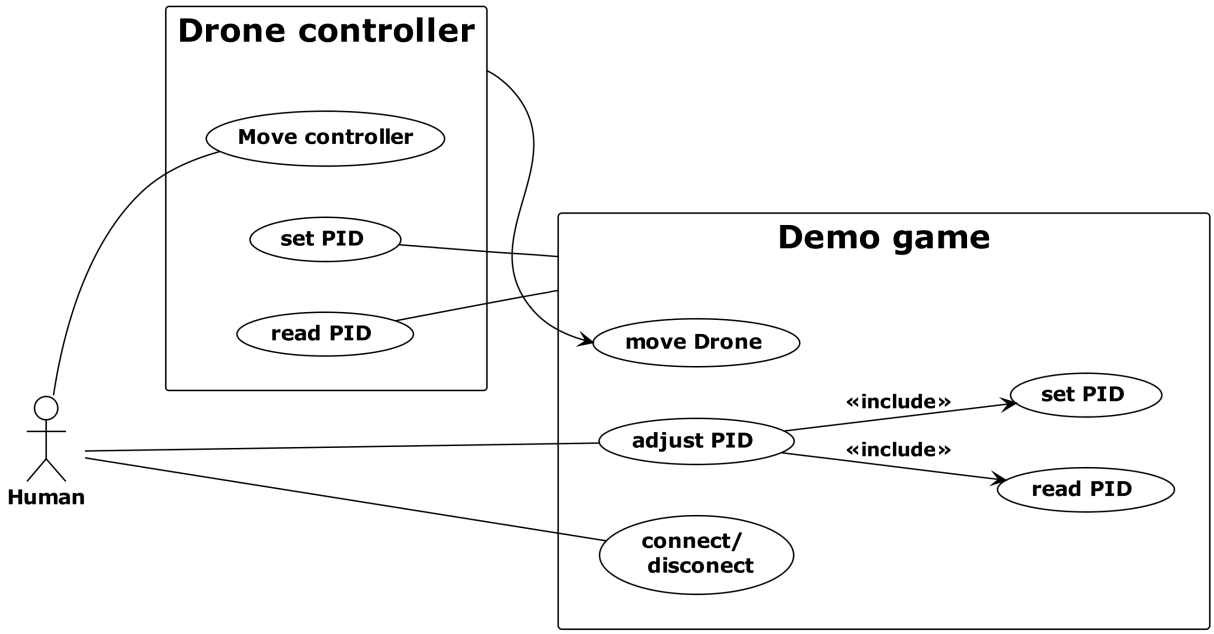

Abb. 2: UseCase Drohnensteuerung

2.2 Sequenzdiagramme

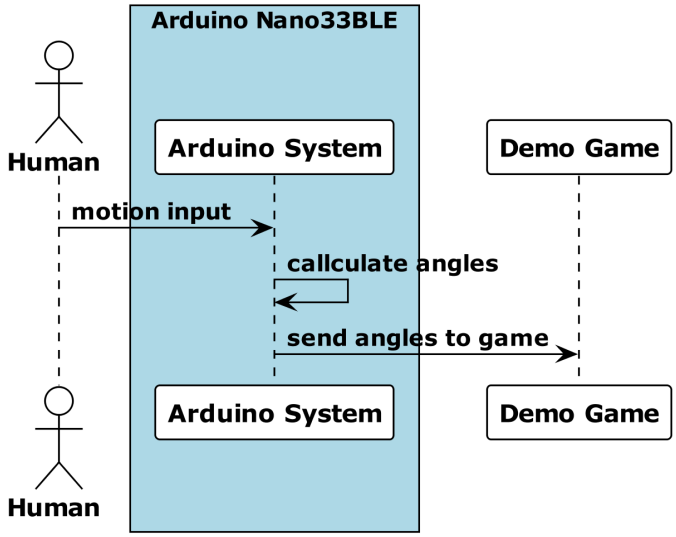

Abb. 3: System

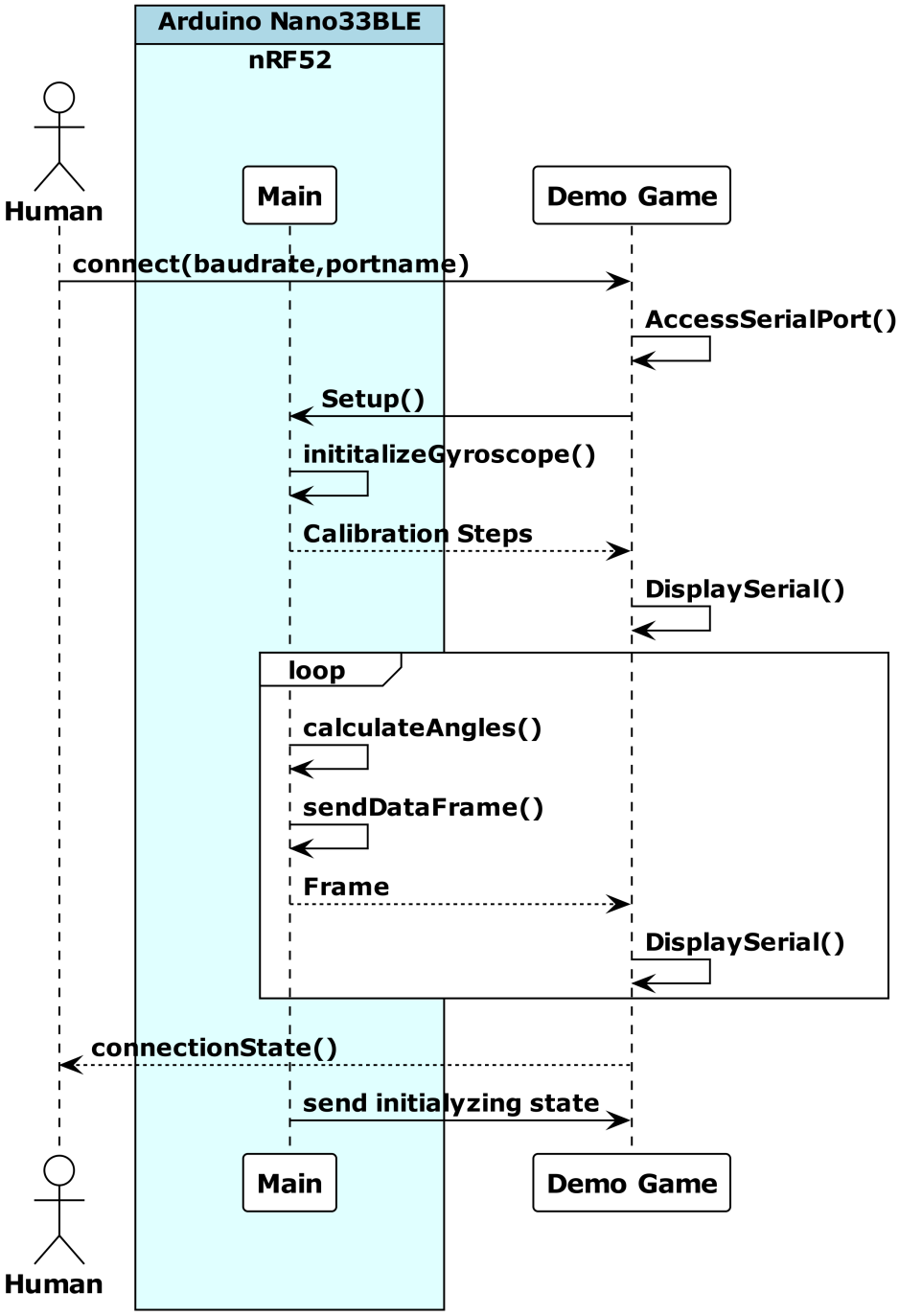

Abb. 4: Verbindung herstellen

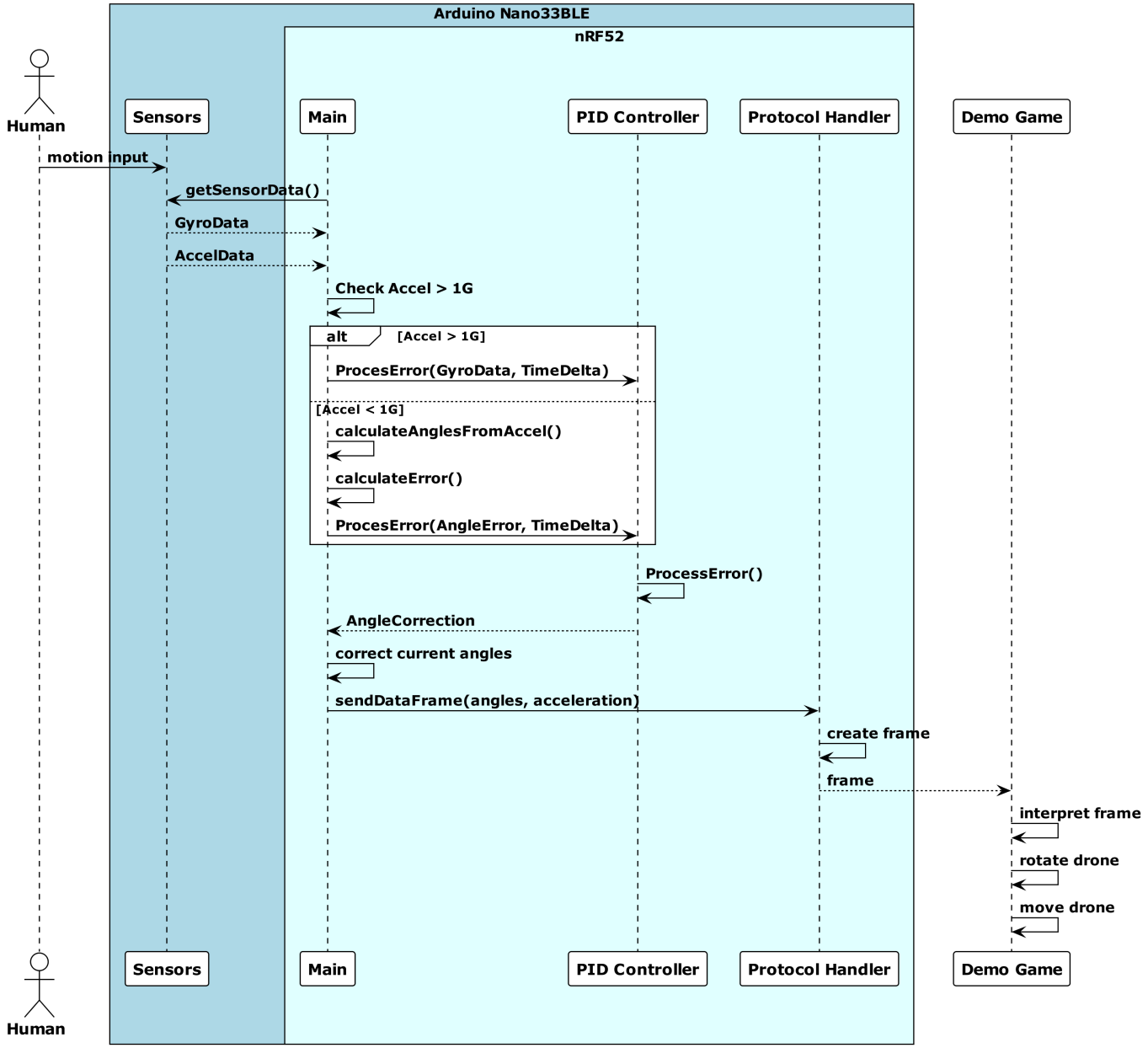

Abb. 5: Move controller

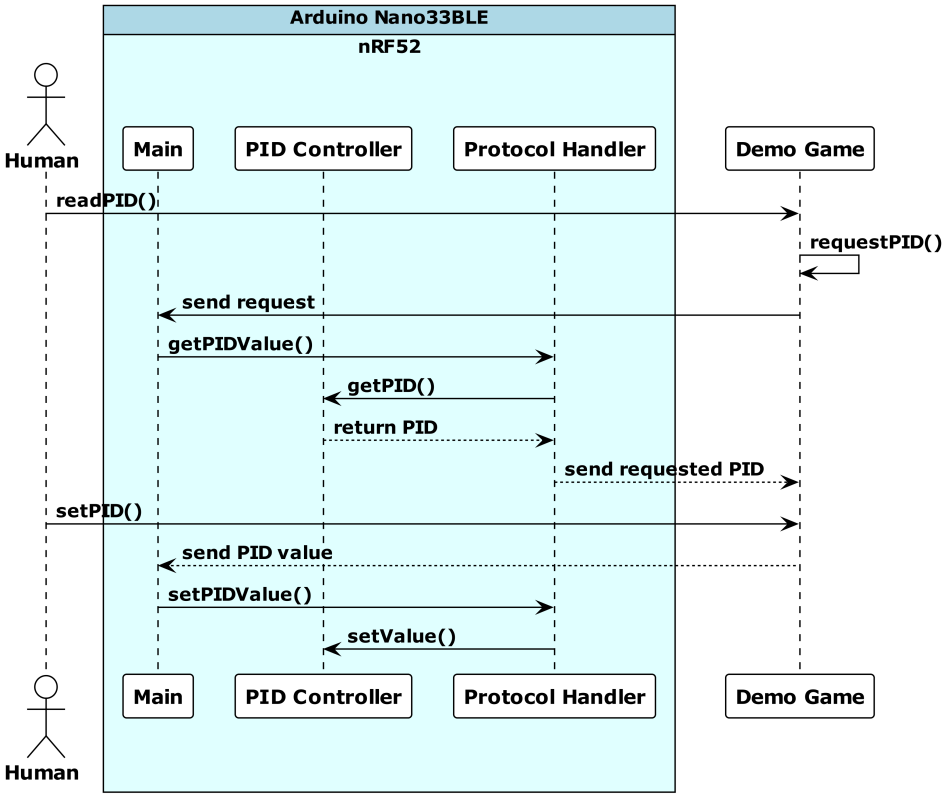

Abb. 6: Adjust PID

2.3 Klassendiagramm

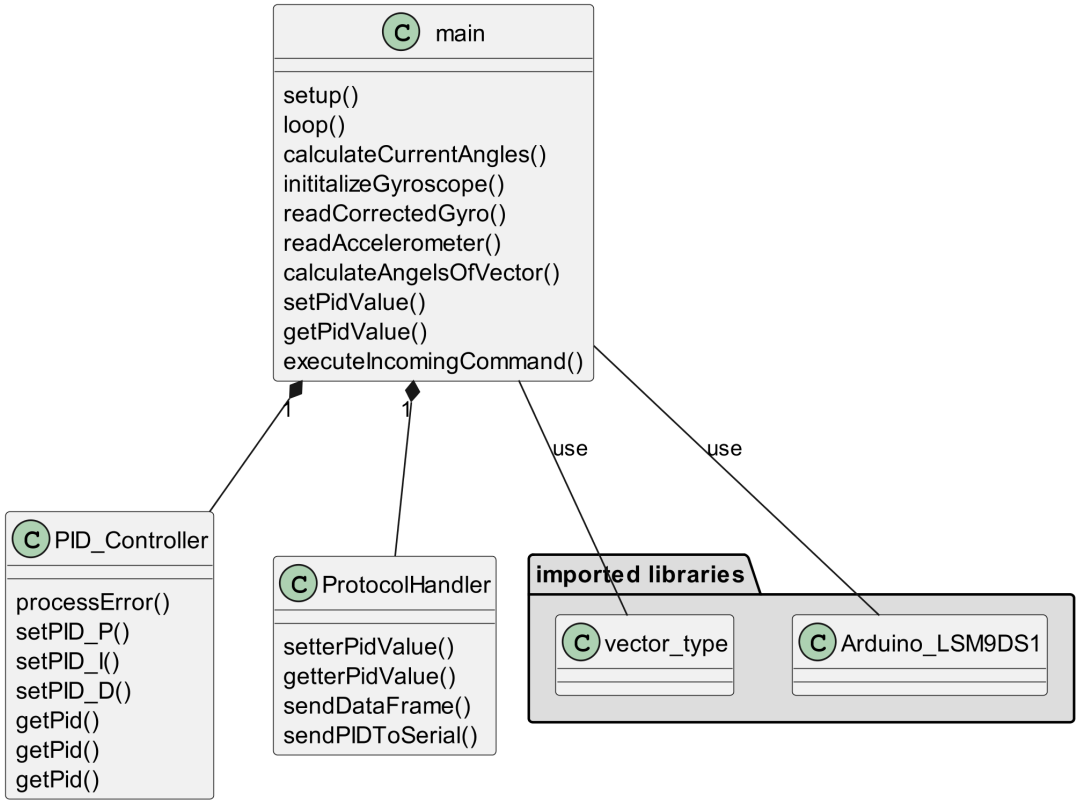

Abb. 7: Klassendiagramm

2.4 Beschreibung der Klassen und Funktionen

In diesem Kapitel werden die benutzten Klassen und deren Funktion kurz beschrieben.

2.4.1 Vector_type

Bibliothek für 3D-Vektoren und Quaternionen. Einfache Bibliothek, die Strukturen für einen 3D-Vektor und eine 4D-Quaternion enthält. Alle grundlegenden Operationen sind zusammen mit Rotationen enthalten.

2.4.2 Arduino_LSM9DS1

Bibliothek um die Accelerometer, Magnetometer und gyroscope Daten des LSM9DS1 IMU vom Arduino auszulesen. Die Magnetometer-Daten haben wir für dieses Projekt nicht verwendet.

2.4.3 "main" Klasse

Die "main" Klasse ist die hauptklasse welche alle anderen Klassen Importiert. Hier befindet sich ebenfalls die setup() Funktion und die loop() Funktion welche im Arduino vorgeben sind.

Funktion	Beschreibung
setup()	Setupfunktion des Arduino. Wird ausgeführt wenn der Arduino gestartet wird.
loop()	Der Loop ist eine Endlosschleife, der nach jedem Durchlauf erneut aufgerufen wird.
initializeGyroscope()	Mit dieser Funktion wird das Gyroskop beim einschalten initialisiert und kalibriert.
readAccelerometer()	Lesen der Beschleunigungssensordaten.
readCorrectetGyro()	Funktion zum Lesen der Gyrodaten und Subtrahieren der Korrekturwerte
calculateCurrentAngles()	Aktuelle Winkel aus den Gyrodaten berechnen.
calculateAnglesOfVector()	Berechnen der X-, Y-Eulerwinkel eines gegebenen Vektors (in Bezug auf die Achse)
executeIncomingCommand()	Diese Funktion wertet den eingegangenen Befehl aus und führt anschliessend die entsprechende Funktion aus.
setPidValue()	Wird aus executeIncomingCommand() aufgerufen und setzt die eingegangenen PID-Werte über den PID Controller.
getPidValue()	Wird aus executeIncomingCommand() aufgerufen und sendet die angeforderten PID Werte zurück.

Tabelle 2: Beschreibung der "main" Klasse

2.4.4 PID Controller

Die PID Controller werden zur Glättung der Sensorsignale verwendet.
Je nach Situation, wird aus den Gyroskop oder Beschleunigungsdaten ein Fehler berechnet. Dieser Fehler beschreibt die Diskrepanz zwischen der theoretischen und der von den Sensoren ausgelesenen Orientation des Boards. Mit Hilfe der PID Controller wird nun die theoretische Orientation entsprechend den Sensordaten angepasst. Der PID-Algorithmus wird hier als eine Art Filter eingesetzt und glättet somit fehlerhafte Sensordaten etwas aus.

Funktion	Beschreibung
processError()	Ein Fehler wird als Input eingegeben und durchläuft den PID Algorithmus
setPID_()	Setzt den eingehenden Wert einsprechend für P, I oder D.
getPID_()	Sendet den aktuellen P,I oder D- Wert zurück.

Tabelle 3: Beschreibung PIDController

2.4.5 ProtocolHandler

Mit dem ProtocolHandler wird das sende und empfangsprotokoll zum Steuern der Gamedemo kontrolliert.

Funktion	Beschreibung
setterPidValue()	Wertet den eingegangen Befehl zum setzen des PID-Werts aus und führt dann die entsprechende Funktion über den PIDController aus.
getterPidValue()	Wertet den eingegangen Befehl zum lesen des Aktuellen PID-Werts aus, holt sich den wert über den PIDController und sendet diesen über die Funktion sendPIDToSerial() zurück.
sendPIDToSerial()	Wird durch getterPIDValue() aufgerufen und sendet PID-Wert über Serial.println() als string zurück.
sendDataFrame()	Wird durch den loop() aufgerufen und sendet die berechneten Gyro- und Beschleunigungsdaten Serial.println() als string an das Demo-Game.

Tabelle 4: Beschreibung ProtocolHandler

3 Implementation

3.1 Berechnung der Orientation

Die Berechnung der aktuellen Orientation des Arduinos war eine grössere Herausforderung, als wir dies im Vorfeld geplant hatten. Das Problem war hauptsächlich, dass das Gyroskop, welches auf dem SOC verbaut wird, nicht sehr genaue Werte ausgibt.

3.1.1 Ursprünglicher Lösungsansatz (Gyroskop)

Ursprünglich haben wir geplant, dass wir anhand der Gyroskop-Daten (Das Gyroskop gibt die aktuellen Winkelgeschwindigkeiten für jede Drehachse in Grad/Sekunde aus) die aktuellen Winkel wie folgt berechnen können:

Die aktuellen Winkel ($CurrAngle_{x,y,z}$) werden zu begin mit 0 initialisiert, In der loop-Funktion des Arduinos werden die aktuellen Winkel mit dem Produkt aus der Winkelgeschwindikeit ($v_{x,y,z}$) und dem Zeitdelta (Δt) addiert.

$$
\begin{aligned}
CurrAngle_x &= CurrAngle_x + v_x \Delta t \\
CurrAngle_y &= CurrAngle_y + v_y \Delta t \\
CurrAngle_z &= CurrAngle_z + v_z \Delta t
\end{aligned}
$$

Diese Methodik hat jedoch dazu geführt, dass die Winkel, durch die Ungenauigkeit der Daten, schon nach kurzer Zeit davon driften. Um diese Problematik etwas in den Griff zu bekommen, haben wir eine Kalibrierungsfunktion implementiert, welche beim Starten des Arduinos durchloffen wird. Zum Kalibrieren darf der Arduino nicht bewegt werden. In der Funktion werden nun 500-mal die Gyroskopdaten abgefragt. Der Durchschnittswert dieser Daten wird anschliessend von zukünftigen Messungen abgezogen. Mit Hilfe der Kalibrierungsfunktion konnten wir bereits etwas genauere Resultate erzielen, brauchbar waren diese jedoch noch nicht.

3.1.2 Erweiterter Lösungsansatz (Beschleunigungssensor)

Da wir durch das Gyroskop keine genauen Winkel-Werte berechnen konnten, haben wir einen weiteren Lösungsansatz ins Auge gefasst:
Durch den Beschleunigungssensor können wir den aktuellen Beschleunigungsvektor auslesen. Solange der Beschleunigungssensor nicht künstlich beschleunigt wird, so zeigt dieser Vektor mit Länge ($9.81\frac{m}{s^2}$) immer Richtung Erdmitte. Anhand des Vektors können wir nun die aktuellen Winkelangaben (mit einigen Einschränkungen) sehr genau berechnen.

$$ang_{x,y} = arccos\frac{vect_{x,y}}{|\overrightarrow{vect}|}$$

```
// Calculate X,Y eulerangles of a given vector (in reference of axis)
void calculateAngelsOfVector(vec3_t _vectorIn, vec3_t* _anglesOut)
{
   //(*180/pi) => convert rad in degrees
  _anglesOut->x = (acosf(_vectorIn.x / _vectorIn.mag())) * 180 / PI;
  _anglesOut->y = (acosf(_vectorIn.y / _vectorIn.mag())) * 180 / PI;
  _anglesOut->z = 0; //Calculation of Z rotation is not possible
}
```

Die Einschränkungen dieser Methodik liegen auf der Hand, so können die Winkel wie gesagt nur berechnet werden, sofern der Arduino in nicht horizontal oder vertikal beschleunigt wird. Des Weiteren ist es nicht möglich die Rotation in der Z-Achse (Senkrecht zur Erdmitte) zu berechnen.
Um die Rotation in der Z-Achse berechnen zu können, müssten wir einen weiteren Sensor hinzuziehen: den Magnetometer. In unserem Projekt haben wir jedoch aufgrund des Zeitmangels darauf verzichtet und uns mit den X und Y werten begnügt.
Das Problem der Beschleunigung konnten wir jedoch fast komplett in den Griff bekommen.

3.1.3 Definitiver Lösungsansatz

Durch berechnen der Winkel mittels des Beschleunigungssensors konnten wir, solange der Arduino nicht bewegt wird, ziemlich genaue Resultate erzielen. In unserem definitiven Lösungsansatz haben wir das Problem der Bewegung wie folgt lösen können:
Wie bereits gesagt, hat der Vektor der Beschleunigung im Ruhezustand einen Betrag von 1G, ist dieser Betrag höher oder tiefer, so wird der Arduino beschleunigt. somit berechnen wir die Winkel anhand des Beschleunigungssensors nur, wenn dieser nicht künstlich beschleunigt wird.

```
void calculateCurrentAngles()
{
  delay(5);

  // Flag if sensor is beeing accelerated
  bool accelerating = !(accelVect.mag() < startVect.mag() +
                        thresholdVect && accelVect.mag() >
                        startVect.mag() - thresholdVect);

  // Calculate time delta
  float dTime = (millis() - timeStamp)/1000;

  // Calculate errors
  float xError = accelerating ? anglespeedVect.x * dTime :
                                accelEulerAnglesVect.x -
                                currentEulerAnglesVect.x;
  float yError = accelerating ? anglespeedVect.y * dTime :
                                accelEulerAnglesVect.y -
                                currentEulerAnglesVect.y;

  float zError = anglespeedVect.y * dTime;

  // Running PID loops for correcting the current angles
  xController.processError(xError, dTime);
  yController.processError(yError, dTime);
  zController.processError(zError, dTime);

  // Applying corrections to current angles
  currentEulerAnglesVect.x += xCorrection;
  currentEulerAnglesVect.y += yCorrection;
  currentEulerAnglesVect.z += zCorrection;

  timeStamp = millis();
}
```

Ansonsten nehmen wir die Daten des Gyroskops, um eine Schätzung vorzunehmen. Damit die Bewegungen geglättet werden, haben wir ausserdem sogenannte PID-Loops verwendet.

3.1.4 PID-Loop

PID-Loops, PID-Controller oder auch PID Regler sind häufig in der Regelungstechnik anzutreffen.
Sie werden hauptsächlich eingesetzt, um Fehler zu korrigieren und dabei das Überschiessen und das daraus folgende Oszillieren zu verhindern.

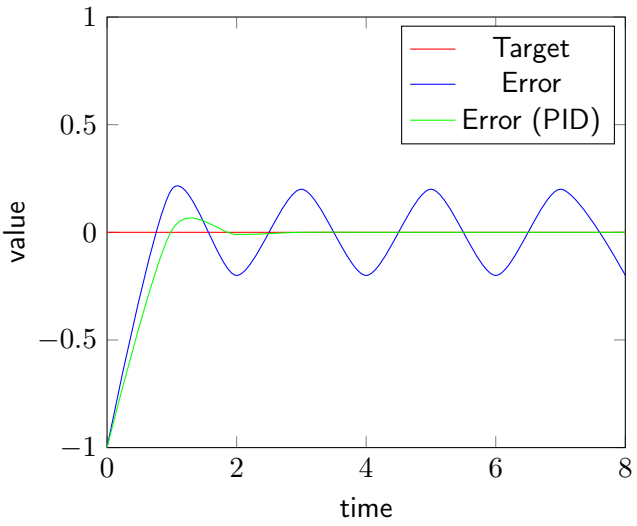

Abb. 8: Regelung mit und ohne PID-Loop

In unserem Projekt verwenden wir für alle Drehachsen des Arduinos jeweils ein PID-Loop. Die Sensordaten werden nicht direkt in einen Output verwandelt, stattdessen werden sie zum Korrigieren der aktuellen Winkel, welche zwischengespeichert werden, verwendet. Wie bereits im Lösungsansatz beschrieben, verwenden wir je nachdem ob eine Beschleunigung vorliegt die Gyro- oder Beschleunigungsdaten.
Werden die Beschleunigungsdaten verwendet so wird der Fehler, welcher durch den PID-Loop verarbeitet wird folgendermassen berechnet:

$$err_{x,y} = accelAngle_{x,y} - currAngle_{x,y}$$

Werden hingegen die Gyroskopdaten verwendet so kann als Fehler die Winkelgeschwindigkeit im Delta der Zeit verwendet werden.

$$err_{x,y} = wV_{x,y}\Delta t$$

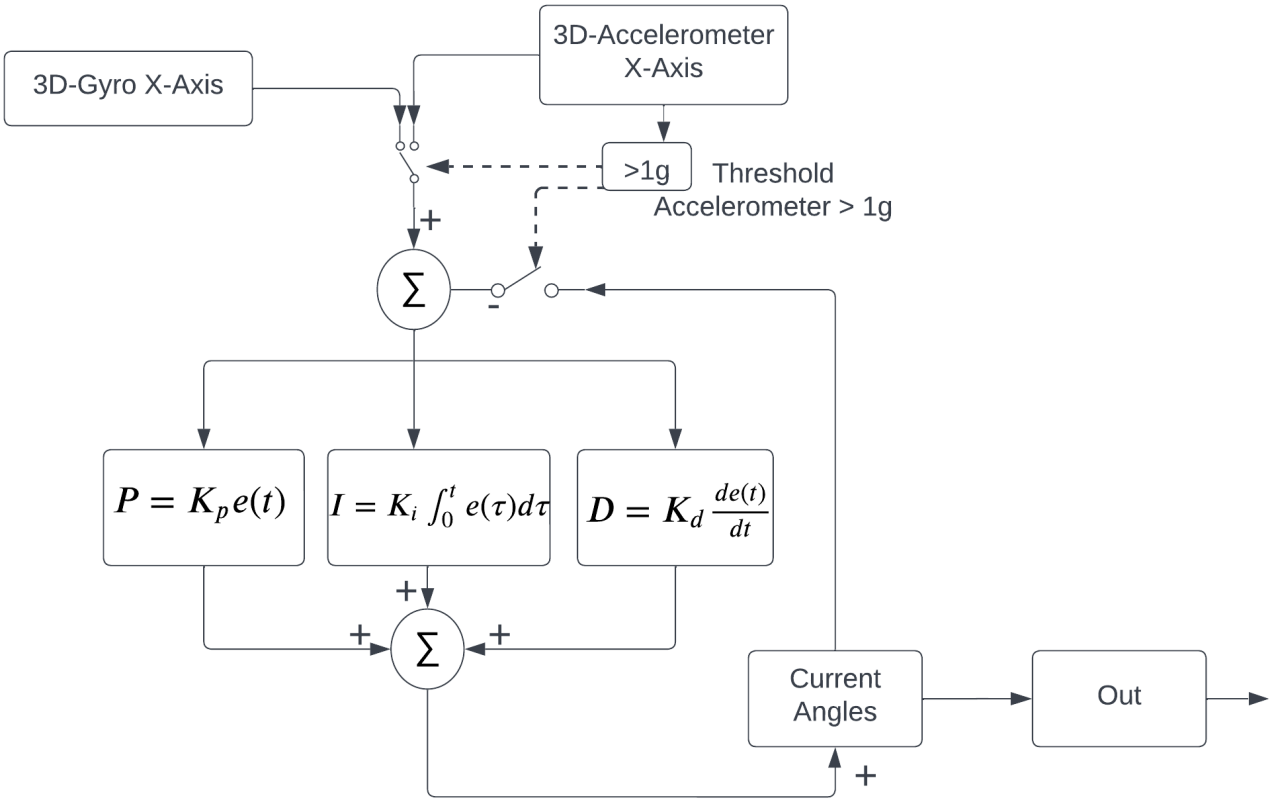

Abb. 9: PID Loop

3.1.5 PID-Algorythmus

$$Korrekturwert = K_p e(t) + K_i e_i \Delta t + \frac{K_d \Delta e}{\Delta t}$$

K_p	Proportional-Konstante
K_i	Integral-Konstante
K_d	Differential-Konstante
$e(t)$	aktueller Fehler
Δt	Zeit seit der letzten Korrektur
Δe	Differenz des aktuellen und letzten Fehlers
e_i	Fehlerintegral ($\int_0^t e(t)$)

Tabelle 5: Variablen Verzeichnis

```
1  // calculate the current error
2  void PIDController::processError(float _error, float _dTime)
3  {
4    integralError += _error;
5
6    *pOutput = p * _error + (i * integralError * _dTime) + (d * (_error - error) / _dTime);
7
8    error = _error;
9  }
```

3.2 Protokoll zum Empfangen und Senden der Daten

Um mit dem Arduino die Game-Demo zu steuern, mussten wir ein entsprechendes Protokoll zum senden und empfangen der Daten implementieren. Mit dem Arduino werden über Serial.println die Daten als string versendet.

Das Protokoll haben wir wie folgt definiert:

- Jeder Befehl, ob ausgehend oder eingehend startet mit vier chars, gefolgt von einer pipe "|" als Trennzeichen. Zum Abschluss des Befehl muss ein Semikolon ";' gesendet werden.
- Setzen der PID-Werte:
  - Beispiel: **PIDS|X|P|0.3;**
  - Der eingehende string muss mit PIDS starten. Das "S" steht für SET.
  - Der zweite char spezifiziert für welche Achse der Wert gesetzt werden soll. X, Y oder Z.
  - Der dritte char definiert welcher Wert, ob P, I oder D.
  - Nach der letzten Pipe wird der zu setzende Wert übertragen.
- Anfordern von PID-Werten:
  - Beispiel: **PIDR|X|I;**
  - Der eingehende string muss mit PIDR starten. Das "R" steht für READ.
  - Der zweite char spezifiziert für welche Achse der Wert gelesen werden soll. X, Y oder Z.
  - Der dritte char definiert welcher Wert, ob P, I oder D.
- Zurücksenden der angeforderten PID-Daten:
  - Beispiel: **PIDT|X|I|0.3;**
  - Der ausgehende string muss mit PIDT starten. Das "T" steht für Transfer.
  - Der zweite char spezifiziert für welche Achse der Wert gesendet wird. X, Y oder Z.
  - Der dritte char definiert welcher Wert, ob P, I oder D.
  - Nach der letzten Pipe wird der gelesene Wert übertragen. Dieser Wert muss beim Empfänger durch 10'000 gerechnet werden, da ein float über Serial.println auf zwei Kommastellen gekürzt wird und deshalb für den Transfer mit 10'000 multipliziert als int übergeben wird.

```
1  // Send the requested PID Value via Serial
2  void ProtocolHandler::sendPIDToSerial(float _fVal,
3                                        String _strAxis,
4                                        String _strPidParam)
5  {
6    int iVal = _fVal * 10000; // transfer float to int because serial (string)
7                              // will only print two digits after comma
8    Serial.println(String(chrTransferPid) +
9                   String(paramSeparator) +
10                  _strAxis               +
11                  String(paramSeparator) +
12                  _strPidParam           +
13                  String(paramSeparator) +
14                  iVal                   +
15                  String(cmdTerminator));
16 }
```

- Senden des Data-Frames zum Steuern der Game-Demo
  - Beispiel: **DATT|90.05|90.2|3.5|0.1|0.01|1.05**
  - Der ausgehende string muss mit DATT starten.
  - Die drei ersten Werte liefern die Gyro-Achswerte für X, Y und Z.
  - Die drei letzten Werte liefern die Accelerometer-Werte für X, Y und Z
  - DATT|X|Y|Z|X|Y|Z

```
void ProtocolHandler::sendDataFrame(float _iData[], int _iDataLength)
{
  String frame = chrDataTransfer;

  for(int i =0; i < _iDataLength ;i++)
  {
    frame = frame + paramSeparator + String(_iData[i]);
  }

  Serial.println(frame);
}
```

3.2.1 Datenempfang

Um die Daten über die Serialschnittstelle empfangen zu können, wird in der loop()-Funktion geprüft ob eine Nachricht eingeht. Ist dies der Fall, wird mit Serial.read() jeder char der eingehenden Nachricht in einen Buffer abgefüllt und ausgewertet. Durch die genaue Definition des Protokolls kann dann der Befehl ausgewertet und anschliessend ausgeführt werden.

```
// Check to see if anything is available in the serial receive buffer
while (Serial.available() > 0)
{
  // Create a place to hold the incoming message
  static char message[MAX_MESSAGE_LENGTH];
  static unsigned int message_pos = 0;

  // Read the next available byte in the serial receive buffer
  char inByte = Serial.read();

  // Message coming check for pipe
  if (inByte != paramSeparator && inByte != cmdTerminator &&
      message_pos < MAX_MESSAGE_LENGTH - 1)
  {
    // Add the incoming byte to message
    message[message_pos] = inByte;
    message_pos++;
  }
  else
  {
    // Adds terminator to message
    message[message_pos] = '\0';
    // Ready for next parameter
    paramPosition++;

    // Reset for the next message
    message_pos = 0;

    // Fill the incoming message into the correct string for later execution
    switch (paramPosition)
    {
    case command:
      strCommand = String(message);
```

```
34        break;
35      case axis:
36        strAxis = String(message);
37        break;
38      case pidParam:
39        strPidParam = String(message);
40        break;
41      case value:
42        fValue = String(message).toFloat();
43        break;
44      default:
45        break;
46      }
47
48      if (inByte == cmdTerminator)
49      {
50        paramPosition = 0;          // Reset parameter position
51        message_pos = 0;            // Reset message position, read for new message
52        executeIncomingCommand(); // Execute the incoming command
53      }
54    }
55  }
```

3.2.2 Senden der Daten

Die PID-Daten werden je Achse und je PID-Term separat versendet. Um die Drohne im Game zu steuern, wollten wir dies ebenfalls so implementieren, dies führte jedoch zu nicht besonders flüssigem Spielverhalten. Aus diesem Grund haben wir uns entschieden, immer alle X,Y und Z- Werte auf einmal zu versenden.

3.3 Game Demo

Das Ursprüngliche Ziel des Projektes war es, eine Drohne mittels Bewegungssteuerung zu Steuern.
Im Laufe des Projektes haben wir jedoch schnell festgestellt, dass dies in der kurzen Zeit und mit unserer begrenzten Erfahrung nicht möglich ist. Um unser System trotzdem testen zu können, haben wir uns dazu entschieden ein kleines Demo-Game zu entwickeln.
Das Game wurde mit Unity3D erstellt, die Kommunikation mit dem Controller funktioniert über eine Serielle Schnittstelle und unserem eigenen Kleinen Protokoll. Dabei war es uns wichtig, dass das die Kommunikation bidirektional erfolgen kann. Das heisst das der Benutzer mit dem Controller das Spiel steuern aber im Game den Controller auch konfigurieren kann.

3.3.1 Hauptmenü

Nach dem Start des Games wird das Game Hauptmenü angezeigt. Hier muss über "Connect" zuerst die Verbindung mit dem Arduino hergestellt werden. Anschliessend kann man das Game starten oder über "PID-Setup" die PID's auf dem Arduino anpassen.

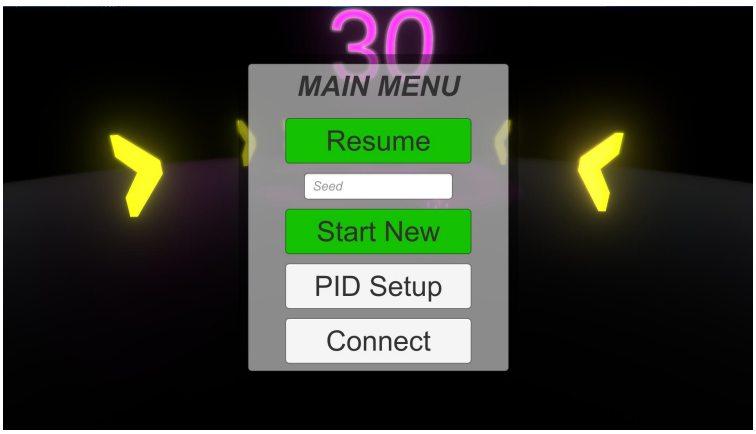

Abb. 10: Game Hauptmenü

3.3.2 Verbindung herstellen

In diesem Submenü wird die Verbindung zum Arduino hergestellt. Dazu muss der richtige COM-Port eingegeben werden, ansonsten kann die Verbindung nicht aufgebaut werden. Ist das herstellen der Verbindung erfolgreich, wird in der Statusleiste oben angezeigt, dass das Gyro Kalibriert wird. Währenddessen darf der Arduino nicht bewegt werden.

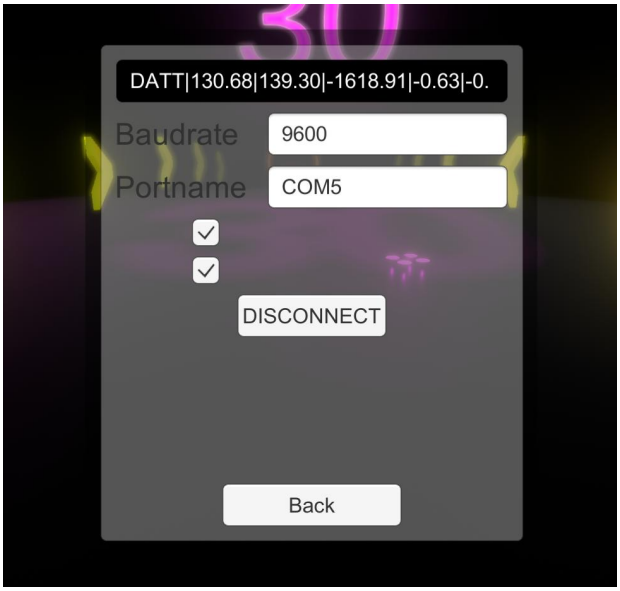

Abb. 11: Verbindung herstellen

3.3.3 PID-Setup

Falls man die voreingestellten Werte der PID-Konstanten anpassen möchte, kann man dies hier über das PID-Setup machen. Zuerst müssen die aktuellen Werte vom Arduino mit "Reload" geladen werden, danach können diese mit "Apply" auf den Arduino geschrieben werden.

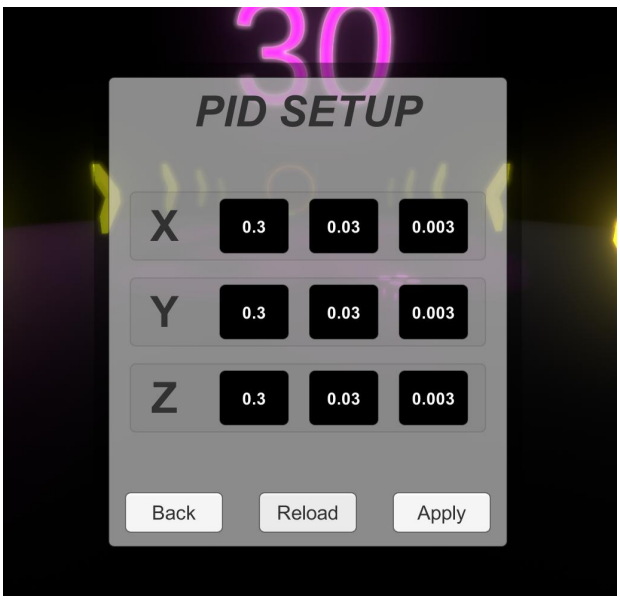

Abb. 12: PID Einstellen

3.3.4 Gameziel

Wenn die Verbindung erfolgreich hergestellt wurde, kann ein neues Game gestartet werden. Durch kippen des Arduino nach vorne fliegt die Drohne vorwärts und dann muss versucht werden durch die eingeblendeten Tore hindurchzufliegen.

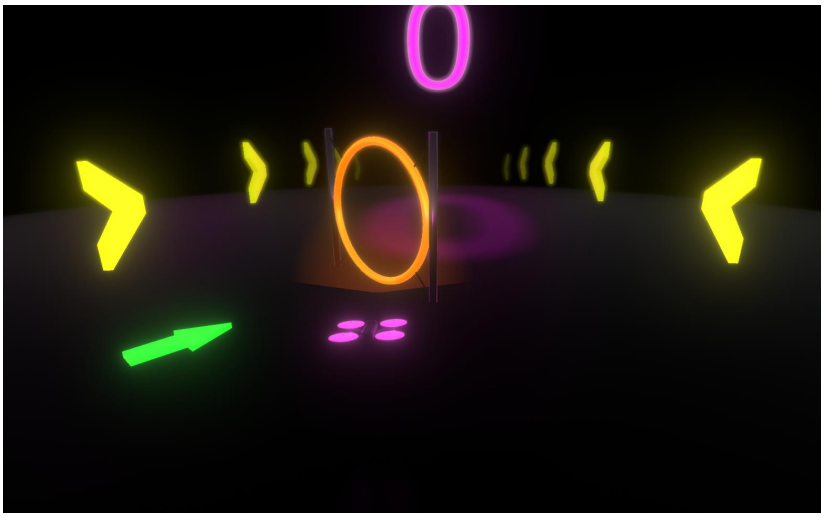

Abb. 13: In-Game Screenshot

3.4 3D Druck

Um das Game mit dem Arduino besser steuern zu können, haben wir einen Joystick gedruckt um das Benutzerfeeling zu verbessern.

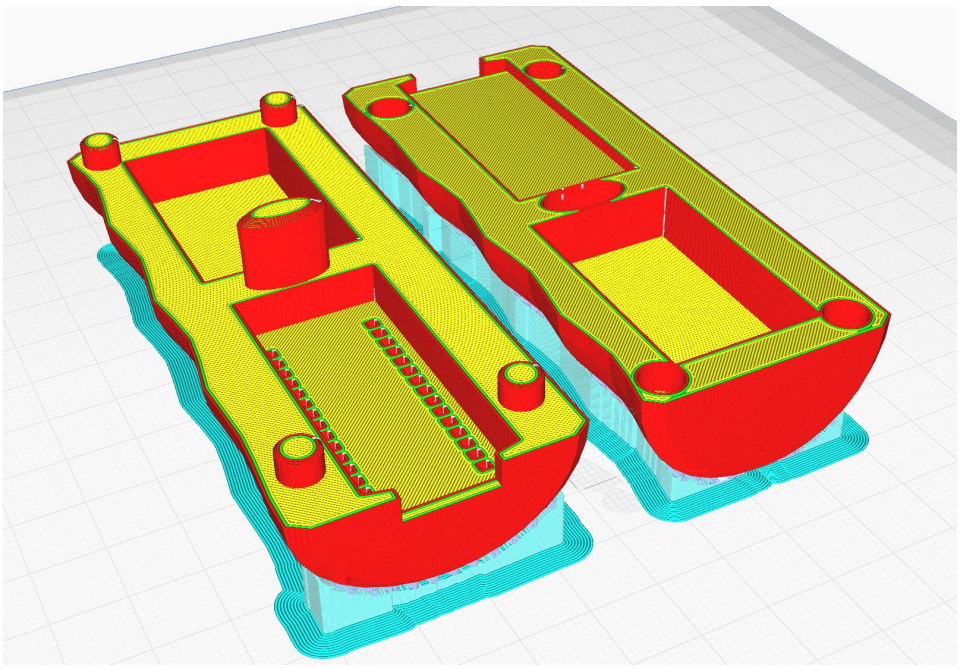

Abb. 14: 3D Druck Slicer

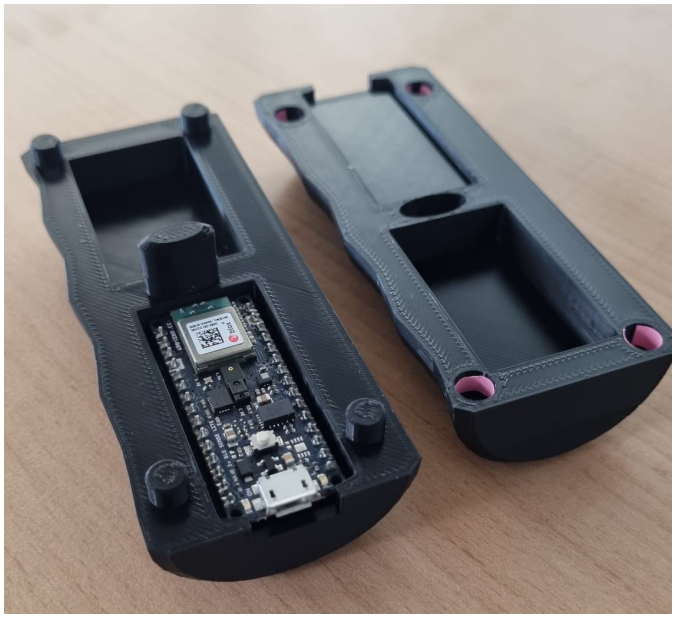

Abb. 15: 3D Druck geöffnet

Abb. 16: 3D Druck geschlossen

4 Ablage Dokumentation und Code

Der gesamte Code und die Dokumentation sind mit GIT über GitHub verwaltet.

Der Code kann über diesen Link abgerufen werden:
`https://github.com/sirtheta/Drohnensteuerung`

Abbildungsverzeichnis

1 Arduino Sensoren . . . 1
2 UseCase Drohnensteuerung . . . 3
3 System . . . 3
4 Verbindung herstellen . . . 4
5 Move controller . . . 5
6 Adjust PID . . . 6
7 Klassendiagramm . . . 6
8 Regelung mit und ohne PID-Loop . . . 12
9 PID Loop . . . 13
10 Game Hauptmenü . . . 17
11 Verbindung herstellen . . . 18
12 PID Einstellen . . . 18
13 In-Game Screenshot . . . 19
14 3D Druck Slicer . . . 20
15 3D Druck geöffnet . . . 20
16 3D Druck geschlossen . . . 20

Tabellenverzeichnis

1 Technische Details 2
2 Beschreibung der "main" Klasse 7
3 Beschreibung PIDController 8
4 Beschreibung ProtocolHandler 8
5 Variablen Verzeichnis 13