Contents

[1. Introduction 1](#_Toc514863429)

[2. Material 1](#_Toc514863430)

[3. Making the parts holding the sensors 2](#_Toc514863431)

[4. Reading and transferring data to Unity 2](#_Toc514863432)

[5. Receiving and processing data 3](#_Toc514863433)

[6. Optimising Data transfer 6](#_Toc514863434)

# Introduction

During the winter seasons, cold days and bad weather; cyclist enthusiasts only have a few options to exercise doing their favorite sport. We were looking for a way to make indoor training with a bike/trainer setup a bit more entertaining but most product available are either costly or just plain boring to use. This is why we started to develop Infinity Bike as an Open Source training video game. Infinity bike will read the speed and direction form your bicycle and offer a level of interactivity that cannot be easily found with bike trainers.

We take advantage of the simplicity available from Arduino microcontroller and a few 3D printed parts to secure inexpensive sensors to a bicycle mounted on a trainer. The information is relayed to a video game made with the popular game making engine, Unity. By the end of this instructable, you should be able to set-up your own sensors on your bike and transfer the information of your sensors to Unity. We even included a track on which you can ride along and test out your new set-up.

# Materials

The material list you will need might vary a bit; for example, the size of you bike will dictate the lengths of the jumper cables you need but here are the main parts you will need. You could probably find cheaper prices for each piece on website like AliExpress but waiting 6 months for shipping isn’t always an option so were using the slightly more expensive parts so the estimate isn’t skewed.

1 x [Arduino nano](https://store.arduino.cc/usa/arduino-nano) ($22.00)

1x [Mini Breadboard](https://www.amazon.com/Elegoo-6PCS-tie-points-Breadboard-Arduino/dp/B01EV6SBXQ/ref=br_lf_m_d2xh2ecztuwj4fa_img?_encoding=UTF8&s=pc) ($1.33/unit)

1 x [10K Potentiometer](https://www.amazon.com/Uxcell-a15011600ux0235-Linear-Rotary-Potentiometer/dp/B01DKCUVMQ/ref=zg_bs_306810011_1?_encoding=UTF8&psc=1&refRID=ZPQ3XYHZQJXNW7W0AH5P) ($1.80/unit)

1 x [Halls sensor](Hall%20Effect%20KY-003%20Magnetic%20Sensor%20Module%20DC%205V%20For%20Arduino%20PIC%20AVR%20Smart%20Cars) ($2.79)

Assorted 3D printed parts

# Making the parts holding the sensors

# Reading and transferring data to Unity

The Arduino and Unity code will work together to collect, transfer and process the data from the sensors on the bike. Unity will request the value from the Arduino by sending a string through the serial and wait for the Arduino to respond with the values requested.

First, we prepare the Arduino with the library [Serial Command](https://github.com/kroimon/Arduino-SerialCommand) which is used to manage the requests from Unity by pairing a request string with a function. A basic set up for this library can be made as follow;

#include "SerialCommand.h"

SerialCommand sCmd;

void setup()

{

sCmd.addCommand("TRIGG", TriggHanlder);

Serial.begin(9600);

}

void loop()

{

while (Serial.available() > 0)

{

sCmd.readSerial();

}

}

void TriggHandler ()

{

/\*Read and transmit the sensors here\*/

}

The function TriggHandler is attached to the object SCmd. If the serial receives a string that matches the attached command (in this case TRIGG), the function TriggHandler is executed.

We use potentiometer to measure the steering direction and a halls sensor to measure the rotation per minute of the bicycle. The readings from the potentiometer can be easily made using the build-in functions from the Arduino. The function TriggHandler can then print the value to the serial with the following change.

void TriggHandler ()

{

/\*Reading the value of the potentiometer\*/

Serial.println(analogRead(ANALOGPIN));

}

The halls sensor has a bit more setting up before we can have useful measurements. Contrary to the potentiometer, the instant value of the halls sensor is not very useful. Since were trying to measure the speed of the wheel, the time between triggers is what were interested in.

Every function used in the Arduino code takes time and if the magnet lines up with the halls sensor at the wrong time the measurement could be delayed at best or skipped entirely at worst. This is obviously bad because the Arduino could report a speed that is MUCH different than the actual speed of the wheel.

To avoid this, we use a feature of Arduinos called attach interrupt which allows us to trigger a function whenever a designated digital pin is triggered with a rising signal. The function rpm\_fun is attached to an interrupt with a single line of code added to the setup code.

void setup()

{

sCmd.addCommand("TRIGG", TriggHanlder);

attachInterrupt(0, rpm\_fun, RISING);

Serial.begin(9600);

}

The function rpm\_fun is used to calculate the speed and is defined as;

unsigned long lastRevolTime = 0;

unsigned long revolSpeed = 0;

void rpm\_fun()

{

unsigned long revolTime = millis();

unsigned long deltaTime = revolTime - lastRevolTime;

/\*revolSpeed is the value transmitted to the Arduino code\*/

revolSpeed = 20000 / deltaTime;

lastRevolTime = revolTime;

}

TriggHandler can then transmit the rest of the information when requested.

void TriggHanlder ()

{

/\*Reading the value of the potentiometer\*/

Serial.println(analogRead(ANALOGPIN));

Serial.println(revolSpeed);

}

We now have all of the building blocks that can be used to build the Arduino code which will transfer data through the serial to when a request is made by Unity. If you want to have a copy of the full code, you can download it on out git hub **(here)**. To test if the code was set-up properly, you can use the serial monitor to send TRIGG; make sure you set the line ending to Carriage return. The next section will focus on how unity can request and receive the information from the Arduino.

# Receiving and processing data

Unity is a great software available for free to hobbyist interested in game making; it comes with a large number of functionalities that can really cut down on time setting up certain things such as threading or GPU programming (AKA shading) without restricting what can be done with the C# scripts. Unity and Arduino microcontrollers can be used together to create unique interactive experiences with a relatively small budget.

The focus of this instructable is to help set-up the communication between Unity and the Arduino so we wont dive too deep into most of the features available with Unity. There are plenty of GREAT tutorials for unity and an incredible community that could do a much better job explaining how Unity works. However, there is a special prize for those who manage to work their way through this instructable that serves as a little showcase of what could be done. You can download on our Github our first attempt a making a track with realistic bike physics.

First, let’s go through the bare minimum that has to be done in order to communicate with an Arduino through the serial. It will be quickly apparent that this code is not suitable to gameplay but it’s good to go through every step and learn what the limitations are.

In Unity, create a new scene with a single empty GameObject named ArduinoReceive at attache a C# script also named ArduinoReceive. This script is where we will add all the code that handles the communication with the Arduino.

There is a library that must be access before we can communicate with the serial ports of your computer; Unity must be set-up to allow certain libraries to be used. Go to Edit->ProjectSerring->Player and next to the Api Compatibility Level under Configuration switch **.NET 2.0 Subset** to **.NET 2.0**. Now add the following code at the top of the script;

using System.IO.Ports;

This will let you access the SerialPort class which you could define as an object to the ArduinoReceive Class. Make it private to avoid any interference from another script.

private SerialPort arduinoPort;

The object arduinoPort can be opened by selecting the correct port (e.g. in which USB the Arduino is connected) and a baud rate (i.e. the speed at which the information is sent). If you’re not sure in which port the Arduino is plugged in you can find out either in the device manager or by opening the Arduino IDE. For the baud rate, the default value on most device is 9600, just make sure you have this value in your Arduino code and it should work.

The code should now look like this;

sing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System.IO.Ports;

public class ArduinoReceive : MonoBehaviour

{

private SerialPort arduinoPort;

// Use this for initialization

void Start()

{

arduinoPort = new SerialPort("COM5", 9600);

arduinoPort.Open();

WriteToArduino("TRIGG");

}

}

Your COM number will most likely be different. If you’re on a MAC, you’re COM name might have a name that looks like this /dev/cu.wchusbserial1420. Make sure the code from the section 4 is uploaded to the Arduino and the serial monitor is closed for the rest of this section and that this code compiles without problem.

Let’s now send a request to the Arduino every frame and write the results to the console window. Add the WriteToArduino function to the class ArduinoReceive. The carriage return and new line are necessary for the Arduino code to parse the incoming instruction properly.

private void WriteToArduino(string message)

{

message = message + "\r\n";

arduinoPort.Write(message);

arduinoPort.BaseStream.Flush ();

}

This function can then be called in the Update loop.

void Update () {

WriteToArduino ("TRIGG");

Debug.Log("First Value : " + arduinoPort.ReadLine());

Debug.Log("Second Value : " + arduinoPort.ReadLine());

}

The code above is the bare minimum you need to read the data from an Arduino. If you pay close attention to the FPS given by unity, you should see a significant drop in performance. In my case, it goes from around 90 FPS without reading/writing to 20 FPS. If your project doesn’t require frequent updates it might be sufficient but for a video game, 20 FPS is much too low. The next section will cover how you can improve the performance by using multi threading.

# Optimising Data transfer

The previous section covered how to set-up basic communication between the Arduino and Unity program. The major problem with this code is the performance. In it’s current implementation, Unity has to wait for the Arduino to receive, process and answer the request. During that time, the Unity code has to wait for the request to be done and does nothing else. We solved this issue by creating a thread that will handle the requests and store the variable on the main thread.

To begin, we must include the threading library by adding;

using System.Threading;

Next, we setup the function that we are starting in the the threads. AsynchronousReadFromArduino starts by writing the request to the Arduino with the WrtieToArduino function. The reading is enclosed in a try-catch bloc, if the read timeout, the variables remain null and the OnArduinoInfoFail function is called instead of the OnArduinoInfoReceive.

private void AsynchronousReadFromArduino()

{

string rotString = null;

string speedString = null;

WriteToArduino("TRIGG");

try {rotString = arduinoPort.ReadLine();}

catch (TimeoutException) {}

try {speedString = arduinoPort.ReadLine();}

catch (TimeoutException) {}

if (rotString != null && speedString != null)

{

OnArduinoInfoReceive(rotString, speedString);

}

else

{

OnArduinoInfoFail();

}

}

Next we define the OnArduinoInfoFail and OnArduinoInfoReceive functions. For this instructable, we print the results to the console but you could store the results into the variables you need for your project.

private void OnArduinoInfoFail()

{

Debug.Log("Reading failed");

}

private void OnArduinoInfoReceived(string rotation, string speed)

{

Debug.Log("Readin Sucessfull");

Debug.Log("First Value : " + rotation);

Debug.Log("Second Value : " + speed);

}

The last step is to start and stop the threads that will request the values from the Arduino. We must ensure that the last thread is done with it’s last task before starting a new one. Otherwise, multiple requests could be done to the Arduino at a single time which could confuse the Arduino/Unity and yield unpredictable results.

private Thread activeThread = null;

void Update()

{

if (activeThread == null || !activeThread.IsAlive)

{

activeThread = new Thread(AsynchronousReadFromArduino);

activeThread.Start();

}

}

If you compare the performance of the code with the one we wrote at the section 5, the performance should be significantly improved.

# Where next?

We prepared a demo that you can download on our Github, download the code and the game and ride through our track. It’s all set up for a quick workout and we hope it can give you a taste of what you could build if you use what we taught you with this instructable.