# Project Report and Demo

B10901112 陳品翔　B10901156 柯育杰　B10901161 張梓安

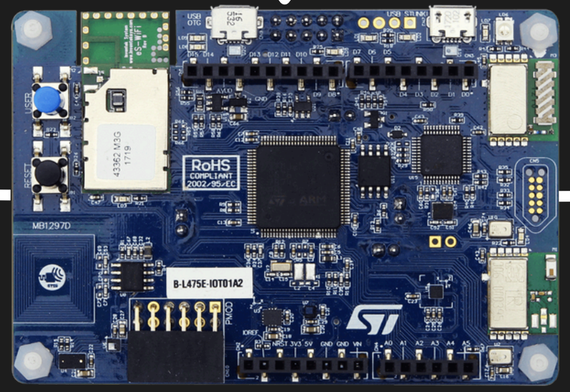
2024.06.06

# Motivation

- The game subway surfer is a popular mobile game
- It will be fun to extend the gamepaly from using only finger to the whole body motion like hopping left and right or jumping.
- We also seen previous project to some game related to motion detection.

# Our original plan

Collect acceleration data
and determine motion type




STM32 board

motions type
send via Wifi

Perform motions based
on received motion type

Python game server

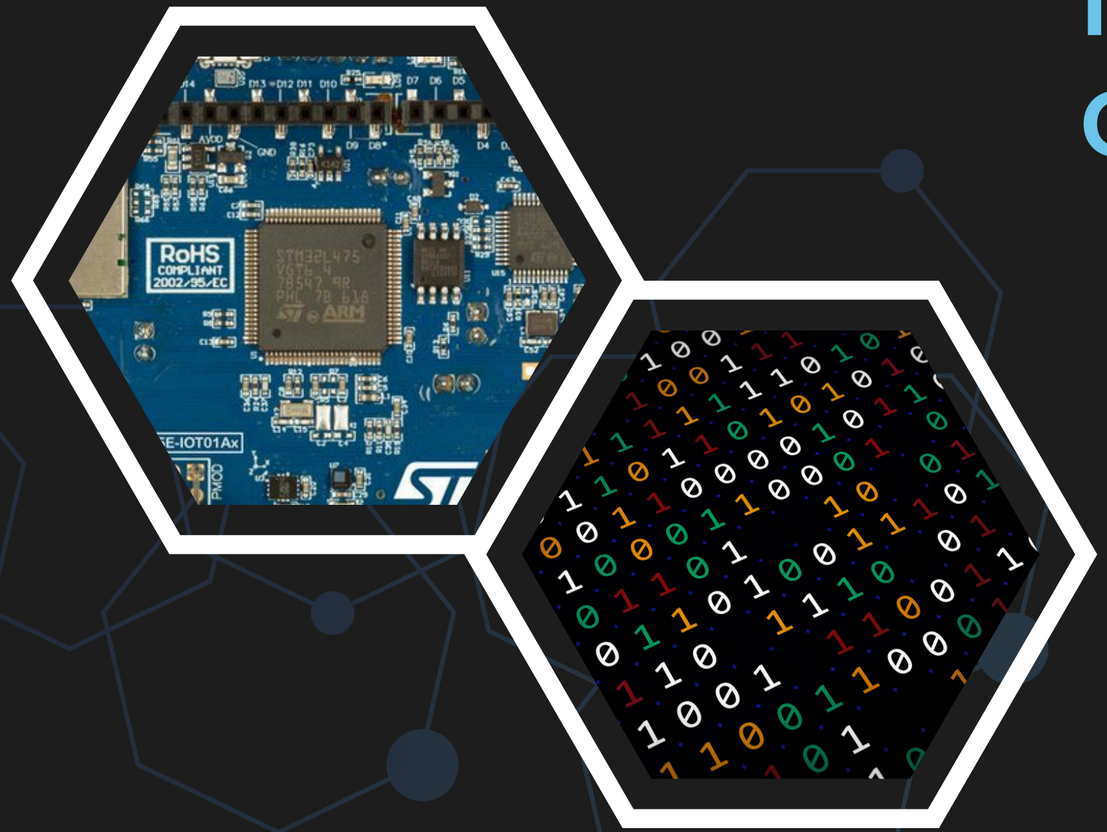
# Our original plan

- Given 3D accelerometer value, use AI model to determine the motion type

- B-L475E-IOT01A
- LSM6DSL sensor
- X-CUBE-AI
- STM32CubeIDE
- STM32Cube.AI
- Keras model

# However...

- The collection of data for our motion is quite difficult
- The performance is extremly poor...
- Slow determination, high delay
- Not accurate enough, often leads to wrong output
- Abandon the original plan and use other technique




# Motion Detection on STM32L4 IoTnode

## What We Use

- B-L475E-IOT01A
- BSP
- Detection Algorithm
- Mbed Studio
- FIR Lowpass Filter

# The Design Flow

**Process Number 01**

Collect Data

**Process Number 02**

Determine Body Motion

**Process Number 03**

Transmit Data

**Detail Process**

We collect the data of body motions using B-L475E-IOT01A, including moving left, right, up, down and stationary

**Detail Process**

We use an algorithm designed by ourselves to determine what type the body motion detected is

**Detail Process**

After determining the body motion type, send the type to the game server to perform the corresponding motion

# Collect Data

Collect accelerometer data and store the data in a buffer.

```
void process_data() {
    // Buffer to hold the last 20 samples
    int16_t buffer[BUFFER_SIZE][3] = {0};
    int bufferIndex = 0;

    // Variables to store standard deviation and amplitude difference
    float stm_x = 0, stm_y = 0, stm_z = 0;
    float stm_diff[3] = {0};
    float stm_all[3] = {0};

    while (true) {
        int16_t pDataXYZ[3] = {0};
        BSP_ACCELERO_AccGetXYZ(pDataXYZ);
```

# Collect Data

Calculate the amplitude of data and store the data in the buffer.

```
// Calculate amplitude
stm_all[0] = pDataXYZ[0];
stm_all[1] = pDataXYZ[1];
stm_all[2] = pDataXYZ[2];
```

```
// Store the current sample in the buffer
buffer[bufferIndex % BUFFER_SIZE][0] = pDataXYZ[0];
buffer[bufferIndex % BUFFER_SIZE][1] = pDataXYZ[1];
buffer[bufferIndex % BUFFER_SIZE][2] = pDataXYZ[2];
bufferIndex++;
```

# Collect Data

Calculate the average of data and standard deviation of data.

```
// Calculate standard deviation every 20 samples
if (bufferIndex >= BUFFER_SIZE) {
    float sumX = 0, sumY = 0, sumZ = 0;
    float meanX = 0, meanY = 0, meanZ = 0;
    float varX = 0, varY = 0, varZ = 0;

    for (int i = 0; i < BUFFER_SIZE; i++) {
        sumX += buffer[i][0];
        sumY += buffer[i][1];
        sumZ += buffer[i][2];
    }
```

```
meanX = sumX / BUFFER_SIZE;
meanY = sumY / BUFFER_SIZE;
meanZ = sumZ / BUFFER_SIZE;

for (int i = 0; i < BUFFER_SIZE; i++) {
    varX += pow(buffer[i][0] - meanX, 2);
    varY += pow(buffer[i][1] - meanY, 2);
    varZ += pow(buffer[i][2] - meanZ, 2);
}

stm_x = sqrt(varX / BUFFER_SIZE);
stm_y = sqrt(varY / BUFFER_SIZE);
stm_z = sqrt(varZ / BUFFER_SIZE);
```

# Collect Data

Send the data to motion_detection function.

```
// Motion detection
motion_detection(stm_x, stm_y, stm_z, stm_all[0], stm_all[1], stm_all[2]);
```

# Motion Detection Algorithm

The type of motion detected depends on the standard deviation and the amplitude.

```
void motion_detection(float stm_x, float stm_y, float stm_z, float stm_all_x, float stm_all_y, float stm_all_z) {
    if (fabs(stm_x - stm_y) < 250 && fabs(stm_y - stm_z) < 250 && fabs(stm_z - stm_x) < 250) {
        // printf("STATIONARY\n");
    } else {
        if (stm_x > stm_y && stm_x > stm_z) {
            if (t.read_ms() >= 500) {
                if (stm_all_x < 0) {
                    printf("LEFT\n");
                } else {
                    printf("RIGHT\n");
                }
                t.reset(); // Restart the timer
                t.start(); // Start the timer again after reset
            } else {
                // printf("STATIONARY\n");
            }
```

# Motion Detection Algorithm

The type of motion detected depends on the standard deviation and the amplitude.

```
        } else if (stm_z > stm_x && stm_z > stm_y) {
            if (t.read_ms() >= 500) {
                if (stm_all_z < 0) {
                    printf("DOWN\n");
                } else {
                    printf("UP\n");
                }
                t.reset(); // Restart the timer
                t.start(); // Start the timer again after reset
            } else {
                // printf("STATIONARY\n");
            }
        }
    }
}
```

# However...

- The performance is not good as we thought
- So we decide to use the FIR lowpass filter

# FIR Lowpass Filter

A FIR lowpass filter similar to the one in homework.

```
arm_fir_instance_f32 fir_x, fir_y, fir_z;
float firStateF32_x[FILTER_TAP_NUM + BUFFER_SIZE - 1];
float firStateF32_y[FILTER_TAP_NUM + BUFFER_SIZE - 1];
float firStateF32_z[FILTER_TAP_NUM + BUFFER_SIZE - 1];

// FIR filter coefficients
const float32_t firCoeffs32[FILTER_TAP_NUM] = {
    0.0018225238f, 0.0020318917f, 0.0023229396f, 0.0026937495f, 0.0031397797f,
    0.0036549973f, 0.0042319678f, 0.0048619938f, 0.0055352264f, 0.0062408831f,
    0.0069674629f, 0.0077029880f, 0.0084352682f, 0.0091521361f, 0.0098416830f,
    0.0104925806f, 0.0110933467f, 0.0116327752f, 0.0121003282f, 0.0124866274f,
    0.0127839390f, 0.0129852983f, 0.0130855638f, 0.0130797380f, 0.0129651520f,
    0.0127432694f, 0.0124126540f, 0.0119761570f, 0.0114372370f
};
```

# FIR Lowpass Filter

Functions for filtering.

```
void initFIRFilter() {
    arm_fir_init_f32(&fir_x, FILTER_TAP_NUM, (float32_t *)&firCoeffs32[0], &firStateF32_x[0], BUFFER_SIZE);
    arm_fir_init_f32(&fir_y, FILTER_TAP_NUM, (float32_t *)&firCoeffs32[0], &firStateF32_y[0], BUFFER_SIZE);
    arm_fir_init_f32(&fir_z, FILTER_TAP_NUM, (float32_t *)&firCoeffs32[0], &firStateF32_z[0], BUFFER_SIZE);
}
```

# FIR Lowpass Filter

Filter the data stored in the buffer.

```
// Store the current sample in the buffer
buffer[bufferIndex % BUFFER_SIZE][0] = pDataXYZ[0];
buffer[bufferIndex % BUFFER_SIZE][1] = pDataXYZ[1];
buffer[bufferIndex % BUFFER_SIZE][2] = pDataXYZ[2];
bufferIndex++;

float32_t temp_in_arr_x[BUFFER_SIZE];
float32_t temp_out_arr_x[BUFFER_SIZE];
for (int i = 0; i < BUFFER_SIZE; i++) {
    temp_in_arr_x[i] = buffer[i][0];
}
arm_fir_f32(&fir_x, temp_in_arr_x, temp_out_arr_x, BUFFER_SIZE);
```

# FIR Lowpass Filter

Filter the data stored in the buffer.

```
float32_t temp_in_arr_y[BUFFER_SIZE];
float32_t temp_out_arr_y[BUFFER_SIZE];
for (int i = 0; i < BUFFER_SIZE; i++) {
    temp_in_arr_y[i] = buffer[i][1];
}
arm_fir_f32(&fir_y, temp_in_arr_y, temp_out_arr_y, BUFFER_SIZE);

float32_t temp_in_arr_z[BUFFER_SIZE];
float32_t temp_out_arr_z[BUFFER_SIZE];
for (int i = 0; i < BUFFER_SIZE; i++) {
    temp_in_arr_z[i] = buffer[i][2];
}
arm_fir_f32(&fir_z, temp_in_arr_z, temp_out_arr_z, BUFFER_SIZE);
```

# FIR Lowpass Filter

Calculate the amplitude of data after filtering.

```
for (int i = 0; i < BUFFER_SIZE; i++) {
    stm_all[0] = temp_out_arr_x[bufferIndex % BUFFER_SIZE];
    stm_all[1] = temp_out_arr_y[bufferIndex % BUFFER_SIZE];
    stm_all[2] = temp_out_arr_z[bufferIndex % BUFFER_SIZE];
}
```

# FIR Lowpass Filter

Calculate the average of data and standard deviation of data after filtering.

```
// Calculate standard deviation every 20 samples
if (bufferIndex >= BUFFER_SIZE) {
    float sumX = 0, sumY = 0, sumZ = 0;
    float meanX = 0, meanY = 0, meanZ = 0;
    float varX = 0, varY = 0, varZ = 0;

    for (int i = 0; i < BUFFER_SIZE; i++) {
        sumX += temp_out_arr_x[i];
        sumY += temp_out_arr_y[i];
        sumZ += temp_out_arr_z[i];
    }
```

```
for (int i = 0; i < BUFFER_SIZE; i++) {
    varX += pow(temp_out_arr_x[i] - meanX, 2);
    varY += pow(temp_out_arr_y[i] - meanY, 2);
    varZ += pow(temp_out_arr_z[i] - meanZ, 2);
}

stm_x = sqrt(varX / BUFFER_SIZE);
stm_y = sqrt(varY / BUFFER_SIZE);
stm_z = sqrt(varZ / BUFFER_SIZE);
```

# FIR Lowpass Filter

The motion detection algorithm is similar, with some constants changed.

```
void motion_detection(float stm_x, float stm_y, float stm_z, float stm_all_x, float stm_all_y, float stm_all_z) {
    if (fabs(stm_x - stm_y) < 5 && fabs(stm_y - stm_z) < 5 && fabs(stm_z - stm_x) < 5) {
        // printf("STATIONARY\n");
    } else {
        if (stm_x > stm_y && stm_x > stm_z) {
            if (t.read_ms() >= 300) {
                if (stm_all_x < 0) {
                    printf("LEFT\n");
                    _wifi.send_motion("LEFT");
                } else {
                    printf("RIGHT\n");
                    _wifi.send_motion("RIGHT");
                }
                t.reset(); // Restart the timer
                t.start(); // Start the timer again after reset
            } else {
                // printf("STATIONARY\n");
            }
```

# FIR Lowpass Filter

The motion detection algorithm is similar, with some constants changed.

```
        } else if (stm_z > stm_x && stm_z > stm_y) {
            if (t.read_ms() >= 300) {
                if (stm_all_z < 200) {
                    printf("DOWN\n");
                    _wifi.send_motion("DOWN");
                } else {
                    printf("UP\n");
                    _wifi.send_motion("UP");
                }
                t.reset(); // Restart the timer
                t.start(); // Start the timer again after reset
            } else {
                // printf("STATIONARY\n");
            }
        }
    }
}
```

# Tried to Calibrate for Tilted Angle

- Pitch and roll calculation through gravity ➡ accurate but not suite for our applications
- Integrating angular velocity through gyroscopes ➡ not accurate even with calibration

# Transmit Data via Wifi

Using module define by ourself to easily sent data thru Wifi

```
private:
    ISM43362Interface _wifi;
    TCPSocket socket;
    SocketAddress address;
    nsapi_error_t ret;

    //buffer for motion sending
    char buffer[1024];
```

```
public:
    my_wifi_sender() : _wifi(false){}
    void connect_wifi();
    void connect_host();
    void send_motion(const char* motion);
```

# Game Design via Ursina

Ursina is  a game engine designed for creating 3D games and applications base on Python.

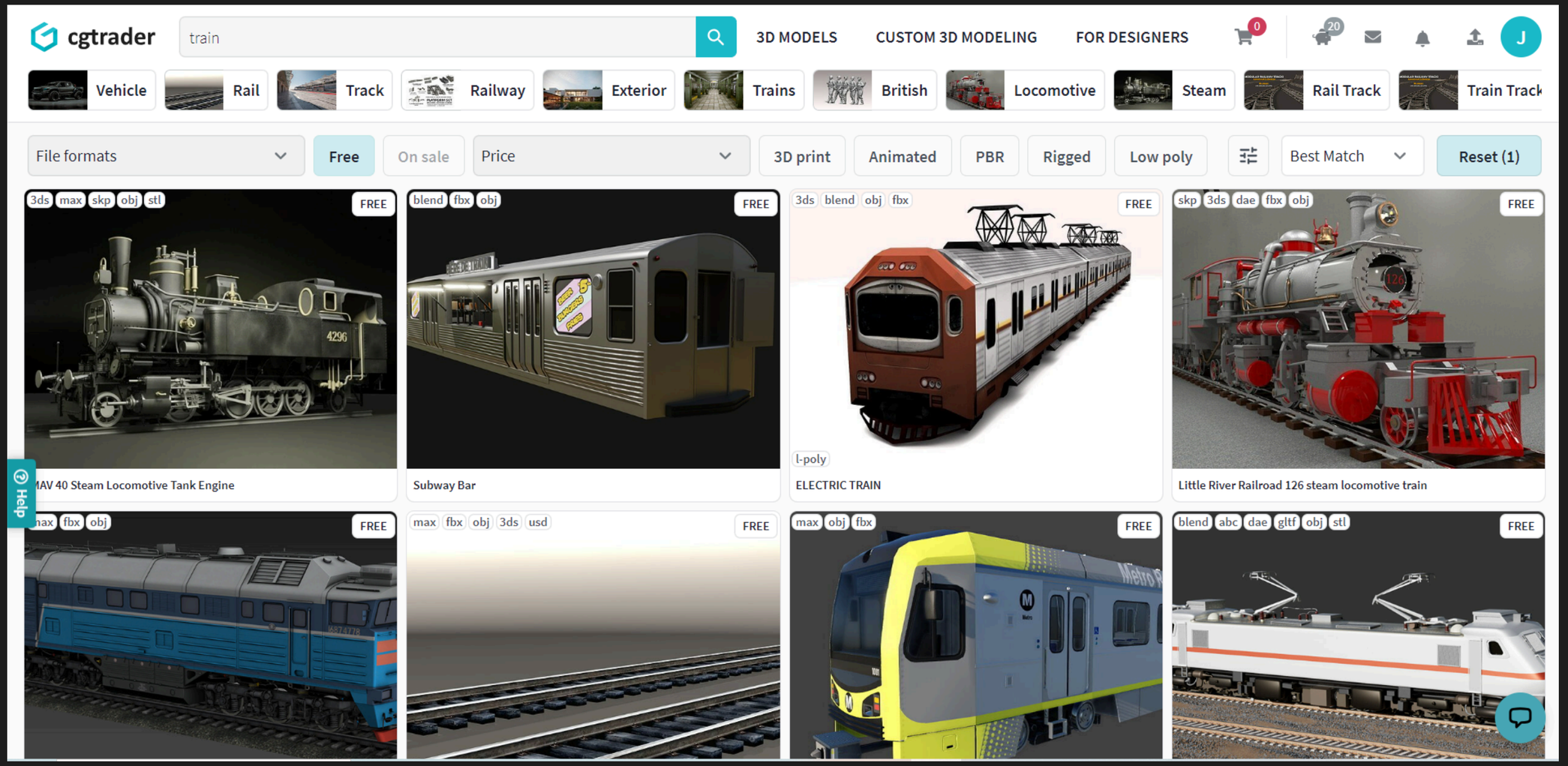
```python
from ursina import *

def update():
    train.z -= 1

app = Ursina()

train = Entity(model = "assets/train.obj", texture = "assets/sky.jpg")

app.run()
```




https://www.cgtrader.com/free-3d-models

# Host Connect to Client via Wi-Fi

```python
from ursina import *
import socket
import json

def update():
    data = conn.recv(1024).decode('utf-8')
    obj = json.loads(data)

app = Ursina()

HOST = '172.20.10.4'  # IP address
PORT = 8000  # Port to listen on (use ports > 1023)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
conn, addr = s.accept()

app.run()
```

**Waiting to receive data causing screen delay**

# Host Connect to Client via Wi-Fi

```python
from ursina import *
import socket
import json
import threading

def receive_data():
    while running:
        try:
            data = conn.recv(1024).decode('utf-8')
        except socket.error:
            running = False
            break

def update():
    if data:
        data = conn.recv(1024).decode('utf-8')
        obj = json.loads(data)
```

```python
app = Ursina()

HOST = '172.20.10.4'  # IP address
PORT = 8000  # Port to listen on (use ports > 1023)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
conn, addr = s.accept()

threading.Thread(target=receive_data).start()

app.run()
```

# Final Result

# Future prospect

- Refined our motion detection algorithm
- Find ways to handle tilt angle
- Add more complicated motions by the assistance of button or BLE tags

# Reference

https://github.com/NTUEE-ESLab/2021Spring-Reality-Motion-Game