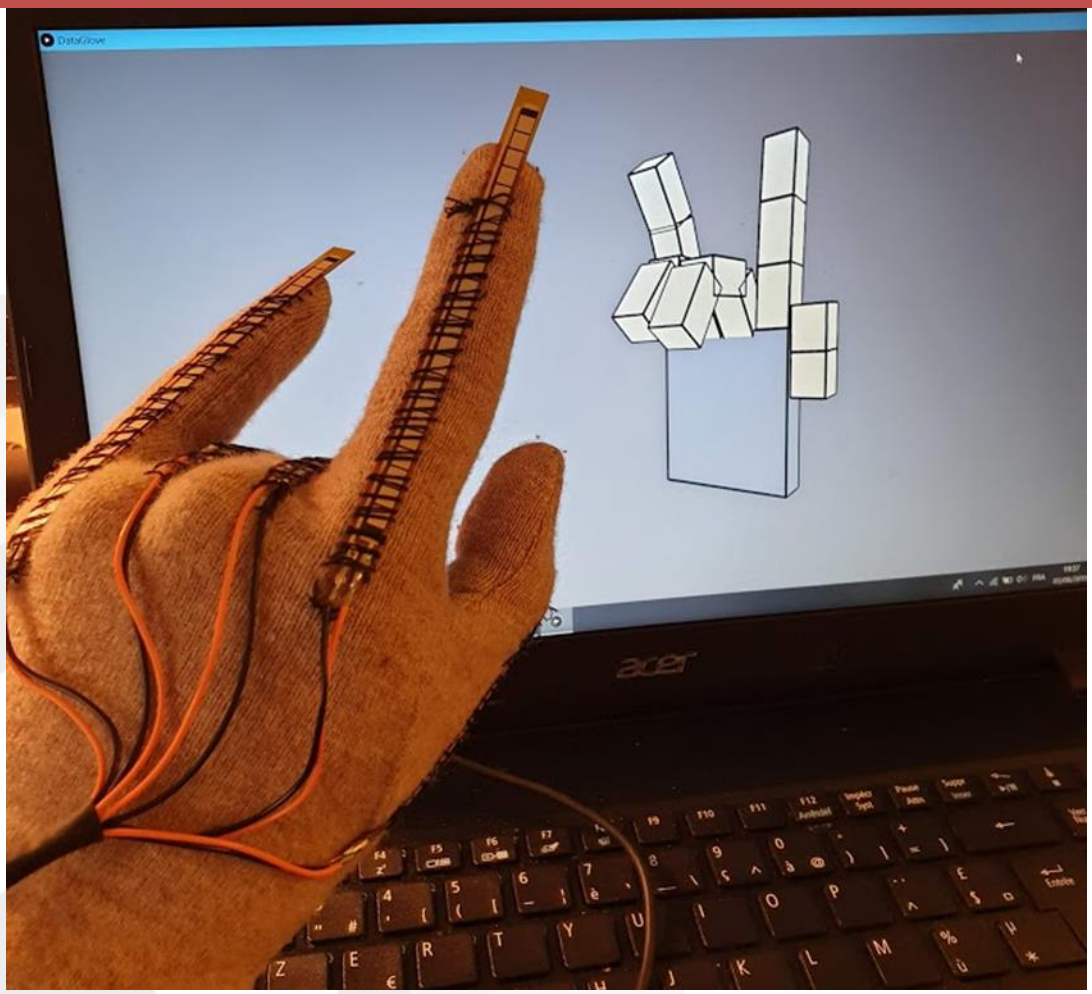
télécom saint-étienne
école d'ingénieurs nouvelles technologies

Rapport Projet d'Application Electronique

Florian VUITON, Julien GIOVINAZZO

Dataglove



École associée INSTITUT Mines-Télécom

UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Table des matières

1. Introduction

Le projet que nous avons choisis de réaliser est un "Datagloves à flex sensor et transmission sans fil". Il consiste à concevoir un gant intelligent capable de transmettre la position des doigts à un PC. Ce PC doit ensuite afficher une main en 3D qui reflète la forme de la vraie main. Nous avons choisi d'aborder ce sujet car il nous semblait plus concret que les autres projets et l'approche de logiciel tel que Processing nous paraissait intéressante.

Tout au long du projet, nous avons fait beaucoup de recherche afin d'essayé de mettre en place une application adaptée au cahier des charges et à nos envies.

2. Conception (Julien Giovinazzo)

I. Cahier des charges

Notre vision de ce projet est de mettre en application nos connaissances d'électronique et les notions du module APP du semestre 5 pour réaliser un prototype réel et fonctionnel. Notre projet nous demande de créer un gant équipé de flex sensor permettant de mesurer la position des doigts et de la transmettre par liaison série radio à un PC. Une datavisualisation en 3D permettra de rendre compte de la position des doigts dans l'espace.

II. Choix d'équipement

Pour réaliser le DataGlove, plusieurs détails nous ont été imposés. Tout d'abord, il faut choisir comment le gant communiquera avec le PC. Le matériel, la carte Arduino ESP32 Thing, et le type de liaison (liaison série hertzienne) sont imposées. Cela nous laisse le choix entre une liaison Wifi ou une liaison Bluetooth. Nous avons décidé d'utiliser le Bluetooth. La mise en place et l'utilisation du Bluetooth est bien plus rapide est efficace que celle du Wifi. De plus les avantages de la wifi (communications à grandes distances) ne sont pas nécessaires car le gant sera toujours proche (quelques mètres) du PC.

Ensuite, cinq flex sensors nous ont été fournis. Ce sont des résistances flexibles qui changent d'impédance lorsqu'elles sont pliées.

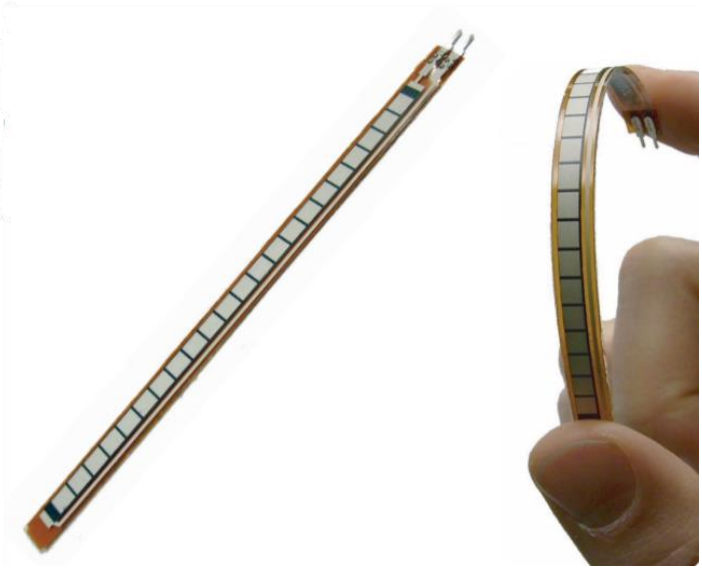


Figure 1 : Flex Sensor

A plat, un flex sensor a une résistance de 10KΩ. Cette valeur augmente linéairement jusqu'à 50kΩ quand la bande est pliée à 180°. Ces valeurs sont des moyennes, et varient beaucoup. A plat, la résistance peut être entre 7kΩ et 13kΩ.

Chaque bande mesure 12cm de long. Il y a aussi un unique flex sensor de 7cm de longueur que nous utilisons pour le pouce. Son fonctionnement est le même, mais son impédance est de 35kΩ à plat et de 150kΩ à 180°.

Comme dit précédemment, nous devons utiliser une carte Arduino ESP32 Thing. Cette carte est un system-on-chip (SoC) produit par Sparkfun à faible cout et fable puissance. Elle intègre déjà une puce pour la communication Bluetooth et une pour la communication Wifi.

Pour finir, nous devions créer notre visualisation 3D en utilisant un nouveau langage : Processing. Processing est un langage orienté-objet conçu pour faciliter la programmation graphique. Nous n'avions j'avais utilisé un tel langage, cependant, la syntaxe étant assez proche du C++ / python et tout était très bien documenté en ligne. Il nous a fallu peu de temps avant d'être un minimum à l'aise avec.

III. Calculs et mesures

Pour que la carte Arduino puisse lire la valeur de la résistance des flex sensors, il faut réaliser des ponts diviseurs de tension.

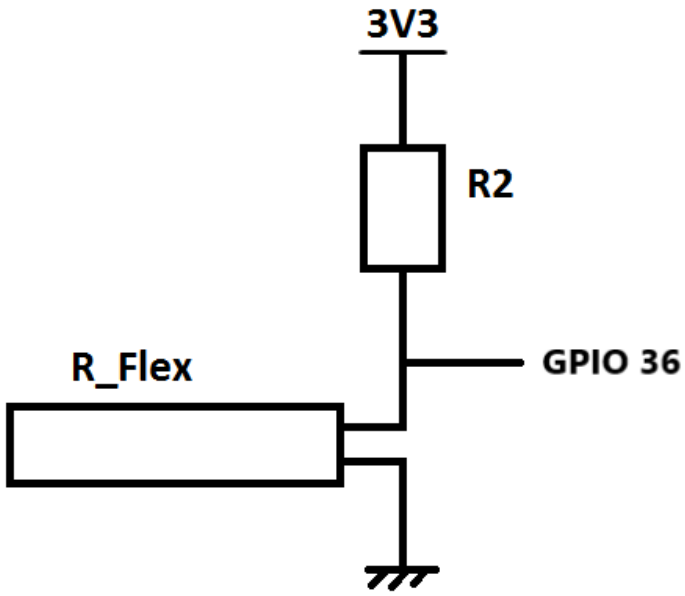


Figure 2 : Schéma de câblage (Pont diviseur de tension)

Sur le schéma ci-dessus, nous pouvons lire une tension sur le GPIO 36 en fonction de la valeur de R_Flex. Si nous prenons R2 = 12kΩ, les tensions lus irons de 1.5V à 2.6V pour un flex sensor de 12cm.

Le port converti la tension lue en valeur entière codée sur 12-bit. C'est-à-dire qu'une tension de 0.0V donne une valeur de 0 et une tension de 3.3V donne 4095. Dans notre exemple, nous lirons des valeurs entre 1861 et 3226.

Pour notre prototype final, nous avons utilisé cinq ponts diviseurs, un pour chaque doigt. Pour le pouce, l'impédance est de 71.5kΩ et les autres doigts 23.7kΩ. Ces valeurs de résistances sont communes aux deux groupes, et ont été choisies afin d'avoir la plus de variations sur une flexion complète. C'est-à-dire d'avoir la meilleure résolution possible. Comme nous verrons plus tard avec notre algorithme de calibration, ces valeurs de résistances importent peu pour notre visualisation (même si avoir une résolution maximale est préférable).

IV. Schémas fonctionnels

a. Niveau 1

Figure 3 : Schéma fonctionnel de niveau 1

b. Niveau 2

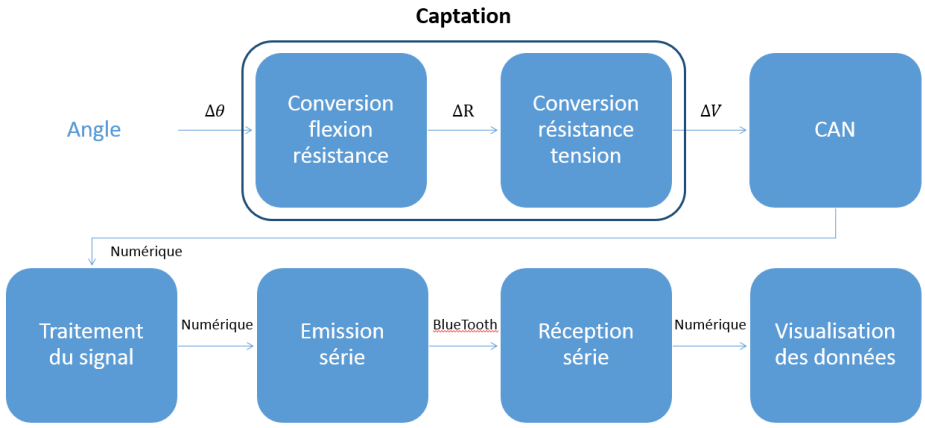


Figure 4 : Schéma fonctionnel de niveau 2

3. Réalisation (Florian Vuiton)

I. Prototypage

a. Gant

Nous avons décidé de coudre les flex sensor sur un gant et de les laisser coulisser le long des doigts afin de ne pas abimer les résistances quand ils sont pliés. Nous avons aussi soudé des câbles entre les flex sensors et la plaque d'essai.

Figure 5 : Prototype du gant

b. Plaque d'essai

De plus, le groupe travaillant sur le même projet que nous à réaliser le câblage des ponts diviseurs de tensions sur une plaque d'essai raccordés à la carte ESP32.

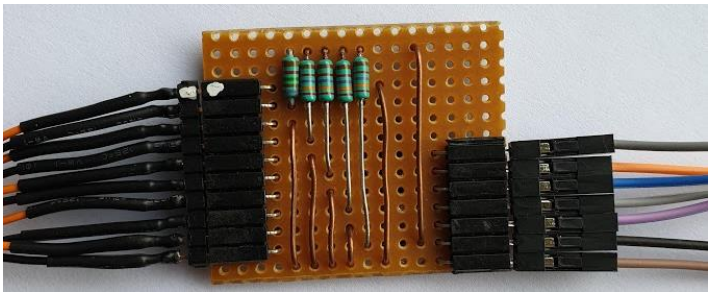


Figure 6 : Prototype de la plaque d'essai

II. Arduino

Arduino est un fabricant de carte électronique libre sur lesquelles se trouve un microcontrôleur. Arduino propose aussi une interface permettant de compiler et téléverser les programmes via l'interface en ligne de commande. Il permet de transférer les programmes sur la carte via une liaison série (RS-232, Bluetooth ou USB). Le langage de programmation utilisé est le C++.

Afin de lire la valeur renvoyée par le flex sensor, nous avons choisis d'utiliser les ports GPIO d'entrée 36, 37 ,38, 39 et 32 de l'ESP32.

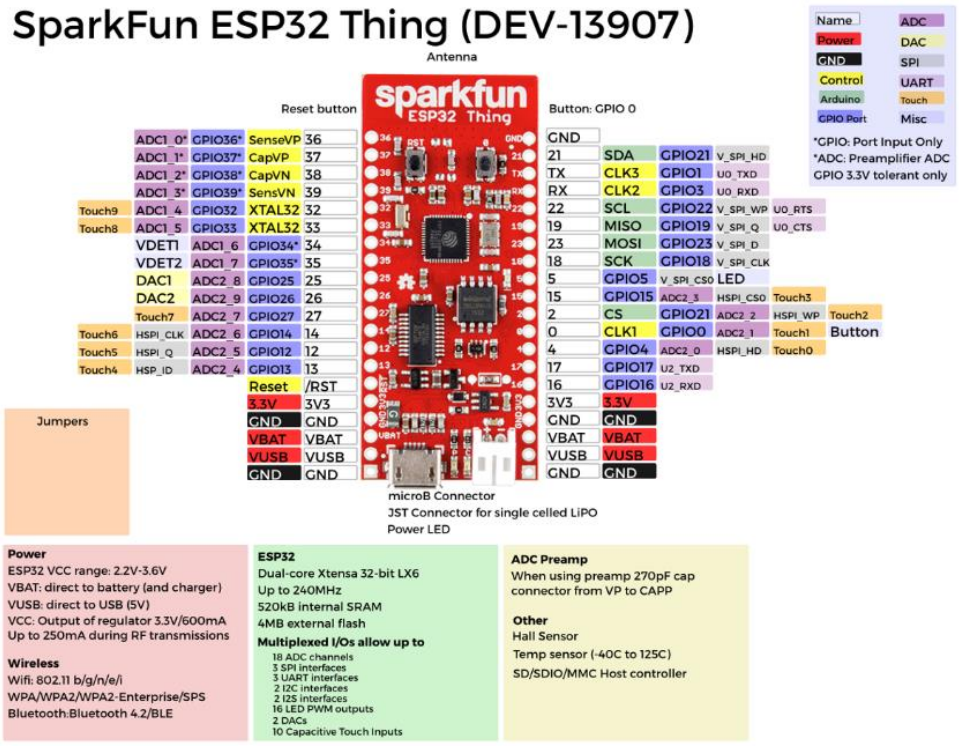


Figure 7 : Pins Mapping

Toutes les 200 millisecondes, nous lisons 100 valeurs sur chacun des 5 pins ; ensuite nous moyennons ces valeurs afin d'éviter les artéfacts c'est-à-dire des valeurs parasites. Sans ce moyenneur, il y aurait des tremblements au niveau de l'affichage dû à la lecture peu précise des ports et aux micros mouvements de la main. Ensuite, nous créons une chaine de caractères qui contient les valeurs moyennées des 5 doigts séparés par un espace que nous envoyons par Bluetooth.

III. Processing

Processing est un environnement de développement libre adapté à la création graphique. Processing a été créé de manière à simplifier le plus possible la mise en œuvre des programmes.

Les deux fonctions prédéfinis `setup()` et `draw()` sont très importantes dans l'utilisation de Processing.

- `setup()` : elle est exécutée qu'une seule fois au lancement du programme et nous permet d'indiquer la taille de la fenêtre graphique ainsi que le port Bluetooth d'entrée des données.
- `draw()` : elle est exécutée 60 fois par seconde par défaut. C'est la boucle principale du programme qui permet de gérer la mise à jour des données et la partie graphique.

Nous avons utilisé une troisième fonction prédéfinie nommée `keyPressed()` qui s'exécute à chaque appuie sur une touche. Elle permet de lancer les différentes fonctions de notre programme : la calibration, le jeu et une partie.

Afin de réaliser ce projet, nous avons créé 3 classes :

- Finger : Une entité qui représente un doigt par sa position en x et en y, la valeur du pli et un booléen pour savoir si c'est le pouce ou non. Elle gère aussi son propre affichage.
- Hand : Gestionnaire des 5 « Finger » pour gérer la main dans sa globalité pour permettre la calibration.
- Game : Gère le fonctionnement du jeu Pierre-Feuille-Ciseaux

Nous reviendrons sur chacune de ces 3 classes plus tard.

Dans un premier temps, dans la fonction `draw()` ci-dessous, nous lisons la chaine de caractères reçus sur le port Bluetooth. Ensuite, nous extrayons chaque valeur dans un tableau pour pouvoir caractériser la position de chaque doigt séparément. Enfin, nous lions chaque valeur à chaque objet « Finger » et nous appelons nos différents modules.

```
void draw() {

  // Récupérer les données du port BT
  if ( myPort.available() > 0) {  // If data is available,
    String serial_in = trim(myPort.readStringUntil('\n'));

    if (serial_in == null) {
      return;  // Quitte la fonction
    }

    // Décomposition de l'entree
    int debut = 0;
    int fin = 0;

    int indice_tab = 0;

    while ( indice_tab < 5 ) {
      debut = fin;
      fin = serial_in.indexOf(' ', debut + 1);
      if (fin != -1)
        tab[indice_tab] = int(trim(serial_in.substring(debut, fin)));
      else
        tab[indice_tab] = int(trim(serial_in.substring(debut)));

      indice_tab++;
    }
  }

  // Lier la valeur lu dans COM8 avec la hauteur des doigts
  hand_player.set_finger_heights(tab[0], tab[1], tab[2], tab[3], tab[4], true);
```

Figure 8 : Fonction Draw

a. Classe Finger

Comme décris précédemment, cette classe permet la description des doigts en fonction de leur position x et y, l'angle du pli, ainsi que sa forme (pouce ou non).

Chaque doigt comporte 3 phalanges. Il s'agit, en partant de la paume, de la phalange, la phalangine et de la phalangette. Nous avons, donc, utilisé 3 `box(w, h, d)` (Cube qui prend en paramètre la longueur de chaque dimension) pour créer un doigt.

Dans un premier temps et grâce aux valeurs de calibration, nous convertissons la valeur reçue par Bluetooth en valeur comprise entre 0.0 et 1.0 : `m_Hfactor`.

```
void set_height(float value) {
  m_Hfactor = (value - get_value_flat()) / (float)(get_value_folded() - get_value_flat());
}
```

Figure 9 : Fonction de conversion de pli du doigt entre 0 et 1

Les fonctions prédéfinis `translate(x, y, z)` et `rotate(angle)` permettent de se déplacer dans le repère de Processing. A chaque translate ou rotate d'une certaine valeur, nous appliquons la même fonction avec la valeur inverse afin de toujours revenir au centre de l'écran.

Pour afficher un doigt, nous nous translatons d'une demie phalange, et nous faisons une rotation plus ou moins grande en fonction de la valeur de `m_Hfactor`. A chaque rotation, nous créons une box. Ceci est répété pour chaque phalange.

b. Classe Hand

Très tôt dans la réalisation du projet, nous avons remarqué que les impédances flex sensors varient beaucoup entre eux. A plat, on peut trouver une différence de plus 1kΩ. Pour y remédier, nous avons implémenté un algorithme de calibration automatique.

Lorsqu'on lance une calibration, l'utilisateur dispose d'une période de 3 secondes pour ouvrir et fermer sa main. Pendant cette période, le script Processing générer une relation entre les valeurs lues et la position des doigts.

Grace à notre calibration, notre gant est indépendant des flex sensors utilisés et des valeurs des résistances utilisées pour les ponts diviseurs. Ceci nous permet de très facilement changer de composants sans toucher au code si la situation se présente.

c. Classe Game

Afin de pouvoir exploiter de manière ludique la main virtuelle, nous avons développé un grand jeu de réflexion : Pierre-Feuille-Ciseaux (Shifoumi). Pour ce faire, nous avons créé une deuxième main (avec notre classe Hand) qui peut prendre 3 positions fixe.

```
float [] rock = {1, 1, 1, 1, 1};
float [] paper = {0, 0, 0, 0, 0};
float [] scissors = {1, 1, 0, 0, 1};
```

Figure 10 : Paramètre de position de la main de l'ordinateur

Lorsque l'utilisateur appuie sur espace, un compte à rebours apparait. Au bout du décompte, la position des doigts du joueur est lue par l'intermédiaire de la valeur de `m_Hfactor`. Ensuite, par de simples comparaisons et avec une valeur de seuil, nous déduisons le choix du joueur. Nous précisons que la position du pouce n'a aucune incidence sur le résultat.

```
void read_player_move() {
  // Trouver quelle formation le joueur joue

  float threshold = 0.4;  // Seuil a partir duquel un doight est considéré 'ouvert' ou 'fermé'
  float fh1 = hand_player.get_finger(1).get_Hfactor();  // Index
  float fh2 = hand_player.get_finger(2).get_Hfactor();  // Majeur
  float fh3 = hand_player.get_finger(3).get_Hfactor();  // Doigt de la bague
  float fh4 = hand_player.get_finger(4).get_Hfactor();  // Petit doigt
  // On ignore le pouce car il n'est pas important
  if (fh1 > threshold && fh2 > threshold && fh3 > threshold && fh4 > threshold) {  // Pierre
    player_move = 1;
    move_player_str = "Pierre";
  } else if (fh1 < threshold && fh2 < threshold && fh3 < threshold && fh4 < threshold) {  // Feuille
    player_move = 2;
    move_player_str = "Feuille";
  } else if (fh1 > threshold && fh2 > threshold && fh3 < threshold && fh4 < threshold) {  // Sciseaux
    player_move = 3;
    move_player_str = "Ciseaux";
  } else {
    player_move = 4;
    move_player_str = "Position non reconnue";
  }
}
```

Figure 11 : Fonction de lecture du choix du joueur

Au même moment, nous tirons un chiffre aléatoire entre 1 et 3 pour faire jouer l'ordinateur. Enfin, nous déduisons des deux positions le vainqueur de la manche et nous affichons les scores grâce à la fonction `text(chaine de caractère)`.

IV. Simulations

```
1707 1796 1675 1655 1328
1707 1797 1674 1653 1328
1707 1797 1675 1654 1328
1708 1799 1676 1654 1328
1706 1796 1674 1653 1328
```

Figure 12 : Exemple de trame envoyé à Processing par la carte ESP32

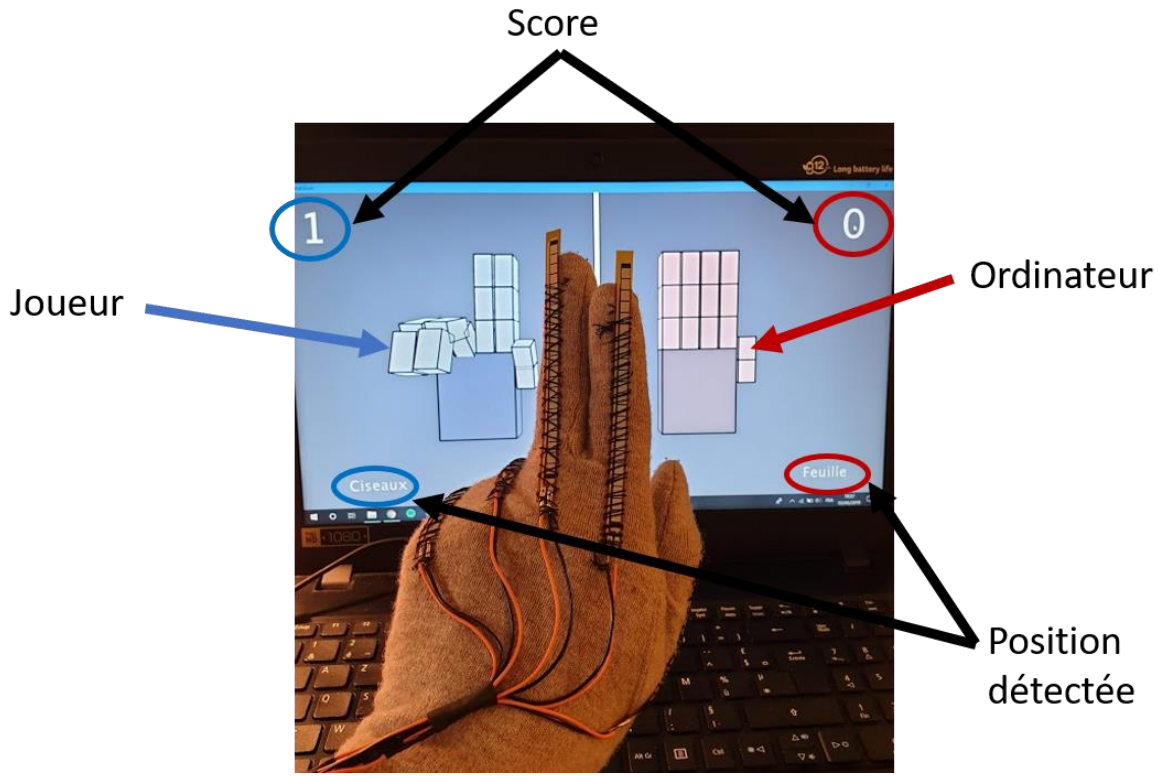


Figure 13 : Simulation du jeu

4. Conclusion

Au cours de ce projet nous avons rencontré différents problèmes :

- Travail avec une autre équipe

Due à une limite de matériel, nous avons dû coopérer avec le groupe du TDB composé de C. Luton, E. Yammine et A. Meyer pour réaliser le gant. Ça nous a demandé plus d'efforts en termes d'organisation et de planification car il fallait se distribuer les taches et se partager le gant. A la fin du compte, tout s'est très bien organiser, et nous avons pu faire un gant commun malgré nos approches différentes au problème.

- Affichage en 3D sous Processing

Nous avons dû nous adapter au système de repère 3D relatif de Processing et notamment aux fonctions translate et rotate. En effet, à chaque déplacement dans le repère, il nous fallait être capable de revenir au centre afin de ne pas nous « perdre » dans l'espace 3D. Chacun d'entre nous a déjà eu un minimum d'expérience avec l'affichage 2D ; mais l'ajout de la troisième dimension avec un nouveau langage a été difficile au début. Avec le temps, nous avons bien été habité à ce système, nous permettant de facilement manipuler les objets dans l'espace.

Pour conclure, la réalisation de notre Dataglove s'est bien déroulée. Nous avons pu respecter la totalité du cahier des charger et nous avons même pu aller au-delà en implémentant un jeu simple. Le jeu de Pierre-Feuille-Ciseaux permet d'aller un peu plus loin qu'une simple démonstration. Notre application est performante.

5. Notice d'utilisation

Lors du lancement de l'application, il suffit d'appuyer sur la touche « c » pour pouvoir lancer la calibration. Ensuite, l'appuie sur la touche « j » permet de lancer le jeu. Pour lancer une partie, il suffit d'appuyer sur la touche « espace ». En rappuyant sur « j », on revient sur la démo.

6. Annexes

```
#include "BluetoothSerial.h"
BluetoothSerial SerialBT;
int pin36 = 36; // A4
int pin37 = 37; // A5
int pin38 = 38; // A6
int pin39 = 39; // A7
int pin32 = 32; // A0

int quantite_moy = 100;  // Plus c'est proche de 0, plus c'est rapide

void setup() {
 Serial.begin(9600);
 SerialBT.begin("ESP32");
}
void loop() {
  delay(200);
  int somme36 = 0;
  int somme37 = 0;
  int somme38 = 0;
  int somme39 = 0;
  int somme32 = 0;

  for (int i = 0; i < quantite_moy; i++) {
    int tension36 = analogRead(pin36);
    int tension37 = analogRead(pin37);
    int tension38 = analogRead(pin38);
    int tension39 = analogRead(pin39);
    int tension32 = analogRead(pin32);
    somme36 += tension36;
    somme37 += tension37;
    somme38 += tension38;
    somme39 += tension39;
    somme32 += tension32;
  }

 SerialBT.print(somme36 / quantite_moy);
 SerialBT.print(' ');
 SerialBT.print(somme37 / quantite_moy);
 SerialBT.print(' ');
 SerialBT.print(somme38 / quantite_moy);
 SerialBT.print(' ');
 SerialBT.print(somme39 / quantite_moy);
 SerialBT.print(' ');
 SerialBT.print(somme32 / quantite_moy);
 SerialBT.print('\n');
}
```

Figure 14 : Programme Arduino

```
void rotationX (boolean set_rotation, int segment) {
  if (set_rotation) {
    if (segment == 1) {
      rotateX((PI/3)*m_Hfactor);
    }
    if (segment == 2) {
      rotateX((PI/2)*m_Hfactor);
    }
    if (segment == 3) {
      rotateX(PI / 2*m_Hfactor);
    }
  }
  else {
    if (segment == 1) {
      rotateX((-PI/3)*m_Hfactor);
    }
    if (segment == 2) {
      rotateX((-PI/2)*m_Hfactor);
    }
    if (segment == 3) {
      rotateX(-PI / 2*m_Hfactor);
    }
  }
}
```

```
void translateSegment() {

    translate(m_pos_x, m_pos_y);
    rotationX(false, 1);
    translate(0, -get_height()/6);

    box(FINGER_W, get_height()/3, FINGER_D);

    translate(0, -get_height()/6);
    rotationX(false, 2);
    translate(0, -get_height()/6);

    box(FINGER_W, get_height()/3, FINGER_D);

    if (!m_is_thumb) {
      translate(0, -get_height()/6);
      rotationX(false,3);
      translate(0, -get_height()/6);

      box(FINGER_W, get_height()/3, FINGER_D);

      translate(0, get_height()/6);
      rotationX(true, 3);
      translate(0, get_height()/6);
    }
    translate(0, get_height()/6);
    rotationX(true, 2);
    translate(0, get_height()/6);

    translate(0, get_height()/6);
    rotationX(true, 1);
    translate(-m_pos_x, -m_pos_y);
}
```

Figure 15 : Algorithme d'affichage de la main

```
void draw_calibrate() {
  background(200);

  camera();  // Camera par defaut

  fill(0);  // Texte noir
  textSize(50);
  text("Calibration en cours...", width/2, height/2);


  if (tab[0] > f1.get_value_folded())
    f1.set_value_folded(tab[0]);
  else if (tab[0] < f1.get_value_flat())
    f1.set_value_flat(tab[0]);

  if (tab[1] > f2.get_value_folded())
    f2.set_value_folded(tab[1]);
  else if (tab[1] < f2.get_value_flat())
    f2.set_value_flat(tab[1]);

  if (tab[2] > f3.get_value_folded())
    f3.set_value_folded(tab[2]);
  else if (tab[2] < f3.get_value_flat())
    f3.set_value_flat(tab[2]);

  if (tab[3] > f4.get_value_folded())
    f4.set_value_folded(tab[3]);
  else if (tab[3] < f4.get_value_flat())
    f4.set_value_flat(tab[3]);

  if (tab[4] > f5.get_value_folded())
    f5.set_value_folded(tab[4]);
  else if (tab[4] < f5.get_value_flat())
    f5.set_value_flat(tab[4]);
```

Figure 16 : Algorithme calibration

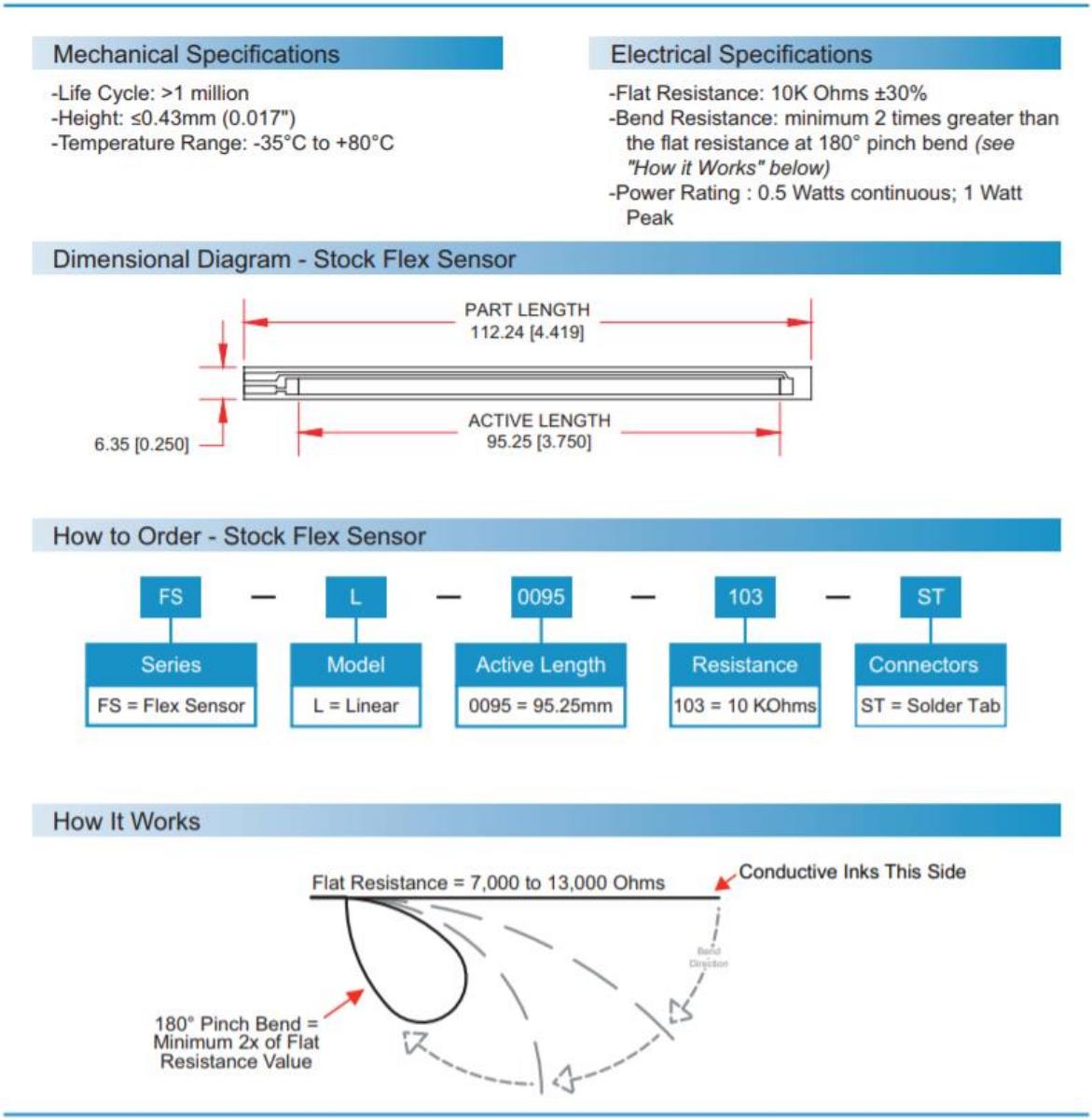


Figure 17 : Datasheet Flex Sensor