Rapport de soutenance 3 - Semestre 4 - EPITA

Clerc Killian (Chef de Projet)
Bellot Baptiste
Bruneteau Alexis
Genillon Paul

Mai 2022

Table des Matières

I. Introduction

Dans le cadre du projet de programmation du semestre 4 de l'EPITA, nous avons choisi de réaliser un projet de type "SLAM" c'est-à-dire "Simultaneous localization and mapping". Le nom de notre projet vient simplement de la traduction du mot "SLAM" en anglais qui signifie "CLAQUER" en français.

L'objectif du projet est de modéliser l'environnement **statique** en trois dimensions d'un robot grâce à plusieurs capteurs de distance. Le robot pourra effectuer différents mouvements qui lui permettront d'établir la modélisation de l'environnement dans lequel il se trouve.Le logiciel pourra sauvegarder la modélisation afin de pouvoir la recharger dans le futur si besoin.

Nous avions découpé les tâches de façon à ce que chacun d'entre nous puisse travailler sur quelque chose qui lui plaisait et qui lui faisait envie. Notre but était avant tout de prendre du plaisir sur ce projet, et nous l'avons tous adoré. Il nous a énormément apporté en termes de compétences, notamment sur la conception d'un robot que seul un membre du groupe connaissait. Ce projet nous a également permis d'améliorer nos compétences en matière de programmation. La répartition des tâches a été faites de façon homogène de manière à ce que nous ayons chacun une quantité de travail similaire entre chaque soutenance.

Nous avons pris énormément de plaisir à travailler dans ce groupe, l'ambiance de travail était très bonne. Nous sommes fiers d'avoir mené cet ambitieux projet et de pouvoir le présenter dans son état actuel.

Cet ultime rapport de soutenance vous présentera l'ensemble du travail effectué ces derniers mois, un travail dont nous ne sommes pas peu fiers.

II. Rappel de la répartition des tâches

Planning de réalisation du projet :

Tâches	Soutenance 1	Soutenance 2	Soutenance 3
Site Web	50%	75%	100%
Données capteur	10%	50%	100%
Nuage de points	0%	50%	100%
OpenGL	25%	75%	100%
GTK	20%	50%	100%
Voxel	10%	50%	100%
Sauvegarde	50%	100%	-
Plus court chemin	25%	75%	100%

Répartition des tâches :

Tâches	Killian	Alexis	Baptiste	Paul
Site Web				X
Données capteur		X		
Nuage de points			X	
OpenGL			X	
GTK				X
Voxel	X			
Sauvegarde			X	
Plus court chemin				X

III. Explication du travail effectué

I. ESP32

I - Arduino

Pour le robot, nous utilisons la librairie Arduino, qui permet de facilement contrôler nos éléments électroniques avec notre ESP32.

Pour le déplacement des roues du robot, nous avons opté pour une rotation de la roue pas à pas. Pour cela nous avons utilisé la librairie AccelStepper de l'environnement Arduino. Le Stepper que nous utilisons est un Nema 17, et d'après la fiche technique, un step correspond à 1.8° de rotation. La roue que nous avons créée avec une imprimante 3D a un diamètre de 80mm soit un rayon de 40mm.

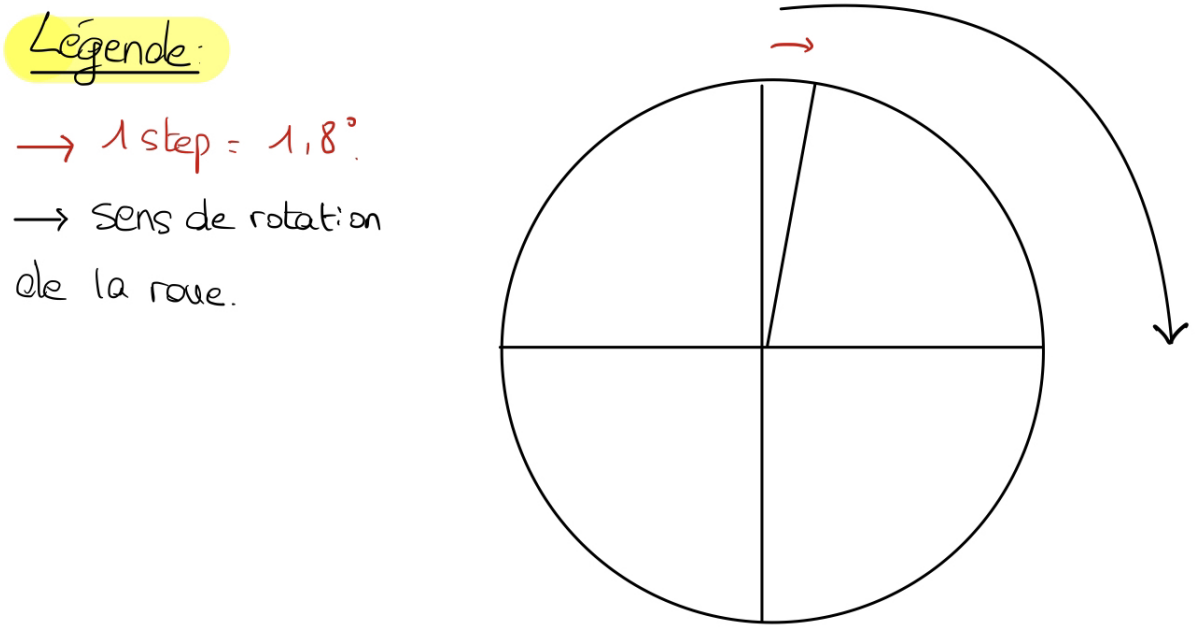

Figure 1: Schéma de rotation d'une roue pas à pas grâce à un stepper.

Nous créons un MultiStepper auquel nous ajoutons 2 AccelStepper qui correspondent aux deux roues du robot (gauche et droite). Lorsque nous souhaitons parcourir une certaine distance en millimètres en ligne droite, nous avons dû convertir le nombre de millimètres en step pour parcourir la bonne distance.

On a :

$$\text{step} = \frac{X*360}{2\pi R*1.8}$$

Où X correspond à la distance en millimètres et 1.8 correspond à l'angle correspond à un step du Nema 17. Pour déterminer le mouvement à réaliser, nous utilisons la fonction moveto de la librairie MultiStepper.h à laquelle nous donnons une distance qui correspond à la position actuelle du robot à laquelle on ajoute le step calculé précédemment. Nous utilisons la même formule pour le calcul de la distance à faire pour la rotation des stepper. Néanmoins, ici, les deux steppers n'iront pas dans le même sens pour que la rotation se fasse correctement. Soit un tableau correspondant aux mouvements effectuer par les roues : [mouvement roue droite, mouvement roue gauche] :

$$\text{moveto\_rectiligne} = [\text{currentposition} + \text{step}, \text{currentposition} + \text{step}]$$
$$\text{moveto\_rotation} = [\text{currentposition} - \text{step}, \text{currrentposition} + \text{step}]$$

Dans le cas d'un déplacement en ligne droite, ou lors de la rotation, nous avions pensé à mettre en place un thread qui aurait permis aux deux steppers de se lancer simultanément. Néanmoins, il existe une librairie de l'environnement MultiStepper.h qui permet d'éxecuter exactement en même temps les steppers.

II. Réseau

I - Explication de l'architecture du réseau

Pour pouvoir envoyer les données récupérées par l'ESP nous avons mis en place un système de réseau où l'ESP sera le point d'accès. Nous pourrons y connecter plusieurs PC. Il nous permettra de communiquer de l'ESP aux PCs et d'un PC à l'ESP. Voici son architecture :

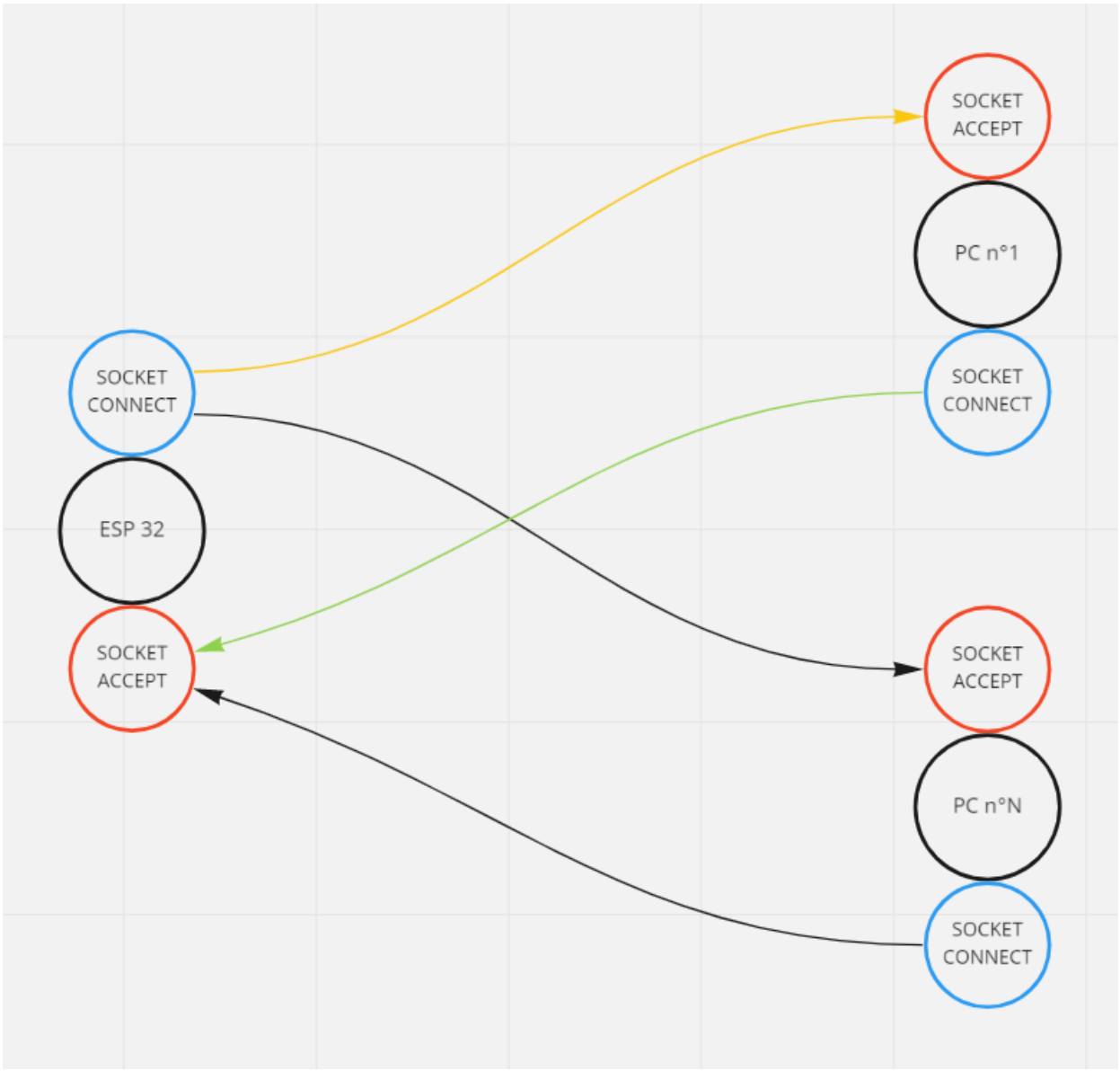

Figure 2: Architecture du réseau

Le principe est simple, nous aurons d'abord une connexion (flèche verte) qui s'effectuera du PC à l'ESP. Par la suite si l'ESP accepte cette connexion (qu'il n'a pas atteint le maximum de client), L'ESP répond en se connectant au PC (flèche jaune). Cela a pour conséquence que nous avons deux connexions. La verte nous servira pour communiquer de L'ESP vers le PC n°i, et la jaune nous servira pour communiquer du PC n°i vers l'ESP. Une fois les connexions mises en place, nous pouvons envoyer des requêtes sur les connexions respectives, c'est-à-dire que si nous voulons faire une requête à l'ESP alors on utilisera la connexion verte et si il y a une réponse alors il écrira sur la connexion jaune, sinon si nous voulons que l'ESP fasse une requête au PC n°i nous utiliserons la connexion jaune, et si il y a une réponse nous écrirons sur la connexion verte.

II - Explication de la mise en place du réseau

Nous avons mis en place plusieurs fichiers pour nous aider à faire le développement du serveur (l'ESP) et du client (Le PC).

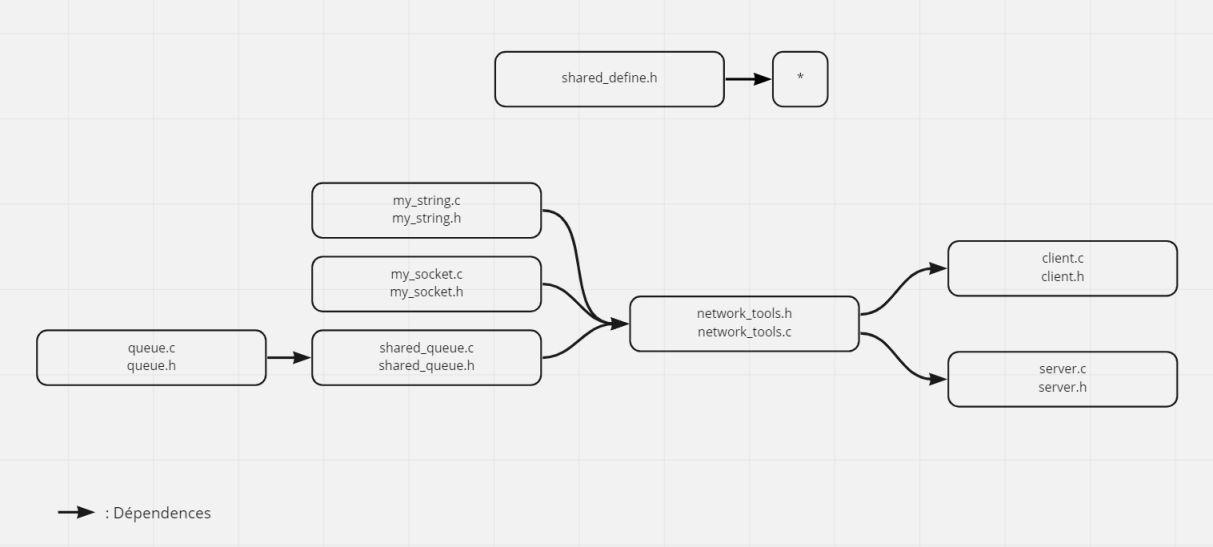

Figure 3: Fichiers utilisés par le reseau

Nous avons voulu faire en sorte de ne pas avoir de doublons dans le code, et donc, regrouper le plus gros du code avant la création du serveur/client.

Comme on peut le voir on a plusieurs outils : my_string, my_socket et shared_queue qui nous permette de nous faciliter la vie. Ensuite nous avons le network_tools qui nous sert à créer la plus grande partie du réseau. Enfin, soit le client soit le serveur qui nous sert à mettre le réseau en place en fonction de ce que l'on souhaite.

Le fichier shared_define.h est un fichier à part qui nous permet de définir des macros importantes comme les DBGs ou encore si on est en mode ESP ou non. En effet l'ESP nous demande de mettre nos fichier en .cpp qui a pour conséquence que l'on doit rajouter des - extern "C" - dans tous nos fichiers lorsque l'on veut compiler donc nous avons mit une macro en place pour que ça le fasse tout seul.

my_string :

Cet outil nous sert à avoir des chaînes de caractères facilements modifiables. Nous aurions pu utiliser GLib, mais dû à un soucis de comptabilité avec l'ESP, nous avons préféré refaire le code que nous utilisions nous même. Le système est très simple nous avons une structure my_string qui nous permet d'avoir accès au string, à sa longueur actuelle, à sa longueur maximal (la taille du malloc) puis enfin le step_malloc qui correspond au multiple que l'on doit avoir pour le malloc. Grâce à ça nous pouvons facilement, ajouter un string/int/char à un string ou encore en faire une troncature, tout en gérant l'espace mémoire avec des reallocs.

my_socket :

Ce module nous permettra de facilement manipuler les sockets. En effet nous avons créé la structure suivante :

```
1  typedef struct _my_socket
2  {
3      int mode;
4      char *port;
5
6      // SOCK_ACCEPT
7      int sfd_accept_mode;
8      pthread_t accept_loop;
9      void* (*fn)(void*);
10     void **fn_args;
11
12     // SOCK_CONNECT
13     int sfd_connect_mode;
14     char *host;
15 } my_socket;
```

Cette structure nous permet soit de gérer une connexion entrante, soit de gérer une connexion sortante, ou soit tout simplement les deux.

Le champ "mode" nous permet de savoir en quel mode nous somme, soit SOCK_CONNECT (Connexion sortante), soit SOCK_ACCEPT (connexion entrante), soit SOCK_BOTH (les deux).

Les champs sfd_accept_mode et sfd_connect_mode sont nos files descriptors créés par nos sockets par leurs modes respectifs.

Le champ "fn" correspond à la fonction à appeler quand il y a une connexion entrante, et le champ fn_args est un tableau d'arguments à donner à cette fonction si cela est nécessaire.

Nous pouvons manipuler cette structure grâce à de multiples fonctions comme : init_my_socket(), start_my_socket(), stop_my_socket() ou encore free_my_socket().

Si nous utilisons le mode SOCK_ACCEPT, alors la fonction start va créer un multithread serveur, et dès qu'elle aura détecté une connexion entrante, elle renverra la nouvelle connexion vers la fonction correspondant au champ "fn".

shared_queue :

Cet outil est une file qui est safe thread, elle nous permettra par la suite de faire une file d'attente de réponse à une requête. (On y revient un peu plus bas).

network_tools :

Cet outil va nous permettre comme dit plus haut de poser les bases de ce qu'on aura besoin dans notre serveur **ET** notre client c'est-à-dire se connecter, recevoir/envoyer des informations, ou se déconnecter.

Nous avons plusieurs structures mises en place. La première la plus importante va recenser toutes les informations importantes d'une connexion, la voici :

```
1  typedef struct _connection_info
2  {
3      int connection_type;
4      int sfd_receive;
5      int sfd_send;
6
7      shared_queue *pending_response;
8
9      pthread_mutex_t mutex;
10     int is_sending;
11
12     char id;
13     my_socket *sock;
14
15     pthread_t loop_thr;
16 } connection_info;
```

Le premier champ va nous dire quel type de connexion nous avons, soit TYP_SEND, soit TYP_RECEIVE ou TYP_BOTH. Ce type de connexion correspond à ce que l'on peut faire, envoyer des requêtes, les recevoir ou encore les deux. Les fields sfd_** correspondent aux files descriptors nous permettant de communiquer.

Le champ pending_response correspond à la file d'attente de réponse de requête. Comme dit un peu plus haut.

Le champ mutex nous sert simplement à rendre safe-thread la fonction permettant d'envoyer une requête.

Le champ sock correspond à ma structure my_socket associée à cette connexion.

Nous avons vraiment voulu rendre notre code modulable, autonome et simple d'utilisation donc nous avons mis en place plusieurs systèmes, le premier est que si l'on veut lancer une connexion nous avons seulement à appeler une fonction se nommant start_connection où l'on renseignera seulement trois informations, la connection_info (déjà initialisée grâce à une autre fonction), le mode de socket à utiliser, puis le type de connexion à utiliser. La fonction se chargera de démarrer la socket avec le mode demandé puis d'établir le type de connexion. Si le type de connexion est TYP_RECEIVE alors il lancera automatiquement un thread permettant de lire les requêtes et de les traiter.

En parlant de requêtes, nous avons aussi voulu faire quelque chose de modulable pour pouvoir facilement ajouter des requêtes si besoin est. Donc nous avons mis en place une structure se nommant cmd_info :

```
typedef struct _cmd_info
{
    int receive_response;
    int send_response;

    //Function that treat the request
    void (*fn)(
        connection_info *info, my_string *arg,
        my_string *response, void **extra_arg
    );
    void **fn_args;
} cmd_info;
```

Nous pouvons y renseigner si la commande peut recevoir une réponse ou en attendre une. Puis nous pouvons lui donner la fonction qui va venir traiter la requête et donner la réponse si nécessaire. Elle prendra aussi un tableau d'arguments si la fonction a besoin d'arguments externes. Nous pourrons ensuite venir utiliser la fonction init_cmd_type où l'on renseigne un int qui sera l'identifiant de la requête puis la structure que l'on vient de créer.

Parlons maintenant de l'envoi de requêtes: nous allons pouvoir facilement envoyer des requêtes grâce à une fonction se nommant : send_request, en effet elle prendra tout simplement en argument, la connection_info, le type de requête à envoyer, un my_string qui correspondra aux arguments (avec des espaces entre chaque) de la requête si elle en à besoin, une fonction qui permettera de traiter la réponse de la requête lorsque elle sera reçu puis enfin un tableau qui contiendra les arguments externes si besoin est.

Si une réponse à cette requête est nécessaire, alors la fonction créera une structure waiting_response :

```
typedef struct _waiting_response
{
    void (*fn)(my_string* r, void** fn_args);
    void **fn_args;
} waiting_response;
```

Elle renseignera simplement les informations données en arguments, l'enfilera dans la shared_queue de la connection_info puis le reste sera géré par le receveur lorsqu'il aura reçu la réponse. Il défilera et obtiendra les informations nécessaire pour pouvoir traiter la réponse.

client/server :

Ces deux modules sont presque équivalent, en effet, ils nous permettront d'initialiser et de lancer tous les éléments nécessaires au bon fonctionnement du client(PC)/serveur(ESP). En effet, ils initialiseront les commandes nécessaires à son bon fonctionnement, les initialisations et lancements de connexions, l'initialisation de la fonction lorsqu'une connexion est entrante, et enfin l'arrêt des connexions. Grâce à la modulabilité mise en place nous avons seulement à imbriquer les briques entre-elles ce qui nous a fait gagner beaucoup de temps.

Informations autres :

L'écriture de requêtes sur les files descriptors se fait de la façons suivantes :

```
1 TYPE_REQUETE\n
2 ARG1 ARG2 ARG3 ... ARGn\n\n\n\n
```

Il est donc possible, si la fonction qui traite la requête le prend en compte d'envoyer plusieurs requêtes en une en suivant le paterne suivant :

```
1 TYPE_REQUETE\n
2 ARG1 ARG2 ARG3 ... ARGn\n
3 ARG1 ARG2 ARG3 ... ARGn\n
4 ARG1 ARG2 ARG3 ... ARGn\n
5 ARG1 ARG2 ARG3 ... ARGn\n\n\n\n
```

L'envoi de réponses s'écrivent en suivant ce pattern:

```
1 -1\n
2 response\n\n\n\n
```

Et donc de la même manière que les requêtes, il est possible, si la fonction qui traite la réponse le prend en compte d'envoyer plusieurs réponses en une seule en suivant le paterne suivant :

```
1 -1\n
2 response\n
3 response\n
4 response\n
5 response\n\n\n\n
```

III. Récupération des données

I - I2C

Pour différents composants de notre projet, nous utilisons une interface I2C (Inter Integrated Circuit), déjà pré-configurée sur la carte de l'ESP32 (pins 21 et 22). Cette interface possède plusieurs avantages et inconvénients. Tout d'abord elle est très simple à mettre en place, en effet, seulement 2 fils sont nécessaires pour mettre en place une interface I2C, un bus d'horloge et un bus de données, et chaque élément à mettre sur l'interface vient se connecter en dérivation sur les bus de façon à ne pas être dépendant des autres éléments du bus I2C.

Cependant il est compliqué de supporter un taux de transfert élevé via cette interface, en effet, comme tout est placé sur le même bus, il devient vite très lourd de transférer les données de chaque éléments puisque à chaque information qui doit passer, l'interface doit être sur que le bus de donnée à été complètement vidé électriquement.

Toutefois, pour notre utilisation, l'interface I2C est largement suffisante.

L'interface I2C fonctionne par adresse. Cela veut dire que chaque composant du bus possède une adresse, donnée par le composant lui même, parfois éditable ou non. De ce fait, pour envoyer ou recevoir des données d'un élément branché en I2C, il faut communiquer avec le composant via son adresse.

II - Capteurs

Les capteurs fonctionnent via le port I2C, ainsi, tous les capteurs se contrôlent à l'aide de leurs adresses, et cela nous permet d'utiliser uniquement 2 pins + 1 par capteur pour en contrôler un grand nombre facilement.

III - Écran

Comme les capteurs, l'écran que nous utilisons, un LCD2004, fonctionne sur le port I2C, ainsi avec une librairie compatible nous pouvons facilement le contrôler, de plus cela permet de ne pas utiliser plus de ports sur la carte de l'ESP32.

IV - Adressage

Lors de l'initialisation, les capteurs et l'écran ont tous la même adresse. L'adresse n'étant pas modifiable sur l'écran, on doit donc changer l'adresse des capteurs, pour cela, la librairie VL53L1X.h contient directement une fonction prévue pour écrire sur la mémoire du capteur une nouvelle adresse, et l'assigner à la structure du capteur dans le code.

V - VL53L1X.h

Adressage :

Pour une utilisation multicapteur, nous avons mis au point un algorithme qui tour à tour active les capteur, pour changer leur adresse au fur et à mesure, et ainsi avoir une adresse pour chaque capteur, car sinon des conflits apparaissent, et dans la majorité des cas, les valeurs renvoyés sont nulles.

Récupération des données :

Pour récupérer les informations des capteurs, nous utilisons avec Arduino une librairie nommée VL53L1X, spécialement prévue pour nos capteurs, qui permets de façon relativement simple de contrôler leur mode de fonctionnement, et de récupérer les valeurs qu'ils renvoient avec de simples fonctions.

1^{ere} version :

Au départ, nous pensions utiliser 5 capteurs afin d'optimiser la vitesse de récupération des données, de ce fait nous avons mis au point une pièce permettant de fixer les capteurs avec chacun un angle différent de 22.5°.

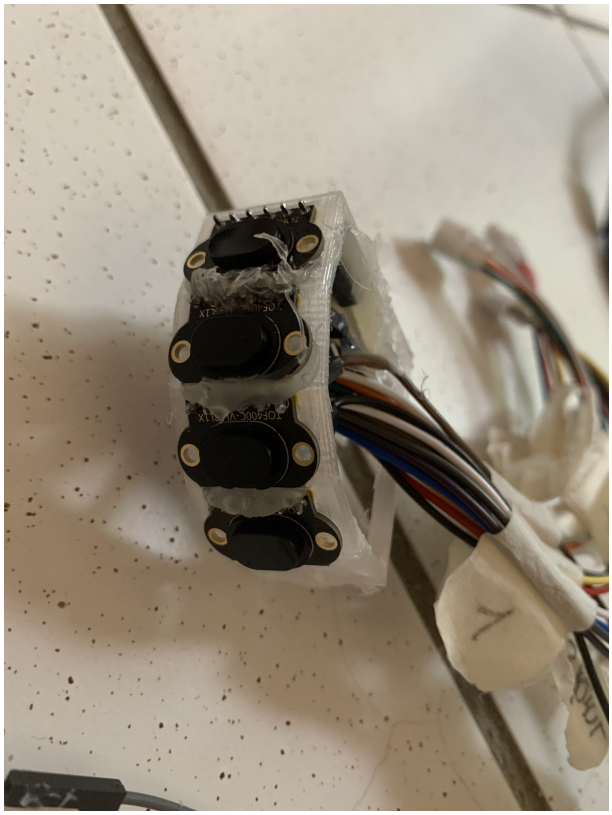

Figure 4: Photo de la V1 du capteur

2^{eme} version :

Ensuite, après plusieurs tests, nous nous sommes rendus comptes que utiliser 1 capteur était tout aussi rapide, car les capteurs ne pouvaient capter qu'un par un, ainsi, pour simplifier le code et la mise en pratique du robot, nous avons fait une version du capteur avec un seul VL53L1X.

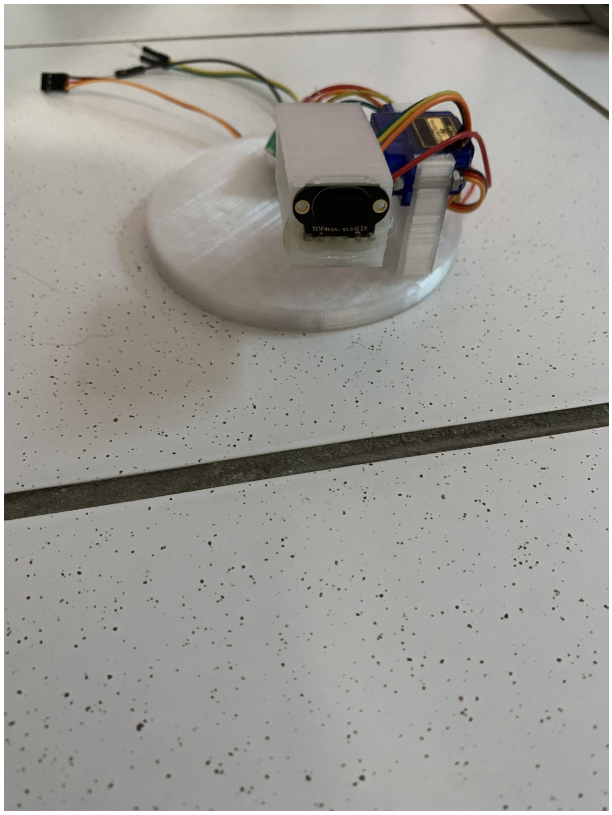

Figure 5: Photo de la V2 du capteur

VI - Mouvements

Pour déplacer le capteurs, nous utilisons 2 choses, tout d'abord, pour régler l'angle horizontal, nous utilisons un moteur pas à pas. Puis pour régler l'angle vertical, nous utilisons un servomoteur, qui peut se régler degré par degré, nous permettant facilement d'orienter notre capteur en hauteur. Ainsi, à l'aide de ces deux éléments, dont je parlerais plus en détail plus tard, nous pouvons contrôler et connaître la position exacte du capteur à 360°, et ainsi pouvoir scanner notre environnement.

IV. Le robot

I - Conceptualisation

Pour la conceptualisation du robot, nous sommes passés par plusieurs étapes. Déjà il a fallu se mettre d'accord sur la façon dont on allais le faire fonctionner, et nous sommes partis sur la possibilité la plus simple à mettre en place dans notre cas, à savoir d'avoir uniquement 2 roues motrices sur les côtés permettant de soit de le faire avancer/reculer, soit de le faire rotationner sur lui même, en faisant fonctionner les moteurs en sens contraire.

Pour les modèles 3D, nous avons eu plusieurs idées, tout d'abords nous voulions faire un robot circulaire, mais nous avons jugée cette idée mauvaise puisque elle nous faisais perdre beaucoup de place effective pour placer l'électronique dans le robot. De plus il était plus facile de designer et de moduler une forme rectangulaire.

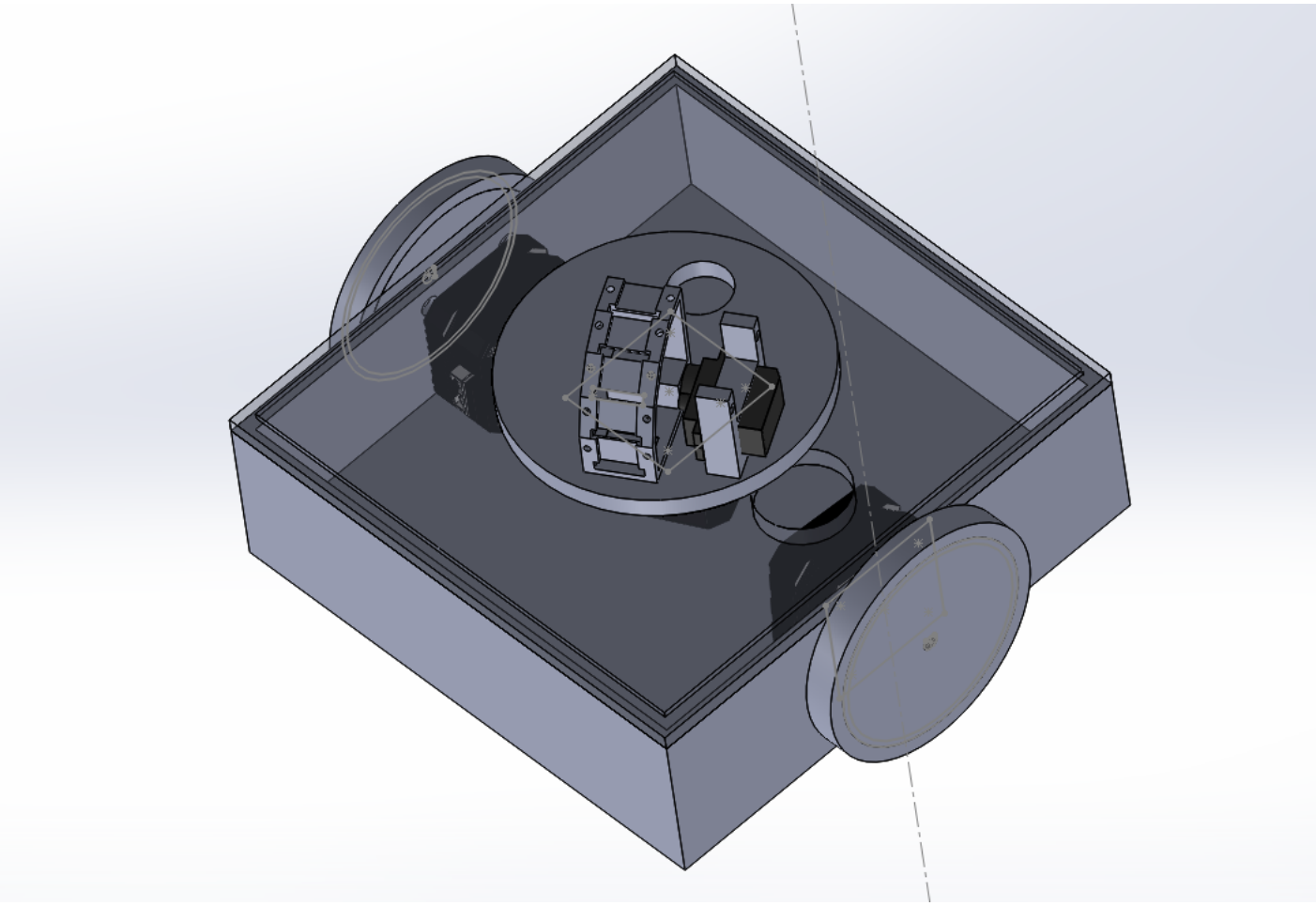

Figure 6: Image de l'assemblage du robot

Pour la fabrication du robot, nous avions à disposition une imprimante 3D permettant de fabriquer en quelques heures les pièces voulu. Pour le matériau nous en avons choisi 2, le PETG pour les pièces mécaniques, très résistant, facile et rapide à imprimer c'est un matériau qui offre une résistance mécanique plus que suffisante pour notre projet. En 2ème matériau, nous avons du TPU, un matériau flexible, nous l'avions choisi à l'origine pour pouvoir faire des roues plus adhérentes au sol, mais nous avons trouvé une alternative plus intéressante pour nos roues.

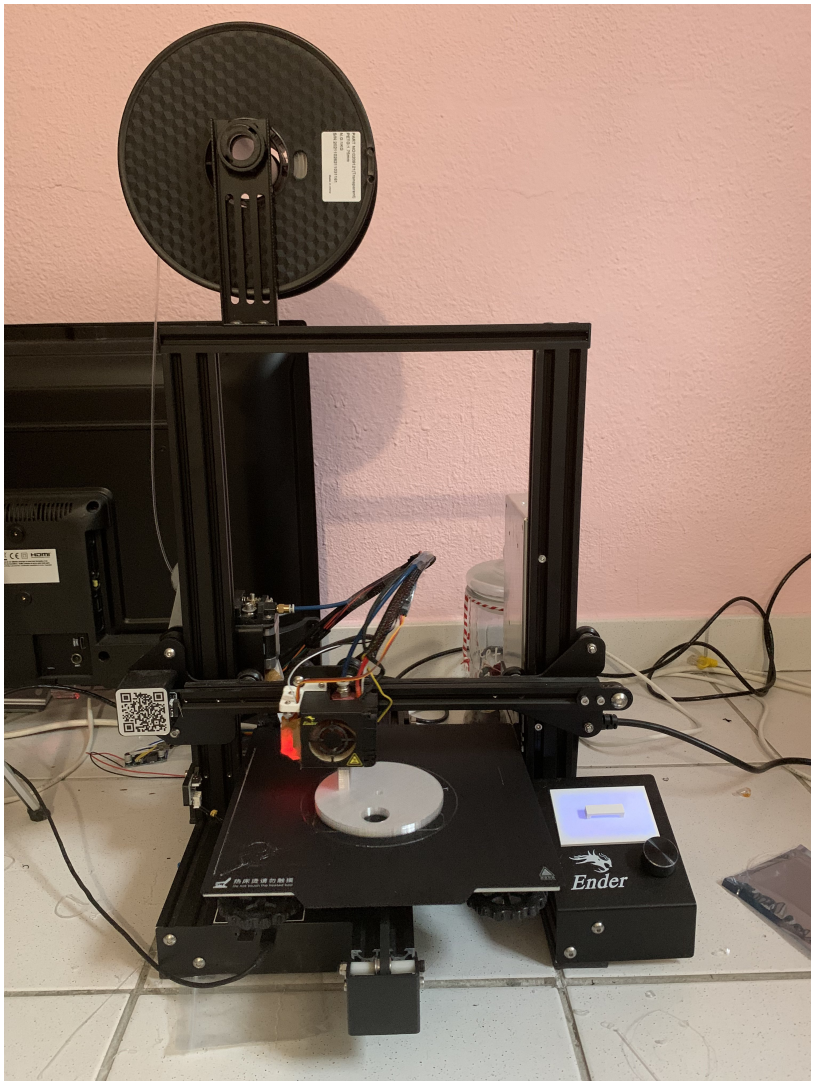

Figure 7: Photo de l'imprimante 3D nous ayant permis d'imprimer la structure du robot

Pour les roues comme indiqué précédemment, nous avions pour idée de les imprimer en 3D en 2 partie, une pièce centrale en PETG et une pièce souple permettant de faire comme un pneu. Cependant nous avions remarqué que l'adhérence n'était pas optimale, donc nous avons fait des recherches, pour au final se mettre d'accord sur l'achat de roues de trottinettes, en caoutchouc, très adhérentes donc, et permettant au robot d'avoir quasiment aucun glissement.

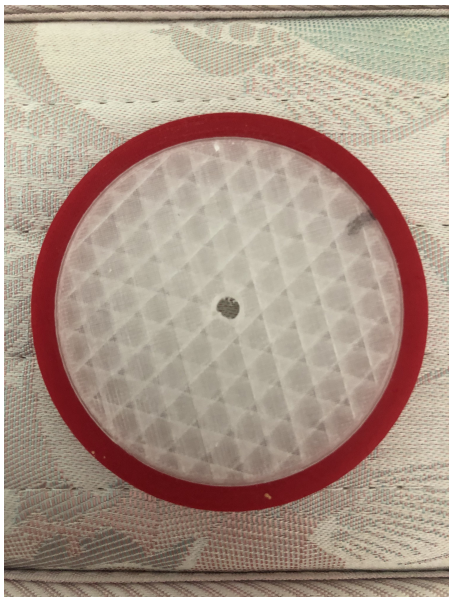

Figure 8: Photo des roues V1

Figure 9: Photo des roues V2

II - Motorisation

Pour la motorisation, nous utilisons 3 moteurs dit pas à pas. 2 pour les roues, et 1 pour la rotation du capteur. Ces moteurs ont plusieurs avantages et inconvénients. Il ont l'avantage d'être très précis de par leur fonctionnement, en effet les moteurs pas à pas sont composés de plusieurs électro-aimants faisant en sorte que le moteur se place à une position très précise à chaque commande. Cependant, contrairement à des moteurs classiques, ces moteurs consomment beaucoup d'énergie, même à l'arrêt, puisqu'il faut alimenter constamment les électro-aimants actif. Les moteurs pas-à-pas que nous utilisons sont des NEMA 17, ils possèdent 200 pas, et donc peuvent prendre par défaut 200 positions différentes, nous menant à des rotations de 1.8° par pas. Cependant il est possible à l'aide de nos contrôleurs de faire des micros pas. Le principe des micros pas est plutôt simple, par exemple pour des demi-pas, au lieu d'alimenter une bobine à la fois, il suffit d'alimenter 2 bobines entre chaque pas, et ainsi le moteur viens se placer dans une position entre les 2 pas "physiques". Les contrôleurs que nous utilisons nous permettent d'aller jusqu'à 1/16 de pas, et donc de virtualiser 3200 pas. Cependant, dans notre cas, nous avons estimer qu'il n'était pas intéressant de faire des micros-pas, car, comme il est nécessaire d'alimenter plusieurs bobines à la fois, les moteurs consomment beaucoup plus qu'en mode "full step", et nous avons estimé qu'il n'était pas nécessaire d'avoir des mouvement en dessous d'1.8°.

III - Électronique

Plusieurs composants électroniques composent notre robot, déjà, comme dit plus haut, les capteurs, l'écran, ensuite s'ajoute à ça 2 boutons dit de fin de course, un bouton à levier, 3 contrôleurs pour les moteurs, les 3 moteurs, et 1 modulateur de tension.

Les contrôleurs :

Les contrôleurs que nous utilisons pour les moteurs pas à pas sont des A4988, ils nous permettent à l'aide de 2 pins de contrôler nos moteurs, un pin pour la direction, et un pin pour les pas, de façon très simple, à chaque impulsion donnée dans le pin de pas, le contrôleur ordonne au moteur de faire un pas dans la direction donnée par le pin de direction. Ces contrôleurs ont besoin de quelques réglages pour pouvoir fonctionner correctement. En effet on doit régler une résistance variable, qui définie l'intensité maximale envoyée dans les moteurs, dans notre cas, nos moteurs doivent fonctionner avec une intensité de 1.2 ampères, ainsi à l'aide d'une formule donnée dans la documentation du contrôleur, j'ai pu déterminer la résistance à avoir, et donc de façon plus simple à mesurer, la tension qui devait passer dans la résistance, à savoir 687 mV.

Alimentation :

Pour l'alimentation, nous avons mis 2 Batteries NI-MH de 7.4V en série pour avoir une tension de sortie d'environ 15v, permettant d'alimenter directement les moteurs dans leurs intervalle de tension idéal.

Modulateur de tension :

Ayant une alimentation en 15v, et l'ESP32 ne pouvant fonctionner qu'avec des tensions situées entre 3.3v et 5v, il a fallu ajouter un modulateur de tension, qui permet, à l'aide d'une résistance variable de moduler la tension d'entrée vers une tension de sortie souhaitée, ici 5v.

Servo moteur :

Le fonctionnement du servo moteur est plutôt simple en effet, ils sont composés de 3 pins, un d'alimentation, une masse, et un pin de contrôle. Ainsi, à l'aide de la librairie ESP32Servo.h nous pouvons indiquer au servo moteur l'angle que l'on veut qu'il prenne.

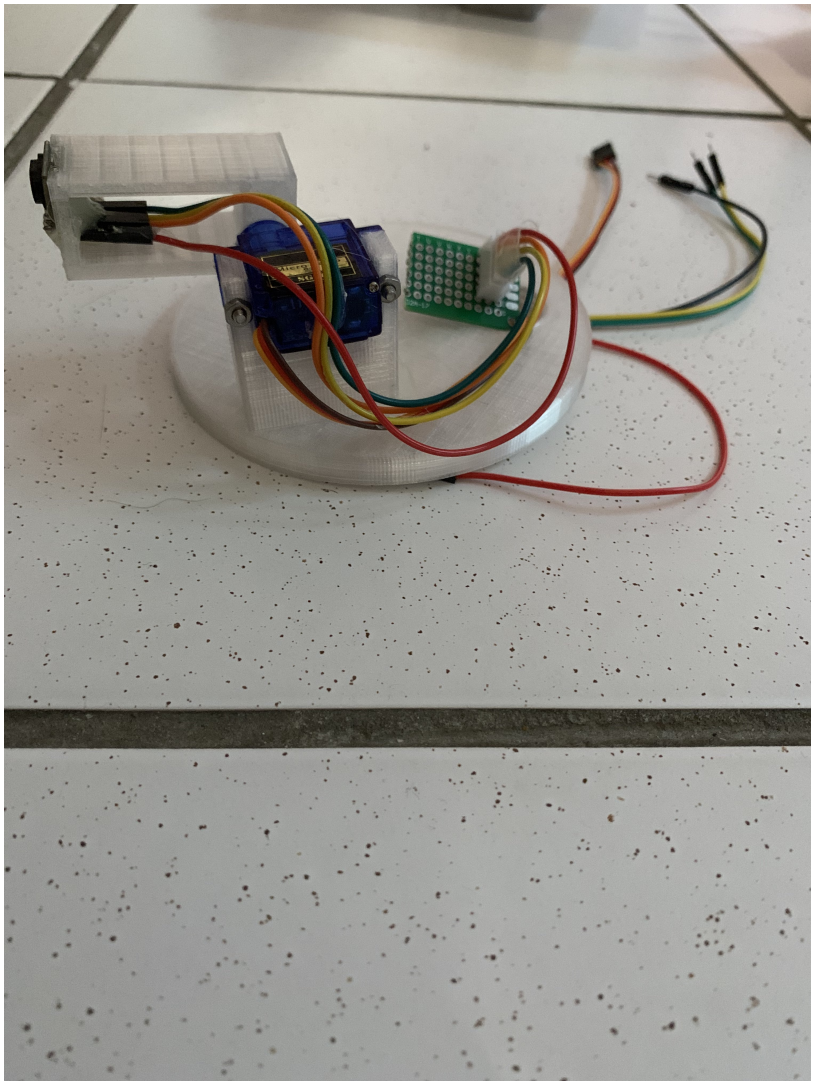

Figure 10: Photo du capteur avec le servo moteur

Câblage :

Pour le câblage du robot, nous utilisons plusieurs choses. Pour l'alimentation, deux connecteur Tamyia viennent relier les deux batterie et à l'aide de 2 pinces (plus facile de débrancher rebrancher le robot) nous alimentons l'électronique. Pour l'électronique, les contrôleurs des moteurs pas à pas sont mis sur une breadboard, ainsi que l'alimentation de la logique en 5v et l'alimentation des moteurs. Pour les moteurs et le capteur nous utilisons des connecteurs JST 4 pins permettant de facilement débrancher et rebrancher les éléments et ainsi debug notre robot de façon beaucoup plus aisée.

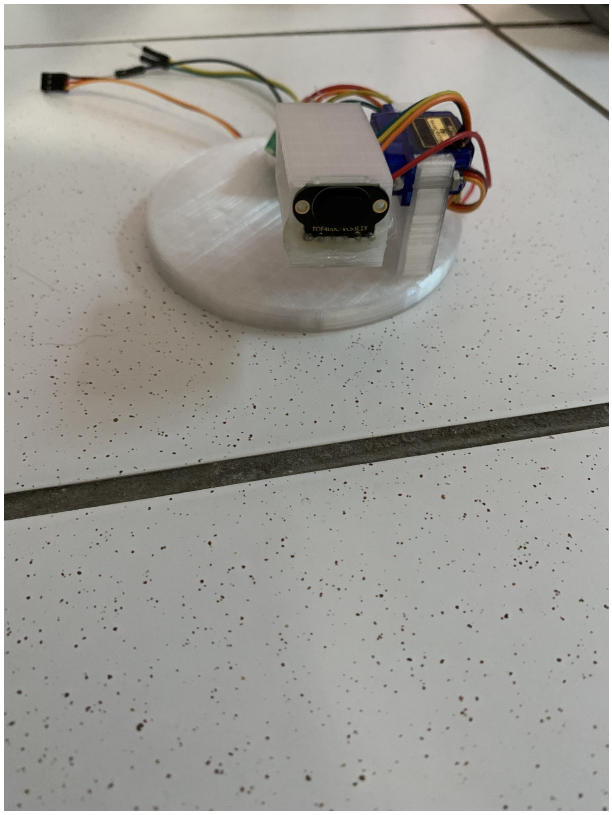

Figure 11: Photo du câblage

Affichage :

Pour l'affichage, nous utilisons l'écran comme montré ci dessous, cet écran est simple et sans couleur mais nous permet de façon accessible d'afficher des informations sur le robot, comme sa position ou son angle d'inclinaison (il peut aussi dire bonjour).

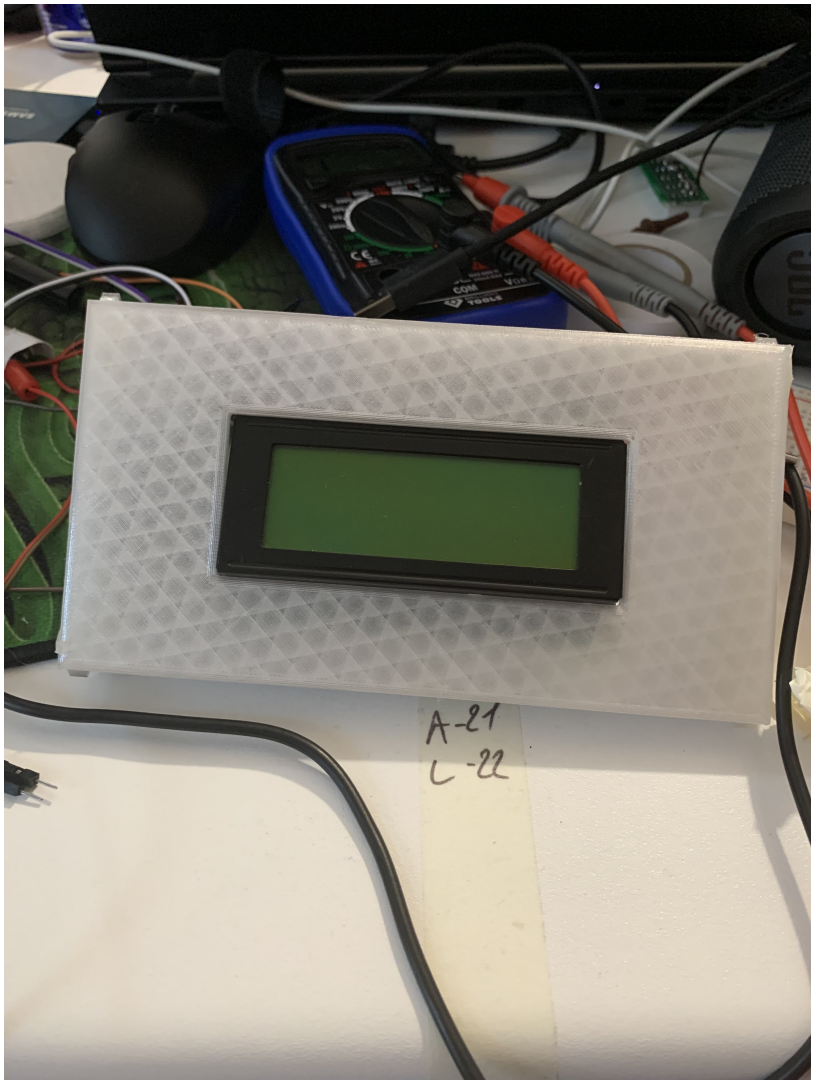

Figure 12: Photo de l'écran utilisé

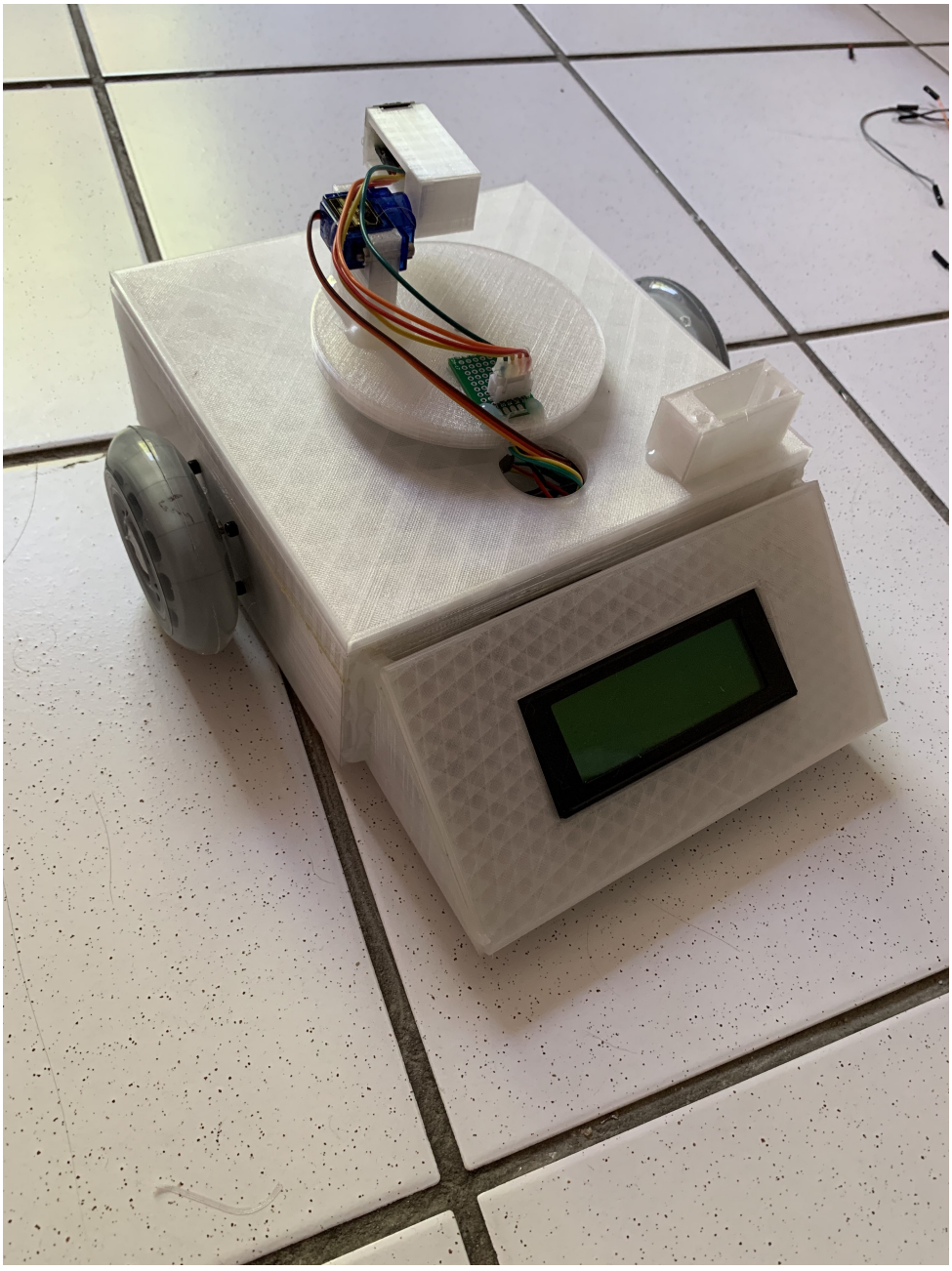

Figure 13: Photo du robot final

IV - Programation

Généralité :

Suite à toutes les choses que nous voulions faire avec le robot, nous avons choisis de créer une structure Robot pour nous facilité les passages de variables dans les fonctions, les variables pour commander les roues, les rotations du capteur ou encore le capteur, etc...
Voici la structure :

```
typedef struct _Robot
{
    int mode;

    //Position
    int pos_x;
    int pos_y;
    float pos_theta;

    //Sensor angles
    float sensor_theta;
    float sensor_phi;

    //Steppers
    MultiStepper wheels;
    AccelStepper* wheel_l;
    AccelStepper* wheel_r;
    int sensors_step;

    //Sensor
    VL53L1X *sensor;
    int sensor_read;
    Servo *servo_sensor_theta;

    //Move pthread
    int is_moving;
    pthread_t thr_move;
    pthread_t thr_move_theta;
    pthread_t thr_move_sensors_phi;

    //server
    server_info* server;

    //ecran
    Lcd* lcd;

    //Catadioptre
    Reflectors reflectors;
} Robot;
```

On peut remarquer beaucoup d'informations à partir de cette structure.
Premièrement, on a décidé d'enregistrer les informations de la positions du robot en cartésien. Pour pouvoir obtenir la position du robot au fur et à mesure de l'avancée nous avons décidé de choisir les données "parfaite" que nous donne les calculs, c'est à dire que l'on calcule les positions grâce à la distance parcourue par le robot. En effet notre robot ne peut que avancer/reculer OU tourner sur lui même, qui rend les calculs très facile (Et nous ne remarquons aucun de glissement lors de l'arret,se qui est très important). Au départ nous avions choisis d'utiliser le système d'Odométrie mais on c'est très vite rendu compte que c'était une solution beaucoup trop "puissante" pour notre cas simpliste de déplacement.

Ensuite nous enregistrons les angles du capteur. Il y a aussi les structure nous permettant d'utiliser les steppers grâce au librairies AccelStepper et MultiStepper. Pour le stepper du capteur qui nous permet de faire les rotations sur le plan horizontale, nous avons décidé de directement utiliser les pins sur lesquels était branché le stepper contrôlant la rotation. En effet, Nous voulions pouvoir faire une vitesse constante et non une vitesse avec accélération se qui est fait dans les deux précédentes librairies cités. Nous avons deux pins, un qui contrôle la rotations, si son pin est à 5V alors alors le stepper tournera dans le sens trigonométrique sinon si il est a 0V il tournera dans le sens contraire. Ensuite nous avons un deuxième pin qui nous sert à faire faire des steps. Pour faire un step nous avons seulement à faire une impulsion de 5V.
Nous avons par la suite différentes variables pthread_t qui vont nous "permettre" de lancer des actions en asynchrone.
Nous avons le serveur qui a été présenté un peu plus haut. Il y a aussi l'écran et les catadioptres qui seront présentés dans les prochaines parties.

Nous avons aussi la variable 'int mode' qui nous permet d'avoir le mode actuel du robot. Nous en avons 5 différents :

- R_MODE_WAITING : Nous permet de savoir que le robot est en attentes de requêtes.
- R_MODE_CONTROLLED : Nous permet de savoir si l'utilisateur est entrain de faire déplacer le robot.
- R_MODE_SCAN : Nous permet de savoir que le robot fait un simple scan
- R_MODE_SCANWALL : Nous permet de savoir que le robot fait un scan de la pièce en longeant les murs
- R_MODE_SCANROOM : Nous permet de savoir que le robot fait un scan de la pièce

Il nous permettent de nous reperer dans les diffèrentes actions du robot et de savoir comment lui dire de quoi faire.

Toutes les commandes sont envoyées depuis l'interface client. Si une action est en cours alors il ne se passera rien.

Donc cette structure nous sert à, faire bouger le robot, le faire tourner sur lui même, faire les scans, faire tourner le PHI du capteur, faire tourner le THETA du capteur et calculer les nouvelles positions du robot.

L'écran :

En se qui concerne l'écran, nous avons eu des difficultés à l'intégrer puisqu'il fonctionne en I²C, se qui veut dire qu'il y a une clock en arrière plan qui contrôle tout les envoies de données à l'écran. Et donc cela signifie que les fonctions le concerant sont inutilisable dans les threads. Or notre programme est fortement constitué de thread pour fonctionner et donc si nous voulons faire de l'affichage en temps réels nous devons donc utiliser l'écran en thread aussi. Donc nous avons pensé à créer une structure nous permettant de mettre en cache les messages que l'on veut envoyé à l'écran et qu'ensuite une boucle étant sur le thread principale, viendra les traités.
Voici les structures utilisées:

```
1  struct Queue
2  {
3      int clear;
4      int row;
5      char str[21];
6
7      struct Queue *next;
8      struct Queue *prev;
9  };
10
11 typedef struct _Lcd
12 {
13     LiquidCrystal_I2C *lcd;
14
15     int mode;
16     void *robot;
17
18     pthread_mutex_t mut;
19     struct Queue *top;
20     struct Queue *tail;
21     int nb_to_show;
22 } Lcd;
```

Nous avons premièrement dans la structure Queue, des valeurs enregistrées de l'utilisateur, si nous devons vider intégralement l'écran, sur qu'elle ligne nous devons afficher et que'est ce que nous devons afficher.
Ensuite nous avons une structure Lcd, qui contient l'écran, son mode (on y reviendra plus tard), le robot et enfin deux pointeurs de Queue, un qui correspond au sommet de la Queue (la plus ancienne valeurs, donc à pop) et le bas de la Queue (la valeur la plus récente, donc on doit push devant celle la.
Nous avons donc une fonction update qui vient tout simplement pop une valeur de la Queue et qui vient l'afficher, cette fonction est appelée dans la boucle principale de l'arduino qui fait que nous somme en accord avec la clock de l'I²C.
Pour faire tout cela nous avons du utiliser un pthread_mutex_t qui nous permet de verrouillé l'utilisation de la Queue à un thread par un thread seulement, pour éviter les conflits d'accès et les segfaults de mémoire.

Nous avons donc aussi des modes, les voici :

- LCD_MODE_POS : Viens print les positions du robot seulement
- LCD_MODE_STEPS : VIens print les etapes que fait actuellement le robot
- LCD_MODE_DEBUG : Viens print de debug, non utile pour l'utilisateur
- LCD_MODE_REFLECTORS : Viens print les emplacements des catadioptres détecter.

Ceci nous à fait rajouter des boutons sur le bord de l'écran pour pouvoir changer de mode facilement.

Position robot
Pos X : +00000 mm
Pos Y : +00000 mm
Theta : +000.00 d

L'affichage pour le mode "LCD_MODE_STEPS", il y aura plusieurs affichage:

- l'affichage des initialisations
- l'affichage des modes scans avec leurs étapes
- l'affichage des déplacements

Et enfin pour le mode : LCD_MODE_REFLECTORS

Position catadioptre
R1:+0000,+0000,+0000
R2:+0000,+0000,+0000
R3:+0000,+0000,+0000

Les catadioptres :

Se système nous permet de localiser le robot grâce à 3 tubes de catadioptres. Il va lorsqu'il se démarre les détecter grâce au capteur et un certain mode ambientLight qui va se mettre à 0 si nous pointons sur un catadioptre. Et donc nous pouvons localiser les 3 tubes, prendre les distances de chacun, comparer aux autres et détecter leurs emplacements comparer au robot qui est en (0, 0). Grâce à ses informations nous pouvons par la suite bouger le robot et détecter en redétectant où sont les catadioptres l'emplacement du robot.

Pour faire tout cela, nous avons mis en place deux structures :

```
typedef struct _Reflector
{
    int x;
    int y;
    int dist;
    double theta; // In rad
} Reflector;

typedef struct _Reflectors
{
    Reflector* refl[3]; // must not change manually

    int dist_0_1; // must not change manually
    int dist_1_2; // must not change manually
    int dist_2_0; // must not change manually

    int dist[3]; // Can change
    int theta[3]; // Can change, IN RAD
} Reflectors;
```

La première sert d'emplacement d'un catadioptre, avec son x, y, sa distance du robot qui est en (0, 0) et le théta du robot lors de la détection du catadioptre.
La deuxième est la structure est la structure principal. Nous avons nous nos 3 catadioptres de références, nos distances entres et enfin les dist détecter par le robot avec les thetas. Si nous voulons nous localiser nous n'avaons qu'à :

- Détecter les catadioptres.
- Prendre les distances entre.
- Les rangers de tel sorte à les avoirs dans le même ordre en fonction de la distance entre deux catadioptres
- Trouver le delta théta
- Les mettre sur la même orientation
- Calculer le delta x et delta y
- On a nos points

Par soucis de temps, nous n'avons pas pu implémenter la localisation par catadioptre. Mais le code décrit juste au dessus a été fait et est fonctionnel.

V. Rendu graphique

I - GTK

L'interface graphique permettra à l'utilisateur de connecter le robot pour qu'il effectue sa modélisation de l'environnement, de charger une sauvegarde, d'afficher cette modélisation qui est réalisée grâce à OpenGL et d'enregistrer cette modélisation.

Cette interface graphique est réalisée grâce à GTK et à Glade. Glade nous a notamment permis de concevoir l'aspect visuel de l'interface. Compte tenu des autres contraintes plus importantes sur la mise en place du robot notamment, nous avons pris la décision de ne pas nous concentrer sur l'aspect visuel mais sur l'aspect fonctionnel.

L'aspect visuel de l'interface graphique a été très légèrement modifié depuis la dernière soutenance, notamment des boutons permettant de modifier le mode de scan du robot (Scan statique, Scan de la salle, Scan des murs) et de lancer le scan selectionné. De plus, nous avons rajouté plusieurs éléments, comme la gestion d'erreur qui apparait directement sur l'interface (par exemple si le robot ne peut pas se lancer). De plus, lorsque nous lançons la visualisation d'un rendu sur OpenGL, nous nous étions rendu compte qu'il était très difficile de pouvoir faire d'autres actions sur l'interface car la visualisation nécessitait beaucoup de ressources. Par conséquent, pour remedier à ce problème nous avons mis en place un thread qui permet d'éviter ce problème.

L'ensemble des boutons présents ont été connectés avec les fonctions nécessaires pour les rendres fonctionnels. Par conséquent il est possible de charger une sauvegarde en naviguant dans l'explorateur de fichier (il faut ouvrir un fichier .off), de connecter/deconnecter le robot, de lancer le rendu, de créer une sauvegarde, de modifier le mode de scan et de le lancer, d'accéder au site web ou de quitter l'interface.

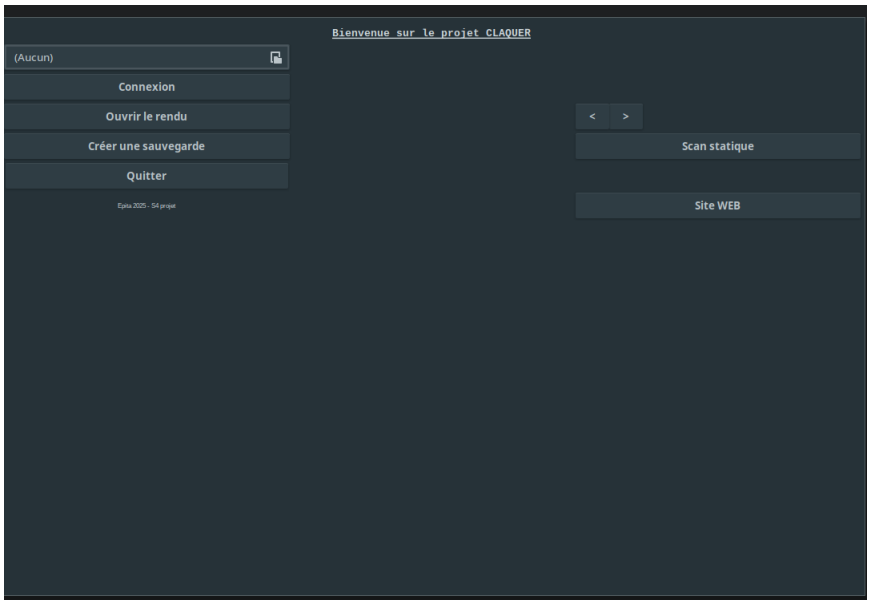

Figure 14: Image de l'aspect graphique de l'interface

II - OpenGL

La fenêtre de rendu graphique se lance depuis l'interface graphique GTK avec le bouton "Ouvrir le rendu". Dans cette fenêtre de rendu, il y a plusieurs entités sur lesquelles nous allons nous pencher : Les points, la caméra et l'éclairage.

Commençons par la caméra. Celle-ci peut se déplacer de deux façons différentes, un zoom et une "arcball". Le zoom est assez explicite donc nous n'allons pas nous attarder deçu mais il est important de noter qu'il n'est pas implémenté comme un changement de champ de vision ou FOV (Field Of View). C'est un déplacement selon l'axe orthogonal au plan de la caméra, c'est-à-dire vers l'objet pointé par la caméra. Ensuite, nous avons l'arcball. Ce mouvement correspond à un déplacement sur la surface d'une sphère dont le rayon est la distance caméra-objet et le centre est donc l'objet. Un des plus grands défis de ce mouvement est d'empêcher le blocage de cardan (gimball lock). Ce blocage survient lorsque sur les 3 axes de rotations, 2 se retrouvent coplanaires. Pour implémenter ce mouvement, nous avons fait appel à des rotations de matrice et des projections orthogonales pour calculer le déplacement de la caméra en fonctions des mouvements de la souris.

Ensuite, notre fenêtre capable d'afficher des points. Pour ce faire, nous utilisons un vecteur de buffer d'objets et un vecteur d'attributs d'objets. Afin de les initialiser, nous avons besoin d'un tableau de flottants, qui trois à trois représentent la coordonnée d'un point. Il faut ensuite générer et lier nos deux vecteurs. Ensuite nous lions les données de notre tableau dans notre buffer objet. Pour expliquer à OpenGL comment se déplacer dans notre tableau, nous allons attribuer des pointeurs à notre première donnée de point, puis que se déplacera de façons à récupérer la donnée du point suivant. Ceci est important à noter puisque chaque point de notre tableau de flottant est défini par 6 éléments. Les trois premiers sont ses coordonnées dans l'espace. Et les 3 suivants sa normale, dont nous expliquerons l'utilité dans le paragraphe suivant. Maintenant que nous avons notre vecteur d'attributs d'objets, nous pouvons dessiner les élements sur notre fenêtre.

Finalement, l'éclairage. A proprement parler, il n'y a pas de lumière, la façon dont nous stimulons l'éclairage est via des shaders. Ces shaders, écrits dans le langage GLSL comprennent trois composantes pour stimuler notre éclairage. Un éclairage ambient, qui est similaire en tout point de l'espace. Un éclairage diffus, qui exige la création de normales à nos points. Plus la lumière est allignée avec la normale de notre point, plus celui-ci sera lumineux depuis la vision de la caméra.

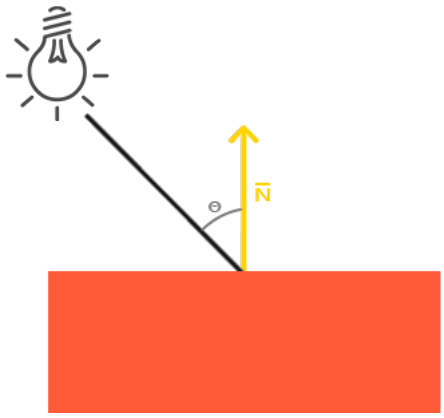

Figure 15: Schéma éclairage diffus

Pour terminer, nous utilisons aussi l'éclairage spéculaire, qui, en plus de la normale de notre point, se base aussi sur la direction depuis laquelle nous regardons.

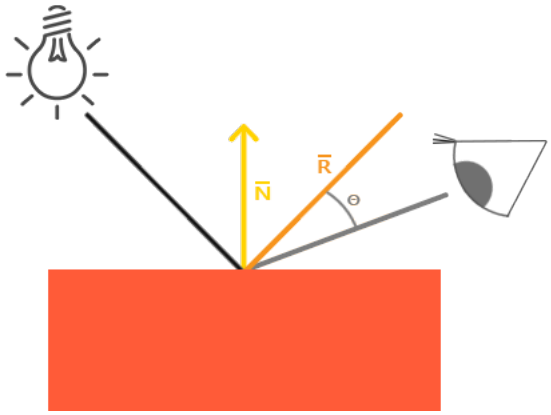

Figure 16: Schéma de l'éclairage spéculaire

Enfin, la combinaison de ces trois types d'éclairage nous donne enfin l'éclairage Phong.

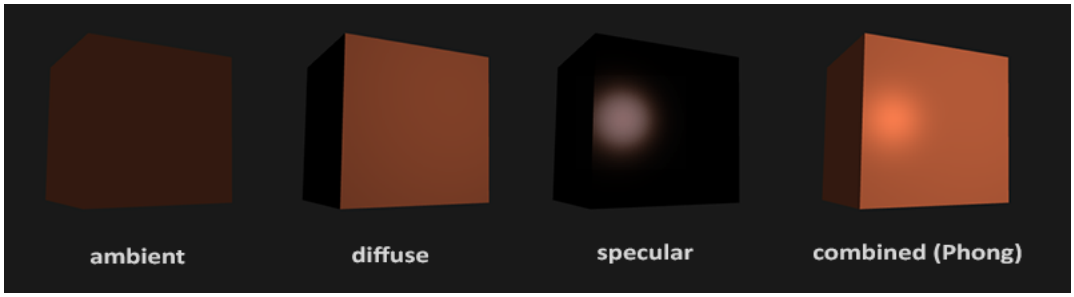

Figure 17: Un rendu de chaque type d'éclairage

Toutes les opérations réalisées pour déterminer ces combinaisons d'éclairages se trouvent donc dans des shaders. Les codes sources de ceux-ci se trouvent dans le dossier interface/shaders/. Un shader est divisé en deux objets distincts: un shader de vecteur et un shader de fragment. Ils sont différents mais indispensables l'un à l'autre. Sans rentrer dans trop de détails, chaque déclaration de vecteur commence par sa version. Ensuite, similairement au langage C, ils sont composés de variables, de fonctions et d'un main. Il y a des déclarations de variables spéciales appelées uniform qui sont des variables modifiables depuis notre code C. Nous avons créer une structure shader, qui contient l'identifiant de son programme, ainsi que les fonctions utilitaire de cette structre. L'initialisation d'un shader depuis ces deux fichiers sources fragment et vecteur, l'utilisation de ce shader et enfin la recherche et modification de variables uniformes.

VI. Stockage de nuage de points

Pour stocker notre nuage de point envoyé par l'ESP, nous nous sommes dirigés vers une structure connue du nom d'octree. Un arbre dont chaque noeud comprend 8 fils. Comme nous ne ferons qu'une rapide approche de cette structure, nous vous conseillons une explication plus détaillée ici. Pour résumer, l'espace est divisé en 8 cubes de tailles égales. Chaque cube, ou bien nœud comporte un nombre quelconque d'objets, entre 0 et autant que la mémoire peut en supporter.

Concentrons nous sur la partie la plus importante : l'implémentation. Il en existe différentes en fonction du type de données qu'il faut stocker. Comme nous aurons beaucoup de points assez rapprochés, nous avions choisi cette implémentation pour les nœuds :

```
1  struct OctreeNode
2  {
3      // Liste des 8 enfants
4      struct OctreeNode   *children;
5      // Liste des objects contenus dans le noeud
6      struct Object       *first_obj;
7      // Coordonn es du centre
8      struct Vector3      *center;
9      // Nombre d'objets contenus (8 max actuellement)
10     int                 n_obj;
11     // Valeur de la moiti  de la longueur d'une ar te
12     float               half_size;
13     int                 is_leaf;
14 };
```

Par rapport à la seconde soutenance, nous avons un peu changé la structure. à la place d'allouer un pointeur de longueur 8 fois la taille d'un OctreeNode, ce qui faisait effectivement des erreurs d'adresses, nous nous sommes tourné vers l'implémentation premier fils, frère droit. Ceci en ajoutant simplement un pointeur next comme ceci :

```
1       struct OctreeNode *next;
```

Et ceci pour l'arbre en lui même, qui lui n'a pas changé depuis sa première implémentation:

```
1   struct Octree
2   {
3       // Pointeur vers la racine de l'arbre
4       struct OctreeNode   *root;
5       // Nom de l'arbre, pour sauvegarder
6       char                *name;
7       // Nombre d'objet dans l'arbre pour la sauvegarde
8       int                 n_obj;
9       // Profondeur  de l'arbre
10      int                 depth;
11  };
```

Pour référence, voici les deux autres structures implémentées pour les octree. La première est notre représentation d'un point en 3D : une structure contanant trois variables pour les coordonnées x, et z du point que nous voulons représenter. La seconde structure nous permet de représenter un point dans un arbre. C'est une liste dynamique chainée, d'où le pointeur next. Le pointeur coords permet de stocker ses coordonnées via notre précédente structure. Enfin le double pointeur edges représente les deux points qui sont reliés avec notre actuel pour stocker une des faces de notre reconstruction.

```
1   struct Vector3
2   {
3       double x, y, z;
4   };
5
6   struct Object
7   {
8       // L'objet suivant dans le noeuds
9       struct Object       *next;
10      // Les coordonn es de l'objet
11      struct Vector3      *coords;
12      // Liste doublement chainee pour representer les arcs
13      struct Vector3      **edges;
14      // nombre pour representer l'objet dans la liste des faces de la sauvegarde
15      double              index;
16  };
```

En ce qui concerne l'implémentation des algorithmes en général, ils ne sont pas très compliqués. En revanche, la recherche d'un nœud dans lequel on voudrait ajouter un objet est fastidieuse, puisqu'il faut à chaque fois comparer les centres de chaque cube.

Nous avons aussi une fonction qui permet de charger une structure depuis un fichier. Ces fichiers de sauvegarde se présente sous la forme suivante.

```
OFF
#nom de la structure
X Y Z
_ _ _
_ _ _
...
```

Dans cette structure, X représente le nombre de sommets/objets de notre arbre. Y représente son nombre d'arêtes, et enfin Z son nombre de face. Ces deux derniers sont purement visuels puisque nous ne les sauvegardons jamais, ni ne les chargeons. Ensuite, les '_' représentent des nombres quelconques qui sont la coordonnée x, puis y, et enfin z de notre point

La fonction de sauvegarde sert aussi à libérer l'espace mémore de notre structure. Dans le cas où nous ne voulons pas sauvegarder la structure, il suffit de mettre le second paramètre à 0

VII. Algorithmes de modélisation

Pour notre algorithme de reconstruction de surface, nous avons implémenté l'algorithme crust. Celui-ci est basé sur les deux éléments que sont la triangulation de Delaunay et son dual, le diagramme de Voronoi. Le principe de cet algorithme est de calculer le diagramme de voronoi d'un nuage de point. On va ensuite injecter tous les somments de ce diagramme dans notre arbre puis en calculer la triangulation de Delaunay. La surface finale sera composée de toutes les arêtes dont les sommets font partis de l'arbre d'origine. Voici en image son déroulement, images issues du cours du professeur Pierre Alliez de l'Inria.

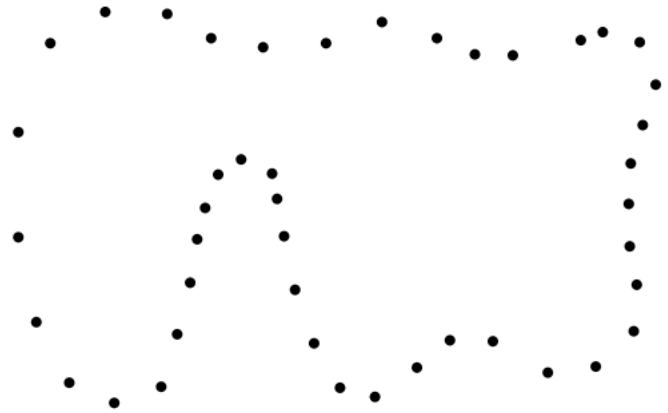

Figure 18: Nuage de point d'origine

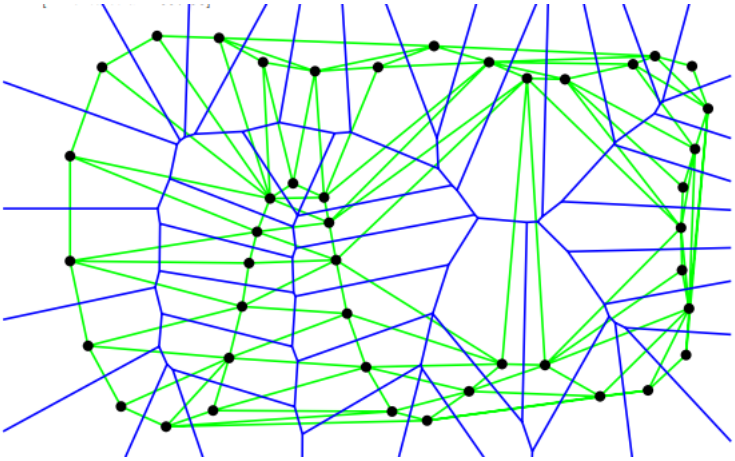

Figure 19: Diagramme de Voronoi(violet) et son dual Delaunay (vert)

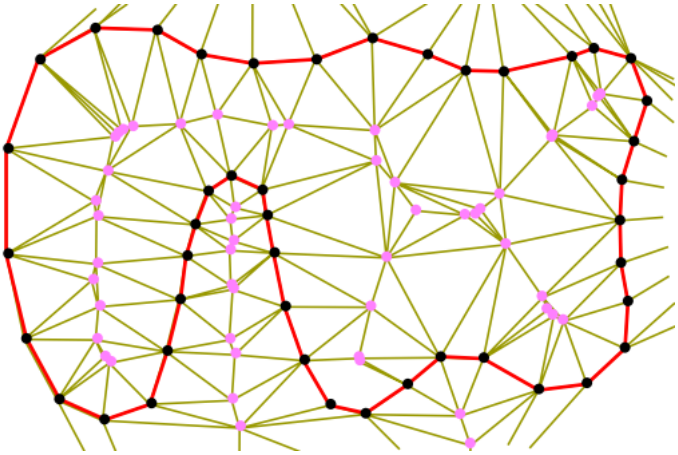

Figure 20: Delaunay réappliqué (jaune) et surface reconstruite (rouge)

L'algorithme que nous avons choisi pour calculer la triangulation de Delaunay est l'algorithme de Bowyer-Watson. Le problème que nous avons rencontré est que celui-ci est fait pour un ensemble de points inclus dans un plan. Hors, notre ensemble de point se trouve dans un espace en 3 dimensions. Nous allons vous expliquer rapidement comment fonctionne cet algorithme, puis les défis d'implémentation en 3 dimensions.

L'algorithme de Bowyer-Watson à un principe similaire à l'algorithme de Prim. On pars d'une triangulation de Delaunay valide (vide si il faut) puis quand on ajoute un point, on fait en sorte que la triangulation reste valide, en ajoutant et supprimant des arêtes. On initialise donc notre triangulation avec un "super triangle" contenant tous les points de notre ensemble. Ensuite, on ajoute un point. Pour chaque triangle de notre triangulation actuelle, on calcule son cercle circonscrit. Si une arête fait partie d'au moins deux triangles dont les cercles circonscrit contiennent notre point, elle est supprimée. Les autres arêtes de ces triangles constituent un polygône dont chaque sera relié par de nouvelles arêtes à notre point, pour créer de nouveaux triangles. Quand tous les points ont étés ajoutés, on supprime de notre triangulation toutes les arêtes dont au moins un des sommets fait partie de notre super triangle initial.

Maintenant en 3 dimensions. Ici on ne parle plus de triangles mais de tetrahèdres. On calcule donc un super tétraèdre, sa sphère circonscrite puis on après avoir définie notre polyèdre, on relie ses arêtes de façon à creér de nouveaux tétraèdres. Au final, on récupère tous les triangles de notre tetraèdrisation qui seront donc notre triangulation.

Pour réaliser cet algorithme, nous avions besoin de plusieurs fonctions auxiliaires. On peut noter une fonction pour calculer le tétraèdre circonscrit à un cube, dans une représentation similaire à celle ci-dessous. Ce n'est pas la représentation la plus optimisée puisque faire tourner le cube peut permettre d'avoir un tétraèdre résultant plus petit.

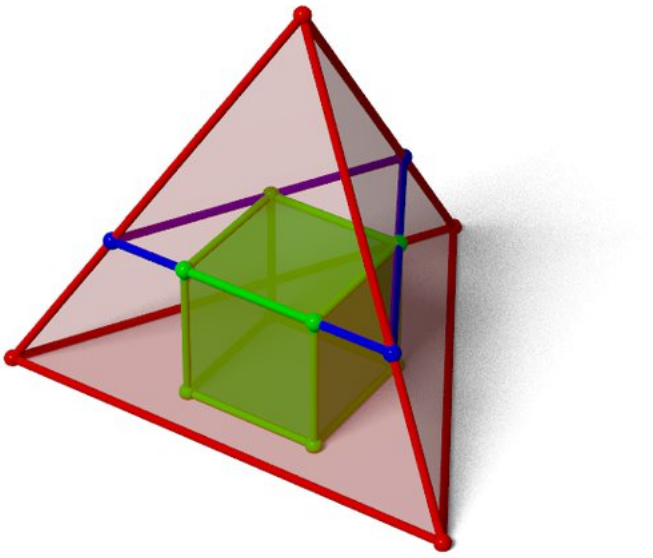

Figure 21: Tétraèdre contenant un cube

Ensuite, nous avions aussi besoin de calculer la sphere circonscrite de nos tétraèdres. Au départ, nous voulions faire ces calculs avec la méthode de Miroslav-Fielder. Elle consiste à

$$\begin{vmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & d_{12}^2 & d_{13}^2 & d_{14}^2 \\ 1 & d_{21}^2 & 0 & d_{23}^2 & d_{24}^2 \\ 1 & d_{31}^2 & d_{32}^2 & 0 & d_{34}^2 \\ 1 & d_{41}^2 & d_{42}^2 & d_{43}^2 & 0 \end{vmatrix},$$

Figure 22: Tétraèdre contenant un cube

prendre la matrice de Cayley-Menger correspondant à notre tétraèdre avec dx,y la distance entre le point x et le point y. Après avoir calculer cette matrice, il faut l'inverser, la multiplier par -2. Avec m la matrices résultante, on obtient le radius de la sphere

$$r = \frac{\sqrt{m_{1,1}}}{2}$$

Et son centre

$$c(x, y, z) = \frac{m_{1,2}P1 + m_{1,3}P2 + m_{1,4}P3 + m_{1,5}P4}{m_{1,2} + m_{1,3} + m_{1,4} + m_{1,5}}$$

Au final, nous avons utilisé d'autres formules. Pour le radius, avec a, b et c les longueurs des trois arêtes qui arrivent sur un sommet, et A, B et C les longueurs de leurs arêtes opposées.

$$R = \frac{\sqrt{(aA+bB+cC)(aA+bB-cC)(aA-bB+cC)(-aA+bB+cC)}}{24V}$$

Et pour le centre :

$$C = A^{-1}B \qquad \text{où} \qquad A = \begin{pmatrix} [x_1 - x_0]^T \\ [x_2 - x_0]^T \\ [x_3 - x_0]^T \end{pmatrix} \qquad \text{et} \qquad B = \frac{1}{2}\begin{pmatrix} \|x_1\|^2 - \|x_0\|^2 \\ \|x_2\|^2 - \|x_0\|^2 \\ \|x_3\|^2 - \|x_0\|^2 \end{pmatrix} \tag{1}$$

$$\tag{2}$$

VIII. Recherche d'un plus court chemin

Initialement, il était prévu que le robot puisse éventuellement se déplacer dans son environnement afin d'y réaliser une reconstruction en 3 dimensions plus étendue. Néanmoins, les contraintes du robot nous ont forcé à devoir rester dans un environnement statique. Le robot ne demeure pas immobile pour autant puisqu'il peut avoir à faire différents mouvements comme des rotations par exemple afin de pouvoir permettre un meilleur rendu en 3 dimensions de l'environnement dans lequel il se trouve. Il pourra également être amené à se déplacer en ligne droite si nécessaire et pourra aller d'un point A à un point B.

Il était initialement prévu de faire une recherche de plus court chemin pour permettre au robot de se déplacer de manière plus rapide dans son environnement. Nous avions pour cela choisi d'implémenter l'algorithme d'Astar (A*) car il était plus adapté pour ce type de recherche de plus court chemin car il utilise des distances réelles. Son fonctionnement permet donc de mieux gérer d'éventuels obstacles pour trouver malgré tout le chemin le plus court chemin grâce aux heuristiques qui représentent des distances à vol d'oiseau.

Pour rappel, nous avions mis en place lors des deux dernières soutenances deux structures de données, une qui correspondaient aux graphes, elle permettait d'initialiser un graph et de faire plusieurs manipulations dessus, notamment l'ajout de noeuds et d'arêtes.
La structure de graph était conçue de la manière suivante :

- Une structure de liste de sommets : elle permettait d'accéder au sommet suivant celui sur lequel on pointe, ainsi que la valeur du sommet pointé (lors d'une recherche de plus court chemin, le graphe doit être valué).
- Une structure pour les listes d'adjacence : elle permet d'accéder à la liste d'adjacence de chaque sommets.
- Une structure de Graphe : elle permet de connaitre l'ordre du graph (son nombre de sommets), s'il est orienté ou non, ainsi que d'accéder à la liste d'adjacence de chaque noeud.

Néanmoins, il s'est avéré que cette structure était obsolète et aurait rendu beaucoup plus compliqué l'implémentation de l'algorithme d'Astar (A*). Par conséquent, nous avons entièrement repensé cette structure de la manière suivante. Elle permet d'initaliser un graphe, ajouter et retirer des noeuds ou des arêtes.
Notre première structure est une structure qui est un couple (x,y) correspondant aux identifiants des noeuds du graph. Il s'agit en fait des coordonnées du point dans l'espace. L'utilisation d'Astar est rendu plus facile avec l'utilisation de points avec des coordonnées :

- Elle permet d'accéder aux coordonnées sur l'axe des abscisses.
- Elle permet d'accéder aux coordonnées sur l'axe des ordonnées

Notre deuxième structure permet de donner des informations sur le graphe :

- Elle permet de savoir si un graphe est orienté ou non.
- Elle permet de savoir si les arêtes du graphe doivent avoir des coûts ou non.
- Elle permet de connaitre l'ordre du graphe.
- Elle permet d'accéder à tous les noeuds qui composent ce graphe (leur identifiant est représenté sous forme de coordonées x et y).

Notre troisième structure permet de donner des informations sur les listes de noeuds.

- Elle permet de connaitre l'identifiant d'un noeud (ce qui correspond à ses coordonnées).
- Elle permet de connaitre les voisins de ce noeud (la liste d'adjacence).
- Elle permet de connaitre le prochain noeud dans la liste.

Notre quatrième et dernière structure permet d'obtenir des informations sur les listes d'adjacences.

- Elle permet de connaitre l'identifiant du noeud (ce qui correspond à ses coordonnées).
- Elle permet de connaitre le coût de l'arc entre un noeud de la liste de noeuds du graph, et un noeud de la liste d'adjacence.
- Elle permet d'accéder à la liste au prochain noeud dans la liste d'adjacence.

Cette structure est plus optimisée et surtout plus adaptée aux besoins du robot. En effet, il était auparavant difficile d'ajouter ou de retirer des noeuds ou des arcs au graphe alors que cela était indispensable de rendre cela possible durant le traitement du robot. Ces fonctions existaient dans notre ancienne structure de données mais rendait très compliqué leur utilisation par le robot, c'est pour cela que nous avons préféré faire une nouvelle structure de données plus optimisée et plus adaptée à nos besoins. Nous avons également adapté cette structure pour rendre plus facile la recherche du plus court chemin avec l'algorithme d'Astar.

Nous avions également mis en place une structure de tas binaire qui nous a servi pour la mise en place de l'algorithme d'Astar.
La structure de tas binaire était conçue de la manière suivante :

- Une structure permettant de gérer les noeuds d'une file : elle permet d'accéder au noeud suivant celui sur lequel on pointe, ainsi que la valeur de ce noeud pointé ainsi que le niveau de priorité de ce dernier.
- Une structure qui gère la file : elle permet d'accéder au premier noeud qui est prêt à sortir de la file, ainsi que la liste de tous les autres noeuds qui suivent.

Nous avons simplement modifié un élément dans cette structure nous permettant de la rendre plus modulable pour l'implémentation de l'algorithme d'Astar. Initialement dans notre heap, nous ajoutions des int, nous avons modifié cela pour pouvoir y ajouter des void* ce qui a rendu plus facile l'implémentation de l'algorithme d'Astar.

Pour l'implémentation de l'algorithme d'Astar (A*), nous avons mis en place deux nouvelles structures. La première structure est une structure de liste de coordonnées. Il s'agit d'une liste chainée.
La seconde est une structure permettant d'obtenir des informations concernant la recherche du plus court chemin avec l'algorithme d'Astar et notamment toutes les informations pour chacun des points du graph :

- Une liste permettant de determiner le plus court chemin pour arriver du noeud source jusqu'au point sur lequel nous pointons.
- Une valeur de distance correspondant à la valeur de la distance permettant d'arriver jusqu'à ce point.
- La valeur de l'heuristique du noeud
- Un pointeur vers le prochain élément de la liste.
- Un booléen permettant de savoir si le noeud a déjà été rencontré ou non.

IX. Site Web

Ce site est le support principal pour retrouver les informations principales liées au projet, ainsi que nos avancées. Il permet à des personnes extérieures de consulter nos différents rapports (Cahier des charges, rapports de soutenance ...).

Comme conseillé lors de la précédente soutenance, nous avons arrêté de faire le site sur la plateforme Wix, et avons développé notre propre site en HTML et CSS. Le site que nous avons développé reste néanmoins visuellement très similaire à celui que nous avions mis en place sur Wix. Il est possible de retrouver sur ce site :

- une page d'accueil avec différentes informations sur le projet.
- une page où sont mentionnés certains composants electroniques utilisés pour la conception du robot.
- une page de téléchargement depuis laquelle il est possible de télécharger les différents documents du projet.
- une page présentant l'avancement du projet ainsi qu'une "foire aux questions" permettant aux utilisateurs d'avoir des réponses à leur question sur notre projet et suite notre avancée (chronologie de réalisation). Ils trouvenront également quelques vidéos de nos premiers tests du robot.
- une page de présentation de l'équipe où sont mentionnés les différentes tâches de chacun.

Compte tenu des coûts assez élevés engendrés par la conception du robot, nous avons pris la décision de ne pas heberger le site sur un hebergeur classique, ni de payer un nom de domaine afin de limiter nos coûts. Le site est par conséquent hebergé sur la plateforme github.io.
Le site est accessible en cliquant ici

IV. Coûts

En ce qui concerne les coûts, nous avons du acheter du matériel (environ 250 €) et notamment :

- 4 ESP 32 : 32€
- 6 TOF 4M : 22.6 €
- (4 TOF 2M : 13,92€)
- 1 bobine PETG : 20 €
- 1 bobine TPU : 20 €
- 5 steppers drivers : 12 €
- 1 ruban reflechissant : 16 €
- 4 breadbords : 13 €
- Des cables d'alimentations : 13 €
- 5 servos : 9 €
- 5 modulateurs de tension : 12 €
- Différents cables : 15 €
- Colle pour pistolet à colle : 17.8 €
- Pinces crocodiles : 3.45 €
- Collier de serrage : 1.5 €
- Isolant electrique : 1.3 €
- Grip : 7.49 €
- Roues pour le robot : 19.98 €

V. Conclusion

Le projet que nous avons réalisé consiste en la réalisation d'un robot de reconstruction tridimensionnelle d'un environnement. L'environnement du robot doit être statique, bien que le robot sera nécessairement amené à faire quelques mouvements pour pouvoir faire cette reconstruction.

Nous avons mis en place plusieurs structures pour le développement de ce projet. Nous avons utilisé des fichiers .c et .h pour le développement en C, des fichiers .ino pour les fichiers arduino et des fichier .cpp pour les fichiers en C++.

L'imprimante 3D d'Alexis peut faire partie intégrante des membres du groupe car elle nous a permis de concevoir notre robot. Ses quelques dizaines d'heures d'impression nous auront bien aidé dans la conception de ce projet.

Nous étions conscients de l'ambition de ce projet et de la difficulté que cela représentait. Même si nous n'avons pas pu réaliser tout ce que nous souhaitions, nous restons tout de même satisfait du résultat du projet que nous sommes en capacité de présenter.

Annexe

Vous trouverez ci-dessous quelques images du robot final :

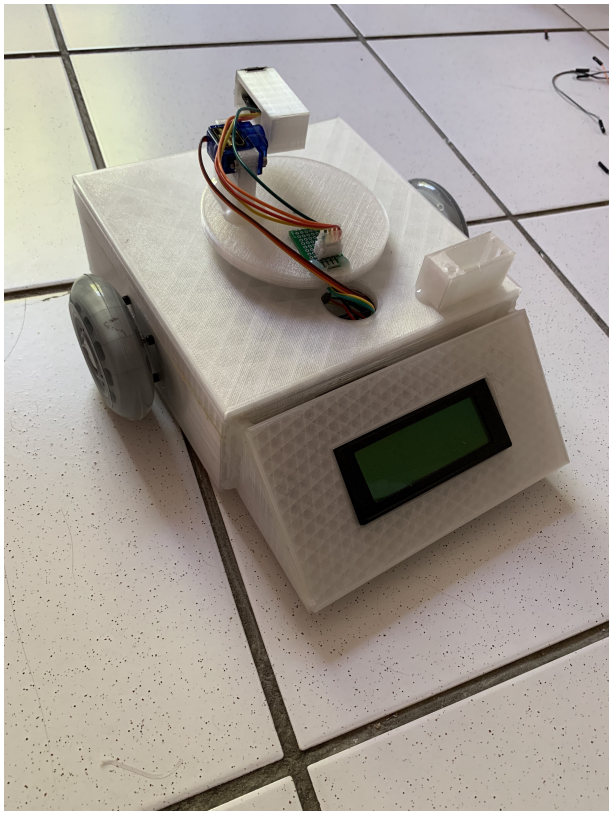

Figure 23: Photo du robot final

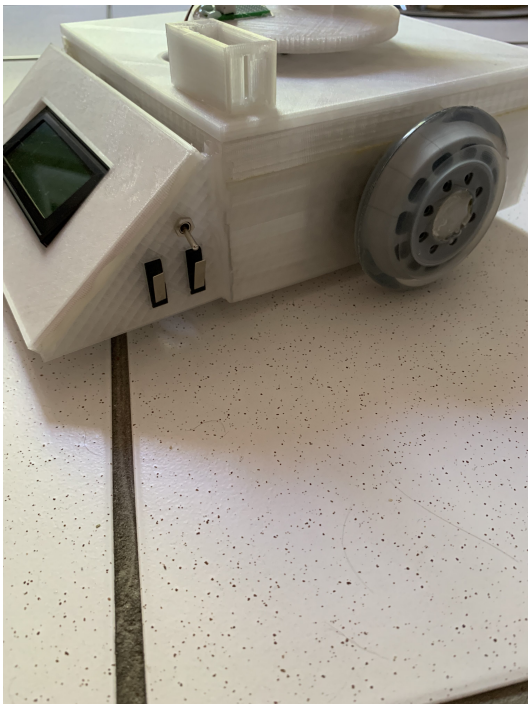

Figure 24: Photo du robot final

Figure 25: Photo du robot final