Thymotel

Louis Heredero
&
Mathéo Beney

18 mars 2022

OC Informatique 4ème

Table des matières

Table des figures

Chapitre 1

Introduction

Le sujet de notre projet est la création d'un hôtel pour thymio ainsi que d'un ascenseur permettant aux robots d'aller aux chambres. L'idée originelle était de créer un trieur de legos par tapis roulant, mais nous avons estimé pouvoir réaliser quelque chose de plus ambitieux.

C'est ainsi que le Thymotel est né. La structure fut découpée par laser dans des planches de peuplier puis assemblée et collée. Il y a aussi une partie faite en legos, comprenant notamment le système de poulies et de palans.

L'un des deux robots est appelé "serveur" et est fixé à une plaque. Grâce à ses roues et à des ficelles, il peut actionner l'ascenseur. Avec sa roue gauche, il contrôle le mouvement horizontal et avec celle de droite le mouvement vertical. L'autre robot est le "client". Il commence par scanner un code-barres, lui indiquant son numéro d'identification. Il se dirige vers le serveur et lui communique son identifiant. Le serveur essaie ensuite de lui attribuer une chambre vide, lui indiquant que l'hôtel est plein le cas échéant. Après cet échange d'informations, le client monte dans la cabine de l'ascenseur, puis le serveur la déplace devant la bonne chambre. Finalement, le client avance et la cabine revient au départ.

Chapitre 2

Code

2.1 Côté client

2.1.1 Fonctionnement général

Le client commence sur une ligne noire, qu'il suit jusqu'à un code-barres fait en legos. Il utilise pour cela les deux capteurs infrarouges du dessous avec ce code.

```
#Both white
when prox.ground.delta[0] >= 450 and prox.ground.delta[1] >= 450 do
    if line_side == -1 then
        motor.left.target = -TURN_SPEED
        motor.right.target = TURN_SPEED
    elseif line_side == 1 then
        motor.left.target = TURN_SPEED
        motor.right.target = -TURN_SPEED
    end
end

#Left black
when prox.ground.delta[0] <= 400 and prox.ground.delta[1] >= 450 do
    motor.left.target = LINE_SPEED
    motor.right.target = TURN_SPEED
    line_side = -1
end

#Right black
when prox.ground.delta[0] >= 450 and prox.ground.delta[1] <= 400 do
    motor.left.target = TURN_SPEED
    motor.right.target = LINE_SPEED
    line_side = 1
end

#Both black
when prox.ground.delta[0] <= 400 and prox.ground.delta[1] <= 400 do
    motor.left.target = LINE_SPEED
    motor.right.target = LINE_SPEED
    line_side = 0
end
```

Figure 2.1 – Suivi de la ligne

Il monte ensuite sur le code-barres et détecte les changements d'angle pour savoir lorsqu'il est de nouveau à plat.

La lecture du code se fait de la manière suivante. Une première plaque noire permet de définir la longueur d'une barre. Celle-ci est suivie d'une plaque blanche qui délimite le début du code. Les barres suivantes représentent la valeur du code en binaire sur 6 bits, où le noir correspond à un 1 et le blanc à un 0.

Le robot calcule les longueurs des différentes zones de couleurs et les divisent par la longueur d'une plaque. (cf. section 2.1.2) Afin d'obtenir un résultat optimal, le robot doit préalablement calibrer les capteurs aux couleurs des legos. Finalement, il convertit la valeur binaire en décimal grâce à la sous-routine suivante.

```
sub bin_to_dec
    dec_val = 0
    for k in 1:BITS do
        dec_val += code[k-1] << (k-1)
    end
```

Figure 2.2 – Binaire -> décimal

Une fois le code scanné, le robot redescend puis suit la ligne en direction du serveur. S'ensuit une discussion par infrarouges entre les deux robots.

Afin qu'ils se comprennent, nous avons dû établir un protocole comme décrit dans l'annexe A.

Tout d'abord, le client émet en continu le message de "handshake". Dès que le serveur le détecte, le processus de communication est entamé (cf. annexe B).

Si le serveur est en train de déplacer l'ascenseur, il renvoie la commande "moving", indiquant au client de s'arrêter mais de continuer à émettre le "handshake".

Si après envoi de l'id, le serveur détermine qu'aucune chambre n'est libre, alors il renvoie la commande "goodbye" et le client se met en rouge.

Finalement, si tout se passe correctement, le client s'allume en vert, continue sa route et monte sur la rampe. Il s'arrête un peu après avoir détecté un retour à l'horizontale.

Lorsque l'ascenseur est en mouvement, le client lance un "timeout" qui se réinitialise à chaque fois qu'il détecte un mur passant devant lui. Quand se "timeout" se termine, le client avance pour entrer dans la chambre et s'arrête dès qu'il ne détecte plus de sol.

2.1.2 Code-barres

Dans cette section, nous allons détailler la lecture du code-barres. Le schéma suivant explique la procédure de manière graphique :

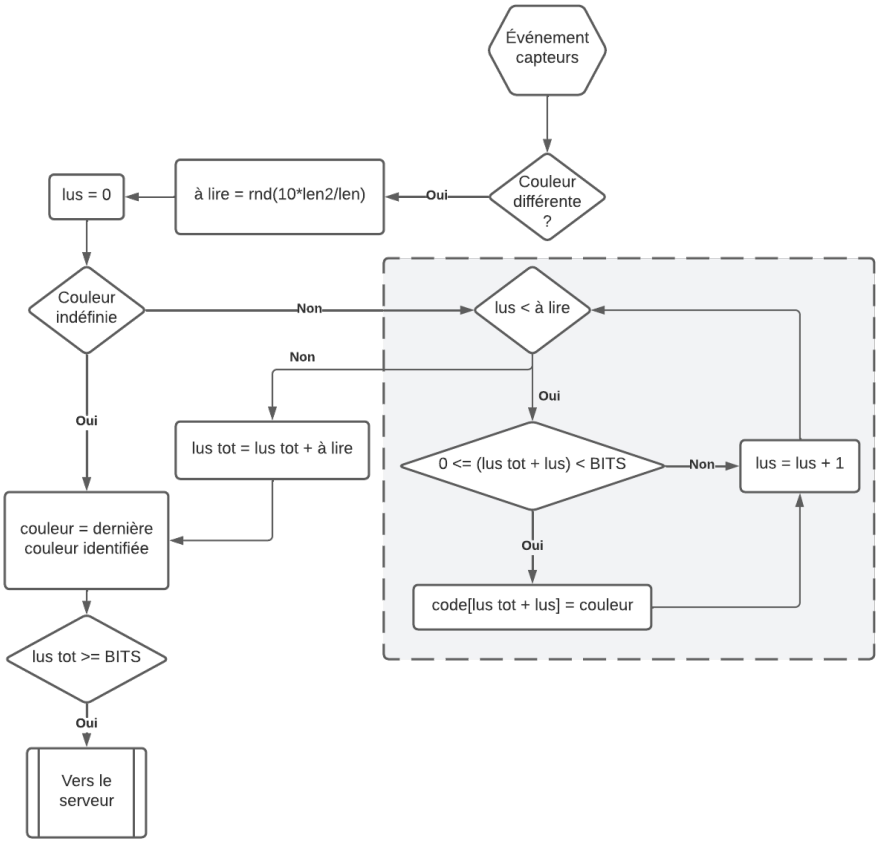


Figure 2.3 – Lecture du code-barres (schéma)

Le point d'entrée est l'événement `prox` qui est émit en boucle lorsque le robot avance. La couleur sous le robot est identifiée grâce à la sous-routine `get_avg` (figure 2.4)

```
#Calcule la couleur moyenne sous le robot
sub get_avg
    avg = (prox.ground.delta[0]+prox.ground.delta[1])/2
    callsub get_col

#Définit s'il s'agit de blanc, noir ou autre
sub get_col
    if black == 0 or white == 0 then
        col = -1

    else
        #10 *... *10 pour rester entre -32768 et 32767
        col = 10*(avg-black)/(white-black)*10

        if col < 40 then
            col = 1
        elseif col > 70 then
            col = 0
        else
            col = -1
        end
    end

    if col != -1 then
        last_col = col
    end
```

Figure 2.4 – Calcul de la couleur captée

Le robot effectue la lecture à proprement dit à chaque changement de couleur (lorsqu'il passe d'une plaque blanche à une noire et inversément). Sur la figure 2.3, `len` est la longueur d'une plaque et `len2` est la distance parcourue depuis le dernier changement de couleur. Ainsi, `rnd(len2/len)` est le nombre de plaques constituants cette zone (où `rnd` est la fonction arrondi).

Si la nouvelle couleur que le robot voit a pu être identifiée, on entre dans la boucle du cadre gris.

Cette boucle rajoute bit à bit la valeur de la couleur dans la liste du résultat, tant que le nombre de plaques lues ne dépasse pas la longueur du code-barres (`BITS` [1]). Il s'agit surtout d'une précaution au cas où il ne détecterait pas correctement toutes les plaques lego.

1. Dans notre cas, `BITS=6`

Finalement, le nombre de bits lus jusqu'ici est mis à jour selon le nombre de valeurs ajoutées à cette étape.

Enfin, si le code-barres a été entièrement lu, c'est à dire que le nombre de plaques comptées est supérieur ou égal à la longueur du code, alors le robot passe à l'état `sTO_SERVER`.

Ce qui traduit en Aseba donne :

```
elseif state == sREAD then
    if col != cur_col then
        to_round = 10*length2/length
        callsub round

        length2 = rounded

        j = 0
        if cur_col != -1 then
            while j < length2 do
                if i+j >= 0 and i+j < BITS then
                    code[i+j] = cur_col
                end
                j++
            end
            i = i+length2
        end
        cur_col = last_col
        length2 = 0

        if i >= BITS then
            state = sTO_SERVER
            callsub bin_to_dec

            id = dec_val
            emit result code

            call prox.comm.enable(1)
            prox.comm.tx = HANDSHAKE
        end
    end
```

Figure 2.5 – Lecture du code-barres (Aseba)

2.1.3 Communication

Cette brève section à pour but d'expliquer les bases du protocole de communication. Pour une meilleur compréhension, veuillez vous référer à l'annexe A.

L'environnement Aseba permet de transmettre des valeurs de 10 bits d'un robot à l'autre via les capteurs de proximité, c'est à dire un nombre entre 0 et 1023 inclus. Nous avons décidé d'utiliser 4 bits pour définir la commande et les 6 autres pour les données associées.

Prenons l'exemple de la commande "send id", envoyée par le client pour transmettre son identifiant au serveur. L'id de la commande est 2_d ou 0010_b [2].

Pour former le message complet, il faut décaler ce nombre de 6 bits vers la gauche pour laisser la place aux 6 bits de données. En Aseba, comme dans de nombreux langages, l'opérateur de décalage binaire est << suivi du nombre de bits à décaler.

Ainsi,

$$2_d = 0010_b$$

$$2_d << 6 = 0010_b << 6 = 00\ 1000\ 0000_b = 128_d$$

On y ajoute l'identifiant du robot convertit en décimal. Cet identifiant étant encodé sur 6 bits, il est limité à une valeur inférieure ou égale à 63. Prenons l'id 43_d ($10\ 1011_b$), on a donc :

$$\begin{array}{r} 00\ 1000\ 0000 \\ +\ 00\ 0010\ 1011 \\ \hline 00\ 1010\ 1011 \end{array}$$

Le client 43_d voulant communiquer son identifiant au serveur enverra donc le message $00\ 1010\ 1011$.

Pour décoder un message reçu, le processus est très simple. Continuons avec notre exemple mais du point de vue du serveur.

2. x_d est la représentation décimale et x_b la représentation binaire

Pour extraire la commande, il suffit de redécaler le message vers la droite de 6 bits. Ainsi,

$$\begin{aligned} 00\ 1010\ 1011\ (>>\ 0) \\ 0\ 0101\ 0101\ (>>\ 1) \\ 0010\ 1010\ (>>\ 2) \\ 001\ 0101\ (>>\ 3) \\ 00\ 1010\ (>>\ 4) \\ 0\ 0101\ (>>\ 5) \\ 0010\ (>>\ 6) \end{aligned}$$

on retrouve bien $0010_b = 2_d$, la commande "send id".

Pour isoler les données, on peut utiliser ce qui s'appelle un masque avec l'opération "et" binaire. L'opérateur $\&$(ou "et" binaire) applique la table de vérité suivante sur chaque les bits de deux nombres deux à deux.

A	B	$A\&B$
0	0	0
0	1	0
1	0	0
1	1	1

Pour conserver les 6 bits les moins significatifs (càd à droite), il faut faire $message\ \&\ 11\ 1111_b$, dans notre cas,

$$\begin{array}{r} 00\ 1010\ 1011 \\ \&\ 00\ 0011\ 1111 \\ \hline 00\ 0010\ 1011 \end{array}$$

On obtient ainsi $10\ 1011_b = 43_d$, l'identifiant du client.

2.2 Côté serveur

Le programme côté serveur est séparé en 2 parties qui sont respectivement séparées en 2 et 4 états.

2.2.1 Communication

La section communication est séparée en 2 états. Le premier consiste uniquement à attendre de recevoir un handshake (cf. annexe B) et de passer à l'état suivant qui lui s'occupe de toute la partie échange d'informations. Les échanges sont gérés par cette section du programme :

```
onevent prox.comm
    callsub read_ir
    if  state ==WAITING_FOR_CLIENT and prox.comm.rx == HANDSHAKE then
        prox.comm.tx = ASK_ID <<6
        state = PROCESSING_CLIENT

    elseif  state>PROCESSING_CLIENT and prox.comm.rx  == HANDSHAKE then
        prox.comm.tx = MOVING <<6

    elseif  state== PROCESSING_CLIENT and ir_cmd == SEND_ID  then
        callsub databasecheck
        if  room_index != -1 then
            prox.comm.tx = (ROOM_NUM <<6)+ room_index
        else
            prox.comm.tx = GOODBYE <<6
            state = WAITING_FOR_CLIENT
        end

    elseif state == PROCESSING_CLIENT and ir_cmd == ASK_TIME then
        callsub calculatetime
        prox.comm.tx = (WAITING_TIME<<6)+wtime
        state = ENTER_LIFT
        timer.period[0] = 20000
    end
```

Figure 2.6 – Partie communication

On peut voir qu'il y a un appel à la sous-routine nommée `databasecheck`. Cette sous-routine s'occupe de tout ce qui concerne la database, qui est une liste avec 16 éléments représentant les 16 chambres. Si la valeur est de -1 la chambre est libre et si la chambre est occupé, le numéro du client remplace le -1. La numérotation des chambres choisie est celle-ci :

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Nous avons opté pour cette numérotation car elle nous permet de facilement récupérer les coordonées de la chambres comme ceci :

```
sub get_roomcoo
    room_coo[0] = room_index%NB_COLUMNS
    room_coo[1] = room_index/NB_ROWS
```

Figure 2.7 – Coordonées de la chambre

Voici un schéma de la fonction `databasecheck` :

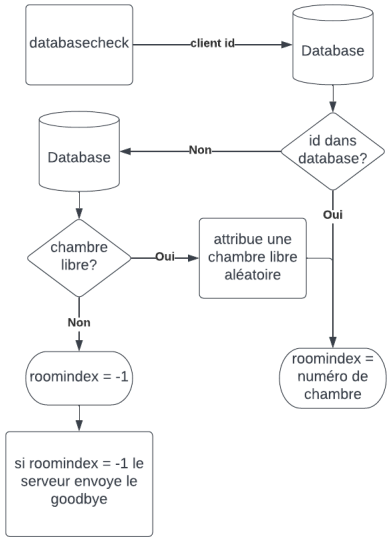


Figure 2.8 – Schéma de la sous-routine `databasecheck`

Le code en aseba de cette fonction est disponible dans l'annexe C.

Une fois la communication terminée le serveur laisse 20 secondes au client pour entrer dans l'ascenseur, après quoi la fonction `start_moving` est appelée.

Cette fonction est utilisée pour initialiser les différentes valeurs (`htime`, `vtime`, `hor`, `ver`, `direction`). L'ascenseur commence par aller à l'horizontale puis à la verticale et pour le retour, le mouvement est inversé. Nous avons dû utiliser le temps moyen par étage/colonne car en aseba les nombres sont des entiers signés sur 16 bits (valeur maximum de 32767) et que les

`timers` sont en millisecondes celà nous laisse un timer maximum de 32 secondes or, nous dépassons ce temps.

Voici la fonction en question avec une fonction similaire pour le retour :

```
sub start_moving
    direction = 1
    callsub get_roomcoo
    emit print2 room_coo
    ver = room_coo[1]
    hor = room_coo[0]

    if  ver > 0 then
        vtime = (ver_time[ver]*10)/ver
    end

    if  hor > 0 then
        htime =(hor_time[hor]*10)/hor
        emit print htime
        callsub start_hor

    elseif ver > 0 then
        callsub start_ver

    else
        state = LEAVING_LIFT
        timer.period[0] = 25000
    end
```

```
sub start_returning
    direction = -1
    ver = room_coo[1]
    hor = room_coo[0]
    if  hor > 0 then
        htime = (hor_return_time[hor]*10)/hor
    end

    if  ver > 0 then
        htime =(ver_return_time[ver]*10)/ver
        emit print htime
        callsub start_ver

    elseif hor > 0 then
        callsub start_hor
    else
        state = WAITING_FOR_CLIENT
    end
```

Figure 2.9 – Fonction `start_moving` et `start_returning`

Les temps verticaux et horizontaux d'aller et de retour ont été calculés grâce au programme de calibration (`calibration.aesl`)

Les fonctions `start_ver` et `start_hor` servent uniquement à démarrer les moteurs et lancer les `timers`. La variable direction sert à changer entre l'aller (= 1) et le retour (= -1).

Voici ce que celà donne en aseba :

```
sub start_ver
    motor.right.target = -MOTORSPEED*direction
    motor.left.target = 0
    timer.period[0] = vtime*10

sub start_hor
    motor.right.target = MOTORSPEED*direction
    motor.left.target = MOTORSPEED*direction
    timer.period[1] = htime*10
```

Figure 2.10 – Fonction `start_ver` et `start_hor`

Ensuite tout le reste du déplacement se déroule dans les `timers`. Le `timer0` sert au mouvement vertical et le `timer1` l'horizontal. Le `timer0` s'occupe aussi de tous les autres moments d'attente (entrée et sortie de l'ascenseur).

À la fin de l'aller l'ascenseur laisse 25 secondes au client pour sortir avant de repartir. Une fois l'ascenseur retourné au point de départ le serveur est prêt à recevoir un nouveau client.

Lorsque le serveur est occupé à déplacer l'ascenseur et qu'un nouveau client arrive, il lui envoie un message pour lui dire d'attendre.

Chapitre 3

Déroulement

3.1 Préparation

Après avoir trouvé l'idée générale, nous avons commencé à esquisser quelques concept pour l'hôtel et l'ascenseur. Pour la structure de l'hôtel, nous avions d'abord pensé à la faire entièrement en legos mais cela n'aurait pas été réalisable. On a ensuite pensé à des profilés en aluminium mais ce n'était pas dans notre budget. Nous avons donc décidé de découper l'hôtel dans du bois.

Nous avons d'abord modélisé grossièrement la structure sur Blender [1], puis dessiné sur Inkscape [2] les fichiers de découpe. Nous sommes ensuite allés aux FabLab de Sion [3] pour utiliser la découpeuse laser et créer un prototype du premier module, que l'on peut voir sur la figure 3.2.

1. https://www.blender.org/
2. https://inkscape.org/
3. https://fablabsion.ch/

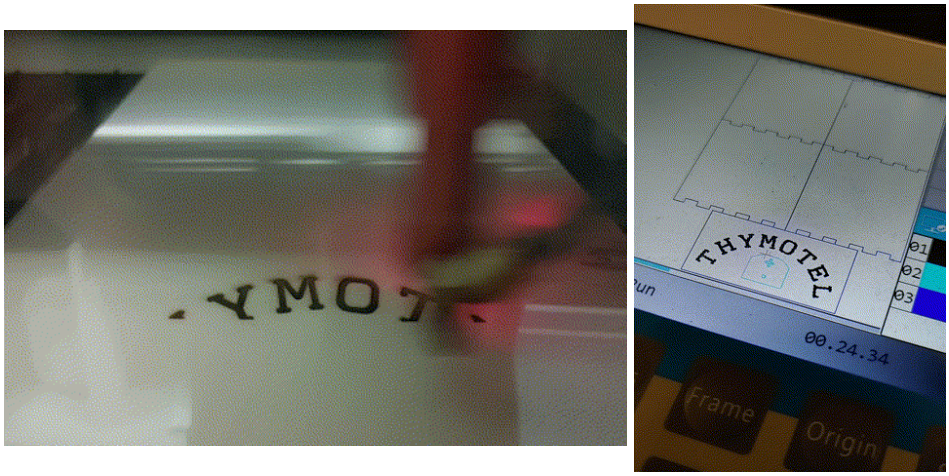


Figure 3.1 – Découpe au laser

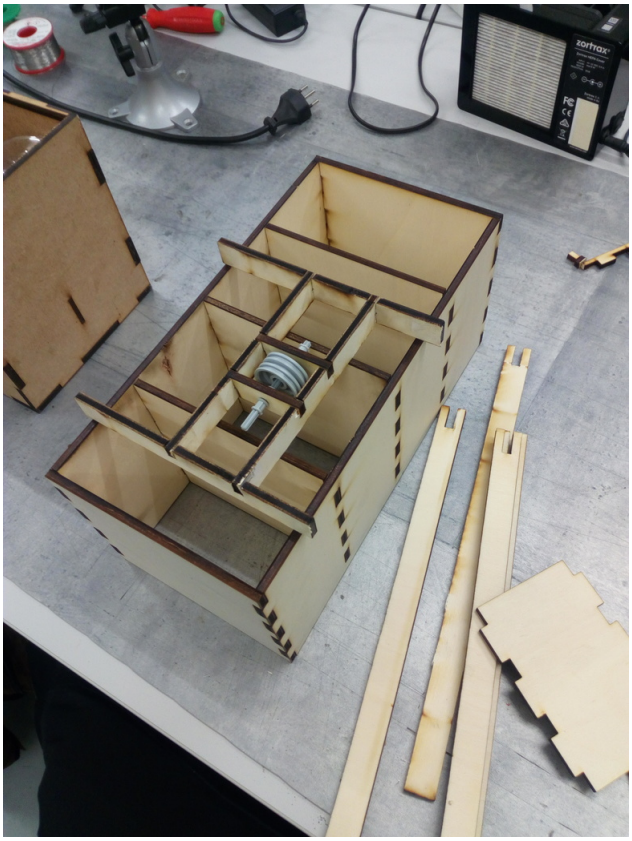


Figure 3.2 – Prototype d'un module

Quant à l'ascenseur, nous avons imaginé de nombreuses possibilités. Nous avions par exemple pensé à le poser sur des rails, à le tirer par en-bas ou même à le suspendre par des cables. Voici quelques croquis des différentes itérations du design de l'ascenseur.

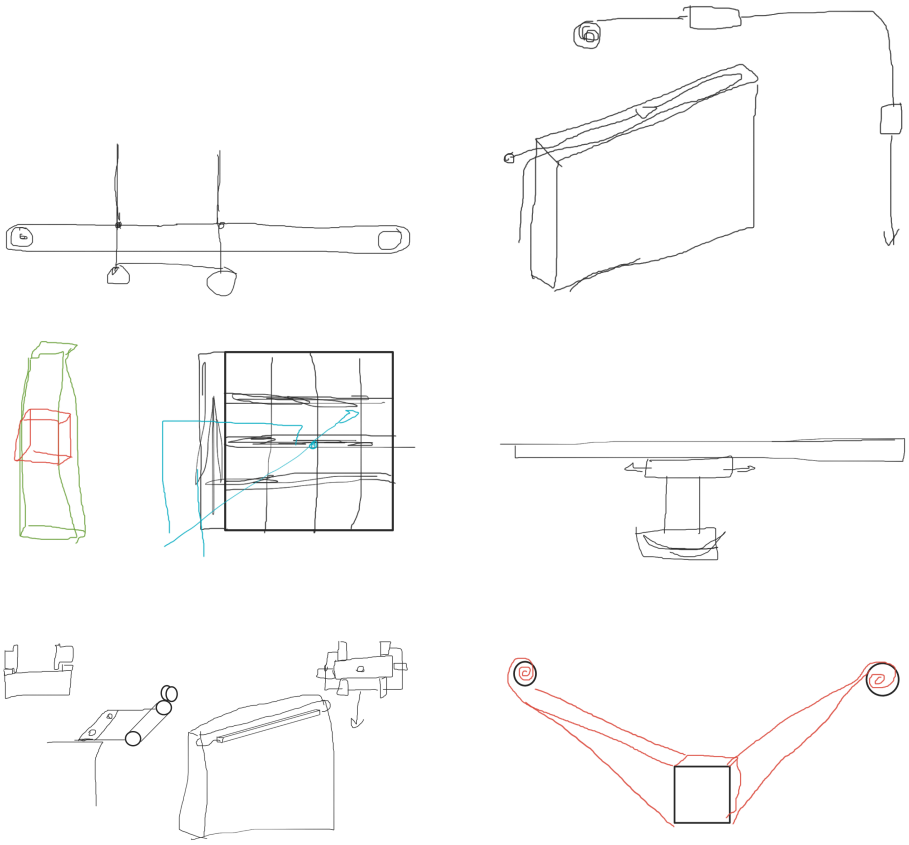


Figure 3.3 – Croquis de brainstorming

Nous avons finalement opté pour une tour montée sur roue, tirée par le haut, dans laquelle se déplace verticalement une cabine.

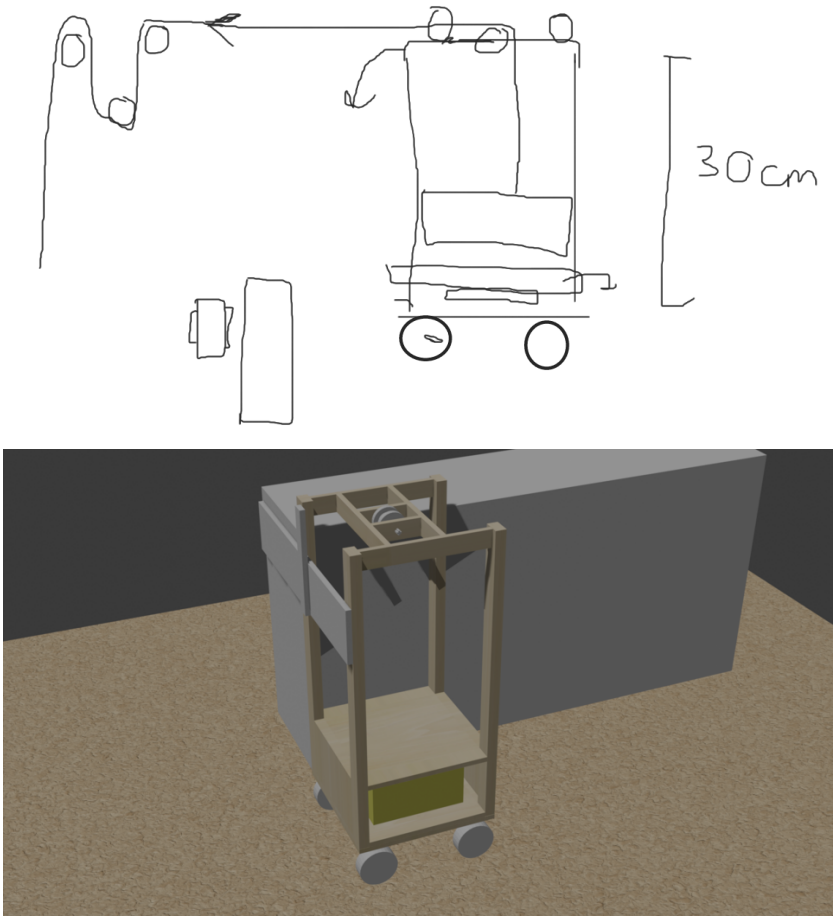


Figure 3.4 – Modèle ascenseur

Comme on le voit sur le croquis de la figure 3.4, la hauteur importante de la tour et les forces en jeu auraient fait basculer la structure. Afin de résoudre ce problème et pour permettre à l'ascenseur de se déplacer vers la droite, nous avons dû installer un contrepoid tirant dans le sens opposé aux deux autres ficelles.

Du point de vue de la programmation, nous avons aussi préparé le terrain en établissant des organigrammes pour représenter l'enchaînement des étapes et des états, comme celui de l'annexe B.

Nous avons aussi défini le protocole de communication (annexe A).

3.2 Réalisation

Après avoir découpé les différents éléments en bois, nous les avons assemblés et collés.

Figure 3.5 – Assemblage

L'assemblage ne nous a pas causé trop de problème car nous avions tout prévu à l'avance. Malheureusement, nous avons fait face à une petite difficulté. Les tiges de soutient de la cage d'ascenseur avaient une surface de contact avec la base qui était trop petite pour que la colle les tiennent. Nous avons donc dû les visser pour que la structure résiste aux tensions.

Ensuite, nous avons planifié et exécuté tout le placement des fils. Cette étape fut assez rapide mais elle a engendré plusieurs problèmes. Premièrement, la stabilité de l'hôtel n'était pas suffisante à cause de l'effet de levier et de forces trop grandes dans les fils. Pour y remédier nous avons ajouté des tiges legos de chaque côté, agissant comme des piliers. Nous avons aussi rigidifier la structure avec des ficelles. Deuxièmement, nous avons dû renforcer les attaches de certaines poulies puisque les legos ne tenait pas suffisamment.

Ensuite, nous avons rencontré certains problèmes pour déplacer l'ascenseur de façon fiable, notamment à cause des incertitudes physiques. C'est pour cela que nous avons créé un système de calibration pour être plus précis.

3.2.1 Organisation agile/adaptive

Nous avons opté pour une organisation dite agile durant tout le projet. Cela veut dire que nous avons établi plusieurs étapes que nous avons réalisé séparément pour ensuite les assembler. Ces étapes étaient les suivantes :

- lecture du code-barres
- déplacement du robot (ligne + rampe)
- communications
- déplacement de l'ascenseur avec une chambre donnée

Cette façon de construire notre projet nous a permis de tester et débugger chaque partie individuellement et d'y apporter les modifications nécessaires (par exemple le système de calibration). Une fois chaque partie individuelle fonctionelle, nous les avons assemblées sans rencontrer d'obstacle. Cela nous a aussi permis de commencer le projet bien avant d'avoir fini la structure et de facilement pouvoir séparer les différentes tâches à faire.

Au niveau de la répartition des éléments, c'est Louis qui s'est chargé des modélisations, des dessins et du côté client. C'est Mathéo qui a développé la partie serveur et qui s'est aussi occupé de la majorité de l'assemblage de l'hôtel et des tests.

Chapitre 4

Conclusion

Pour nous, cet exercice fut très enrichissant et intéressant. Nous sommes fier du projet à son état actuel mais si nous devions le refaire voici les changement que nous apporterions.

Premièrement, nous changerions la ficelle pour une qui s'étire moins. En effet, nous pensons que les imprécisions du système sont du aux petites variations de longueur des ficelles. Le système d'enroulage des ficelles pourrait aussi faire une grande différence car actuellement, la vitesse d'enroulage/déroulage varie un peu selon la quantité de ficelle enroulée autour de l'axe.

Les points importants que nous avons appris sont les suivants :

- prendre le temps d'envisager plusieurs possibilités et d'aborder les problèmes sous différents angles.
- bien planifier avant de commencer quoi que ce soit, afin d'éviter les potentiels obstacles importants
- structurer le code de manière modulable, séparé en plusieurs étapes et réutilisable

La multidisciplinarité de notre projet (mélange entre technique, physique et informatique) fut aussi un point très intéressant pour nous.

Nous avons eu beaucoup de plaisir a réaliser ce projet. La souplesse des consignes nous a permis de laisser libre cours à notre créativité pour atteindre une certaine satisfaction personnelle.

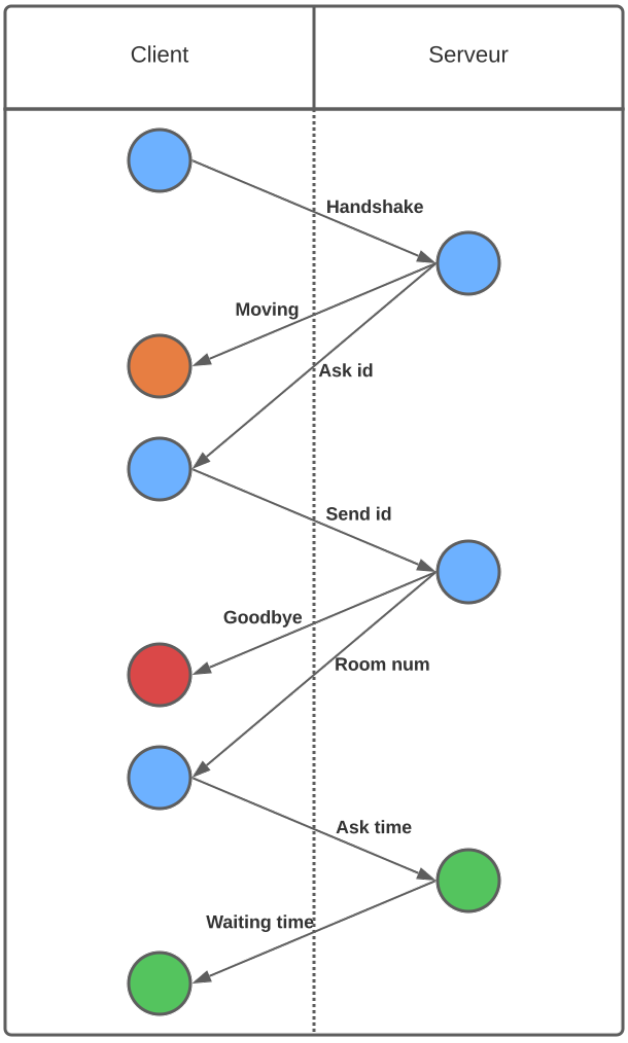
Annexe A

Protocole de communication IR

	9	8	7	6	5	4	3	2	1	0
	Cmd				Data					
Handshake	1	1	1	1	1	1	1	1	1	1
Ask id	0	0	0	1	-					
Send id	0	0	1	0	id (max 63)					
Goodbye	0	0	1	1	-					
Room num	0	1	0	0	room num (max 63)					
Ask time	0	1	0	1	-					
Waiting time	0	1	1	0	half time in seconds					
Moving	0	1	1	1	-					

Annexe B

Séquence de communication



Annexe C

Sous-routine `databasecheck`

```
#regarde si l'id dans num_database a une chambre et change la valeur de is_in_data_base si nécessaire
sub indatabase
    is_in_data_base = 0
    for i in 1:NB_ROOM do
        if num_database == data_base[i-1] then
            is_in_data_base = 1
            # met l'index de la première apparition du nombre dans la variable roomindex
            room_index = i-1
            return
        end
    end

#regarde ce que le robot doit faire après avoir reçu l'id
sub databasecheck
    room_index = -1
    num_database = ir_data
    callsub indatabase
    if is_in_data_base == 0 then
        length = 0
        for i in 1:NB_ROOM do
            #crée une liste d'index avec les chambres disponible et garde la longueur de la liste
            if -1 == data_base[i-1] then
                free_room[length]= i-1
                length += 1
            end
        end
        if length > 0 then
            # génère un nombre entre 0 et length-1
            call math.rand(new_room)
            new_room = abs(new_room%length)
            # attribut l'id à la chambre aléatoire
            data_base[free_room[new_room]] = ir_data
            room_index = free_room[new_room]
            emit printlist_16_ data_base
        end
    end
```