Rapport de soutenance 2 - Semestre 4 - EPITA

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

Avril 2022

Table des Matières

I. Introduction

I. Rappel du sujet

L'objectif du projet est de modéliser l'environnement **statique** en 3D d'un robot grâce à un capteur de distance. On pourra se déplacer dans cette modélisation pour pouvoir découvrir l'environnement. Il sera possible d'ordonner au robot de se déplacer vers un point précis, il pourra alors prendre le plus court chemin pour s'y rendre et continuer à modéliser l'environnement à partir de cet endroit. Le logiciel pourra sauvegarder la modélisation afin de pouvoir la recharger dans le futur au besoin.

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 des tâches

I. Mise en place de l'ESP32

I - Le Capteur

Pour le capteur, nous avons conçu un support permettant de fixer nos 5 capteurs et de pouvoir les faire tourner à 360° afin de pouvoir capter tout notre environnement.

Nous utiliserons un moteur pas à pas et un servo-moteur, permettant à la fois d'avoir une vitesse de rotation rapide, mais aussi de récupérer les informations de positions facilement.

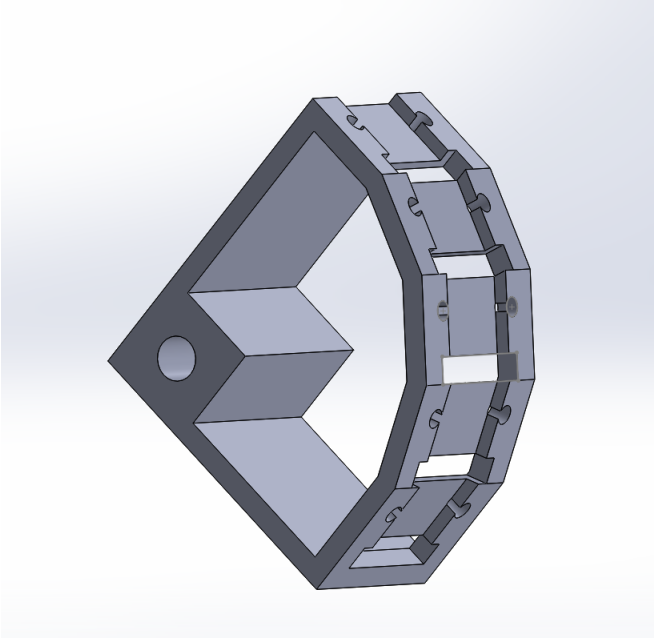

Figure 1: Le support du capteur

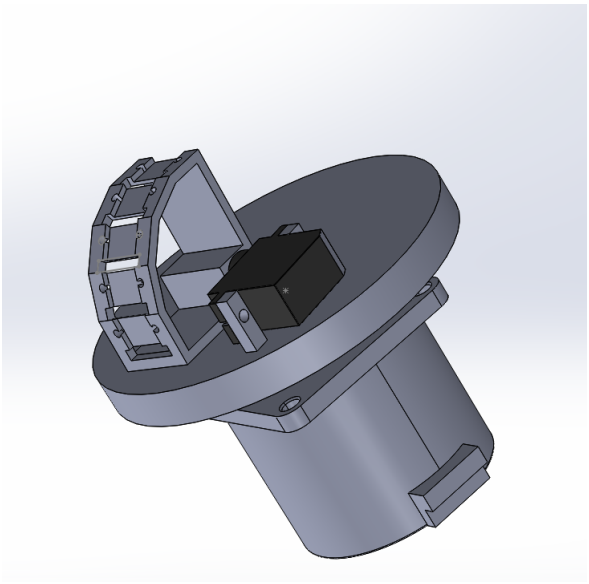

Figure 2: Le capteur entier

II - Passage à Arduino

Pour des raisons de simplicité nous avons décidé d'utiliser Arduino pour coder sur l'ESP-32. Arduino est une plateforme de prototypage Open-Source, qui est basé sur les langages C et C++. De ce fait, nous pouvons utiliser notre code en C et l'adapter facilement à l'ESP, notamment pour la gestion du réseau. Arduino utilise les mêmes fonctions de compilations que nous utilisions avant pour l'ESP, mais viens rajouter une surcouches de librairies venant nous aider fortement dans la mise en place de notre code.

III - Librairies

Arduino fonctionnant beaucoup avec des librairies, Arduino.h est la librairie faisant le lien entre la plateforme et le code C compilé sur l'ESP. Nous utiliserons plusieurs librairies, certaines fournies de bases par Arduino (Servo.h, WiFi.h, etc.). Mais nous utiliserons aussi des librairies trouvées sur internet comme celles des capteurs et finalement des librairies codées par nous même, comme nous comptons le faire pour la mise en place du réseau entre le robot et la machine.

II. Récupération des données capteur

I - I2C

Comme dit dans les anciens rapport, les capteurs que nous utilisons (VL53L1X / VL53L0X) se servent d'une interface I2C pour communiquer. Cette interface possède plusieurs avantages et inconvénients.

Tout d'abord elle est très simple à mettre en place, en effet 2 fils sont nécessaires pour mettre en place une interface I2C, un bus d'horloge et un bus de donnée, et chaque éléments à mettre sur l'interface viens 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, pour nos capteurs, l'adresse est là même à la base et stocké sur une mémoire programmable qui se reset quand on coupe l'alimentation. De ce fait nous avons besoin d'allouer une adresse différente à chaque capteur à chaque démarrage du robot, pour cela nous faisons en sorte d'allumer les capteurs un par un pour pouvoir changer leur adresse (car si ils sont tous allumer ils vont tous changer d'adresse en même temps).

Une fois chaque capteur correctement adressé, il suffit de les appeler avec leurs adresses via le bus I2C pour récupérer les informations dont nous avons besoin.

II - VL53L1X.h

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.

III. Le robot

Pour le robot, nous avons commencé à le conceptualiser, car notre priorité est d'abord d'avoir un capteur complètement fonctionnel, cependant il fonctionnera sur le même principe qu'un aspirateur robot classique, avec 2 roues motorisées sur les cotés et 2 roulettes à l'avant et à l'arrière, permettant de simplement contrôler l'avant l'arrière et la rotation du robot, ce qui simplifiera le fonctionnement avec le capteur.

IV. Réseau

I - Explication de l'architecture du réseau

Pour pouvoir envoyer les données récupérées par l'ESP nous avons mit 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 :

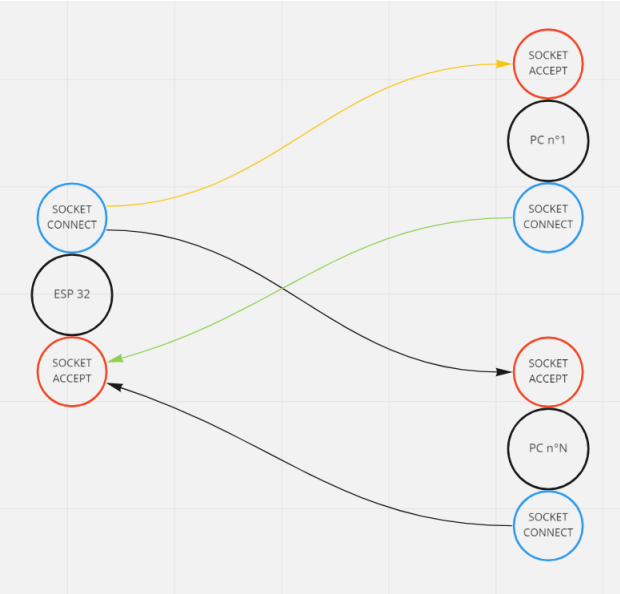

Figure 3: Architecture du réseau

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 respective, 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).

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.

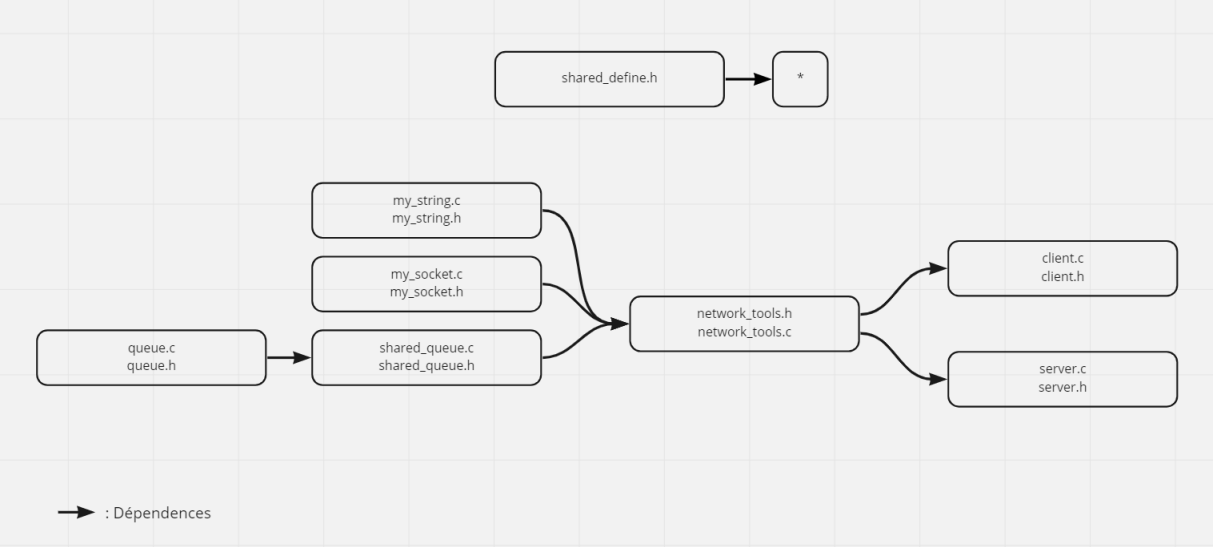

Figure 4: Fichiers utilisés par le reseau

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 strings facilement modifiable. Nous aurions pu utiliser GLib, mais du 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 :

```c
typedef struct _my_socket
{
    int mode;
    char *port;

    // SOCK_ACCEPT
    int sfd_accept_mode;
    pthread_t accept_loop;
    void* (*fn)(void*);
    void **fn_args;

    // SOCK_CONNECT
    int sfd_connect_mode;
    char *host;
} 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 créer par nos sockets par leur mode respectif.

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'argument à donner à cette fonction si besoin est.

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 :

```c
typedef struct _connection_info
{
    int connection_type;
    int sfd_receive;
```

```
    int sfd_send;

    shared_queue *pending_response;

    pthread_mutex_t mutex;
    int is_sending;

    char id;
    my_socket *sock;

    pthread_t loop_thr;
} 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 associeée 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 à 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 correspondront aux 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 à peu près équivalent, en effet, ils nous permettront d'initialiser et de lancer toutes les choses nécessaires au bon fonctionnement du client(PC)/serveur(ESP). En effet, ils initialiseront les commandes nécessaires au 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 :

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

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

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

```
ARG1 ARG2 ARG3 ... ARGn\n
ARG1 ARG2 ARG3 ... ARGn\n\n\n
```

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

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

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, d'afficher cette modélisation qui est réalisée grâce à OpenGL et d'enregistrer cette modélisation.

Cette interface graphique sera réalisée grâce à GTK et à Glade. Glade nous permettra notamment de concevoir l'aspect visuel de l'interface. Nous allons faire dans un premier temps une interface assez simple pour assurer un résultat fonctionnel, et si nous avons le temps nous tenterons de l'améliorer pour la rendre plus agréable d'utilisation.

Nous avons commencé à réaliser l'aspect visuel de l'interface graphique. Nous étions confrontés à un problème avec GTK et OpenGL car ils ne semblaient pas être compatibles entre eux. Pour afficher du OpenGL, nous avions envisagé d'utiliser un "GtkGLArea widget" mais ne semblaient pas être compatible avec GTK. Par conséquent, nous avons réfléchi à diverses solutions et celles qui semblaient être la plus propices pour l'utilisateur est de conserver cette interface réalisée grâce à GTK et de créer un bouton qui lancerai directement le rendu openGL (peut être à l'aide de la fonction execvp de la librairie "unistd.h" qui permettra d'ouvrir directement le rendu sur OpenGL au lieu de l'afficher sur l'interface graphique). Nous souhaitions que l'utilisateur ait tout de même une solution efficace afin de pouvoir l'utiliser convenablement.

II - OpenGL

Parlons maintenant de la partie modélisation de notre projet. Nous avons commencé à travailler sur OpenGL et ses différents aspects. La plupart du projet en terme de déplacement de caméra est terminée. OpenGL est une API assez complexe à comprendre mais se révèle très utile sur de nombreux aspects. Nous utilisons la version 3.3 de l'API. Pour utiliser OpenGL, nous utilisons plusieurs librairies. La première est GLFW, qui nous donne les nécessaires pour créer un rendu graphique. Sur Linux, il est plus simple de compiler cette librairie nous même. Ceci est fait avec les options -lm -lglfw -lGL -lX11 -lXi -ldl -lXrandr -lpthread de GCC. Nous utilisons aussi la librairie cglm (C OpenGL Math) qui nous permet de réaliser un bon nombre d'opérations sur des matrices, vecteurs et autre, très utile pour translations et rotations d'objets et de caméra. Le lien de la librairie est ici. La documentation nécessaire se situe dans le repo github. Il faut bien sûr l'ajouter dans le dossier include. Enfin, Nous avons besoin d'installer les drivers spécifiques pour OpenGL. Cela ce fait via la librairie GLAD, qui dépend de la version d'OpenGL nous utilisons (pour rappel 3.3). Pour l'installer, il faut aller sur ce site, sélectionner l'API gl 3.3 en langage C/C++ et choisir le profil "Core" avec l'option "Generate a loader" cochée. Le fichier src/glad.c est déjà inséré dans le projet. Mais les dossiers KHR et glad doivent être rajoutés dans le dossier include.

En terme de visuel actuel sur notre projet, il n'y a rien de plus que le tutoriel de qui nous a permis de bien comprendre le fonctionnement de l'API et ses différentes notions. Ce tutoriel comprend des mouvements de caméra en 3D, placer des points et créer des surfaces ainsi que

des arcs entre différents points. Nous n'allons pas rentrer dans les détails du fonctionnement d'OpenGL car c'est assez lourd en terme de fonctions, mot clés et mathématiques. Il y a pour le moment un fichier shader.h ne contenant que des définitions de fonctions mais qui sera sans doutes utile pour la suite du projet. Nous avons préféré ne pas l'implémenter pour le moment, au cas où il ne servirai pas. En revanche, l'ouverture de la fenêtre est déjà liée à l'interface graphique.

VI. Stockage de nuage de points

Pour stocker notre nuage de point envoyé par l'ESP, nous avons changé la structure utilisée lors de la première soutenance. Nous nous sommes dirigés vers une structure connue du nom d'octree. Un arbre dont chaque noeud comprend 8 fils. Toutes les informations concernant cette structure ce situent ici. Pour résumer, l'espace est divisé en 8 cubes de tailles égales. Chaque nœud contient au maximum 8 "Objects" (points) et est divisé en 8 sous noeuds si il doit en contenir un de plus.

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 avons 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 };
```

Si vous vous demandez où sont les 8 fils, ils sont alloués tous en même temps, ce qui nous permet d'avoir une représentation mémoire des plus modestes. Mais elle nous oblige à avoir 8 fils, et à les libérer en même temps. Ce qui ne pose pas de problème pour ce que nous voulons réaliser. Et ceci pour l'arbre en lui même :

```
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 maximum de l'arbre
10     int                 depth;
11 };
```

Pour référence, voici les deux autres structures implémentées pour les octree's :

```
struct Vector3
{
    double x, y, z;
};

struct Object
{
    // L'objet suivant dans le noeuds
    struct Object       *next;
    // Les coordonn es de l'objet
    struct Vector3      *coords;
    // Liste doublement chainee pour representer les arcs
    struct Vector3      **edges;
    // nombre pour representer l'objet dans la liste des faces de la sauvegarde
    double              index;
};
```

VII. Algorithmes de modélisation

Nous n'avons encore rien implémenté, mais nous avons 3 algorithmes dans notre champ de vision:

- Alpha Shapes : Edelsbrunner - 1983
- Ball pivoting : Bernardini - 1999
- Poisson Surface Reconstruction : Kazhdan - 2006

VIII. Recherche d'un plus court chemin

I - Création de la structure de tas binaire

Pour la recherche du plus court chemin, nous avons implémenté les bases de la structure de Graph lors de la précédente soutenance, et désormais nous avons implémenté la structure de tas binaire qui est nécessaire pour la bonne application de l'algorithme que nous utiliserons et que nous détaillerons dans la prochaine partie.

Voici les différentes structures mises en place concernant le tas binaire :

- 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.

II - Recherche du plus court chemin

Nous n'avons pas encore pu commencer le développement de cette partie. Après réflexion nous nous orientons sur l'algorithme d'Astar (A*) qui semble être la solution la plus adaptée et la plus optimisée pour notre projet.

IX. Site Web

Ce site sera le support principal pour suivre notre avancée. Il permettra à des personnes extérieures de consulter nos différents rapports (Cahier des charges, rapports de soutenance ..) ainsi que de télécharger le projet.

Nous avons continué à élaborer les pages ci dessous pour rendre le site encore plus agréable d'utilisation, et surtout de le compléter pour y mettre le plus d'information possible concernant ce projet.

- une page d'accueil
- une page pour l'historique du projet
- une page contenant un devblog
- une présentation des membres du groupe
- une page de chronologie de réalisation
- une page de ressources (liens vers les sites et outils qui nous ont été utiles pour la conception du projet)
- une page pour télécharger les différents rapports de soutenance ainsi que le cahier des charges
- une page pour télécharger le projet avec ses différents manuels (avec une version lite sans les documents)

Le site est accessible en cliquant ici

IV. Coûts

En ce qui concerne les coûts, nous avons du acheter du matériel et notamment :

- 1 ESP 32 : 9€
- 1 TOF 4M : 5,65€
- 4 TOF 2M : 13,92€

V. Conclusion

Pour conclure, le sommeil vient à manquer mais le projet prend forme petit à petit. Nous commençons à connecter nos différentes parties et nous sommes confiants pour la fin de notre projet. Nous commençons à voir le résultat de plus en plus proche et nous en sommes content. A bientôt pour la fin de notre aventure !