



ENSEIRB-MATMECA

PR 214

PROJET THÉMATIQUE

Twin Peaks

Auteurs :

Samir Mammeri
Maxime Descos

Table des matières

1	Introduction	2
2	Structure de la tourelle	3
3	Contrôle de la tourelle avec un Arduino	5
3.1	Montage électronique	5
3.2	Programmation de l'Arduino	6
4	Récupération des données sur l'ordinateur en Lua	7
5	Traitement graphique des données	9
5.1	OpenGL	10
5.2	GLUT ToolKit	13
5.3	Algorithme de l'implémentation	13
5.3.1	Fonction InitGL	14
5.3.2	Fonction Reshape	15
5.3.3	Fonction Draw	16
5.3.4	Évènements du clavier et de la souris	18
6	Conclusion	19
7	Bibliographie	20

1 Introduction

La construction du programme de la filière Électronique nous permet de choisir une UE optionnelle lors du semestre 8 pour nous permettre d'approfondir nos connaissances dans un domaine particulier et voir si on veut continuer dans cette voie pour le semestre 9. Nous avons choisi la spécialité numérique car l'électronique numérique est désormais partout et sera encore plus présente à l'avenir.

Dans ces UE optionnelles, il nous est demandé de réaliser un projet dans la thématique de l'UE. Avec M. Crenne, nous avons choisi de réaliser un scanner 3D grâce au capteur LIDAR (Light Detection And Ranging). Ce capteur utilise les ondes infrarouges et par écho, détermine la distance entre lui-même et l'objet touché par les ondes. C'est le principe classique du sonar ou du radar, mis à part que les ondes infrarouges permettent la détection de très petits objets.

Aujourd'hui, on peut trouver ce type de capteur pour une centaine d'euros en vente libre. Les projets l'utilisant ont donc fleuri. De plus, ce projet permet de toucher à plusieurs domaines autour du numérique et de la création de produit. En effet, aujourd'hui un ingénieur ne travaille pas forcément dans un grand groupe avec tout le matériel que cela induit. Il peut se lancer dans une start-up, il créera alors un produit avec les moyens qu'il a à sa disposition. Pour cela, le projet nous montre cette méthode de travail.

Passons au projet en lui-même, nous l'avons défini de la manière suivante :

Le scanner 3D sera composé d'une tourelle pivotant sur 2 axes, ce qui permet au LIDAR de capturer un ensemble de données représentant une scène 3D. La tourelle et le LIDAR seront pilotés via un Arduino qui lui-même sera relié à un ordinateur recevant les données. On affichera alors ces données de manière à reproduire une scène 3D. On pourra se déplacer dans la scène affichée pour l'observer de différents points de vues.

Bien que n'ayant pas travaillé dans cette ordre, nous allons expliquer le projet de façon linéaire, c'est-à-dire, partir du matériel pour aller jusqu'à la visualisation des données sur l'ordinateur. On va donc, dans un premier temps, voir comment la tourelle accueillant le LIDAR a été designé et fabriqué et comment elle permet au LIDAR de pivoter pour accueillir les données de différents points. Nous passerons ensuite au codage de l'Arduino qui pilote la tourelle et fait le lien entre le LIDAR et l'ordinateur. De plus, nous aborderons le programme Lua qui récupère les données venant de l'Arduino et les stocke dans un fichier texte. Et enfin, nous verrons comment afficher les données pour reproduire la 3D grâce à la librairie OpenGL.

2 Structure de la tourelle

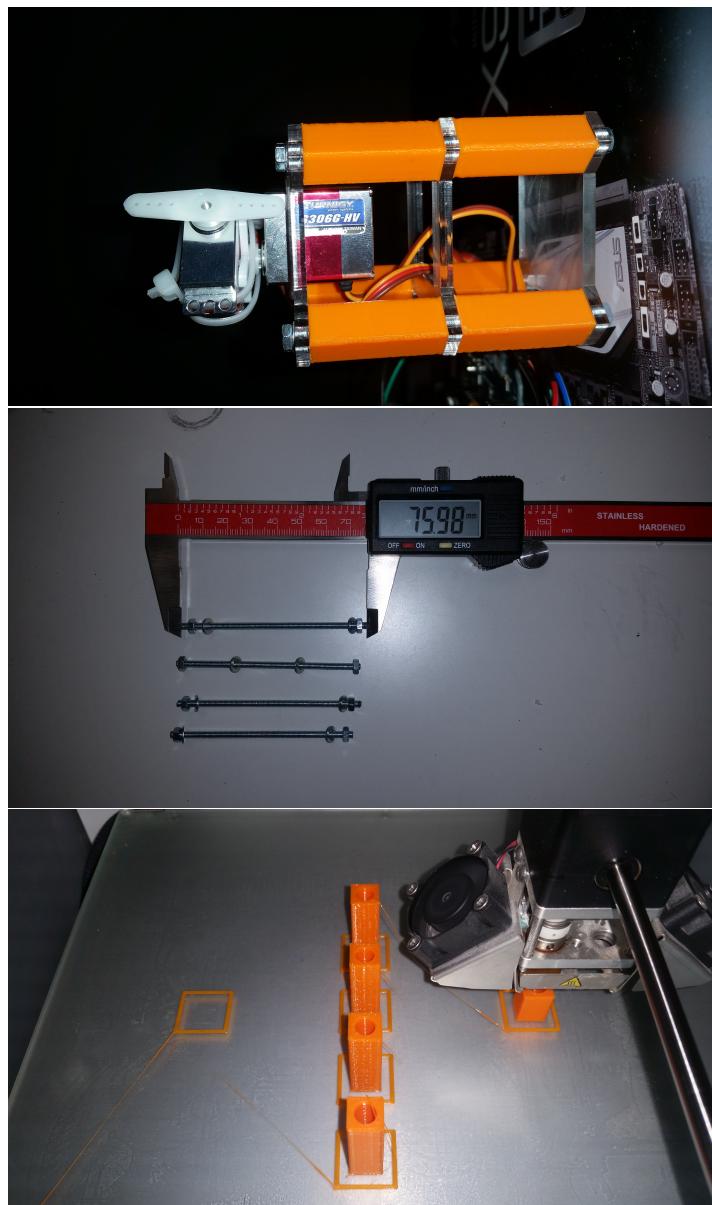


FIGURE 1 – Image du haut : la tourelle montée avec les servomoteurs, Image du milieu : Visse métalliques faites sur mesure, Image du bas : Impression 3D des entretoises

Twin Peaks

La dernière étape du projet a été de concevoir une tourelle pour venir soutenir les deux servomoteurs ainsi que le Lidar. Nous avons pris pour exemple un des nombreux Pan&Tilt Kit présent sur internet.

Nous avons donc commencé par réaliser les trois étages de la tourelle. Les schémas ont été réalisé sur InkScape, un logiciel de dessin, pas industriel, permettant de générer des fichiers en .svg. Les pièces ont été réalisée au *FABLAB* de l'*ENSEIRB MATMECA*, avec la participation de M. Julien Allali. Le matériau choisi pour les trois niveaux de la structure fut du plexiglas La base de la structure n'a pas de trou en son centre contrairement aux deux autres niveaux. En effet, l'étage critique est celui du haut, puisqu'il a fallu prendre soin de dimensionner la fente, au millimètre près, en fonction des dimensions du servomoteur censé se loger au centre du niveau supérieur.

Enfin, toujours avec la collaboration de M. Allali, nous avons conçu les entretoises, des parallélépipèdes avec en leurs centre un cylindre vide de rayon 3mm, pour permettre d'y rentrer les visse métalliques voir l'image du milieu ci dessus. puis nous les avons imprimé avec l'une des nombreuses imprimantes 3D que propose le *FABLAB*.

3 Contrôle de la tourelle avec un Arduino

3.1 Montage électronique

Passons maintenant au contrôle des servomoteurs et du LIDAR. Comme nous l'avons vu précédemment, la tourelle est mobile sur 2 axes, chacun dépendant d'un servomoteur. Ceci permet de faire tourner le LIDAR sur une grande plage de l'espace. Cependant, nous avons besoin de contrôler ses servomoteurs et de savoir où ils sont pour, en associant ces données avec la distance de l'objet donnée par le LIDAR, reproduire la scène.

Pour cela, nous allons utiliser un Arduino. Ce genre de microcontrôleur permet un prototypage très rapide et simple car son programme est codé en C. On va, dans la suite de cette partie, analyser le code arduino.ino, mais tout d'abord, voyons le schéma électrique :

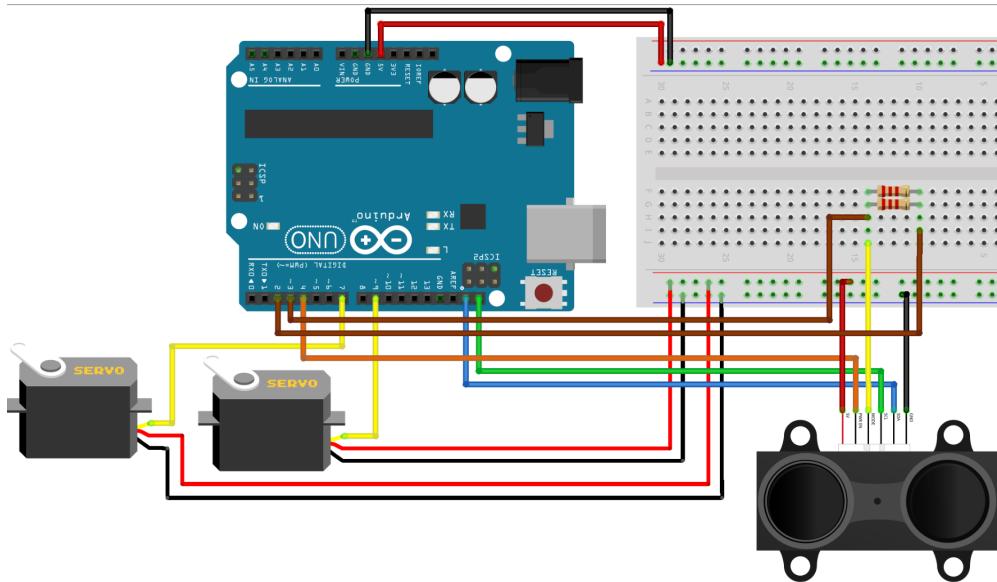


FIGURE 2 – Schéma électrique du montage

On voit que les deux servomoteurs sont reliés sur les pins 7 et 9 de l'Arduino et le LIDAR est relié sur les pins SDA (Serial Data Line) et SCL (Serial Clock Line) qui sont les deux lignes de communication du protocole I^2C (on ne prend pas en compte ici les lignes d'alimentation et de masse). Le protocole I^2C est un protocole classique de communication pour échanger des données avec un microcontrôleur, il est donc utilisé pour le LIDAR.

3.2 Programmation de l'Arduino

Passons au programme (`arduino.ino`) :

Dans un premier temps, on a la déclaration des bibliothèques et les variables globales. Dans cette partie, nous définissons les adresses des registres du LIDAR dans lesquelles nous allons lire les données. Nous déclarons aussi deux servomoteurs : un pour la rotation autour de l'axe z (le pan : `servoRotate`) et un autour de l'axe y (le tilt : `sevoScan`) et on inclue les bibliothèques nous permettant les contrôles sur les servomoteurs et sur le périphérique I^2C , i.e. le LIDAR.

Voyons ensuite l'initialisation (fonction `setup`) :

Classiquement, nous rattachons les servomoteurs aux pins définis plus haut. Nous engageons la communication série et I^2C . Ensuite, on active le capteur et on le positionne sur le mode qui nous intéresse, i.e. continuous read (mesure en continue). De plus, on fait effectuer un mouvement aux servomoteurs pour vérifier qu'ils répondent bien. Enfin, on attend que la communication série (avec l'ordinateur) soit bien établie et on positionne les servomoteurs dans leurs positions initiales.

On a ensuite la fonction qui va s'exécuter en boucle (fonction `loop`) :

L'algorithme est une boucle sur les servomoteurs : on incrémente `servoRotate` de un degrés à chaque fois qu'une distance a été récupéré. Lorsqu'on arrive à 180 degrés, on incrémente `servoScan` de 1 et on repart dans l'autre sens. La distance récupérée est sur 2 octets. On stocke alors tout ça dans l'entier `distance` créé au début en tant que variable globale.

Voici ci-dessous l'algorithme du programme :

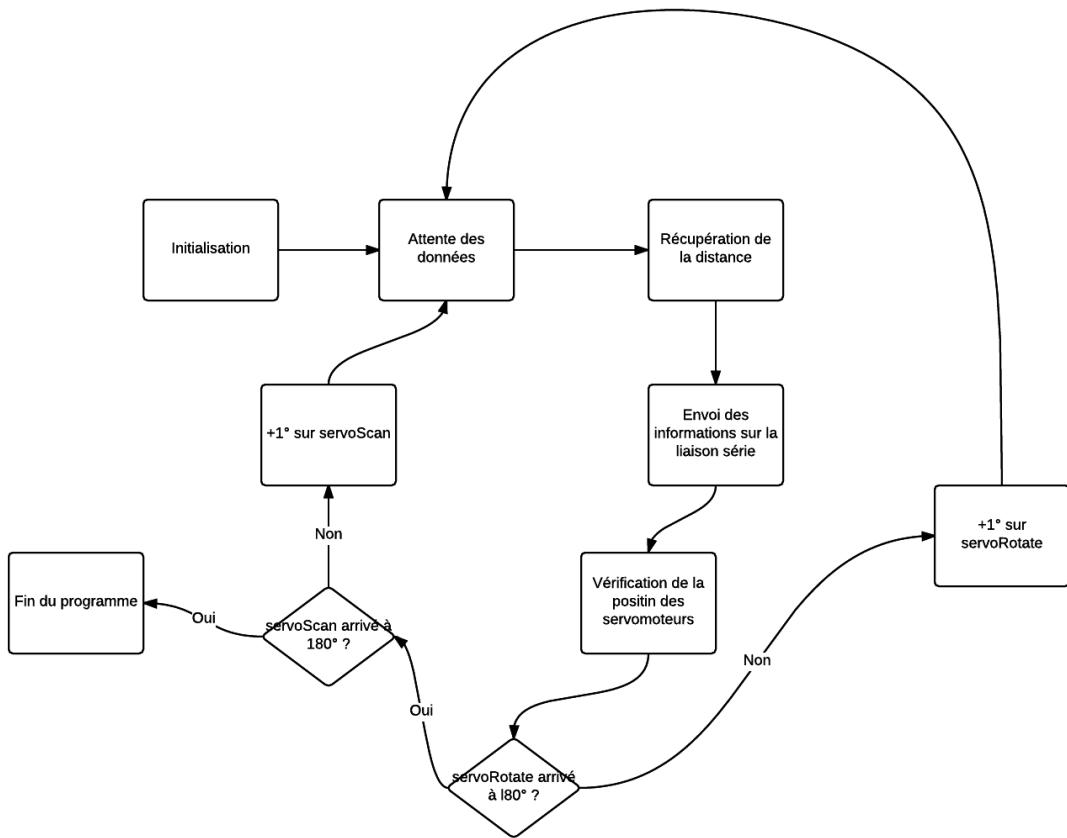


FIGURE 3 – Algorithme de arduino.ino

Enfin, on envoie les données : position des deux servomoteurs et la distance de l'objet avec la fonction `printOutput`. Cette fonction est classique et envoie les données dans un certain ordre. On va passer ensuite à la récupération des données du côté de l'ordinateur.

4 Récupération des données sur l'ordinateur en Lua

Ayant découvert le langage Lua dans le cours MI201 Microinformatique de M. Crenne, nous avons voulu apprendre à l'utiliser un peu mieux à travers le programme de récupération et de stockage des données. Nous allons alors expliquer le programme utilisé (`main.lua`) :

Tout d'abord, on inclut la bibliothèque permettant la communication série via le programme Lua, c'est la bibliothèque "luars232". Ensuite, on choisit le port sur lequel lire les données suivant le systèmes d'exploitation de la machine. On ouvre ce port, on fixe les paramètres de la communication, tels que le débit de baud ou la parité. De plus, on ouvre un fichier texte dans lequel on va stocker les données reçues. Ici, c'est le fichier "test.txt" qui les recevra. Enfin, on effectue une boucle qui reçoit les données, jette les données non pertinentes (utilisées pour le débogage) et écrit les bonnes données dans le fichier texte suivant le format :

- position de servoRotate
- position de servoScan
- distance de l'objet

On ferme ensuite le fichier. C'est après que le code C, avec la bibliothèque OpenGL, traite les données comme nous allons le voir dans la partie suivante.

Voici ci-dessous l'algorithme du programme main.lua :

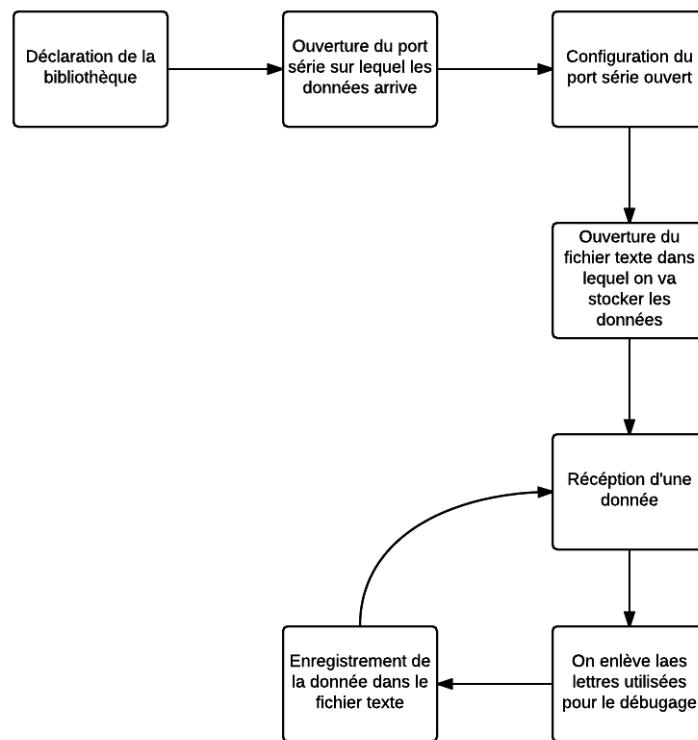


FIGURE 4 – Algorithme de main.lua

5 Traitement graphique des données

La seconde partie de ce projet a été le prototypage d'un environnement 3D assurant l'affichage des données recueillies depuis un LIDAR par l'intermédiaire d'un fichier .txt servant d'interface entre les parties Recueil et Traitement de l'information. Le schéma ci dessous nous montre comment nous avons choisi d'articuler notre projet.

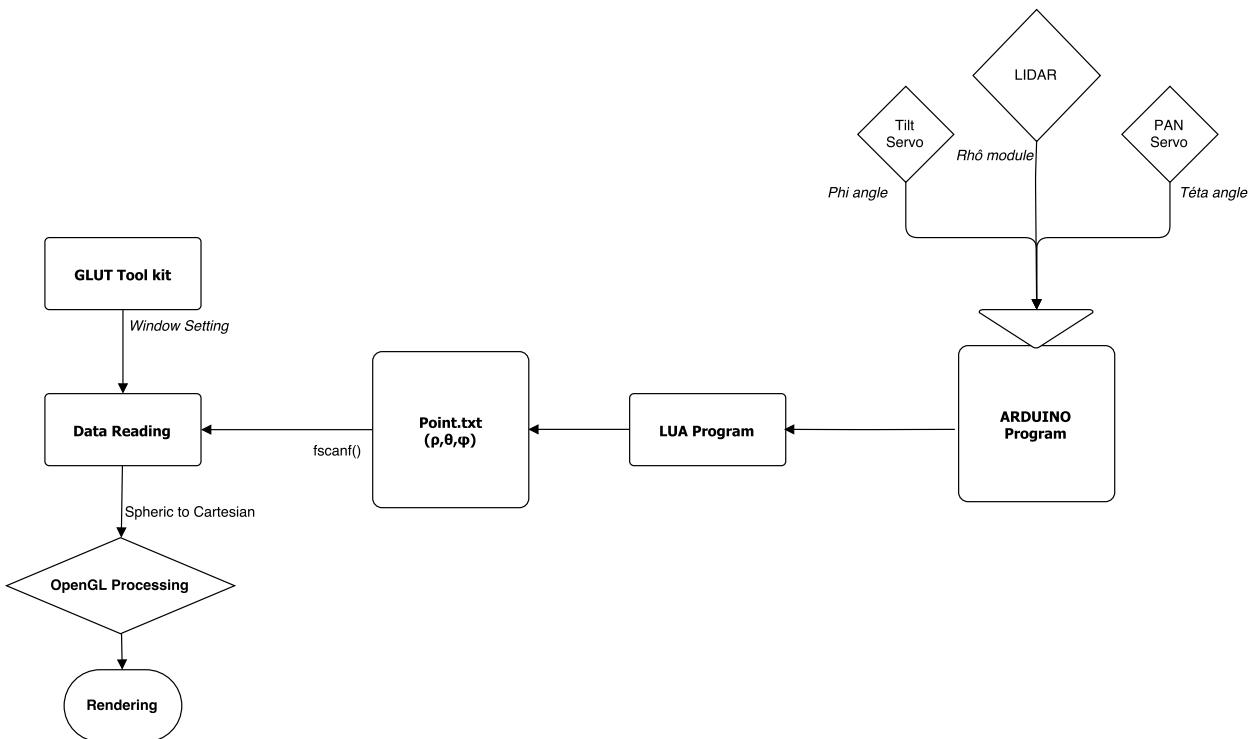


FIGURE 5 – Schéma synoptique du projet

Avant d'entrer dans les détails de l'implémentation choisie, nous allons présenter succinctement ce qu'est OpenGL et le ToolKitGlut. Premièrement OpenGL est une librairie graphique 3D. Le principe de fonctionnement est le suivant : On lui donne des primitives 3D (Carrés, Cubes, Triangles, Textures, Position de la source lumineuse, Position de la caméra...), ces dernières sont interprétées et traitées. En effet, OpenGL se charge de faire les projections en perspective vis à vis de l'écran, les changements de repère, ect. Le schéma ci dessous reprend les différentes étapes suivies par la machine OpenGL.

5.1 OpenGL

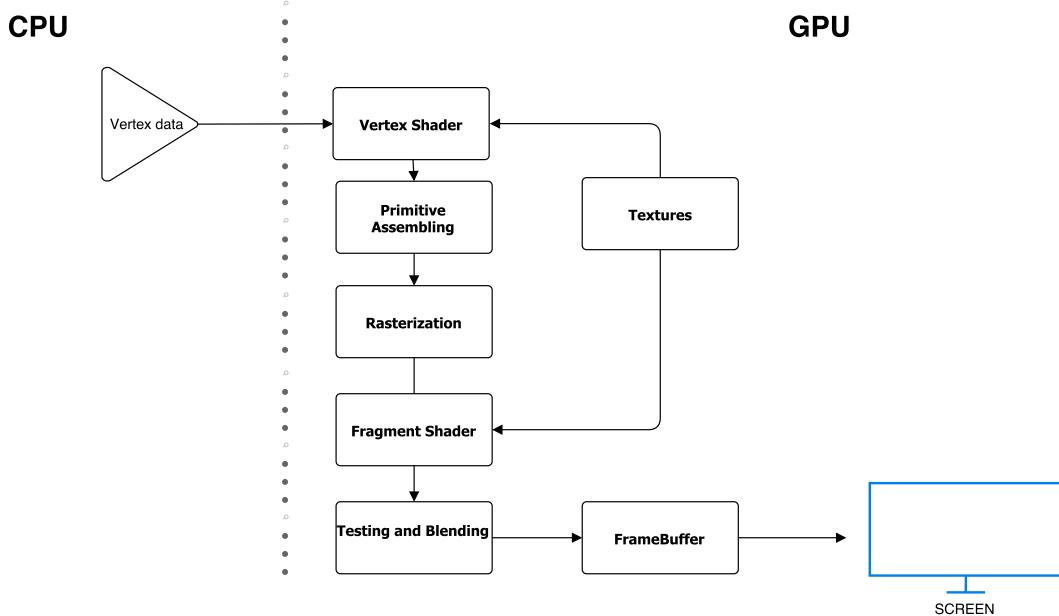


FIGURE 6 – Pipeline de la machine OpenGL

OpenGL va communiquer directement avec la carte graphique pour gérer les différentes étapes d'affichage, ci dessous une brève définition des phases suivies (chacune de ces étapes mériterait des pages d'explications et de précisions) :

- **Vertex Shader** : Programme qui prend un ensemble de sommet comme entrée et qui renvoie en sortie un ensemble d'attributs qui seront envoyés vers l'étape de la rasterization. Le GPU va lire chaque sommet présent dans la matrice de sommet et lui appliquer le programme Vertex Shader qui calculera la position projetée du sommet dans l'espace de l'écran.

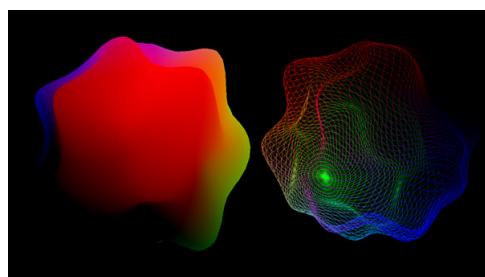


FIGURE 7 – Vertex shader [1]

- **Primitive Assembly** : Puis, le GPU connecte les sommets projetés par deux, trois, ... en fonction de la primitive spécifiée. Voici quelques exemples de primitive sur OpenGL : POINTS, LINES, QUADS, POLYGON, ... :

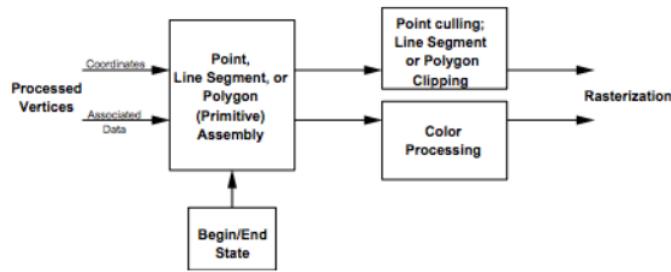


FIGURE 8 – Primitive assembly processing [2]

- **Rasterization** : Lors de cette étape, toutes les formes précédemment assemblées vont être traduites en lignes de fragment (un fragment est l'état intermédiaire du pixel de sortie final). C'est lors de cette étape où les parties des formes qui dépassent le cadre d'affichage vont être "tronquées". Et les parties restantes seront subdivisées en plusieurs fragments (pixels) et placées selon un gradient de couleurs.

Ci dessous, un exemple de rasterization d'un triangle.

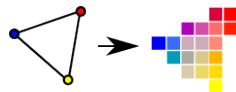


FIGURE 9 – Principe simplifié d'une rasterization

- **Fragment Shader** : Les fragments générés lors de l'étape de rasterization vont être traduit en une profondeur et une couleur qui seront dessinées dans le FrameBuffer. C'est lors de cette étape que l'on prend en considération l'éclairage et le mapping des textures. Le fragment shader d'OpenGL permet de traiter chaque pixel indépendamment laissant lieu à d'éventuel accélération logicielle et matérielle.
- **Testing and Blending** : Le résultat de cette pipeline est le Framebuffeur ; qui peut être un simple buffer (Par défaut) ou un double pour prendre en considération la profondeur des objets d'une scène 3D. Ainsi, le FrameBuffeur est la sortie de cette dernière étapes, durant laquelle il se verra remplir des informations relatives aux profondeurs et aux couleurs des différents objets de la scène. Dans le projet Twin Peaks la framebuffer sera un double

buffer contenant toutes les informations de scene 3D : COLORBUFFER et DEPTHBUFFER.

OpenGL utilise un système de matrice pour gérer les changements de repère et les transformations géométriques, comme le montre le schéma ci dessous :

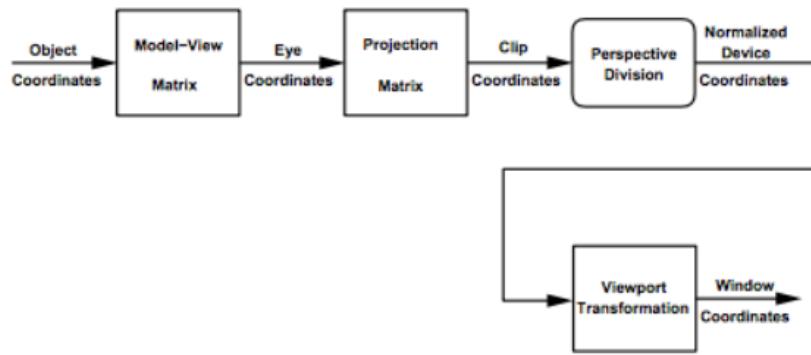


FIGURE 10 – Vertex transformation sequence [3]

De gauche à droite, voici les différentes coordonnées en fonction des différentes transformations :

Les coordonnées de l'objet sont les coordonnées initiales et l'orientation initiale d'un objet qui va subir une ou des transformations (*glRotatef()*, *gltranslatef()*, ...)

Les coordonnées de l'oeil, sont obtenues en multipliant les coordonnées initiales par la matrice *MODELVIEW*. Cette dernière est elle-même une combinaison de deux matrice *MODEL* et *VIEW*, la première transforme les coordonnées de l'espace de l'objet vers l'espace principale puis la deuxième les transforme vers l'espace de l'oeil. Puis les coordonnées sont multipliées par la matrice *PROJECTION* et deviennent les coordonnées "coupées", Clipped coordinates en anglais car les coordonnées (x,y,z) dépassant la borne fixée par la largeur de l'écran (+ ou - *WIDTH*) seront tronquées. C'est la matrice *PROJECTION* qui permet de prendre en compte la perspective, que l'on explicitera lors du détail de la fonction Reshape du projet. Les coordonnées coupées sont ensuite normalisées par la largeur de l'écran *WIDHT*, cette étape s'appelle la division de perspective.

Enfin les coordonnées normalisées sont transformées par viewport, qui permet de les afficher dans l'écran de rendu. Les coordonnées de la fenêtre sont ensuite envoyées vers l'étape de rasterization du pipeline d'OpenGL pour devenir des fragments.

5.2 GLUT ToolKit

OpenGL, à lui seul ne permet pas l'affichage d'une fenêtre, c'est à cette fin que nous avons choisi d'utiliser GLUT ToolKit. Bien que la version utilisée lors de notre projet est dépréciée et n'est plus maintenue, mais elle reste tout de même effective et simple d'utilisation.

5.3 Algorithme de l'implémentation

L'ensemble des données lues sur le fichier .txt doivent être affichées dans un espace tridimensionnel. La structure de l'algorithme retenue pour réaliser ce besoin est la suivante :

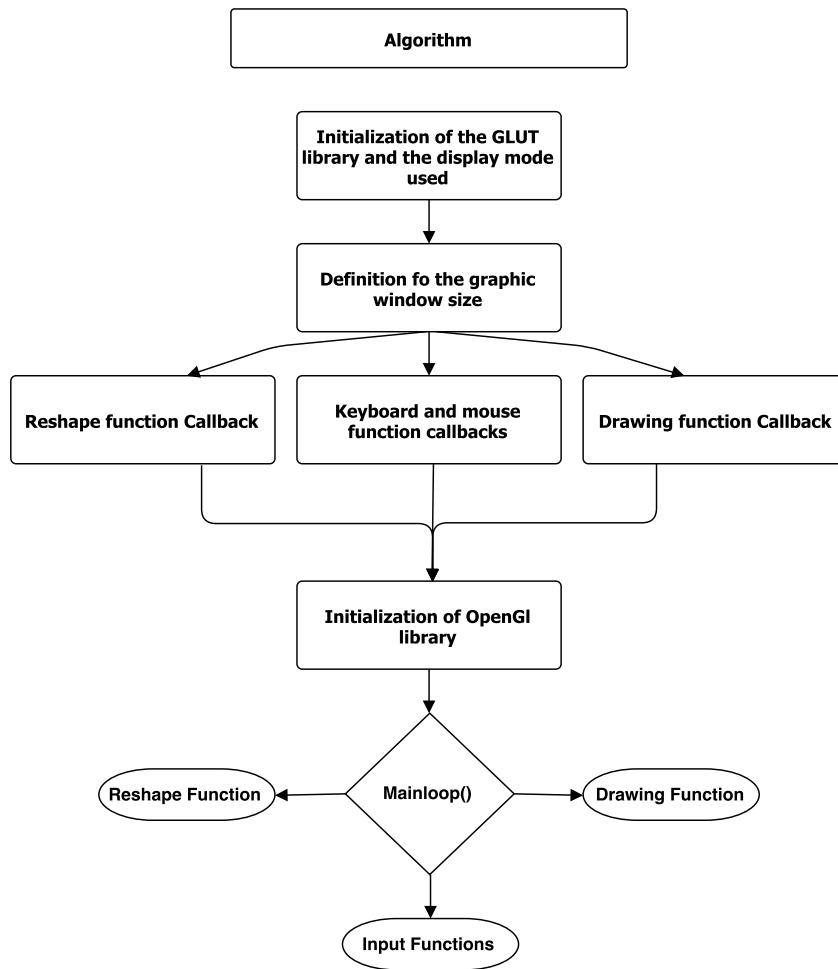


FIGURE 11 – Structure de l'algorithme du projet

Le schéma ci dessus nous montre le fil d'appel et d'exécution du programme principale. Après avoir initialisé la librairie ainsi que le mode d'affichage un premier appel est effectué, visant à créer les liens vers les fonctions utilisées et appelées dans la boucle principale (après avoir initialisé la librairie OpenGL).

Nous allons voir dans la suite de ce rapport une explication des fonctions principales de ce projet : *InitGl()*, *Reshape()*, *Draw()*, *InputFunctions()*.

5.3.1 Fonction InitGL



FIGURE 12 – Étapes de la fonction InitGL

La fonction *InitGL* sert à créer un contexte OpenGL et à spécifier des paramètres d'initialisation :

- **Background Color** : En effet, la première étape de la fonction est de définir la couleur de l'arrière plan avec la fonction *glClearColor()*, dans ce projet nous avons choisi la couleur grise.
- **Depth Test** : Puis lors de la seconde étape, le test sur la profondeur doit être activé avec la fonction *glEnable()* pour s'assurer que les faces cachées des objets de la scène ne vont pas être affichées.
- **Viewport** : La fonction *glViewport()* définit la transformation affine de x et de y, en coordonnées normalisées, vers les coordonnées de la fenêtre. Lors de cette appel, nous lui spécifions le couple de coordonnée (0,0) et les dimensions de notre fenêtre WIDTH et HEIGHT. Les formules [4] ci dessous nous montrent comment est appliquée cette transformation.

Soit x_{nd} et y_{nd} les coordonnées normalisés, et x_w et y_w les coordonnées relatives à la fenêtre :

$$x_w = (x_{nd} + 1) \left[\frac{\text{width}}{2} \right] + x$$

$$y_w = (y_{nd} + 1) \left[\frac{\text{height}}{2} \right] + y$$

FIGURE 13 – Détermination des coordonnées de la fenêtre

- **Color and Texture** : Dans cette étape, la couleur et les textures sont activées la fonction *glEnable()*.

- **Lightning and Light** : La fonction *InitGl* se termine par l'activation de l'éclairage et le choix de la lumière *GL_LIGHT1*, toujours avec la fonction *glEnable()*.

5.3.2 Fonction Reshape

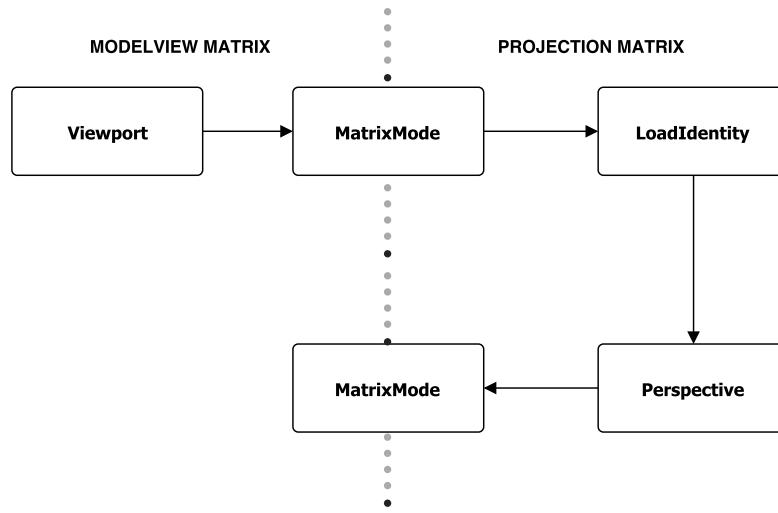


FIGURE 14 – Étapes de la fonction Reshape

La fonction *Reshape* sert à mettre en place la perspective en définissant le champ de profondeur lointain et le champ de profondeur proche, mais il faut dans un premier temps prendre le contrôle de la matrice de projection. Cette fonction est appelée en premier dans la séquence d'appel de la Mainloop. Voici donc le squelette de la fonction *Reshape* :

- **Viewport** : La fonction *glViewport()* doit être appelée en lui spécifiant comme arguments le couple de coordonnées relatif au point le plus bas à gauche , $(0,0)$ par défaut, ainsi que les dimensions de la fenêtre *WIDTH* et *HEIGHT*.
- **MatrixMode** : Puis la fonction *glMatrixMode()* doit être appelée pour prendre le contrôle de la matrice de projection, qui est initialisé à la matrice identité avec la fonction *glLoadIdentity()*.
- **Perspective** : Les deux étapes préliminaires étant accomplies, l'appel à la fonction *gluPerspective()* peut être effectué. Comme précisé précédemment, nous allons définir les champs de profondeur proche et lointain. La fonction prend aussi en paramètres le ratio $\frac{WIDTH}{HEIGHT}$ ainsi le champ de vue (*FOV*) exprimé en degré.

Ci dessous un schéma représentatif des notions de champs de profondeur :

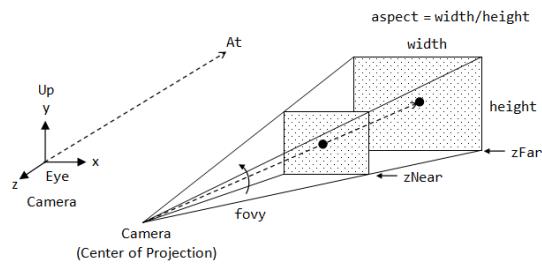


FIGURE 15 – Perspective de profondeur

- **MatrixMode** : Enfin la fonction *glMatrixMode()* est rappelée pour reprendre le contrôle de la matrice *ModelView*.

5.3.3 Fonction Draw

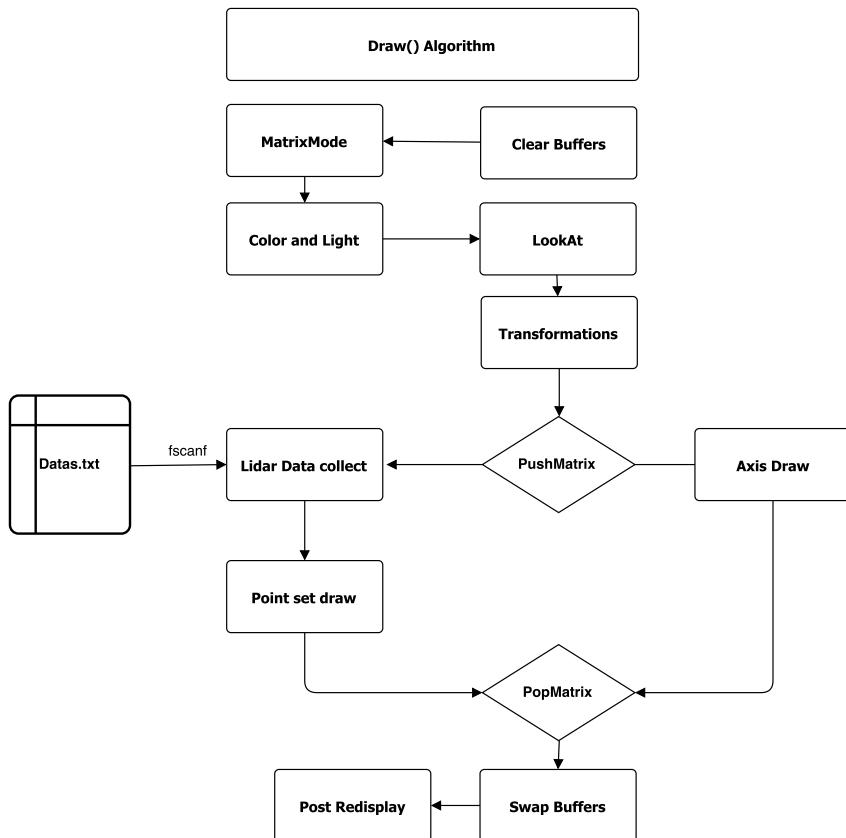


FIGURE 16 – Algorithme de la fonction *draw()*

Draw() est articulée en plusieurs points comme le montre le schéma au dessus. Ceux ci sont nécessaires pour pouvoir afficher les données du LIDAR. Voici quelques détails sur ces étapes :

- **Clear Buffers** : La fonction *glClear()* doit être appelée en lui spécifiant comme arguments les buffeurs à vider. Dans ce projet les buffeurs sont : *GL_COLOR_BUFFER_BIT* et *GL_DEPTH_BUFFER_BIT*
- **MatrixMode** : Puis la fonction *glMatrixMode()* doit être appelée pour prendre le contrôle de la matrice *MODELVIEW*, qui est initialisé à la matrice identité avec la fonction *glLoadIdentity()*.
- **Color and Light** : Puis nous activons les couleurs sur les arrière et avant des objets avec *glColorMaterial()* et nous choisissons le mode *GL_EMISSION*. Puis avec *glLightiv()* nous fixons le nombre de lumière à 1 et nous lui passons en arguments les coordonnées de la source lumineuse.
- **LookAt** : Puis nous paramétrons les informations relatives à la caméra avec *gluLookAt()*. Le premier argument de la fonction correspond à les coordonnées relatives (x,y,z) de la caméra, le second les coordonnées (x,y,z) du point de visé de la caméra et le troisième argument correspond à la "normale de visée" (Up vector sur le schéma) de la caméra.

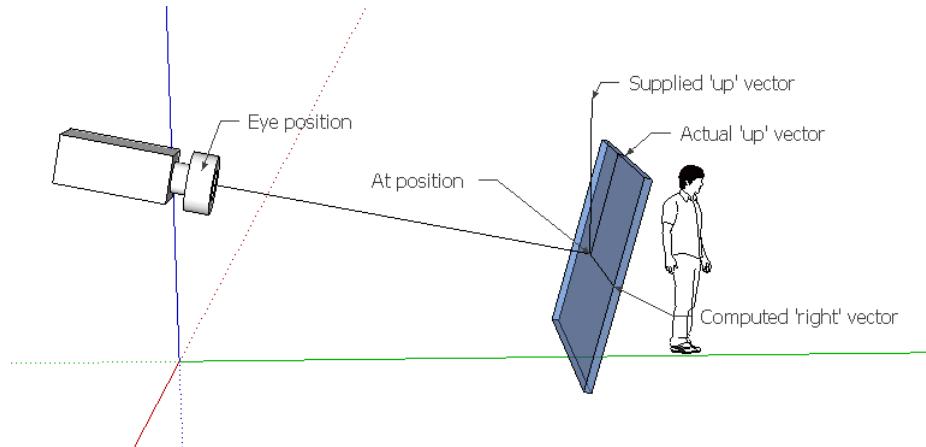


FIGURE 17 – Principe de *gluLookAt()*

- **Transformations** : Avec les fonctions *glTranslated()* et *glRotated()* nous multiplions la matrice courante (*MODELVIEW*) par une matrice de translation ou de rotation. Les arguments de ces fonctions sont des variables globales qui vont évoluer en fonction des évènements du clavier et/ou de la souris.
- **PushMatrix** : Avec la fonction *glPushMatrix()* on sauvegarde la position courante de la matrice *MODELVIEW* pour pouvoir y retourner après toutes les modifications que peux subir la matrice.

- **Axis Draw** : La fonction *AxisDraw()* est appelée pour définir des axes XYZ sur notre repère 3D orthonormé.
- **Lidar Data Collect** : Cette partie est chargée de lire sur le fichier txt, par le biais d'un *fscanf* les données de coordonnées sphériques sont récupérés trois par trois et sont ensuite retranscrites en coordonnées cartésiennes.
- **Point set draw** : Pour chaque trio $(x_{lue}, y_{lue}, z_{lue})$ lu la fonction *Squaredraw()* est appelée. Elle permet de dessiner un cube autour du point de coordonnées $(x_{lue}, y_{lue}, z_{lue})$ de taille générique *SIZE*.
- **PopMatrix** : Avec la fonction *glPopMatrix()* on revient à la position sauvegardée de la matrice *MODELVIEW*.
- **Swap Buffers** : Avec la fonction *glSwapBuffers()* on échange les buffers courants, ceux que l'utilisateur voit, par les nouveaux buffers où se trouvent les cubes précédemment dessinés. C'est ce qui permet d'afficher la nouvelle scène calculée.
- **Post redisplay** : Enfin, avec la fonction *glutPostRedisplay()* on échange l'affichage courant de la fenêtre par celui calculé lors du court de la fonction *draw()*.

5.3.4 Évènements du clavier et de la souris

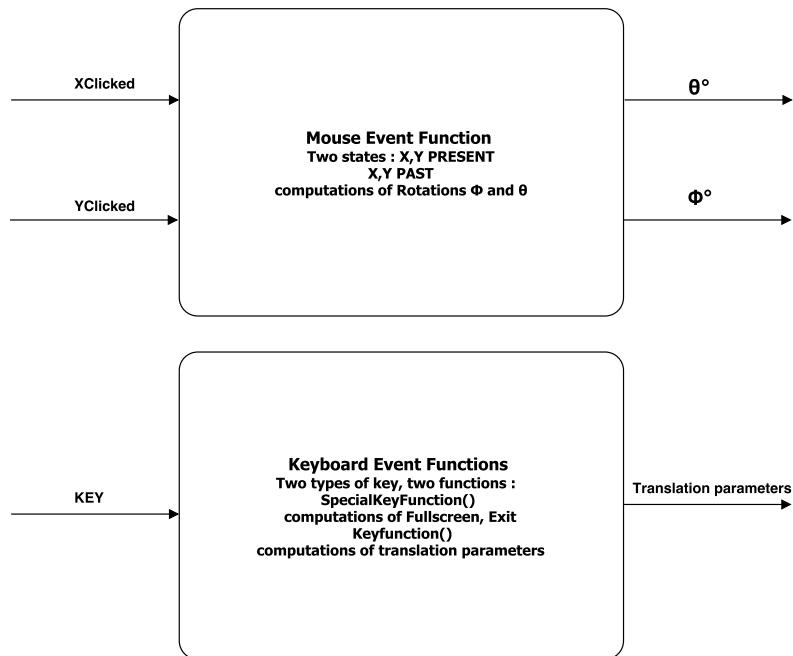


FIGURE 18 – Fonctions gérant clavier et souris

Les fonctions `glutKeyboardFunc()`, `glutSpecialFunc()` et `glutMouseFunc()` sont associées respectivement aux fonctions `KeyboardControl()`, `SpecialKeyboardControl()` et `MouseControl()`.

Une première fonction sert à gérer les touches "normales" du clavier, en voici un descriptif non exhaustif : Sur le clavier en fonction de la touche appuyée, on peut translater la matrice *MODELVIEW* selon les 3 axes X, Y et Z. La touche "F" permet de passer en mode plein écran, la touche "X" permet de quitter le mode plein écran, et la touche "Q" permet de quitter l'application.

Une deuxième fonction sert à gérer les touches "spéciales" du clavier, en voici un descriptif complet : La touche "F1" permet de revenir à la position initiale, les touches "*UP*" et "*DOWN*" permettent de jouer sur la rotation autour de l'axe Z. Les touches "*LEFT*" et "*RIGHT*" agissent sur la rotation autour de l'axe Y. de quitte le mode plein écran, et la touche "Q" permet de quitter

Lorsque l'utilisateur presse la touche gauche de la souris l'état *GL_DOWN* est renvoyé par `glutMouseFunc()` et les coordonnées X,Y de la fenêtre sont sauvegardés dans un structure globale. Lorsque la touche gauche de la souris est relâchée l'état *GL_UP* est renvoyé par la fonction et en fonction de l'écart relatif entre le couple (X_{DOWN}, Y_{DOWN}) et (X_{UP}, Y_{UP}) nous affectons des paramètres de rotation sur la matrice *MODELVIEW*.

6 Conclusion

La tourelle est fonctionnelle, malheureusement nous n'avons pas reçu le capteur LIDAR après plusieurs mois d'attentes et donc nous n'avons pas pu tester concrètement ce projet. Bien que décevant, nous avons quand même pu travailler en moment sur toutes les parties du scanner 3D pour en faire presque un système "plug & play", bien qu'il y ait toujours des choses à modifier après des tests pratiques. On ne peut donc pas dire que notre projet soit totalement fini, mais cela n'a pas été de notre ressort, ni de celui de M. Crenne.

Néanmoins, ce projet nous a appris de nouvelles compétences et a permis d'en consolider d'autres. En effet, nous avons pu faire de l'impression 3D et du découpage laser. Cette partie fut fascinante car elle a permis de voir le résultat de plusieurs mois de travail lorsqu'on a monté les servomoteurs. Bien que connaissant l'Arduino, nous avons pu améliorer nos connaissances, particulièrement dans l'écriture propre de codes. De plus, la mise en pratique dans un projet du langage Lua nous a montré son utilité. Enfin, la découverte de la bibliothèque OpenGL fut passionnante. Nous avons entrevu la puissance de cet outil et ses domaines d'applications, particulièrement pour la 3D. Il n'est pas improbable de dire que nous réutiliserons cette bibliothèque dans l'avenir. Cette introduction ne pourra qu'améliorer notre travail en tant qu'ingénieur

De plus, nous avons aussi vu deux parties importantes du travail d'un ingénieur : la recherche documentaire et la gestion d'un projet. Une grande partie de notre travail a été consacré à la recherche documentaire au travers d'Internet. En effet, un ingénieur ne peut passer son temps à "réinventer la roue", il doit être en permanence en veille technologique et lors d'un projet, il doit se renseigner sur l'existant pour être le plus efficace possible. En ce sens, notre projet s'est rapproché du vrai travail d'ingénieur. De plus, l'ingénieur travaille la plupart du temps en équipe, quelle soit petite ou grande, il doit apprendre à découper correctement un projet pour travailler en parallèle et être le plus efficace possible, surtout à une époque où le "time to market" devient de plus en plus court. Ainsi, nous avons découpé le projet en deux grandes parties : le hardware et le software. L'un a travaillé sur le code C, la bibliothèque OpenGL et l'autre sur l'Arduino et le Lua. Nous nous sommes retrouvés à la fin pour concevoir la tourelle. Cependant, nous communiquions souvent pour savoir l'avancée de l'autre et le travail qui restait à effectuer. Ce projet fut donc une très bonne expérience de travail en équipe.

Malgré la déception de ne pas avoir pu tester notre scanner 3D, ce projet fut enrichissant. Il s'est effectué dans un domaine nous intéressant tous les deux : les objets connectés et l'électronique numérique. Étant donné que nous travaillerons probablement dans ces thématiques plus tard, ce projet fut une parfaite introduction à notre vie professionnelle future.

7 Bibliographie

[1] Github Project, MasDennis, *Tutorial 23 Custom Vertex Shader* 5 Apr 2013 disponible sur : <<https://github.com/Rajawali/Rajawali/wiki/Tutorial-23-Custom-Vertex-Shader>>

[2][3]Mark Segal, Kurt Akeley, *The OpenGL Graphics System : A Specification* (Version 2.0) - October 22, 2004
disponible sur : <<http://isites.harvard.edu/fs/docs/icb.topic942925.files/glspec20.pdf>>

[4]Microsoft, Ressources pour les développeurs *glViewport function*, disponible sur : <[https://msdn.microsoft.com/fr-fr/library/windows/desktop/dd374202\(v=vs.85\).aspx](https://msdn.microsoft.com/fr-fr/library/windows/desktop/dd374202(v=vs.85).aspx)>