

Lucas Pavin



Numérisation 3D avec un LIDAR

Membres du groupe :

Ewen Horvais – Romain Hellard – Robin Leparq – Thomas Noel

Sommaire :

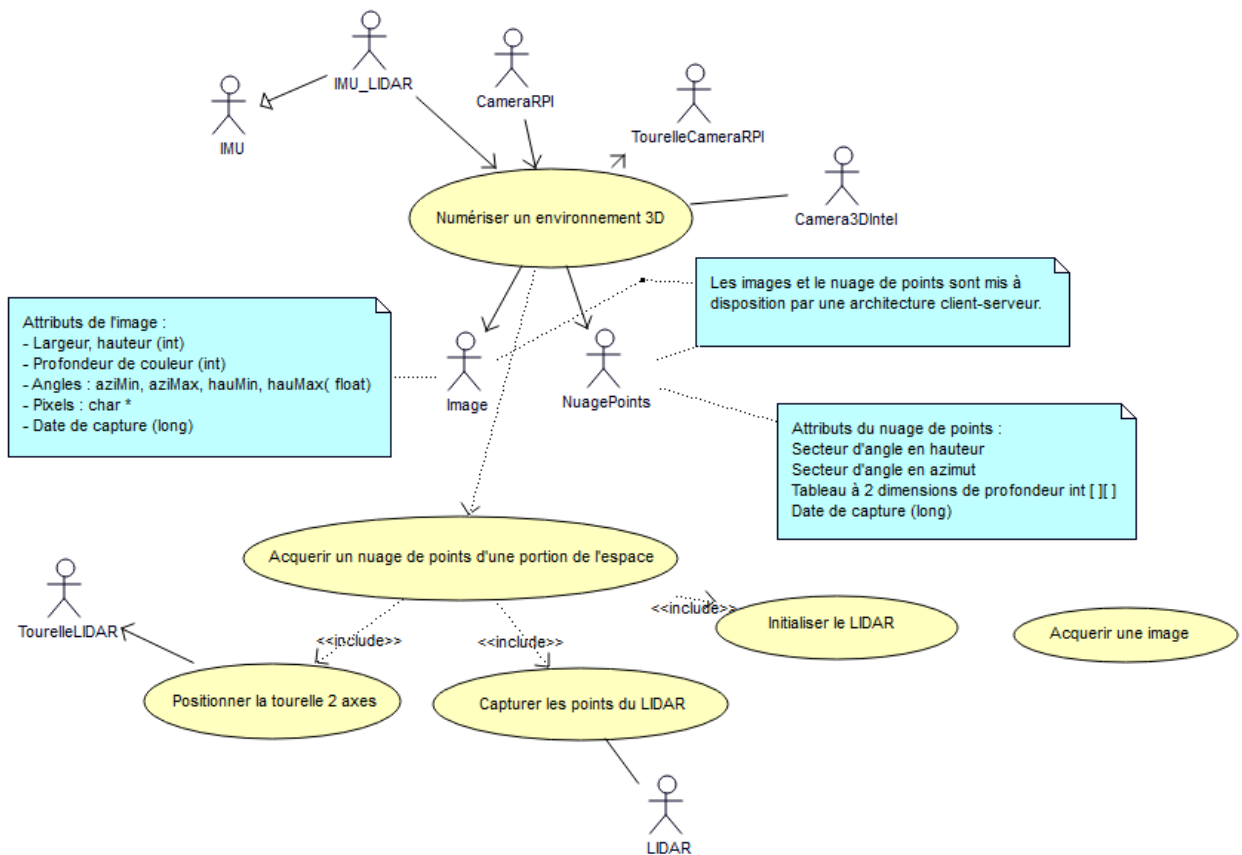
Analyse : Présentation du projet global.....	4
a. Diagramme des cas d'utilisations.....	5
b. Présentation des tâches.....	5
c. Répartition de mon travail (Gantt).....	6
Présentation de ma partie.....	6
a. Présentation du matériel-logiciel.....	6
b. Présentation de mon diagramme des cas d'utilisations.....	7
c. Conception : diagramme des classes.....	7
d. Présentation du Lidar LS01D.....	7
e. Réalisation : test de la connectivité du lidar.....	8
f. Réalisation : test lecture de trame avec Putty.....	9
g. Recherche de la librairie RS232.....	10
Réalisation : RS232.....	10
a. Introduction à la classe RS232.....	10
b. Include/Define RS232.....	11
c. Attribut de la classe RS232.....	11
d. Ouverture du port :.....	11
e. Fermeture du port série.....	12
f. Lecture d'un octet.....	12
g. Lecture d'une trame.....	13
h. Écriture d'un octet :.....	13
i. Écriture d'une trame.....	13
j. Main + Console de la classe RS232.....	13
Réalisation : FichierConfiguration.....	14
a. Introduction à la classe FichierConfiguration.....	14
b. Contenu du fichier texte.....	15
c. Résultat que l'on doit obtenir :.....	15
Réalisation : ParametresRS232.....	16
a. Introduction à la classe ParametresRS232.....	16
b. Méthode chargerParametresRS232.....	16
c. Set() et get().....	17
d. Le constructeur surchargé.....	18
e. Main + Console de la classe ParametresRS232.....	18
Réalisation : DonneesLidar.....	19
a. Introduction à la classe DonneesLidar.....	19
b. Constructeur/Destructeur :.....	19
c. Set() et get().....	20
d. Fonction afficher.....	20
Réalisation : Lidar.....	20
a. Introduction à la classe Lidar.....	20
b. Connecter / déconnecter.....	21
c. Démarrer rotation / acquisition.....	22
d. Stopper acquisition / rotation.....	22

Réalisation : ThreadLidar.....	23
a. Introduction à la classe ThreadLidar.....	23
b. Héritage + attribut.....	23
c. Constructeur ThreadLidar.....	24
d. Méthode principal run().....	24
e. Test unitaire.....	27
Problème rencontré avec la vitesse en bauds.....	28
Conclusion.....	30

Analyse : Présentation du projet global

Le projet global est la conception et la fabrication d'un système informatique autonome permettant la construction automatisée d'une scène 3D sonore à partir d'un environnement réel. Pour l'acquisition des données 3D, deux solutions techniques sont proposées. Un LIDAR* couplé à une caméra (solution 1) ou une caméra 3D (solution 2), sont montés sur une tourelle mobile, numérisent l'environnement proche et fournissent un nuage de points et des images. Ces derniers permettent l'identification de formes géométriques neutres et peu typées, comme des points, des lignes, des surfaces, etc, mais aussi des objets plus complexes et fortement typés comme des portes, des escaliers, des tables, etc. Ces objets typés sont définis dans une librairie. Ces divers objets identifiés grâce aux images peuvent être positionnés dans l'espace grâce au triplet LIDAR/caméra/IMU ou à la caméra 3D. La scène 3D ainsi créée va permettre de constituer une scène 3D sonore équivalente, par association à chaque objet identifié d'une signature sonore. Ainsi, une personne équipée d'un casque audio stéréo et d'un système de capture de l'orientation de sa tête (ou d'un casque de réalité virtuelle) sur 3 axes devra pouvoir percevoir de façon audio et spatialisée son environnement 3D proche. Le système global pourra être utilisé par des personnes mal ou non voyantes, ou encore dans des lieux dépourvus de lumière. Les sous-systèmes décrits plus loin (par exemple le typage des objets par LIDAR et caméra) pourront également être réutilisés dans d'autres projets.

a. Diagramme des cas d'utilisations

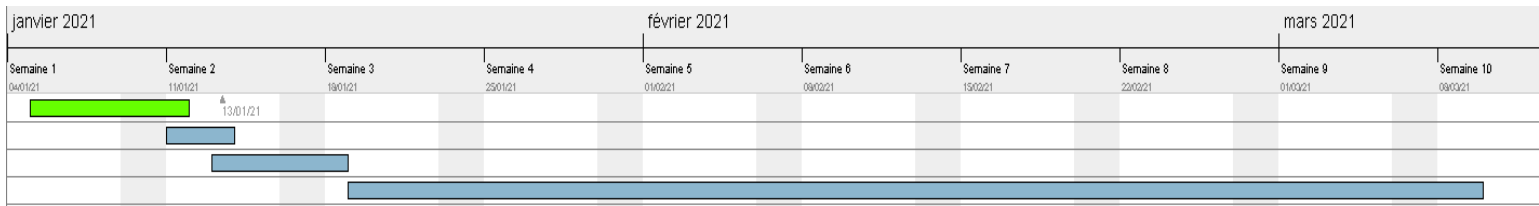


b. Présentation des tâches

Numérisation 3D avec un LIDAR	Spécifications	Étudiant
Acquérir les données du module IMU, calculer les axes – Piloter la tourelle 2 axes	RS232, C++, PWM, (Raspberry)	HORVAIS Ewen
Encapsulation du module LIDAR LS_01 – Acquisition des points – Intégration avec tourelle et l'IMU	RS232, C++ (Raspberry)	PAVIN Lucas
Capture d'images / Extraction de portions d'image / Correction d'image / Assemblage d'images	Image, C++ (Raspberry)	NOEL Thomas
Gérer le nuage de points et les images (système LIDAR/RPI/IMU)	XML, C++ (Raspberry)	LEPARQ Robin
Encapsulation de la caméra Intel – Acquisition d'un nuage de points / acquisition des images	C# ou C++ (PC)	HELLARD Romain

c. Répartition de mon travail (Gantt)

Gantt project		
Nom	Date de début	Date de fin
• Composition des UML	05/01/21	11/01/21
• Renseignement sur les composants	11/01/21	13/01/21
• Recherche d'une librairie pour gérer la RS232 en C++	13/01/21	18/01/21
• Création du code pour RS232 avec la librairie "SerialL...	19/01/21	09/03/21



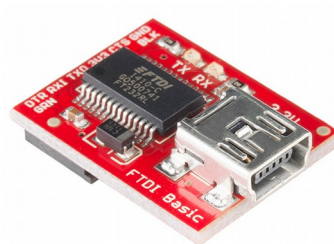
Présentation de ma partie

Ma partie est principalement de pouvoir gérer la gestion du Lidar, mais pour cela il faut que je puisse gérer la RS232, puisque que le lidar sur lequel je travaille est en RS232. Pour gérer cette dernière il faut donc que je puisse écrire et recevoir les données de la trame avec le langage de programmation C++.

a. Présentation du matériel-logiciel



Lidar LS01D Upgrade

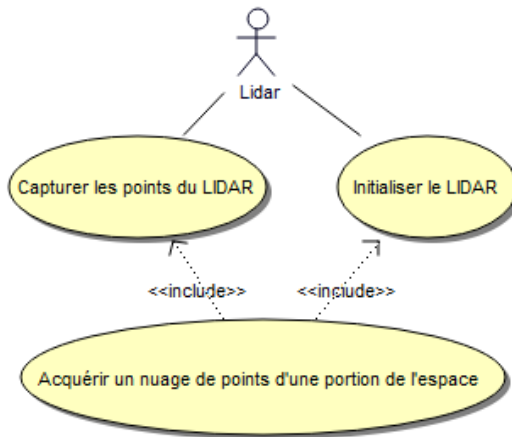


Convertisseur RS232 vers
micro-USB

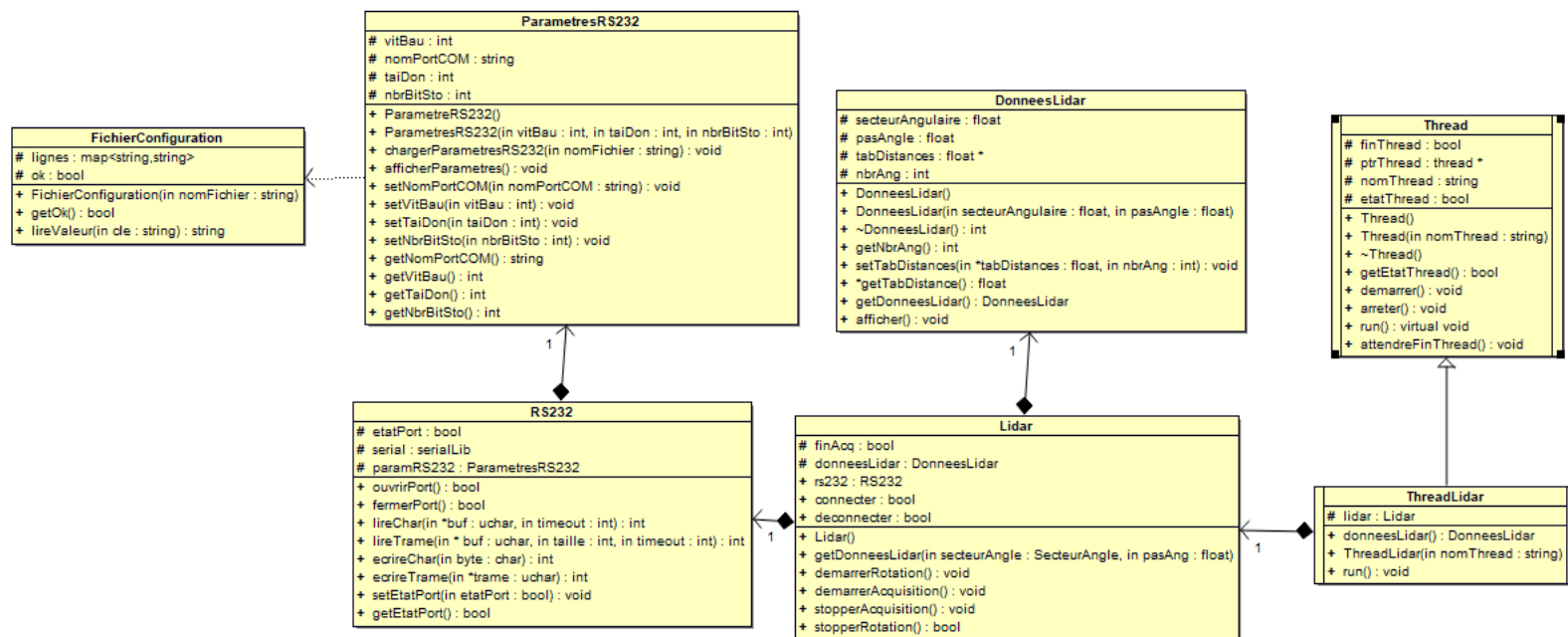


CodeBlocks

b. Présentation de mon diagramme des cas d'utilisations



c. Conception : diagramme des classes



d. Présentation du Lidar LS01D

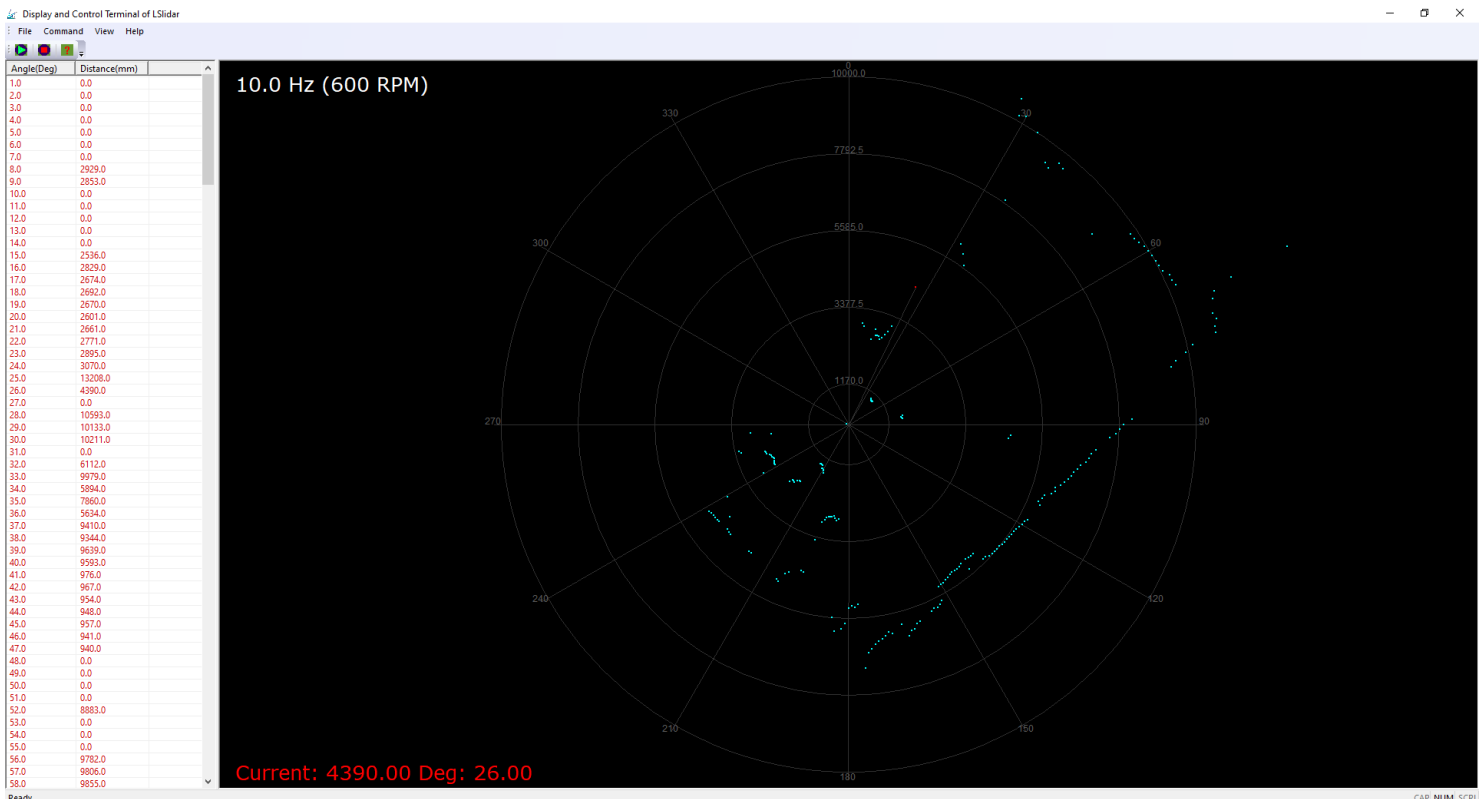
Le lidar LS01D est une solution de scanner laser développée par LeiShen Intelligent System Co. Au niveau de ses caractéristiques le lidar effectue un balayage laser couvrant 360 degrés, et pour une distance minimal de 15 cm et maximal de 8 mètres.

Sa résolution angulaire est de 1°, mais peut également être de 0,5° selon l'envie de la personne qui l'utilise, ceci est personnalisable. Pour ce qui est de sa fréquence d'échantillonnage, il se situe environ entre 3 600 et 4 000 Hz. Sa fréquence d'acquisition est de 10Hz, mais encore une fois elle peut être ajustable entre 3 à 11Hz.

Au niveau du protocole pour ce lidar, il existe plusieurs solutions pour l'interface de communication puisqu'elle peut être constituée soit de I2C, soit d'une RS-232 ou bien une RS-485 etc. Mais ici dans notre projet, nous utilisons de la RS-232 pour ce Lidar.

e. Réalisation : test de la connectivité du lidar

Tout d'abord j'ai commencé par tester si le lidar fonctionnait grâce à une application qui se prénomme « LS01D(V1.02.171213) », cette dernière permet de pouvoir l'allumer ou l'éteindre.



Ainsi on peut visionner à gauche de l'application les angles mesurés en degré allant de 1° à 360°. Ensuite en face de la mesure de l'angle, nous pouvons voir la distance en mm pour laquelle le degré correspond, pour exemple à 9° nous voyons que la distance est de 2853 mm.

9.0	2853.0
-----	--------

Nous pouvons voir dans le grand rectangle noir les points qui ont été acquis par le Lidar, il montre donc une représentation graphique en forme de plan. Ici nous voyons la représentation graphique de notre salle de projet.

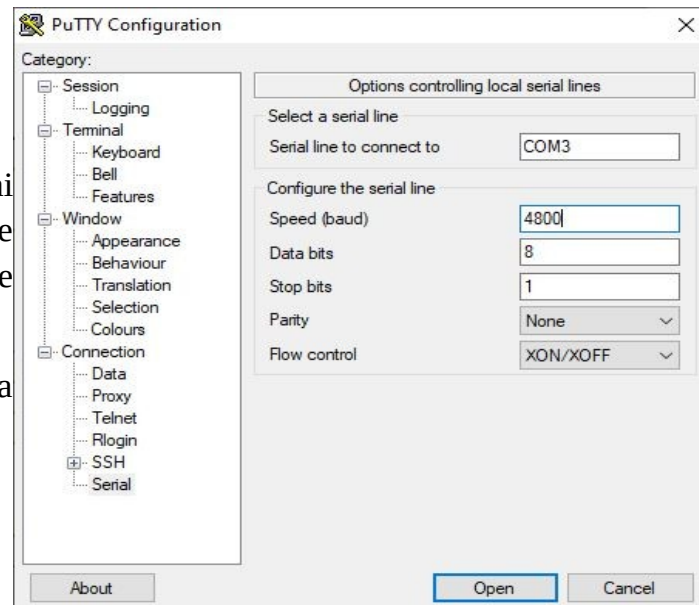
Après avoir effectué ce test on peut donc dire que le Lidar fonctionne, donc à nous de tout programmer maintenant !

f. Réalisation : test lecture de trame avec Putty

Pour pouvoir déterminer le port série sur lequel nous allons travailler, il faut donc ouvrir l'invite de commande, puis taper « mode » là nous verrons défiler tous les modes disponibles sur l'ordinateur.

Premièrement pour pouvoir réaliser ce test, j'ai utilisé un module GPS. Ce dernier m'a permis de pouvoir visualiser ce que donne la réception de trames.

Ce module GPS ne tourne évidemment pas à la même vitesse que le Lidar que nous avons utilisé.



Voilà le résultat que nous pouvons observer sur putty.

```
COM3 - PuTTY
$GPRMC,000024.042,V,,,,,010209,,N*47
$GPVTG,,T,,M,N,K,N*2C
$GPGGA,000025.053,,,,,0,00,,M,0.0,M,,0000*57
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPGSV,1,1,00*79
$GPRMC,000025.053,V,,,,,010209,,N*46
$GPVTG,,T,,M,N,K,N*2C
$GPGGA,000026.040,,,,,0,00,,M,0.0,M,,0000*56
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,000026.040,V,,,,,010209,,N*47
$GPVTG,,T,,M,N,K,N*2C
$GPGGA,000027.040,,,,,0,00,,M,0.0,M,,0000*57
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,000027.040,V,,,,,010209,,N*46
$GPVTG,,T,,M,N,K,N*2C
$GPGGA,000028.052,,,,,0,00,,M,0.0,M,,0000*5B
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,000028.052,V,,,,,010209,,N*4A
$GPVTG,,T,,M,N,K,N*2C
$GPGGA,000029.043,,,,,0,00,,M,0.0,M,,0000*5A
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,000029.043,V,,,,,010209,,N*4B
$GPVTG,,T,,M,N,K,N*2C
```

Nous avons bien reçu les trames, ici on parle bien des trames du module GPS.

g. Recherche de la librairie RS232

Lors de la recherche d'une librairie pour la RS232, j'étais en collaboration avec Ewen Horvais puisque nous avons tous les deux besoins de la classe RS232. Pour pouvoir choisir, nous avons donc privilégié une librairie qui sera bien documentée, une simple d'utilisation, mais également un des critères les plus important : portatif sur Linux. Personnellement j'ai étudié 2 librairies qui se nomme « rs232-master » ainsi que « serialib-master ». Celle qui correspondait le plus à nos critères énoncés auparavant était « serialib-master » donc nous l'avons choisis. Elle équipée d'une documentation détaillée avec sur les différents éléments dont elle est composée.

	Documentation	Simplicité	Portabilité
Serialib-master	***	*****	*****
rs232-master	**	*****	*

Réalisation : RS232

a. Introduction à la classe RS232

J'ai donc commencé par la création de la classe RS232 puisqu'elle est élémentaire pour pouvoir faire fonctionner le Lidar, je me suis donc mis à sa création avec Ewen comme énoncé précédemment vu que nous avons tous les deux besoins de la classe. Elle est donc liée à la classe « FichierConfiguration », puis également par la classe « ParametresRS232 ».

La classe que nous avons fait pour RS232 :

RS232
<pre># etatPort : bool # serial : serialLib # paramRS232 : ParametresRS232 + ouvrirPort() : bool + fermerPort() : bool + lireChar(in *buf : uchar, in timeout : int) : int + lireTrame(in * buf : uchar, in taille : int, in timeout : int) : int + ecrireChar(in byte : char) : int + ecrireTrame(in *trame : uchar) : int + setEtatPort(in etatPort : bool) : void + getEtatPort() : bool</pre>

b. Include/Define RS232

Nous pouvons voir ci-dessous la librairie « serialib.h » donc la librairie que nous avons recherché et travaillé.

Les autres librairies sont là pour pouvoir apporter les éléments manquants pour faire fonctionner le code.

```

/*****Include*****/
#include <iostream>
#include <string>
#include <stdio.h>
#include <thread>    /// Initialisation de la librairie pour pouvoir utiliser les threads
#include "serialib.h"
#include <fstream>   ///Utilisé dans fichierConfiguration
#include <map>

```

c. Attribut de la classe RS232

```

class RS232
{
protected:
    bool etatPort;
    serialib serial;
    ParametresRS232 paramRS232;

```

Les éléments sont protégés pour pouvoir les réutilisés dans la classe en questions, mais sont pas utilisable en dehors de la classe, ce qui permet une sécurité.

d. Ouverture du port :

```

public:
bool ouvrirPort()    ///Fonction ouverture du port
{
    paramRS232.chargerParametresRS232("parametresRS232");
    if(serial.openDevice(paramRS232.getNomPortCOM().c_str(), paramRS232.getVitBau())==1)    /// Si le port série est connecté COM"NB"
        ///avec une vitesse de : n
    {
        printf("\nOuverture du port : %s\n", paramRS232.getNomPortCOM().c_str());    ///Affiche un message de connexion
        etatPort = true;    ///Renvoie vrai au bool etatPort
        return true;    ///Retourne vrai
    }
    else
    {
        printf("\nErreur d'ouverture du port : %s\n", paramRS232.getNomPortCOM().c_str());
        etatPort = false;    ///Sinon renvoie faux au bool etatPort
        return false;    ///Retourne faux
    }
}
}

```

Comme nous pouvons le voir on vient chercher la méthode de la classe ParametresRS232, qui se nomme « **chargerParametresRS232**(« **parametresRS232** ») », nous allons donc voir l'intérieur de la fonction ParametresRS232 prochainement, pour essayer de comprendre.

e. Fermeture du port série

```

bool fermerPort()    ///Fonction fermeture du port
{
    if(etatPort==true)
    {
        serial.closeDevice();
        printf("\nFermeture du port : %s\n\n", paramRS232.getNomPortCOM().c_str());
        etatPort = false;
        return true;
    }
    else
        return false;
}

```

f. Lecture d'un octet

```

int lireChar(char * buf, int timeout)
{
    return serial.readChar(buf, timeout);    /// On vient directement prendre dans la librairie
    ///Serial la fonction readChar();
}

```

g. Lecture d'une trame

```
int lireTrame (unsigned char *buf, int taille , int timeout)
{
    return serial.readBytes(buf, taille, timeout);  /// On vient directement prendre dans la librairie
                                                    ///Serial la fonction readBytes();
}
```

h. Écriture d'un octet :

```
int ecrireChar(char byte)
{
    if(etatPort == true)
    {
        cout << "\n-----Ecrire OCTET-----\n";
        serial.writeChar(byte);
        cout << byte;
    }
}
```

i. Écriture d'une trame

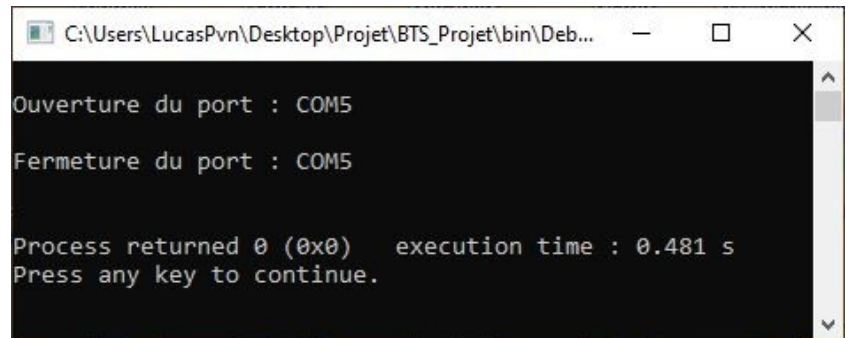
```
int ecrireTrame(unsigned char *trame, int taille)
{
    if(etatPort == true)
    {
        serial.writeBytes(trame, taille);
    }
}
```

j. Main + Console de la classe RS232

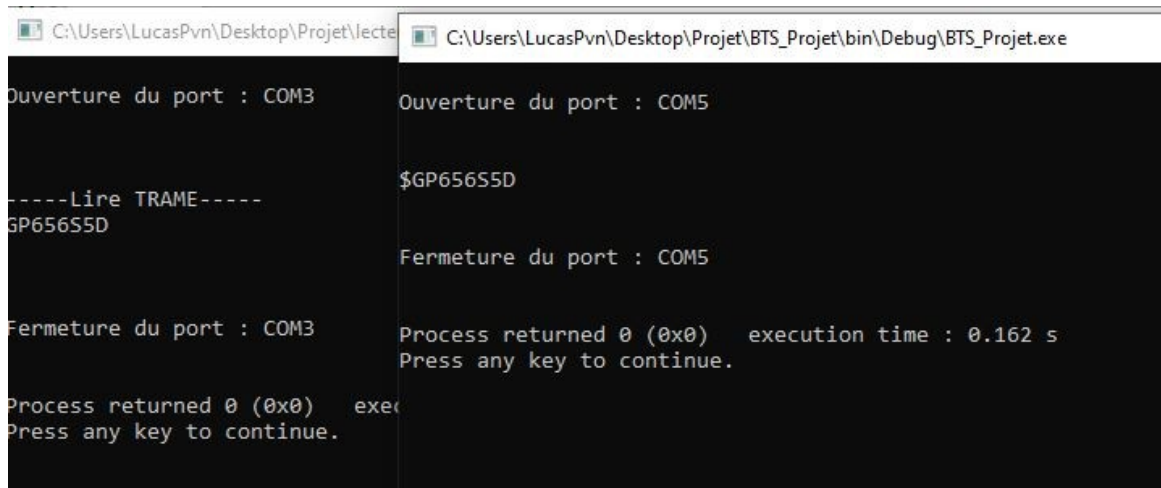
Pour ces tests unitaires nous avons donc utilisé deux CodeBlocks pour pouvoir écrire un octet ou bien une trame sur l'un, et sur l'autre mettre le code pour lire afin de voir si les informations sont bien envoyées, ainsi que reçues.

```
int main()
{

    RS232 rs232;
    rs232.ouvrirPort();
    rs232.fermerPort();
}
```

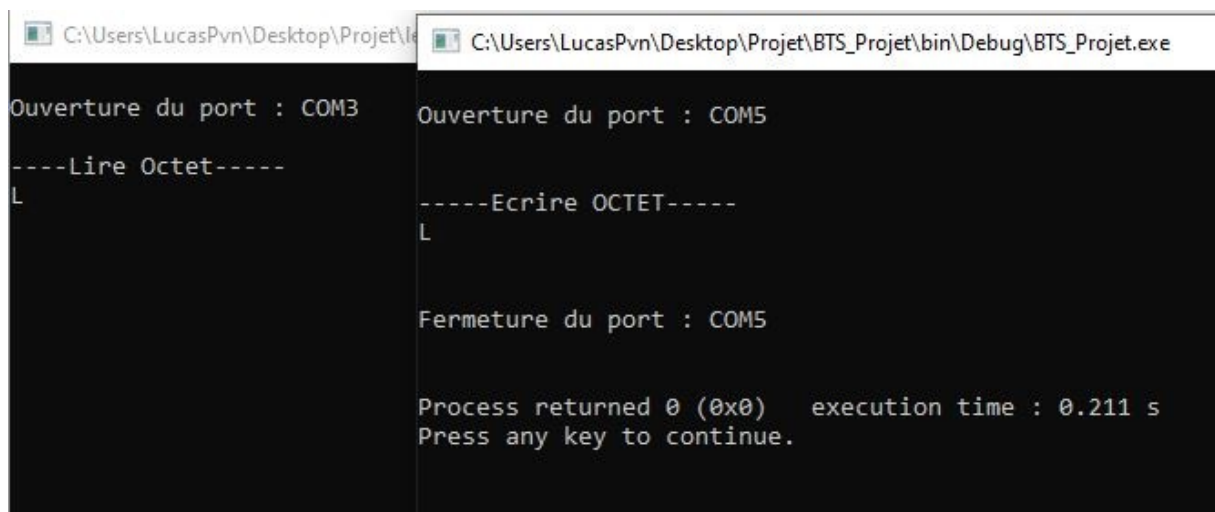


```
C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Deb...
Ouverture du port : COM5
Fermeture du port : COM5
Process returned 0 (0x0) execution time : 0.481 s
Press any key to continue.
```



```
C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Debug\BTS_Projet.exe
Ouverture du port : COM3
-----Lire TRAME-----
GP65655D
Fermeture du port : COM3
Process returned 0 (0x0) execution time : 0.162 s
Press any key to continue.

C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Debug\BTS_Projet.exe
Ouverture du port : COM5
$GP65655D
Fermeture du port : COM5
Process returned 0 (0x0) execution time : 0.162 s
Press any key to continue.
```



```
C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Debug\BTS_Projet.exe
Ouverture du port : COM3
-----Lire Octet-----
L
Fermeture du port : COM3
Process returned 0 (0x0) execution time : 0.211 s
Press any key to continue.

C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Debug\BTS_Projet.exe
Ouverture du port : COM5
-----Ecrire OCTET-----
L
Fermeture du port : COM5
Process returned 0 (0x0) execution time : 0.211 s
Press any key to continue.
```

Réalisation : FichierConfiguration

a. Introduction à la classe FichierConfiguration

La classe FichierConfiguration est une classe permettant de pouvoir modifier les données de la RS232 c'est-à-dire le choix du port série, ou bien la vitesse en bauds, ou le nombre de bits de données ainsi que le nombre de bits de stop depuis l'extérieur du

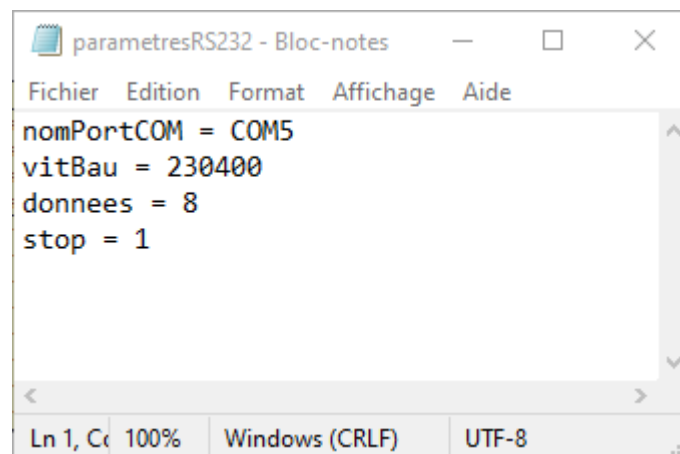
code. Cela permet à un utilisateur de pouvoir modifier sans même aller dans le code du programme. Cela permet de gagner en gain de temps également.

Il faut savoir que cette classe n'a pas été faite de mes propres soins, c'est une classe que Ewen a configuré et me l'a donné puisque j'en avais besoin par la suite.

FichierConfiguration
lignes : map<string,string>
ok : bool
+ FichierConfiguration(in nomFichier : string)
+ getOk() : bool
+ lireValeur(in cle : string) : string

b. Contenu du fichier texte

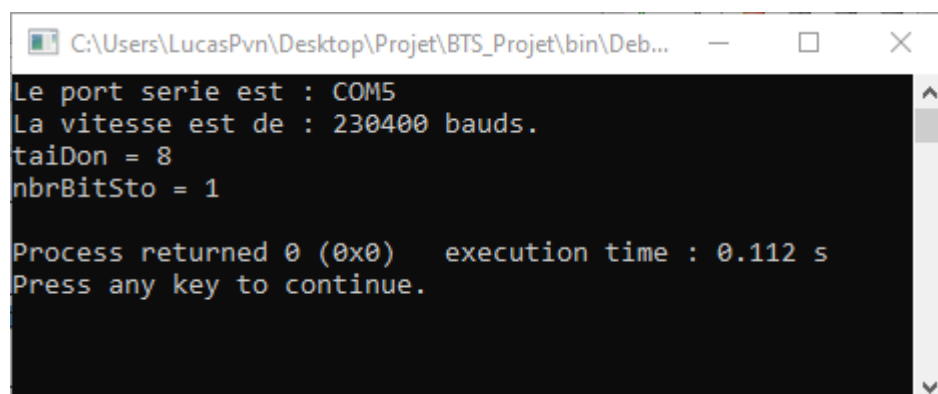
Voici le contenu de la classe, cette dernière permet de pouvoir lire le contenu que nous avons créé en dehors du code sur le bloc note en « .txt », le contenu de celui est le suivant :



```

Fichier  Edition  Format  Affichage  Aide
nomPortCOM = COM5
vitBau = 230400
donnees = 8
stop = 1
  
```

c. Résultat que l'on doit obtenir :



```

C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Deb...
Le port serie est : COM5
La vitesse est de : 230400 bauds.
taiDon = 8
nbrBitSto = 1

Process returned 0 (0x0)   execution time : 0.112 s
Press any key to continue.
  
```

Réalisation : ParametresRS232

a. Introduction à la classe ParametresRS232

Le but principal de la classe ParametresRS232 est par la suite de pouvoir avoir une concordance avec la classe FichierConfiguration, pour ensuite pouvoir les intégrer à la liaison série afin de mettre ces valeurs en paramètres.

ParametresRS232
vitBau : int # nomPortCOM : string # taiDon : int # nbrBitSto : int
+ ParametreRS232() + ParametresRS232(in vitBau : int, in taiDon : int, in nbrBitSto : int) + chargerParametresRS232(in nomFichier : string) : void + afficherParametres() : void + setNomPortCOM(in nomPortCOM : string) : void + setVitBau(in vitBau : int) : void + setTaiDon(in taiDon : int) : void + setNbrBitSto(in nbrBitSto : int) : void + getNomPortCOM() : string + getVitBau() : int + getTaiDon() : int + getNbrBitSto() : int

b. Méthode chargerParametresRS232

```
void chargerParametresRS232(string nomFichier)
{
    FichierConfiguration fg("parametresRS232.txt"); //On vient demander le nom du fichier à récupérer.
    if (fg.getOk())
    {
        this->nomPortCOM = fg.lireValeur("nomPortCOM"); //On met la valeur mise dans le fichier dans this->nomPortCOM
        this->vitBau = stoi(fg.lireValeur("vitBau")); //On met la valeur mise dans le fichier dans this->vitBau
        this->taiDon = stoi(fg.lireValeur("donnees")); //On met la valeur mise dans le fichier dans this->taiDon
        this->nbrBitSto = stoi(fg.lireValeur("stop")); //On met la valeur mise dans le fichier dans this->nbrBitSto
        //Stoi convertit les strings en entier.
    }
}
```


La méthode ici fait appel à la classe FichierConfiguration en faisant afficher le contenu du fichier **.txt**. Il sera par la suite affiché grâce à **fg.lirevaleur()** qui recherche le contenu souhaité.

Il faut veiller à bien respecter que les mots inscrits à l'intérieur de **fg.lirevaleur(« ... »)** soient bien les mêmes que ceux mis dans le fichier **.txt**, sinon cela va poser problème lors de la lecture de ce dernier.

c. Set() et get()

On va voir maintenant les **set()** et les **get()** que j'ai utilisé dans la classe ParametresRS232. J'ai donc condensé en ouvrant que un **set()** et qu'un **get()**, mais ce que j'ai développé dans la classe **void setNomPortCom(string nomPortCOM)** est le principe pour les autres **set()** de cette classe, le seul point qui change à l'intérieur est le nom du paramètre. Même principe également pour les **get()**, puisque j'ai également fait la même chose dans les autres **get()** que dans **string getNomPortCOM()**. Encore une fois la seule chose qui change est le paramètre à renvoyer, ainsi que le type de la fonction (int, string...).

```
void setNomPortCom (string nomPortCOM)
{
    this->nomPortCOM = nomPortCOM;
}
void setVitBau (int vitBau)
{
}
void setTaiDon (int taiDon)
{
}
void setNbrBitSto (int nbrBitSto)
{
}
string getNomPortCOM ()
{
    return nomPortCOM;
}
int getVitBau ()
{
}
int getTaiDon ()
{
}
int getNbrBitSto ()
{
}
```

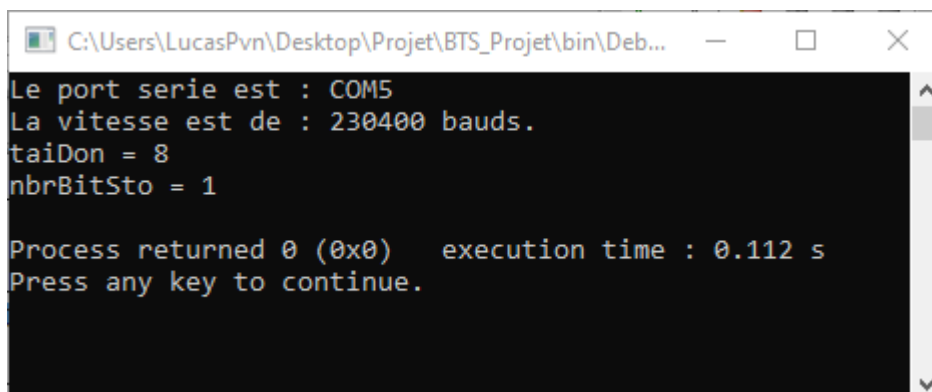
d. Le constructeur surchargé

```
ParametreRS232(string nomPortCOM, int vitBau, int taiDon, int nbrBitSto)
{
    setNomPortCom(nomPortCOM);
    setVitBau(vitBau);
    setTaiDon(taiDon);
    setNbrBitSto(nbrBitSto);
}
```

Dans un premier temps j'ai utilisé le constructeur précédent pour pouvoir réaliser les tests. Le défaut de ce constructeur est que les valeurs des paramètres doivent être spécifiées directement à partir du code source, ce qui nécessite une recompilation du code, en cas de modification du port COM ou bien de la vitesse en bauds. C'est donc pour cela que dans un second temps j'ai décidé de créer la méthode ***chargeParametresRS232(string nomFichier)***, ce qui permet de modifier sans recompilation du code source les valeurs du fichier texte.

e. Main + Console de la classe ParametresRS232

```
int main()
{
    ParametresRS232 paramRS232;
    paramRS232.chargerParametresRS232("parametresRS232");
    paramRS232.afficherParametre();
}
```



```
C:\Users\LucasPvn\Desktop\Projet\BTS_Projet\bin\Deb...
Le port serie est : COM5
La vitesse est de : 230400 bauds.
taiDon = 8
nbrBitSto = 1

Process returned 0 (0x0)   execution time : 0.112 s
Press any key to continue.
```

Nous pouvons voir que nous retrouvons bien le contenu émis dans le fichier texte.

Réalisation : DonneesLidar

a. Introduction à la classe DonneesLidar

La classe DonneesLidar permettra par la suite de pouvoir choisir un secteur angulaire si on veut capturer les points que dans un certain endroit angulaire. Elle permet aussi de régler le pas de l'angle, ici il est de 1, donc nous irons de 0° à 1° puis 2°... Ensuite on mettra les résultats obtenus dans un tableau.

DonneesLidar
secteurAngulaire : float # pasAngle : float # tabDistances : float * # nbrAng : int
+ DonneesLidar() + DonneesLidar(in secteurAngulaire : float, in pasAngle : float) + ~DonneesLidar() : int + getNbrAng() : int + setTabDistances(in *tabDistances : float, in nbrAng : int) : void + *getTabDistance() : float + getDonneesLidar() : DonneesLidar + afficher() : void

b. Constructeur/Destructeur :

```
DonneesLidar(float secteurAngulaire, float pasAngle)
{
    this->secteurAngulaire = secteurAngulaire;
    this->pasAngle = pasAngle;
    this->nbrAng = this->secteurAngulaire / this->pasAngle;
    this->tabDistances = new float[nbrAng]; /// Allouer un tableau dynamique adapté au LIDAR

    cout << "Const donneesLidar " << nbrAng << " " << this->tabDistances << endl;
}
~DonneesLidar() /// Destruction du constructeur DonneesLidar
{
    delete this->tabDistances; /// Supprimer le tableau dynamique
}
```

Nous avons donc instancié un constructeur surchargé qui permet de mettre en paramètre le secteur angulaire qu'on désire, ainsi que le pas de l'angle qu'on désire (le pas d'angle par défaut est de 1°). Ensuite on fait une division entre « *this->secteurAngulaire* » et « *this->pasAng* » ce qui donne ensuite le nombre d'angle, on vient par la suite mettre la valeur dans un tableau dynamique adapté qu'au lidar, prénommé « *tabDistances* ».

En deuxième lieu une fois le constructeur effectué, il faut détruire le tableau mais proprement donc j'ai écrit un destructeur qui permet cela.

c. Set() et get()

```
int getNbrAng()
{
    return this->nbrAng;
}
float *getTabDistances()
{
    return this->tabDistances;
}
void setTabDistances(float *tabDistances, int nbrAng)  /// Pour la classe LIDAR
{
    /// lock_mutex
    for (int i=0; i<nbrAng; i++)
        this->tabDistances[i] = tabDistances[i];
    /// unlock_mutex
}
```

Le « *setTabDistances* » permet de créer un tableau à l'aide d'une boucle « *for()* », ce dernier va mettre les distances que l'on découvre dans le tableau, la taille du tableau dépend totalement du nombre d'angles inclus en paramètres. Il faudra par la suite faire un mutex pour permettre de ne pas avoir de fausses valeurs qui se joignent dans le tableau.

d. Fonction afficher

```
void afficher()
{
    cout << "Donnees LIDAR : \n";
    for (int i=0; i<nbrAng; i++)
        cout << "dist[" << i << "] = " << this->tabDistances[i] << '\n';
}
```

J'ai créé encore une boucle « *for()* » pour venir afficher les distances mises dans le tableau. Dans le « *cout << »* nous venons afficher la valeur à 0° pour exemple : distance[0] = (la distance trouvée par le lidar).

Réalisation : Lidar

a. Introduction à la classe Lidar

La classe Lidar permet de faire fonctionner dans la globalité le Lidar, elle permet de démarrer et arrêter. Tout ça se fait grâce à l'envoi de trame spécifique. Elle s'appuie principalement sur la classe RS232 créée auparavant, et oui elle était bien essentielle pour la suite du projet.

Lidar
<pre># finAcq : bool # donneesLidar : DonneesLidar + rs232 : RS232 + connecter : bool + deconnecter : bool + Lidar() + getDonneesLidar(in secteurAngle : SecteurAngle, in pasAng : float) + demarrerRotation() : void + demarrerAcquisition() : void + stopperAcquisition() : void + stopperRotation() : bool</pre>

b. Connecter / déconnecter

```
public :
    RS232 rs232;
    bool connecter()
    {
        return rs232.ouvrirPort();
    }
    bool deconnecter()
    {
        return rs232.fermerPort();
    }
```

Nous avons passé RS232 en public pour pouvoir aller chercher les méthodes, cela ne pose pas trop de problèmes de sécurité.

Dans la méthode « **connecter()** » on vient chercher la classe RS232 et la méthode ouvrirPort() de cette dernière, on va donc venir demander de retourner si le port est ouvert ou non.

Ensuite pour la méthode « **déconnecter()** » on vient également chercher la classe RS232, ainsi que la méthode fermerPort(), on va donc venir demander la valeur de retour de fermerPort.

c. Démarrer rotation / acquisition

```
void demarrerRotation()
{
    unsigned char trameDemRotatLidar[] = {0xA5, 0x2C, 0xE1, 0xAA, 0xBB, 0xCC, 0xDD}; //Trame démarrage de la rotation "0x2C"
    rs232.ecrireTrame(trameDemRotatLidar, 15); //Ecrire la trame
    cout << "La trame de mise en rotation vient d'etre envoyee.... \n\n" ; //Message de prevention
}
```

La méthode « **demarrerRotation()** » permet comme son nom l'indique de démarrer la rotation du lidar, pour cela il faut écrire une trame « **0xA5 0x2C 0xE1 0xAA 0xBB 0xCC 0xDD** » dans un « **unsigned char** ». Nous faisons ensuite appelle à la méthode « **ecrireTrame** » de la classe « **RS232** », et on passe en paramètre la trame de rotation, ainsi que la taille de la trame envoyé, ici elle est de 15. Un message est envoyé en prévention pour indiquer que la trame de mise en rotation a été envoyée.

```
void demarrerAcquisition()
{
    unsigned char trameDemAcq[] = {0xA5, 0x20, 0xE1, 0xAA, 0xBB, 0xCC, 0xDD}; //Trame démarrage d'acquisition "0x20"
    rs232.ecrireTrame(trameDemAcq, 15); //Ecrire la trame
    cout << "La trame d'acquisition vient d'etre envoyee.... \n\n" ; //Message de prevention
}
```

La méthode « **demarrerAcquisition()** » permet comme son nom l'indique de démarrer l'acquisition de points du lidar, pour cela il faut écrire une trame « **0xA5 0x20 0xE1 0xAA 0xBB 0xCC 0xDD** » dans un « **unsigned char** ». Nous faisons ensuite appelle à la méthode « **ecrireTrame** » de la classe « **RS232** », et on passe en paramètre la trame d'acquisition, ainsi que la taille de la trame envoyée, ici elle est de 15. Un message est envoyé en prévention pour indiquer que la trame d'acquisition a été envoyée.

d. Stopper acquisition / rotation

```
void stopperAcquisition()
{
    unsigned char trameStopAcq[] = {0xA5, 0x21, 0xE1, 0xAA, 0xBB, 0xCC, 0xDD}; //Trame de stop acquisition "0x21"
    rs232.ecrireTrame(trameStopAcq, 15); // Ecrire trame
    cout << "Fin de l'acquisition." << endl; // Message de prevention
}
```

La méthode « **stopperAcquisition()** » permet comme son nom l'indique de stopper proprement l'acquisition de points du lidar, pour cela il faut écrire une trame « **0xA5 0x21 0xE1 0xAA 0xBB 0xCC 0xDD** » dans un « **unsigned char** ». Nous faisons ensuite appel à la méthode « **ecrireTrame** » de la classe « **RS232** », et on passe en paramètre la trame de fin d'acquisition, ainsi que la taille de la trame envoyée, ici elle est de 15. Un message est envoyé en prévention pour indiquer que la trame de fin d'acquisition a été envoyée.

```
bool stopperRotation()
{
    unsigned char trameStop[] = {0xA5, 0x25, 0xE1, 0xAA, 0xBB, 0xCC, 0xDD}; //Trame de stop Rotation "0x25"
    rs232.ecrireTrame(trameStop, 15); //Ecrire Trame
    cout << "Fin de la rotation." << endl; //Message de fin
}
```

La méthode « **stopperRotation()** » permet comme son nom l'indique de stopper proprement l'acquisition de points du lidar, pour cela il faut écrire une trame « **0xA5 0x25 0xE1 0xAA 0xBB 0xCC 0xDD** » dans un « **unsigned char** ». Nous faisons ensuite appel à la méthode « **ecrireTrame** » de la classe « **RS232** », et on passe en paramètre la trame de fin de rotation, ainsi que la taille de la trame envoyée ici elle est de 15. Un message est envoyé en prévention pour indiquer que la trame de fin d'acquisition a été envoyée.

Réalisation : ThreadLidar

a. Introduction à la classe ThreadLidar

La classe ThreadLidar s'appuie grâce à un héritage à la classe thread donné par windows. ThreadLidar permet de pouvoir avoir tout qui fonctionne en continue dans un sens de priorité. Ici ThreadLidar permettra de démarrer la rotation – l'acquisition, ensuite permettra de lire les trames reçues, pour finir il mettra fin à l'acquisition et la rotation.

b. Héritage + attribut

```
class Thread
{
    class ThreadLidar : public Thread
    {
    protected :
        Lidar lidar;
    public :
```

Comme nous pouvons le voir la classe ThreadLidar est la classe fille de Thread grâce à « **class ThreadLidar : public Thread** ». Comme évoqué au dessus je n'ai pas détaillé la classe thread puisque ce n'est pas la mienne, c'est une classe trouvée sur internet.

Également nous pouvons observer que ThreadLidar s'appuie sur la classe Lidar, le lien entre les deux est une agrégation forte.

c. Constructeur ThreadLidar

```
ThreadLidar(string nomThread) : Thread(nomThread)
{
}
```

Le constructeur ThreadLidar a pour paramètre « **string nomThread** » et s'appuie d'une méthode déjà créée dans la classe Thread.

Pour comprendre la démarche de cette méthode nous allons aller regarder la méthode de la classe Thread.

```
Thread(string nomThread)
{
    this->nomThread = nomThread;
    this->ptrThread = NULL;
}
```

Nous pouvons donc voir que le créateur de la classe a fait en sorte que notre thread peut porter un nom afin de se différencier des autres threads qui vont tourner à la fin sur l'application qui sera raspberry.

d. Méthode principale run()

J'ai donc créé une méthode qui fera tourner le lidar, ainsi qu'acquérir les points, les lises pendant un certain moment, puis mettre fin à tout.

```
void run()
{
    if (lidar.connecter()==true)
    {
        lidar.demarrerRotation();
        Sleep(100);
        lidar.demarrerAcquisition();
        Sleep(500);
        unsigned char buf;
        unsigned char trame[1806]; ///1806 est la longueur de la trame 6 bits d'entête et 1800 points de données
        float tabDistances[360]; /// = NULL;
    }
}
```


Pour que le thread puisse fonctionner j'ai posé une condition avec le « **if** » donc avec pour condition d'exécution « **if(lidar.connecter==true)** », cela signifie que si la fonction connecter de la classe Lidar arrive bien à se connecter au port série alors on peut démarrer la boucle.

Ensuite si la condition est respectée on lance la fonction « **démarrerRotation** » de la classe Lidar, j'ai appliqué une petite pause de 100ms à l'aide du sleep afin de ne pas démarrer l'acquisition avant même que la rotation soit bien lancée. Une fois la pause finie, on démarre la méthode « **démarrerAcquisition** ». Après le sleep de 500 ms cette fois, j'ai créé deux unsigned char afin de pouvoir lire les valeurs. « **Trame[1806]** » correspond à 6bits d'entête pour la trame, ainsi que les 1800 de données.

Le « **float tabDistance[360]** » est un tableau de 360, correspondant aux 360 points que nous voulons.

```
while (!finThread)
{
    int c = lidar.rs232.lireChar((char *)&buf, 1000);
    if (buf==0xA5)
    {
        int angle, dist;
        c = lidar.rs232.lireTrame(trame, 1806, 1000); //1806 est la longueur de la trame 6 bits d'entête
                                                    //et 1800 points de données

        for (int i = 0; i<360; i++)
        {
            angle = ((trame[6+1+5*i]+256*trame[6+2+5*i])/10)-1;
            if (angle<0 || angle>359)
            {
                angle = 0;
            }
            dist = trame[6+3+5*i]+256*trame[6+4+5*i];
            if (angle==359)
                cout << "distance = " << dist << "\n";
            tabDistances[angle]=dist;
        }
    }
}
```

La boucle ici est un tant que c'est pas la fin du thread la fonction va tourner.

Donc après j'ai été chercher la fonction lireTrame de la classe RS232 à partir de la classe Lidar. Donc là je suis venu chercher caractère par caractère les éléments de la trame, puis les placer dans c. Ensuite une autre fonction rentre en jeu c'est la fonction « **if(buff==0xA5)** », donc 0xA5 qui correspond au première octet de la trame. Une fois le caractère trouvé on vient recopier dans c la trame avec comme paramètres à la méthode « **lireTrame** » le « **unsigned char trame[1806]** » puis la longueur de la trame donc toujours 1806, et pour finir le timeout qui est de 1000ms.

La boucle « **for()** » permet de mettre les valeurs dans un tableau de 360. Donc dans le tableau je suis venu mettre dans

J'ai vu dans la documentation du lidar que le 1 et le 2 byte étaient réservés à l'angle. (voir ci-dessous).

Byte No.	Data	Description
0	flag	Flag bit, fixed as 0xff
1	angle (7:0)	8 bytes for low angle
2	angle (15:8)	8 bytes for high angle
3	distance (7:0)	8 bytes for low distance
4	distance (15:8)	8 bytes for high distance

(LS01D Lidar Communication Interface Protocol and Application Manual).

C'est donc pour cela que dans angle on met « **((trame[6+1+5*i] +256*trame[6+2+5*i])/10)-1** », 6 correspond au 6 bits d'entête, 1 et 2 correspondent byte de l'angle (voir tableau ci-dessus), et 5*i c'est pour que cela se répète à la même position pendant le nombre de points demandés, /10 est pour remettre le tout sur 360, et le -1 car sur le tableau les points vont de 0° à 359°. Le « **+256** » c'est parce que le bytes est plus fort donc $2^8=256$.

Par la suite j'ai mis un « **if** » car si les valeurs sont plus petite que 0° ou plus grande que 359°, on mette systématiquement 0 à la valeur de angle.

Ensuite avec la distance c'est le même principe que pour les angles, sauf que comme vu dans le tableau mettant c'est plus 1 et 2, mais 3 et 4.

Pour finir on vient à l'intérieur de la boucle « **if** » afficher toutes les distances correspondantes à l'angle 359°.

On vient aussi mettre dans le tableau d'angle la distance correspondante.

```
lidar.stopperAcquisition();
Sleep(500);
lidar.stopperRotation();
lidar.deconnecter();
```

Une fois que j'ai fini la fonction « ***while(!finThread)*** » on vient stopper l'acquisition de points, puis attendre 500 ms pour ensuite stopper la rotation du Lidar et se déconnecter du port série. La fin est importante pour pouvoir mettre fin proprement au lidar.

```
else
{
    cout << "Erreur de connexion au lidar\n";
}
```

En cas d'erreur d'ouverture du port série, j'ai mis un petit « ***else*** » qui viendra afficher le message « ***Erreur de connexion au lidar.*** ».

e. Test unitaire

```
int main()
{

    ThreadLidar thLidar("Thread Lucas");
    thLidar.demarrer();
    Sleep(15000);
    thLidar.arreter();
    exit(0);
}
```

```

Ouverture du port : COM5
La trame de démarrage vient d'être envoyée...
La trame d'acquisition vient d'être envoyée...

TabDistances = 0x6ef190
distance = 1990
distance = 1989
distance = 2003
distance = 1995
distance = 1991
distance = 1993
distance = 1997
distance = 1977
distance = 1977
distance = 1997
distance = 1988
distance = 1989
distance = 1999
distance = 1995
distance = 1988
distance = 1973
distance = 1994
distance = 1978
distance = 1977
distance = 2001
distance = 1978
distance = 1997

```

Problème rencontré avec la vitesse en bauds

Ce qu'il faut c'est que comme dit précédemment le Lidar à une vitesse de 230400 bauds, or j'ai rencontré le problème que la librairie ne disposait pas d'une vitesse aussi grande, alors j'ai dû changer quelque élément dans « *serialib.cpp* ».

J'ai d'abord dû mettre un « *#define CBR_230400 230400* ».

```
#define CBR_230400 230400
```

Deuxièmement la rajouter parmi les autres vitesses.

```

switch (Bauds)
{
case 110 : dcbSerialParams.BaudRate=CBR_110; break;
case 300 : dcbSerialParams.BaudRate=CBR_300; break;
case 600 : dcbSerialParams.BaudRate=CBR_600; break;
case 1200 : dcbSerialParams.BaudRate=CBR_1200; break;
case 2400 : dcbSerialParams.BaudRate=CBR_2400; break;
case 4800 : dcbSerialParams.BaudRate=CBR_4800; break;
case 9600 : dcbSerialParams.BaudRate=CBR_9600; break;
case 14400 : dcbSerialParams.BaudRate=CBR_14400; break;
case 19200 : dcbSerialParams.BaudRate=CBR_19200; break;
case 38400 : dcbSerialParams.BaudRate=CBR_38400; break;
case 56000 : dcbSerialParams.BaudRate=CBR_56000; break;
case 57600 : dcbSerialParams.BaudRate=CBR_57600; break;
case 115200 : dcbSerialParams.BaudRate=CBR_115200; break;
case 128000 : dcbSerialParams.BaudRate=CBR_128000; break;
case 230400 : dcbSerialParams.BaudRate=CBR_230400; break;
case 256000 : dcbSerialParams.BaudRate=CBR_256000; break;
default : return -4;
}

```

Et pour finir rajouter aussi la vitesse 230400, ici :

```

switch (Bauds)
{
case 110 : Speed=B110; break;
case 300 : Speed=B300; break;
case 600 : Speed=B600; break;
case 1200 : Speed=B1200; break;
case 2400 : Speed=B2400; break;
case 4800 : Speed=B4800; break;
case 9600 : Speed=B9600; break;
case 19200 : Speed=B19200; break;
case 38400 : Speed=B38400; break;
case 57600 : Speed=B57600; break;
case 115200 : Speed=B115200; break;
case 230400 : Speed=B230400; break;
default : return -4;
}

```

Et pour finir mettre le « **#define CBR_230400 230400** » dans le main.cpp, afin de pouvoir configurer la nouvelle vitesse en bauds dans le « **serialib.cpp** ».

```
#define CBR_230400 230400
```

Conclusion

En conclusion je peux dire que le projet dans sa globalité est presque fini puisque la plus part des classes fonctionnent, et nous arrivons bien à acquérir les distances correspondantes aux angles. On arrive également à faire tourner le thread qui permet de faire tourner le lidar, d'acquérir les points, de recevoir les trames souhaitées, d'en ressortir les éléments demandés. Il y a quelques points encore à peaufiner.