



# Projet Caméra

Projet d'informatique

GIRAUD-CARRIER Ewen  
GAUDET Pierre  
BOISSENIN Enzo

Professeur : J. RODRIGUEZ

Date  
Année scolaire 2023/2024

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>La caméra Sony EVI D-70P</b>	<b>4</b>
<b>3</b>	<b>Le code</b>	<b>6</b>
3.1	Structure du Code . . . . .	7
3.1.1	Fichiers Principaux . . . . .	7
3.1.2	Gestion du Clavier . . . . .	7
3.1.3	Contrôle de la Caméra . . . . .	7
3.1.4	Composition des Commandes . . . . .	7
3.1.5	Communication Série . . . . .	7
3.2	Fonctionnement des différents modules . . . . .	8
3.2.1	Cas du PC . . . . .	9
3.2.2	Cas de l'Arduino . . . . .	9
3.3	Lire la touche pressée . . . . .	9
3.3.1	Sur PC . . . . .	10
3.3.2	Sur Arduino . . . . .	11
3.4	Le fichier central . . . . .	12
3.5	La classe camera.h . . . . .	13
3.6	Construction de la commande Visca . . . . .	14
3.7	Envoi de la commande . . . . .	16
3.7.1	Sur PC . . . . .	16
3.7.2	Sur Arduino . . . . .	16
<b>4</b>	<b>CMake et la compilation</b>	<b>18</b>
4.1	CMake . . . . .	18
4.1.1	Qu'est-ce que CMake ? . . . . .	18
4.1.2	Comment l'utiliser avec Arduino ? . . . . .	19
4.2	Notre CMakeLists.txt . . . . .	20
4.3	Comment compiler ? . . . . .	24
4.3.1	Compiler pour PC . . . . .	24
4.3.2	Compiler pour Arduino : . . . . .	24
<b>5</b>	<b>Résultats</b>	<b>25</b>
5.0.1	Résultat sous PC . . . . .	25
5.0.2	Résultat sous Arduino . . . . .	28



# Chapitre 1

## Introduction

La caméra EVI-D70P est une caméra de surveillance à distance développée par Sony, pouvant tourner de manière horizontale et verticale simultanément. Notre objectif ici est de contrôler ces déplacements depuis un ordinateur ou depuis un montage spécifique sur une carte Arduino Mega.

Pour cela, nous avons développé un logiciel en C++ dont les détails sont disponibles dans la suite de ce document.



## Chapitre 2

# La caméra Sony EVI D-70P

La caméra que nous allons utiliser dans notre projet est la EVI-D70P de la marque Sony. C'est une caméra couleur qui la spécificité d'être de type Pan/Tilt/Zoom :

-Pan = déplacement panoramique : la caméra peut tourner avec un angle de +/- 170° à partir de son axe d'origine (présence de butées à +170° et à -170° qui ne permet pas de faire une rotation complète). La vitesse maximale d'utilisation est de 100°/s.

-Tilt = inclinaison verticale : la caméra peut tourner avec un angle de +90°/-30° à partir de son axe d'origine (de la même manière, la présence de butées à +90° et à -30° ne permet pas de faire une rotation complète à 360°). La vitesse maximale d'utilisation est de 90°/s.

-Zoom = Capacité de zoom : x18, Angle de vue horizontal de 48 degrés (grand angle) à 2,7 degrés (téléobjectif), possède 440 000 pixels (752 x 582).



Une caméra est de type Pan/Tilt/Zoom, ce qui lui permet de suivre automatiquement le mouvement. Dès que l'objet en mouvement quitte le champ de vision de la caméra celle-ci revient à sa position préprogrammée. De plus, on a la possibilité de la fixer au plafond : elle peut donc être très adaptée pour faire de la vidéo-surveillance.

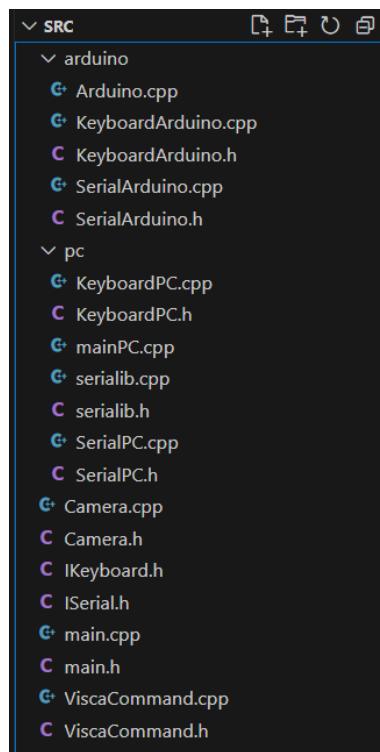
La caméra n'est actuellement plus disponible sur le marché mais est très utile dans des applications pratiques pour apprendre à manier le c++. C'est pourquoi nous l'utilisons dans notre projet. Ici, nous n'allons pas nous préoccuper du fonctionnement de la caméra en elle-même : notre but est d'envoyer grâce à notre code des ordres afin que l'on puisse contrôler les mouvements (Pan et Tilt).

En ce qui concerne l'interface, notre caméra possède 2 ports série : le port RS-232C qui permet à la caméra d'être contrôlée à distance (jusqu'à 15m). Le second port série (RS-422) quant à lui permet d'être contrôlé à une distance maximum de 1200m. Les 2 ports prennent en charge le protocole de communication VISCA : c'est ce protocole qui va nous permettre de contrôler avec la caméra depuis un ordinateur.

# Chapitre 3

## Le code

Tout d'abord, nous avons organisé notre espace de travail avec le code contenu dans un dossier "src" avec deux sous dossier : un pour les fichiers contenant du code exclusif à l'Arduino et un autre pour les fichiers concernant l'Arduino. Les fichiers en commun aux deux plate-formes ne sont dans aucun sous dossier.



### **3.1 Structure du Code**

Le code est organisé en plusieurs fichiers et classes, chacun ayant une spécificité. Voici un aperçu de la structure du code :

#### **3.1.1 Fichiers Principaux**

- `main.h` : On définit les fonctions communes à `Arduino.ino` et `mainPC.cpp` qui vont être les fonctions principales du programme.
- `main.cpp` : Le fichier central de notre programme.
- `Arduino.ino` : Le point d'entrée du programme Arduino.
- `mainPC.cpp` : Le point d'entrée du programme pour le PC.

#### **3.1.2 Gestion du Clavier**

- `IKeyboard.cpp` : Interface pour la gestion du clavier.
- `KeyboarPC.h` : Classe pour la gestion du clavier sur PC.
- `KeyboardArduino.h` : Classe pour la gestion du clavier sur Arduino.
- `KeyboardPC.cpp` : Implémentation de la gestion du clavier sur PC.
- `KeyboardArduino.cpp` : Implémentation de la gestion du clavier sur Arduino.

#### **3.1.3 Contrôle de la Caméra**

- `Camera.h` : Classe pour le contrôle de la caméra en fonction des touches pressées.
- `Camera.cpp` : Implémentation des fonctions de la classe `Camera.h`.

#### **3.1.4 Composition des Commandes**

- `ViscaCommand.cpp` : Fichier responsable de la composition des commandes à envoyer à la caméra.

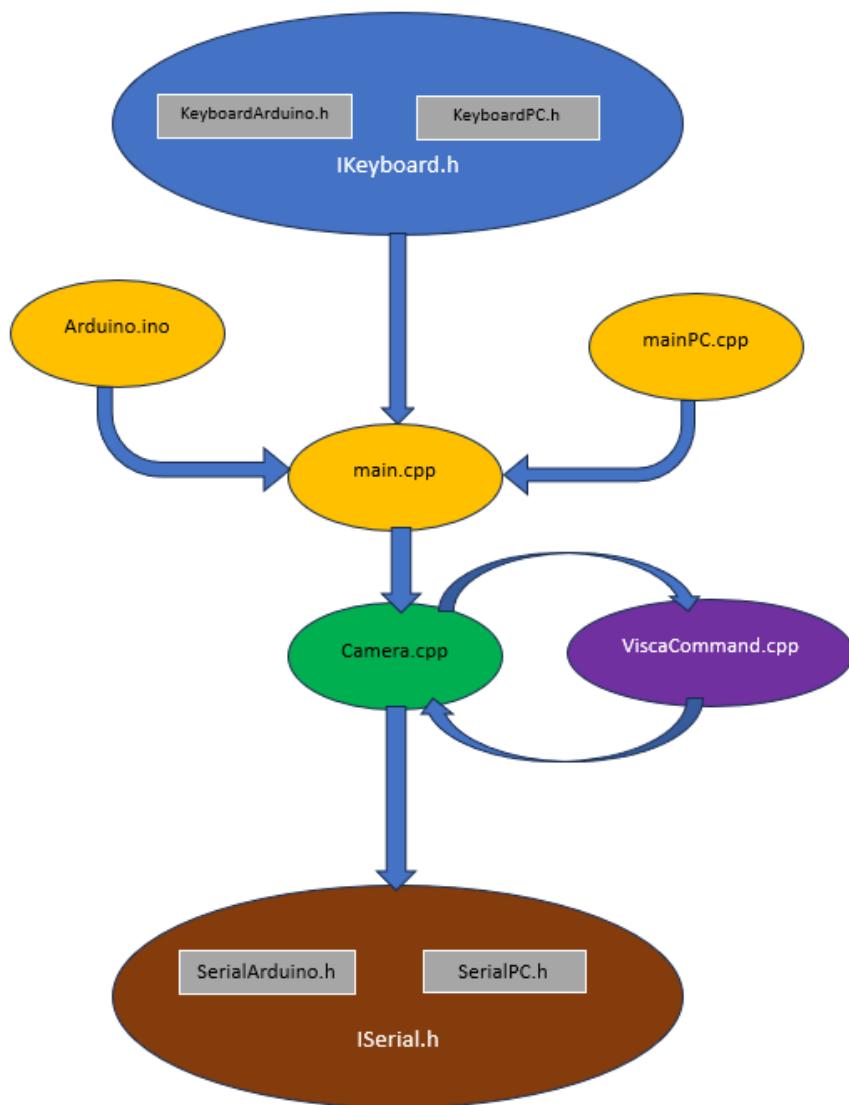
#### **3.1.5 Communication Série**

- `ISerial.h` : Interface pour la communication série.
- `SerialPC.h` : Classe pour la communication série sur PC.
- `SerialArduino.h` : Classe pour la communication série sur Arduino.
- `SerialPC.cpp` : Implémentation de l'envoi via la liaison serie sur PC.
- `SerialArduino.cpp` : Implémentation de l'envoi via la liaison serie sur Arduino.

- `serialib.h` et `serialib.cpp` : Librairie permettant l'envoi de données via la liaison série sur PC.

### 3.2 Fonctionnement des différents modules

Maintenant, nous allons voir comment fonctionnent les différents modules. Le schéma ci-dessous résume les liaisons des différents éléments du programme :



### 3.2.1 Cas du PC

Si on est sur PC, la fonction principale, le *int main()* se situe dans *mainPC.cpp*. Ce fichier contient 2 fonctions dont une qui initialise les composants et l'autre dans une boucle infinie qui fait tourner le programme. Les fonctions sont implémentées dans *main.cpp*. C'est dans ce fichier que l'on choisit le numéro de la caméra et le numéro de port COM utilisé par la caméra sur le PC. Ensuite, une fois les composants initialisés, le programme nous redirige dans le fichier *camera.cpp* qui va demander la touche pressée par l'utilisateur et choisir la commande à exécuter en fonction de la touche pressée. Il demande ensuite au fichier *ViscaCommand.cpp* de créer une commande en fonction du numéro de la caméra et du mouvement à réaliser. Ensuite nous sommes redirigés dans *serialPC.cpp* où l'on va ouvrir le port de communication série dans lequel on va envoyer la commande. Ce procédé est réalisé en boucle jusqu'à ce que l'utilisateur ferme le programme. Par défaut, on envoie la commande **STOP** pour que si aucune touche n'est pressée, la caméra ne bouge pas.

### 3.2.2 Cas de l'Arduino

La structure du code est la même que sur PC mais les fonctions pour détecter la touche pressée et l'envoi de la donnée sont différentes. La fonction principale se situe dans le fichier *Arduino.ino* où on a un *void setup*, qui appelle la fonction permettant d'initialiser les composants, et le *void loop* contenant la fonction à exécuter en boucle pour faire tourner le programme. Le choix des pins où est branché le clavier se situent lors de la déclaration de l'objet dans *main.cpp*.

Penchons-nous maintenant sur les détails plus techniques du fonctionnement de notre programme :

## 3.3 Lire la touche pressée

Pour commencer, nous avons créé une interface *IKeyboard.h* qui contient les fonctions de *KeyboardPC.h* et *KeyboardArduino.cpp*. Cette interface nous permet d'utiliser les fonctions présentes dans ces 2 classes sans que l'on ait besoin de préciser si on est sur arduino ou sur PC lors de l'appel de la fonction dans le fichier *Camera.cpp*. Cela permet d'éviter d'avoir deux fichiers *Camera.cpp* identiques mais bien en traitant séparément le cas de l'arduino et le cas du PC.

### 3.3.1 Sur PC

Pour récupérer la touche pressée sur PC, on utilise le code suivant :

```
1 #include "KeyboardPC.h"
2
3 using namespace std;
4
5
6 char KeyboardPC::readKey() {
7
8
9     char key = '0'; // Par défaut, aucune touche n'est pressée
10
11    if (GetAsyncKeyState('Z') & 0x8000) {
12        std::cout << "Haut" << std::endl;
13        return key = 'Z';
14    }
15    else if (GetAsyncKeyState('Q') & 0x8000) {
16        std::cout << "Gauche" << std::endl;
17        return key = 'Q';
18    }
19    else if (GetAsyncKeyState('S') & 0x8000) {
20        std::cout << "Bas" << std::endl;
21        return key = 'S';
22    }
23    else if (GetAsyncKeyState('D') & 0x8000) {
24        std::cout << "Droite" << std::endl;
25        return key = 'D';
26    }
27    else { return key = '0'; }
28
29
30 }
31 void KeyboardPC::initKeyboard(){}
32
33
```

On utilise les fonctions de la librairie *Windows.h*. La fonction `GetAsyncKeyState` renvoie un entier de 16 bits qui contient des informations sur l'état de la touche spécifiée. Le bit le plus significatif (le bit de signe) de cet entier est utilisé pour indiquer si la touche est actuellement enfoncée ou non. Si le bit de signe est positionné (c'est-à-dire égal à 1), cela signifie que la touche est enfoncée ; sinon, elle est relâchée.

L'opérateur `&` (ET bit à bit) est utilisé pour effectuer des opérations de masquage, où chaque bit d'une valeur binaire est combiné avec le bit correspondant d'une autre valeur binaire. Dans le code, `GetAsyncKeyState('Z') & 0x8000` est une opération de masquage pour vérifier si le bit de signe du résultat de `GetAsyncKeyState('Z')` est positionné.

Ainsi, la condition `GetAsyncKeyState('Z') & 0x8000` vérifie si le bit de signe du code de la touche 'Z' est positionné, ce qui signifie que la touche 'Z' est enfoncée. Si le bit de signe est égal à 1, la condition est vraie, indiquant que la touche 'Z' est pressée. Si le bit de signe est égal à 0, la condition est fausse, ce qui signifie que la touche 'Z' n'est pas pressée. Ensuite, on affiche un message dans le terminal pour vérifier si le programme a bien lu la touche pressée et on retourne le caractère correspondant à la touche pressée.

### 3.3.2 Sur Arduino

Sur Arduino, pour récupérer la touche pressée, on utilise ce code :

```
char key = 'o';
if (digitalRead(Pin1) == LOW) {
    Serial.print("BAS");
    delay(1000);
    return key='s'; // Vous pouvez retourner une valeur spécifique pour le bouton 1
}

else if (digitalRead(Pin2) == LOW) {
    delay(1000);
    Serial.print("GAUCHE");
    return key='q'; // Vous pouvez retourner une valeur spécifique pour le bouton 2
}

else if (digitalRead(Pin3) == LOW) {
    Serial.print("DROIT");
    delay(1000);
    return key='d'; // Vous pouvez retourner une valeur spécifique pour le bouton 3
}

else if (digitalRead(Pin4) == LOW) {
    Serial.print("HAUT");
    delay(1000);
    return key='z'; // Vous pouvez retourner une valeur spécifique pour le bouton 4
}

else { return key='o'; }
```

Il est important de noter que les valeurs de Pin1, Pin2, Pin3 et Pin4 ont été initialisées à la déclaration de l'objet de la classe *KerboardArduino* dans le fichier *main.cpp*. Le clavier Arduino est un ensemble de 4 boutons poussoirs laissant circuler un courant en continu. Lorsque l'on appuie sur un bouton, le courant arrête de circuler dans une des broches. Notre programme sert à déterminer quelle broche ne reçoit plus de courant. Ainsi, on peut déterminer quel bouton est pressé et on renvoie un caractère correspondant. Renvoyer un caractère Z,Q,S,D nous permet d'utiliser le même code que pour le PC pour déterminer l'action à réaliser.

### 3.4 Le fichier central

Voici notre fichier central : le *main.cpp*. C'est dans ce fichier que l'on va rentrer les différents paramètres pour notre programme :

```
 1  #include "main.h"
 2
 3
 4  #ifdef ARDUINO_BUILD
 5  KeyboardArduino Keyboard(46,48,50,52); // Broche : Bas, Gauche, Droite, Haut
 6
 7  SerialArduino SerialPort;
 8
 9  #elif PC_BUILD
10
11  KeyboardPC Keyboard;
12  SerialPc SerialPort("\\\\.\\COM10"); //Veuillez entrer le port de cette manière \\\\,.\\COMx
13
14  #endif
15  Camera Cam(1);
16
17
18  void InitComposant() {
19      SerialPort.initSerialCamera();
20      Keyboard.initKeyboard();
21  }
22
23  void oneIteration() {
24      Cam.moove(Keyboard, SerialPort);
25  }
26
27
```

Dans ce fichier, on peut voir 2 fonctions. *InitComposant*, est la fonction appelé dans le *void setup* coté Arduino et hors de la boucle infini dans le fichier *mainPC.cpp*. Il sert à initialiser le clavier et la caméra. Si on est sur Arduino, on rentre la valeur des différents Pin des boutons et si on est sur PC, on rentre le numéro du port COM. Lors de la déclaration de l'objet Camera, on entre le numéro de la caméra correspondante (un chiffre allant de 1 à 7). Nous avons 2 commandes de préprocesseur pour savoir quelle partie du code exécuter en fonction de la plate-forme utilisée. On détermine la plate-forme au moment de la compilation.

La fonction *moove* de la classe *Camera*, est une fonction qui va déterminer l'action à réaliser en fonction de la touche pressée.

### 3.5 La classe camera.h

Nous avons besoin de l'objet **Keyboard** pour récupérer la touche pressée et de l'objet **SerialPort** pour envoyer la commande. Le problème étant que deux cas existent : Soit on est sur PC, soit on est sur Arduino, ce qui signifie que soit l'objet est de type KeyboardPC, soit il est de type KeyboardArduino. Donc pour transmettre l'objet dans la fonction **moove**, on utilise un passage par référence de type **IKeyboard** et **ISerial**. Ainsi, nous n'avons pas besoin d'écrire 2 codes identiques pour la partie PC et la partie Arduino. Voici ce qu'il se passe dans la fonction **moove** :

```
void Camera::moove(IKeyboard& keyboard, ISerial& serial)
{
    char keyPressed = 'O';
    keyPressed = keyboard.readKey();
    switch (keyPressed) {
        case 'Z': serial.send_to_camera(ViscaCommand(id,'z')); break;
        case 'Q': serial.send_to_camera(ViscaCommand(id,'Q')); break;
        case 'S': serial.send_to_camera(ViscaCommand(id,'S')); break;
        case 'D': serial.send_to_camera(ViscaCommand(id,'D')); break;
        case 'O' : serial.send_to_camera(ViscaCommand(id,'O')); break;
    }
}
```

Tout d'abord, on initialise **keyPressed** à O pour envoyer la commande **STOP** en permanence si on appuie pas sur un bouton. Cela évite certains bugs, en particulier sur Arduino. Ensuite, on va lire la touche pressée. Selon celle renvoyée via le code vu précédemment, on va pouvoir envoyer une commande à la caméra. Pour envoyer une commande, on a une fonction **send\_to\_camera** implémentée dans l'interface **ISerial**. Nous allons voir comment est créée notre commande VISCA dans le paragraphe suivant. Il est important de savoir que la variable **id** en paramètre de la fonction **ViscaCommand** contient le numéro de la caméra entré dans le fichier *main.cpp* lors de la déclaration de l'objet **Camera**.

### 3.6 Construction de la commande Visca

Dans le manuel du constructeur de la caméra Sony D-70P, Nous pouvons trouver ceci :

Pan-tiltDrive	Up	8x 01 06 01 VV WW 03 01 FF	VV: Pan speed 01 to 18 WW: Tilt Speed 01 to 17
	Down	8x 01 06 01 VV WW 03 02 FF	YYYY: Pan Position F725 to 08DB (center 0000)
	Left	8x 01 06 01 VV WW 01 03 FF	ZZZZ: Tilt Position FE70 to 04B0 (Image Flip: OFF) (center 0000)
	Right	8x 01 06 01 VV WW 02 03 FF	Tilt Position FB50 to 0190 (Image Flip: ON) (center 0000)
	UpLeft	8x 01 06 01 VV WW 01 01 FF	See page 51.
	UpRight	8x 01 06 01 VV WW 02 01 FF	
	DownLeft	8x 01 06 01 VV WW 01 02 FF	
	DownRight	8x 01 06 01 VV WW 02 02 FF	
	Stop	8x 01 06 01 VV WW 03 03 FF	
	AbsolutePosition	8x 01 06 02 VV WW 0Y 0Y 0Y 0Y 0Z 0Z 0Z 0Z FF	
	RelativePosition	8x 01 06 03 VV WW 0Y 0Y 0Y 0Y 0Z 0Z 0Z 0Z FF	
	Home	8x 01 06 04 FF	
	Reset	8x 01 06 05 FF	

Pour piloter la caméra, nous avons accès à plusieurs commandes. Nous avons d'abord essayé avec les positions relatives, mais on obtenait pas un mouvement fluide. On a donc décidé d'utiliser les commandes Up, Down, Left et Right. Ces commandes effectuent un mouvement continu dans la direction indiquée jusqu'à ce qu'on envoie **STOP**. C'est pourquoi dans notre code on envoie **STOP** tant que l'on n'appuie pas sur un bouton. C'est pour garder la caméra à l'arrêt et pour qu'elle s'arrête si on relâche le bouton. Les valeurs indiquées sont en hexadécimal et la valeur x représente le numéro de la caméra contenu dans notre variable id. Le pan speed représente la vitesse de rotation de la caméra de manière horizontal et le tilt speed est la vitesse de rotation de manière vertical. Pour composer la commande dans notre code, voici comment nous avons choisi de procéder :

```

C: ViscaCommand.cpp > ...
1
2 #include "ViscaCommand.h"
3
4 #define END_MSG 0xFF
5 #define MOVE_CMD 0x06
6 #define HEADER 0x80
7 #define CMD_VAL 0x01
8 #define INQ_VAL 0x09
9 #define PAN_SPEED 0x10
10 #define TILT_SPEED 0x10
11
12 const char* ViscaCommand(int id, char Selec) {
13
14     static char command[9]; // Tableau pour stocker la commande
15
16     // Composez la commande en fonction de 'id' et 'Selec'
17     command[0] = HEADER + id;
18     command[1] = CMD_VAL;
19     command[2] = MOVE_CMD;
20     command[3] = CMD_VAL;
21     command[4] = PAN_SPEED;
22     command[5] = TILT_SPEED;
23
24     switch (Selec) {
25         case 'Z':
26             command[6] = 0x03;
27             command[7] = 0x02;
28             break;
29         case 'S':
30             command[6] = 0x03;
31             command[7] = 0x01;
32             break;
33         case 'Q':
34             command[6] = 0x01;
35             command[7] = 0x03;
36             break;
37         case 'D':
38             command[6] = 0x02;
39             command[7] = 0x03;
40             break;
41         case '0' :
42             command[6] = 0x03;
43             command[7] = 0x03;
44     }
45
46     command[8] = END_MSG;
47
48
49     return command;
50
51 }
52

```

On va composer les commandes dans un tableau de taille 9 car on a 9 caractères à intégrer et on note les valeurs sous forme hexadécimal. Les valeurs sont différentes aux cases 6 et 7 de notre commande en fonction du mouvement effectué. On retourne ensuite notre tableau qui contient la commande et c'est ce tableau qui va ensuite être envoyé dans la caméra.

## 3.7 Envoi de la commande

On se dirige maintenant à l'envoie de la commande.

### 3.7.1 Sur PC

```
#define BAUD_RATE 9600

SerialPc::SerialPc(const char *COM):com(COM){}

void SerialPc::send_to_camera(const char *command)
{
    serialib serial;

    serial.openDevice(com, BAUD_RATE); //Ouvre la connection serie

    if (serial.isDeviceOpen()) { // Permet de verifier si la connection serie a bien ete ouverte
        //writeString renvoie -1 si operation echouee et 1 si operation reussi
        int envoie = serial.writeString(command);
        if (envoie == -1) {
            std::cerr << "Erreur lors de l'envoi de la commande." << std::endl;
        }
        else {
            std::cout << "Commande envoyee avec succes." << std::endl;
        }
        serial.closeDevice();
    }
    else {
        std::cerr << "Erreur lors de l'ouverture de la communication serie." << std::endl;
    }
}
```

Le code utilise la bibliothèque ‘serialib’ pour gérer la communication série. Il commence par définir la vitesse de communication (baud rate) à 9600 bauds. Ensuite, la classe *SerialPC* est utilisée pour initialiser la communication série avec le port spécifié (‘COM’) dans le constructeur que nous choisissons dans le fichier *main.cpp*.

La fonction *send\_to\_camera* permet d’envoyer une commande Visca à l’appareil. Elle ouvre la connexion série, vérifie si la liaison a bien été ouverte et envoie la commande en utilisant ‘writeString’ de ‘serialib’. La fonction *writeString* renvoie 1 sur le message est partie et -1 si l’opération a échoué. En cas de succès, un message est affiché, sinon, une erreur est signalée.

### 3.7.2 Sur Arduino

Sur Arduino, c’est beaucoup plus simple mais cela nécessite de bien connaître la carte Arduino Mega. Sur ce type de carte, l’utilisation de ‘Serial1’ pour la communication série est justifiée en raison des caractéristiques de celle-ci :

1. **Plusieurs Ports Série Matériels** : La carte Arduino Mega est équipée de plusieurs ports série matériels, tels que ‘Serial’, ‘Serial1’, ‘Serial2’, et ‘Serial3’. Chacun de ces ports possède ses propres broches dédiées pour la transmission (TX) et la réception (RX), permettant de communiquer avec plusieurs périphériques série simultanément.
2. **Usage de ‘Serial’ pour la Communication avec l’Ordinateur Hôte** : Le port ‘Serial’ (également connu sous le nom de ‘Serial0’) est souvent utilisé

pour la communication avec l'ordinateur hôte lors de la programmation et du débogage de l'Arduino via le port USB.

3. ‘Serial1‘ pour un Port Série Supplémentaire : Pour communiquer avec des périphériques série externes, tels que des capteurs, des afficheurs, ou une caméra, vous pouvez utiliser ‘Serial1‘ (ou d’autres ports comme ‘Serial2‘, ‘Serial3‘, si disponibles). Chaque port série est configuré pour utiliser des broches spécifiques (RX1/TX1, RX2/TX2, RX3/TX3) pour la communication.

```
void SerialArduino::initSerialCamera()
{
    Serial1.begin(9600);
}

void SerialArduino::send_to_camera(const char* command)
{
    |   Serial1.write(command); //Utiliser Serial1 pour Arduino Mega
}
```

Nous avons utilisé Serial1 pour communiquer avec la caméra. Pour cela nous avons initialisé la vitesse de transmission de 9600 baud rate dans le void setup() et pour envoyer une commande à la caméra, nous avons juste à utiliser la commande Serial1.write().

## Chapitre 4

# CMake et la compilation

### 4.1 CMake

#### 4.1.1 Qu'est-ce que CMake ?

Cette étape nous a fait rencontrer des difficultés et a nécessité une grande concentration pour sa réalisation, car nous devions faire en sorte que notre programme se compile soit pour PC, soit pour Arduino. Nous avons choisis d'utiliser Cmake, un système de génération de code open source et multiplateforme conçu pour faciliter la gestion et la construction de projets logiciels. Il a pour objectif de simplifier la configuration du processus de compilation, qu'il s'agisse de projets C++, C, Python ou d'autres langages de programmation. CMake permet aux développeurs de spécifier comment leur logiciel doit être construit de manière indépendante de l'IDE (environnement de développement intégré) ou du compilateur utilisé.

Pour cela, CMake a besoin d'un fichier *CMakeLists.txt* pour configurer et générer la construction d'un projet logiciel. Le *CMakeLists.txt* sert de script de configuration qui spécifie comment le projet doit être construit, en définissant les sources, les cibles de construction, les dépendances, les variables de compilation, les options de configuration, et d'autres paramètres. En l'absence de *CMakeLists.txt*, CMake ne saurait pas comment organiser le projet, générer les fichiers de construction (comme les Makefiles), ni comment effectuer la compilation, rendant ainsi le processus de construction du projet difficile à automatiser et à personnaliser. Ce fichier *CMakeLists.txt* doit être placé à la racine de notre projet car c'est le premier fichier que CMake recherche lorsqu'il démarre la génération de projet, donc le placer de telle sorte assure qu'il est rapidement repérable. De plus, la racine du projet est un emplacement logique pour stocker les informations de configuration du projet.

#### 4.1.2 Comment l'utiliser avec Arduino ?

Pour compiler sous Arduino en utilisant CMake, nous avons effectué quelques étapes supplémentaires dans le cas de la partie PC. Pour commencer, nous nous sommes rendus sur le projet GitHub queezythegreat/arduino-cmake, et nous avons suivi les instructions inscrites dans le fichier ReadMe.md de ce projet. Ensuite, il faut installer Arduino-SDK qui correspond aux fichiers qui composent l'Arduino IDE. Ce projet GitHub fournit des exemples pour utiliser arduino avec CMake mais aussi un dossier CMake qui fournit la toolchain Arduino. La toolchain Arduino joue un rôle essentiel dans la compilation de projets Arduino en utilisant CMake. Une toolchain (chaîne d'outils) est un ensemble d'outils, de scripts et de configurations permettant de compiler du code source pour une plateforme matérielle spécifique, telle qu'une carte Arduino. La toolchain Arduino incluse dans ce projet facilite la génération de fichiers de construction (comme des Makefiles) adaptés à Arduino en se basant sur les spécifications de la carte cible. Voici les étapes pour compiler la partie arduino avec CMake en utilisant la toolchain de ce projet GitHub :

1. Mettre le fichier dossier CMake à la racine du projet : Pour commencer, nous devons télécharger le projet queezythegreat/arduino-cmake, et copier le dossier "cmake" de ce projet à la racine de notre projet. C'est ce fichier qui contient la toolchain pour utiliser Arduino sous CMake.
2. Créer un CMakeLists.txt pour votre Projet Arduino : Dans le répertoire racine de votre projet Arduino, on doit créer un fichier CMakeLists.txt. Ce fichier servira à configurer notre projet pour la compilation avec CMake. Nous devons spécifier les sources, les en-têtes, la carte Arduino cible, le port série, et d'autres paramètres dans ce fichier.
3. Inclure la Toolchain Arduino : Dans notre CMakeLists.txt, nous devons inclure la toolchain Arduino du projet queezythegreat/arduino-cmake en utilisant la commande set(CMAKE\_TOOLCHAIN\_FILE 'suivi du chemin vers le fichier ArduinoToolchain.cmake'). Par exemple :

```
set(CMAKE_TOOLCHAIN_FILE ${CMAKE_SOURCE_DIR}/cmake/ArduinoToolchain.cmake)
```

Cela signifie que le fichier ArduinoToolchain.cmake, se situe dans le dossier cmake lui même situé dans le même répertoire que le CMakeLists.txt

4. Configurer les Paramètres Arduino : Dans notre CMakeLists.txt, nous devons configurer les paramètres spécifiques à Arduino.

De plus, indiquer le port série sur lequel la carte Arduino est connectée :

```
set(PORT /dev/ttyUSB0)
```

5. Spécifier les Sources et En-têtes : On définit les fichiers sources et en-têtes de notre projet Arduino en utilisant les commandes `set`. Par exemple :

```
set(SOURCES
    src/main.cpp
    src/mylibrary.cpp
)

set(HEADERS
    include/mylibrary.h
)
```

6. Appeler `generate_arduino_firmware` : Pour générer le firmware Arduino, on utilise la commande `generate_arduino_firmware` en spécifiant les sources, les en-têtes, la carte cible, le port série, et d'autres paramètres nécessaires. Par exemple :

```
generate_arduino_firmware(MyArduinoProject
    SRCS ${SOURCES}
    HDRS ${HEADERS}
    BOARD ${BOARD}
    PORT ${PORT}
)
```

Ainsi, nous pouvons compiler pour Arduino en utilisant CMake.

## 4.2 Notre CMakeLists.txt

Maintenant, qu'on connaît CMake, et comment compiler pour Arduino en utilisant CMake, on va pouvoir présenter notre `CMakeLists.txt` utilisé pour le projet de la caméra. Pour commencer, voici son contenu :

```
# Spécifier la version minimale de CMake requise
cmake_minimum_required(VERSION 3.10)

set(ARDUINO_SDK_PATH)
set(CMAKE_TOOLCHAIN_FILE)
set (CMAKE_CXX_STANDARD 11)

if(ARDUINO_BUILD)
    # Configuration pour la compilation Arduino
```

```

set(ARDUINO_SDK_PATH ${CMAKE_SOURCE_DIR}/arduino)

set(CMAKE_TOOLCHAIN_FILE ${CMAKE_SOURCE_DIR}/cmake/ArduinoToolchain.cmake)
add_definitions(-DARDUINO_BUILD)
endif()

# Nom du Projet
project(Camera)

set(COMMON_HEADERS
    src/	Camera.h
    src/IKeyboard.h
    src/ISerial.h
    src/main.h
    src/ViscaCommand.h
)

set(COMMON_SOURCES
    src/	Camera.cpp
    src/ViscaCommand.cpp
    src/main.cpp
)

# Vérifie les macros PC_BUILD et ARDUINO_BUILD
if(PC_BUILD)
    set(HEADERS
        ${COMMON_HEADERS}
        src/pc/KeyboardPC.h
        src/pc/serialib.h
        src/pc/SerialPC.h
    )

    set(SOURCES
        ${COMMON_SOURCES}
        src/pc/KeyboardPC.cpp
        src/pc/mainPC.cpp
        src/pc/serialib.cpp
        src/pc/SerialPC.cpp
    )

```

```

# Configuration pour la compilation PC
add_executable(Camera ${SOURCES} ${HEADERS})
add_definitions(-DPC_BUILD)

elseif(ARDUINO_BUILD)
    # Configuration pour la compilation Arduino

    set(SOURCES
        ${COMMON_SOURCES}
        src/arduino/Arduino.cpp
        src/arduino/KeyboardArduino.cpp
        src/arduino/SerialArduino.cpp)

    set(HEADERS
        ${COMMON_HEADERS}
        src/arduino/KeyboardArduino.h
        src/arduino/SerialArduino.h)

register_hardware_platform(${ARDUINO_SDK_PATH}/hardware/arduino/avr)

set(mega.build.mcu atmega2560)

generate_arduino_firmware(
    Camera
    SRCS ${SOURCES}
    HDRS ${HEADERS}
    BOARD mega
    PORT COM13
    SERIAL -b 9600 -l)
endif()

```

Tout d'abord, il est important d'indiquer la version minimal nécessaire de CMake pour compiler le projet au début de celui-ci. Ensuite, nous déclarons la variable ARDUINO\_SDK\_PATH qui contiendra l'adresse du fichier Arduino-SDK et on déclare également CMAKE\_TOOLCHAIN\_FILE qui contiendra l'adresse de la toolchain Arduino. L'adresse de ces fichiers n'est déterminée uniquement si l'on compile sous Arduino. Mais dans le programme, ces 2 lignes doivent impérativement être placées avant la ligne project(Camera). La ligne set(CMAKE\_CXX\_STANDARD 11) spécifie la version du langage C++ à utiliser lors de la compilation. Dans notre cas, elle indique que le code C++ doit être compilé en respectant le standard C++11.

Pour savoir qu'est-ce que CMake doit compiler, nous avons créé 2 macro : ARDUINO\_BUILD et PC\_BUILD. On appelle ces macros au moment de la compilation en fonction de si on veut compiler pour Arduino ou pour PC.

Ensuite, nous avons organisé notre code de la manière suivante :

-Déclaration des fichiers sources communs aux 2 projets

-Déclaration des fichiers en-tête communs aux 2 projets

-On vérifie si on compile sous PC ou sous Arduino

Si on compile pour PC :

-On déclare les fichiers sources utilisés par le PC qu'on enregistre dans la variable SOURCES

-On déclare les fichiers en-tête utilisés par le PC qu'on enregistre dans la variable HEADERS

-Et on crée une cible exécutable nommée "Camera" à partir des fichiers sources (SOURCES) et des fichiers d'en-tête (HEADERS) spécifiés.

-Pour finir add\_definitions(-DPC\_BUILD) ajoute une définition de préprocesseur (-DPC\_BUILD) à la compilation de votre projet. En d'autres termes, elle définit la macro de préprocesseur "PC\_BUILD" lors de la compilation. Cette macro peut être utilisée dans le code source pour activer ou désactiver des parties spécifiques du code en fonction de la cible de compilation. Si on compile sous Arduino : -On reproduit les 2 mêmes premières étapes que pour le PC, c'est à dire on définit les fichiers SOURCES et HEADERS utilisés par l'Arduino.

-La ligne register\_hwplatform(\$ARDUINO\_SDK\_PATH/hardware/arduino/avr) dans notre fichier CMakeLists.txt sert à enregistrer la plate-forme matérielle Arduino pour la compilation. Cela est spécifique à la compilation pour Arduino et fait partie de la configuration nécessaire pour utiliser CMake avec Arduino.

-set(mega.build.mcu atmega2560) : Cette ligne définit la cible microcontrôleur (MCU) à utiliser lors de la compilation pour Arduino. -Pour finir, generate\_arduino\_firmware(...) : Cette ligne génère le firmware Arduino pour notre projet en utilisant les paramètres spécifiés.

## 4.3 Comment compiler ?

### 4.3.1 Compiler pour PC

Pour compiler sur PC, nous nous plaçons dans le répertoire 'Build PC' pour garder notre espace de travail propre et on utilise la commande suivante :

```
cmake -G "Unix Makefiles" -D PC_BUILD=ON ..
```

1. cmake : C'est la commande CMake elle-même, utilisée pour configurer et générer les fichiers nécessaires à la compilation de votre projet.
2. -G "Unix Makefiles" : Cette option spécifie le générateur que l'on souhaite utiliser pour générer les fichiers de construction. Dans notre cas, on utilise le générateur "Unix Makefiles" car c'est celui indiqué dans le projet GitHub queezythegreat/arduino-cmake. Cela signifie que CMake générera des fichiers de construction adaptés au système de compilation Make. Cela permettra de compiler le projet en utilisant les règles de Make.
3. -D PC\_BUILD=ON : Cette option permet de définir une variable CMake au moment de la génération. Nous définissons ici la variable PC\_BUILD à ON, ce qui indique à CMake que on souhaite activer la configuration de compilation pour la plate-forme PC. En d'autres termes, cela conditionne l'exécution des parties du code CMake destinées à la compilation PC.
4. .. : Ce chemin indique à CMake où se trouve le répertoire source de votre projet. En l'occurrence, .. signifie que le répertoire parent du répertoire actuel doit être utilisé comme répertoire source.

Une fois que nous avons utilisé CMake pour générer les fichiers de construction pour notre projet, on utilise la commande make pour réellement compiler le projet en fonction des règles définies dans ces fichiers de construction.

### 4.3.2 Compiler pour Arduino :

Pour compiler sous Arduino, nous nous plaçons dans le répertoire 'Build Arduino' et on utilise la commande suivante :

```
cmake -G "Unix Makefiles" -D ARDUINO_BUILD=ON ..
```

L'explication de la commande est la même que pour compiler sous PC. On utilise ensuite la commande make et pour finir on utilise la commande 'make upload' pour téléverser le firmware généré sur une carte Arduino connectée à votre ordinateur.

# Chapitre 5

## Résultats

Cette partie est consacrée aux résultats obtenus après avoir branché la caméra à l'ordinateur et exécuté les différents codes.

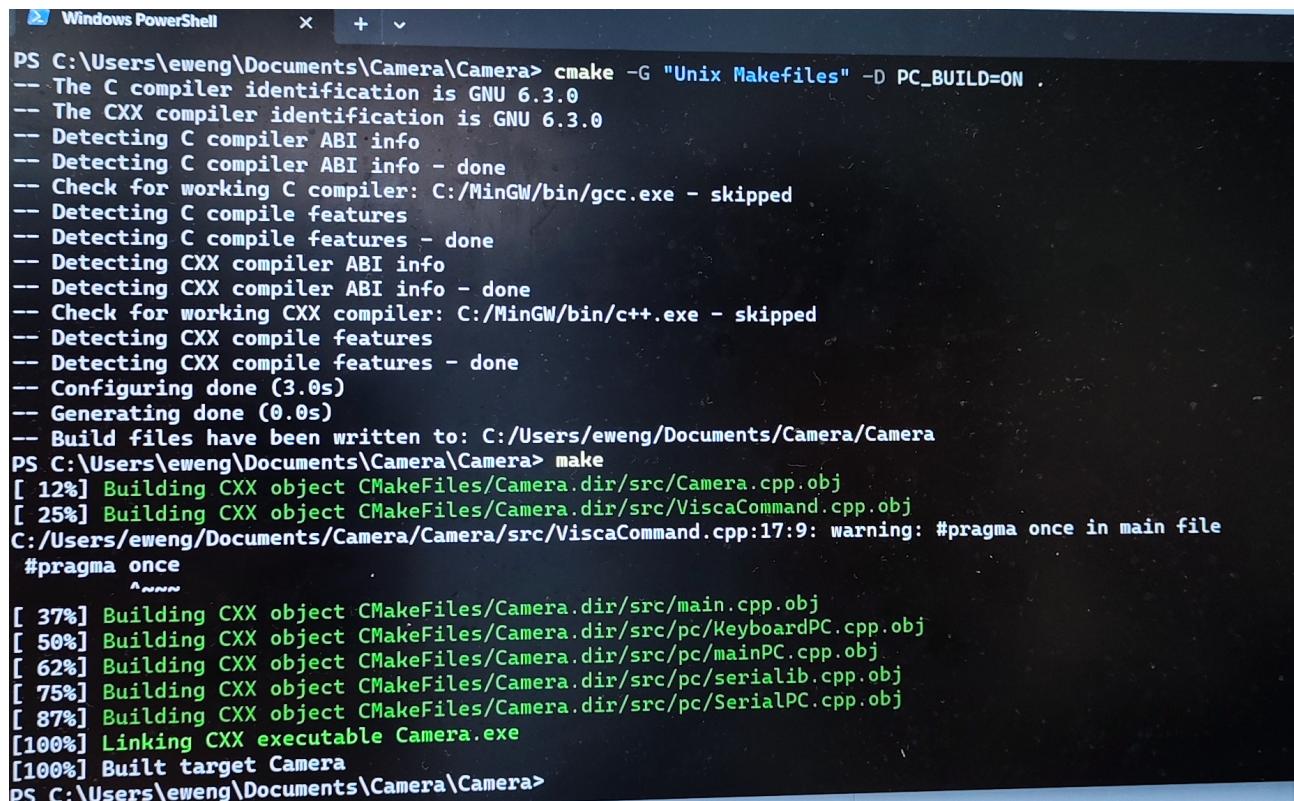
### 5.0.1 Résultat sous PC

Nous avons réussi dans un premier temps à contrôler la caméra sans utiliser le cmake. A cause d'un problème de câble survenu il y a quelques semaines, nous n'avons pas pu prendre plus de photos : les photos qui suivent vous montrent les branchements et les différentes positions prises par la caméra (bas,gauche,haut,droite). Nous vous assurons que la caméra a bien fonctionné au moment où nous avons testé notre code.





Puis dans un second temps, après avoir réussi à faire fonctionner le cmake, nous avons de nouveau réussi. Voici une capture d'écran qui montre que la compilation a bien fonctionné :



```
PS C:\Users\eweng\Documents\Camera> cmake -G "Unix Makefiles" -D PC_BUILD=ON .
-- The C compiler identification is GNU 6.3.0
-- The CXX compiler identification is GNU 6.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/MinGW/bin/gcc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/MinGW/bin/c++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (3.0s)
-- Generating done (0.0s)
-- Build files have been written to: C:/Users/eweng/Documents/Camera/Camera
PS C:\Users\eweng\Documents\Camera> make
[ 12%] Building CXX object CMakeFiles/Camera.dir/src/Camera.cpp.obj
[ 25%] Building CXX object CMakeFiles/Camera.dir/src/ViscaCommand.cpp.obj
C:/Users/eweng/Documents/Camera/Camera/src/ViscaCommand.cpp:17:9: warning: #pragma once in main file
  #pragma once
  ^~~~~
[ 37%] Building CXX object CMakeFiles/Camera.dir/src/main.cpp.obj
[ 50%] Building CXX object CMakeFiles/Camera.dir/src/pc/KeyboardPC.cpp.obj
[ 62%] Building CXX object CMakeFiles/Camera.dir/src/pc/mainPC.cpp.obj
[ 75%] Building CXX object CMakeFiles/Camera.dir/src/pc/serialib.cpp.obj
[ 87%] Building CXX object CMakeFiles/Camera.dir/src/pc/SerialPC.cpp.obj
[100%] Linking CXX executable Camera.exe
[100%] Built target Camera
PS C:\Users\eweng\Documents\Camera>
```

### 5.0.2 Résultat sous Arduino

De la même manière que pour les résultats sous PC, nous avons réussi dans un premier temps à contrôler la caméra sans utiliser le cmake. La photo qui suis vous montre les branchements et une des positions prise par la caméra. Des problèmes de branchements nous ont empêché de prendre plusieurs photos mais de nouveau nous vous assurons que la caméra à bien fonctionné au moment où nous avons testé notre code.



Puis dans un second temps, après avoir réussi à faire fonctionné le cmake, nous avons de nouveau réussi. Voici une capture d'écran qui montre que la compilation a bien fonctionné :

```
Administrator : Windows PowerShell
-- Check for working CXX compiler: C:/Users/eweng/Documents/Camera/Camera/arduino/hardware/tools/avr/bin/avr-g++.exe - s
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Generating Camera
-- Configuring done (1.7s)
-- Generating done (0.0s)
-- Build files have been written to: C:/Users/eweng/Documents/Camera/Camera
ps C:\Users\eweng\Documents\Camera\Camera> make
[ 3%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/cores/arduino/CDC.cpp.obj
[ 6%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/cores/arduino/HardwareSerial.cpp.obj
[ 9%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/HardwareSerial0.cpp.obj
[12%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/HardwareSerial1.cpp.obj
[15%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/HardwareSerial2.cpp.obj
[18%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/HardwareSerial3.cpp.obj
[21%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/IPAddress.cpp.obj
[25%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/PluggableUSB.cpp.obj
[28%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/Print.cpp.obj
[31%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/Stream.cpp.obj
[34%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/Tone.cpp.obj
[37%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/USBCore.cpp.obj
[40%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/WInterrupts.c.obj
[43%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/WMath.cpp.obj
[46%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/WString.cpp.obj
[50%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/abi.cpp.obj
[53%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/hook.c.obj
[56%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/main.cpp.obj
[59%] Building CXX object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/new.cpp.obj
[62%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/wiring.c.obj
[65%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/wiring_analog.c.obj
[68%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/wiring_digital.c.obj
[71%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/wiring_pulse.c.obj
[75%] Building C object CMakeFiles/mega_CORE.dir/arduino/hardware/arduino/avr/cores/arduino/wiring_shift.c.obj
[78%] Linking CXX static library libmega_CORE.a
[78%] Built target mega_CORE
[81%] Building CXX object CMakeFiles/Camera.dir/src/Camera.cpp.obj
[84%] Building CXX object CMakeFiles/Camera.dir/src/ViscaCommand.cpp.obj
[87%] Building CXX object CMakeFiles/Camera.dir/src/main.cpp.obj
[90%] Building CXX object CMakeFiles/Camera.dir/src/Arduino.cpp.obj
[93%] Building CXX object CMakeFiles/Camera.dir/src/arduino/KeyboardArduino.cpp.obj
[96%] Building CXX object CMakeFiles/Camera.dir/src/arduino/SerialArduino.cpp.obj
[100%] Linking CXX executable Camera.elf
Generating EEP image
Generating HEX image
Calculating image size
Firmware Size: [Program: 4110 bytes (1.6%)] [Data: 422 bytes (5.2%)] on atmega2560
EEPROM  Size: [Program: 0 bytes (0.0%)] [Data: 0 bytes (0.0%)] on atmega2560
[100%] Built target Camera
PS C:\Users\eweng\Documents\Camera\Camera>
```

## Chapitre 6

# Conclusion

Ce projet qui nous était donné consistait à contrôler la caméra Sony EVI D-70P en utilisant le langage C++. Nous avons décrit les caractéristiques et les commandes de la caméra, ainsi que le code que nous avons développé pour la contrôler. Puis, nous avons expliqué la structure du code, les différents modules et fonctions, et la communication série entre le PC et l'Arduino. Nous avons également montré comment nous avons utilisé CMake pour compiler le code sur différentes plateformes. Nous avons ainsi réussi à réaliser un programme qui permet de contrôler la caméra à distance à partir du clavier. Ce projet nous a permis de mettre en pratique nos connaissances en C++, en électronique et en communication série. Il nous a également fait découvrir la caméra Sony EVI D-70P et le protocole Visca.

Nous avons eu des problèmes durant le développement de notre code mais avec beaucoup de recherches et d'entraide, nous avons finalement réussi à mener notre projet à bien. Nous en profitons pour vous remercier de votre aide car nous étions par moment bloqués et vos conseils nous avez permis d'avancer.